ACM-ICPC Team Reference Document

Mordovian State University (Plotnikova, Martynov, Deniskin)

Contents

1 General

1.1 Python Template

```
import sys
import re
from math import ceil, log, sqrt, floor

__local_run__ = False
if __local_run__:
    sys.stdin = open('input.txt', 'r')
    sys.stdout = open('output.txt', 'w')

def main():
    a = int(input())
    b = int(input())
    print(a*b)

main()
```

1.2 C++ Template

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp> // gp_hash_table
        <int, int> == hash map
#include <ext/pb_ds/tree_policy.hpp>
using namespace std;
using namespace __gnu_pbds;
typedef long long ll;
typedef unsigned long long ull;
typedef long double ld;
typedef pair<int, int> pii;
typedef pair<ll, ll> pll;
typedef pair<double, double> pdd;
template <typename T> using min_heap = priority_queue<T,
         vector<T>, greater<T>>;
template <typename T> using max_heap = priority_queue<T,
         vector<T>, less<T>>;
template <typename T> using ordered_set = tree<T,
        null_type, less<T>, rb_tree_tag,
        tree_order_statistics_node_update>;
template <typename K, typename V> using hashmap =
        gp_hash_table<K, V>;

template<typename A, typename B> ostream& operator<<(
        ostream& out, pair<A, B> p) { out << "(" << p.first
        << ", " << p.second << ")"; return out;}
template<typename T> ostream& operator<<(ostream& out,
        vector<T> v) { out << "["; for(auto& x : v) out << x
        << ", "; out << "]";return out;}
template<typename T> ostream& operator<<(ostream& out,
        set<T> v) { out << "{"; for(auto& x : v) out << x <<
        ", "; out << "}"; return out; }
template<typename K, typename V> ostream& operator<<(
        ostream& out, map<K, V> m) { out << "{"; for(auto& e
         : m) out << e.first << " -> " << e.second << ", "; out
         << "}"; return out; }
template<typename K, typename V> ostream& operator<<(
        ostream& out, hashmap<K, V> m) { out << "{"; for(
        auto& e : m) out << e.first << " -> " << e.second <<
        ", "; out << "}"; return out; }

#define FAST_IO ios_base::sync_with_stdio(false); cin.tie(
        NULL)
#define TESTS(t) int NUMBER_OF_TESTS; cin >>
        NUMBER_OF_TESTS; for(int t = 1; t <=
        NUMBER_OF_TESTS; t++)
#define FOR(i, begin, end) for (int i = (begin) - ((begin) > (
        end)); i != (end) - ((begin) > (end)); i += 1 - 2 * ((begin)
         > (end)))
#define sgn(a) ((a) > eps ? 1 : ((a) < -eps ? -1 : 0))
#define precise(x) fixed << setprecision(x)
#define debug(x) cerr << "> " << #x << " = " << x <<
        endl;
#define pb push_back
#define rnd(a, b) (uniform_int_distribution<int>((a), (b))(rng
        ))
#ifndef LOCAL
    #define cerr if(0)cout
    #define endl "\n"
#endif
mt19937 rng(chrono::steady_clock::now().time_since_epoch().
        count());
clock_t ___clock___;
void startTime() {___clock___ = clock();}
void timeit(string msg) {cerr << "> " << msg << ": " <<
        precise(6) << ld(clock()-___clock___)/
        CLOCKS_PER_SEC << endl;}
const ld PI = asin(1) * 2;
const ld eps = 1e-14;
const int oo = 2e9;
const ll OO = 2e18;
const ll MOD = 1000000007;
const int MAXN = 1000000;

int main() {
    FAST_IO;
    startTime();

    timeit("Finished");
    return 0;
}
```

1.3 C++ Visual Studio Includes

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <set>
#include <map>
#include <cmath>
#include <queue>
#include <iomanip>
#include <bitset>
#include <unordered_map>
#include <stack>
#include <memory.h>
#include <list>
#include <numeric>
#include <functional>
#include <complex>
#include <cassert>
#include <regex>
#include <random>
#include <iomanip>
#include <climits>
#pragma comment(linker, "/STACK:360777216")

using LL = long long;
using ll = long long;
using ld = long double;
#define all(x) (x).begin(),(x).end()
#define rall(x) (x).rbegin(),(x).rend()
#define pii pair<int,int>
#define pll pair<LL,LL>
#define vi vector<int>
#define vll vector<LL>
#define vvll vector<vector<LL>>
#define vpii vector<pii>
#define vpll vector<pll>
#define vvi vector<vector<int>>
#define forn(it,from,to) for(int (it)=from; (it)<to; (it)++)
const int INF = 2 * 1000 * 1000 * 1000;
LL MOD = 1e9 + 7;
LL LINF = (LL)4e18;
double EPS = 1e-7;

using namespace std;


int main() {
#ifdef _DEBUG
        freopen("input.txt", "r", stdin);
        freopen("output.txt", "w", stdout);
#else
        //freopen("input.txt", "r", stdin);
        //freopen("output.txt", "w", stdout);
#endif
        ios::sync_with_stdio(false);
        cin.tie(0); cout.tie(0);
        cout << fixed << setprecision(10);
        srand(time(nullptr));

        LL ___ = 1, n, m, k, r, u, v, m1, m2, x, y, l, a, b;


        cin >> ___;

        forn(_, 0, ___)
        {
                cin >> n;


        }


        return 0;
}
```

2 Data Structures

2.1 Treap

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

template<typename T>
struct Treap {
    struct Node {
        Node *l, *r;
        T x;
        int y, size;

        Node() {}

        Node(T _x) : x(_x), y(rng()), l(nullptr), r(nullptr), size
            (1) {}
    };

    typedef Node *NodePtr;
    NodePtr root;

    Treap() : root(nullptr) {}

    inline int sz(NodePtr a) const {
        return a ? a->size : 0;
    }

    inline void recalc(NodePtr a) {
        if (!a) return;
        a->size = sz(a->l) + sz(a->r) + 1;
    }


private:
    void merge(NodePtr a, NodePtr b, NodePtr &c) {
        if (!a) c = b;
        else if (!b) c = a;
        else {
            if (a->y > b->y) {
                merge(a->r, b, a->r);
                c = a;
            } else {
                merge(b->l, a, b->l);
                c = b;
            }
            recalc(c);
        }
    }

    void split(NodePtr c, T k, NodePtr &a, NodePtr &b) {
        if (!c) { a = b = nullptr; }
        else {
            if (c->x < k) {
                split(c->r, k, c->r, b);
                a = c;
            } else {
                split(c->l, k, a, c->l);
                b = c;
            }
        }
        recalc(c);
    }

    void insert(NodePtr &ptr, NodePtr val) {
        if (!ptr) ptr = val;
        else if (ptr->x != val->x) {
            if (val->y > ptr->y) {
                split(ptr, val->x, val->l, val->r);
                ptr = val;
            } else {
                if (val->x > ptr->x) insert(ptr->r, val);
                else insert(ptr->l, val);
            }
        }
        recalc(ptr);
    }

    void erase(NodePtr ptr, T k) {
        if (!ptr) return;
        if (ptr->x == k) {
            merge(ptr->l, ptr->r, ptr);
        } else if (ptr->x > k) erase(ptr->l, k);
        else erase(ptr->r, k);
        recalc(ptr);
    }

    int count(NodePtr ptr, T k) {
        if (!ptr) return 0;
        if (ptr->x == k) return 1;
        if (ptr->x > k) return count(ptr->l, k);
        else return count(ptr->r, k);
    }

    int order_of_key(NodePtr ptr, T k) {
        if (!ptr) return 0;
        if (ptr->x < k) return sz(ptr->l) + 1 + order_of_key(
            ptr->r, k);
        else return order_of_key(ptr->l, k);
    }

    T get_by_id(const NodePtr ptr, int id) const {
        if (sz(ptr->l) == id) return ptr->x;
        if (id < sz(ptr->l)) return get_by_id(ptr->l, id);
        else return get_by_id(ptr->r, id - sz(ptr->l) - 1);
    }

public:
    inline unsigned int size() {
        return sz(root);
    }

    inline void insert(T k) {
        insert(root, new Node(k));
    }

    inline void erase(T k) {
        erase(root, k);
    }

    inline int count(T k) {
        return count(root, k);
    }

    inline int order_of_key(T k) {
        return order_of_key(root, k);
    }

    inline T operator[](int pos) const {
        return get_by_id(root, pos);
    }
};
```

2.2 Disjoin Set Union

```
class DSU
{
private:
        vector<int> p;
public:
        DSU(int sz) { p.resize(sz); }

        void make_set(int v) {
                p[v] = v;
        }

        int get(int v) {
                return (v == p[v]) ? v : (p[v] = get(p[v]));
        }

        void unite(int a, int b) {
                a = get(a);
                b = get(b);
                if (rand() & 1)
                        swap(a, b);
                if (a != b)
                        p[a] = b;
        }
};
```

2.3 Fenwick Tree Range Update And Range Query

```
struct RangedFenwick {
    Fenwick F1, F2; // support range query and point update
    RangedFenwick(int _n) {
        F1 = Fenwick(_n+1);
        F2 = Fenwick(_n+1);
    }
    void add(int l, int r, ll v) { // arr[l..r] += v
```

```
        F1.add(l, v);
        F1.add(r+1, -v);
        F2.add(l, v*(l-1));
        F2.add(r+1, -v*r);
    }
    ll sum(int i) { // arr[1..i]
        return F1.sum(i)*i-F2.sum(i);
    }
    ll sum(int l, int r) { // arr[l..r]
        return sum(r)-sum(l-1);
    }
};
```

2.4 Persistent Segment Tree

```
template<class T>
class Vertex
{
public:
        Vertex* left, * right;
        T val;
        Vertex(T _val) { left = right = nullptr; val = _val; }
        Vertex(Vertex* _left, Vertex* _right, function<T(T, T)
            > BinF, T _val)
        {
                left = _left;
                right = _right;
                val = _val;
                if (left) val = BinF(val, left->val);
                if (right) val = BinF(val, right->val);
        }
};


template<class T, int sz, class ArrT>
class SegTree
{
private:
        T SideVal;
        function<T(T, T)> BinF;
        function<T(ArrT)> BuildLF;
public:
        SegTree(T _SideVal, function<T(T, T)> _BinF,
            function<T(ArrT)> _BuildLF) {
                SideVal = _SideVal;
                BinF = _BinF;
                BuildLF = _BuildLF;
        }

        Vertex<T>* build(vector<ArrT>& a, int tl, int tr) {
                if (tl == tr) return new Vertex<T>(BuildLF(a[tl
                    ]));
                else {
                        int tm = (tl + tr) / 2;
                        return new Vertex<T>(build(a, tl, tm),
                            build(a, tm + 1, tr), BinF, SideVal);
                }
        }

        Vertex<T>* update(Vertex<T> *t, int tl, int tr, int pos
            , ArrT val)
        {
                if (tl == tr) return new Vertex<T>(BuildLF(val)
                    );
                else
                {
                        int tm = (tl + tr) / 2;
                        if (pos <= tm) return new Vertex<T>(
                            update(t->left, tl, tm, pos, val), t->
                            right, BinF, SideVal);
                        else return new Vertex<T>(t->left,
                            update(t->right, tm + 1, tr, pos, val
                            ), BinF, SideVal);
                }
        }

        T get_val(Vertex<T>* t, int tl, int tr, int l, int r)
        {
                if (l > r) return SideVal;
                if (tl == l && tr == r) return t->val;
                int tm = (tl + tr) / 2;
                auto left = get_val(t->left, tl, tm, l, min(tm, r));
                auto right = get_val(t->right, tm + 1, tr, max(
                    tm + 1, l), r);
                return BinF(left, right);
        }
};

auto BinF = [](LL left, LL right) -> $SegTreeType$ {
};

auto BuildLeaf = [](LL val) -> $SegTreeType$ {

};

SegTree<LL, 200100, LL> st(0, BinF, BuildLeaf);
```

2.5 Fenwick Tree Point Update And Range Query

```
struct Fenwick {
    vector<ll> tree;
    int n;
    Fenwick(){}
    Fenwick(int _n) {
        n = _n;
        tree = vector<ll>(n+1, 0);
    }
    void add(int i, ll val) { // arr[i] += val
        for(; i <= n; i += i&(-i)) tree[i] += val;
    }
    ll get(int i) { // arr[i]
        return sum(i, i);
    }
    ll sum(int i) { // arr[1]+...+arr[i]
        ll ans = 0;
        for(; i > 0; i -= i&(-i)) ans += tree[i];
        return ans;
    }
    ll sum(int l, int r) {// arr[l]+...+arr[r]
        return sum(r) - sum(l-1);
    }
};
```

2.6 Fenwick Tree Range Update And Point Query

```
struct Fenwick {
    vector<ll> tree;
    vector<ll> arr;
    int n;
    Fenwick(vector<ll> _arr) {
        n = _arr.size();
        arr = _arr;
        tree = vector<ll>(n+2, 0);
    }
    void add(int i, ll val) { // arr[i] += val
        for(; i <= n; i += i&(-i)) tree[i] += val;
    }
    void add(int l, int r, ll val) {// arr[l..r] += val
        add(l, val);
        add(r+1, -val);
    }
    ll get(int i) { // arr[i]
        ll sum = arr[i-1]; // zero based
        for(; i > 0; i -= i&(-i)) sum += tree[i];
        return sum; // zero based
    }
};
```

2.7 Implicit Treap

```
template <typename T>
struct Node {
    Node* l, *r;
    ll prio, size, sum;
    T val;
    bool rev;
```

```
    Node() {}
    Node(T _val) : l(nullptr), r(nullptr), val(_val), size(1), sum(
        _val), rev(false) {
        prio = rand() ^ (rand() << 15);
    }
};
template <typename T>
struct ImplicitTreap {
    typedef Node<T>* NodePtr;
    int sz(NodePtr n) {
        return n ? n->size : 0;
    }
    ll getSum(NodePtr n) {
        return n ? n->sum : 0;
    }

    void push(NodePtr n) {
        if (n && n->rev) {
            n->rev = false;
            swap(n->l, n->r);
            if (n->l) n->l->rev ^= 1;
            if (n->r) n->r->rev ^= 1;
        }
    }

    void recalc(NodePtr n) {
        if (!n) return;
        n->size = sz(n->l) + 1 + sz(n->r);
        n->sum = getSum(n->l) + n->val + getSum(n->r);
    }

    void split(NodePtr tree, ll key, NodePtr& l, NodePtr& r) {
        push(tree);
        if (!tree) {
            l = r = nullptr;
        }
        else if (key <= sz(tree->l)) {
            split(tree->l, key, l, tree->l);
            r = tree;
        }
        else {
            split(tree->r, key-sz(tree->l)-1, tree->r, r);
            l = tree;
        }
        recalc(tree);
    }

    void merge(NodePtr& tree, NodePtr l, NodePtr r) {
        push(l); push(r);
        if (!l || !r) {
            tree = l ? l : r;
        }
        else if (l->prio > r->prio) {
            merge(l->r, l->r, r);
            tree = l;
        }
        else {
            merge(r->l, l, r->l);
            tree = r;
        }
        recalc(tree);
    }

    void insert(NodePtr& tree, T val, int pos) {
        if (!tree) {
            tree = new Node<T>(val);
            return;
        }
        NodePtr L, R;
        split(tree, pos, L, R);
        merge(L, L, new Node<T>(val));
        merge(tree, L, R);
        recalc(tree);
    }

    void reverse(NodePtr tree, int l, int r) {
        NodePtr t1, t2, t3;
        split(tree, l, t1, t2);
        split(t2, r - l + 1, t2, t3);
        if(t2) t2->rev = true;
        merge(t2, t1, t2);
        merge(tree, t2, t3);
    }

    void print(NodePtr t, bool newline = true) {
        push(t);
        if (!t) return;
        print(t->l, false);
        cout << t->val << " ";
        print(t->r, false);
        if (newline) cout << endl;
    }

    NodePtr fromArray(vector<T> v) {
        NodePtr t = nullptr;
        FOR(i, 0, (int)v.size()) {
            insert(t, v[i], i);
        }
        return t;
    }

    ll calcSum(NodePtr t, int l, int r) {
        NodePtr L, R;
        split(t, l, L, R);
        NodePtr good;
        split(R, r - l + 1, good, L);
        return getSum(good);
    }
};
/* Usage: ImplicitTreap<int> t;
Node<int> tree = t.fromArray(someVector); t.reverse(tree, l, r);
    ...
*/
```

2.8 Fenwick 2D

```
struct Fenwick2D {
    vector<vector<ll>> bit;
    int n, m;
    Fenwick2D(int _n, int _m) {
        n = _n; m = _m;
        bit = vector<vector<ll>>(n+1, vector<ll>(m+1, 0));
    }
    ll sum(int x, int y) {
        ll ret = 0;
        for (int i = x; i > 0; i -= i & (-i))
            for (int j = y; j > 0; j -= j & (-j))
                ret += bit[i][j];
        return ret;
    }
    ll sum(int x1, int y1, int x2, int y2) {
        return sum(x2, y2) - sum(x2, y1-1) - sum(x1-1, y2) +
            sum(x1-1, y1-1);
    }
    void add(int x, int y, ll delta) {
        for (int i = x; i <= n; i += i & (-i))
            for (int j = y; j <= m; j += j & (-j))
                bit[i][j] += delta;
    }
};
```

2.9 Segment Tree

```
template<typename T, class F = function<T(const T &, const
    T &)>>
struct SegmentTree {
    int n{};
    vector<T> st;
    F merge = [&](const T &i, const T &j) {
        return i ^ j;
    };
    T neutral = 0;

    SegmentTree() = default;

    explicit SegmentTree(const vector<T> &a) {
        n = (int) a.size();
        st.resize(2 * (int) a.size());
        for (int i = 0; i < n; i++) st[i + n] = a[i];
        for (int i = n - 1; i > 0; i--) st[i] = merge(st[i << 1], st[i
            << 1 | 1]);
    }

    T get(int l, int r) {
        T resl = neutral, resr = neutral;
        for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
            if (l & 1) resl = merge(resl, st[l++]);
            if (r & 1) resr = merge(st[--r], resr);
```

```
        }
        return merge(resl, resr);
    }

    void upd(int p, T val) {
        for (st[p += n] = val; p > 1; p >>= 1) {
            if (p & 1) {
                st[p >> 1] = merge(st[p ^ 1], st[p]);
            } else {
                st[p >> 1] = merge(st[p], st[p ^ 1]);
            }
        }
    }
};
```

2.10 Segment Tree With Lazy Propagation

```
template<typename T = int, typename TU = int>
struct SegmentTree {
    SegmentTree() = default;
    explicit SegmentTree(const vector<T> &a) {
        n = (int) a.size();
        st.resize(4 * n);
        upd_val.resize(4 * n);
        upd_fl.resize(4 * n);
        build(1, 0, n, a);
    }

    explicit SegmentTree(int __n) {
        n = __n;
        st.resize(4 * n);
        upd_val.resize(4 * n);
        upd_fl.resize(4 * n);
    }

    T get(int l, int r) {
        return get(1, 0, n, l, r);
    }

    T get(int p) {
        return get(p, p + 1);
    }

    void upd(int p, TU val) {
        upd(p, p + 1, val);
    }

    void upd(int l, int r, TU val) {
        upd(1, 0, n, l, r, val);
    }

private:
    void push(int tv, int tl, int tr) {
        if (upd_fl[tv] == 1 && tr - tl > 1) {
            int tm = (tl + tr) >> 1;
            st[tv * 2] = recalc_on_segment(st[tv * 2], upd_val[
                tv], tl, tm);
            st[tv * 2 + 1] = recalc_on_segment(st[tv * 2 + 1],
                upd_val[tv], tm, tr);
            if (upd_fl[tv * 2]) upd_val[tv * 2] = upd_push_val(
                upd_val[tv * 2], upd_val[tv]);
            else upd_val[tv * 2] = upd_val[tv];
            if (upd_fl[tv * 2 + 1]) upd_val[tv * 2 + 1] =
                upd_push_val(upd_val[tv * 2 + 1], upd_val[tv
                ]);
            else upd_val[tv * 2 + 1] = upd_val[tv];
            upd_fl[tv * 2] = upd_fl[tv * 2 + 1] = 1;
            upd_fl[tv] = 0;
        }
    }

    inline bool intersect(int l1, int r1, int l2, int r2) {
        return l1 < r2 && l2 < r1;
    }

    T get(int tv, int tl, int tr, int l, int r) {
        if (tl >= l && tr <= r) return st[tv];
        push(tv, tl, tr);
        int tm = (tl + tr) >> 1;
        if (!intersect(tl, tm, l, r)) return get(tv * 2 + 1, tm, tr, l,
            r);
        if (!intersect(tm, tr, l, r)) return get(tv * 2, tl, tm, l, r);
        return merge_nodes(get(tv * 2, tl, tm, l, r), get(tv * 2 +
            1, tm, tr, l, r), tl, tr);
    }

    void build(int tv, int tl, int tr, const vector<T> &a) {
        if (tr - tl == 1) {
            st[tv] = a[tl];
        } else {
            int tm = (tl + tr) >> 1;
            build(tv * 2, tl, tm, a);
            build(tv * 2 + 1, tm, tr, a);
            st[tv] = merge_nodes(st[tv * 2], st[tv * 2 + 1], tl, tr)
                ;
        }
    }

    void upd(int tv, int tl, int tr, int l, int r, TU val) {
        if (tl >= l && tr <= r) {
            st[tv] = recalc_on_segment(st[tv], val, tl, tr);
            if (upd_fl[tv]) upd_val[tv] = upd_push_val(upd_val
                [tv], val);
            else upd_val[tv] = val;
            upd_fl[tv] = 1;
        } else {
            push(tv, tl, tr);
            int tm = (tl + tr) >> 1;
            if (intersect(tl, tm, l, r)) upd(tv * 2, tl, tm, l, r, val);
            if (intersect(tm, tr, l, r)) upd(tv * 2 + 1, tm, tr, l, r,
                val);
            st[tv] = merge_nodes(st[tv * 2], st[tv * 2 + 1], tl, tr)
                ;
        }
    }

    int n{};
    vector<T> st;
    vector<TU> upd_val;
    vector<char> upd_fl;
    T merge_nodes(const T &i, const T &j, int tl, int tr) {
        return i + j;
    };
    T recalc_on_segment(const T &i, const TU &j, int tl, int tr
        ) {
        return i + (tr - tl) * j;
    };
    TU upd_push_val(const TU &i, const TU &j, int tl = 0, int
        tr = 1){
        return i + j;
    }
};
```

2.11 Rope

```
template<typename T>
class rope
{
public:
        int size() { if (root) return root->cnt; else return 0; }
private:
        struct item
        {
                ~item(void) { if (left) { delete left; left = nullptr;
                    } if (right) { delete right; right = nullptr; }
                    };
                item(void) : left(nullptr), right(nullptr), cnt(0),
                    prior(0), rev(false) {};
                item(T val) : left(nullptr), right(nullptr), cnt(0),
                    prior(rand()), rev(false), value(val), res(val)
                    {};

                T value;
                T res;
                item *left, *right;
                int prior;
                int cnt;
                bool rev;
        };
        using pitem = item*;

        pitem root;

        int cnt(pitem it) {
                return it ? it->cnt : 0;
        }
```

```
	void upd_vals(pitem it) {
		if (it)
		{
			it->cnt = cnt(it->left) + cnt(it->right) +
				1;
			it->res = it->value;
			if (it->left) it->res += it->left->res;
			if (it->right) it->res += it->right->res;
		}
	}

	friend void push(pitem it) {
		if (it && it->rev) {
			it->rev = false;
			swap(it->left, it->right);
			if (it->left) it->left->rev ^= true;
			if (it->right) it->right->rev ^= true;
		}
	}

	void merge(pitem& t, pitem l, pitem r) {
		push(l);
		push(r);
		if (!l || !r) t = l ? l : r;
		else if (l->prior > r->prior) merge(l->right, l->
			right, r), t = l;
		else merge(r->left, l, r->left), t = r;
		upd_vals(t);
	}

	void split(pitem t, pitem& l, pitem& r, int key, int add
		= 0) {
		if (!t) return void(l = r = 0);
		push(t);
		int cur_key = add + cnt(t->left);
		if (key <= cur_key) split(t->left, l, t->left, key,
			add), r = t;
		else split(t->right, t->right, r, key, add + 1 + cnt
			(t->left)), l = t;
		upd_vals(t);
	}

	void erase(pitem& t, int pos) {
		if (t->cnt == pos) merge(t, t->left, t->right);
		else erase(pos < t->cnt ? t->left : t->right, pos);
	}

	void heapify(pitem t) {
		if (!t) return;
		pitem mx = t;
		if (t->left && t->left->prior > mx->prior) mx =
			t->left;
		if (t->right && t->right->prior > mx->prior) mx
			= t->right;
		if (mx != t) {
			swap(t->prior, mx->prior);
			heapify(mx);
		}
	}

	pitem build(const T* a, int n) {
		if (n == 0) return nullptr;
		int mid = n / 2;
		pitem t = new item(a[mid]);
		t->left = build(a, mid);
		t->right = build(a + mid + 1, n - mid - 1);
		heapify(t);
		upd_vals(t);
		return t;
	}

public:
	rope() { root = nullptr; }
	~rope() { if (root) { delete root; root = nullptr; } }
	rope(int n, const T* a)
	{
		root = build(a, n);
	}

	void insert(int pos, T val) {
		pitem new_item = new item(val);
		pitem t1, t2;
		split(root, t1, t2, pos);
		merge(t1, t1, new_item);
		merge(root, t1, t2);
	}

	void insert(int pos, rope<T> *t) {
		pitem new_item = t->root;
		pitem t1, t2;
		split(root, t1, t2, pos);
		merge(t1, t1, new_item);
		merge(root, t1, t2);
	}

	void push_back(T val) { insert(size(), val); }
	void push_back(rope<T> *t) { insert(size(), t); }

	void erase(int pos) { erase(root, pos); }

	rope<T>* erase(int l, int r) {
		pitem t1, t2, t3;
		split(root, t1, t2, l);
		split(t2, t2, t3, r - l + 1);
		merge(root, t1, t3);
		auto t = new rope<T>;
		t->root = t2;
		return t;
	}

	void reverse(int l, int r) {
		pitem t1, t2, t3;
		split(root, t1, t2, l);
		split(t2, t2, t3, r - l + 1);
		t2->rev ^= true;
		merge(root, t1, t2);
		merge(root, root, t3);
	}

	T get_val(int l, int r) {
		pitem t1, t2, t3;
		split(root, t1, t2, l);
		split(t2, t2, t3, r - l + 1);
		auto cur_res = t2->res;
		merge(root, t1, t2);
		merge(root, root, t3);
		return cur_res;
	}

	rope<T>* substr(int l, int r)
	{
		auto t = new rope<T>;
		pitem t1, t2, t3;
		split(root, t1, t2, l);
		split(t2, t2, t3, r - l + 1);
		auto cur_res = t2;
		merge(root, t1, t2);
		merge(root, root, t3);
		t->root = t2;
		return t;
	}

	friend ostream& operator<< (ostream& out, pitem t) {
		if (!t) return out;
		push(t);
		out << t->left;
		out << t->value;
		out << t->right;
		return out;
	}

	friend ostream& operator<< (ostream& out, rope<T>
		&t) {
		out << t.root;
		return out;
	}

};
```

2.12 Trie

```
struct Trie {
    const int ALPHA = 26;
    const char BASE = 'a';
    vector<vector<int>> nextNode;
    vector<int> mark;
    int nodeCount;
    Trie() {
        nextNode = vector<vector<int>>(MAXN, vector<int>(
            ALPHA, -1));
        mark = vector<int>(MAXN, -1);
```

```
        nodeCount = 1;
    }
    void insert(const string& s, int id) {
        int curr = 0;
        FOR(i, 0, (int)s.length()) {
            int c = s[i] - BASE;
            if(nextNode[curr][c] == -1) {
                nextNode[curr][c] = nodeCount++;
            }
            curr = nextNode[curr][c];
        }
        mark[curr] = id;
    }

    bool exists(const string& s) {
        int curr = 0;
        FOR(i, 0, (int)s.length()) {
            int c = s[i] - BASE;
            if(nextNode[curr][c] == -1) return false;
            curr = nextNode[curr][c];
        }
        return mark[curr] != -1;
    }
};
```

3 Geometry

3.1 Common Tangents To Two Circles

```
struct pt {
    double x, y;

    pt operator- (pt p) {
        pt res = { x-p.x, y-p.y };
        return res;
    }
};
struct circle : pt {
    double r;
};
struct line {
    double a, b, c;
};
void tangents (pt c, double r1, double r2, vector<line> & ans) {
    double r = r2 - r1;
    double z = sqr(c.x) + sqr(c.y);
    double d = z - sqr(r);
    if (d < -eps) return;
    d = sqrt (abs (d));
    line l;
    l.a = (c.x * r + c.y * d) / z;
    l.b = (c.y * r - c.x * d) / z;
    l.c = r1;
    ans.push_back (l);
}
vector<line> tangents (circle a, circle b) {
    vector<line> ans;
    for (int i=-1; i<=1; i+=2)
        for (int j=-1; j<=1; j+=2)
            tangents (b-a, a.r*i, b.r*j, ans);
    for (size_t i=0; i<ans.size(); ++i)
        ans[i].c -= ans[i].a * a.x + ans[i].b * a.y;
    return ans;
}
```

3.2 Usage Of Complex

```
typedef long long C; // could be long double
typedef complex<C> P; // represents a point or vector
#define X real()
#define Y imag()
...
P p = {4, 2}; // p.X = 4, p.Y = 2
P u = {3, 1};
P v = {2, 2};
P s = v+u; // {5, 3}
P a = {4, 2};
P b = {3, -1};
auto l = abs(b-a); // 3.16228
auto plr = polar(1.0, 0.5); // construct a vector of length 1 and
    angle 0.5 radians
v = {2, 2};
auto alpha = arg(v); // 0.463648
v *= plr; // rotates v by 0.5 radians counterclockwise. The
    length of plt must be 1 to rotate correctly.
auto beta = arg(v); // 0.963648
a = {4, 2};
b = {1, 2};
C p = (conj(a)*b).Y; // 6 <- the cross product of a and b
```

3.3 Circle Line Intersection

```
double r, a, b, c; // ax+by+c=0, radius is at (0, 0)
// If the center is not at (0, 0), fix the constant c to translate
    everything so that center is at (0, 0)
double x0 = -a*c/(a*a+b*b), y0 = -b*c/(a*a+b*b);
if (c*c > r*r*(a*a+b*b)+eps)
    puts ("no points");
else if (abs (c*c - r*r*(a*a+b*b)) < eps) {
    puts ("1 point");
    cout << x0 << ' ' << y0 << '\n';
}
else {
    double d = r*r - c*c/(a*a+b*b);
    double mult = sqrt (d / (a*a+b*b));
    double ax, ay, bx, by;
    ax = x0 + b * mult;
    bx = x0 - b * mult;
    ay = y0 - a * mult;
    by = y0 + a * mult;
    puts ("2 points");
    cout << ax << ' ' << ay << '\n' << bx << ' ' << by
        << '\n';
}
```

3.4 2d Vector

```
template <typename T>
struct Vec {
    T x, y;
    Vec(): x(0), y(0) {}
    Vec(T _x, T _y): x(_x), y(_y) {}
    Vec operator+(const Vec& b) {
        return Vec<T>(x+b.x, y+b.y);
    }
    Vec operator-(const Vec& b) {
        return Vec<T>(x-b.x, y-b.y);
    }
    Vec operator*(T c) {
        return Vec(x*c, y*c);
    }
    T operator*(const Vec& b) {
        return x*b.x + y*b.y;
    }
    T operator^(const Vec& b) {
        return x*b.y-y*b.x;
    }
    bool operator<(const Vec& other) const {
        if(x == other.x) return y < other.y;
        return x < other.x;
    }
    bool operator==(const Vec& other) const {
        return x==other.x && y==other.y;
    }
    bool operator!=(const Vec& other) const {
        return !(*this == other);
    }
    friend ostream& operator<<(ostream& out, const Vec& v) {
        return out << "(" << v.x << ", " << v.y << ")";
    }
    friend istream& operator>>(istream& in, Vec<T>& v) {
        return in >> v.x >> v.y;
    }
    T norm() { // squared length
        return (*this)*(*this);
    }
```

```
    ld len() {
        return sqrt(norm());
    }
    ld angle(const Vec& other) { // angle between this and
          other vector
        return acosl((*this)*other/len()/other.len());
    }
    Vec perp() {
        return Vec(-y, x);
    }
};
/* Cross product of 3d vectors: (ay*bz-az*by, az*bx-ax*bz, ax*
     by-ay*bx)
 */
```

3.5 Convex Hull With Graham's Scan

```
// Takes in >= 3 points
// Returns convex hull in clockwise order
// Ignores points on the border
vector<Vec<int>> buildConvexHull(vector<Vec<int>> pts) {
    if(pts.size() <= 3) return pts;
    sort(pts.begin(), pts.end());
    stack<Vec<int>> hull;
    hull.push(pts[0]);
    auto p = pts[0];
    sort(pts.begin()+1, pts.end(), [&](Vec<int> a, Vec<int> b)
          -> bool {
        // p->a->b is a ccw turn
        int turn = sgn((a-p)^(b-a));
        //if(turn == 0) return (a-p).norm() > (b-p).norm();
        // ^ among collinear points, take the farthest one
        return turn == 1;
    });
    hull.push(pts[1]);
    FOR(i, 2, (int)pts.size()) {
        auto c = pts[i];
        if(c == hull.top()) continue;
        while(true) {
            auto a = hull.top(); hull.pop();
            auto b = hull.top();
            auto ba = a-b;
            auto ac = c-a;
            if((ba^ac) > 0) {
                hull.push(a);
                break;
            } else if((ba^ac) == 0) {
                if(ba*ac < 0) c = a;
                // ^ c is between b and a, so it shouldn't be
                      added to the hull
                break;
            }
        }
        hull.push(c);
    }
    vector<Vec<int>> hullPts;
    while(!hull.empty()) {
        hullPts.pb(hull.top());
        hull.pop();
    }
    return hullPts;
}
```

3.6 Misc

Distance from point to line.

We have a line $l(t) = \vec{a} + \vec{b}t$ and a point $\vec{p}$. The distance from this point to the line can be calculated by expressing the area of a triangle in two different ways. The final formula: $d = \frac{(\vec{p}-\vec{a})\times(\vec{p}-\vec{b})}{|\vec{b}-\vec{a}|}$

Point in polygon.

Send a ray (half-infinite line) from the points to an arbitrary direction and calculate the number of times it touches the boundary of the polygon. If the number is odd, the point is inside the polygon, otherwise it's outside.

Using cross product to test rotation direction.

Let's say we have vectors $\vec{a}$, $\vec{b}$ and $\vec{c}$. Let's define $\vec{ab} = b - a, \vec{bc} = c - b$ and $s = sgn(\vec{ab} \times \vec{bc})$. If $s = 0$, the three points are collinear. If $s = 1$, then $\vec{bc}$ turns in the counterclockwise direction compared to the direction of $\vec{ab}$. Otherwise it turns in the clockwise direction.

Line segment intersection.

The problem: to check if line segments ab and cd intersect. There are three cases:

1. **The line segments are on the same line.** Use cross products and check if they're zero - this will tell if all points are on the same line. If so, sort the points and check if their intersection is non-empty. If it is non-empty, there are an infinite number of intersection points.
2. **The line segments have a common vertex.** Four possibilities: $a = c, a = d, b = c, b = d$.
3. **There is exactly one intersection point that is not an endpoint.** Use cross product to check if points c and d are on different sides of the line going through a and b and if the points a and b are on different sides of the line going through c and d.

Angle between vectors.

$arccos(\frac{\vec{a}\cdot\vec{b}}{|\vec{a}|\cdot|\vec{b}|})$.

Dot product properties.

If the dot product of two vectors is zero, the vectors are orthogonal. If it is positive, the angle is acute. Otherwise it is obtuse.

Lines with line equation.

Any line can be described by an equation $ax + by + c = 0$.

- Construct a line using two points A and B:
    1. Take vector from A to B and rotate it 90 degrees ($(x, y) \rightarrow (-y, x)$). This will be (a, b).
    2. Normalize this vector. Then put A (or B) into the equation and solve for c.
- Distance from point to line: put point coordinates into line equation and take absolute value. If (a, b) is not normalized, you still need to divide by $\sqrt{a^2 + b^2}$.

- Distance between two parallel lines: $|c_1 - c_2|$ (if they are not normalized, you still need to divide by $\sqrt{a^2+b^2}$).
- Project a point onto a line: compute signed distance d between line L and point P. Answer is $P - d(\vec{a,b})$.
- Build a line parallel to a given one and passing through a given point: compute the signed distance d between line and point. Answer is $ax + by + (c - d) = 0$.
- Intersect two lines: $d = a_1 b_2 - a_2 b_1, x = \frac{c_2 b_1 - c_1 b_2}{d}, y = \frac{c_1 a_2 - c_2 a_1}{d}$. If $abs(d) < \epsilon$, then the lines are parallel.

Half-planes.

Definition: define as line, assume a point (x, y) belongs to half plane iff $ax + by + c \geq 0$.

Intersecting with a convex polygon:

1. Start at any point and move along the polygon's traversal.
2. Alternate points and segments between consecutive points.
3. If point belongs to half-plane, add it to the answer.
4. If segment intersects the half-plane's line, add it to the answer.

Some more techniques.

- Check if point A lies on segment BC:
  1. Compute point-line distance and check if it is 0 (abs less than ϵ).
  2. $\vec{BA} \cdot \vec{BC} \geq 0$ and $\vec{CA} \cdot \vec{CB} \geq 0$.
- Compute distance between line segment and point: project point onto line formed by the segment. If this point is on the segment, then the distance between it and original point is the answer. Otherwise, take minimum of distance between point and segment endpoints.

3.7 Circle Circle Intersection

Let's say that the first circle is centered at $(0, 0)$ (if it's not, we can move the origin to the center of the first circle and adjust the coordinates), and the second one is at (x_2, y_2). Then, let's construct a line $Ax + By + C = 0$, where $A = -2x_2, B = -2y_2, C = x_2^2 + y_2^2 + r_1^2 - r_2^2$. Finding the intersection between this line and the first circle will give us the answer. The only tricky case: if both circles are centered at the same point. We handle this case separately.

3.8 Line

```
template <typename T>
struct Line { // expressed as two vectors
    Vec<T> start, dir;
    Line() {}
    Line(Vec<T> a, Vec<T> b): start(a), dir(b-a) {}
    Vec<ld> intersect(Line l) {
        ld t = ld((l.start-start)^l.dir)/(dir^l.dir);
        // For segment-segment intersection this should be in
            range [0, 1]
        Vec<ld> res(start.x, start.y);
        Vec<ld> dirld(dir.x, dir.y);
        return res + dirld*t;
    }
};
```

3.9 Number Of Lattice Points On Segment

Let's say we have a line segment from (x_1, y_1) to (x_2, y_2). Then, the number of lattice points on this segment is given by

$$gcd(x_2 - x_1, y_2 - y_1) + 1.$$

3.10 Pick's Theorem

We are given a lattice polygon with non-zero area. Let's denote its area by S, the number of points with integer coordinates lying strictly inside the polygon by I and the number of points lying on the sides of the polygon by B. Then:

$$S = I + \frac{B}{2} - 1.$$

4 Graphs

4.1 Finding Bridges And Cutpoints

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, fup;
int timer;

void processCutpoint(int v) {
    // problem-specific logic goes here
    // it can be called multiple times for the same v
}

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = fup[v] = timer++;
    int children=0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            fup[v] = min(fup[v], tin[to]);
        } else {
            dfs(to, v);
            fup[v] = min(fup[v], fup[to]);
            if (fup[to] >= tin[v] && p!=-1)
                processCutpoint(v);
            ++children;
        }
    }
    if(p == -1 && children > 1)
        processCutpoint(v);
}

void findCutpoints() {
    timer = 0;
```

```
    visited.assign(n, false);
    tin.assign(n, -1);
    fup.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs (i);
    }
}
```

4.2 Heavy Light Decomposition

```
template<typename T, class F = function<T(const T &, const
        T &)>>
struct SegmentTree {
    int n{};
    vector<T> st;
    F merge;
    T neutral{};
    SegmentTree()= default;
    explicit SegmentTree(const vector<T> &a, F _merge, T
            _neutral) {
        n = (int) a.size();
        st.resize(2 * (int) a.size());
        merge = _merge;
        neutral = _neutral;
        for (int i = 0; i < n; i++) st[i + n] = a[i];
        for (int i = n - 1; i > 0; i--) st[i] = merge(st[i << 1], st[i
                << 1 | 1]);
    }

    T get(int l, int r) {
        T res = neutral;
        for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
            if (l & 1) res = merge(res, st[l++]);
            if (r & 1) res = merge(res, st[--r]);
        }
        return res;
    }

    void upd(int p, T val) {
        for (st[p += n] = val; p > 1; p >>= 1) st[p >> 1] =
                merge(st[p], st[p ^ 1]);
    }
};

template<typename T, class F = function<T(const T &, const
        T &)>>
struct HLD {
    HLD(int n, vector<T> &values, vector<pair<int, int>> &
            edges) {
        this->timer = 0;
        this->n = n;
        this->g.resize(n);
        this->treeSize.resize(n);
        this->topId.resize(n);
        this->pos.resize(n);
        this->parent.resize(n);
        this->depth.resize(n);
        for (auto[u, v] : edges) {
            g[u].emplace_back(v);
            g[v].emplace_back(u);
        }

        Calculate(0, 0);
        for (int i = 0; i < n; i++) {
            std::sort(g[i].rbegin(), g[i].rend(), [&](int u, int v) {
                return treeSize[u] < treeSize[v];
            });
        }

        BuildHld(0, 0, 0);
        auto orderedValues = values;
        for (int i = 0; i < n; i++) {
            orderedValues[pos[i]] = values[i];
        }
        tree = SegmentTree<T>(orderedValues, merge, neutral);
    }

    T get(int u, int v) {
        T ans = neutral;
        while (topId[u] != topId[v]) {
            if (depth[topId[u]] < depth[topId[v]]) swap(u, v);
            int low = pos[topId[u]];
            int high = pos[u] + 1;
            ans = merge(ans, tree.get(low, high));
            u = parent[topId[u]];
        }

        int low = pos[u];
        int high = pos[v];
        if (low > high) swap(low, high);
        ans = merge(ans, tree.get(low, high + 1));
        return ans;
    }

    void upd(int v, T val) {
        tree.upd(pos[v], val);
    }

private:
    int n{};
    int timer{};
    vector<int> depth;
    vector<int> treeSize;
    vector<vector<int>> g;
    vector<int> topId;
    vector<int> pos;
    SegmentTree<T> tree{};
    vector<int> parent;

    void BuildHld(int v, int curTopId, int curParent = -1) {
        pos[v] = timer++;
        topId[v] = curTopId;
        bool heavyEdge = true;
        for (auto to : g[v]) {
            if (to == curParent) continue;
            if (heavyEdge) {
                BuildHld(to, curTopId, v);
                heavyEdge ^= 1;
            } else {
                BuildHld(to, to, v);
            }
        }
    }

    void Calculate(int v, int curParent) {
        parent[v] = curParent;
        treeSize[v] = 1;
        depth[v] = depth[curParent] + 1;
        for (auto to : g[v]) {
            if (to == curParent) continue;
            Calculate(to, v);
            treeSize[v] += treeSize[to];
        }
    }
    F merge = [&](const T &i, const T &j) { return i + j; };
    T neutral = 0;
};
```

4.3 Max Flow With Dinic

```
struct flow_graph {
    struct edge {
        int u, v, f = 0, c;
        edge(int u, int v, int c) : u(u), v(v), c(c) {}
    };
    int n, s{}, t{};
    vector<edge> edges;
    vector<vector<int>> gr;
    vector<int> ptr, d;
    long long max_flow = 0;
    explicit flow_graph(int n) : n(n){
        gr.resize(n);
    }
    void add_edge(int u, int v, int c, int back = 0) {
        gr[u].emplace_back(edges.size());
        edges.emplace_back(u, v, c);
        gr[v].emplace_back(edges.size());
        edges.emplace_back(v, u, back);
    }
    bool bfs() {
        d.assign(n, -1);
        d[s] = 0;
        vector<int> q = {s};
        for (int i = 0; i < q.size(); ++i) {
            for (auto &id: gr[q[i]]) {
                edge &e = edges[id];
                if (edges[id].f < edges[id].c && d[e.v] == -1) {
```

```
                d[e.v] = d[e.u] + 1;
                q.emplace_back(e.v);
            }
        }
    }
    return d[t] != -1;
}
int dfs(int v, int flow) {
    if (flow == 0) return 0;
    if (v == t) return flow;
    for (; ptr[v] < gr[v].size(); ptr[v]++) {
        int id = gr[v][ptr[v]];
        edge &e = edges[id];
        if (d[e.v] != d[e.u] + 1) continue;
        int delta = dfs(e.v, min(flow, e.c - e.f));
        if (delta){
            e.f += delta;
            edges[id ^ 1].f -= delta;
            return delta;
        }
    }
    return 0;
}
long long get_flow(int _s, int _t) {
    s = _s, t = _t;
    max_flow = 0;
    while (bfs()) {
        ptr.assign(n, 0);
        while(int cur = dfs(s, 2e9)) max_flow += cur;
    }
    return max_flow;
}
};
```

4.4 Lowest Common Ancestor

```
int n, l; // l == logN (usually about ~20)
vector<vector<int>> adj;

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

void dfs(int v, int p)
{
    tin[v] = ++timer;
    up[v][0] = p;
    // wUp[v][0] = weight[v][u]; // <- path weight sum to 2^i-th
        ancestor
    for (int i = 1; i <= l; ++i)
        up[v][i] = up[up[v][i-1]][i-1];
        // wUp[v][i] = wUp[v][i-1] + wUp[up[v][i-1]][i-1];

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

    tout[v] = ++timer;
}

bool isAncestor(int u, int v)
{
    return tin[u] <= tin[v] && tout[v] <= tout[u];
}

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

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

4.5 Dijkstra

```
void dijkstra(vector<vector<pll>>& g, vi& p, vll& d, int start)
{
        priority_queue<pll, vector<pll>, greater<pll>> q;
        d[start] = 0;
        q.push({ 0, start });
        while (!q.empty())
        {
                auto from = q.top().second;
                auto dist = q.top().first;
                q.pop();
                if (dist > d[from]) continue;
                for (auto& cur : g[from])
                {
                        auto to = cur.second;
                        auto to_dist = cur.first;
                        if (d[from] + to_dist < d[to])
                        {
                                d[to] = d[from] + to_dist;
                                p[to] = from;
                                q.push({ d[to], to });
                        }
                }
        }
}
```

4.6 Dfs With Timestamps

```
vector<vector<int>> adj;
vector<int> tIn, tOut, color;
int dfs_timer = 0;

void dfs(int v) {
    tIn[v] = dfs_timer++;
    color[v] = 1;
    for (int u : adj[v])
        if (color[u] == 0)
            dfs(u);
    color[v] = 2;
    tOut[v] = dfs_timer++;
}
```

4.7 Bipartite Graph

```
class BipartiteGraph {
private:
    vector<int> _left, _right;
    vector<vector<int>> _adjList;
    vector<int> _matchR, _matchL;
    vector<bool> _used;

    bool _kuhn(int v) {
        if (_used[v]) return false;
        _used[v] = true;
        FOR(i, 0, (int)_adjList[v].size()) {
            int to = _adjList[v][i] - _left.size();
            if (_matchR[to] == -1 || _kuhn(_matchR[to])) {
                _matchR[to] = v;
                _matchL[v] = to;
                return true;
            }
        }
        return false;
    }
    void _addReverseEdges() {
        FOR(i, 0, (int)_right.size()) {
            if (_matchR[i] != -1) {
                _adjList[_left.size() + i].pb(_matchR[i]);
            }
        }
    }
```

```
    void __dfs(int p) {
        if (__used[p]) return;
        __used[p] = true;
        for (auto x : __adjList[p]) {
            __dfs(x);
        }
    }
    vector<pii> __buildMM() {
        vector<pair<int, int> > res;
        FOR(i, 0, (int)__right.size()) {
            if (__matchR[i] != -1) {
                res.push_back(make_pair(__matchR[i], i));
            }
        }

        return res;
    }
public:
    void addLeft(int x) {
        __left.pb(x);
        __adjList.pb({});
        __matchL.pb(-1);
        __used.pb(false);
    }
    void addRight(int x) {
        __right.pb(x);
        __adjList.pb({});
        __matchR.pb(-1);
        __used.pb(false);
    }
    void addForwardEdge(int l, int r) {
        __adjList[l].pb(r + __left.size());
    }
    void addMatchEdge(int l, int r) {
        if(l != -1) __matchL[l] = r;
        if(r != -1) __matchR[r] = l;
    }
    // Maximum Matching
    vector<pii> mm() {
        __matchR = vector<int>(__right.size(), -1);
        __matchL = vector<int>(__left.size(), -1);
        // ^ these two can be deleted if performing MM on
            already partially matched graph
        __used = vector<bool>(__left.size() + __right.size(), false
            );


        bool path_found;
        do {
            fill(__used.begin(), __used.end(), false);
            path_found = false;
            FOR(i, 0, (int)__left.size()) {
                if (__matchL[i] < 0 && !__used[i]) {
                    path_found |= __kuhn(i);
                }
            }
        } while (path_found);

        return __buildMM();
    }

    // Minimum Edge Cover
    // Algo: Find MM, add unmatched vertices greedily.
    vector<pii> mec() {
        auto ans = mm();
        FOR(i, 0, (int)__left.size()) {
            if (__matchL[i] != -1) {
                for (auto x : __adjList[i]) {
                    int ridx = x - __left.size();
                    if (__matchR[ridx] == -1) {
                        ans.pb({ i, ridx });
                        __matchR[ridx] = i;
                    }
                }
            }
        }
        FOR(i, 0, (int)__left.size()) {
            if (__matchL[i] == -1 && (int)__adjList[i].size() > 0)
                {
                int ridx = __adjList[i][0] - __left.size();
                __matchL[i] = ridx;
                ans.pb({ i, ridx });
            }
        }
        return ans;
    }

    // Minimum Vertex Cover
    // Algo: Find MM. Run DFS from unmatched vertices from
        the left part.
    // MVC is composed of unvisited LEFT and visited RIGHT
        vertices.
    pair<vector<int>, vector<int>> mvc(bool runMM = true)
        {
        if (runMM) mm();
        __addReverseEdges();
        fill(__used.begin(), __used.end(), false);
        FOR(i, 0, (int)__left.size()) {
            if (__matchL[i] == -1) {
                __dfs(i);
            }
        }
        vector<int> left, right;
        FOR(i, 0, (int)__left.size()) {
            if (!__used[i]) left.pb(i);
        }
        FOR(i, 0, (int)__right.size()) {
            if (__used[i + (int)__left.size()]) right.pb(i);
        }
        return { left,right };
    }

    // Maximal Independant Vertex Set
    // Algo: Find complement of MVC.
    pair<vector<int>, vector<int>> mivs(bool runMM = true)
        {
        auto m = mvc(runMM);
        vector<bool> containsL(__left.size(), false), containsR(
            __right.size(), false);
        for (auto x : m.first) containsL[x] = true;
        for (auto x : m.second) containsR[x] = true;
        vector<int> left, right;
        FOR(i, 0, (int)__left.size()) {
            if (!containsL[i]) left.pb(i);
        }
        FOR(i, 0, (int)__right.size()) {
            if (!containsR[i]) right.pb(i);
        }
        return { left, right };
    }

};
```

4.8 Shortest Paths Of Fixed Length

Define $A \bigodot B = C \iff C_{ij} = min_{p=1..n}(A_{ip} + B_{pj})$. Let G be the adjacency matrix of a graph. Also, let $L_k = G \bigodot \ldots \bigodot G = G^{\bigodot k}$. Then the value $L_k[i][j]$ denotes the length of the shortest path between i and j which consists of exactly k edges.

4.9 Number Of Paths Of Fixed Length

Let G be the adjacency matrix of a graph. Then $C_k = G^k$ gives a matrix, in which the value $C_k[i][j]$ gives the number of paths between i and j of length k.

4.10 Strongly Connected Components

```
void dfs(int v, vvi& g, vi& used, vi &topsort)
{
        used[v] = 1;
        for (auto& to : g[v])
        {
                if (!used[to]) dfs(to, g, used, topsort);
        }
        topsort.push_back(v);
}
```

```
void dfs(int v, vvi& g, vi& used, vvi &components)
{
	used[v] = 1;
	components.back().push_back(v);
	for (auto& to : g[v])
	{
		if (!used[to]) dfs(to, g, used, components);
	}
}

vvi build_scc(vvi& g, vvi &rg)
{
	vi used(g.size(), 0);
	vi rused(rg.size(), 0);
	vvi components;
	vi topsort;
	int n = g.size();

	for (int i = 1; i < n; i++)
	{
		if (used[i]) continue;
		dfs(i, g, used, topsort);
	}

	reverse(all(topsort));

	for (int i = 0; i < topsort.size(); i++)
	{
		int v = topsort[i];
		if (rused[v]) continue;
		components.push_back(vi());
		dfs(v, rg, rused, components);
	}

	return components;
}
```

4.11 Min Spanning Tree

```
class DSU
{
private:
	vector<int> p;
public:
	DSU(int sz) { p.resize(sz); }

	void make_set(int v) {
		p[v] = v;
	}

	int get(int v) {
		return (v == p[v]) ? v : (p[v] = get(p[v]));
	}

	void unite(int a, int b) {
		a = get(a);
		b = get(b);
		if (rand() & 1)
			swap(a, b);
		if (a != b)
			p[a] = b;
	}
};

vector<pair<pii, LL>> min_spanning_tree(vector<pair<pii,
	LL>>& edges, LL n)
{
	vector<pair<pii, LL>> res;
	sort(all(edges), [](pair<pii, LL>& x1, pair<pii, LL>& x2
		) { return x1.second < x2.second; });
	DSU dsu(n + 1);

	for (int i = 1; i < n + 1; i++) dsu.make_set(i);

	for (auto &cur_edge : edges)
	{
		int u = cur_edge.first.first;
		int v = cur_edge.first.second;
		LL w = cur_edge.second;
		auto p1 = dsu.get(u);
		auto p2 = dsu.get(v);
		if (p1 != p2)
		{
			res.push_back(cur_edge);
			dsu.unite(u, v);
		}
	}

	return res;
}
```

4.12 Min Cut

```
init();
ll f = maxFlow(); // Ford-Fulkerson
cur_time++;
dfs(0);
set<int> cc;
for (auto e : edges) {
    if (timestamp[e.from] == cur_time && timestamp[e.to] !=
        cur_time) {
        cc.insert(e.idx);
    }
}
// (# of edges in min-cut, capacity of cut)
// [indices of edges forming the cut]
cout << cc.size() << " " << f << endl;
for (auto x : cc) cout << x + 1 << " ";
```

4.13 Bellman Ford Algorithm

```
/* Finds SSSP with negative edge weights.
 * Possible optimization: check if anything changed in a
	relaxation step. If not - you can break early.
 * To find a negative cycle: perform one more relaxation step. If
	anything changes - a negative cycle exists.
 */
set<int> ford_bellman(vector<pair<pll, LL>> &edges, vll &d,
	vi &p, int start, LL n)
{
	d[start] = 0;
	set<int> cycle_vertexes;
	for (int i = 0; i < n; i++)
	{
		cycle_vertexes.clear();
		for (auto &cur_edge : edges)
		{
			auto from = cur_edge.first.first;
			auto to = cur_edge.first.second;
			auto dist = cur_edge.second;
			if (d[from] < LLONG_MAX && d[to] > d
				[from] + dist)
			{
				d[to] = d[from] + dist;
				p[to] = from;
				cycle_vertexes.insert(to);
			}
		}
	}
	set<int> res;
	for (auto& v : cycle_vertexes)
	{
		int cur_v = v;
		for (int i = 0; i < n; i++) cur_v = p[cur_v];
		res.insert(cur_v);
	}

	return res;
}
```

5 Strings

5.1 Prefix Function Automaton

```
// aut[oldPi][c] = newPi
vector<vector<int>> computeAutomaton(string s) {
    const char BASE = 'a';
    s += "#";
```

```
    int n = s.size();
    vector<int> pi = prefixFunction(s);
    vector<vector<int>> aut(n, vector<int>(26));
    for (int i = 0; i < n; i++) {
        for (int c = 0; c < 26; c++) {
            if (i > 0 && BASE + c != s[i])
                aut[i][c] = aut[pi[i-1]][c];
            else
                aut[i][c] = i + (BASE + c == s[i]);
        }
    }
    return aut;
}
vector<int> findOccurs(const string& s, const string& t) {
    auto aut = computeAutomaton(s);
    int curr = 0;
    vector<int> occurs;
    FOR(i, 0, (int)t.length()) {
        int c = t[i]-'a';
        curr = aut[curr][c];
        if(curr == (int)s.length()) {
            occurs.pb(i-s.length()+1);
        }
    }
    return occurs;
}
```

5.2 Prefix Function

```
// pi[i] is the length of the longest proper prefix of the substring
        s[0..i] which is also a suffix
// of this substring
vector<int> prefixFunction(const string& s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
```

5.3 KMP

```
// Knuth-Morris-Pratt algorithm
vector<int> findOccurences(const string& s, const string& t) {
    int n = s.length();
    int m = t.length();
    string S = s + "#" + t;
    auto pi = prefixFunction(S);
    vector<int> ans;
    FOR(i, n+1, n+m+1) {
        if(pi[i] == n) {
            ans.pb(i-2*n);
        }
    }
    return ans;
}
```

5.4 Aho Corasick Automaton

```
// alphabet size
const int K = 70;

// the indices of each letter of the alphabet
int intVal[256];
void init() {
    int curr = 2;
    intVal[1] = 1;
    for(char c = '0'; c <= '9'; c++, curr++) intVal[(int)c] =
            curr;
    for(char c = 'A'; c <= 'Z'; c++, curr++) intVal[(int)c] =
            curr;
    for(char c = 'a'; c <= 'z'; c++, curr++) intVal[(int)c] =
            curr;
}

struct Vertex {
    int next[K];
    vector<int> marks;
    // ^ this can be changed to int mark = -1, if there will be
            no duplicates
    int p = -1;
    char pch;
    int link = -1;
    int exitLink = -1;
    // ^ exitLink points to the next node on the path of suffix
            links which is marked
    int go[K];

    // ch has to be some small char
    Vertex(int _p=-1, char ch=(char)1) : p(_p), pch(ch) {
        fill(begin(next), end(next), -1);
        fill(begin(go), end(go), -1);
    }
};

vector<Vertex> t(1);

void addString(string const& s, int id) {
    int v = 0;
    for (char ch : s) {
        int c = intVal[(int)ch];
        if (t[v].next[c] == -1) {
            t[v].next[c] = t.size();
            t.emplace_back(v, ch);
        }
        v = t[v].next[c];
    }
    t[v].marks.pb(id);
}

int go(int v, char ch);

int getLink(int v) {
    if (t[v].link == -1) {
        if (v == 0 || t[v].p == 0)
            t[v].link = 0;
        else
            t[v].link = go(getLink(t[v].p), t[v].pch);
    }
    return t[v].link;
}

int getExitLink(int v) {
    if(t[v].exitLink != -1) return t[v].exitLink;
    int l = getLink(v);
    if(l == 0) return t[v].exitLink = 0;
    if(!t[l].marks.empty()) return t[v].exitLink = l;
    return t[v].exitLink = getExitLink(l);
}

int go(int v, char ch) {
    int c = intVal[(int)ch];
    if (t[v].go[c] == -1) {
        if (t[v].next[c] != -1)
            t[v].go[c] = t[v].next[c];
        else
            t[v].go[c] = v == 0 ? 0 : go(getLink(v), ch);
    }
    return t[v].go[c];
}

void walkUp(int v, vector<int>& matches) {
    if(v == 0) return;
    if(!t[v].marks.empty()) {
        for(auto m : t[v].marks) matches.pb(m);
    }
    walkUp(getExitLink(v), matches);
}

// returns the IDs of matched strings.
// Will contain duplicates if multiple matches of the same string
        are found.
vector<int> walk(const string& s) {
    vector<int> matches;
    int curr = 0;
    for(char c : s) {
        curr = go(curr, c);
```

```
        if(!t[curr].marks.empty()) {
            for(auto m : t[curr].marks) matches.pb(m);
        }
        walkUp(getExitLink(curr), matches);
    }
    return matches;
}
/* Usage:
 * addString(strs[i], i);
 * auto matches = walk(text);
 * .. do what you need with the matches - count, check if some
       id exists, etc ..
 * Some applications:
 * - Find all matches: just use the walk function
 * - Find lexicographically smallest string of a given length that
        doesn't match any of the given strings:
 * For each node, check if it produces any matches (it either
       contains some marks or walkUp(v) returns some marks).
 * Remove all nodes which produce at least one match. Do DFS
        in the remaining graph, since none of the remaining
       nodes
 * will ever produce a match and so they're safe.
 * - Find shortest string containing all given strings:
 * For each vertex store a mask that denotes the strings which
       match at this state. Start at (v = root, mask = 0),
 * we need to reach a state (v, mask=2^n-1), where n is the
       number of strings in the set. Use BFS to transition
       between states
 * and update the mask.
 */
```

5.5 Suffix Fsm

```
class SuffFSM
{
private:
        vector<SuffFSMState> st;
        vector<SuffFSMState> sorted_st;
        vector<LL> dp;
        int last_state;

        void extend(char c)
        {
                int cur_state = st.size();
                st.push_back(SuffFSMState(cur_state, st[
                        last_state].len + 1, -1, 1, st[last_state].len))
                        ;
                int p;
                for (p = last_state; p != -1 && !st[p].next.count(
                        c); p = st[p].link)
                          st[p].next[c] = cur_state;
                if (p == -1) st[cur_state].link = 0;
                else
                {
                        int q = st[p].next[c];
                        if (st[p].len + 1 == st[q].len) st[cur_state
                                ].link = q;
                        else
                        {
                                int clone_state = st.size();
                                st.push_back(SuffFSMState(
                                        clone_state, st[p].len + 1, st[
                                        q].link, 0, 0));
                                st[clone_state].next = st[q].next;
                                st[clone_state].first_pos = st[q].
                                        first_pos;
                                for (; p != -1 && st[p].next[c] == q
                                        ; p = st[p].link)
                                          st[p].next[c] = clone_state;
                                st[q].link = st[cur_state].link =
                                        clone_state;
                        }
                }
                last_state = cur_state;
        }

        void build(string& s)
        {
                for (int i = 0; i < s.length(); i++) extend(s[i]);
        }

        void calc_cnts()
        {
                sorted_st = st;
                sort(all(sorted_st), [](SuffFSMState& st1,
                        SuffFSMState& st2) { return st1.len > st2.
                        len; });
                vi id_map(st.size());

                for (int i = 0; i < sorted_st.size(); i++)
                {
                        id_map[sorted_st[i].id] = i;
                }


                for (auto &cur_state : sorted_st)
                {
                        if (cur_state.link != -1)
                        {
                                st[cur_state.link].cnt += cur_state
                                        .cnt;
                                sorted_st[id_map[cur_state.link]].
                                        cnt = st[cur_state.link].cnt;
                        }
                }
        }


        LL fsm_dfs(int v)
        {
                if (dp[v] != -1) return dp[v];
                LL sum = 0;
                for (auto& to : st[v].next)
                {
                        sum += fsm_dfs(to.second);
                }
                dp[v] = sum + 1;
                return dp[v];
        }


public:
        SuffFSM() : last_state(0) { st.push_back(SuffFSMState
                ()); }
        SuffFSM(string &s) : last_state(0) {
                st.push_back(SuffFSMState());
                build(s);
                calc_cnts();
        }

        bool check_occurrence(string& t)
        {
                int cur_state = 0;
                for (int i = 0; i < t.length(); i++)
                {
                        if (st[cur_state].next.count(t[i]) == 0)
                                return false;
                        cur_state = st[cur_state].next[t[i]];
                }
                return true;
        }

        LL calc_occurrence(string& t)
        {
                int cur_state = 0;
                for (int i = 0; i < t.length(); i++)
                {
                        if (st[cur_state].next.count(t[i]) == 0)
                                return 0;
                        cur_state = st[cur_state].next[t[i]];
                }
                return st[cur_state].cnt;
        }

        int get_pos(string& t)
        {
                int cur_state = 0;
                for (int i = 0; i < t.length(); i++)
                {
                        if (st[cur_state].next.count(t[i]) == 0)
                                return -1;
                        cur_state = st[cur_state].next[t[i]];
                }
                return st[cur_state].first_pos - t.length() + 1;
        }

        LL distinct_substrs_cnt()
        {
                dp.clear();
                dp.resize(st.size(), -1);
                fsm_dfs(0);
                return dp[0] - 1;
```

```
    }

    string get_kth_substr(LL k)
    {
        distinct_substrs_cnt();
        string res = "";
        int cur_state = 0;
        while (k)
        {
            for (auto& cur : st[cur_state].next)
            {
                if (k <= dp[cur.second])
                {
                    res.push_back(cur.first);
                    cur_state = cur.second;
                    k--;
                    break;
                }
                else k -= dp[cur.second];
            }
        }
        return res;
    }
};
```

5.6 Suffix Array

```
vector<int> sortCyclicShifts(string const& s) {
    int n = s.size();
    const int alphabet = 256; // we assume to use the whole
        ASCII range
    vector<int> p(n), c(n), cnt(max(alphabet, n), 0);
    for (int i = 0; i < n; i++)
        cnt[s[i]]++;
    for (int i = 1; i < alphabet; i++)
        cnt[i] += cnt[i-1];
    for (int i = 0; i < n; i++)
        p[--cnt[s[i]]] = i;
    c[p[0]] = 0;
    int classes = 1;
    for (int i = 1; i < n; i++) {
        if (s[p[i]] != s[p[i-1]])
            classes++;
        c[p[i]] = classes - 1;
    }
    vector<int> pn(n), cn(n);
    for (int h = 0; (1 << h) < n; ++h) {
        for (int i = 0; i < n; i++) {
            pn[i] = p[i] - (1 << h);
            if (pn[i] < 0)
                pn[i] += n;
        }
        fill(cnt.begin(), cnt.begin() + classes, 0);
        for (int i = 0; i < n; i++)
            cnt[c[pn[i]]]++;
        for (int i = 1; i < classes; i++)
            cnt[i] += cnt[i-1];
        for (int i = n-1; i >= 0; i--)
            p[--cnt[c[pn[i]]]] = pn[i];
        cn[p[0]] = 0;
        classes = 1;
        for (int i = 1; i < n; i++) {
            pair<int, int> cur = {c[p[i]], c[(p[i] + (1 << h)) % n
                ]};
            pair<int, int> prev = {c[p[i-1]], c[(p[i-1] + (1 << h))
                % n]};
            if (cur != prev)
                ++classes;
            cn[p[i]] = classes - 1;
        }
        c.swap(cn);
    }
    return p;
}
vector<int> constructSuffixArray(string s) {
    s += "$"; // <- this must be smaller than any character in
        s
    vector<int> sorted_shifts = sortCyclicShifts(s);
    sorted_shifts.erase(sorted_shifts.begin());
    return sorted_shifts;
}
```

5.7 Manacher's Algorithm

```
vector<int> manacher_odd(string s) {
    int n = s.size();
    s = "$" + s + "^";
    vector<int> p(n + 2);
    int l = 0, r = -1;
    for(int i = 1; i <= n; i++) {
        p[i] = max(0, min(r - i, p[l + (r - i)]));
        while(s[i - p[i]] == s[i + p[i]]) {
            p[i]++;
        }
        if(i + p[i] > r) {
            l = i - p[i], r = i + p[i];
        }
    }
    return vector<int>(begin(p) + 1, end(p) - 1);
}

vector<int> manacher(string s) {
    string t;
    for(auto c: s) {
        t += string("#") + c;
    }
    auto res = manacher_odd(t + "#");
    return vector<int>(begin(res) + 1, end(res) - 1);
}
```

5.8 Hashing

```
struct HashedString {
    const ll A1 = 999999929, B1 = 1000000009, A2 =
        1000000087, B2 = 1000000097;
    vector<ll> A1pwrs, A2pwrs;
    vector<pll> prefixHash;
    HashedString(const string& _s) {
        init(_s);
        calcHashes(_s);
    }
    void init(const string& s) {
        ll a1 = 1;
        ll a2 = 1;
        FOR(i, 0, (int)s.length()+1) {
            A1pwrs.pb(a1);
            A2pwrs.pb(a2);
            a1 = (a1*A1)%B1;
            a2 = (a2*A2)%B2;
        }
    }
    void calcHashes(const string& s) {
        pll h = {0, 0};
        prefixHash.pb(h);
        for(char c : s) {
            ll h1 = (prefixHash.back().first*A1 + c)%B1;
            ll h2 = (prefixHash.back().second*A2 + c)%B2;
            prefixHash.pb({h1, h2});
        }
    }
    pll getHash(int l, int r) {
        ll h1 = (prefixHash[r+1].first - prefixHash[l].first*A1pwrs
            [r+1-l]) % B1;
        ll h2 = (prefixHash[r+1].second - prefixHash[l].second*
            A2pwrs[r+1-l]) % B2;
        if(h1 < 0) h1 += B1;
        if(h2 < 0) h2 += B2;
        return {h1, h2};
    }
};
```

6 Math

6.1 Factorization With Sieve

```
// Use linear sieve to calculate minDiv
vector<pll> factorize(ll x) {
    vector<pll> res;
```

```
    ll prev = -1;
    ll cnt = 0;
    while(x != 1) {
        ll d = minDiv[x];
        if(d == prev) {
            cnt++;
        } else {
            if(prev != -1) res.pb({prev, cnt});
            prev = d;
            cnt = 1;
        }
        x /= d;
    }
    res.pb({prev, cnt});
    return res;
}
```

6.2 Euler Totient Function

```
// Number of numbers x < n so that gcd(x, n) = 1
ll phi(ll n) {
    if(n == 1) return 1;
    auto f = factorize(n);
    ll res = n;
    for(auto p : f) {
        res = res - res/p.first;
    }
    return res;
}
```

6.3 Gaussian Elimination

```
// The last column of a is the right-hand side of the system.
// Returns 0, 1 or oo - the number of solutions.
// If at least one solution is found, it will be in ans
int gauss (vector < vector<ld> > a, vector<ld> & 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) {
                ld 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) {
        ld 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 oo;
    return 1;
}
```

6.4 Burnside's Lemma

Let G be a finite group that acts on a set X. For each g in G let X^g denote the set of elements in X that are fixed by g. Burnside's lemma asserts the following formula for the number of orbits:

$$|X/G| = \frac{1}{|G|}\sum_{g\in G}|X^g|.$$

Example. Coloring a cube with three colors.

Let X be the set of 3^6 possible face color combinations. Let's count the sizes of the fixed sets for each of the 24 rotations:

- one 0-degree rotation which leaves all 3^6 elements of X unchanged
- six 90-degree face rotations, each of which leaves 3^3 elements of X unchanged
- three 180-degree face rotation, each of which leaves 3^4 elements of X unchanged
- eight 120-degree vertex rotations, each of which leaves 3^2 elements of X unchanged
- six 180-degree edge rotations, each of which leaves 3^3 elements of X unchanged

The average is then $\frac{1}{24}(3^6 + 6\cdot 3^3 + 3\cdot 3^4 + 8\cdot 3^2 + 6\cdot 3^3) = 57$. For n colors: $\frac{1}{24}(n^6 + 3n^4 + 12n^3 + 8n^2)$.

Example. Coloring a circular stripe of n cells with two colors.

X is the set of all colored striped (it has 2^n elements), G is the group of rotations (n elements - by 0 cells, by 1 cell, ..., by $(n-1)$ cells). Let's fix some K and find the number of stripes that are fixed by the rotation by K cells. If a stripe becomes itself after rotation by K cells, then its 1st cell must have the same color as its $(1+K \mod n)$-th cell, which is in turn the same as its $(1+2K \mod n)$-th cell, etc., until $mK \mod n = 0$. This will happen when $m = n/gcd(K,n)$. Therefore, we have $n/gcd(K,n)$ cells that must all be of the same color. The same will happen when starting from the second cell and so on. Therefore, all cells are separated into $gcd(K,n)$ groups, with each group being of one color, and that yields $2^{gcd(K,n)}$ choices. That's why the answer to the original problem is $\frac{1}{n}\sum_{k=0}^{n-1} 2^{gcd(k,n)}$.

6.5 Modular Inverse

```
bool invWithEuclid(ll a, ll m, ll& aInv) {
    ll x, y, g;
    if(!solveEqNonNegX(a, m, 1, x, y, g)) return false;
    aInv = x;
    return true;
}
// Works only if m is prime
ll invFermat(ll a, ll m) {
    return pwr(a, m-2, m);
}
// Works only if gcd(a, m) = 1
ll invEuler(ll a, ll m) {
    return pwr(a, phi(m)-1, m);
}
```

6.6 Eratosthenes

```
const LL max_er = 1e7;
vll min_div(max_er + 1, 0);
vi er_used(max_er + 1, 1);
vll primes;
vector<pii> divs;

void eratosthenes()
{
	er_used[0] = er_used[1] = 0;
	for (LL i = 2; i <= max_er; ++i)
	{
		if (!er_used[i]) continue;
		primes.push_back(i);
		min_div[i] = i;
		for (LL j = i * i; j <= max_er; j += i)
		{
			er_used[j] = 0;
			if (!min_div[j]) min_div[j] = i;
		}
	}
}

void get_divs(LL n)
{
	while (n != 1)
	{
		LL cur = min_div[n];
		LL cnt = 0;
		while (n % cur == 0)
		{
			n /= cur;
			cnt++;
		}
		divs.push_back({ cur, cnt });
	}
}
```

6.7 Gcd

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

6.8 Sprague Grundy Theorem

We have a game which fulfills the following requirements:

- There are two players who move alternately.
- The game consists of states, and the possible moves in a state do not depend on whose turn it is.
- The game ends when a player cannot make a move.
- The game surely ends sooner or later.
- The players have complete information about the states and allowed moves, and there is no randomness in the game.

Grundy Numbers. The idea is to calculate Grundy numbers for each game state. It is calculated like so: $mex(\{g_1, g_2, ..., g_n\})$, where $g_1, g_2, ..., g_n$ are the Grundy numbers of the states which are reachable from the current state. mex gives the smallest nonnegative number that is not in the set ($mex(\{0, 1, 3\}) = 2, mex(\emptyset) = 0$). If the Grundy number of a state is 0, then this state is a losing state. Otherwise it's a winning state.

Grundy's Game. Sometimes a move in a game divides the game into subgames that are independent of each other. In this case, the Grundy number of a game state is $mex(\{g_1, g_2, ..., g_n\}), g_k = a_{k,1} \oplus a_{k,2} \oplus ... \oplus a_{k,m}$ meaning that move k divides the game into m subgames whose Grundy numbers are $a_{i,j}$.

Example. We have a heap with n sticks. On each turn, the player chooses a heap and divides it into two nonempty heaps such that the heaps are of different size. The player who makes the last move wins the game. Let $g(n)$ denote the Grundy number of a heap of size n. The Grundy number can be calculated by going though all possible ways to divide the heap into two parts. E.g. $g(8) = mex(\{g(1) \oplus g(7), g(2) \oplus g(6), g(3) \oplus g(5)\})$. Base case: $g(1) = g(2) = 0$, because these are losing states.

6.9 Formulas

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$; $\sum_{i=1}^{n} i^2 = \frac{n(2n+1)(n+1)}{6}$; $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$; $\sum_{i=1}^{n} i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$; $\sum_{i=a}^{b} c^i = \frac{c^{b+1}-c^a}{c-1}, c \neq 1$; $\sum_{i=1}^{n} a_1 + (i-1)d = \frac{n(a_1+a_n)}{2}$; $\sum_{i=1}^{n} a_1 r^{i-1} = \frac{a_1(1-r^n)}{1-r}, r \neq 1$; $\sum_{i=1}^{\infty} ar^{i-1} = \frac{a_1}{1-r}, |r| \leq 1$.

6.10 C

```
const int maxn = 2000;
LL C[maxn + 1][maxn + 1];
void comb(LL mod = LLONG_MAX)
{
	for (int nn = 0; nn <= maxn; ++nn) {
		C[nn][0] = C[nn][nn] = 1;
		for (int kk = 1; kk < nn; ++kk)
			C[nn][kk] = (C[nn - 1][kk - 1] + C[nn - 1][
				kk]) % mod;
	}
}
```

6.11 Simpson Integration

```
const int N = 1000 * 1000; // number of steps (already
    multiplied by 2)

double simpsonIntegration(double a, double b){
    double h = (b - a) / N;
    double s = f(a) + f(b); // a = x_0 and b = x_2n
    for (int i = 1; i <= N - 1; ++i) {
        double x = a + h * i;
        s += f(x) * ((i & 1) ? 4 : 2);
    }
    s *= h / 3;
    return s;
}
```

6.12 Matrix

```
class Matrix
{
public:
	int cols, rows;
```

```
    vvll data;
    int mod;
    Matrix(int _rows, int _cols, int _mod, bool ones = false
        )
    {
        rows = _rows; cols = _cols;
        mod = _mod;
        data.clear();
        data.resize(rows, vll(cols, 0));
        if (ones)
        {
            for (int i = 0; i < min(rows, cols); i++)
                data[i][i] = 1;
        }
    }

    vll& operator[] (int idx)
    {
        return data[idx];
    }

    Matrix operator*(Matrix& b)
    {
        Matrix res(rows, b.cols, mod);
        for (int i = 0; i < rows; ++i)
        {
            for (int j = 0; j < b.cols; ++j)
            {
                for (int k = 0; k < cols; ++k)
                {
                    res[i][j] += data[i][k] * b[k][j
                        ] % mod;
                }
                res[i][j] %= mod;
            }
        }
        return res;
    }

    Matrix operator%(int mod)
    {
        Matrix res = *this;
        for (int i = 0; i < rows; ++i)
            for (int j = 0; j < cols; ++j)
                res[i][j] %= mod;
        return res;
    }

    Matrix binpow(int nn) {
        if (nn == 0)
            return Matrix(rows, cols, mod, true);
        if (nn % 2 == 1)
            return (binpow(nn - 1) % mod * (*this))
                % mod;
        else {
            auto bb = binpow(nn / 2) % mod;
            return (bb * bb) % mod;
        }
    }
};
```

6.13 Linear Sieve

```
ll minDiv[MAXN+1];
vector<ll> primes;

void sieve(ll n){
    FOR(k, 2, n+1){
        minDiv[k] = k;
    }
    FOR(k, 2, n+1) {
        if(minDiv[k] == k) {
            primes.pb(k);
        }
        for(auto p : primes) {
            if(p > minDiv[k]) break;
            if(p*k > n) break;
            minDiv[p*k] = p;
        }
    }
}
```

6.14 Extended Euclidean Algorithm

```
// ax+by=gcd(a,b)
void solveEq(ll a, ll b, ll& x, ll& y, ll& g) {
    if(b==0) {
        x = 1;
        y = 0;
        g = a;
        return;
    }
    ll xx, yy;
    solveEq(b, a%b, xx, yy, g);
    x = yy;
    y = xx-yy*(a/b);
}
// ax+by=c
bool solveEq(ll a, ll b, ll c, ll& x, ll& y, ll& g) {
    solveEq(a, b, x, y, g);
    if(c%g != 0) return false;
    x *= c/g; y *= c/g;
    return true;
}
// Finds a solution (x, y) so that x >= 0 and x is minimal
bool solveEqNonNegX(ll a, ll b, ll c, ll& x, ll &y, ll& g) {
    if(!solveEq(a, b, c, x, y, g)) return false;
    ll k = x*g/b;
    x = x - k*b/g;
    y = y + k*a/g;
    if(x < 0) {
        x += b/g;
        y -= a/g;
    }
    return true;
}
```

6.15 Chinese Remainder Theorem

Let's say we have some numbers m_i, which are all mutually coprime. Also, let $M = \prod_i m_i$. Then the system of congruences

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \ldots \\ x \equiv a_k \pmod{m_k} \end{cases}$$

is equivalent to $x \equiv A \pmod{M}$ and there exists a unique number A satisfying $0 \leq A \leq M$.

Solution for two: $x \equiv a_1 \pmod{m_1}, x \equiv a_2 \pmod{m_2}$. Let $x = a_1 + km_1$. Substituting into the second congruence: $km_1 \equiv a_2 - a_1 \pmod{m_2}$. Then, $k = (m_1)^{-1}_{m_2}(a_2 - a_1) \pmod{m_2}$. and we can easily find x. This can be extended to multiple equations by solving them one-by-one.

If the moduli are not coprime, solve the system $y \equiv 0 \pmod{\frac{m_1}{g}}, y \equiv \frac{a_2-a_1}{g} \pmod{\frac{m_2}{g}}$ for y. Then let $x \equiv gy + a_1 \pmod{\frac{m_1 m_2}{g}}$. All other solutions can be found like this:

$$x' = x - k\frac{b}{g}, y' = y + k\frac{a}{g}, k \in \mathbb{Z}$$

6.16 Binpow

```
LL binpow(LL aa, LL nn, LL mod) {
    if (nn == 0)
        return 1ll;
    if (nn % 2 == 1)
        return (binpow(aa, nn - 1, mod) % mod * aa) %
            mod;
```

```
        else {
                LL bb = binpow(aa, nn / 2, mod) % mod;
                return (bb * bb) % mod;
        }
}
```

7 Dynamic Programming

7.1 Divide And Conquer

```
/*
Let A[i][j] be the optimal answer for using i objects to satisfy j
        first
requirements.
The recurrence is:
A[i][j] = min(A[i-1][k] + f(i, j, k)) where f is some function that
        denotes the
cost of satisfying requirements from k+1 to j using the i-th
        object.
Consider the recursive function calc(i, jmin, jmax, kmin, kmax),
        that calculates
all A[i][j] for all j in [jmin, jmax] and a given i using known A[i
        -1][*].
*/

void calc(int i, int jmin, int jmax, int kmin, int kmax) {
    if(jmin > jmax) return;
    int jmid = (jmin+jmax)/2;
    // calculate A[i][jmid] naively (for k in kmin...min(jmid,
        kmax){...})
    // let kmid be the optimal k in [kmin, kmax]
    calc(i, jmin, jmid-1, kmin, kmid);
    calc(i, jmid+1, jmax, kmid, kmax);
}

int main() {
    // set initial dp values
    FOR(i, start, k+1){
        calc(i, 0, n-1, 0, n-1);
    }
    cout << dp[k][n-1];
}
```

7.2 Convex Hull Trick

```
/*
Let's say we have a relation:
dp[i] = min(dp[j] + h[j+1]*w[i]) for j<=i
Let's set k_j = h[j+1], x = w[i], b_j = dp[j]. We get:
dp[i] = min(b_j+k_j*x) for j<=i.
This is the same as finding a minimum point on a set of lines.
After calculating the value, we add a new line with
k_i = h[i+1] and b_i = dp[i].
*/
struct Line {
    int k;
    int b;

    int eval(int x) {
        return k*x+b;
    }

    int intX(Line& other) {
        int x = b-other.b;
        int y = other.k-k;
        int res = x/y;
        if(x%y != 0) res++;
        return res;
    }
};

struct BagOfLines {
    vector<pair<Line, int>> lines;

    void addLine(int k, int b) {
        Line current = {k, b};
        if(lines.empty()) {
            lines.pb({current, -OO});
            return;
        }
        int x = -OO;
        while(true) {
            auto line = lines.back().first;
            int from = lines.back().second;
            x = line.intX(current);
            if(x > from) break;
            lines.pop_back();
        }
        lines.pb({current, x});
    }

    int findMin(int x) {
        int lo = 0, hi = (int)lines.size()-1;
        while(lo < hi) {
            int mid = (lo+hi+1)/2;
            if(lines[mid].second <= x) {
                lo = mid;
            } else {
                hi = mid-1;
            }
        }
        return lines[lo].first.eval(x);
    }
};
```

7.3 Optimizations

1. Convex Hull 1:
   - Recurrence: $dp[i] = \min_{j<i}\{dp[j] + b[j] \cdot a[i]\}$
   - Condition: $b[j] \geq b[j+1], a[i] \leq a[i+1]$
   - Complexity: $\mathcal{O}(n^2) \to \mathcal{O}(n)$
2. Convex Hull 2:
   - Recurrence: $dp[i][j] = \min_{k<j}\{dp[i-1][k] + b[k] \cdot a[j]\}$
   - Condition: $b[k] \geq b[k+1], a[j] \leq a[j+1]$
   - Complexity: $\mathcal{O}(kn^2) \to \mathcal{O}(kn)$
3. Divide and Conquer:
   - Recurrence: $dp[i][j] = \min_{k<j}\{dp[i-1][k] + C[k][j]\}$
   - Condition: $A[i][j] \leq A[i][j+1]$
   - Complexity: $\mathcal{O}(kn^2) \to \mathcal{O}(kn\log(n))$
4. Knuth:
   - Recurrence: $dp[i][j] = \min_{i<k<j}\{dp[i][k] + dp[k][j]\} + C[i][j]$
   - Condition: $A[i][j-1] \leq A[i][j] \leq A[i+1][j]$
   - Complexity: $\mathcal{O}(n^3) \to \mathcal{O}(n^2)$

Notes:

- $A[i][j]$ - the smallest k that gives the optimal answer
- $C[i][j]$ - some given cost function

8 Misc

8.1 Builtin GCC Stuff

- __builtin_clz(x): the number of zeros at the beginning of the bit representation.
- __builtin_ctz(x): the number of zeros at the end of the bit representation.
- __builtin_popcount(x): the number of ones in the bit representation.
- __builtin_parity(x): the parity of the number of ones in the bit representation.

- __gcd(x, y): the greatest common divisor of two numbers.
- __int128_t: the 128-bit integer type. Does not support input/output.

8.2 Mo's Algorithm

Mo's algorithm processes a set of range queries on a static array. Each query is to calculate something base on the array values in a range $[a, b]$. The queries have to be known in advance. Let's divide the array into blocks of size $k = O(\sqrt{n})$. A query $[a_1, b_1]$ is processed before query $[a_2, b_2]$ if $\lfloor \frac{a_1}{k} \rfloor < \lfloor \frac{a_2}{k} \rfloor$ or $\lfloor \frac{a_1}{k} \rfloor = \lfloor \frac{a_2}{k} \rfloor$ and $b_1 < b_2$.

Example problem: counting number of distinct values in a range. We can process the queries in the described order and keep an array count, which knows how many times a certain value has appeared. When moving the boundaries back and forth, we either increase count$[x_i]$ or decrease it. According to value of it, we will know how the number of distinct values has changed (e.g. if count$[x_i]$ has just become 1, then we add 1 to the answer, etc.).

8.3 Ternary Search

```
double ternary_search(double l, double r) {
    while (r - l > eps) {
        double m1 = l + (r - l) / 3;
        double m2 = r - (r - l) / 3;
        double f1 = f(m1);
        double f2 = f(m2);
        if (f1 < f2)
            l = m1;
        else
            r = m2;
    }
    return f(l); //return the maximum of f(x) in [l, r]
}
```