Tit For Tat

icorx0, nhrot, vinnie26

November 8, 2025

Contents

1 Base

1.1 Template

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

using i64 = long long;
using u64 = unsigned long long;
using i128 = __int128_t;
using u128 = __uint128_t;
using ld = long double;

constexpr int INF32 = 0x3f3f3f3f; // ~1e9
constexpr i64 INF64 = (i64)4e18;
constexpr ld EPS = 1e-12L;
constexpr ld PI = 3.14159265358979323846264338327950288L;
constexpr int MOD = 1e9 + 7;

#define fastio                                                  \
  ios::sync_with_stdio(false);                                  \
  cin.tie(nullptr);                                             \
  cout.tie(nullptr);

// ---------- optional ----------
template <class T> using V = vector<T>;
template <class K, class Val> using umap = unordered_map<K, Val>;
template <class K> using uset = unordered_set<K>;

#define rep(i, n) for (int i = 0; i < n; ++i)
#define per(i, n) for (int i = n - 1; i >= 0; --i)
#define reps(i, a, b) for (int i = a; i < b; ++i)
#define pers(i, a, b) for (int i = b - 1; i >= a; --i)
#define all(x) begin(x), end(x)
#define rall(x) rbegin(x), rend(x)
#define len(x) (x.size())

template <class T> using MinHeap = priority_queue<T, vector<T>, greater<T>>;
template <class T> using MaxHeap = priority_queue<T>;
// ---------- optional ----------

void solve() {}

int main() {
  fastio;
  int T = 1;
  // cin >> T;
  while (T--) solve();
  return 0;
}
```

Listing 1: template.cpp

1.2 Tester

```bash
#!/bin/bash
# Uso: ./tester A.cpp  -> buscará in/A.in, in/A1.in, etc.

SRC=$1
BIN="${SRC%.*}"

BASENAME=$(basename "$BIN")
mkdir -p out
g++ -std=c++17 -O2 -Wall -DLOCAL -fsanitize=address,undefined "$SRC" -o "$BIN" || exit 1
shopt -s nullglob
for IN in in/"${BASENAME}"*.in; do
    TEST_FILE=$(basename "$IN")
    TEST_NAME="${TEST_FILE%.in}"
    echo -e "\nTesting $IN..."
    /usr/bin/time -v "./$BIN" < "$IN" > "out/$TEST_NAME.out" 2>&1 || {
```

```
        time "./$BIN" < "$IN" > "out/$TEST_NAME.out"
    }
    echo "--- INPUT ($IN) ---"
    head -n 5 "$IN"
    echo -e "\n--- OUTPUT (out/$TEST_NAME.out) ---"
    head -n 10 "out/$TEST_NAME.out"
    echo "========================================"
done
```

Listing 2: tester.sh

2 Data Structures

2.1 DSU

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

struct DSU {
    vector<int> parent, size;
    DSU(int n) {
        parent.resize(n);
        size.resize(n);
    }
    void make_set(int v) {
        parent[v] = v;
        size[v] = 1;
    }
    int find_set(int v) {
        if(v == parent[v]) return v;
        return parent[v] = find_set(parent[v]);
    }
    void union_sets(int a, int b) {
        a = find_set(a);
        b = find_set(b);
        if(a == b) return;
        if(size[a] < size[b]) swap(a, b);
        parent[b] = a;
        size[a] += size[b];
    }
};
```

Listing 3: DSU.cpp

2.2 MergeSortTree

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

struct MergeSortTree {
    vector<vector<int>> heap;
    // s.build(array, 1, 0, n-1);
    void build(vector<int> &array, int v, int tl, int tr) {
        if(heap.size() <= v) heap.resize(v+1);
        if(tl == tr) heap[v] = vector<int> (1, array[tl]);
        else {
            int tm = (tl + tr) / 2;
            build(array, 2*v, tl, tm);
            build(array, 2*v+1, tm+1, tr);
            merge(heap[2*v].begin(), heap[2*v].end(), heap[2*v+1].begin(),
                heap[2*v+1].end(), back_inserter(heap[v]));
        }
    }
    // s.get(1, 0, n-1, l, r);
    vector<int> get(int v, int tl, int tr, int l, int r) {
        if(l > r) return vector<int> ();
        if(l == tl and r == tr) return heap[v];
        int tm = (tl + tr) / 2;
```

```
        vector<int> v1 = get(v*2, tl, tm, l, min(r, tm)), v2 = get(v*2+1, tm+1, tr, max(l, tm+1), r), result;
        merge(v1.begin(), v1.end(), v2.begin(), v2.end(), back_inserter(result));
        return result;
    }
    // s.update(1, 0, n-1, pos, x);
    void update(int v, int tl, int tr, int pos, int x) {
        if(tl == tr) {
            heap[v] = vector<int> (1, x);
        } else {
            int tm = (tl + tr) / 2;
            if(pos <= tm) update(2*v, tl, tm, pos, x);
            else update(2*v+1, tm+1, tr, pos, x);
            heap[v] = vector<int> ();
            merge(heap[2*v].begin(), heap[2*v].end(), heap[2*v+1].begin(),
                heap[2*v+1].end(), back_inserter(heap[v]));
        }
    }
};
```

Listing 4: MergeSortTree.cpp

2.3 SegmentTree

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

struct SegmentTree {
    vector<int> heap;
    vector<long long> lazy;

    // s.init(n);
    void init(int n) {
        int size = 4 * n + 1;
        heap.assign(size, 0);
        lazy.assign(size, 0);
    }

    // s.build(array, 1, 0, n-1);
    void build(vector<int> &array, int v, int tl, int tr) {
        if(tl == tr) heap[v] = array[tl];
        else {
            int tm = (tl + tr) / 2;
            build(array, 2*v, tl, tm);
            build(array, 2*v+1, tm+1, tr);
            heap[v] = heap[2*v] + heap[2*v+1];
        }
    }

    // s.sum(1, 0, n-1, l, r);
    int sum(int v, int tl, int tr, int l, int r) {
        if(l > r) return 0;
        if(l == tl and r == tr) return heap[v];
        int tm = (tl + tr) / 2;
        return sum(v*2, tl, tm, l, min(r, tm)) + sum(v*2+1, tm+1, tr, max(l, tm+1), r);
    }

    // s.update(1, 0, n-1, pos, x); in position pos set value x
    void update(int v, int tl, int tr, int pos, int x) {
        if(tl == tr) heap[v] = x;
        else {
            int tm = (tl + tr) / 2;
            if(pos <= tm) update(2*v, tl, tm, pos, x);
            else update(2*v+1, tm+1, tr, pos, x);
            heap[v] = heap[2*v] + heap[2*v+1];
        }
    }

    // push for lazy propagation
    void push(int v, int tl, int tr) {
        if (lazy[v] != 0 && tl != tr) {
```

```cpp
            int tm = (tl + tr) / 2;
            long long addval = lazy[v];
            heap[2*v] += addval * (tm - tl + 1);
            heap[2*v+1] += addval * (tr - tm);
            lazy[2*v] += addval;
            lazy[2*v+1] += addval;
        }
        lazy[v] = 0;
    }

    // s.lazyupdate(1, 0, n-1, l, r, addend); // add 'addval' to range [l, r]
    void lazyupdate(int v, int tl, int tr, int l, int r, int addval) {
        push(v, tl, tr); // Propagate any pending updates
        if (l > r) return;
        if (l > tr || r < tl) return;
        if (l <= tl && tr <= r) {
            // Apply update to the current heap value
            heap[v] += (long long)addval * (tr - tl + 1);
            // Mark the lazy tag for future children updates
            lazy[v] += addval;
            return;
        }

        int tm = (tl + tr) / 2;
        lazyupdate(2*v, tl, tm, l, r, addval);
        lazyupdate(2*v+1, tm+1, tr, l, r, addval);
        heap[v] = heap[2*v] + heap[2*v+1];
    }
};
```

Listing 5: SegmentTree.cpp

2.4 SparseTable

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

struct SparseTable {
    int N, K;
    vector<vector<int>> table;
    vector<int> logs;

    // Constructor to build the table
    SparseTable(const vector<int>& arr) {
        N = arr.size();
        K = floor(log2(N)) + 1; // Max power of 2
        table.assign(N, vector<int>(K));
        logs.resize(N + 1);

        logs[1] = 0;
        for (int i = 2; i <= N; i++) {
            logs[i] = logs[i / 2] + 1;
        }

        for (int i = 0; i < N; i++) {
            table[i][0] = arr[i];
        }

        for (int j = 1; j < K; j++) {
            for (int i = 0; i + (1 << j) <= N; i++) {
                table[i][j] = min(table[i][j - 1],
                                  table[i + (1 << (j - 1))][j - 1]);
            }
        }
    }

    // Query the range [L, R] (inclusive) in O(1)
    int query(int L, int R) {
        // Find largest k such that 2^k <= (R - L + 1)
        int k = logs[R - L + 1];

```

```
        // Return min of the two overlapping ranges
        return min(table[L][k], table[R - (1 << k) + 1][k]);
    }
};
```

Listing 6: SparseTable.cpp

2.5 Fenwick (BIT)

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

struct FenwickTree {
    vector<int> bit;  // binary indexed tree
    int n;

    FenwickTree(int n) {
        this->n = n;
        bit.assign(n, 0);
    }

    FenwickTree(vector<int> const &a) : FenwickTree(a.size()) {
        for (size_t i = 0; i < a.size(); i++)
            add(i, a[i]);
    }

    ll sum(int r) {
        ll ret = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret += bit[r];
        return ret;
    }

    ll sum(int l, int r) {
        return sum(r) - (l == 0 ? 0LL : sum(l - 1));
    }

    ll add(int idx, int delta) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] += delta;
    }
};

// Gemini implementation
struct RangeFenwickTree {
    FenwickTree B1;
    FenwickTree B2;
    int n;

    RangeFenwickTree(int size) : n(size), B1(size), B2(size) {}

    ll prefix_sum(int idx) {
        return B1.sum(idx) * (idx + 1) - B2.sum(idx);
    }

    ll range_sum(int l, int r) {
        ll sum_r = prefix_sum(r);
        ll sum_l_minus_1 = (l == 0) ? 0LL : prefix_sum(l - 1);
        return sum_r - sum_l_minus_1;
    }

    void range_add(int l, int r, ll x) {
        B1.add(l, x);
        if (r + 1 < n) {
            B1.add(r + 1, -x);
        }

        B2.add(l, x * l);
        if (r + 1 < n) {
            B2.add(r + 1, -x * (r + 1));
```

```
        }
    }
};
```

Listing 7: Fenwick.cpp

3 Trees

3.1 Heavy-Light Decomposition (HLD)

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

vector<int> parent, depth, heavy, head, pos;
int cur_pos;

int dfs(int v, vector<vector<int>> const& adj) {
    int size = 1;
    int max_c_size = 0;
    for (int c : adj[v]) {
        if (c != parent[v]) {
            parent[c] = v, depth[c] = depth[v] + 1;
            int c_size = dfs(c, adj);
            size += c_size;
            if (c_size > max_c_size)
                max_c_size = c_size, heavy[v] = c;
        }
    }
    return size;
}

void decompose(int v, int h, vector<vector<int>> const& adj) {
    head[v] = h, pos[v] = cur_pos++;
    if (heavy[v] != -1)
        decompose(heavy[v], h, adj);
    for (int c : adj[v]) {
        if (c != parent[v] && c != heavy[v])
            decompose(c, c, adj);
    }
}

void init(vector<vector<int>> const& adj) {
    int n = adj.size();
    parent = vector<int>(n);
    depth = vector<int>(n);
    heavy = vector<int>(n, -1);
    head = vector<int>(n);
    pos = vector<int>(n);
    cur_pos = 0;

    dfs(0, adj);
    decompose(0, 0, adj);
}

// how queries should be implemented
int segment_tree_query(int a, int b);
int query(int a, int b) {
    int res = 0;
    for (; head[a] != head[b]; b = parent[head[b]]) {
        if (depth[head[a]] > depth[head[b]])
            swap(a, b);
        int cur_heavy_path_max = segment_tree_query(pos[head[b]], pos[b]);
        res = max(res, cur_heavy_path_max);
    }
    if (depth[a] > depth[b])
        swap(a, b);
    int last_heavy_path_max = segment_tree_query(pos[a], pos[b]);
    res = max(res, last_heavy_path_max);
    return res;
}
```

Listing 8: HLD.cpp

3.2 Centroid Decomposition

```cpp
#include <bits/stdc++.h>
using namespace std;
// A centroid of a tree is defined as a node such that when the tree is rooted at it, no other
    nodes have a subtree of size greater than  $\frac{N}{2}$

const int maxn = 200010;

int n;
vector<int> adj[maxn];
int subtree_size[maxn];

// must be called first at 0 to fill subtree_size
int get_subtree_size(int node, int parent = -1) {
  int &res = subtree_size[node];
  res = 1;
  for (int i : adj[node]) {
    if (i == parent) { continue; }
    res += get_subtree_size(i, node);
  }
  return res;
}

int get_centroid(int node, int parent = -1) {
  for (int i : adj[node]) {
    if (i == parent) { continue; }

    if (subtree_size[i] * 2 > n) { return get_centroid(i, node); }
  }
  return node;
}
```

Listing 9: CentroidDecomposition.cpp

3.3 Lowest Common Ancestor (LCA) - Binary Lifting

```cpp
#include <bits/stdc++.h>
using namespace std;
int n, l;
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;
    for (int i = 1; i <= l; ++i)
        up[v][i] = up[up[v][i-1]][i-1];

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

    tout[v] = ++timer;
}

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

int lca(int u, int v) {
```

```cpp
    if (is_ancestor(u, v))
        return u;
    if (is_ancestor(v, u))
        return v;
    for (int i = l; i >= 0; --i) {
        if (!is_ancestor(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);
}
```

Listing 10: LCA.cpp

4 Graphs

4.1 Dijkstra

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

const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;
int n_vertices;

void dijkstra(int source, vector<int> &distances, int current_time) {
    distances.assign(n_vertices, INF);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> tour;

    distances[source] = current_time;
    tour.push({distances[source], source});

    while(!tour.empty()) {
        auto u = tour.top();
        tour.pop();

        if (distances[u.second] != u.first) continue;
        for (pair<int, int> &v: adj[u.second]) {
            int d = u.first + v.second;
            if (d < distances[v.first]) {
                distances[v.first] = d;
                tour.push({d, v.first});
            }
        }
    }
}
```

Listing 11: Dijkstra.cpp

4.2 Floyd-Warshall

```cpp
#include <bits/stdc++.h>
using i64 = long long;
using namespace std;
const i64 INF = 1e18;

void floydWarshall(vector<vector<i64>>& d, int N) {
    d.resize(N, vector<i64>(N, INF)); // all paths inf by default
    for (int k = 0; k < N; ++k) {
        for (int i = 0; i < N; ++i) {
```

```cpp
            for (int j = 0; j < N; ++j) {
                if (d[i][k] < INF && d[k][j] < INF)
                    d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
            }
        }
    }
}
```

Listing 12: FloydWarshall.cpp

4.3 Kruskal

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

struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};
int n;
vector<Edge> edges;

vector<Edge> kruskal() {
    int cost = 0;
    vector<int> tree_id(n);
    vector<Edge> result;

    for (int i = 0; i < n; i++)
        tree_id[i] = i;

    sort(edges.begin(), edges.end());

    for (Edge e : edges) {
        if (tree_id[e.u] != tree_id[e.v]) {
            cost += e.weight;
            result.push_back(e);

            int old_id = tree_id[e.u], new_id = tree_id[e.v];
            for (int i = 0; i < n; i++) {
                if (tree_id[i] == old_id)
                    tree_id[i] = new_id;
            }
        }
    }
    return result;
}
```

Listing 13: Kruskal.cpp

4.4 Max Flow (Dinic)

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

struct FlowEdge {
    int v, u;
    long long cap, flow = 0;
    FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
};

struct Dinic {
    const long long flow_inf = 1e18;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    int n, m = 0;
    int s, t;
    vector<int> level, ptr;
```

```cpp
    queue<int> q;

    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        level.resize(n);
        ptr.resize(n);
    }

    void add_edge(int v, int u, long long cap) {
        edges.emplace_back(v, u, cap);
        edges.emplace_back(u, v, 0);
        adj[v].push_back(m);
        adj[u].push_back(m + 1);
        m += 2;
    }

    bool bfs() {
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int id : adj[v]) {
                if (edges[id].cap == edges[id].flow)
                    continue;
                if (level[edges[id].u] != -1)
                    continue;
                level[edges[id].u] = level[v] + 1;
                q.push(edges[id].u);
            }
        }
        return level[t] != -1;
    }

    long long dfs(int v, long long pushed) {
        if (pushed == 0)
            return 0;
        if (v == t)
            return pushed;
        for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
            int id = adj[v][cid];
            int u = edges[id].u;
            if (level[v] + 1 != level[u])
                continue;
            long long tr = dfs(u, min(pushed, edges[id].cap - edges[id].flow));
            if (tr == 0)
                continue;
            edges[id].flow += tr;
            edges[id ^ 1].flow -= tr;
            return tr;
        }
        return 0;
    }

    long long flow() {
        long long f = 0;
        while (true) {
            fill(level.begin(), level.end(), -1);
            level[s] = 0;
            q.push(s);
            if (!bfs())
                break;
            fill(ptr.begin(), ptr.end(), 0);
            while (long long pushed = dfs(s, flow_inf)) {
                f += pushed;
            }
        }
        return f;
    }
};
```

Listing 14: Dinic.cpp

4.5 Topological Sort (Kahn's Algorithm)

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

int n; // number of vertices
vector<vector<int>> adj; // adjacency list of graph
vector<bool> visited;
vector<int> ans;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u]) {
            dfs(u);
        }
    }
    ans.push_back(v);
}

void topological_sort() {
    visited.assign(n, false);
    ans.clear();
    for (int i = 0; i < n; ++i) {
        if (!visited[i]) {
            dfs(i);
        }
    }
    reverse(ans.begin(), ans.end());
}
```

Listing 15: TopoSort.cpp

5 Mathematics and Number Theory

5.1 Sieve

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

const int N = 10000000;
vector<int> lp(N+1);
vector<int> pr;

void sieve_init() {
    for (int i=2; i <= N; ++i) {
        if (lp[i] == 0) {
            lp[i] = i;
            pr.push_back(i);
        }
        for (int j = 0; i * pr[j] <= N; ++j) {
            lp[i * pr[j]] = pr[j];
            if (pr[j] == lp[i]) {
                break;
            }
        }
    }
}

// Get all primes in range [L, R] with R up to 10e12 and R-L up to 10e7
vector<char> segmentedSieve(long long L, long long R) {
    // generate all primes up to sqrt(R)
    long long lim = sqrt(R);
    vector<char> mark(lim + 1, false);
    vector<long long> primes;
    for (long long i = 2; i <= lim; ++i) {
        if (!mark[i]) {
            primes.emplace_back(i);
            for (long long j = i * i; j <= lim; j += i)
                mark[j] = true;
```

```cpp
        }
    }

    vector<char> isPrime(R - L + 1, true);
    for (long long i : primes)
        for (long long j = max(i * i, (L + i - 1) / i * i); j <= R; j += i)
            isPrime[j - L] = false;
    if (L == 1)
        isPrime[0] = false;
    return isPrime;
}
```

Listing 16: Sieve.cpp

5.2 Extended Euclidean Algorithm

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

int gcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}

bool find_any_solution(int a, int b, int c, int &x0, int &y0, int &g) {
    g = gcd(abs(a), abs(b), x0, y0);
    if (c % g) {
        return false;
    }

    x0 *= c / g;
    y0 *= c / g;
    if (a < 0) x0 = -x0;
    if (b < 0) y0 = -y0;
    return true;
}

void shift_solution(int & x, int & y, int a, int b, int cnt) {
    x += cnt * b;
    y -= cnt * a;
}

int find_all_solutions(int a, int b, int c, int minx, int maxx, int miny, int maxy) {
    int x, y, g;
    if (!find_any_solution(a, b, c, x, y, g))
        return 0;
    a /= g;
    b /= g;

    int sign_a = a > 0 ? +1 : -1;
    int sign_b = b > 0 ? +1 : -1;

    shift_solution(x, y, a, b, (minx - x) / b);
    if (x < minx)
        shift_solution(x, y, a, b, sign_b);
    if (x > maxx)
        return 0;
    int lx1 = x;

    shift_solution(x, y, a, b, (maxx - x) / b);
    if (x > maxx)
        shift_solution(x, y, a, b, -sign_b);
    int rx1 = x;
```

```cpp

    shift_solution(x, y, a, b, -(miny - y) / a);
    if (y < miny)
        shift_solution(x, y, a, b, -sign_a);
    if (y > maxy)
        return 0;
    int lx2 = x;

    shift_solution(x, y, a, b, -(maxy - y) / a);
    if (y > maxy)
        shift_solution(x, y, a, b, sign_a);
    int rx2 = x;

    if (lx2 > rx2)
        swap(lx2, rx2);
    int lx = max(lx1, lx2);
    int rx = min(rx1, rx2);

    if (lx > rx)
        return 0;
    return (rx - lx) / abs(b) + 1;
}
```

Listing 17: ExtendedEuclid.cpp

5.3 Binomial Coefficients (nCr) and Modular Arithmetic

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

using i64 = long long;
using u64 = unsigned long long;
using i128 = __int128_t;
using u128 = __uint128_t;
using ld = long double;

constexpr int MOD = 1e9 + 7;

inline i64 addmod(i64 a, i64 b, i64 mod = MOD) {
  a %= mod;
  b %= mod;
  a += b;
  if (a >= mod)
    a -= mod;
  return a;
}

inline i64 submod(i64 a, i64 b, i64 mod = MOD) {
  a %= mod;
  b %= mod;
  a -= b;
  if (a < 0)
    a += mod;
  return a;
}

inline i64 mulmod(i64 a, i64 b, i64 mod = MOD) {
  return (i64)((i128)a * b % mod);
}

i64 binpow(i64 a, i64 e, i64 mod = MOD) {
  i64 r = 1 % mod;
  a %= mod;
  while (e) {
    if (e & 1)
      r = mulmod(r, a, mod);
    a = mulmod(a, a, mod);
    e >>= 1;
  }
  return r;
}
```

```
inline i64 lcm_ll(i64 a, i64 b) {
  if (a == 0 || b == 0)
    return 0;
  return a / std::gcd(a, b) * b;
}

// Devuelve el GCD de a y b, y x, y tales que ax + by = GCD(a, b)
i64 egcd(i64 a, i64 b, i64 &x, i64 &y) {
  if (!b) {
    x = 1;
    y = 0;
    return a;
  }
  i64 x1, y1;
  i64 g = egcd(b, a % b, x1, y1);
  x = y1;
  y = x1 - (a / b) * y1;
  return g;
}

// Inverso modular: usar Fermat si mod es primo, si no, usar egcd
inline i64 invmod(i64 a, i64 mod = MOD) {
  if (std::gcd(a, mod) != 1) {
    return -1; // no existe
  }
  // Fermat si MOD es primo
  // return binpow((a % mod + mod) % mod, mod - 2, mod);
  i64 x, y;
  egcd(a, mod, x, y);
  x %= mod;
  if (x < 0)
    x += mod;
  return x;
}

inline i64 moddiv(i64 a, i64 b, i64 mod = MOD) {
  i64 inv = invmod(b, mod);
  return inv == -1 ? -1 : mulmod((a % mod + mod) % mod, inv, mod);
}

// -------------------- Binomial coefficients --------------------
// Exact binomial coefficient (integer). For large n this may overflow i64.
inline i64 binom_exact(i64 n, i64 k) {
  if (k < 0 || k > n)
    return 0;
  k = min(k, n - k);
  i64 res = 1;
  // Compute iteratively: res = C(n, i) for i from 1..k
  for (i64 i = 1; i <= k; ++i) {
    // Multiply then divide; this stays integral at every step because
    // res = C(n, i-1) and res * (n - i + 1) / i = C(n, i).
    res = res * (n - i + 1) / i;
  }
  return res;
}

// Modular binomial coefficients using precomputed factorials.
// Usage: call init_fact(MAX_N) once (MAX_N >= maximum n you'll query).
static vector<i64> fact_mod, invfact_mod;

inline void init_fact(int N, i64 mod = MOD) {
  fact_mod.assign(N + 1, 1);
  invfact_mod.assign(N + 1, 1);
  for (int i = 1; i <= N; ++i)
    fact_mod[i] = mulmod(fact_mod[i - 1], i, mod);
  // invfact[N] = inverse of fact[N]
  invfact_mod[N] = invmod(fact_mod[N], mod);
  // compute invfact downwards
  for (int i = N; i > 0; --i)
    invfact_mod[i - 1] = mulmod(invfact_mod[i], i, mod);
}

```

```
inline i64 nCr_mod(int n, int r, i64 mod = MOD) {
  if (r < 0 || r > n)
    return 0;
  if ((int)fact_mod.size() <= n) {
    // Not initialized for this n; fallback to multiplicative method modulo (works if mod is
    prime)
    // This multiplicative method requires mod to be prime for modular division.
    i64 num = 1, den = 1;
    r = min(r, n - r);
    for (int i = 1; i <= r; ++i) {
      num = mulmod(num, n - r + i, mod);
      den = mulmod(den, i, mod);
    }
    i64 inv = invmod(den, mod);
    return mulmod(num, inv, mod);
  }
  return mulmod(fact_mod[n], mulmod(invfact_mod[r], invfact_mod[n - r], mod), mod);
}

// -------------------- Catalan numbers --------------------
// Exact Catalan number: C_n = binom(2n, n) / (n+1). May overflow for large n.
inline i64 catalan_exact(i64 n) {
  if (n < 0)
    return 0;
  return binom_exact(2 * n, n) / (n + 1);
}

// Catalan modulo MOD (MOD should be prime for invmod to work reliably)
inline i64 catalan_mod(int n, i64 mod = MOD) {
  if (n < 0)
    return 0;
  i64 c = nCr_mod(2 * n, n, mod);
  i64 inv = invmod(n + 1, mod);
  if (inv == -1) // no inverse
    return -1;
  return mulmod(c, inv, mod);
}
```

Listing 18: BinomialCoefficients.cpp

6 Strings

6.1 Hashing

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

using i64 = long long;
using u64 = unsigned long long;
constexpr int MOD = 1e9 + 7;

struct StringHash {
    vector<u64> hashed, pwrs;
    u64 MOD_VAL;
    u64 BASE;

    StringHash(const string &s, u64 base = 257, u64 mod = MOD)
        : MOD_VAL(mod), BASE(base) {
        int n = s.length();
        hashed.resize(n + 1, 0);
        pwrs.resize(n + 1, 0);

        pwrs[0] = 1;
        for (int i = 1; i <= n; i++) {
            pwrs[i] = (pwrs[i - 1] * BASE) % MOD_VAL;
        }

        for (int i = 0; i < n; i++) {
            hashed[i + 1] = (hashed[i] * BASE + s[i]) % MOD_VAL;
        }
```

```cpp
    }

    u64 getHash(int l, int r) {
        u64 hash = (hashed[r + 1] -(hashed[l] * pwrs[r - l + 1] % MOD_VAL) + MOD_VAL) % MOD_VAL;
        return hash;
    }
};

struct DoubleHash {
    StringHash hash1, hash2;

    DoubleHash(const string &s)
        : hash1(s, 257, 1e9 + 7), hash2(s, 353, 1e9 + 9) {}

    bool areEqual(int l1, int r1, int l2, int r2) {
        if (r1 - l1 != r2 - l2) return false;
        return hash1.getHash(l1, r1) == hash2.getHash(l2, r2) && hash1.getHash(l1, r1) == hash2.getHash(l2, r2);
    }

    bool areEqualStrict(int l1, int r1, int l2, int r2) {
        if (r1 - l1 != r2 - l2) return false;
        return hash1.getHash(l1, r1) == hash1.getHash(l2, r2) && hash2.getHash(l1, r1) == hash2.getHash(l2, r2);
    }
};
```

Listing 19: Hashing.cpp

6.2 KMP

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

vector<int> prefix_function(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;
}

// Busqueda de patron P en texto T
vector<int> kmp_search(const string &P, const string &T) {
    string s = P + "#" + T;
    vector<int> pi = prefix_function(s);
    vector<int> matches;
    int m = P.length();
    for (int i = m + 1; i < s.length(); i++) {
        if (pi[i] == m) {
            matches.push_back(i - 2 * m); // posicion en T
        }
    }
    return matches;
}
```

Listing 20: KMP.cpp

6.3 Manacher

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

```cpp

// manachers finds all palindromic substrings in O(n) time
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] = 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);
}
```

Listing 21: Manacher.cpp

6.4 Aho-Corasick - Trie

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

const int K = 26;

struct Vertex {
    int next[K];
    bool output = false;
    int p = -1;
    char pch;
    int link = -1;
    int go[K];

    Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
        fill(begin(next), end(next), -1);
        fill(begin(go), end(go), -1);
    }
};

vector<Vertex> t(1);

void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (t[v].next[c] == -1) {
            t[v].next[c] = t.size();
            t.emplace_back(v, ch);
        }
        v = t[v].next[c];
    }
    t[v].output = true;
}

int go(int v, char ch);

int get_link(int v) {
    if (t[v].link == -1) {
        if (v == 0 || t[v].p == 0)
```

```
            t[v].link = 0;
        else
            t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
}

int go(int v, char ch) {
    int c = ch - 'a';
    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(get_link(v), ch);
    }
    return t[v].go[c];
}
```

Listing 22: AhoCorasick.cpp

6.5 Suffix Array + LCP Array

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

vector<int> sort_cyclic_shifts(string const& s) {
    int n = s.size();
    const int alphabet = 256;

    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> suffix_array_construction(string s) {
    s += "$";
    vector<int> sorted_shifts = sort_cyclic_shifts(s);
    sorted_shifts.erase(sorted_shifts.begin());
    return sorted_shifts;
}

vector<int> lcp_construction(string const& s, vector<int> const& p) {
    int n = s.size();
    vector<int> rank(n, 0);
    for (int i = 0; i < n; i++)
        rank[p[i]] = i;

    int k = 0;
    vector<int> lcp(n-1, 0);
    for (int i = 0; i < n; i++) {
        if (rank[i] == n - 1) {
            k = 0;
            continue;
        }
        int j = p[rank[i] + 1];
        while (i + k < n && j + k < n && s[i+k] == s[j+k])
            k++;
        lcp[rank[i]] = k;
        if (k)
            k--;
    }
    return lcp;
}
```

Listing 23: SuffixArray.cpp

7 Geometry

7.1 Convex Hull

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

struct pt {

    double x, y;
    bool operator == (pt const& t) const {
        return x == t.x && y == t.y;
    }
};

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

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

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

```cpp
        return o < 0;
    });
    if (include_collinear) {
        int i = (int)a.size()-1;
        while (i >= 0 && collinear(p0, a[i], a.back())) i--;
        reverse(a.begin()+i+1, a.end());
    }

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

    if (include_collinear == false && st.size() == 2 && st[0] == st[1])
        st.pop_back();

    a = st;
}
```

Listing 24: ConvexHull.cpp

7.2 Closest Pair

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

using i64 = long long;
using ld = long double;


struct pt {
    i64 x, y;
    pt() {}
    pt(i64 x_, i64 y_) : x(x_), y(y_) {}
    void read() {
        cin >> x >> y;
    }
};

bool operator==(const pt& a, const pt& b) {
    return a.x == b.x and a.y == b.y;
}


struct CustomHashPoint {
    size_t operator()(const pt& p) const {
        static const uint64_t C = chrono::steady_clock::now().time_since_epoch().count();
        return C ^ ((p.x << 32) ^ p.y);
    }
};


i64 dist2(pt a, pt b) {
    i64 dx = a.x - b.x;
    i64 dy = a.y - b.y;
    return dx*dx + dy*dy;
}


pair<int,int> closest_pair_of_points(vector<pt> P) {
    int n = int(P.size());
    assert(n >= 2);

    // if there is a duplicated point, we have the solution
    unordered_map<pt,int,CustomHashPoint> previous;
    for (int i = 0; i < int(P.size()); ++i) {
        auto it = previous.find(P[i]);
        if (it != previous.end()) {
            return {it->second, i};
```

```
        }
        previous[P[i]] = i;
    }

    unordered_map<pt,vector<int>,CustomHashPoint> grid;
    grid.reserve(n);

    mt19937 rd(chrono::system_clock::now().time_since_epoch().count());
    uniform_int_distribution<int> dis(0, n-1);

    i64 d2 = dist2(P[0], P[1]);
    pair<int,int> closest = {0, 1};

    auto candidate_closest = [&](int i, int j) -> void {
        i64 ab2 = dist2(P[i], P[j]);
        if (ab2 < d2) {
            d2 = ab2;
            closest = {i, j};
        }
    };

    for (int i = 0; i < n; ++i) {
        int j = dis(rd);
        int k = dis(rd);
        while (j == k) k = dis(rd);
        candidate_closest(j, k);
    }

    i64 d = i64( sqrt(ld(d2)) + 1 );

    for (int i = 0; i < n; ++i) {
        grid[{P[i].x/d, P[i].y/d}].push_back(i);
    }

    // same block
    for (const auto& it : grid) {
        int k = int(it.second.size());
        for (int i = 0; i < k; ++i) {
            for (int j = i+1; j < k; ++j) {
                candidate_closest(it.second[i], it.second[j]);
            }
        }
    }

    // adjacent blocks
    for (const auto& it : grid) {
        auto coord = it.first;
        for (int dx = 0; dx <= 1; ++dx) {
            for (int dy = -1; dy <= 1; ++dy) {
                if (dx == 0 and dy == 0) continue;
                pt neighbour = pt(
                    coord.x  + dx,
                    coord.y + dy
                );
                for (int i : it.second) {
                    if (not grid.count(neighbour)) continue;
                    for (int j : grid.at(neighbour)) {
                        candidate_closest(i, j);
                    }
                }
            }
        }
    }

    return closest;
}
```

Listing 25: ClosestPair.cpp

8 DP

8.1 LIS - Longest Increasing Subsequence

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

// longest increasing subsequence in O(n log n)
int lis(vector<int> const& a) {
    int n = a.size();
    const int INF = 1e9;
    vector<int> d(n+1, INF);
    d[0] = -INF;

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

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

Listing 26: LIS.cpp

8.2 DP Divide and Conquer

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

int m, n;
vector<long long> dp_before, dp_cur;

long long C(int i, int j);

// compute dp_cur[l], ... dp_cur[r] (inclusive)
void compute(int l, int r, int optl, int optr) {
    if (l > r)
        return;

    int mid = (l + r) >> 1;
    pair<long long, int> best = {LLONG_MAX, -1};

    for (int k = optl; k <= min(mid, optr); k++) {
        best = min(best, {(k ? dp_before[k - 1] : 0) + C(k, mid), k});
    }

    dp_cur[mid] = best.first;
    int opt = best.second;

    compute(l, mid - 1, optl, opt);
    compute(mid + 1, r, opt, optr);
}

long long solve() {
    dp_before.assign(n,0);
    dp_cur.assign(n,0);

    for (int i = 0; i < n; i++)
        dp_before[i] = C(0, i);

    for (int i = 1; i < m; i++) {
        compute(0, n - 1, 0, n - 1);
        dp_before = dp_cur;
    }
```

```
    return dp_before[n - 1];
}
```

Listing 27: DPDivideAndConquer.cpp

9 Bitmasking

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

bool is_set(unsigned int number, int x) {
    return (number >> x) & 1;
}

int set_bit(int number, int x) {
    return number | (1 << x);
}

int clear_bit(int number, int x) {
    return number & ~(1 << x);
}

int toggle_bit(int number, int x) {
    return number ^ (1 << x);
}

int modify_bit(int number, int x, bool val) {
    return (number & ~(1 << x)) | (val << x);
}

bool isDivisibleByPowerOf2(int n, int k) {
    int powerOf2 = 1 << k;
    return (n & (powerOf2 - 1)) == 0;
}

bool isPowerOfTwo(unsigned int n) {
    return n && !(n & (n - 1));
}

int nextPowerOfTwo(int n) {
    n--;
    n |= n >> 1;
    n |= n >> 2;
    n |= n >> 4;
    n |= n >> 8;
    n |= n >> 16;
    return n + 1;
}

// Contar bits (g++ >= 20)
long long countSetBits(unsigned int n) {
    long long count = 0;
    while (n > 0) {
        int x = std::bit_width(n) - 1;
        if (x > 0) {
            count += (long long)x * (1LL << (x - 1));
        }
        n -= (1U << x);
        count += n + 1;
    }
    return count;
}

// Contar bits (g++ < 20)
long long countSetBits(unsigned int n) {
    long long count = 0;
    while (n > 0) {
        int x = static_cast<int>(std::log2(n));
        if (x > 0) {
```

```
            count += (long long)x * (1LL << (x - 1));
        }
        n -= (1U << x);
        count += n + 1;
    }
    return count;
}

int countSetBitsBuiltin(int n) {
    return __builtin_popcount(n);
}

int countSetBitsLong(long long n) {
    return __builtin_popcountll(n);
}

// Posición de bits
int lowestSetBit(int n) {
    return __builtin_ffs(n) - 1; // First set bit (0-indexed)
}

int highestSetBit(int n) {
    return 31 - __builtin_clz(n); // Count leading zeros
}

int trailingZeros(int n) {
    return __builtin_ctz(n);
}

int leadingZeros(int n) {
    return __builtin_clz(n);
}

// Operaciones con máscaras de bits
int getAllSetBits(int n) {
    return (1 << n) - 1; // Máscara con los primeros n bits en 1
}

int getRangeMask(int l, int r) {
    // Máscara con bits de l a r en 1 (0-indexed)
    return ((1 << (r - l + 1)) - 1) << l;
}

int clearRightmostSetBit(int n) {
    return n & (n - 1);
}

int isolateRightmostSetBit(int n) {
    return n & (-n);
}

int isolateRightmostZeroBit(int n) {
    return ~n & (n + 1);
}

int setRightmostZeroBit(int n) {
    return n | (n + 1);
}

// Iterar sobre subconjuntos
void iterateSubsets(int mask) {
    // Iterar sobre todos los subconjuntos de mask
    for (int s = mask; s > 0; s = (s - 1) & mask) {
        // Procesar subconjunto s
    }
}

void iterateAllMasks(int n) {
    // Iterar sobre todas las máscaras de n bits
    for (int mask = 0; mask < (1 << n); mask++) {
        // Procesar mask
    }
}
```

```

void iterateSetBits(int mask) {
    // Iterar sobre las posiciones de los bits en 1
    while (mask) {
        int pos = __builtin_ctz(mask);
        // Procesar posición pos
        mask &= mask - 1; // Eliminar el bit más bajo
    }
}

// Operaciones avanzadas
int reverseBits(unsigned int n) {
    n = ((n >> 1) & 0x55555555) | ((n & 0x55555555) << 1);
    n = ((n >> 2) & 0x33333333) | ((n & 0x33333333) << 2);
    n = ((n >> 4) & 0x0F0F0F0F) | ((n & 0x0F0F0F0F) << 4);
    n = ((n >> 8) & 0x00FF00FF) | ((n & 0x00FF00FF) << 8);
    n = (n >> 16) | (n << 16);
    return n;
}

int swapBits(int n, int i, int j) {
    // Intercambiar bits en posiciones i y j
    if (((n >> i) & 1) != ((n >> j) & 1)) {
        n ^= (1 << i) | (1 << j);
    }
    return n;
}

bool parityBit(unsigned int n) {
    // true si número impar de bits en 1
    return __builtin_parity(n);
}

// Operaciones con máscaras de bits para DP y combinatoria
int addElement(int mask, int pos) {
    return mask | (1 << pos);
}

int removeElement(int mask, int pos) {
    return mask & ~(1 << pos);
}

bool hasElement(int mask, int pos) {
    return (mask >> pos) & 1;
}

int maskSize(int mask) {
    return __builtin_popcount(mask);
}

int complementMask(int mask, int n) {
    // Complemento de mask con respecto a n bits
    return ((1 << n) - 1) ^ mask;
}

int unionMask(int a, int b) {
    return a | b;
}

int intersectionMask(int a, int b) {
    return a & b;
}

int differenceMask(int a, int b) {
    return a & ~b;
}

bool isSubset(int subset, int mask) {
    return (subset & mask) == subset;
}

// XOR útil
int xorRange(int l, int r) {
```

```cpp
    // XOR de todos los números de l a r
    auto xor_till = [](int n) {
        int mod = n % 4;
        if (mod == 0) return n;
        if (mod == 1) return 1;
        if (mod == 2) return n + 1;
        return 0;
    };
    return xor_till(r) ^ xor_till(l - 1);
}

int findXorPair(int arr[], int n) {
    // XOR de todos los elementos
    int xor_all = 0;
    for (int i = 0; i < n; i++) {
        xor_all ^= arr[i];
    }
    return xor_all;
}

// Máximo XOR de dos números en un rango
int maxXOR(int l, int r) {
    int xor_val = l ^ r;
    int msb = highestSetBit(xor_val);
    return (1 << (msb + 1)) - 1;
}
```

Listing 28: Bitwise.cpp