

Fast egg to chicken transformation - ICPC Library

Contents

1	Data Structures	1
1.1	Segment Tree	1
1.2	Segment Tree Lazy Propagation	2
1.3	Segment Tree Range Update Point Query	2
1.4	Merge Sort Tree	3
1.5	Fenwick Tree	3
1.6	Disjoint Set Union	4
1.7	Sparse Table	4
2	Dynamic Programming	4
2.1	Prefix Sum Array	4
3	Graph Algorithms	5
3.1	LCA	5
3.2	Articulation Point	5
3.3	Bridges	6
3.4	HLD Edges	6
3.5	HLD Nodes	8
3.6	Strongly Connected Component	9
3.7	Maximum Flow	10
4	Math	10
4.1	Basic Math	10
4.2	Fatorial	11
4.3	Prime Number	11
4.4	Matrix Exponentiation	11
4.5	Array Permutation	12
4.6	Modular Arithmetic	13
4.7	Extended Euclidean	14
4.8	Geometric Operations	14
5	Geometry	15
5.1	Basic Geometry	15
5.2	Convex Hull	19
5.3	Convex Polygon	19
6	String Algorithms	19
6.1	Prefix Function	19
6.2	Z Function	20
6.3	String Hashing	20
6.4	Trie	21
6.5	Trie Int	22
6.6	Aho Corasick	22
6.7	Suffix Array	23
6.8	Manacher	24
6.9	Lyndon Factorization	25
7	Miscellaneous	25
7.1	Ternary Search	25
7.2	Longest Increasing Subsequence	26
7.3	Mo Algorithm	26
7.4	Inversions Count	26

1 Data Structures

1.1 Segment Tree

```
#include <vector>
template<class T>
```

```
class SegmentTree {
    struct Node {
        T val;
        Node(T x) : val(x) {}
        Node () : val(0) {}
    };
    int N;
    std::vector<T> a;
    std::vector<Node> tr;
    Node neutral;
    inline Node join(const Node &a, const Node &b) {
        return Node(a.val + b.val);
    }
    void build(int node, int l, int r) {
        if (l == r) {
            tr[node] = Node(a[l]);
            return;
        }
        int mid = l+(r-l)/2, lc = (node << 1);
        build(lc, l, mid);
        build(lc+1, mid+1, r);
        tr[node] = join(tr[lc], tr[lc+1]);
    }
    void update(int node, int l, int r, int idx, T x) {
        if (l == r) {
            tr[node] = Node(x);
            return;
        }
        int mid = l+(r-l)/2, lc = (node << 1);
        if(idx <= mid) update(lc, l, mid, idx, x);
        else update(lc+1, mid+1, r, idx, x);
        tr[node] = join(tr[lc], tr[lc+1]);
    }
    Node query(int node, int l, int r, int ql, int qr) {
        if (r < l or qr < l or r < ql) return neutral;
        if (ql <= l and r <= qr) return tr[node];
        int mid = l+(r-l)/2, lc = (node << 1);
        return join(query(lc, l, mid, ql, std::min(qr, mid)),
            query(lc+1, mid+1, r, std::max(mid+1, ql), qr));
    }
    // Searching for the first element greater than a given amount (
    // segtree of max)
    int get_first(int node, int l, int r, int ql, int qr, T x) {
        if (r < l or qr < l or r < ql) return -1;
        if (ql <= l and r <= qr) {
            if (tr[node].val <= x) return -1;
            while (l != r) {
                int mid = l+(r-l)/2, lc = (node << 1);
                if (tr[lc].val > x) {
                    node = lc;
                    r = mid;
                } else {
                    node = lc+1;
                    l = mid+1;
                }
            }
            return l;
        }
        int mid = l+(r-l)/2, lc = (node << 1);
        int rs = get_first(lc, l, mid, ql, std::min(qr, mid), x);
        if (~rs) return rs;
```

```

    return get_first(lc+1, mid+1, r, std::max(ql, mid+1), qr, x);
}
public:
template<class MyIterator>
SegmentTree(MyIterator begin, MyIterator end) {
    N = end-begin-1;
    tr.assign(4*N, 0);
    a = std::vector<T>(begin, end);
    build(1, 1, N);
}
SegmentTree(int n) : N(n) {
    tr.assign(4*N, 0);
    a.assign(N+1, 0);
}
T query(int l, int r) {
    return query(1, 1, N, l, r).val;
}
void update(int idx, T x) {
    update(1, 1, N, idx, x);
}
};

```

1.2 Segment Tree Lazy Propagation

```

#include <vector>
template<class T>
class SegmentTreeLazy {
    struct Node {
        T val;
        Node(T x) : val(x) {}
        Node() : val(0) {}
    };
    int N;
    std::vector<T> a, lazy;
    std::vector<Node> tr;
    Node neutral;
    inline Node join(const Node &a, const Node &b) {
        return Node(a.val + b.val);
    }
    inline void upLazy(int node, int l, int r) {
        if (lazy[node] == 0) return;
        tr[node].val += lazy[node]*(r-l+1);
        int lc = (node << 1);
        (l != r ? lazy[lc] += lazy[node], lazy[lc+1] += lazy[node] : 0);
        lazy[node] = 0;
    }
    void build(int node, int l, int r) {
        if (l == r) tr[node] = Node(a[l]);
        else {
            int mid = l+(r-l)/2, lc = (node << 1);
            build(lc, l, mid);
            build(lc+1, mid+1, r);
            tr[node] = join(tr[lc], tr[lc+1]);
        }
    }
    void update(int node, int l, int r, int ul, int ur, T x) {
        upLazy(node, l, r);
        if (r < l or ur < ul or r < ul) return;
        if (ul <= l and r <= ur) {
            lazy[node] += x;

```

```

        upLazy(node, l, r);
    } else {
        int mid = l+(r-l)/2, lc = (node << 1);
        update(lc, l, mid, ul, std::min(ur, mid), x);
        update(lc+1, mid+1, r, std::max(mid+1, ul), ur, x);
        tr[node] = join(tr[lc], tr[lc+1]);
    }
}
Node query(int node, int l, int r, int ql, int qr) {
    upLazy(node, l, r);
    if (r < l or qr < ql or qr < l or r < ql) return neutral;
    if (ql <= l and r <= qr) return tr[node];
    int mid = l+(r-l)/2, lc = (node << 1);
    return join(query(lc, l, mid, ql, std::min(qr, mid)),
               query(lc+1, mid+1, r, std::max(mid+1, ql), qr));
}
public:
template<class MyIterator>
SegmentTreeLazy(MyIterator begin, MyIterator end) {
    N = end-begin-1;
    tr.assign(4*N, 0);
    lazy.assign(4*N, 0);
    a = std::vector<T>(begin, end);
    build(1, 1, N);
}
SegmentTreeLazy(int n) : N(n) {
    tr.assign(4*N, 0);
    lazy.assign(4*N, 0);
    a.assign(N+1, 0);
}
T query(int l, int r) {
    return query(1, 1, N, l, r).val;
}
void update(int l, int r, T x) {
    update(1, 1, N, l, r, x);
}
};

```

1.3 Segment Tree Range Update Point Query

```

#include <vector>
template<class T>
class SegmentTree {
    struct Node {
        T val;
        Node(T x) : val(x) {}
        Node() : val(0) {}
    };
    int N;
    std::vector<T> a;
    std::vector<Node> tr;
    Node neutral;
    inline Node join(const Node &a, const Node &b) {
        return Node();
    }
    void build(int node, int l, int r) {
        if (l == r) {
            tr[node] = Node(a[l]);
        } else {
            int mid = l+(r-l)/2, lc = (node << 1);

```

```

        build(lc, l, mid);
        build(lc+1, mid+1, r);
    }
}
T query(int node, int l, int r, int idx) {
    if (l == r) return tr[node].val;
    int mid = l+(r-l)/2, lc = (node << 1);
    if (idx <= mid) return tr[node].val + query(lc, l, mid, idx);
    else return tr[node].val + query(lc+1, mid+1, r, idx);
}
void update(int node, int l, int r, int ql, int qr, T x) {
    if (r < l or qr < l or r < ql) return;
    if (ql <= l and r <= qr) {
        int delta = x-tr[node].val;
        tr[node].val += delta;
    } else {
        int mid = l+(r-l)/2, lc = (node << 1);
        update(lc, l, mid, ql, std::min(qr, mid), x);
        update(lc+1, mid+1, r, std::max(mid+1, ql), qr, x);
    }
}
public:
    template<class MyIterator>
    SegmentTree(MyIterator begin, MyIterator end) {
        N = end-begin-1;
        tr.assign(4*N, 0);
        a = std::vector<T>(begin, end);
        build(1, 1, N);
    }
    SegmentTree(int n) : N(n) {
        tr.assign(4*N, 0);
        a.assign(N+1, 0);
    }
    T query(int idx) {
        return query(1, 1, N, idx);
    }
    void update(int l, int r, T x) {
        update(1, 1, N, l, r, x);
    }
};

```

1.4 Merge Sort Tree

```

#include <vector>
#define all(x) (x).begin(), (x).end()
const int64_t INF = 0x3f3f3f3f;
template<class T>
class MergeSortTree {
    typedef std::vector<T> Node;
    inline Node join(const Node &a, const Node &b) {
        Node ans;
        merge(all(a), all(b), std::back_inserter(ans));
        return ans;
    }
    int N;
    std::vector<Node> tr;
    std::vector<T> a;
    Node neutral;
    inline int szEq(int node, int k) {

```

```

        return upper_bound(all(tr[node]), k)-lower_bound(all(tr[node]), k)
            ; }
    inline int szLe(int node, int k) {
        return upper_bound(all(tr[node]), k)-tr[node].begin(); }
    inline int szLt(int node, int k) {
        return lower_bound(all(tr[node]), k)-tr[node].begin(); }
    void build(int node, int l, int r) {
        if (l == r) return (void)tr[node].push_back(a[l]);
        int mid = l+(r-l)/2, lc = (node << 1);
        build(lc, l, mid);
        build(lc+1, mid+1, r);
        tr[node] = join(tr[lc], tr[lc+1]);
    }
    // Find the amount of value (lower, lower or equal, equal) than x
    int query(int node, int l, int r, int ql, int qr, int k, int op) {
        if (r < l or qr < l or r < ql) return 0;
        if (ql <= l and r <= qr) return (op == -1 ? szLt(node, k) : op ==
            1 ? szLe(node, k) : szEq(node, k));
        int mid = l+(r-l)/2, lc = (node << 1);
        return query(lc, l, mid, ql, std::min(qr, mid), k, op) +
            query(lc+1, mid+1, r, std::max(ql, mid+1), qr, k, op);
    }
    //Find the smallest number greater or equal to x
    T query(int node, int l, int r, int ql, int qr, T x) {
        if (r < l or qr < l or r < ql) return INF;
        if (ql <= l and r <= qr) {
            auto pos = lower_bound(all(tr[node]), x);
            if (pos != tr[node].end()) return *pos;
            return INF;
        }
        int mid = l+(r-l)/2, lc = (node << 1);
        return std::min(query(lc, l, mid, ql, std::min(mid, qr), x),
            query(lc+1, mid+1, r, std::max(ql, mid+1), qr, x));
    }
public:
    template<class MyIterator>
    MergeSortTree(MyIterator begin, MyIterator end) {
        N = end-begin-1;
        a = std::vector<T>(begin, end);
        tr.assign(4*N, std::vector<T>());
        build(1, 1, N);
    }
    int lt(int l, int r, int k) {
        return query(1, 1, N, l, r, k, -1);
    }
    int le(int l, int r, int k) {
        return query(1, 1, N, l, r, k, 1);
    }
    int eq(int l, int r, int k) {
        return query(1, 1, N, l, r, k, 0);
    }
    T query(int l, int r, T x) {
        return query(1, 1, N, l, r, x);
    }
};

```

1.5 Fenwick Tree

```

#include <vector>
template<class T>

```

```

class FenwickTree {
    int N;
    std::vector<T> tr, a;
public:
    void add(int idx, T x){
        a[idx] = x;
        for (; idx <= N; idx += (idx & -idx))
            tr[idx] += x;
    }
    void set(int idx, T x){
        T delta = x-a[idx];
        a[idx] = x;
        for (; idx <= N; idx += (idx & -idx))
            tr[idx] += delta;
    }
    T query(int idx){
        T res = 0;
        for (; idx > 0; idx -= (idx & -idx))
            res += tr[idx];
        return res;
    }
    T query(int l, int r){
        return query(r)-query(l-1);
    }
    FenwickTree(int n) : N(n) {
        tr.resize(N+1, 0);
        a.resize(N+1, 0);
    }
};

```

1.6 Disjoint Set Union

```

#include <vector>
#include <numeric>
class DSU {
    int N;
    std::vector<int> link, sz;
public:
    int id(int x) { return link[x] = (link[x] == x ? x : id(link[x])); }
    int same(int x, int y) { return (id(x) == id(y)); }
    void unite(int x, int y) {
        x = id(x); y = id(y);
        if (x == y) return;
        if (sz[x] < sz[y]) std::swap(x,y);
        link[y] = x;
        sz[x] += sz[y];
    }
    int size(int x) { return sz[id(x)]; }
    DSU (int n) : N(n) {
        sz.assign(N+1, 1);
        link.resize(N+1);
        iota(link.begin(), link.end(), 0);
    }
};

```

1.7 Sparse Table

```

#include <vector>

```

```

template<class T>
class SparseTable {
    std::vector<std::vector<T>> st;
    std::vector<int> log2;
    T neutral = 0x3f3f3f3f;
    const int nLog = 20;
    T join(T a, T b) {
        return std::min(a, b);
    }
public:
    template<class MyIterator>
    SparseTable(MyIterator begin, MyIterator end) {
        int n = end-begin;
        log2.resize(n+1);
        log2[1] = 0;
        for (int i = 2; i <= n; ++i)
            log2[i] = log2[i/2]+1;
        st.resize(n, std::vector<T>(nLog, neutral));
        for (int i = 0; i < n; ++i, ++begin)
            st[i][0] = *begin;
        for (int j = 1; j < nLog; ++j)
            for (int i = 0; i+(1<<(j-1)) < n; ++i)
                st[i][j] = join(st[i][j-1], st[i+(1<<(j-1))][j-1]);
    }
    T query(int l, int r) {
        int sz = r-l+1;
        T ans = neutral;
        for (int j = nLog-1; j >= 0; --j) {
            if (sz & (1 << j)) {
                neutral = join(neutral, st[l][j]);
                l += (1 << j);
            }
        }
        return ans;
    }
    T queryRMQ(int l, int r) {
        int j = log2[r-l+1];
        return join(st[l][j], st[r-(1 << j)+1][j]);
    }
};

```

2 Dynamic Programming

2.1 Prefix Sum Array

```

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

const int maxn = 1e5+5;

/*
Answer queries:
Q(L, R) = 1*A[L] + 2*A[L+1] + 3*A[L+2] + ... (R-L+1)*A[R]
*/

int n, a[maxn];
int psa[maxn], ips[maxn];

```

```

int q, l, r, ans;

void computePSA() {
    for (int i = 1; i <= n; ++i) {
        psa[i] = psa[i-1] + a[i];
        ips[i] = ips[i-1] + i * a[i];
    }
    while (q--) {
        cin >> l >> r;
        ans = ips[r] - ips[l-1] - (l - 1) * (psa[r] - psa[l-1]);
        cout << ans << '\n';
    }
}

```

3 Graph Algorithms

3.1 LCA

```

#include <bits/stdc++.h>
using namespace std;
using ii = pair<int, int>;

const int maxn = 1e4+5;
const int L = 21;

namespace LCA {
    int anc[maxn][L], tin[maxn], tout[maxn], deep[maxn], h[maxn];
    vector<ii> gr[maxn];
    int n, timer;

    inline void init(int _n) {
        n = _n;
        timer = 0;
        for (int i = 0; i < n; ++i) {
            deep[i] = 0;
            h[i] = 0;
            tin[i] = tout[i] = 0;
            gr[i].clear();
            for (int j = 0; j < L; ++j) anc[i][j] = 0;
        }
    }

    void dfs(int u, int p) {
        tin[u] = ++timer;
        anc[u][0] = p;
        for (int i = 1; i < L; ++i) anc[u][i] = anc[ anc[u][i-1] ][i-1];
        for (auto [to, w] : gr[u]) if (to != p) {
            deep[to] = deep[u] + 1;
            h[to] = h[u] + w;
            dfs(to, u);
        }
        tout[u] = ++timer;
    }

    inline void addEdge(int u, int v, int w) {
        gr[u].emplace_back(v, w);
    }
}

```

```

inline bool is_anc(int u, int v) {
    return (tin[u] <= tin[v] and tout[v] <= tout[u]);
}

inline int lca(int u, int v) {
    if (is_anc(u, v)) return u;
    if (is_anc(v, u)) return v;
    for (int i = L-1; i >= 0; --i) if (!is_anc(anc[u][i], v)) u = anc[u][i];
    return anc[u][0];
}

inline int kth_anc(int u, int k) {
    if (--k == 0) return u;
    for (int i = L-1; i >= 0; --i) {
        if (k - (1 << i) >= 0) {
            u = anc[u][i];
            k -= (1 << i);
        }
    }
    return u;
}

inline int dist_w(int a, int b) {
    return h[a] + h[b] - 2 * h[lca(a, b)];
}

inline int dist(int a, int b) {
    return deep[a] + deep[b] - 2 * deep[lca(a, b)];
}

inline void build() {
    dfs(0, 0);
}
};

```

3.2 Articulation Point

```

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

const int maxn = 1e5+5;

vector<int> gr[maxn];
int used[maxn], tin[maxn], low[maxn];
int n, timer;

void is_cutpoint(int u) {
    return;
}

void dfs(int u, int p = -1) {
    used[u] = true;
    tin[u] = low[u] = timer++;
    int children = 0;
    for (int to : gr[u]) if (to != p) {
        if (used[to]) { //Is a back edge
            low[u] = min(low[u], tin[to]);
        } else {
            dfs(to, u);

```

```

    low[u] = min(low[u], low[to]);
    if (low[to] >= tin[u] and p != -1) {
        is_cutpoint(u);
    }
    ++children;
}
}
if (p == -1 and children > 1)
    is_cutpoint(u);
}

void find_cutpoints() {
    timer = 0;
    for (int i = 0; i < n; ++i) {
        used[i] = false;
        tin[i] = -1;
        low[i] = -1;
    }
    for (int i = 0; i < n; ++i)
        if (!used[i]) dfs(i);
}

```

3.3 Bridges

```

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

const int maxn = 1e5+5;

vector<int> gr[maxn];
int used[maxn], tin[maxn], low[maxn];
int n, timer;

void is_bridge(int u, int v) {
    return;
}

void dfs(int u, int p = -1) {
    used[u] = true;
    tin[u] = low[u] = timer++;
    for (int to : gr[u]) if (to != p) {
        if (used[to]) { //Is a back edge
            low[u] = min(low[u], tin[to]);
        } else {
            dfs(to, u);
            low[u] = min(low[u], low[to]);
            if (low[to] > tin[u]) {
                is_bridge(u, to);
            }
        }
    }
}

void find_bridges() {
    timer = 0;
    for (int i = 0; i < n; ++i) {
        used[i] = false;
        tin[i] = -1;
        low[i] = -1;
    }
}

```

```

for (int i = 0; i < n; ++i)
    if (!used[i]) dfs(i);
}

```

3.4 HLD Edges

```

#include <bits/stdc++.h>
using namespace std;
using i64 = long long int;
using vi64 = vector<i64>;
using ii = pair<int, int>;

#define fi first
#define se second

const int maxn = 1e5+5;

namespace SegmentTree {

    struct Node {
        i64 val;
        Node(i64 x) : val(x) {}
        Node () : val(0LL) {}
    };

    inline Node join(const Node &a, const Node &b) {
        return Node(a.val + b.val);
    }

    int n;
    i64 lazy[4*maxn];
    Node tree[4*maxn];
    Node neutral;
    i64 lazyNeutral = -1LL;

    inline void upLazy(int node, int l, int r) {
        if (lazy[node] == lazyNeutral) return;

        tree[node].val += lazy[node] * (r - l + 1LL); //To increment value
        // tree[node].val = lazy[node] * (r - l + 1LL); //To set value

        if (l != r) {
            int lc = (node << 1);
            // lazy[lc] = lazy[node]; //To set value
            // lazy[lc+1] = lazy[node]; //To set value
            lazy[lc] = (lazy[lc] == lazyNeutral ? lazy[node] : lazy[lc] +
                lazy[node]); //To increment value
            lazy[lc+1] = (lazy[lc+1] == lazyNeutral ? lazy[node] : lazy[lc]
                +1 + lazy[node]); //To increment value
        }

        lazy[node] = lazyNeutral;
    }

    void build(int node, int l, int r, int *v) {
        lazy[node] = lazyNeutral;
        if (l == r) { tree[node].val = v[l]; return; }
        int mid = l+(r-l)/2, lc = (node << 1);
        build(lc, l, mid, v);
        build(lc+1, mid+1, r, v);
    }
}

```

```

    tree[node] = join(tree[lc], tree[lc+1]);
}

void update(int node, int l, int r, int ul, int ur, i64 val) {
    upLazy(node, l, r);
    if (r < l or ur < ul or ur < l or r < ul) return;
    if (ul <= l and r <= ur) {
        lazy[node] = val; // To set value
        lazy[node] = (lazy[node] == lazyNeutral ? val : lazy[node] + val);
        // To increment value
        upLazy(node, l, r);
        return;
    }
    int mid = l+(r-l)/2, lc = (node << 1);
    update(lc, l, mid, ul, min(mid, ur), val);
    update(lc+1, mid+1, r, max(mid+1, ul), ur, val);
    tree[node] = join(tree[lc], tree[lc+1]);
}

Node query(int node, int l, int r, int ql, int qr) {
    upLazy(node, l, r);
    if (r < l or qr < ql or qr < l or r < ql) return neutral;
    if (ql <= l and r <= qr) return tree[node];
    int mid = l+(r-l)/2, lc = (node << 1);
    return join(query(lc, l, mid, ql, min(mid, qr)), query(lc+1, mid
        +1, r, max(mid+1, ql), qr));
}

void build(int _n, int *v) {
    n = _n;
    build(1, 1, n, v);
}

i64 query(int l, int r) {
    return query(1, 1, n, l, r).val;
}

void update(int l, int r, i64 val) {
    update(1, 1, n, l, r, val);
}
};

namespace HLD {
    struct edge {
        int a; i64 w;
        edge () {}
        edge (int to, i64 ww) : a(to), w(ww) {}
    };

    vector<edge> gr[maxn];
    int pos[maxn], st[maxn], pai[maxn];
    int sobe[maxn], h[maxn], v[maxn], timer;
    int hei[maxn], deep[maxn];

    inline void addEdge(int a, int b, i64 w = 1LL) {
        gr[a].push_back(edge(b, w));
    }

    //O(n)
    void dfs(int u, int p = -1) {

```

```

        st[u] = 1;
        for (auto &e : gr[u]) if (e.a != p) {
            sobe[e.a] = e.w;
            dfs(e.a, u);
            st[u] += st[e.a];
            if (st[e.a] > st[gr[u][0].a] or gr[u][0].a == p) swap(e, gr[u]
                [0]);
        }
    }

    //O(n)
    void build_hld(int u, int p = -1) {
        pos[u] = ++timer;
        v[pos[u]] = sobe[u];
        for (auto e : gr[u]) if (e.a != p) {
            pai[e.a] = u;
            h[e.a] = (e.a == gr[u][0].a ? h[u] : e.a);
            build_hld(e.a, u);
        }
    }

    inline void build(int root = 0) {
        timer = 0;
        h[root] = 0;
        hei[root] = 0;
        deep[root] = 0;
        dfs(root);
        build_hld(root);
        SegmentTree::build(timer, v);
    }

    //O(log^2 (n))
    i64 query_path(int a, int b) {
        if (a == b) return 0LL;
        if (pos[a] < pos[b]) swap(a, b);

        if (h[a] == h[b]) return SegmentTree::query(1+pos[b], pos[a]);
        return SegmentTree::query(pos[h[a]], pos[a]) +
            query_path(pai[h[a]], b);
    }

    //O(log^2 (n))
    void update_path(int a, int b, int x) {
        if (a == b) return;
        if (pos[a] < pos[b]) swap(a, b);
        if (h[a] == h[b]) return (void) SegmentTree::update(1+pos[b], pos[
            a], x);
        SegmentTree::update(1, 1, timer, pos[h[a]], pos[a], x);
        update_path(pai[h[a]], b, x);
    }

    //O(log(n))
    inline i64 query_subtree(int a) {
        if (st[a] == 1) return 0LL;
        return SegmentTree::query(1+pos[a], pos[a]+st[a]-1);
    }

    //O(log(n))
    inline void update_subtree(int a, int x) {
        if (st[a] == 1) return;
        SegmentTree::update(1+pos[a], pos[a]+st[a]-1, x);
    }
}

```

```

}

//O(log(n))
int lca(int a, int b) {
    if (pos[a] < pos[b]) swap(a, b);
    return (h[a] == h[b] ? b : lca(pai[h[a]], b));
}

//O(log(n))
i64 distw(int a, int b) {
    return hei[a] + hei[b] - 2 * hei[lca(a, b)];
}

//O(log(n))
int dist(int a, int b) {
    return deep[a] + deep[b] - 2 * deep[lca(a, b)];
}
};

```

3.5 HLD Nodes

```

#include <bits/stdc++.h>
using namespace std;
using i64 = long long int;
using vi64 = vector<i64>;

const int maxn = 1e5+5;

namespace SegmentTree {

    struct Node {
        i64 val;
        Node(i64 x) : val(x) {}
        Node () : val(0LL) {}
    };

    inline Node join(const Node &a, const Node &b) {
        return Node(a.val + b.val);
    }

    int n;
    i64 lazy[4*maxn];
    Node tree[4*maxn];
    Node neutral;
    i64 lazyNeutral = -1LL;

    inline void upLazy(int node, int l, int r) {
        if (lazy[node] == lazyNeutral) return;

        tree[node].val += lazy[node] * (r - l + 1LL); //To increment value
        // tree[node].val = lazy[node] * (r - l + 1LL); //To set value

        if (l != r) {
            int lc = (node << 1);
            // lazy[lc] = lazy[node]; //To set value
            // lazy[lc+1] = lazy[node]; //To set value
            lazy[lc] = (lazy[lc] == lazyNeutral ? lazy[node] : lazy[lc] +
                lazy[node]); //To increment value
            lazy[lc+1] = (lazy[lc+1] == lazyNeutral ? lazy[node] : lazy[lc
                +1] + lazy[node]); //To increment value
        }
    }
}

```

```

}

lazy[node] = lazyNeutral;
}

void build(int node, int l, int r, int *v) {
    lazy[node] = lazyNeutral;
    if (l == r) { tree[node].val = v[l]; return; }
    int mid = l+(r-l)/2, lc = (node << 1);
    build(lc, l, mid, v);
    build(lc+1, mid+1, r, v);
    tree[node] = join(tree[lc], tree[lc+1]);
}

void update(int node, int l, int r, int ul, int ur, i64 val) {
    upLazy(node, l, r);
    if (r < l or ur < ul or r < ul) return;
    if (ul <= l and r <= ur) {
        // lazy[node] = val; // To set value
        lazy[node] = (lazy[node] == lazyNeutral ? val : lazy[node] + val
            ); // To increment value
        upLazy(node, l, r);
        return;
    }
    int mid = l+(r-l)/2, lc = (node << 1);
    update(lc, l, mid, ul, min(mid, ur), val);
    update(lc+1, mid+1, r, max(mid+1, ul), ur, val);
    tree[node] = join(tree[lc], tree[lc+1]);
}

Node query(int node, int l, int r, int ql, int qr) {
    upLazy(node, l, r);
    if (r < l or qr < ql or r < ql) return neutral;
    if (ql <= l and r <= qr) return tree[node];
    int mid = l+(r-l)/2, lc = (node << 1);
    return join(query(lc, l, mid, ql, min(mid, qr)), query(lc+1, mid
        +1, r, max(mid+1, ql), qr));
}

void build(int _n, int *v) {
    n = _n;
    build(1, 1, n, v);
}

i64 query(int l, int r) {
    return query(1, 1, n, l, r).val;
}

void update(int l, int r, i64 val) {
    update(1, 1, n, l, r, val);
}

};

namespace HLD {
    struct edge {
        int a; i64 w;
        edge () {}
        edge (int to, i64 ww) : a(to), w(ww) {}
    };
}

```



```

vector<edge> gr[maxn];
int pos[maxn], st[maxn], pai[maxn];
int h[maxn], v[maxn], val[maxn], timer;
int deep[maxn], hei[maxn];

inline void addEdge(int a, int b, i64 w = 1LL) {
    gr[a].push_back(edge(b, w));
}

//O(n)
void dfs(int u, int p = -1) {
    st[u] = 1;
    for (auto &e : gr[u]) if (e.a != p) {
        pai[e.a] = u;
        deep[e.a] = deep[u] + 1;
        hei[e.a] = hei[u] + e.w;
        dfs(e.a, u);
        st[u] += st[e.a];
        if (st[e.a] > st[gr[u][0].a] or gr[u][0].a == p) swap(e, gr[u][0]);
    }
}

//O(n)
void build_hld(int u, int p = -1) {
    pos[u] = ++timer;
    v[pos[u]] = val[u];
    for (auto e : gr[u]) if (e.a != p) {
        h[e.a] = (e.a == gr[u][0].a ? h[u] : e.a);
        build_hld(e.a, u);
    }
}

void build(int root = 0) {
    timer = 0;
    h[root] = 0;
    hei[root] = 0;
    deep[root] = 0;
    dfs(root);
    build_hld(root);
    SegmentTree::build(timer, v);
}

//O(log^2(n))
i64 query_path(int a, int b) {
    if (pos[a] < pos[b]) swap(a, b);
    if (h[a] == h[b]) return SegmentTree::query(pos[b], pos[a]);
    return SegmentTree::query(pos[h[a]], pos[a]) + query_path(pai[h[a]], b);
}

//O(log^2(n))
void update_path(int a, int b, i64 x) {
    if (pos[a] < pos[b]) swap(a, b);
    if (h[a] == h[b]) return (void) SegmentTree::update(pos[b], pos[a], x);
    SegmentTree::update(pos[h[a]], pos[a], x);
    update_path(pai[h[a]], b, x);
}

//O(log(n))

```

```

inline i64 query_subtree(int a) {
    return SegmentTree::query(pos[a], pos[a]+st[a]-1);
}

//O(log(n))
inline void update_subtree(int a, i64 x) {
    SegmentTree::update(pos[a], pos[a]+st[a]-1, x);
}

//O(log(n))
int lca(int a, int b) {
    if (pos[a] < pos[b]) swap(a, b);
    return (h[a] == h[b] ? b : lca(pai[h[a]], b));
}

//O(log(n))
i64 distw(int a, int b) {
    return hei[a] + hei[b] - 2 * hei[lca(a, b)];
}

//O(log(n))
int dist(int a, int b) {
    return deep[a] + deep[b] - 2 * deep[lca(a, b)];
}
};

```

3.6 Strongly Connected Component

```

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

const int maxn = 1e5+5;

namespace SCC {
    vector<int> gr[maxn], gt[maxn];
    vector<int> order;
    int comp[maxn], used[maxn];
    int n, timer, scc;

    void init(int _n) {
        n = _n;
        scc = 0;
        order.clear();
        for (int i = 0; i < n; ++i) {
            used[i] = false;
            comp[i] = 0;
            gr[i].clear();
            gt[i].clear();
        }
    }

    void addEdge(int u, int v) {
        gr[u].push_back(v);
        gt[v].push_back(u);
    }

    void dfs1(int u) {
        used[u] = timer;
        for (int to : gr[u]) if (used[to] != timer) {
            dfs1(to);
        }
    }
}

```

```

    }
    order.push_back(u);
}

void dfs2(int u) {
    used[u] = timer;
    comp[u] = scc;
    for (int to : gt[u]) if (used[to] != timer) {
        dfs2(to);
    }
}

int get_scc() {
    ++timer;
    for (int u = 0; u < n; ++u)
        if (used[u] != timer) dfs1(u);

    ++timer;
    for (int i = n-1; i >= 0; --i)
        if (used[order[i]] != timer) {
            dfs2(order[i]);
            ++scc;
        }
    return scc;
}
};

```

3.7 Maximum Flow

```

#include <vector>
#include <queue>
const int INF = 0x3f3f3f3f;
namespace MaxFlow {
    std::vector<std::vector<int>> capacity;
    std::vector<std::vector<int>> gr;
    int N;
    void init(int n) { N = n;
        capacity.assign(N, std::vector<int>(N));
        gr.assign(N, std::vector<int>());
    }
    void addEdge(int u, int v, int cap) {
        gr[u].push_back(v);
        gr[v].push_back(u);
        capacity[u][v] += cap;
        capacity[v][u] += 0;
    }
    int bfs(int s, int t, std::vector<int> &parent) {
        fill(parent.begin(), parent.end(), -1);
        parent[s] = -2;
        std::queue<std::pair<int, int>> q;
        q.push({s, INF});
        while (!q.empty()) {
            auto [cur, flow] = q.front(); q.pop();
            for (int next : gr[cur]) {
                if (parent[next] == -1 and capacity[cur][next]) {
                    parent[next] = cur;
                    int new_flow = std::min(flow, capacity[cur][next]);
                    if (next == t)
                        return new_flow;
                    q.push({next, new_flow});
                }
            }
        }
    }
}

```

```

    }
    }
    return 0;
}
int maxflow(int s, int t) {
    int flow = 0;
    std::vector<int> parent(N);
    int new_flow;
    while (new_flow = bfs(s, t, parent)) {
        flow += new_flow;
        int cur = t;
        while (cur != s) {
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }
    return flow;
}
};

```

4 Math

4.1 Basic Math

```

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

namespace ModHash{
    const uint64_t MOD = (1ll<<61) - 1;
    uint64_t modmul(uint64_t a, uint64_t b){
        uint64_t l1 = (uint32_t)a, h1 = a>>32, l2 = (uint32_t)b, h2 = b
        >>32;
        uint64_t l = l1*l2, m = l1*h2 + l2*h1, h = h1*h2;
        uint64_t ret = (l&MOD) + (l>>61) + (h << 3) + (m >> 29) + ((m <<
        35) >> 3) + 1;
        ret = (ret & MOD) + (ret>>61);
        ret = (ret & MOD) + (ret>>61);
        return ret-1;
    }
};

uint64_t modMul(uint64_t a, uint64_t b, uint64_t MOD) {
    return (__uint128_t)a*b%MOD;
}

uint64_t binpow(uint64_t base, uint64_t exp, uint64_t MOD) {
    base %= MOD;
    uint64_t res = 1;
    while (exp > 0) {
        if (exp & 1) res = modMul(res, base, MOD);
        base = modMul(base, base, MOD);
        exp >>= 1;
    }
    return res;
}

```

```

uint64_t bigExp(uint64_t base, string exp, uint64_t MOD) {
    base %= MOD;
    uint64_t ans = 1LL;
    for (char c : exp) {
        ans = binpow(ans, 10LL, MOD);
        ans = modMul(ans, binpow(base, c-'0', MOD));
    }
    return ans;
}

uint64_t gcd(uint64_t a, uint64_t b) { return (b == 0 ? a : gcd(b, a%b)); }
uint64_t binary_gcd(uint64_t a, uint64_t b) {
    if (a == 0 or b == 0)
        return a ^ b;
    int shift = __builtin_ctzll(a | b);
    a >>= __builtin_ctzll(a);
    do {
        b >>= __builtin_ctzll(b);
        if (a > b)
            swap(a, b);
        b -= a;
    } while (b);
    return a << shift;
}
uint64_t lcm(uint64_t a, uint64_t b) { return a / binary_gcd(a, b) * b; }

```

4.2 Fatorial

```

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

const int64_t MOD = 1000000007LL;
const int MAXV = 1e6+5;

int64_t fat[MAXV], ifat[MAXV];

void init() {
    fat[0] = 1LL;
    for (int64_t i = 1; i < MAXV; ++i) {
        fat[i] = (i * fat[i-1]) % MOD;
    }
    ifat[MAXV-1] = inv_mod(fat[MAXV-1], MOD);
    for (int64_t i = MAXV-1; i >= 1; --i) {
        ifat[i-1] = (ifat[i] * i) % MOD;
    }
}

```

4.3 Prime Number

```

#include <bits/stdc++.h>
#include "math.h"

using namespace std;

```

```

bool check_composite(uint64_t n, uint64_t a, uint64_t d, int s) {
    uint64_t x = binpow(a, d, n);
    if (x == 1 or x == n - 1)
        return false;
    for (int r = 1; r < s; ++r) {
        x = modMul(x, x, n);
        if (x == n - 1)
            return false;
    }
    return true;
}

bool MillerRabin(uint64_t n) {
    if (n < 4)
        return (n == 2 or n == 3);

    uint64_t d = n - 1;
    int s = __builtin_ctzll(d);
    d >>= s;

    for (uint64_t a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
        // int a = 2 + rand() % (n - 3); (nondeterministic version)
        if (n == a)
            return true;
        if (check_composite(n, a, d, s))
            return false;
    }
    return true;
}

uint64_t pollard(uint64_t n) {
    auto f = [n](uint64_t x) { return (modMul(x, x, n) + 1) % n; };
    uint64_t x = 0, y = 0, t = 30, prd = 2, i = 1, q;
    while (t++ % 40 or binary_gcd(prd, n) == 1) {
        if (x == y)
            x = ++i, y = f(x);
        if ((q = modMul(prd, max(x, y) - min(x, y), n))
            prd = q;
            x = f(x), y = f(f(y));
        }
    }
    return binary_gcd(prd, n);
}

vector<uint64_t> factor(uint64_t n) {
    if (n == 1)
        return {};
    if (MillerRabin(n))
        return {n};
    uint64_t x = pollard(n);
    auto l = factor(x), r = factor(n / x);
    l.insert(l.end(), r.begin(), r.end());
    return l;
}

```

4.4 Matrix Exponentiation

```

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

typedef long long int i64;

```

```

const int mod = 1e9+7;
const int D = 3;

int d = D;

struct M {
    i64 m[D][D];

    i64* operator[](int i) {
        return m[i];
    }

    M operator-(M oth) {
        M res;
        for (int i = 0; i < d; ++i) {
            for (int j = 0; j < d; ++j) {
                res[i][j] = m[i][j] - oth[i][j];
            }
        }
        return res;
    }

    M operator+(M oth) {
        M res;
        for (int i = 0; i < d; ++i) {
            for (int j = 0; j < d; ++j) {
                res[i][j] = m[i][j] + oth[i][j];
            }
        }
        return res;
    }

    M operator*(M oth) {
        M res;
        for (int i = 0; i < d; ++i) {
            for (int j = 0; j < d; ++j) {
                res[i][j] = 0;
                for (int k = 0; k < d; ++k) {
                    res[i][j] = (res[i][j] + m[i][k] * oth[k][j] % mod + mod) %
                        mod;
                }
            }
        }
        return res;
    }

    M exp(i64 e) {
        M res;
        for (int i = 0; i < d; ++i)
            for (int j = 0; j < d; ++j)
                res[i][j] = (i==j);
        M base = *this;
        while (e > 0) {
            if (e & 1LL) res = res * base;
            base = base * base;
            e >>= 1LL;
        }
        return res;
    }
};

```

4.5 Array Permutation

```

#include <bits/stdc++.h>
using namespace std;
using i64 = long long int;

const int mod = 1e9+7;
const int D = 15;

int d = D;

struct M {
    i64 m[D][D];

    i64* operator[](int i) {
        return m[i];
    }

    M operator-(M oth) {
        M res;
        for (int i = 0; i < d; ++i) {
            for (int j = 0; j < d; ++j) {
                res[i][j] = m[i][j] - oth[i][j];
            }
        }
        return res;
    }

    M operator+(M oth) {
        M res;
        for (int i = 0; i < d; ++i) {
            for (int j = 0; j < d; ++j) {
                res[i][j] = m[i][j] + oth[i][j];
            }
        }
        return res;
    }

    M operator*(M oth) {
        M res;
        for (int i = 0; i < d; ++i) {
            for (int j = 0; j < d; ++j) {
                res[i][j] = m[i][j] + oth[i][j];
            }
        }
        return res;
    }

    M operator*(M oth) {
        M res;
        for (int i = 0; i < d; ++i) {
            for (int j = 0; j < d; ++j) {
                res[i][j] = 0;
                for (int k = 0; k < d; ++k) {
                    res[i][j] = (res[i][j] + m[i][k] * oth[k][j] % mod + mod) %
                        mod;
                }
            }
        }
        return res;
    }

    M exp(i64 e) {
        M res;
        for (int i = 0; i < d; ++i)
            for (int j = 0; j < d; ++j)
                res[i][j] = (i==j);
        M base = *this;
        while (e > 0) {
            if (e & 1LL) res = res * base;

```

```

        base = base * base;
        e >>= 1LL;
    }
    return res;
}
};

// O(n^(3)*log(k))
void apply_permutation(vector<int> &seq, vector<int> &perm, int k) {
    d = perm.size();
    M base;
    for (int i = 0; i < d; ++i) {
        for (int j = 0; j < d; ++j) {
            base[i][j] = 0;
        }
    }
    for (int i = 0; i < d; ++i) {
        base[i][perm[i]-1] = 1;
    }
    base = base.exp(k);
    vector<int> ans(d, 0);
    for (int i = 0; i < d; ++i) {
        for (int j = 0; j < d; ++j) {
            ans[i] += seq[j] * base[i][j];
        }
    }
    for (int i = 0; i < d; ++i) {
        seq[i] = ans[i];
    }
}

// O(n)
int dfs(int u, vector<vector<int>> &gr, vector<bool> &used, vector<int>
    > &order) {
    int rs = 1;
    order.push_back(u);
    used[u] = true;
    if (!used[gr[u][0]]) {
        rs += dfs(gr[u][0], gr, used, order);
    }
    return rs;
}

// O(n)
void apply_permutation_with_graph(vector<int> &seq, vector<int> &perm,
    int k) {
    int n = seq.size();
    vector<vector<int>> gr(n+1);
    for (int i = 0; i < n; ++i) {
        gr[perm[i]].push_back(i+1);
    }

    vector<bool> used(n+1, false);
    vector<int> ans(n+1);
    vector<int> order;

    for (int i = 1; i <= n; ++i) {
        if (!used[i]) {
            order.clear();
            int sz = dfs(i, gr, used, order);
            int pos = k % sz;

```

```

        for (int j = 0; j < sz; ++j) {
            int u = order[j];
            int to = order[(j+pos)%sz];
            ans[to-1] = seq[u-1];
        }
    }

    for (int i = 0; i < n; ++i) {
        seq[i] = ans[i];
    }
}

```

4.6 Modular Arithmetic

```

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

const int64_t MOD = 1e9+7;

inline int64_t modSum(int64_t a, int64_t b) {
    return (a+b >= MOD ? a+b-MOD : a+b);
}

inline int64_t modSub(int64_t a, int64_t b) {
    return (a+b < 0 ? a-b+MOD : a-b);
}

inline int64_t modMul(int64_t a, int64_t b) {
    return (a*1LL*b)%MOD;
}

int64_t inv_mod(int64_t a, int64_t mod = MOD) {
    int64_t x, y;
    extended_gcd(a, mod, x, y);
    return (x%mod + mod)%mod;
}

int64_t modDiv(int64_t a, int64_t b) {
    return modMul(a, inv_mod(b, MOD));
}

/*
    O(log(a))
*/
int64_t bigModMul(int64_t a, int64_t b) {
    int64_t ans = 0LL;
    b %= MOD;
    while (a > 0) {
        if (a & 1) ans = modAdd(ans, b, MOD);
        b = modMul(b, 2LL, MOD);
        a >>= 1;
    }
    return ans;
}

uint64_t bigModMul_2(uint64_t a, uint64_t b) {
    long double x;
    uint64_t c;
    int64_t r;
    if (a >= MOD) a %= MOD;
    if (b >= MOD) b %= MOD;
    x = a;
    c = (x * b) / MOD;
    r = (int64_t) (a * b - c * MOD) % (int64_t)MOD;
}

```

```
    return (r < 0 ? r+MOD : r);
}
```

4.7 Extended Euclidean

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

int64_t extended_gcd(int64_t a, int64_t b, int64_t &x, int64_t &y) {
    if (b == 0) {
        x = 1; y = 0;
        return a;
    }
    int64_t g = extended_gcd(b, a%b, y, x);
    y -= x*(a/b);
    return g;
}
```

4.8 Geometric Operations

```
#include <bits/stdc++.h>
using namespace std;
typedef long long int i64;
typedef double ftype;

const int mod = 1e9+7;
const int D = 4;

int d = D;

struct M {
    ftype m[D][D];

    ftype* operator[](int i) {
        return m[i];
    }

    M operator-(M oth) {
        M res;
        for (int i = 0; i < d; ++i) {
            for (int j = 0; j < d; ++j) {
                res[i][j] = m[i][j] - oth[i][j];
            }
        }
        return res;
    }

    M operator+(M oth) {
        M res;
        for (int i = 0; i < d; ++i) {
            for (int j = 0; j < d; ++j) {
                res[i][j] = m[i][j] + oth[i][j];
            }
        }
        return res;
    }

    M operator*(M oth) {
```

```
        M res;
        for (int i = 0; i < d; ++i) {
            for (int j = 0; j < d; ++j) {
                res[i][j] = 0;
                for (int k = 0; k < d; ++k) {
                    // res[i][j] = (res[i][j] + m[i][k] * oth[k][j] % mod + mod)
                    // % mod;
                    res[i][j] += m[i][k] * oth[k][j];
                }
            }
        }
        return res;
    }

    M exp(i64 e) {
        M res;
        for (int i = 0; i < d; ++i)
            for (int j = 0; j < d; ++j)
                res[i][j] = (i==j);
        M base = *this;
        while (e > 0) {
            if (e & 1LL) res = res * base;
            base = base * base;
            e >>= 1LL;
        }
        return res;
    }
};

struct operation {
    int op, a, b, c;
    double ang;
};

void geometric_operations(int &x, int &y, int &z, vector<pair<
    operation, int>> &v) {
    int m = v.size();
    M bases[m];
    for (int i = 0; i < m; ++i) {
        operation op = v[i].first;
        int k = v[i].second;
        if (op.op == 1) { //Shift operation
            M base;
            for (int j = 0; j < 4; ++j)
                for (int k = 0; k < 4; ++k)
                    base[j][k] = (j == k);

            base[3][0] = op.a;
            base[3][1] = op.b;
            base[3][2] = op.c;
            bases[i] = base.exp(k);
        } else if (op.op == 2) { //Scaling operation
            M base;
            for (int j = 0; j < 4; ++j)
                for (int k = 0; k < 4; ++k)
                    base[j][k] = 0;

            base[0][0] = op.a;
            base[1][1] = op.b;
            base[2][2] = op.c;
            base[3][3] = 1;
        }
    }
}
```

```

bases[i] = base.exp(k);
} else if (op.op == 3) { //Rotation operation around x
M base;
for (int j = 0; j < 4; ++j)
    for (int k = 0; k < 4; ++k)
        base[j][k] = 0;

base[0][0] = 1.0;
base[1][1] = +cos(op.ang); //cos(theta)
base[1][2] = -sin(op.ang); //sin(theta)
base[2][2] = +cos(op.ang); //cos(theta)
base[2][1] = +sin(op.ang); //sin(theta)
base[3][3] = 1.0;
bases[i] = base.exp(k);
}
}
for (int i = 1; i < m; ++i) {
    bases[i] = bases[i-1] * bases[i];
}
M ans = bases[m-1];
int _x = x, _y = y, _z = z;
x = _x * ans[0][0] + _y * ans[1][0] + _z * ans[2][0] + 1 * ans[3][0];
y = _x * ans[0][1] + _y * ans[1][1] + _z * ans[2][1] + 1 * ans[3][1];
z = _x * ans[0][2] + _y * ans[1][2] + _z * ans[2][2] + 1 * ans[3][2];
}

```

5 Geometry

5.1 Basic Geometry

```

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

#define eps 1e-9
#define eq(a, b) (abs(a - b) < eps)
#define lt(a, b) (a < b - eps)
#define gt(a, b) (a > b + eps)
#define le(a, b) (a < b + eps)
#define ge(a, b) (a > b - eps)
#define ftype long double
/*#define eq(a, b) (a == b)
#define lt(a, b) (a < b)
#define gt(a, b) (a > b)
#define le(a, b) (a <= b)
#define ge(a, b) (a >= b)
#define ftype long long*/

// Begin Point 2D
struct point2d {
    ftype x, y;

    point2d() : x(0.0), y(0.0) {}
    point2d(const ftype& x, const ftype& y) : x(x), y(y) {}

    point2d& operator=(const point2d& oth) {

```

```

        x = oth.x; y = oth.y;
        return (*this);
    }
    point2d& operator+=(const point2d& oth) {
        x += oth.x; y += oth.y;
        return (*this);
    }
    point2d& operator-=(const point2d& oth) {
        x -= oth.x; y -= oth.y;
        return (*this);
    }
    point2d& operator*=(const ftype& factor) {
        x *= factor; y *= factor;
        return (*this);
    }
    point2d& operator/=(const ftype& factor) {
        x /= factor; y /= factor;
        return (*this);
    }
};
point2d operator+(const point2d& a, const point2d& b) {
    return point2d(a.x + b.x, a.y + b.y);
}
point2d operator-(const point2d& a, const point2d& b) {
    return point2d(a.x - b.x, a.y - b.y);
}
point2d operator*(const point2d& a, const ftype& factor) {
    return point2d(a.x * factor, a.y * factor);
}
point2d operator*(const ftype& factor, const point2d& a) {
    return point2d(factor * a.x, factor * a.y);
}
point2d operator/(const point2d& a, const ftype& factor) {
    return point2d(a.x / factor, a.y / factor);
}
bool operator==(const point2d& a, const point2d& b) {
    return (eq(a.x, b.x) and eq(a.y, b.y));
}
bool operator!=(const point2d& a, const point2d& b) {
    return !(a==b);
}
bool operator < (const point2d& a, const point2d& b) {
    return (lt(a.x, b.x) or (eq(a.x, b.x) and lt(a.y, b.y)));
}
bool operator > (const point2d& a, const point2d& b) {
    return (b < a);
}
bool operator <= (const point2d& a, const point2d& b) {
    return !(a > b);
}
bool operator >= (const point2d& a, const point2d& b) {
    return !(a < b);
}
// > 0 if |angle| < pi/2
// = 0 if |angle| = pi
// < 0 if |angle| > pi/2
ftype operator*(const point2d& a, const point2d& b) {
    return (a.x * b.x + a.y * b.y);
}
// < 0 if a comes before b in ccw
// = 0 if a is collinear to b

```

```

// > 0 if a comes after b in ccw
ftype operator^(const point2d& a, const point2d& b) {
    return (a.x * b.y - a.y * b.x);
}
ftype ccw(const point2d& a, const point2d& b) {
    return (a ^ b);
}
// ccw(a, b, c) :> 0 if a comes before b counterclockwise in origin
// ccw(a, b, c) :< 0 if a comes after b counterclockwise in origin
ftype ccw(const point2d& a, const point2d& b, const point2d& origin) {
    return ccw(a - origin, b - origin);
}
ftype abs(const point2d& a) {
    return (a * a);
}
ftype norm(const point2d& a) {
    return sqrt(abs(a));
}
ftype dist(const point2d& a, const point2d& b) {
    return norm(a - b);
}
ftype dist2(const point2d& a, const point2d& b) {
    return abs(a - b);
}
ftype dist_point_to_line(const point2d& a, const point2d& p1, const
    point2d& p2) {
    return (a-p1)^(p2-p1)/norm(p2-p1);
}
ftype distance_segment_to_point(const point2d& p, const point2d& q,
    const point2d& a) {
    ftype l2 = dist2(p, q);
    if (eq(l2, 0)) return dist(p, a);
    ftype t = max((ftype)0, min((ftype)1, (a-p)*(q-p)/l2));
    point2d proj = p + t * (q-p);
    return dist(a, proj);
}
ftype proj(const point2d& a, const point2d& b) {
    return (a*b)/(b*b);
}
point2d pointProj(const point2d& a, const point2d& b) {
    return proj(a, b)*b;
}
ftype angle(const point2d& a) {
    return atan2(a.y, a.x);
}
ftype angle(const point2d& a, const point2d& b) {
    return atan2(a ^ b, a * b);
}
ftype angle(const point2d& a, const point2d& b, const point2d& origin) {
    return angle(a - origin, b - origin);
}
// Left rotation. Angle (rad)
point2d rotate(const point2d& a, const ftype& angleSin, const ftype&
    angleCos) {
    return point2d(a.x * angleCos - a.y * angleSin, a.x * angleSin + a
        .y * angleCos);
}
point2d rotate(const point2d& a, const ftype& angle) {
    return rotate(a, sin(angle), cos(angle));
}

```

```

// Pi/2 left rotation
point2d perp(const point2d& a) {
    return point2d(-a.y, a.x);
}
// 0 to 1 and 2 quadrant. 1 to 3 and 4
int half(const point2d& p) {
    if (gt(p.y, 0) or (eq(p.y, 0) and ge(p.x, 0))) return 0;
    return 1;
}
// angle(a) < angle(b)
bool cmpByAngle(const point2d& a, const point2d& b) {
    int ha = half(a), hb = half(b);
    if (ha != hb) return ha < hb;
    ftype c = a^b;
    if (eq(c, 0)) return lt(norm(a), norm(b));
    return gt(c, 0);
}
inline int sgn(ftype x) {
    return (ge(x, 0) ? (eq(x, 0) ? 0 : 1) : -1);
}
// Intersection of lines r : a + d1 * t
point2d intersect(const point2d& a1, const point2d& d1, const point2d&
    a2, const point2d& d2) {
    return a1 + ((a2-a1)^d2)/(d1^d2) * d1;
}
ftype area(vector<point2d> &pts){
    ftype ret = 0.0;
    for (int i = 2; i < (int)pts.size(); i++) {
        ret += ccw(pts[i] - pts[0], pts[i - 1] - pts[0]);
    }
    return abs(ret * 0.5);
}
ftype signed_area_parallelogram(const point2d& a, const point2d& b,
    const point2d& c) {
    return ccw(a, b, c);
}
ftype triangle_area(const point2d& a, const point2d& b, const point2d&
    c) {
    return abs(signed_area_parallelogram(a, b, c) * 0.5);
}
bool point_in_triangle(const point2d& a, const point2d& b, const
    point2d& c, const point2d& p) {
    ftype s1 = abs(ccw(b, c, a));
    ftype s2 = abs(ccw(a, b, p)) + abs(ccw(b, c, p)) + abs(ccw(c, a, p
        ));
    return eq(s1, s2);
}
bool pointInSquare(const point2d& A, const point2d& B, const point2d&
    C, const point2d& D, const point2d& P) {
    ftype s1 = 2*abs((B-A)^(D-A));
    ftype s2 = abs((B-P)^(A-P)) + abs((C-P)^(B-P)) + abs((D-P)^(C-P))
        + abs((A-P)^(D-P));
    return eq(s1, s2);
}
bool between(ftype l, ftype r, ftype x) {
    return (le(min(l, r), x) and ge(max(l, r), x));
}
bool pointInSegment(const point2d& a, const point2d& b, const point2d&
    p) {
    if (!eq(ccw(a, b, p), 0.0)) return false;
    return between(a.x, b.x, p.x) and between(a.y, b.y, p.y);
}

```



```

}
ftype up2(ftype a) {
    return (ftype)a * a;
}
// End Point 2D

// Begin Line
ftype det(ftype a, ftype b, ftype c, ftype d){
    return a * d - b * c;
}
struct Line {
    ftype a, b, c;
    Line () {}
    Line (ftype a1, ftype b1, ftype c1) : a(a1), b(b1), c(c1) {
        normalize();
    }
    Line (const point2d& p1, const point2d& p2) {
        a = p1.y - p2.y;
        b = p2.x - p1.x;
        c = -a * p1.x - b * p1.y;
        normalize();
    }
    void normalize() {
        ftype z = sqrt(up2(a) + up2(b));
        if (!eq(z, 0)) { a /= z, b /= z, c /= z; }
        if (lt(a, 0.0) or (eq(a, 0.0) and lt(b, 0.0))) {
            a = -a;
            b = -b;
            c = -c;
        }
    }
};
bool intersection_point_of_lines(const Line& m, const Line& n, point2d
&res) {
    ftype zn = det(m.a, m.b, n.a, n.b);
    if (eq(zn, 0.0)) return false;
    res.x = -det(m.c, m.b, n.c, n.b) / zn;
    res.y = -det(m.a, m.c, n.a, n.c) / zn;
    return true;
}
bool parallel(const Line& m, const Line& n) {
    return eq(det(m.a, m.b, n.a, n.b), 0.0);
}
bool equivalent(const Line& m, const Line& n) {
    return eq(det(m.a, m.b, n.a, n.b), 0.0) and
        eq(det(m.a, m.c, n.a, n.c), 0.0) and
        eq(det(m.b, m.c, n.b, n.c), 0.0);
}
ftype dist(const Line& m, const point2d& p) {
    return abs(m.a * p.x + m.b * p.y + m.c) /
        sqrt(up2(m.a) + up2(m.b));
}
// End Line
// Begin Segment
struct Segment {
    point2d a, b;
    Segment () {}
    Segment (const point2d& a1, const point2d b1) : a(a1), b(b1) {}
};
bool inter1(ftype a, ftype b, ftype c, ftype d) {
    if (a > b) swap(a, b);

```

```

    if (c > d) swap(c, d);
    return le(max(a, c), min(b, d));
}
bool check_intersection(const Segment& s1, const Segment& s2) {
    point2d a = s1.a, b = s1.b, c = s2.a, d = s2.b;
    if (ccw(a, d, c) == 0 and ccw(b, d, c) == 0)
        return (inter1(a.x, b.x, c.x, d.x) and
            inter1(a.y, b.y, c.y, d.y));
    return sgn(ccw(b, c, a) != ccw(b, d, a) and
        ccw(d, a, c) != ccw(d, b, c));
}
bool intersection_point_of_segments(const Segment& s1, const Segment&
s2, Segment &ans) {
    point2d a = s1.a, b = s1.b, c = s2.a, d = s2.b;
    if (!inter1(a.x, b.x, c.x, d.x) or
        !inter1(a.y, b.y, c.y, d.y)) return false;
    Line m(a, b);
    Line n(c, d);
    if (parallel(m, n)) {
        if (!equivalent(m, n)) return false;
        if (b < a) swap(a, b);
        if (d < c) swap(c, d);
        ans = Segment(max(a, c), min(b, d));
        return true;
    } else {
        point2d p(0, 0);
        intersection_point_of_lines(m, n, p);
        ans = Segment(p, p);
        return between(a.x, b.x, p.x) and between(a.y, b.y, p.y) and
            between(c.x, d.x, p.x) and between(c.y, d.y, p.y);
    }
}
// End Segment

// Begin Circle
struct Circle {
    ftype x, y, r;
    Circle () {}
    Circle(ftype x1, ftype y1, ftype r1) : x(x1), y(y1), r(r1) {}
};
bool pointInCircle(const Circle& c, const point2d& p) {
    return ge(c.r, dist(point2d(c.x, c.y), p));
}
Circle circumCircle(const point2d& a, const point2d& b, const point2d&
c) {
    point2d u((b-a).y, -((b-a).x));
    point2d v((c-a).y, -((c-a).x));
    point2d n = (c-b) * 0.5;
    ftype t = (u^n) / (v^u);
    point2d ct = ((a+c) * 0.5) + (v * t);
    ftype r = dist(ct, a);
    return Circle(ct.x, ct.y, r);
}
Circle inCircle(const point2d& a, const point2d& b, const point2d& c)
{
    ftype m1 = dist(a, b);
    ftype m2 = dist(a, c);
    ftype m3 = dist(b, c);
    point2d ct = ((c * m1) + (b * m2) + a * m3) / (m1 + m2 + m3);
    ftype sp = 0.5 * (m1 + m2 + m3);
    ftype r = sqrt(sp * (sp - m1) * (sp - m2) * (sp - m3)) / sp;

```

```

    return Circle(ct.x, ct.y, r);
}
// Minimum enclosing circle
Circle minimumCircle(vector<point2d> p) {
    random_shuffle(p.begin(), p.end());
    Circle c = Circle(p[0].x, p[0].y, 0.0);
    for (int i = 0; i < (int)p.size(); ++i) {
        if (pointInCircle(c, p[i])) continue;
        c = Circle(p[i].x, p[i].y, 0.0);
        for (int j = 0; j < i; ++j) {
            if (pointInCircle(c, p[j])) continue;
            c = Circle((p[j].x + p[i].x)*0.5, (p[j].y + p[i].y)*0.5,
                0.5*dist(p[j], p[i]));
            for (int k = 0; k < j; ++k) {
                if (pointInCircle(c, p[k])) continue;
                c = circumCicle(p[j], p[i], p[k]);
            }
        }
    }
    return c;
}
int circle_line_intersection(const Circle& circ, const Line& line,
    point2d& p1, point2d& p2) {
    ftype r = circ.r;
    ftype a = line.a, b = line.b, c = line.c + line.a * circ.x + line.
        b * circ.y; //take a circle to the (0, 0)
    ftype x0 = -a * c / (up2(a) + up2(b)), y0 = -b * c / (up2(a) + up2
        (b)); // (x0, y0) is the shortest distance point of the
        line for (0, 0)
    if (gt(up2(c), up2(r) * (up2(a) + up2(b)))) return 0;
    if (eq(up2(c), up2(r) * (up2(a) + up2(b)))) {
        p1.x = p2.x = x0 + circ.x;
        p1.y = p2.y = y0 + circ.y;
        return 1;
    } else {
        ftype d_2 = up2(r) - up2(c) / (up2(a) + up2(b));
        ftype mult = sqrt(d_2 / (up2(a) + up2(b)));
        p1.x = x0 + b * mult + circ.x;
        p2.x = x0 - b * mult + circ.x;
        p1.y = y0 - a * mult + circ.y;
        p2.y = y0 + a * mult + circ.y;
        return 2;
    }
}
int circle_intersection(const Circle& c1, const Circle& c2, point2d&
    p1, point2d& p2) {
    if (eq(c1.x, c2.x) and eq(c1.y, c2.y)) {
        if (eq(c1.r, c2.r)) return -1; //INF
        else return 0;
    } else {
        Circle circ(0, 0, c1.r);
        Line line;
        line.a = -2 * (c2.x - c1.x);
        line.b = -2 * (c2.y - c1.y);
        line.c = up2(c2.x - c1.x) + up2(c2.y - c1.y) + up2(c1.r) - up2
            (c2.r);
        int sz = circle_line_intersection(circ, line, p1, p2);
        p1.x += c1.x;
        p2.x += c1.x;
        p1.y += c1.y;
        p2.y += c1.y;
    }
}

```

```

    return sz;
}
bool check_segment_covered_by_circles(const vector<Circle> &vc, const
    Segment& s) {
    vector<point2d> v = {s.a, s.b};
    Line l(s.a, s.b);
    for (Circle c : vc) {
        point2d p1, p2;
        int inter = circle_line_intersection(c, l, p1, p2);
        if (inter >= 1 and between(s.a.x, s.b.x, p1.x) and between(s.a
            .y, s.b.y, p1.y))
            v.push_back(p1);
        if (inter == 2 and between(s.a.x, s.b.x, p2.x) and between(s.a
            .y, s.b.y, p2.y))
            v.push_back(p2);
    }
    sort(v.begin(), v.end());
    bool ans = true;
    for (int i = 1; i < (int)v.size(); i++) {
        bool has = false;
        for (Circle c : vc) {
            if (pointInCircle(c, v[i - 1]) and pointInCircle(c, v[i]))
                {
                    has = true;
                    break;
                }
        }
        ans &= has;
    }
    return ans;
}
void tangents(const point2d& c, double r1, double r2, vector<Line> &
    ans) {
    double r = r2 - r1;
    double z = up2(c.x) + up2(c.y);
    double d = z - up2(r);
    if (lt(d, 0)) return;
    d = sqrt(abs(d));
    Line l;
    l.a = (c.x * r + c.y * d) / z;
    l.a = (c.y * r + c.x * d) / z;
    l.c = r1;
    ans.push_back(l);
}
vector<Line> tangents(const Circle& a, const Circle& b) {
    vector<Line> ans;
    for (int i = -1; i <= 1; i += 2)
        for (int j = -1; j <= 1; j += 2)
            tangents(point2d(b.x - a.x, b.y - a.y), a.r * i, b.r * j,
                ans);
    for (int i = 0; i < (int)ans.size(); ++i) {
        ans[i].c -= ans[i].a * a.x + ans[i].b * a.y;
        ans[i].normalize();
    }
    return ans;
}
// End Circle

```

5.2 Convex Hull

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

vector<point2d> convex_hull(vector<point2d> a){
    if (a.size() == 1) return a;
    sort(a.begin(), a.end());
    a.erase(unique(a.begin(), a.end()), a.end());

    vector<point2d> up, down;
    point2d p1 = a[0], p2 = a.back();
    up.push_back(p1);
    down.push_back(p1);

    for (int i = 1; i < (int)a.size(); i++) {
        if ((i == (int)a.size() - 1) or ge(ccw(p2, a[i], p1), 0)) { //
            Accept collinear points
            // if ((i == (int)a.size() - 1) or gt(ccw(p2, a[i], p1), 0)) {
            // Don't accept collinear points
            while (up.size() >= 2 and lt(ccw(a[i], up.back(), up[up.
                size()-2]), 0)) up.pop_back(); // Accept collinear
            points
            // while (up.size() >= 2 and le(ccw(a[i], up.back(), up[up.
                size()-2]), 0)) up.pop_back(); // Don't accept
            collinear points
            up.push_back(a[i]);
        }
        if ((i == (int)a.size() - 1) or ge(ccw(a[i], p2, p1), 0)) { //
            Accept collinear points
            // if ((i == (int)a.size() - 1) or gt(ccw(a[i], p2, p1), 0)) {
            // Don't accept collinear points
            while (down.size() >= 2 and gt(ccw(a[i], down.back(), down.
                [down.size()-2]), 0)) down.pop_back(); // Accept
            collinear points
            // while (down.size() >= 2 and ge(ccw(a[i], down.back(),
                down[down.size()-2]), 0)) down.pop_back(); // Don't
            accept collinear points
            down.push_back(a[i]);
        }
    }

    a.clear();
    for (int i = 0; i < (int)up.size(); i++) a.push_back(up[i]);
    for (int i = (int)down.size()-2; i >= 1; i--) a.push_back(down[i]);
    return a;
}
```

5.3 Convex Polygon

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

namespace ConvexPolygon {
    vector<point2d> vp;
    void init(const vector<point2d>& aux) {
        vp = convex_hull(aux);
    }
}
```

```
bool pointInPolygon(const point2d& point) {
    if (vp.size() < 3) return pointInSegment(vp[0], vp[1], point);

    if (!eq(ccw(vp[1], point, vp[0]), 0.0) and
        sgn(ccw(vp[1], point, vp[0])) != sgn(ccw(vp[1], vp.back(),
            , vp[0]))) return false;

    if (!eq(ccw(vp.back(), point, vp[0]), 0.0) and
        sgn(ccw(vp.back(), point, vp[0])) != sgn(ccw(vp.back(),
            vp[1], vp[0]))) return false;

    if (eq(ccw(vp[1], point, vp[0]), 0.0)) return ge(norm(vp[1]-vp[0]),
        norm(point-vp[0]));

    int pos = 1, l = 1, r = vp.size() - 2;
    while (l <= r) {
        int mid = (l + r) / 2;
        if (le(ccw(vp[mid], point, vp[0]), 0.0)) {
            pos = mid;
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }

    return point_in_triangle(vp[0], vp[pos], vp[pos+1], point);
};
```

6 String Algorithms

6.1 Prefix Function

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

/*
    p[i] is the length of the longest proper prefix of s[0..i]
    which is also a suffix of this string
    Run in O(|s|)
*/
vector<int> prefix_function(const string &s) {
    int n = s.size();
    vector<int> pi(n);
    for (int i = 1, j = 0; i < n; ++i) {
        while (j > 0 and s[i] != s[j]) j = pi[j-1];
        if (s[i] == s[j]) ++j;
        pi[i] = j;
    }
    return pi;
}

/*
    Returns a vector with the initial positions of
    all occurrences of s in t
    Using O(|s|) memory
    Run in O(|s|+|t|)
*/
```

```

*/
vector<int> kmp(const string &s, const string &t) {
    vector<int> p = prefix_function(s+'$'), match;
    for (int i = 0, j = 0; i < (int)t.size(); ++i) {
        while (j > 0 and s[j] != t[i]) j = p[j-1];
        if (s[j] == t[i]) ++j;
        if (j == (int)s.size()) match.push_back(i-j+1);
    }
    return match;
}

/*
ans[i] is the amount of occurrences of the prefix s[0..i] in s
*/
vector<int> prefix_occurrences(const string &s) {
    vector<int> pi = prefix_function(s);
    int n = pi.size();
    vector<int> ans(n+1);
    for (int i = 0; i < n; i++)
        ans[pi[i]]++;
    for (int i = n-1; i > 0; i--)
        ans[pi[i-1]] += ans[i];
    for (int i = 0; i <= n; i++)
        ans[i]++;
    return ans;
}

inline int getId(char c) {
    return c-'a';
}

/*
Run in O(26*|s|)
*/
struct autKMP {
    vector<vector<int>> nxt;
    autKMP (const string &s : nxt(26, vector<int>(s.size()+1))) {
        vector<int> p = prefix_function(s);
        nxt[getId(s[0])][0] = 1;
        for (char c = 0; c < 26; ++c) {
            for (int i = 1; i <= (int)s.size(); ++i) {
                nxt[c][i] = (getId(s[i-1]) == c ? i+1 : nxt[c][p[i-1]]);
            }
        }
    };

    /*
    Returns a vector with the initial positions of
    all occurrences of s in t
    Run in O(|t|)
    */
    vector<int> matching_aut(const string& s, const string& t) {
        auto aut = autKMP(s);
        vector<int> match;
        int at = 0;
        for (int i = 0; i < (int)t.size(); ++i) {
            at = aut.nxt[getId(t[i])][at];
            if (at == (int)s.size()) match.push_back(i-at+1);
        }
        return match;
    }
};

```

6.2 Z Function

```

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

/*
z[i] is the length of the largest common prefix
between s[0..n-1] and s[i..n-1]
*/
vector<int> z_function(const string &s) {
    int n = (int)s.size();
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r)
            z[i] = min(r - i + 1, z[i - l]);
        while (i + z[i] < n and s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > r)
            l = i, r = i + z[i] - 1;
    }
    return z;
}

```

6.3 String Hashing

```

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

/*
Small Primes:
31, 53
Large Primes:
(1e6+3), (1e8+7), 100003621, (1e9+7), (1e9+9), (1LL<<61)-1
*/
struct StringHashing {
    const uint64_t MOD = (1LL<<61)-1;
    const int base = 31;
    uint64_t modMul(uint64_t a, uint64_t b) {
        uint64_t l1 = (uint32_t)a, h1 = a>>32, l2 = (uint32_t)b, h2 = b
        >>32;
        uint64_t l = l1*l2, m = l1*h2 + l2*h1, h = h1*h2;
        uint64_t ret = (l&MOD) + (l>>61) + (h << 3) + (m >> 29) + ((m <<
        35) >> 3) + 1;
        ret = (ret & MOD) + (ret>>61);
        ret = (ret & MOD) + (ret>>61);
        return ret-1;
    }
    inline int getInt(char c) {
        return (c-'a'+1);
    }

    /*
    hs[i] = s[0]*p^(i) + s[1]*p^(i-1) + ... + s[i-1]*p + s[i]
    */
    vector<uint64_t> hs, p;
    StringHashing (const string &s) {

```

```

    int n = s.size();
    hs.resize(n); p.resize(n);
    p[0] = 1;
    hs[0] = getInt(s[0]);
    for (int i = 1; i < n; ++i) {
        p[i] = modMul(p[i-1], base);
        hs[i] = (modMul(hs[i-1], base) + getInt(s[i]))%MOD;
    }
}
/*
    hs[i..j] = hs[j] - hs[i-1] * p^(j-i+1)
*/
uint64_t getValue(int l, int r) {
    if (l > r) return -1;
    uint64_t res = hs[r];
    if (l > 0) res = (res + MOD - modMul(p[r-l+1], hs[l-1]))%MOD;
    return res;
}
};

struct StringHashingDoubleMod {
    const uint64_t MOD1 = 1e6+3;
    const uint64_t MOD2 = 1e8+7;
    const int base = 31;
    uint64_t modMul(uint64_t a, uint64_t b, const uint64_t &MOD) {
        uint64_t l1 = (uint32_t)a, h1 = a>>32, l2 = (uint32_t)b, h2 = b
            >>32;
        uint64_t l = l1*l2, m = l1*h2 + l2*h1, h = h1*h2;
        uint64_t ret = (l&MOD) + (l>>61) + (h << 3) + (m >> 29) + ((m <<
            35) >> 3) + 1;
        ret = (ret & MOD) + (ret>>61);
        ret = (ret & MOD) + (ret>>61);
        return ret-1;
    }
    int getInt(char c) {
        return (c-'a'+1);
    }
    vector<uint64_t> h1, h2, p1, p2;
    StringHashingDoubleMod (const string &s) {
        int n = s.size();
        h1.resize(n); h2.resize(n); p1.resize(n); p2.resize(n);
        p1[0] = 1;
        p2[0] = 1;
        h1[0] = getInt(s[0]);
        h2[0] = getInt(s[0]);
        for (int i = 1; i < n; ++i) {
            p1[i] = modMul(p1[i-1], base, MOD1);
            p2[i] = modMul(p2[i-1], base, MOD2);
            h1[i] = (modMul(h1[i-1], base, MOD1) + getInt(s[i]))%MOD1;
            h2[i] = (modMul(h2[i-1], base, MOD2) + getInt(s[i]))%MOD2;
        }
    }
    pair<uint64_t, uint64_t> getValue(int l, int r) {
        if (l > r) return {-1, -1};
        pair<uint64_t, uint64_t> res;
        res.first = h1[r];
        if (l > 0) res.first = (res.first + MOD1 - modMul(p1[r-l+1], h1[l-1], MOD1))%MOD1;
        res.second = h2[r];
        if (l > 0) res.second = (res.second + MOD2 - modMul(p2[r-l+1], h2[l-1], MOD2))%MOD2;
    }
};

```

```

        return res;
    }
};

```

6.4 Trie

```

#include <bits/stdc++.h>
using namespace std;
const int K = 26;
inline int getId(char c) {
    return c-'a';
}
namespace Trie {
    struct Vertex {
        int next[K];
        int leaf, count;
        Vertex () {
            fill(begin(next), end(next), -1);
            leaf = count = 0;
        }
    };
    vector<Vertex> trie;
    void init() {
        trie.clear();
        trie.emplace_back();
    }
    /*
        Insert a string in O(|s|)
    */
    void add(const string &s) {
        int v = 0;
        ++trie[v].count;
        for (char ch : s) {
            int c = getId(ch);
            if (trie[v].next[c] == -1) {
                trie[v].next[c] = trie.size();
                trie.emplace_back();
            }
            v = trie[v].next[c];
            ++trie[v].count;
        }
        ++trie[v].leaf;
    }
    /*
        Get amount of occurrences of s in O(|s|)
    */
    int countStr(const string &s) {
        int v = 0;
        for (char ch : s) {
            int c = getId(ch);
            if (trie[v].next[c] == -1) return 0;
            v = trie[v].next[c];
        }
        return trie[v].leaf;
    }
    /*
        Get amount of occurentes of prefix s in O(|s|)
    */
    int countPre(const string &s) {
        int v = 0;
    }
};

```

```

    for (char ch : s) {
        int c = getId(ch);
        if (trie[v].next[c] == -1) return 0;
        v = trie[v].next[c];
    }
    return trie[v].count;
}
/*
Remove a string s in O(|s|) and returns true if it's removed
*/
bool remove(const string &s) {
    vector<int> rm;
    int v = 0;
    rm.push_back(v);
    for (char ch : s) {
        int c = getId(ch);
        if (trie[v].next[c] == -1) return false;
        v = trie[v].next[c];
        rm.push_back(v);
    }
    if (trie[v].leaf > 0) {
        --trie[v].leaf;
        for (int x : rm) --trie[x].count;
        return true;
    }
    return false;
}
};

```

6.5 Trie Int

```

#include <bits/stdc++.h>
using namespace std;
const int K = 2;
const int SZ = 32;
namespace Trie {
    struct Vertex {
        int next[K];
        int val, pre;
        Vertex () {
            fill(begin(next), end(next), -1);
            pre = val = 0;
        }
    };
    vector<Vertex> trie;
    void build() {
        trie.clear();
        trie.emplace_back();
    }
    void add(int val) {
        int v = 0;
        ++trie[v].pre;
        for (int i = SZ-1; i >= 0; --i) {
            bool b = val & (1 << i);
            if (trie[v].next[b] == -1) {
                trie[v].next[b] = trie.size();
                trie.emplace_back();
            }
            v = trie[v].next[b];
            ++trie[v].pre;
        }
    }
}

```

```

    }
    trie[v].val = val;
}
int min_xor(int val) {
    int v = 0;
    for (int i = SZ-1; i >= 0; --i) {
        bool b = val & (1 << i);
        if (trie[v].next[b] != -1) {
            v = trie[v].next[b];
        } else {
            v = trie[v].next[b^1];
        }
    }
    return val ^ trie[v].val;
}
int max_xor(int val) {
    int v = 0;
    for (int i = SZ-1; i >= 0; --i) {
        bool b = val & (1 << i);
        if (trie[v].next[b^1] != -1) {
            v = trie[v].next[b^1];
        } else {
            v = trie[v].next[b];
        }
    }
    return val ^ trie[v].val;
}
};

```

6.6 Aho Corasick

```

#include <bits/stdc++.h>
using namespace std;
#define fi first
#define se second

typedef pair<int, int> ii;

const int K = 26;
inline int getId(char c) {
    return c - 'a';
}
namespace Aho {
    struct Vertex {
        int next[K], go[K];
        int suff_link = -1, end_link = -1;
        int leaf = -1, p = -1, sz, match = -1;
        char pch;
        Vertex(int p1 = -1, char ch = '$', int sz1 = 0) : p(p1), pch(ch),
            sz(sz1) {
            fill(begin(next), end(next), -1);
            fill(begin(go), end(go), -1);
        }
    };
    vector<Vertex> trie;
    inline void init() {
        trie.clear();
        trie.emplace_back();
    }
    int add_string(const string &s, int id = 1) {

```

```

int v = 0;
for (char ch : s) {
    int c = getId(ch);
    if (trie[v].next[c] == -1) {
        trie[v].next[c] = trie.size();
        trie.emplace_back(v, ch, trie[v].sz + 1);
    }
    v = trie[v].next[c];
}
trie[v].leaf = id;
return v;
}
int go(int v, char ch);
int get_suff_link(int v) {
    if (trie[v].suff_link == -1) {
        if (v == 0 or trie[v].p == 0) {
            trie[v].suff_link = 0;
        } else {
            trie[v].suff_link = go(get_suff_link(trie[v].p), trie[v].pch);
        }
    }
    return trie[v].suff_link;
}
int get_end_link(int v) {
    if (trie[v].end_link == -1) {
        if (v == 0 or trie[v].p == 0) {
            trie[v].end_link = 0;
        } else {
            int suff_link = get_suff_link(v);
            if (trie[suff_link].leaf != -1) {
                trie[v].end_link = suff_link;
            } else {
                trie[v].end_link = get_end_link(suff_link);
            }
        }
    }
    return trie[v].end_link;
}
int go(int v, char ch) {
    int c = getId(ch);
    if (trie[v].go[c] == -1) {
        if (trie[v].next[c] != -1) {
            trie[v].go[c] = trie[v].next[c];
        } else {
            trie[v].go[c] = (v == 0 ? 0 : go(get_suff_link(v), ch));
        }
    }
    return trie[v].go[c];
}
}
};

/*
Get match positions in O(|t| * sqrt(|t|))
Answer: {i, j} -> Range of match
*/
vector<ii> getMatch(const string &t) {
    auto addMatch = [&](vector<ii> &ans, int v, int i) {
        while (v != 0) {
            ans.emplace_back(i - Aho::trie[v].sz + 1, i);
            v = Aho::get_end_link(v);
        }
    };

```

```

};
int v = 0;
vector<ii> ans;
for (int i = 0; i < (int)t.size(); ++i) {
    v = Aho::go(v, t[i]);
    if (Aho::trie[v].leaf != -1) {
        addMatch(ans, v, i);
    } else {
        addMatch(ans, Aho::get_end_link(v), i);
    }
}
sort(ans.begin(), ans.end());
return ans;
}

int countMatch(int v) {
    if (Aho::trie[v].match == -1) {
        if (v == 0 or Aho::trie[v].p == 0) {
            Aho::trie[v].match = (Aho::trie[v].leaf != -1 ? 1 : 0);
        } else {
            Aho::trie[v].match = (Aho::trie[v].leaf != -1 ? 1 : 0) +
                countMatch(Aho::get_end_link(v));
        }
    }
    return Aho::trie[v].match;
}

/*
Get match amount in O(|t|)
Answer: Amount of matches
*/
int64_t matchAmount(const string &t) {
    int v = 0;
    int64_t ans = 0;
    for (char ch : t) {
        v = Aho::go(v, ch);
        ans += countMatch(v);
    }
    return ans;
}

```

6.7 Suffix Array

```

#include <numeric>
#include <vector>
#include <string>
typedef std::pair<int, int> ii;
class SuffixArray {
    std::vector<int> RA, SA, tempRA, tempSA, c;
    std::vector<int> LCP, Phi, PLCP;
    std::string S;
    int N;
    void countingSort(int k) {
        int sum = 0, maxi = std::max(256, N);
        c.assign(maxi, 0);
        for (int i = 0; i < N; ++i)
            c[RA[(i+k)%N]]++;
        for (int i = 1; i < maxi; ++i)
            c[i] += c[i-1];
        for (int i = N-1; i >= 0; --i)

```

```

    tempSA[--c[RA[(SA[i]+k)%N]]] = SA[i];
    SA = tempSA;
}
void constructSA() {
    iota(SA.begin(), SA.end(), 0);
    for (int i = 0; i < N; ++i) RA[i] = S[i];
    for (int k = 1; k < N; k <= 1) {
        countingSort(k);
        countingSort(0);
        int r = 0;
        tempRA[SA[0]] = 0;
        for (int i = 1; i < N; ++i) {
            tempRA[SA[i]] =
                (ii(RA[SA[i]], RA[(SA[i]+k)%N]) == ii(RA[SA[i-1]], RA[(SA[i-1]+k)%N]) ? r : ++r);
        }
        RA = tempRA;
        if (RA[SA[N-1]] == N-1) break;
    }
}
void constructLCP() {
    Phi[SA[0]] = -1;
    for (int i = 1; i < N; ++i) {
        Phi[SA[i]] = SA[i-1];
    }
    for (int i = 0, k = 0; i < N; ++i) {
        if (Phi[i] == -1) { PLCP[i] = 0; continue; }
        while (S[i+k] == S[Phi[i]+k]) ++k;
        PLCP[i] = k;
        k = std::max(k-1, 0);
    }
    for (int i = 0; i < N; ++i)
        LCP[i] = PLCP[SA[i]];
}
ii stringMatching(const std::string &s) {
    int m = s.size();
    int lo = 0, hi = N-2, mid;
    while (lo < hi) { /*Find the lower bound*/
        mid = lo+(hi-lo)/2;
        if (S.substr(SA[mid], m) >= s) hi = mid;
        else lo = mid + 1;
    }
    if (S.substr(SA[lo], m) != s) return {-1, -1};
    ii ans = {lo, lo};
    lo = 0, hi = N-2;
    while (lo < hi) { /*Find the upper bound*/
        mid = lo+(hi-lo)/2;
        if (S.substr(SA[mid], m) > s) hi = mid;
        else lo = mid + 1;
    }
    if (S.substr(SA[lo], m) != s) --hi;
    ans.second = hi;
    return ans;
}
public:
SuffixArray (const std::string &s) : S(s) {
    S += '$';
    N = S.size();
    RA.assign(N, 0);
    SA.assign(N, 0);
    tempSA.assign(N, 0);

```

```

    tempRA.assign(N, 0);
    LCP.assign(N, 0);
    PLCP.assign(N, 0);
    Phi.assign(N, 0);
    constructSA();
    constructLCP();
    SA.erase(SA.begin());
    LCP.erase(LCP.begin());
}
std::vector<int> getSA() {
    return SA;
}
std::vector<int> getLCP() {
    return LCP;
}
ii getStringMatching(const std::string &s) {
    return stringMatching(s);
}
/*
    Number of different substrings:
    (n^2+n)/2 - sum_{i=0 to n-2} lcp[i]
*/
};

```

6.8 Manacher

```

#include <vector>
#include <string>
struct Palindrome {
    std::vector<int> d1, d2;
    int N;
    void manacher(const std::string &s) {
        int l, r = -1;
        N = s.size();
        d1.resize(N), d2.resize(N);
        for (int i = 0; i < N; ++i) {
            int k = i > r ? 1 : std::min(d1[l+(r-i)], r-i+1);
            while (k <= i and i + k < N and s[i-k] == s[i+k])
                ++k;
            d1[i] = k--;
            if (i+k>r) l = i-k, r=i+k;
        }
        l = 0, r = -1;
        for (int i = 0; i < N; ++i) {
            int k = i > r ? 0 : std::min(d2[l+(r-i)+1], r-i+1);
            while (k+1 <= i and i + k < N and s[i-k-1] == s[i+k])
                ++k;
            d2[i] = k--;
            if (i+k>r) l = i-k-1, r=i+k;
        }
    }
    Palindrome (const std::string &s) {
        manacher(s);
    }
    bool isPalindrome(int i, int j) {
        int sz = j-i+1;
        return (sz & 1 ? d1[i+sz/2] >= sz : d2[i+sz/2+1] >= sz);
    }
};

```


6.9 Lyndon Factorization

```
#include <string>
#include <vector>
std::vector<std::string> duval(const std::string &s) {
    int n = s.size();
    std::vector<std::string> fac;
    for (int i = 0; i < n; i++) {
        int j = i + 1, k = i;
        while (j < n and s[k] <= s[j]) {
            if (s[k] < s[j])
                k = i;
            else
                ++k;
            ++j;
        }
        while (i <= k) {
            fac.push_back(s.substr(i, j-k));
            i += j-k;
        }
    }
    return fac;
}

std::string min_cyclic_string(std::string s) {
    s += s;
    int n = s.size();
    int ans = 0;
    for (int i = 0; i < n/2; i++) {
        ans = i;
        int j = i+1, k = i;
        while (s[k] <= s[j]) {
            if (s[k] < s[j])
                k = i;
            else
                ++k;
            ++j;
        }
        while (i <= k)
            i += j-k;
    }
    return s.substr(ans, n/2);
}
```

7 Miscellaneous

7.1 Ternary Search

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

#define ftype long double
#define f(x) x

const int inf = 0x3f3f3f3f;

void ternary_search_of_min(ftype top) {
    ftype lo = 0.0, hi = top, m1, m2, ans;
```

```
    for (int i = 0; i < 100; ++i) {
        m1 = (lo * 2 + hi) / 3.0;
        m2 = (lo + 2 * hi) / 3.0;
        if (f(m1) > f(m2)) {
            lo = m1;
            ans = m2;
        } else {
            hi = m2;
            ans = m1;
        }
    }
    cout << f(ans) << " = " << ans << '\n';
}
```

```
void ternary_search_of_max(ftype top) {
    ftype lo = 0.0, hi = top, m1, m2, ans;
    for (int i = 0; i < 100; ++i) {
        m1 = (lo * 2 + hi) / 3.0;
        m2 = (lo + 2 * hi) / 3.0;
        if (f(m1) < f(m2)) {
            lo = m1;
            ans = m2;
        } else {
            hi = m2;
            ans = m1;
        }
    }
    cout << f(ans) << " = " << ans << '\n';
}
```

```
void ternary_search_of_min_on_integers(int top) {
    int lo = 0, hi = top, ans = inf, m1, m2;
    while (hi - lo > 4) {
        int m1 = (lo + hi) / 2;
        int m2 = m1 + 1;
        if (f(m1) > f(m2)) {
            lo = m1;
        } else {
            hi = m2;
        }
    }
    for (int i = lo; i <= hi; ++i) {
        ans = min(ans, f(i));
    }
}
```

```
void ternary_search_of_max_on_integers(int top) {
    int lo = 0, hi = top, ans = -inf, m1, m2;
    while (hi - lo > 4) {
        int m1 = (lo + hi) / 2;
        int m2 = m1 + 1;
        if (f(m1) < f(m2)) {
            lo = m1;
        } else {
            hi = m2;
        }
    }
    for (int i = lo; i <= hi; ++i) {
        ans = max(ans, f(i));
    }
}
```

7.2 Longest Increasing Subsequence

```
#include <vector>
int lis(std::vector<int> &aux) {
    std::vector<int> d;
    for (int &x : aux) {
        auto it = std::lower_bound(d.begin(), d.end(), x);
        if (it == d.end()) d.push_back(x);
        else *it = x;
    }
    return (int)d.size();
}
```

7.3 Mo Algorithm

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

typedef vector<int> vi;

const int BLOCK_SIZE = 800;
const int maxn = 5e5+5;

int v[maxn], f[maxn];
int ans;

void remove(int idx) {
    --f[v[idx]];
    if (f[v[idx]] == 0) --ans;
}

void add(int idx) {
    ++f[v[idx]];
    if (f[v[idx]] == 1) ++ans;
}

int get_answer() {
    return ans;
}

struct Query {
    int l, r, idx;
    bool operator < (const Query oth) const {
        if (l / BLOCK_SIZE != oth.l / BLOCK_SIZE) return l < oth.l;
        return (l / BLOCK_SIZE & 1) ? (r < oth.r) : (r > oth.r);
    }
}
```

```
};

vi mo_s_algorithm(vector<Query> queries) {
    vi answers(queries.size());
    sort(queries.begin(), queries.end());
    int l = 0, r = 0;
    for (Query q : queries) {
        while (q.l < l) add(--l);
        while (r < q.r) add(++r);
        while (l < q.l) remove(l++);
        while (q.r < r) remove(r--);
        answers[q.idx] = get_answer();
    }
    return answers;
}
```

7.4 Inversions Count

```
#include <vector>
int mergeSort(std::vector<int> &a) {
    int n = a.size();
    if (n <= 1) return 0;
    int mid = n/2;
    std::vector<int> b, c;
    for (int i = 0; i < mid; ++i)
        b.push_back(a[i]);
    for (int i = mid; i < n; ++i)
        c.push_back(a[i]);
    int inv = 0;
    inv += mergeSort(b);
    inv += mergeSort(c);
    int i = 0, j = 0;
    for (int k = 0; k < n; ++k) {
        if (i == mid) {
            a[k] = c[j++];
        } else if (j == n-mid) {
            a[k] = b[i++];
        } else if (b[i] <= c[j]) {
            a[k] = b[i++];
        } else {
            a[k] = c[j++];
            inv += mid-i;
        }
    }
    return inv;
}
```