Rio de Janeiro State University ICPC Team Codebook (2022)

Contents

1 Data Structures

1.1 BIT

```cpp
// Fenwick Tree | Binary Indexed Tree
// Complexity:- update: O(log(n)); query: O(log(n))
struct BIT {
    vector<int> tree;
    int LOGN = 25;
    int n;

    BIT(int n) : n(n) {
        tree.resize(n);
        LOGN = (int)ceil(log2(n));
    }

    void update(int idx, int val) {
        for (int i = idx; i <= n; i += (i & -i)) {
            tree[i] += val;
        }
    }

    int query(int idx) {
        int sum = 0;

        for (int i = idx; i > 0; i -= (i & -i)) {
            sum += tree[i];
        }

        return sum;
    }

    int query(int l, int r) {
        return query(r) - query(l - 1);
    }

    int lowerBound(int val) {
        int sum = 0, pos = 0;

        for (int i = LOGN; i >= 0; i--) {
            if (pos + (1 << i) < n and sum + tree[pos + (1 << i)] < val) {
                sum += tree[pos + (1 << i)];
                pos += (1 << i);
            }
        }

        return pos + 1;
    }
};
```

1.2 BIT 2D

```cpp
// Complexidade:- update -> O(logN)
//              - query  -> O(logN)

#define MAXN 1010

int bit[MAXN][MAXN], a[MAXN][MAXN], x, y;

void update(int idx, int idx2, int val){
    for(int i = idx; i <= x; i += i & -i){
        for(int j = idx2; j <= y; j += j & -j){
            bit[i][j] += val;
        }
    }
}

void query(int idx, int idx2){
    int sum = 0;
    for(int i = idx; i > 0; i -= i & -i){
        for(int j = idx2; j > 0; j -= j & -j){
            sum += bit[i][j];
        }
    }
    return sum;
}

void query(int xmin, int ymin, int xmax, int ymax){
    if(xmin > xmax) swap(xmin, xmax);
    if(ymin > ymax) swap(ymin, ymax);
    return query(xmax, ymax) - query(xmax, ymin - 1) - query(xmin - 1, ymax) + query(xmin - 1, ymin -
        1);
}
```

1.3 Color Update

```cpp
// Update -> Change all colors in range [left, right] to some color.
// Query -> How many indexes has some color.
struct ColorUpdate {
    struct Node {
        long long l, r;
        int color;

        Node(long long l, long long r, int color) : l(l), r(r), color(color) {}
    };

    struct Comparator {
        bool operator()(Node a, Node b) {
            if (a.r == b.r) {
                return a.l < b.l;
            }
            return a.r < b.r;
        }
    };

    set<Node, Comparator> st;
    vector<long long> ans;

    ColorUpdate(long long left, long long right, int max_color) {
        ans.resize(max_color + 1);
        ans[0] = right - left + 1LL;
        st.insert(Node(left, right, 0));
    }

    void update(long long left, long long right, int new_color){
        auto p = st.lower_bound(Node(0, left, -1));

        assert(p != st.end());

        long long l = p->l;
        long long r = p->r;
        int old_color = p->color;

        ans[old_color] -= (r - l + 1LL);
        p = st.erase(p);

        if (l < left) {
            ans[old_color] += (left - l);
            st.insert(Node(l, left - 1LL, old_color));
        }

        if (right < r){
            ans[old_color] += (r - right);
            st.insert(Node(right + 1LL, r, old_color));
        }

        while ((p != st.end()) && (p->l <= right)) {
            l = p->l;
            r = p->r;
            old_color = p->color;
            ans[old_color] -= (r - l + 1LL);

            if (right < r) {
                ans[old_color] += (r - right);
                st.insert(Node(right + 1LL, r, old_color));
                st.erase(p);
                break;
            } else {
                p = st.erase(p);
            }
        }

        ans[new_color] += (right - left + 1LL);
        st.insert(Node(left, right, new_color));
    }

    int getColor(long long pos) {
        auto p = st.lower_bound(Node(0, pos, -1));
        return p->color;
    }

    long long countColor(int color){
        return ans[color];
    }
};
```

1.4 Merge Sort

```cpp
// Complexity: O(n * log(n))
int merge_sort(vector<int> &v){
    int inv = 0;

    if (v.size() == 1) return 0;

    vector<int> m1, m2;
    for (int i = 0; i < v.size()/2; i++) m1.push_back(v[i]);
    for (int i = v.size()/2; i < v.size(); i++) m2.push_back(v[i]);

    inv += merge_sort(m1);
    inv += merge_sort(m2);

    m1.push_back(INT_MAX);
    m2.push_back(INT_MAX);

    int idx = 0, idx2 = 0;
    for (int i = 0; i < v.size(); i++) {
        if (m1[idx] <= m2[idx2]) {
            v[i] = m1[idx++];
        } else {
            v[i] = m2[idx2++];
            inv += m1.size() - idx - 1;
        }
    }

    return inv;
}
```

1.5 Min Queue

```cpp
struct MinQueue {
    int sz = 0;
    int add_val = 0;
    deque<pair<int, int>> dq, aux;

    bool empty() { return dq.empty(); }

    int size() { return sz; }

    void clear() {
        add_val = 0;
        sz = 0;
        dq.clear();
    }

    void push(int x) {
        x -= add_val;
        int amt = 1;

        while (!dq.empty() && dq.back().first >= x) {
            amt += dq.back().second;
            dq.pop_back();
        }

        dq.push_back({ x, amt });
        sz++;
    }

    void pop() {
        dq.front().second--;
        sz--;

        if (!dq.front().second) {
            dq.pop_front();
        }
    }

    void addToEveryNumber(int val) { add_val += val; }

    int getMin() { return dq.front().first + add_val; }
};
```

1.6 Order Statistic Tree

```cpp
#include <ext/pb_ds/tree_policy.hpp>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/rope>
using namespace __gnu_pbds;
using namespace __gnu_cxx;
template <class T> using Tree = tree<T, null_type, less<T>, rb_tree_tag,
      tree_order_statistics_node_update>;
```

```cpp
Tree<int> ord_s;
ord_s.insert(2);
ord_s.insert(5);
ord_s.insert(7);
ord_s.insert(9);

// find_by_order returns an iterator to the element at a given position
auto x = ord_s.find_by_order(2);
cout << *x << "\n"; // 7

// order_of_key returns the position of a given element
cout << ord_s.order_of_key(7) << "\n"; // 2

// If the element does not appear in the set, we get the position that the element would have in the
      set
cout << ord_s.order_of_key(6) << "\n"; // 2
cout << ord_s.order_of_key(8) << "\n"; // 3
```

1.7 Segment Tree

```cpp
// build: O(n), update: O(log(n)), query: O(log(n))
// 0-indexed, query in the closed interval [l, r].
struct SegmentTree {
    struct Node {
        long long val;
    };

    int size;
    vector<Node> nodes;

    Node merge(Node a, Node b) {
        return { min(a.val, b.val) };
    }

    Node single(int v) {
        return { v };
    }

    SegmentTree(int n, vector<int> &v) {
        size = n;
        nodes.resize(4 * n);
        build(v, 0, 0, size);
    }

    void build(vector<int> &v, int ptr, int left, int right) {
        if (right == left) {
            if (left < (int) v.size()) {
                nodes[ptr] = single(v[left]);
            }
        } else {
            int mid = (right + left) / 2;

            build(v, 2 * ptr + 1, left, mid);
            build(v, 2 * ptr + 2, mid + 1, right);

            nodes[ptr] = merge(nodes[2 * ptr + 1], nodes[2 * ptr + 2]);
        }
    }

    void update(int idx, int val, int ptr, int left, int right) {
        if (right == left) {
            nodes[ptr] = single(val);
        } else {
            int mid = (right + left) / 2;

            if (idx <= mid) {
                update(idx, val, 2 * ptr + 1, left, mid);
            } else {
                update(idx, val, 2 * ptr + 2, mid + 1, right);
            }

            nodes[ptr] = merge(nodes[2 * ptr + 1], nodes[2 * ptr + 2]);
        }
    }

    void update(int idx, int val) {
        update(idx, val, 0, 0, size);
    }

    Node query(int l, int r, int ptr, int left, int right) {
        if (left >= l && right <= r) {
            return nodes[ptr];
        }

        int mid = (right + left) / 2;

        if (r <= mid) {
            return query(l, r, 2 * ptr + 1, left, mid);
        } else if (l > mid) {
            return query(l, r, 2 * ptr + 2, mid + 1, right);
        } else {
            Node qry1 = query(l, r, 2 * ptr + 1, left, mid);
            Node qry2 = query(l, r, 2 * ptr + 2, mid + 1, right);
            return merge(qry1, qry2);
        }
    }

    Node query(int l, int r) {
        return query(l, r, 0, 0, size);
    }
};
```

1.8 Segment Tree Lazy

```cpp
struct SegmentTreeLazy {
    struct Node {
        long long val;
    };

    int size;
    vector<Node> nodes, lazy;

    SegmentTreeLazy(int n, vector<long long> &v) {
        size = n;
        nodes.resize(4 * n);
        lazy.resize(4 * n, { -1 });
        build(v, 0, 0, size);
    }

    Node merge(Node a, Node b) {
        return { (a.val + b.val) };
    }

    Node single(long long v) {
        return { v };
    }

    void build(vector<long long> &v, int ptr, int left, int right){
        if (right == left) {
            if (left < (int) v.size()) {
                nodes[ptr] = single(v[left]);
            }
        } else {
            int mid = (right + left) / 2;

            build(v, 2 * ptr + 1, left, mid);
            build(v, 2 * ptr + 2, mid + 1, right);

            nodes[ptr] = merge(nodes[2 * ptr + 1], nodes[2*ptr + 2]);
        }
    }

    void propagate(int ptr, int left, int right){
        if (lazy[ptr].val != -1) {
            nodes[ptr].val = (right - left + 1) * lazy[ptr].val; // RSQ = (right - left + 1) * lazy[
                ptr].val

            if (left != right) {
                lazy[2 * ptr + 1].val = lazy[ptr].val;
                lazy[2 * ptr + 2].val = lazy[ptr].val;
            }

            lazy[ptr].val = -1;
        }
    }

    void update(int l, int r, int val, int ptr, int left, int right){
        propagate(ptr, left, right);

        if (right < l || left > r) {
            return;
        }

        if (left >= l && right <= r) {
            lazy[ptr].val = val;
            propagate(ptr, left, right);
        } else {
            int mid = (right + left) / 2;

            update(l, r, val, 2 * ptr + 1, left, mid);
            update(l, r, val, 2 * ptr + 2, mid + 1, right);

            nodes[ptr] = merge(nodes[2 * ptr + 1], nodes[2 * ptr + 2]);
        }
```

```
    }

    void update(int l, int r, int val) {
        update(l, r, val, 0, 0, size);
    }

    Node query(int l, int r, int ptr, int left, int right){
        propagate(ptr, left, right);

        if (left >= l && right <= r) {
            return nodes[ptr];
        }

        int mid = (right + left) / 2;

        if (r <= mid) {
            return query(l, r, 2 * ptr + 1, left, mid);
        } else if (l > mid) {
            return query(l, r, 2 * ptr + 2, mid + 1, right);
        } else {
            Node qry1 = query(l, r, 2 * ptr + 1, left, mid);
            Node qry2 = query(l, r, 2 * ptr + 2, mid + 1, right);

            return merge(qry1, qry2);
        }
    }

    Node query(int l, int r) {
        return query(l, r, 0, 0, size);
    }
};
```

1.9 Sparse Table

```
// Complexity: build: O(n * log(n)); query: O(1)
struct SparseTable {
    int n;
    vector<vector<int>> table;

    SparseTable(vector<int> &v) {
        n = (int)v.size();
        int max_log = 32 - __builtin_clz(n);
        table.resize(max_log);
        table[0] = v;

        for (int lg = 1; lg < max_log; lg++) {
            table[lg].resize(n - (1 << lg) + 1);
            for (int i = 0; i <= n - (1 << lg); i++) {
                table[lg][i] = min(table[lg - 1][i], table[lg - 1][i + (1 << (lg - 1))]);
            }
        }
    }

    int getRMQ(int left, int right) {
        assert(0 <= left && left <= right && right <= n - 1);
        int lg = 32 - __builtin_clz(right - left + 1) - 1;
        return min(table[lg][left], table[lg][right - (1 << lg) + 1]);
    }
};
```

1.10 Union Find

```
struct DSU {
    vector<int> parent, sz;

    DSU(int n) {
        parent.resize(n);
        sz.resize(n);

        iota(parent.begin(), parent.end(), 0);
    }

    int find(int u) {
        return (parent[u] == u ? u : parent[u] = find(parent[u]));
    }

    void join(int u, int v){
        u = find(u);
        v = find(v);

        if (u != v) {
            if (sz[u] < sz[v]) {
                swap(u, v);
            }

            parent[v] = u;
            sz[u] += sz[v];
        }
    }

    bool insideSameSet(int u, int v) {
        return find(u) == find(v);
    }

    int getSize(int u) {
        return sz[find(u)];
    }
};
```

2 Geometry

2.1 Basics

```
// returns -1 for negative numbers, 0 for zero, and 1 for positive numbers.
template <class T> int sgn(T x) {
        return (T(0) < x) - (x < T(0));
}

template <class T> struct Point2D {
        T x, y;

        Point2D(T x = 0, T y = 0) : x(x), y(y) {}

        bool operator < (Point2D p) { return tie(x,y) < tie(p.x, p.y); }
        bool operator == (Point2D p) { return tie(x,y) == tie(p.x, p.y); }
        bool operator != (Point2D p) { return !(tie(x,y) == tie(p.x, p.y)); }
        Point2D operator - (Point2D p) { return Point2D(x - p.x, y - p.y); }
        Point2D operator + (Point2D p) { return Point2D(x + p.x, y + p.y); }
        Point2D operator * (T k) { return Point2D(k * x, k * y); }
        Point2D operator / (T k) { return Point2D(x / k, y / k); }

        T dot(Point2D p) { return x * p.x + y * p.y; }
        T cross(Point2D p) { return x * p.y - y * p.x; }
        T cross(Point2D a, Point2D b) const { return (a - *this).cross(b - *this); }
        T dist2() const { return x * x + y * y; }
        double dist() const { return sqrt((double)dist2()); }
        double angle() const { return atan2(y, x); } // angle to x-axis in interval [-pi, pi]
        Point2D unit() const { return *this / dist(); } // makes dist()=1
        Point2D perp() const { return Point2D(-y, x); } // rotates +90 degrees
        Point2D normal() const { return perp().unit(); }

        // Returns point rotated 'a' radians ccw around the origin
        Point2D rotate(double a) const {
                return Point2D(x * cos(a) - y * sin(a), x * sin(a) + y * cos(a));
        }

        double distanceToPoint(Point2D b) {
                return sqrt(((x - b.x) * (x - b.x)) + ((y - b.y) * (y - b.y)));
        }

        // Returns true if this point lies on the line segment from point 's' to point'e'
        // Use (distanceToSegment(s, e) <= EPS) instead when using Point2D<double>.
        bool onSegment(Point2D s, Point2D e) {
                return (*this).cross(s, e) == 0 && (s - *this).dot(e - *this) <= 0;
        }

        // Returns the shortest distance between this point and the line segment from point s to e.
        double distanceToSegment(Point2D &s, Point2D &e) {
                if (s == e) {
                        return ((*this) - s).dist();
                }

                auto d = (e - s).dist2();
                auto t = min(d, max(.0,((*this) - s).dot(e - s)));

                return (((*this) - s) * d - (e - s) * t).dist() / d;
        }

        // Returns the signed perpendicular distance between point p and the line containing points a
              and b.
        // Positive value on left side and negative on right as seen from a towards b. a==b gives nan.
        double distanceToLine(Point2D &a, Point2D &b) {
                return (double)(b - a).cross((*this) - a) / (b - a).dist();
        }

        // Returns true if p lies within the polygon.
        // If strict is true, it returns false for points on the boundary.
        // The algorithm uses products in intermediate steps so watch out for overflow.
```

```
        bool isInsidePolygon(vector<Point2D> &polygon, bool strict = true) {
                int cnt = 0, n = int(polygon.size());

                for (int i = 0; i < n; i++) {
                        Point2D q = polygon[(i + 1) % n];

                        if (onSegment(polygon[i], q)) {
                                return !strict;
                        }
                        //or: if (distanceToSegment(polygon[i], q) <= EPS) return !strict;
                        cnt ^= (((*this).y < polygon[i].y) - ((*this).y < q.y)) * (*this).cross(
                               polygon[i], q) > 0;
                }

                return cnt;
        }

        // Returns where p is as seen from s towards e. (1/0/-1) -> left/on line/right.
        // If the optional argument eps is given 0 is returned if p is within distance eps from the
               line.
        int sideOf(Point2D s, Point2D e) {
                return sgn(s.cross(e, (*this)));
        }

        int sideOf(Point2D &s, Point2D &e, double eps) {
                auto a = (e - s).cross((*this) - s);
                double l = (e - s).dist() * eps;

                return (a > l) - (a < -l);
        }

        friend ostream& operator << (ostream& os, Point2D p) {
                return os << "(" << p.x << "," << p.y << ")";
        }
};

//The circumcirle of a triangle is the circle intersecting all three vertices.
// ccRadius returns the radius of the circle going through points A, B and C.
double ccRadius(Point2D<double> &A, Point2D<double> &B, Point2D<double> &C) {
        return (B - A).dist() * (C - B).dist() * (A - C).dist() / abs((B - A).cross(C - A)) / 2;
}

// ccCenter returns the center of the same circle
Point2D<double> ccCenter(Point2D<double> &A, Point2D<double> &B, Point2D<double> &C) {
        Point2D<double> b = C - A, c = B - A;
        assert(b.cross(c) != 0); // no circumcircle if A,B,C aligned
        return A + (b * c.dist2() - c * b.dist2()).perp() / b.cross(c) / 2;
}

// returns true if is collinear or false otherwise
bool collinear(Point2D<double> p1, Point2D<double> p2, Point2D<double> p3) {
    return ((p2.x - p1.x) * (p3.y - p2.y)) == ((p2.y - p1.y) * (p3.x - p2.x));
}

// Return -1: a < b, 0: a == b, 1: a > b
int compareFloats(double a, double b, double eps = EPS) {
        return (a > b + eps) - (a < b - eps);
}
```

2.2 Closest Pair Of Points

```
// Complexity: O(n * log(n))
typedef Point2D<double> P;
double closestPairOfPoints(vector<P> pts) {
        auto cmp = [](P a, P b) { return a.y != b.y ? a.y < b.y : a.x < b.x; };
        set<P, decltype(cmp)> st(cmp);
        int n = (int)pts.size();

        sort(pts.begin(), pts.end(), [](P a, P b) { return a.x != b.x ? a.x < b.x : a.y < b.y; });

        double min_dist = DBL_MAX;

        for (int i = 1; i < n; i++) {
                min_dist = min(min_dist, (pts[i] - pts[i - 1]).dist());
        }

        st.insert(pts[0]);
        st.insert(pts[1]);

        for (int i = 2, j = 0; i < n; i++) {
                while (pts[j].x < pts[i].x - min_dist) {
                        st.erase(pts[j++]);
                }

                for (auto pt : st) {
                        if (abs(pt.y - pts[i].y) <= min_dist) {
                                min_dist = min(min_dist, (pt - pts[i]).dist());
                        }

                        st.insert(pts[i]);
        }

        return min_dist;
}
```

2.3 Convex Hull

```
// Complexity: O (n * log(n))
// Returns a vector of indices of the convex hull in counterclockwise order.
// Points on the edge of the hull between two other points are not considered part of the hull.
// If you want to also include points which are on the edges of the convex hull, change the '<=' to
       '<' in cross product.
typedef Point2D<double> P;
vector<P> ConvexHull(vector<P> pts) {
    sort(pts.begin(), pts.end());
    vector<P> h;

    for (int step = 1; step <= 2; step++) {
        int start = (int)h.size();

        for (auto p : pts) {
            while ((int)h.size() >= start + 2 && h[(int)h.size() - 2].cross(h[(int)h.size() - 1], p)
                  <= 0) {
                h.pop_back();
            }

            h.push_back(p);
        }

        h.pop_back();
        reverse(pts.begin(), pts.end());
    }

    return h;
}
```

2.4 Polygon

```
typedef Point2D<double> Points;
double areaPolygon(vector<Points> polygon) {
    double area = 0.0;

    for (int i = 0, n = polygon.size(); i < n; i++) {
        area += polygon[i].cross(polygon[(i + 1) % n]);
    }

    return abs(area) / 2.0;
}
```

3 Graphs

3.1 Articulation Points

```
int vis[MAXN], low[MAXN], ap[MAXN], cont;

void articulation_point(int p, int v){
    vis[v] = low[v] = ++cont;
    for(auto u : g[v]){
        if(!vis[u]){
            articulation_point(v, u);
            if(low[u] >= vis[v]) ap[v]++;
            low[v] = min(low[v], low[u]);
        }else if(u != p){
            low[v] = min(low[v], vis[u]);
        }
    }
}

vector<int> isArticulationPoint(){
    vector<int> points;
    for(int i = 1; i <= n; i++){
        if(!vis[i]) articulation_point(i, i);
    }
```

```
    for(int i = 1; i <= n; i++){
        if(i == 1 && ap[1] > 1) points.push_back(1);
        else if(i != 1 && ap[i] > 0) points.push_back(i);
    }

    return points;
}

void init(){
    cont = 0;
    memset(vis, 0, sizeof(vis));
    memset(low, 0, sizeof(low));
    memset(ap, 0, sizeof(ap));
    memset(g, 0, sizeof(g));
}
```

3.2 BFS

```
vector<int> adj[n];
int dist[n];

void BFS(){
    memset(dist, -1, sizeof(dist));
    queue<int> q;
    dist[1] = 0;
    q.push(1);

    while(!q.empty()){
        int u = q.front();
        q.pop();
        for(int v : adj[u]){
            if(dist[v] == -1 || dist[v] > dist[u] + 1){
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }
}
```

3.3 Biconnected Components

```
// Finds all Biconnected Components in an undirected graph using Tarjan's Algorithm
// An edge which is not in a component (bccs[i].size() > 2) is a bridge, i.e., not part of any cycle.
// Complexity: O(n + m)
struct BCC {
    int n, timer, bccnum = 0;
    stack<pair<int, int>> stck;
    vector<int> in, low, vis, what_block;
    vector<bool> is_articulation;
    vector<vector<int>> block_cut_tree, bccs;

    BCC(int n) : n(n) {
        bccnum = 0;
        timer = 0;
        bccs.resize(n);
        in.resize(n);
        low.resize(n);
        vis.resize(n);
        what_block.resize(n);
        is_articulation.resize(n);
        block_cut_tree.resize(2 * n);
        while (!stck.empty()) stck.pop();
    }

    void DFS(int u, int p = -1) {
        vis[u] = true;
        low[u] = in[u] = timer++;
        int children = 0;

        for (int v : g[u]) {
            if (v != p) {
                if (!vis[v]) {
                    stck.emplace( v, u );
                    DFS(v, u);
                    low[u] = min(low[u], low[v]);

                    if (low[v] >= in[u]) {
                        if (p != -1) {
                            is_articulation[u] = true;
                        }

                        while (true) {
                            auto edge = stck.top();
                            stck.pop();

                            int a = edge.first, b = edge.second;

                            what_block[a] = bccnum;
                            bccs[bccnum].push_back(a);

                            what_block[b] = bccnum;
                            bccs[bccnum].push_back(b);

                            if (a == v && b == u) {
                                break;
                            }
                        }

                        bccnum++;
                    }

                    children++;
                } else if (in[v] < in[u]) {
                    low[u] = min(low[u], in[v]);
                    stck.emplace(v, u);
                }
            }
        }

        if (p == -1 && children > 1) {
            is_articulation[u] = true;
        }
    }

    void findBCCs() {
        for (int i = 0; i < n; i++) {
            if (!vis[i]) {
                DFS(i);
            }
        }
    }

    // 0 ... bccnum - 1 are blocks.
    // bccnum ... bccnum + number of articulations points are cut points.
    void buildBlockCutTree() {
        int cuts = bccnum;

        for (int v = 0; v < n; v++) {
            if (is_articulation[v]) {
                what_block[v] = cuts++;
            }
        }

        for (int blck = 0; blck < bccnum; blck++) {
            for (auto v : bccs[blck]) {
                if (is_articulation[v]) {
                    block_cut_tree[blck].push_back(what_block[v]);
                    block_cut_tree[what_block[v]].push_back(blck);
                }
            }
        }
    }
};
```

3.4 Bridges

```
// Description: Returns in undirected graph all bridges.
// Complexity: O(n + m)
struct Bridges {
    vector<int> tin, low;
    int timer, n;
    map<pair<int, int>, bool> bridges;

    Bridges(int n) : n(n) {
        tin.resize(n);
        low.resize(n);
        timer = 0;
    }

    void DFS(int u, int p = -1) {
        tin[u] = low[u] = ++timer;

        for (auto v : g[u]) {
            if (v == p) {
                continue;
            }

            if (tin[v]) {
                low[u] = min(low[u], tin[v]);
```

```
            } else {
                DFS(v, u);
                low[u] = min(low[u], low[v]);

                if (low[v] > tin[u]) {
                    bridges[{ u, v }] = true;
                }
            }
        }
    }

    void findBridges() {
        for (int i = 1; i <= n; i++) {
            if (!tin[i]) {
                DFS(i, i);
            }
        }
    }
};
```

3.5 Centroid Decomposition

```
int sub_tree_size[MAXN], n;
vector<int> centroids;

void DFS(int u, int p) {
    sub_tree_size[u] = 1;
    bool flag = true;

    for (auto v : g[u]) {
        if (v != p) {
            DFS(v, u);
            sub_tree_size[u] += sub_tree_size[v];
            if (sub_tree_size[v] > n / 2) {
                flag = false;
            }
        }
    }

    if ((n - sub_tree_size[u]) > n / 2) {
        flag = false;
    }

    if (flag) {
        centroids.push_back(u);
    }
}

void decomposition() {
    // to implement...
}
```

3.6 DFS

```
vector<int> g[n];
int vis[n];

int tam = 0;

void DFS(int v){
    vis[v] = 1;
    for(auto u : g[v])
        if(!vis[v]) DFS(u);
}

// Cobertura Minima
// 0 = Nao visitado, 1 = Visitado, 2 = Vertice da cobertura
void DFS(int v){
    vis[v] = 1;
    for(auto u : g[v]){
        if(vis[u] == 0){
            DFS(u);
            if(vis[u] == 1) vis[v] = 2;
        }
    }
}

// O vertice u esta conectado com dest?
bool isConnect(int u, int dest){
    vis[u] = true;
    if(u == dest) return true;
    for(auto v : adj[u])
        if(connect(v, dest)) return true;
    return false;
}
```

3.7 Dijkstra

```
// Complexity: O(m * log(n))
void dijkstraSparse(int s){
    vector<int> dist(n, INF), parent(n);

    dist[s] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>> > pq;
    pq.push({ 0, s });

    while(!pq.empty()){
        int u = pq.top().second;
        int d = pq.top().first;
        pq.pop();

        if (d > dist[u]) {
            continue;
        }

        for (auto e : g[u]) {
            int v = e.first;
            int w = e.second;
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                parent[v] = u;
                pq.push({ dist[v], v });
            }
        }
    }
}

// Complexity: O(n^2 + m)
void dijkstraDense(int s) {
    vector<int> dist(n, INF);
    vector<vector<int>> parent(n);
    vector<bool> vis(n, false);

    dist[s] = 0;
    parent[s] = { -1 };

    for (int i = 0; i < n; i++) {
        int u = -1;
        for (int j = 0; j < n; j++) {
            if (!vis[j] && (u == -1 || dist[j] < dist[u])) {
                u = j;
            }
        }

        if (dist[u] == INF) {
            break;
        }

        vis[u] = true;

        for (int v = 0; v < n; v++) {
            if (g[u][v] == 0) {
                continue;
            }

            int w = g[u][v];

            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                parent[v].clear();
                parent[v].push_back(u);
            } else if (dist[u] + w == dist[v]) {
                parent[v].push_back(u);
            }
        }
    }
}
```

3.8 Dinic

```
// Complexity:
// Dinic - O(V^2 * E)
// Bipartite Graph or Unit Flow - O(sqrt(V) * E)
// Small flow - O(F * (V + E))

struct Edge {
```

```
    int to;
    int flow;
    int cap;
    int rev;
    int id;

    Edge(int to, int flow, int cap, int rev, int id = -1) :
        to(to), flow(flow), cap(cap), rev(rev), id(id) {}
};

vector<Edge> g[MAXN];
int dist[MAXN], ptr[MAXN];

void addEdge(int from, int to, int cap, int id = 0) {
    g[from].push_back(Edge(to, 0, cap, g[to].size(), id));
    g[to].push_back(Edge(from, 0, 0, g[from].size() - 1, -id));
}

bool dinicBFS(int s, int t) {
    memset(dist, -1, sizeof(dist));
    queue<int> q;

    dist[s] = 0;
    q.push(s);

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

        for (Edge e : g[u]) {
            int v = e.to;

            if (dist[v] < 0 && e.flow < e.cap) {
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }

    return dist[t] >= 0;
}

int dinicDFS(int s, int t, int flow) {
    if (s == t) {
        return flow;
    }

    for (int &i = ptr[s]; i < int(g[s].size()); i++) {
        Edge &e = g[s][i];

        if (e.cap > e.flow) {
            int v = e.to;

            if (dist[v] == dist[s] + 1) {
                int tmp_flow = dinicDFS(v, t, min(flow, e.cap - e.flow));

                if (tmp_flow > 0) {
                    e.flow += tmp_flow;
                    g[v][e.rev].flow -= tmp_flow;

                    return tmp_flow;
                }
            }
        }
    }

    return 0;
}

int dinic(int s, int t) {
    int max_flow = 0;

    while (dinicBFS(s, t)) {
        memset(ptr, 0, sizeof(ptr));

        while (int flow = dinicDFS(s, t, INF)) {
            max_flow += flow;
        }
    }

    return max_flow;
}

bool cut[MAXN];

void minCutDFS(int u) {
    cut[u] = 1;

    for (auto x : g[u]) {
        if (x.cap > x.flow && !cut[x.to]) {
            minCutDFS(x.to);
        }
    }
}

vector<int> findMinCut() {
    vector<int> idx_edges;

    for (int i = 1; i <= n; i++) {
        for (Edge e : g[i]) {
            if (cut[i] && !cut[e.to]) {
                idx_edges.push_back(e.id);
            }
        }
    }
    return idx_edges;
}
```

3.9 Floyd Warshall

```
// Calcula os menores caminhos entre todos pares de vertices.

int dist[MAXN][MAXN]; // dist = graph, sem aresta = infinito ou 0 para fechamento transitivo.

for(int k = 1; k <= n; k++){
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= n; j++){
            dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
            // dist[i][j] = dist[i][j] || (dist[i][k] && dist[k][j]); // Fechamento Transitivo
        }
    }
}
```

3.10 Heavy Light Decomposition

```
// Heavy Light Decomposition
// Complexity:
// build: O(n)
// queryPath, updatePath: O(log^2(n))
// querySubtree, updateSubtree, LCA: O(log(n))
struct HLD {
    vector<int> parent, sz, heavy, head, pos, weight, chains;
    int curr_pos, neutral;
    bool WEIGHTS_IN_EDGE;
    SegmentTreeLazy *segtree; // Import SegmentTreeLazy algorithm and change to class and put all in
         public.

    HLD(int n, bool in_edge = false) {
        parent.resize(n);
        sz.resize(n);
        heavy.resize(n);
        head.resize(n);
        pos.resize(n);
        weight.resize(n);
        chains.resize(n);

        WEIGHTS_IN_EDGE = in_edge;
        curr_pos = -1;
        neutral = 0;
        DFS(0);
        buildHLD(0);
        segtree = new SegmentTreeLazy(curr_pos, chains);
    }

    void DFS(int u, int p = -1) {
        sz[u] = 1;
        for (auto &edge : g[u]) {
            int v = edge.first, w = edge.second;
            if (v != p) {
                if (WEIGHTS_IN_EDGE) {
                    weight[v] = w;
                }

                DFS(v, u);
                sz[u] += sz[v];

                if (sz[v] > sz[g[u][0].first] or g[u][0].first == p) {
                    swap(edge, g[u][0]);
                }
            }
        }
    }

    void buildHLD(int u, int p = -1) {
        parent[u] = p;
```

```
        pos[u] = ++curr_pos;
        chains[pos[u]] = weight[u];

        for (auto &edge : g[u]) {
            int v = edge.first;
            if (v != p) {
                heavy[v] = (edge == g[u].front() ? heavy[u] : v);
                buildHLD(v, u);
            }
        }
    }

    int queryPath(int a, int b) {
        if (WEIGHTS_IN_EDGE and a == b) return neutral;
        if (pos[a] < pos[b]) swap(a, b);
        if (heavy[a] == heavy[b]) return segtree->query(pos[b] + WEIGHTS_IN_EDGE, pos[a]);
        return max(segtree->query(pos[heavy[a]], pos[a]), queryPath(parent[heavy[a]], b));
    }

    void updatePath(int a, int b, int x) {
        if (WEIGHTS_IN_EDGE and a == b) return;
        if (pos[a] < pos[b]) swap(a, b);
        if (heavy[a] == heavy[b]) return segtree->update(pos[b] + WEIGHTS_IN_EDGE, pos[a], x);
        segtree->update(pos[heavy[a]], pos[a], x);
        updatePath(parent[heavy[a]], b, x);
    }

    int querySubtree(int a) {
        if (WEIGHTS_IN_EDGE and sz[a] == 1) return neutral;
        return segtree->query(pos[a] + WEIGHTS_IN_EDGE, pos[a] + sz[a] - 1);
    }

    void updateSubtree(int a, int x) {
        if (WEIGHTS_IN_EDGE and sz[a] == 1) return;
        segtree->update(pos[a] + WEIGHTS_IN_EDGE, pos[a] + sz[a] - 1, x);
    }

    int LCA(int a, int b) {
        if (pos[a] < pos[b]) swap(a, b);
        return heavy[a] == heavy[b] ? b : LCA(parent[heavy[a]], b);
    }
};
```

3.11 Kruskal

```
// Finding Minimum Spanning Tree
// Complexity: O(m * log(n))

struct Edge {
    int u, v, w;

    Edge(int u, int v, int w) {
        this->u = u;
        this->v = v;
        this->w = w;
    }

    bool operator < (Edge const& e) {
        return w < e.w;
    }
};

int p[MAXN], sz[MAXN];

int findSet(int v) {
    if (p[v] == v) {
        return v;
    }

    return p[v] = findSet(p[v]);
}

void unionSet(int a, int b) {
    int u = findSet(a);
    int v = findSet(b);

    if (u != v) {
        if (sz[u] < sz[v]) {
            swap(u, v);
        }

        p[v] = u;
        sz[u] += sz[v];
    }
}

vector<Edge> kruskal(vector<Edge> edges) {
    vector<Edge> mst;

    for (int i = 1; i <= n; i++) {
        p[i] = i;
        sz[i] = 0;
    }

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

    int cost = 0;

    for (auto edge : edges) {
        if (findSet(edge.u) != findSet(edge.v)) {
            cost += edge.w;
            mst.push_back(edge);
            unionSet(edge.u, edge.v);
        }
    }

    return mst;
}
```

3.12 LCA

```
// Complexity: Preprocess: O(n * log(n)); Query LCA: O(log(n));

int LOG_MAX_NODES = 25;
vector<vector<int>> up;
vector<pair<int, int>> g[MAXN];
vector<int> depth;
vector<int> dist;
vector<vector<int>> mx_edge_weight;

void addEdge(int u, int v, int w = 0) {
    g[u].push_back({ v, w });
    g[v].push_back({ u, w });
}

void DFS(int u, int p, int w = 0) {
    up[u][0] = p;
    // mx_edge_weight[u][0] = w;

    for (int l = 1; l <= LOG_MAX_NODES; l++) {
        up[u][l] = up[up[u][l - 1]][l - 1];
        // mx_edge_weight[u][l] = max(mx_edge_weight[up[u][l - 1]][l - 1], mx_edge_weight[u][l - 1]);
    }

    for (auto &edge : g[u]) {
        int v = edge.first;
        long long w = edge.second;

        if (v != p) {
            depth[v] = depth[u] + 1;
            // dist[v] = dist[u] + w;
            DFS(v, u, w);
        }
    }
}

int LCA(int a, int b) {
    if (depth[a] > depth[b]) {
        swap(a, b);
    }

    int mx_edge = 0;

    for (int i = LOG_MAX_NODES; i >= 0; --i) {
        if (depth[b] - (1 << i) >= depth[a]) {
            // mx_edge = max(mx_edge, mx_edge_weight[b][i]);
            b = up[b][i];
        }
    }

    if (a == b) {
        return a;
    }

    for (int i = LOG_MAX_NODES; i >= 0; i--) {
        if (up[a][i] != up[b][i]) {
            // mx_edge = max({ mx_edge, mx_edge_weight[a][i], mx_edge_weight[b][i] });
            a = up[a][i];
            b = up[b][i];
        }
    }

    // mx_edge = max({ mx_edge, mx_edge_weight[a][0], mx_edge_weight[b][0] });
    return up[a][0];
}
```

```
int liftingUp(int a, int x) {
    for (int i = LOG_MAX_NODES; i >= 0; i--) {
        if (x - (1 << i) >= 0) {
            a = up[a][i];
            x -= (1 << i);
        }
    }

    return a;
}

void preprocess(int root, int n) {
    LOG_MAX_NODES = (int)ceil(log2(n));
    depth.resize(n);
    up.assign(n, vector<int>(LOG_MAX_NODES + 1));
    // mx_edge_weight.assign(n, vector<int>(LOG_MAX_NODES + 1));
    // dist.resize(n);

    DFS(root, root);
}
```

3.13 Min Cost Max Flow

```
struct Edge {
    int to;
    int flow;
    long long cap;
    long long cost;
    int rev;

    Edge() {}

    Edge (int to, int flow, long long cap, long long cost, int rev) :
        to(to), flow(flow), cap(cap), cost(cost), rev(rev) {}
};

vector<Edge> g[MAXN];
long long dist[MAXN], phi[MAXN];
pair<int, int> parent[MAXN];
int n, m;

void add_edge(int from, int to, long long cap, long long cost) {
    g[from].push_back(Edge(to, 0, cap, cost, g[to].size()));
    g[to].push_back(Edge(from, 0, 0, -cost, g[from].size() - 1));
}

void MCMFBellmanFord(int s) {
    fill(phi, phi + MAXN, INF);
    phi[s] = 0;

    for (int i = 0; i < n - 1; i++) {
        for (int u = 0; u < n; u++) {
            for (Edge e : g[u]) {
                if (!e.cap) {
                    continue;
                }

                int v = e.to;
                long long w = e.cost;
                phi[v] = min(phi[v], phi[u] + w);
            }
        }
    }
}

// Complexity: O(m * log(n))
bool MCMFDjikstraSparse(int s, int t) {
    for (int i = 0; i < MAXN; i++) {
        dist[i] = INF;
        parent[i] = { -1, -1 };
    }

    dist[s] = 0;
    priority_queue<pair<long long, int>, vector<pair<long long, int>>, greater<pair<long long, int>>>
         pq;
    pq.push({ 0, s });

    bool flag = false;
    while (!pq.empty()) {
        long long d = pq.top().first;
        int u = pq.top().second;
        pq.pop();

        if (u == t) flag = true;

        if (d != dist[u]) continue;

        for (int i = 0; i < int(g[u].size()); i++) {
            Edge e = g[u][i];
            int v = e.to;

            if (e.cap - e.flow <= 0) continue;

            long long w = e.cost + phi[u] - phi[v];
            if (dist[v] > d + w) {
                dist[v] = d + w;
                parent[v] = { u, i };
                pq.push({ dist[v], v });
            }
        }
    }

    for (int i = 0; i < MAXN; i++) {
        if (phi[i] < INF && dist[i] < INF) {
            phi[i] += dist[i];
        }
    }

    return flag;
}

// Complexity: O(n^2 + m)
bool MCMFDjikstraDense(int s, int t) {
    for (int i = 0; i < MAXN; i++) {
        dist[i] = INF;
        parent[i] = { -1, -1 };
    }

    vector<bool> vis(n, false);

    dist[s] = 0;
    for (int i = 0; i < n; i++) {
        int u = -1;
        for (int j = 0; j < n; j++) {
            if (!vis[j] && (u == -1 || dist[j] < dist[u])) {
                u = j;
            }
        }

        if (dist[u] == INF) {
            break;
        }

        vis[u] = true;
        for (int j = 0; j < int(g[u].size()); j++) {
            Edge e = g[u][j];
            int v = e.to;

            if (e.cap - e.flow <= 0) {
                continue;
            }

            long long w = e.cost + phi[u] - phi[v];

            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                parent[v] = { u, j };
            }
        }
    }

    for (int i = 0; i < MAXN; i++) {
        if (phi[i] < INF && dist[i] < INF) {
            phi[i] += dist[i];
        }
    }

    return parent[t].first >= 0;
}

pair<long long, long long> MCMF(int s, int t, int k = INF) {
    long long min_cost = 0, max_flow = 0;

    MCMFBellmanFord(s);

    while (MCMFDjikstraSparse(s, t)) {
        long long flow = INF, cost = 0; // Flow and Cost on each augmented path found.

        for (int u = t; u != s; u = parent[u].first) {
            Edge e = g[parent[u].first][parent[u].second];
            flow = min(flow, e.cap - e.flow);
        }

        for (int u = t; u != s; u = parent[u].first) {
            Edge &e = g[parent[u].first][parent[u].second];
            e.flow += flow;
            g[e.to][e.rev].flow -= flow;
            min_cost += e.cost * flow;
            cost += e.cost;
        }
```

```
        max_flow += flow;
    }

    return { min_cost, max_flow };
}
```

3.14 Prim

```
struct Edge {
    int u, v, w;

    Edge(int u, int v, int w){
        this->u = u;
        this->v = v;
        this->w = w;
    }
};

vector<Edge> G[MAXN], MST;
bool vis[MAXN];

bool comp(Edge &a, Edge &b){ return a.w > b.w; }

int prim(int v){
    priority_queue<Edge, vector<Edge>, decltype(&comp)> pq(&comp);
    int min_cost = 0;

    pq.push(Edge(v, v, 0));

    while(!pq.empty()){
        Edge top = pq.top();
        pq.pop();

        if(vis[top.v]) continue;

        vis[top.v] = 1;
        min_cost += top.w;

        MST.push_back(top);

        for(auto u : G[top.v]){
            if(!vis[u.v]) pq.push(u);
        }
    }

    return min_cost;
}
```

3.15 Strongly Connected Components

```
// Finds all Strongly Connected Components in a directed graph using Tarjan's Algorithm.
// Complexity: O(n + m)
struct SCC {
    vector<int> tin, low, in_stck;
    int timer = 0, sccs_cnt = 0, n;
    stack<int> stck;
    vector<vector<int>> SCCS;

    SCC(int n) : n(n) {
        SCCS.resize(n);
        tin.resize(n);
        low.resize(n);
        in_stck.resize(n);
        timer = 0;
        sccs_cnt = 0;
        while (!stck.empty()) stck.pop();
    }

    void DFS(int u) {
        tin[u] = low[u] = ++timer;
        stck.push(u);
        in_stck[u] = true;

        for (auto v : g[u]) {
            if (tin[v] == 0) {
                DFS(v);
                low[u] = min(low[u], low[v]);
            } else if (in_stck[v]) {
                low[u] = min(low[u], tin[v]);
            }
        }

        if (tin[u] == low[u]) {
            while (!stck.empty()) {
                int x = stck.top();
                stck.pop();

                in_stck[x] = false;
                SCCS[sccs_cnt].push_back(x);

                if (x == u) {
                    break;
                }
            }
            sccs_cnt++;
        }
    }
};
```

3.16 Topological Sort

```
int vis[MAXN], dist[MAXN], maior, end_point;
vector<int> g[MAXN];
stack<int> topoSort;

void topological_sort(int v){
    vis[v] = 1;
    for(auto u : g[v]){
        if(!vis[u]) topological_sort(u);
    }
    topoSort.push(v);
}

void longest_path(int src){
    dist[src] = 0;
    maior = 0;
    while(!topoSort.empty()){
        int v = topoSort.top();
        topoSort.pop();
        if(dist[v] != -1){
            for(auto u : g[v]){
                dist[u] = max(dist[u], dist[v] + 1);
            }
        }
        maior = max(maior, dist[v]);
    }
    for(int i = 1; i <= n; i++){
        if(maior == dist[i]){
            end_point = i;
            break;
        }
    }
}
```

4 Math

4.1 Basics

```
// Complexity: O(log(n))
long long gcd(long long a, long long b) {
    return b ? gcd(b, a % b) : a;
}

// Complexity: O(log(n))
long long lcm(long long a, long long b) {
    return a / gcd(a, b) * b;
}

// Complexity: O(log(n))
long long gcde(long long a, long long b, long long &x, long long &y) {
    if (b == 0) {
        x = 1, y = 0;
        return a;
    } else {
        long long d = gcde(b, a % b, y, x);
        y -= x * (a / b);
        return d;
    }
}
```

4.2 Binomial Coefficients

```
// Complexity: O(n^2)
vector<vector<long long>> binomialCoefficient(int n, int mod = MOD) {
    vector<vector<long long>> C(n + 1, vector<long long>(n + 1));
    C[0][0] = 1;

    for (int i = 1; i <= n; i++){
        C[i][0] = C[i][i] = 1;
        for (int j = 1; j < i; j++) {
            C[i][j] = (C[i - 1][j - 1] + C[i - 1][j]) % mod;
        }
    }

    return C;
}

// Complexity: O(n)
vector<long long> fact, inv_fact;

void buildBinomial(int n, int mod = MOD) {
    fact.resize(n + 1);
    inv_fact.resize(n + 1);

    fact[0] = 1;
    for (int i = 1; i <= n; i++) fact[i] = (fact[i - 1] * 1LL * i) % mod;

    inv_fact[n] = powMod(fact[n], mod - 2);
    for (int i = n - 1; i >= 0; i--) inv_fact[i] = (inv_fact[i + 1] * 1LL * (i + 1)) % mod;
}

long long binomialCoefficient(int n, int k, int mod = MOD) {
    if (n < k) return 0;
    return ((fact[n] * inv_fact[k]) % mod) * inv_fact[n - k] % mod;
}
```

4.3 Chinese Remainder

```
long long chineseRemainder(vector<pair<long long, long long>> v) {
    long long rem = v[0].first, mod = v[0].second;
    long long ans = rem, m = mod;

    for (int i = 1; i < (int)v.size(); i++) {
        long long x, y;
        rem = v[i].first, mod = v[i].second;
        long long g = gcde(mod, m, x, y);

        if ((ans - rem) % g != 0) {
            return -1;
        }

        ans = ans + 1LL * (rem - ans) * (m / g) * y;
        m = (mod / g) * (m / g) * g;
        ans = (ans % m + m) % m;
    }

    if (ans == 0) {
        long long _lcm = v[0].second;

        for (int i = 1; i < (int)v.size(); i++) {
            _lcm = lcm(_lcm, v[i].second);
        }

        return _lcm;
    }

    return ans;
}
```

4.4 Factors And Divisors

```
// For number greater than 9*10^14 see Pollard Rho Algorithm.
// Complexity: O(sqrt(n))
vector<long long> getFactors(long long n) {
    vector<long long> factors;

    for (long long d = 2; d * d <= n; d++) {
        while (n % d == 0) {
            factors.push_back(d);
            n /= d;
        }
    }

    if (n > 1) factors.push_back(n);

    return factors;
}

// Complexity: O(sqrt(n))
vector<long long> getDivisors(long long n) {
    vector<long long> divisors;

    for (long long d = 1 ; d * d <= n; d++) {
        if (n % d == 0) {
            if (n / d == d) divisors.push_back(d);
            else divisors.push_back(d), divisors.push_back(n / d);
        }
    }

    return divisors;
}
```

4.5 Matrix Fast Exponentiation

```
typedef vector<vector<long long>> matrix;

matrix matrixMultiplication(matrix a, matrix b){
    matrix c(int(a.size()), vector<long long>(int(b[0].size())));

    for (int i = 0; i < int(a.size()); i++) {
        for (int j = 0; j < int(b[0].size()); j++) {
            for (int k = 0; k < int(b.size()); k++) {
                c[i][j] = (c[i][j] + a[i][k] * b[k][j]);
            }
        }
    }

    return c;
}

matrix matrixExponentiation(matrix base, long long n){
    if (n == 1) {
        return base;
    }

    if (n & 1) {
        return matrixMultiplication(base, matrixExponentiation(base, n - 1));
    }

    matrix tmp = matrixExponentiation(base, n / 2);

    return matrixMultiplication(tmp, tmp);
}
```

4.6 Miller Rabin

```
// Primality Test for Large Number
// Complexity: O(k * log(n)^3)
bool millerRabin(ull n) {
        if (n < 2) return false;

        ull s = __builtin_ctzll(n - 1);
        ull d = n >> s;
        ull base[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 };

        for (ull b : base) {
                ull p = powMod(b % n, d, n);
                ull i = s;
                while (p != 1 && p != n - 1 && b % n && i--) {
                        p = multMod(p, p, n);
                }
                if (p != n - 1 && i != s) return false;
        }

        return true;
}
```

4.7 Modular Arithmetic

```
long long addMod(long long a, long long b, long long mod = MOD) {
    return (a + b) % mod;
}

long long subMod(long long a, long long b, long long mod = MOD) {
    return (a - b + mod) % mod;
}

long long inverseMod(long long a, long long mod = MOD) {
    long long x, y;
    long long g = gcde(a, mod, x, y);

    if (g != 1) {
        return -1;
    } else {
        return (x % mod + mod) % mod;
    }
}

long long multMod(long long a, long long b, long long mod = MOD) {
    return (a * b) % mod;
}

long long divMod(long long a, long long b, long long mod = MOD) {
    return multMod(a, inverseMod(b, mod));
}

long long powMod(long long b, long long e, long long mod = MOD) {
        long long p = 1;
        for (; e; b = b * b % mod, e /= 2) {
                if (e & 1) {
            p = p * b % mod;
        }
    }
        return p;
}

// to avoid long long overflow and increase speed of mult and pow use the functions below.
typedef unsigned long long ull;
ull multMod(ull a, ull b, ull mod = MOD) {
        long long ret = a * b - mod * ull(1.L / mod * a * b);
        return ret + mod * (ret < 0) - mod * (ret >= (long long)mod);
}

ull powMod(ull b, ull e, ull mod = MOD) {
        ull p = 1;
        for (; e; b = multMod(b, b, mod), e /= 2) {
                if (e & 1) {
            p = multMod(p, b, mod);
        }
    }
        return p;
}
```

4.8 Pollard Rho

```
// Integer Factorization For Large Number
// Complexity: O(n^1/4)
ull pollard(ull n) {
        auto f = [n](ull x) { return multMod(x, x, n) + 1; };
        ull x = 0, y = 0, t = 30, prd = 2, i = 1, q;

        while (t++ % 40 || __gcd(prd, n) == 1) {
                if (x == y) {
                        x = ++i;
                        y = f(x);
                }
                if ((q = multMod(prd, max(x, y) - min(x, y), n))) {
                        prd = q;
                }
                x = f(x), y = f(f(y));
        }

        return __gcd(prd, n);
}

void getFactors(ull n, set<ull> &factors) {
        if (millerRabin(n)) {
                factors.insert(n);
                return;
        }

        ull f = pollard(n);
        getFactors(f, factors);
        getFactors(n / f, factors);
}
```

4.9 Sieve

```
// Complexity: O(n * log(log(n)))
vector<long long> sieve(int n = 10'000'000) {
    vector<long long> primes;
    vector<bool> is_prime(n + 1, true);
    is_prime[0] = false, is_prime[1] = false;

    for (int i = 2; i * i <= n; i++) {
        if (is_prime[i]) {
            for (int j = i * i; j <= n; j += i) {
                is_prime[j] = false;
            }
        }
    }

    for (int i = 2; i <= n; i++) {
        if (is_prime[i]) primes.push_back(i);
    }

    return primes;
}
```

5 Miscellaneous

5.1 Histogram Problem

```
// Funcao que retorna a maior area retangular de um histograma.

long long getMaxArea(vector<long long> &hist, long long n){
    stack<long long> s;

    long long max_area = 0;
    long long tp;
    long long area_with_top;

    long long i = 0;
    while(i < n){
        if(s.empty() || hist[s.top()] <= hist[i]){
            s.push(i++);
        }else{
            tp = s.top();
            s.pop();
            area_with_top = hist[tp] * (s.empty() ? i : i - s.top() - 1);
            if(max_area < area_with_top)
                max_area = area_with_top;
        }
    }

    while(!s.empty()){
        tp = s.top();
        s.pop();
        area_with_top = hist[tp] * (s.empty() ? i : i - s.top() - 1);
        if(max_area < area_with_top)
            max_area = area_with_top;
    }
    return max_area;
}
```

6 Strings

6.1 Manacher

```
// Complexity: O(n)
struct Manacher {
    int n;
    string s;
    vector<int> odd, even, p;

    Manacher(const string &s) {
        this->s = s;
        this->n = (int)s.size();
        this->odd.resize(n);
        this->even.resize(n);
```

```
        this->p.resize(2 * n - 1);
        preprocess();
    }

    void preprocess() {
        for (int i = 0, l = 0, r = -1; i < n; i++) {
            int k = (i > r) ? 1 : min(odd[l + r - i], r - i + 1);
            while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) {
                k++;
            }
            odd[i] = k--;
            if (i + k > r) {
                l = i - k;
                r = i + k;
            }
        }

        for (int i = 0, l = 0, r = -1; i < n; i++) {
            int k = (i > r) ? 0 : min(even[l + r - i + 1], r - i + 1);
            while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) {
                k++;
            }
            even[i] = k--;
            if (i + k > r) {
                l = i - k - 1;
                r = i + k ;
            }
        }

        for (int i = 0; i < n; i++) {
            p[2 * i] = 2 * odd[i] - 1;
        }

        for (int i = 0; i < n - 1; i++) {
            p[2 * i + 1] = 2 * even[i + 1];
        }
    }

    bool isPalindrome(int l, int r) {
        return p[l + r] >= r - l + 1;
    }

    int longestPalindromeSize() {
        return *max_element(p.begin(), p.end());
    }

    vector<pair<pair<int, int>, string>> getAllPalindromes() {
        vector<pair<pair<int, int>, string>> palindromes;

        for (int i = 0; i < n; i++) {
            if (odd[i]) {
                int l = i - odd[i] + 1, r = i + odd[i] - 1;
                palindromes.push_back({{ l, r }, s.substr(l, r - l + 1) });
            }

            if (even[i]) {
                int l = i - even[i], r = i + even[i] - 1;
                palindromes.push_back({{ l, r }, s.substr(l, r - l + 1) });
            }
        }

        return palindromes;
    }

    // For starting simply reverse string input and the vector ending.
    vector<int> getMaxPalindromeEndingInEachIndex() {
        vector<int> ending(n);
        ending[0] = 1;

        for (int i = 1; i < n; i++) {
            ending[i] = min(ending[i - 1] + 2, i + 1);
            while (!isPalindrome(i - ending[i] + 1, i)) {
                ending[i]--;
            }
        }

        return ending;
    }
};
```

6.2 Suffix Array

```
// Returns suffixArray that each index of the array is the starting of the suffix which is ith in the
      sorted suffix array
// Complexity: O(n * log(n))
vector<int> suffixArray(string s) {
    s.push_back('$');
    int n = (int)s.size();
    const int alphabet = 256;

    vector<int> p(n), c(n), cnt(max(alphabet, n), 0);

    for (auto ch : s) cnt[ch]++;
    for (int i = 1; i < alphabet; i++) cnt[i] += cnt[i - 1];
    for (int i = 0; i < n; i++) p[--cnt[s[i]]] = i;
    for (int i = 1; i < n; i++) c[p[i]] = c[p[i - 1]] + (s[p[i]] != s[p[i - 1]]);

    vector<int> p_new(n), c_new(n);

    for (int k = 0; (1 << k) < n; ++k) {
        int classes = c[p[n - 1]] + 1;
        fill(cnt.begin(), cnt.begin() + classes, 0);

        for (int i = 0; i < n; i++) p_new[i] = (p[i] - (1 << k) + n) % n;
        for (int i = 0; i < n; i++) cnt[c[p_new[i]]]++;
        for (int i = 1; i < classes; i++) cnt[i] += cnt[i - 1];
        for (int i = n - 1; i >= 0; i--) p[--cnt[c[p_new[i]]]] = p_new[i];

        c_new[p[0]] = 0;

        for (int i = 1; i < n; i++) {
            pair<int, int> curr = { c[p[i]], c[(p[i] + (1 << k)) % n] };
            pair<int, int> prev = { c[p[i - 1]], c[(p[i-1] + (1 << k)) % n] };

            c_new[p[i]] = c_new[p[i - 1]] + (prev != curr);
        }

        c.swap(c_new);
    }

    // p.erase(p.begin());

    return p;
}

// Complexity: O(n)
vector<int> longestCommonPrefix(string s, vector<int> &suffix_array) {
    s.push_back('$');
    int n = (int)s.size();
    vector<int> pi(n);

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

    int k = 0;
    vector<int> lcp(n - 1);

    for (int i = 0; i < n - 1; i++) {
        if (pi[i] < n - 1) {
            int j = suffix_array[pi[i] + 1];
            while (max(i, j) + k < n && s[i + k] == s[j + k]) {
                k++;
            }

            lcp[pi[i]] = k;
            if (k > 0) k--;
        }
    }

    return lcp;
}

// To calc LCS for multiple texts use a slide window with minqueue.
// Complexity: O(n)
string longestCommonSubstring(string a, string b) {
    string s = a + '$' + b + '#';
    vector<int> suffix_array = suffixArray(s);
    vector<int> lcp = longestCommonPrefix(s, suffix_array);
    int lcs = 0, idx = -1;

    for (int i = 0; i < (int)s.size(); i++) {
        if ((suffix_array[i] < (int)a.size()) != (suffix_array[i + 1] < (int)a.size())) {
            if (lcp[i] > lcs) {
                lcs = lcp[i];
                idx = suffix_array[i];
            }
        }
    }

    return s.substr(idx, lcs);
}

// Complexity: O(n)
long long numberOfDifferentSubstrings(string s) {
    long long n = (long long)s.size();
    vector<int> suffix_array = suffixArray(s);
    vector<int> lcp = longestCommonPrefix(s, suffix_array);

    long long cnt = n * (n + 1) / 2;
    for (int i = 0; i < n; i++) cnt -= lcp[i];
```

```cpp
    return cnt;
}

// Complexity: O(n)
int longestRepeatedSubstring(string s) {
    int lrs = 0;
    vector<int> suffix_array = suffixArray(s);
    vector<int> lcp = longestCommonPrefix(s, suffix_array);

    for (int i = 0; i < n; i++) lrs = max(lrs, lcp[i]);

    return lrs;
}
```

6.3 ZFunction

```cpp
// Returns a vector that each z[i] is the length of the longest common prefix between s and the suffix
       of s starting at i.
// e.g: "aaabaab" - [0,2,1,0,2,1,0]
// z[i] == n - i -> so the prefix and suffix is the same.
// Search substring T in S, just concat T + 'some char' + S and search for z[i + length(T) + 1] ==
      length(T), in [i..length(S)].
// Complexity: O(n)
vector<int> ZFunction(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 && s[z[i]] == s[i + z[i]]) ++z[i];
        if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
    }

    return z;
}
```

7 Templates

7.1 C++

```cpp
#include <bits/stdc++.h>

using namespace std;

string to_string(string s) { return '"' + s + '"'; }
string to_string(const char* s) { return to_string((string) s); }
template <typename A, typename B>
string to_string(pair<A, B> p) {
    return "(" + to_string(p.first) + ", " + to_string(p.second) + ")";
}

template <typename A>
string to_string(A v) {
    bool first = true;
    string res = "\n{";
    for (const auto &x : v) {
        if (!first) res += ", ";
        first = false;
        res += to_string(x);
    }
    res += "}";
    return res;
}

void debug_out() { cerr << endl; }

template <typename Head, typename... Tail>
void debug_out(Head H, Tail... T) {
    cerr << " " << to_string(H);
    debug_out(T...);
}

#ifdef LOCAL
    #define debug(...) cerr << "[" << #__VA_ARGS__ << "]:", debug_out(__VA_ARGS__)
#else
    #define debug(...) 42
#endif

const int INF = 0x3f3f3f3f;
const int MOD = 1e9+7;
const long long INFLL = 0x3f3f3f3f3f3f3f3fLL;
const long double EPS = 1e-9;
const long double PI = acos(-1.0);

int main(void){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int t;

    cin >> t;

    while (t--) {
        int n;

        cin >> n;

        vector<int> v(n);
        for (auto &x : v) cin >> x;
    }

    return 0;
}
```