UFSC Livro do time (2024-2025)

Sumário

1 Data Structures

1.1 Centroid Decomposition

```cpp
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>

using namespace std;
using pii = pair<int, int>;

// `closest_red`, query and update were used for solving xenia and the tree.
struct CentroidDecomposition {
  vector<vector<int>> tree;
  vector<int> subtrees_sz, closest_red;
  vector<vector<pii>> parents;
  vector<bool> removed;

  CentroidDecomposition(vector<vector<int>> adj)
    :tree{adj} {
    int n = tree.size();

    subtrees_sz.resize(n);
    removed.assign(n, false);
    closest_red.assign(n, 1e9);
    parents.resize(n);

    centroid_decomposition(0, -1);
  }

  void calculate_subtree_sizes(int u, int p = -1) {
    subtrees_sz[u] = 1;
    for (auto v : tree[u]) {
      if (v == p || removed[v])
        continue;
      calculate_subtree_sizes(v, u);
      subtrees_sz[u] += subtrees_sz[v];
    }
  }

  int find_centroid(int u, int p, int n) {
    for (auto v : tree[u]) {
      if (v == p || removed[v])
        continue;
      if (subtrees_sz[v] > n / 2)
        return find_centroid(v, u, n);
    }

    return u;
  }

  void calculate_distance_to_centroid(int u, int p, int centroid, int d) {
    for (auto v : tree[u]) {
      if (v == p || removed[v])
        continue;
      calculate_distance_to_centroid(v, u, centroid, d + 1);
    }
    parents[u].push_back({centroid, d});
  }

  void centroid_decomposition(int u, int p = -1) {
    calculate_subtree_sizes(u);
    int centroid = find_centroid(u, p, subtrees_sz[u]);

    for (auto v : tree[centroid]) {
      if (removed[v])
        continue;
      calculate_distance_to_centroid(v, centroid, centroid, 1);
    }

    removed[centroid] = true;

    for (auto v : tree[centroid]) {
      if (removed[v])
        continue;
      centroid_decomposition(v, u);
    }
  }

  int query(int u) {
    int ret = closest_red[u];
    for (auto&[p, pd] : parents[u])
      ret = min(ret, pd + closest_red[p]);

    return ret;
  }

  void update(int u) {
    closest_red[u] = 0;
    for (auto &[p, pdist] : parents[u])
      closest_red[p] = min(closest_red[p], pdist);
  }
};
```

1.2 Fenwick Tree

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

using namespace std;

// 1-indexed FenwickTree
```

```cpp
struct FenwickTree {
  FenwickTree(int n) { ft.assign(n + 1, 0); }

  FenwickTree(vector<int> &vec) {
    ft.assign(vec.size() + 1, 0);
    for (int i = 0; i < vec.size(); ++i)
      update(i + 1, vec[i]);
  }

  inline int ls_one(int x) { return x & (-x); }

  int query(int r) {
    int sum = 0;
    while (r) {
      sum += ft[r];
      r -= ls_one(r);
    }
    return sum;
  }

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

  void update(int i, int v) {
    while (i < ft.size()) {
      ft[i] += v;
      i += ls_one(i);
    }
  }

  // Finds smallest index i on FenwickTree such that query(1, i) >= rank.
  // I.e: smallest i for [1, i] >= k
  int select(long long k) { // O(log^2 m)
    int lo = 1, hi = ft.size() - 1;
    for (int i = 0; i < 30; ++i) {
      int mid = (lo + hi) / 2;
      (query(1, mid) < k) ? lo = mid : hi = mid;
    }
    return hi;
  }

  vector<int> ft;
};

// Range Update - Point query
struct RUPQ {
  FenwickTree ft;
  RUPQ(int m) : ft{m} {}

  void range_update(int ui, int uj, int v) {
    ft.update(ui, v);
    ft.update(uj + 1, -v);
  }

  int point_query(int i) { return ft.query(i); }
};

// Range Update - Range Query
struct RURQ {
  RUPQ rupq;
  FenwickTree ft;

  RURQ(int m) : ft{m}, rupq(m) {}

  void range_update(int ui, int uj, int v) {
    rupq.range_update(ui, uj, v);
    ft.update(ui, v * (ui - 1));
    ft.update(uj + 1, -v * uj);
  }

  int query(int j) { return rupq.point_query(j) * j - ft.query(j); }

  int query(int i, int j) { return query(j) - query(i - 1); }
};
```

1.3 Ordered Set

```cpp
// C++ program to demonstrate the
// ordered set in GNU C++
#include <iostream>
using namespace std;

// Header files, namespaces,
// macros as defined above
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

#define ordered_set tree<int, null_type,less<int>, rb_tree_tag,tree_order_statistics_node_update>

// Driver program to test above functions
int main()
{
    // Ordered set declared with name o_set
    ordered_set o_set;

    // insert function to insert in
    // ordered set same as SET STL
    o_set.insert(5);
    o_set.insert(1);
    o_set.insert(2);

    // Finding the second smallest element
    // in the set using * because
    //  find_by_order returns an iterator
    cout << *(o_set.find_by_order(1))
         << endl;

    // Finding the number of elements
    // strictly less than k=4
    cout << o_set.order_of_key(4)
         << endl;

    // Finding the count of elements less
    // than or equal to 4 i.e. strictly less
    // than 5 if integers are present
    cout << o_set.order_of_key(5)
         << endl;

    // Deleting 2 from the set if it exists
    if (o_set.find(2) != o_set.end())
        o_set.erase(o_set.find(2));

    // Now after deleting 2 from the set
    // Finding the second smallest element in the set
    cout << *(o_set.find_by_order(1))
         << endl;

    // Finding the number of
    // elements strictly less than k=4
    cout << o_set.order_of_key(4)
         << endl;

    return 0;
}
```

1.4 Segment Tree

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

#ifdef DEBUG
#define PRINT(s) std::cout << s << '\n';
#endif

using namespace std;
using ll = long long;

#include <bits/stdc++.h>

using namespace std;
using pii = pair<int, int>;
using ll = long long;
using ull = unsigned long long;

struct SegmentTree {

  inline int left(int p) { return p * 2; }

  inline int right(int p) { return p * 2 + 1; }

  SegmentTree(vector<int> &vec) {
    n = vec.size();
    int sz = 4 * n;

    tree.assign(sz, 1e9);
    lazy.assign(sz, -1);

    build(1, 0, n - 1, vec);
  }

  inline int merge(int a, int b) { return min(a, b); }

  void build(int p, int l, int r, vector<int> &vec) {
    if (l == r) {
      tree[p] = vec[l];
      return;
    }

    int m = (l + r) / 2;
    build(left(p), l, m, vec);
    build(right(p), m + 1, r, vec);

    tree[p] = merge(tree[left(p)], tree[right(p)]);
  }

  void propagate(int p, int l, int r) {
    if (lazy[p] == -1)
      return;

    tree[p] = lazy[p];

    if (l != r)
      lazy[left(p)] = lazy[right(p)] = lazy[p];

    lazy[p] = -1;
  }

  int query(int i, int j) { return query(1, 0, n - 1, i, j); }

  void update(int i, int j, int v) { update(1, 0, n - 1, i, j, v); }

  vector<int> lazy, tree;
  int n;

private:
  int query(int p, int l, int r, int i, int j) {
    propagate(p, l, r);
    if (i > j) // valor impossível. merge() deve ignorá-lo
      return 1e9;
    if (l >= i && r <= j)
      return tree[p];

    int m = (l + r) / 2;
    return merge(query(left(p), l, m, i, min(m, j)),
                 query(right(p), m + 1, r, max(i, m + 1), j));
  }

  void update(int p, int l, int r, int i, int j, int v) {
    propagate(p, l, r);
    if (i > j)
      return;
    if (l >= i && r <= j) {
      tree[p] = v;
      lazy[p] = v;
      return;
    }

    int m = (l + r) / 2;
    update(left(p), l, m, i, min(j, m), v);
    update(right(p), m + 1, r, max(i, m + 1), j, v);
    tree[p] = merge(tree[left(p)], tree[right(p)]);
  }
};
```

1.5 Sparse Table

```
#include<vector>
#include<utility>

// preprocessing: O(n log n)
// range minimum query  (minimum element in [L, R] interval): O(1)
struct SparseTable {
  vector<vector<int>> st;
  int k = 25;
  int n;

  SparseTable(const vector<int>& vec) {
    n = vec.size();
    st.assign(k+1, vector<int>(n));
    st[0] = vec;

    for (int i = 1; i <= k; ++i)
      for (int j = 0; j + (1 << i) <= n; ++j)
        st[i][j] = min(st[i-1][j], st[i-1][j + (1 << (i-1))]);
  }

  int query(int l, int r) {
    int i = bit_width((unsigned long) (r - l + 1)) - 1; // change to log2 and memoization if c++20 is
         not available.
    return min(st[i][l], st[i][r - (1 << i) + 1]);
  }
};
```

```
s
```

1.6 Suffix Array

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

using namespace std;
using vi = vector<int>;
using ii = pair<int, int>;

class SuffixArray {

private:
  vi RA;                                  // rank array
  void countingSort(int k) {              // O(n)
    int maxi = max(300, n);               // up to 255 ASCII chars
    vi c(maxi, 0);                        // clear frequency table
    for (int i = 0; i < n; ++i)           // count the frequency
      ++c[i + k < n ? RA[i + k] : 0];   // of each integer rank
    for (int i = 0, sum = 0; i < maxi; ++i) {
      int t = c[i];
      c[i] = sum;
      sum += t;
    }
    vi tempSA(n);
    for (int i = 0; i < n; ++i) // sort SA
      tempSA[c[SA[i] + k < n ? RA[SA[i] + k] : 0]++] = SA[i];
    swap(SA, tempSA); // update SA
  }
  void constructSA() { // can go up to 400K chars
    SA.resize(n);
    iota(SA.begin(), SA.end(), 0); // the initial SA
    RA.resize(n);
    for (int i = 0; i < n; ++i)
      RA[i] = T[i];                       // initial rankings
    for (int k = 1; k < n; k <<= 1) { // repeat log_2 n times
      // this is actually radix sort
      countingSort(k); // sort by 2nd item
      countingSort(0); // stable-sort by 1st item
      vi tempRA(n);
      int r = 0;
      tempRA[SA[0]] = r;             // re-ranking process
      for (int i = 1; i < n; ++i) // compare adj suffixes
        tempRA[SA[i]] = // same pair => same rank r; otherwise, increase r
            ((RA[SA[i]] == RA[SA[i - 1]]) &&
             (RA[SA[i] + k] == RA[SA[i - 1] + k]))
                ? r
                : ++r;
      swap(RA, tempRA); // update RA
      if (RA[SA[n - 1]] == n - 1)
        break; // nice optimization
    }
  }

public:
  const char *T; // the input string
  const int n;   // the length of T
  vi SA;         // Suffix Array
  SuffixArray(const char *initialT, const int _n) : T(initialT), n(_n) {
    constructSA(); // O(n log n)
  };
```

1.7 Union Find

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

struct UnionFind {
  UnionFind(int n) {
    p.resize(n);
    rank.assign(n, 1);
    set_size.assign(n, 1);
    iota(p.begin(), p.end(), 0);
  }

  int find_set(int i) {
    if (p[i] == i)
      return i;

    return p[i] = find_set(p[i]);
  }
  inline bool same_set(int i, int j) { return find_set(i) == find_set(j); }

  void union_set(int i, int j) {
    if (same_set(i, j))
      return;

    i = p[i];
    j = p[j];

    if (rank[i] > rank[j])
      swap(i, j);

    p[i] = j;
    if (rank[i] == rank[j])
      rank[j]++;

    set_size[j] += set_size[i];
  }

  vector<int> p, rank, set_size;
};
```

2 Dynamic Programming

2.1 Coin Change

```
'''
Returns minimum amount of coins from the "coins" list
such that their sum is equal to "val".
Every element on the "coins" list can be used an unlimited
amount of times.
If no sum of coins is equal to "val", returns -1.
'''
def coin_change(coins, val):
    dp = [float("inf")] * (val + 1)
    dp[0] = 0

    for amount in range(1, val + 1):
        for coin in coins:
            if amount - coin >= 0:
                temp = 1 + dp[amount - coin]
                if temp < dp[amount]:
                    dp[amount] = temp
    return dp[val] if dp[val] != float("inf") else -1
```

2.2 Knapsack

```
'''
Returns largest possible sum of elements' values such that the sum
of the weights of these elements does not exceed "capacity".
"weights" and "values" are 0 indexed (i.e. index 0 is not empty).
Elements can be used only once.
'''
def knapsack(capacity, weights, values, element_count=None):
    if element_count is None:
        element_count = len(weights)
    dp = [0] * (capacity + 1)

    for i in range(element_count):
        for w in range(capacity, 0, -1):
            if w >= weights[i]:
                dp[w] = max(dp[w], dp[w-weights[i]] + values[i])
            else:
                break
    return dp[capacity]
```

2.3 Longest Common Subsequence

```
'''
Returns the length of the longest common subsequence
between strings "p" and "q". The sequence doesn't need
to be contiguous.
Example:
p -> "ABCXYZ"
q -> "ABXPZ"
longest_common_subsequence(p, q) -> "ABXZ", of length 4.
'''
def longest_common_subsequence(p, q):
    dp = [[0] * (len(q) + 1) for _ in range(len(p) + 1)]

    for i in range(len(p)):
        for j in range(len(q)):
            if p[i] == q[j]:
                dp[i+1][j+1] = 1 + dp[i][j]
            else:
                dp[i+1][j+1] = max(dp[i+1][j], dp[i][j+1])
    return dp[len(p)][len(q)]
```

3 Geometry

3.1 Circle Intersection

```
struct Circle {
  int x, y, r;
};

inline bool is_inside(Circle &a, Circle &b) {
  double d = sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));

  return d <= a.r + b.r || d <= a.r - b.r || d <= b.r - a.r;
}
```

3.2 Circuncenter

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

using namespace std;

struct Point {
  double x, y;
  Point(double x, double y) : x{x}, y{y} {}
};

inline double distance(const Point &p1, const Point &p2) {
  double x = p1.x - p2.x, y = p1.y - p2.y;
  return sqrt(x * x + y * y);
}

// Returns point equidistant to all vertices of the triangle
Point circuncenter(Point &A, Point &B, Point &C) {

  // LINE AB
  // ax + by = c
  double a = B.y - A.y, b = A.x - B.x;
  double c = a * A.x + b * A.y;

  // LINE BC
  double e = C.y - B.y, f = B.x - C.x;
  double g = e * B.x + f * B.y;

  // convert AB to perpendicular bisector
  c = -b * (A.x + B.x) / 2 + a * (A.y + B.y) / 2;
  b = exchange(a, -b);

  // convert BC to perpendicular bisector
  g = -f * (B.x + C.x) / 2 + e * (B.y + C.y) / 2;
  f = exchange(e, -f);

  double determinant = a * f - e * b;
  if (determinant == 0)
    return Point(1e9, 1e9);
  return Point((f * c - b * g) / determinant, (a * g - e * c) / determinant);
}
```

3.3 Tetrahedron Volume

```
#include <bits/stdc++.h>
using namespace std;
using pii = pair<int, int>;
using pdd = pair<double, double>;
using ll = long long;
```

```cpp
using ull = unsigned long long;

template <typename T> struct Point {
  T x, y, z;
  Point() : x{0}, y{0}, z{0} {}
  Point(T x, T y, T z) : x{x}, y{y}, z{z} {}

  T dot(Point<T> &other) { return x * other.x + y * other.y + z * other.z; }

  T cross(Point<T> &other) {
    return x * other.x + x * other.y + x * other.z + y * other.x + y * other.y +
           y * other.z + z * other.x + z * other.y + z * other.z;
  }

  friend istream &operator>>(istream &is, Point<T> &p) {
    is >> p.x >> p.y >> p.z;
    return is;
  }

  Point<T> operator-(Point<T> &other) {
    return Point<T>(x - other.x, y - other.y, z - other.z);
  }
};

double volume(Point<double> &p1, Point<double> &p2, Point<double> &p3,
              Point<double> &p4) {
  auto pa = p1 - p4, pb = p2 - p4, pc = p3 - p4;

  double determinant = pa.x * (pb.y * pc.z - pc.y * pb.z) -
                       pb.x * (pa.y * pc.z - pc.y * pa.z) +
                       pc.x * (pa.y * pb.z - pb.y * pa.z);
  return abs(determinant) / 6.0;
}

int main() {
  ios::sync_with_stdio(0);
  cin.tie(0);

  int t;
  cin >> t;

  while (t--) {
    Point<double> p1, p2, p3, p4;
    cin >> p1 >> p2 >> p3 >> p4;
    cout << fixed << setprecision(6) << volume(p1, p2, p3, p4) << '\n';
  }
}
```

4 Graphs

4.1 Class Graph

```python
class Graph:

    def __init__(self):
        self.graph = {}

    def add_unidirectional_edge(self, u, v):
        if u in self.graph:
            self.graph[u].add(v)
        else:
            self.graph[u] = {v}

    def add_bidirectional_edge(self, u, v):
        self.add_unidirectional_edge(u, v)
        self.add_unidirectional_edge(v, u)

    def get_next_vertices(self, u):
        if u in self.graph:
            return self.graph[u]
        else:
            return set()
```

4.2 Dijkstra

```cpp
// The graph's pair<int, int> should be {distance, vertex}

#include <bits/stdc++.h>

using namespace std;

#define INF 1000000000
#define ii pair<int, int>

vector<vector<ii>> graph;

vector<int> dijkstra(int s, int n) {
    vector<int> d(n+1, INF);
    d[s] = 0;

    priority_queue<ii, vector<ii>, greater<ii>> pq;
    pq.push({0, s});

    while (!pq.empty()) {
        auto [dist, v] = pq.top(); pq.pop();

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

        for (auto [len, next] : graph[v]) {
            int newDist = dist + len;
            if (newDist < d[next]) {
                d[next] = newDist;
                pq.push({newDist, next});
            }
        }
    }
    return d;
}
```

4.3 Floyd Warshall

```python
'''
Returns the distances between all vertex pairs in O(n^3)
Overrides the matrix "m" of distances between vertices.

IMPORTANT: the matrix "m" must have value INFINITY for
vertices that don't have a direct connection between them.
'''

def floyd_warshall(m):
    n = len(m)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                m[i][j] = min(m[i][j], m[i][k] + m[k][j])
    return m
```

4.4 Max Flow

```cpp
#include <bits/stdc++.h>
using namespace std;
int n;
vector<vector<int>> capacity;
vector<vector<int>> adj;

int bfs(int s, int t, vector<int> &parent) {
  fill(parent.begin(), parent.end(), -1);
  parent[s] = -2;
  queue<pair<int, int>> q;
  q.push({s, 1e9});

  while (!q.empty()) {
    int cur = q.front().first;
    int flow = q.front().second;
    q.pop();

    for (int next : adj[cur]) {
      if (parent[next] == -1 && capacity[cur][next]) {
        parent[next] = cur;
        int new_flow = 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;
  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;
}
```

4.5 Prim

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

#ifdef DEBUG
#define PRINT(s) std::cout << s << '\n';
#endif

using namespace std;
using pii = pair<int, int>;
using ull = unsigned long long;
using ll = long long;

// O(E log V)
// For finding minimum spanning trees
ull prim(vector<vector<pii>> &adj, vector<bool> &visited, int og,
         int &num_visited) {
  priority_queue<pii, vector<pii>, greater<pii>> pq;

  ull cost = 0;
  // vector<bool> visited(adj.size(), false);

  visited[og] = true;
  int n = adj.size();

  for (auto &[w, v] : adj[og])
    if (!visited[v])
      pq.push({w, v});

  while (!pq.empty() && num_visited != n - 1) {
    auto [w, u] = pq.top();
    pq.pop();

    if (visited[u])
      continue;

    visited[u] = true;
    num_visited++;
    cost += w;

    for (auto &[wv, v] : adj[u])
      if (!visited[v])
        pq.push({wv, v});
  }

  return cost;
}
```

5 Linear Sorting

5.1 Radix Sort

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

using namespace std;
using ll = long long;
using ull = unsigned long long;

// If numbers are too large, MAYBE increase base.
// Once I had to use 2^15 to get AC
// Some numbers are using ll. In practice, this is only needed for very large
// bases.
// int base = 32768;
int base = 512; // IDK a good value

int get_digit(int a, int divisor) { return a / divisor % base; }

bool cmp(int a, int b, int divisor) {
  return get_digit(a, divisor) < get_digit(b, divisor);
}

void counting_sort(vector<int> &vec, vector<int> &output, int divisor) {
  int l = INT32_MAX, r = 0;

  vector<int> aux(vec.begin(), vec.end());
  for (auto &v : aux) {
    v = get_digit(v, divisor);
    l = min(l, v);
    r = max(r, v);
  }

  vector<int> f(r - l + 1);

  for (auto &v : aux)
    ++f[v - l];

  for (int i = 1; i < f.size(); ++i)
    f[i] = f[i - 1] + f[i];

  for (ll i = vec.size() - 1; i >= 0; --i) {
    int d = aux[i];
    output[f[d - l] - 1] = vec[i];
    f[d - l]--;
  }
}

void radix_sort(vector<int> &vec) {
  auto [il, ir] = minmax_element(vec.begin(), vec.end());
  int l = *il, r = *ir;

  int num_digits = 1;
  int tmp = r;
  while (tmp >= base) {
    num_digits++;
    tmp = tmp / base;
  }

  vector<int> a(vec.begin(), vec.end()), b(vec.begin(), vec.end());

  auto *output = &a;
  auto *aux = &b;

  int divisor = 1;
  for (int i = 0; i < num_digits; ++i) {
    swap(aux, output);
    counting_sort(*aux, *output, divisor);
    divisor *= base;
  }

  for (int i = 0; i < vec.size(); ++i)
    vec[i] = (*output)[i];
}
```

6 Math

6.1 Eratostenes

```
#Returns ascending list of primes until "n", inclusive.
def primes(n):
    is_prime = [True] * (n + 1)
    primes = [2]

    for i in range(3, n + 1, 2):
        if is_prime[i]:
            primes.append(i)
            for j in range(i + i, n + 1, i):
                is_prime[j] = False
    return primes
```

6.2 Factorize

```
'''
```

```
Returns ascending list of prime factors of "n".
Repetitions allowed.
'''
def factorize(n):
    factors = []
    while n % 2 == 0:
        factors.append(2)
        n //= 2
    i = 3
    while n > 1:
        while n % i == 0:
            factors.append(i)
            n //= i
        i += 2
    return factors
```

6.3 Is Prime

```
from math import sqrt


def is_prime(n):
    sq = int(sqrt(n))

    if n % 2 == 0:
        return False
    for i in range(3, sq + 1, 2):
        if n % i == 0:
            return False
    return True
```

6.4 Mdc E Mmc

```
def mdc(a, b):
    dividendo = a
    divisor = b
    if b > a:
        dividendo = b
        divisor = a
    while 1:
        resto = dividendo % divisor
        if resto == 0:
            break
        dividendo, divisor = divisor, resto
    return divisor

#Dont use without defining mdc().
def mmc(a, b):
    return a * b // mdc(a, b)
```

7 String Algorithms

7.1 Knuth Morris Pratt

```
'''
Returns list with the "string" indexes on which the pattern "substr" starts.
'''
def build_lps_list(sub):
    lps = [0] * len(sub)

    for i in range(1, len(sub)):
        j = lps[i-1]
        while j > 0 and sub[i] != sub[j]:
            j = lps[j-1]
        if sub[i] == sub[j]:
            j += 1
        lps[i] = j
    return lps


def knuth_morris_pratt(substr, string):
    lps = build_lps_list(substr)
    i = 0
    j = 0
    res = []
    while i < len(string):
        if string[i] == substr[j]:
            i += 1
            j += 1
            if j == len(substr):
                res.append(i-j)
                j = lps[j-1]
        elif j == 0:
            i += 1
        else:
            j = lps[j-1]
    return res
```

7.2 Longest Palindromic Substring

```
'''
Manacher's algorithm. There are two implementations of it below.
First implementation:
 - Returns the longest palindromic substring of "s" with smallest starting index.
Second implementation:
 - Returns the length of the longest palindromic substring of "s" (a little bit
   faster because doesn't need to slice the substring out of "s").
All of this in O(n) time complexity.
First time writing this algo. Do not ask me how it works.
'''
#Returns the actual substring.
def longest_palindromic_substring(s):
    str = "#" + "#".join(s) + "#"
    c = 0
    r = 0
    lps = [0] * len(str)
    best_length = 0
    best_idx = 0

    for i in range(1, len(str) - 1):
        if i < r:
            lps[i] = min(r-i, lps[2*c - i])
        while len(str) - 1 - lps[i] > i and str[i + 1 + lps[i]] == str[i - 1 - lps[i]]:
            lps[i] += 1
        if lps[i] > best_length:
            best_length = lps[i]
            best_idx = i
        if i + lps[i] > r:
            c = i
            r = i + lps[i]
    return s[(best_idx - best_length)//2 : (best_idx + best_length)//2]

#Returns the length of the substring.
def longest_palindromic_substring(s):
    str = "#" + "#".join(s) + "#"
    c = 0
    r = 0
    lps = [0] * len(str)
    best_length = 0

    for i in range(1, len(str) - 1):
        if i < r:
            lps[i] = min(r-i, lps[2*c - i])
        while len(str) - 1 - lps[i] > i and str[i + 1 + lps[i]] == str[i - 1 - lps[i]]:
            lps[i] += 1
        if lps[i] > best_length:
            best_length = lps[i]
        if i + lps[i] > r:
            c = i
            r = i + lps[i]
    return best_length
```

8 Utils

8.1 Binary Search

```
'''
Returns index of target, in ascending iterables.
For descending iterables, swap "<" with ">".
If number doesn't exist, returns -1.
'''
def binary_search(iterable, target, down_idx=0, top_idx=None):
    if top_idx is None:
        top_idx = len(iterable) - 1
    while down_idx <= top_idx:
        cur = (down_idx + top_idx) // 2
        if iterable[cur] == target:
            return cur
        elif iterable[cur] < target: #swap here
            down_idx = cur + 1
        else:
            top_idx = cur - 1
    return -1


'''
Returns index of smallest number in iterable bigger than or equal
to target, in ascending iterables.
Swapping ">=" with "<=" returns index of biggest number in iterable
smaller than or equal to target, in descending iterables.
If number doesn't exist, returns -1.
'''
def binary_search(iterable, target, down_idx=0, top_idx=None):
    if top_idx is None:
        top_idx = len(iterable) - 1
    res = -1

    while down_idx <= top_idx:
        cur = (down_idx + top_idx) // 2
        if iterable[cur] >= target: #swap here
            res = cur
            top_idx = cur - 1
        else:
            down_idx = cur + 1
    return res


'''
Returns index of smallest number in iterable bigger than or equal
to target, in descending iterables.
Swapping ">=" with "<=" returns index of biggest number in iterable
smaller than or equal to target, in ascending iterables.
If number doesn't exist, returns -1.
'''
def binary_search(iterable, target, down_idx=0, top_idx=None):
    if top_idx is None:
        top_idx = len(iterable) - 1
    res = -1

    while down_idx <= top_idx:
        cur = (down_idx + top_idx) // 2
        if iterable[cur] >= target: #swap here
            res = cur
            down_idx = cur + 1
        else:
            top_idx = cur - 1
    return res
```

8.2 Binary Search For Smallest Possible Value

```
using namespace std;


bool valid(ull time, ull goal,vull& machines) {
  ull sum = 0;
  for (auto& m : machines)
    sum += time/m;

  return sum >= goal;
}


int main() {
  fast_io();

  ull n, t;
  cin >> n >> t;

  vull machines;
  while (n--) {
    ull tmp;
    cin >> tmp;
    machines.push_back(tmp);
  }


  ull boundary = t*(*max_element(machines.begin(), machines.end())) + 1;
  DEBUG(boundary);
  ull k = 0;
  for (ull b = boundary/2; b >= 1; b /= 2) {
    DEBUG(valid(k+b, t,machines));
    while (!valid(k+b, t,machines)) k+=b;
  }

  cout << k+1 << '\n';


}
```

8.3 Fast Io

```
import sys

input = lambda: sys.stdin.readline().removesuffix('\n')
print = lambda s="", end="\n": sys.stdout.write(str(s)+end)
```

8.4 Inversion Counting

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

using namespace std;
using pii = pair<int, int>;
using ll = long long;
using ull = unsigned long long;

struct FenwickTree {
  FenwickTree(int n) { ft.assign(n + 1, 0); }

  FenwickTree(vector<int> &vec) {
    ft.assign(vec.size() + 1, 0);
    for (int i = 0; i < vec.size(); ++i)
      update(i + 1, vec[i]);
  }

  inline int ls_one(int x) { return x & (-x); }

  int query(int r) {
    int sum = 0;
    while (r) {
      sum += ft[r];
      r -= ls_one(r);
    }
    return sum;
  }

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

  void update(int i, int v) {
    while (i < ft.size()) {
      ft[i] += v;
      i += ls_one(i);
    }
  }

  // Finds smallest index i on FenwickTree such that query(1, i) >= rank.
  // I.e: smallest i for [1, i] >= k
  int select(long long k) { // O(log^2 m)
    int lo = 1, hi = ft.size() - 1;
    for (int i = 0; i < 30; ++i) {
      int mid = (lo + hi) / 2;
      (query(1, mid) < k) ? lo = mid : hi = mid;
    }
    return hi;
  }

  vector<int> ft;
};

int main() {
  ios::sync_with_stdio(0);
  cin.tie(0);

  int n;
  while (cin >> n && n) {
    vector<int> seq(n);
    for (auto &v : seq)
      cin >> v;

    FenwickTree ft(n);

    int inv = 0;
    for (int i = 0; i < n; ++i) {
      inv += ft.query(seq[i] + 1, n);
      ft.update(seq[i], 1);
    }
    cout << (inv % 2 == 0 ? "Carlos\n" : "Marcelo\n");
  }
}
```

8.5 Inversion Counting

```
"""
Conta inversões presentes em um array

3,2,4,5

3 e 2 representam uma inversão (2 está a frente de 3, mas é menor que ele)
"""


from fenwick_tree import FenwickTree


def normalize(iteratable) -> dict[any, int]:
    """
    Cria um dicionário mapeando cada elemento do array para um número no intervalo [1, len(iteratable)
        ]
    """
    sorted_iteratable = sorted(iteratable)

    mp = {}
    num = 1

    for val in sorted_iteratable:
        if val not in mp:
            mp[val] = num
            num += 1
    return mp

def inversion_count(iteratable) -> int:
    """
    conta a quantidade total de inversões encontradas no array.
    """
    """
    A fenwick tree conta a frequência de elementos encontrados no array até o momento, permitindo
    a realização de queries para saber quantos valores já apareceram em um determinado intervalo de nú
        meros.
    por exemplo:
        suponha que um loop itere sobre os valores 5 4 3 2 1.

        na terceira iteração do for loop (quando o valor for 3),
        a árvore indicará que foram encontrados dois valores no intervalo [3:5]

    isso permite encontrar inversões de forma rápida, uma vez que tudo que precisamos fazer para
        encontrar todas
    as inversões de um número n é descobrir quantos números maiores que ele aparecem antes dele.
    i.e: bast fazer uma query a fenwick tree no intervalo [n:len(iteratable)].
    """

    ft = FenwickTree(len(iteratable))
    mp = normalize(iteratable)
    inv = 0
    for val in iteratable:
        inv += ft.query(mp[val], len(iteratable))
        ft.update(mp[val], 1)
    return inv
```

8.6 Random Number Generation

```
#include <random>

using namespace std;

const int L = 1;
const int R = 1e9;

default_random_engine gen;

uniform_int_distribution<int> distribution(L, R);

int num() { return distribution(gen); }
```