Team Notebook

Pontificia Universidad Católica de Chile - Masterkrab

1 Data-structures

1.1 segment tree lazy

```
template <class T1, class T2, T1 merge(T1, T1),
          void pushUpdate(T2 parent, T2 &child, int, int,
int, int),
          void applyUpdate(T2 update, T1 &node, int, int)>
struct SegmentTreeLazy {
  int n;
  vector<T1> tree;
  vector<T2> lazy;
  vector<bool> isUpdated;

  void build(int i, int left, int right, const vector<T1>
&values) {
    if (left == right) {
      tree[i] = values[left];
      return;
    }

    int mid = (left + right) >> 1;

    build(i << 1, left, mid, values);
    build(i << 1 | 1, mid + 1, right, values);

    tree[i] = merge(tree[i << 1], tree[i << 1 | 1]);
  }

  SegmentTreeLazy(const vector<T1> &values) {
    n = values.size();
    int size = n << 2 | 3;

    tree.resize(size);
    lazy.resize(size);
    isUpdated.resize(size);

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

  SegmentTreeLazy() {}

  void push(int i, int left, int right) {
    if (!isUpdated[i])
      return;

    applyUpdate(lazy[i], tree[i], left, right);

    if (left != right) {
      if (isUpdated[i << 1])
        pushUpdate(lazy[i], lazy[i << 1], left, right, left,
                   (left + right) / 2);
      else
        lazy[i << 1] = lazy[i];

      if (isUpdated[i << 1 | 1])
        pushUpdate(lazy[i], lazy[i << 1 | 1], left, right,
                   (left + right) / 2 + 1, right);
      else
        lazy[i << 1 | 1] = lazy[i];

      isUpdated[i << 1] = 1;
      isUpdated[i << 1 | 1] = 1;
    }

    isUpdated[i] = false;
  }

  void update(int i, int left, int right, int queryLeft, int
queryRight,
              T2 &value) {
    if (left >= queryLeft and right <= queryRight) {
      if (isUpdated[i])
        pushUpdate(value, lazy[i], queryLeft, queryRight,
left, right);
      else
        lazy[i] = value;

      isUpdated[i] = true;
    }

    push(i, left, right);

    if (left > queryRight or right < queryLeft)
      return;

    if (left >= queryLeft and right <= queryRight)
      return;

    update(i << 1, left, (left + right) >> 1, queryLeft,
queryRight, value);
    update(i << 1 | 1, (left + right) / 2 + 1, right,
queryLeft, queryRight,
           value);

    tree[i] = merge(tree[i << 1], tree[i << 1 | 1]);
  }

  void update(int left, int right, T2 value) {
    update(1, 0, n - 1, left, right, value);
  }

  T1 query(int i, int left, int right, int queryLeft, int
queryRight) {
    push(i, left, right);

    if (queryLeft <= left and right <= queryRight)
      return tree[i];

    int mid = (left + right) >> 1;

    if (mid < queryLeft)
      return query(i << 1 | 1, mid + 1, right, queryLeft,
queryRight);

    if (mid >= queryRight)
      return query(i << 1, left, mid, queryLeft,
queryRight);

    return merge(query(i << 1, left, mid, queryLeft,
queryRight),
                 query(i << 1 | 1, mid + 1, right,
queryLeft, queryRight));
  }

  T1 query(int left, int right) { return query(1, 0, n - 1,
left, right); }
};
```

1.2 segment tree

```
template <class T, T merge(T, T)> struct SegmentTree {
  int n;
  vector<T> tree;
  SegmentTree() {}
  SegmentTree(vector<T> &values) {
    n = values.size();
    tree.resize(n << 1);
    for (int i = 0; i < n; i++)
      tree[i + n] = values[i];
    for (int i = n - 1; i > 0; i--)
      tree[i] = merge(tree[i << 1], tree[i << 1 | 1]);
  }
  void update(int position, T value) {
    position += n;
    tree[position] = value;
    for (position >>= 1; position > 0; position >>= 1)
      tree[position] = merge(tree[position << 1],
tree[position << 1 | 1]);
  }
  T query(int left,
          int right) // [left, right]
  {
    bool hasAnswer = false;
    T answer = T();
    for (left += n, right += n + 1; left < right; left >>=
1, right >>= 1) {
      if (left & 1) {
        answer = hasAnswer ? merge(answer, tree[left]) :
tree[left];
        hasAnswer = true;
        left++;
      }
      if (right & 1) {
```

```
        right--;
        answer = hasAnswer ? merge(answer, tree[right]) :
tree[right];
        hasAnswer = true;
      }
    }
    return answer;
  }
};
```

1.3 ordered set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

template <typename T>
using ordered_set =
    tree<T, null_type, less<T>, rb_tree_tag,
tree_order_statistics_node_update>;

template <typename T>
using ordered_multiset = tree<T, null_type, less_equal<T>,
rb_tree_tag,

tree_order_statistics_node_update>;

template <typename T> void erase(ordered_multiset<T>
&values, T value) {
  int rank = values.order_of_key(value);
  auto it = values.find_by_order(rank);
  values.erase(it);
}
```

1.4 merge sort tree

```
struct MergeSortTree {
  int n;
  vector<vector<int>> tree;

  vector<int> merge(const vector<int> &left, const
vector<int> &right) {
    vector<int> result(left.size() + right.size());

    int i = 0, j = 0;

    for (int &x : result) {
      if (i == left.size())
        x = right[j++];
      else if (j == right.size())
        x = left[i++];
      else if (left[i] < right[j])
        x = left[i++];
      else
        x = right[j++];
    }

    return result;
  }

  void build(int i, int left, int right, const vector<int>
&values) {
    if (left == right) {
      tree[i] = {values[left]};
      return;
    }

    int mid = (left + right) >> 1;

    build(i << 1, left, mid, values);
    build(i << 1 | 1, mid + 1, right, values);

    tree[i] = merge(tree[i << 1], tree[i << 1 | 1]);
  }

  MergeSortTree(const vector<int> &values) {
    n = values.size();
    tree.resize(n << 2 | 3);
    build(1, 0, n - 1, values);
  }

  int query(int i, int left, int right, int queryLeft, int
queryRight, int min,
            int max) {
    if (left > queryRight or right < queryLeft)
      return 0;

    if (left >= queryLeft and right <= queryRight) {
      auto leftIterator = lower_bound(tree[i].begin(),
tree[i].end(), min);
      auto rightIterator = upper_bound(tree[i].begin(),
tree[i].end(), max);

      return distance(leftIterator, rightIterator);
    }

    int mid = (left + right) >> 1;

    return query(i << 1, left, mid, queryLeft, queryRight,
min, max) +
           query(i << 1 | 1, mid + 1, right, queryLeft,
queryRight, min, max);
  }

  int query(int left, int right, int min, int max) {
    return query(1, 0, n - 1, left, right, min, max);
  }
};
```

2 Math

2.1 128bits

```
#if defined(__LP64__) || defined(_WIN64)
using int128 = __int128;
using uint128 = __uint128_t;
#else
using int128 = ll;
using uint128 = ull;
#endif

template <typename T>
typename enable_if<is_same<T, int128>::value || is_same<T,
uint128>::value,
                   istream &>::type
operator>>(istream &is, T &value) {
  string input;
  is >> input;

  value = 0;

  for (char character : input)
    if (isdigit(character))
      value = 10 * value + character - '0';

  if (input[0] == '-')
    value *= -1;

  return is;
}

template <typename T>
typename enable_if<is_same<T, int128>::value || is_same<T,
uint128>::value,
                   ostream &>::type
operator<<(ostream &os, T value) {
  string output;

  bool isNegative = value < 0;

  if (isNegative) {
    value *= -1;
    output += '-';
  }

  do {
    output += value % 10 + '0';
    value /= 10;
  } while (value > 0);

  reverse(output.begin() + isNegative, output.end());
```

```
  return os << output;
}
```

2.2 gcd lcm

```
#define gcd __gcd

template <typename T>
typename enable_if<is_integral<T>::value, T>::type lcm(T a,
T b) {
  return a / gcd(a, b) * b;
}
```

2.3 system mod 2

```
struct Mod2System {
  vector<vector<int>> matrix;
  int rows, columns, variables;
  vector<int> pivotPosition, freeColumns;

  Mod2System(vector<vector<int>> matrixTosolve) {
    matrix = matrixTosolve;
    rows = matrix.size();
    columns = (rows > 0 ? matrix[0].size() : 0);
    variables = columns - 1;

    pivotPosition = vector<int>(rows, -1);

    int pivotRow = 0;
    for (int pivotColumn = 0; pivotColumn < variables &&
pivotRow < rows;
         pivotColumn++) {
      int pivotIndex = pivotRow;

      while (pivotIndex < rows && matrix[pivotIndex]
[pivotColumn] == 0)
        pivotIndex++;

      if (pivotIndex == rows)
        continue;

      swap(matrix[pivotRow], matrix[pivotIndex]);
      pivotPosition[pivotRow] = pivotColumn;

      for (int i = 0; i < rows; i++)
        if (i != pivotRow && (matrix[i][pivotColumn] & 1))
          for (int j = pivotColumn; j < columns; j++)
            matrix[i][j] = matrix[i][j] ^ matrix[pivotRow]
[j];

      pivotRow++;
    }

    vector<bool> isPivot(variables);
    for (int position : pivotPosition)
      if (position != -1)
        isPivot[position] = true;

    for (int j = 0; j < variables; j++)
      if (!isPivot[j])
        freeColumns.push_back(j);
  }

  vector<int> findOneSolution() {
    vector<int> solution(variables);

    if (freeColumns.empty())
      return solution;

    solution[freeColumns[0]] = 1;

    for (int i = 0; i < rows; i++) {
      if (pivotPosition[i] == -1)
        continue;

      int position = pivotPosition[i];
      solution[position] = matrix[i][freeColumns[0]];
    }

    return solution;
  }

  vector<vector<int>> findSolutions(int maxSolutions = 1e5)
{
    if (freeColumns.empty())
      return {vector<int>(variables)};

    vector<vector<int>> solutions;

    for (int freeColumn : freeColumns) {
      vector<int> solution(variables);
      solution[freeColumn] = 1;

      for (int i = 0; i < rows; i++) {
        if (pivotPosition[i] == -1)
          continue;

        int pivotCol = pivotPosition[i];
        solution[pivotCol] = matrix[i][freeColumn];
      }

      solutions.push_back(solution);

      if (solutions.size() == maxSolutions)
        break;
    }

    return solutions;
  }

  bool isTrivial(vector<int> &solution) {
    return count(solution.begin(), solution.end(), 1) == 0;
  }

  bool onlyHasTrivialSolution() {
    if (freeColumns.empty())
      return true;

    vector<int> solution = findOneSolution();

    return isTrivial(solution);
  }
};
```

2.4 sieve

```
struct Sieve {
  vector<int> primes, smallestFactor;

  Sieve(int n) {
    smallestFactor.resize(n + 1);

    for (ll i = 2; i <= n; i++) {
      if (smallestFactor[i] == 0) {
        smallestFactor[i] = i;
        primes.push_back(i);
      }

      for (ll prime : primes) {
        if (prime * i > n || prime > smallestFactor[i])
          break;

        smallestFactor[prime * i] = prime;
      }
    }
  }

  bool getIsPrime(int n) { return smallestFactor[n] == n; }

  map<int, int> factorize(int n) {
    map<int, int> factors;

    while (n > 1) {
      int factor = smallestFactor[n];

      factors[factor]++;
      n /= factor;
    }

    return factors;
  }
};
```

2.5 gauss mod 2

```
vector<vector<int>>
solveHomogeneousSystemMod2(vector<vector<int>> &A,
                                              int
maxSolutions = 1e5) {
  int rows = A.size(), columns = A[0].size();

  int variables = columns - 1;

  vector<int> pivotPosition(rows, -1);

  int pivotRow = 0;

  for (int pivotColumn = 0; pivotColumn < variables &&
pivotRow < rows;
       pivotColumn++) {
    int pivotIndex = pivotRow;

    while (pivotIndex < rows && A[pivotIndex][pivotColumn]
== 0)
      pivotIndex++;

    if (pivotIndex == rows)
      continue;

    swap(A[pivotRow], A[pivotIndex]);

    pivotPosition[pivotRow] = pivotColumn;

    for (int i = 0; i < rows; i++)
      if (i != pivotRow && A[i][pivotColumn] & 1)
        for (int j = pivotColumn; j < columns; j++)
          A[i][j] = A[i][j] ^ A[pivotRow][j];

    pivotRow++;
  }

  vector<bool> isPivot(variables);

  for (int position : pivotPosition) {
    if (position == -1)
      continue;

    isPivot[position] = true;
  }

  vector<int> freeColumns;

  for (int j = 0; j < variables; j++)
    if (!isPivot[j])
      freeColumns.push_back(j);

  if (freeColumns.empty())
    return {vector<int>(variables)};

  vector<vector<int>> solutions;

  for (int freeColumn : freeColumns) {
    vector<int> solution(variables);

    solution[freeColumn] = 1;

    for (int i = 0; i < rows; i++) {
      if (pivotPosition[i] == -1)
        continue;

      int positionColumn = pivotPosition[i];
      solution[positionColumn] = A[i][freeColumn];
    }

    solutions.push_back(solution);

    if (solutions.size() == maxSolutions)
      break;
  }

  return solutions;
}
```

3 Algorithms

3.1 ternary search

```
template <typename T = double>
typename enable_if<is_floating_point<T>::value>
ternarySearch(function<T(T)> calculate, T left, T right,
bool searchMin = true,
              T epsilon = 1e-9) { // [left, right]
  while (right - left > epsilon) {
    T midA = left + (right - left) / 3, midB = right -
(right - left) / 3;

    T valueA = calculate(midA), valueB = calculate(midB);

    if (searchMin)
      valueA *= -1, valueB *= -1;

    if (valueA < valueB)
      left = midA;
    else
      right = midB;
  }

  return calculate(left);
}

template <typename T = int>
typename enable_if<is_integral<T>::value>
ternarySearch(function<T(T)> calculate, T left, T right,
bool searchMin = true,
              T epsilon = 5) { // [left, right]
  while (right - left > epsilon) {
    T midA = left + (right - left) / 3, midB = right -
(right - left) / 3;

    T valueA = calculate(midA), valueB = calculate(midB);

    if (searchMin)
      valueA *= -1, valueB *= -1;

    if (valueA < valueB)
      left = midA;
    else
      right = midB;
  }

  T answer = calculate(left);

  for (T i = left + 1; i <= right; i++) {
    T value = calculate(i);

    if (searchMin)
      answer = min(answer, value);
    else
      answer = max(answer, value);
  }

  return answer;
}
```

3.2 prefix sum

```
template <typename T> struct PrefixSum {
  static_assert(is_arithmetic<T>::value);

  vector<T> prefix;

  PrefixSum(vector<T> &values) {
    int n = values.size();
    prefix.resize(n);

    prefix[0] = values[0];

    for (int i = 1; i < n; i++)
      prefix[i] = prefix[i - 1] + values[i];
  }

  T query(int left, int right) // [left, right]
  {
    return prefix[right] - (left > 0 ? prefix[left - 1] :
0);
```

```
  }
};
```

4 Graphs

4.1 mincostmaxflow

```
template <class Flow, class Cost> struct MinCostMaxFlow {
  struct Edge {
    int from, to;
    Flow capacity;
    Cost cost;
    Flow flow = 0;
  };

  int n, source, sink;
  vector<vector<int>> graph;
  vector<Edge> edges;
  vector<ll> distance, potential;
  vector<int> parent;

  MinCostMaxFlow() {}
  MinCostMaxFlow(int n, int source, int sink)
      : n(n), source(source), sink(sink), graph(n) {}

  void addEdge(int from, int to, Flow capacity, Cost cost) {
    graph[from].push_back(edges.size());
    edges.push_back({from, to, capacity, cost});

    graph[to].push_back(edges.size());
    edges.push_back({to, from, 0, -cost});
  }

  void calculateDistances() {
    if (potential.empty()) {
      potential.assign(n, 0);

      for (int i = 0; i < n - 1; ++i)
        for (Edge &edge : edges) {
          if (edge.flow >= edge.capacity)
            continue;

          potential[edge.to] =
              min(potential[edge.to], potential[edge.from] +
edge.cost);
        }
    }

    parent.assign(n, -1);
    distance.assign(n, numeric_limits<Cost>::max());

    distance[source] = 0;

    priority_queue<pair<Cost, int>> pq;
    pq.push({0, source});

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

      if (distance[node] < -current + potential[node])
        continue;

      for (int index : graph[node]) {
        Edge &edge = edges[index];

        Cost newDistance = distance[node] + edge.cost;

        if (edge.flow >= edge.capacity || newDistance >=
distance[edge.to])
          continue;

        distance[edge.to] = newDistance;
        parent[edge.to] = index;
        pq.push({potential[edge.to] - newDistance,
edge.to});
      }
    }

    potential = distance;
  }

  Flow calculateMaxFlow() {
    potential.clear();

    Flow total = 0;

    while (true) {
      calculateDistances();

      if (parent[sink] == -1)
        break;

      Flow pathFlow = numeric_limits<Flow>::max();

      for (int node = sink; parent[node] != -1; node =
edges[parent[node]].from)
        pathFlow = min(pathFlow,
                       edges[parent[node]].capacity -
edges[parent[node]].flow);

      for (int node = sink; parent[node] != -1;
           node = edges[parent[node]].from) {
        edges[parent[node]].flow += pathFlow;
        edges[parent[node] ^ 1].flow -= pathFlow;
      }

      total += pathFlow;
    }

    return total;
  }

  pair<Flow, vector<vector<int>>> searchMaxFlowPaths() {
    Flow maxFlow = calculateMaxFlow();
    vector<vector<int>> paths;

    while (true) {
      vector<int> path;
      int current = source;
      path.push_back(current);

      while (current != sink) {
        bool found = false;

        for (int index : graph[current]) {
          Edge &edge = edges[index];

          if (edge.flow == 0)
            continue;

          edge.flow -= 1;
          edges[index ^ 1].flow += 1;
          current = edge.to;
          path.push_back(current);
          found = true;
          break;
        }

        if (!found)
          break;
      }

      if (path.back() != sink)
        break;

      paths.push_back(path);
    }

    return {maxFlow, paths};
  }

  pair<Flow, vector<pii>> searchMinCutEdges() {
    Flow maxFlow = calculateMaxFlow();

    vector<bool> visited(n);

    function<void(int)> dfs = [&](int u) {
      visited[u] = true;

      for (int index : graph[u]) {
        Edge &edge = edges[index];
```

```
        if (visited[edge.to] || edge.flow == edge.capacity)
          continue;

        dfs(edge.to);
      }
    };

    dfs(source);

    vector<pii> minCutEdges;

    for (Edge &edge : edges) {
      if (edge.capacity == 0 || edge.flow != edge.capacity)
        continue;

      if (visited[edge.from] && !visited[edge.to])
        minCutEdges.emplace_back(edge.from, edge.to);
    }

    return {maxFlow, minCutEdges};
  };
};
```

4.2 dsu

```
struct DisjointUnionSet {
  int n;
  vector<int> parent, size;

  DisjointUnionSet(int n) : n(n) {
    parent.resize(n);
    size.resize(n);

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

  int Find(int u) {
    if (parent[u] == u)
      return u;

    return parent[u] = Find(parent[u]);
  }
  int Size(int u) { return size[Find(u)]; }
  void Union(int u, int v) {
    u = Find(u);
    v = Find(v);

    if (u == v)
      return;

    if (size[u] < size[v])
      swap(u, v);

    parent[v] = u;
    size[u] += size[v];
  }
  bool areConnected(int u, int v) { return Find(u) ==
Find(v); }
};
```

4.3 heavylightdecomposition

```
template <class ST, class T, T merge(T a, T b)> struct
HeavyLightDecomposition {
  vector<vector<int>> tree;
  vector<int> parent, depth, heavy, head, position, size;
  int currentPosition;
  ST segmentTree;

  HeavyLightDecomposition() {}
  HeavyLightDecomposition(vector<vector<int>> &tree) :
tree(tree) {
    int n = tree.size();

    parent.resize(n);
    depth.resize(n);
    heavy.assign(n, -1);
    head.resize(n);
    position.resize(n);
    size.resize(n);
    currentPosition = 0;

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

  int dfs(int node) {
    size[node] = 1;
    int maxSubtreeSize = 0;

    for (int child : tree[node]) {
      if (child == parent[node])
        continue;

      parent[child] = node;
      depth[child] = depth[node] + 1;

      int subtreeSize = dfs(child);
      size[node] += subtreeSize;

      if (subtreeSize <= maxSubtreeSize)
        continue;

      maxSubtreeSize = subtreeSize;
      heavy[node] = child;
    }
    return size[node];
  }

  void decompose(int node, int nodeHead) {
    head[node] = nodeHead;
    position[node] = currentPosition++;

    if (heavy[node] != -1)
      decompose(heavy[node], nodeHead);

    for (int child : tree[node])
      if (child != parent[node] && child != heavy[node])
        decompose(child, child);
  }

  int query(int a, int b) {
    T answer;
    bool hasValue = false;

    while (head[a] != head[b]) {
      if (depth[head[a]] > depth[head[b]])
        swap(a, b);

      T current = segmentTree.query(position[head[b]],
position[b]);

      answer = hasValue ? merge(answer, current) : current;
      hasValue = true;

      b = parent[head[b]];
    }

    if (depth[a] > depth[b])
      swap(a, b);

    T last = segmentTree.query(position[a], position[b]);

    return hasValue ? merge(answer, last) : last;
  }

  vector<T> sorted(vector<T> &values) {
    int n = values.size();

    vector<T> result(n);

    for (int i = 0; i < n; i++)
      result[position[i]] = values[i];

    return result;
  }

  void update(int node, T value)
{ segmentTree.update(position[node], value); }

  // void update(int subtree, T value) {
```

```
  //   segmentTree.update(position[subtree],
position[subtree] + size[subtree] -
  //   1,
  //                      value);
  // }
};
```

5 Strings

5.1 rolling hash

```
struct RollingHash {
  static const ll BASE = 31;
  static const ll MOD1 = 1e9 + 9, MOD2 = 1e9 + 7, MOD3 =
(119LL << 23) | 1;
  const vector<ll> MODS = {MOD1, MOD2, MOD3};

  vector<vector<ll>> prefixes, powers;
  int n;

  RollingHash(const string &text) {
    n = text.size();
    prefixes.assign(3, vector<ll>(n + 2));
    powers.assign(3, vector<ll>(n + 2, 1));

    for (int j = 0; j < 3; j++)
      for (int i = 1; i <= n + 1; i++)
        powers[j][i] = (powers[j][i - 1] * BASE) % MODS[j];

    for (int j = 0; j < 3; j++) {
      for (int i = 0; i < n; i++)
        prefixes[j][i + 1] = (prefixes[j][i] * BASE +
text[i]) % MODS[j];

      prefixes[j][n + 1] = (prefixes[j][n] * BASE) %
MODS[j];
    }
  }

  tuple<int, int, int> query(int left, int right) {
    vector<int> result(3);

    for (int i = 0; i < 3; i++) {
      ll hash = (prefixes[i][right + 1] -
                 prefixes[i][left] * powers[i][right - left
+ 1] + MODS[i]) %
                MODS[i];

      if (hash < 0)
        hash += MODS[i];

      result[i] = hash;
    }
    return {result[0], result[1], result[2]};
  }
};
```

6 General

6.1 template

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

using ll = long long;
using ull = unsigned long long;
using ld = long double;
using pii = pair<int, int>;
using pll = pair<ll, ll>;

const ll MOD = 1e9 + 7;
const ll FFT_MOD = 119 << 23 | 1;
const ld PI = acos(-1);

void solve() {}

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

  //   freopen("input.txt", "r", stdin);
  //   freopen("output.txt", "w", stdout);

  int t = 1;
  //   cin >> t;

  while (t--)
    solve();
}
```

6.2 complete template

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

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

template <typename T>
using ordered_set =
    tree<T, null_type, less<T>, rb_tree_tag,
tree_order_statistics_node_update>;

template <typename T>
using ordered_multiset = tree<T, null_type, less_equal<T>,
rb_tree_tag,
tree_order_statistics_node_update>;

template <typename T> void erase(ordered_multiset<T>
&values, T value) {
  int rank = values.order_of_key(value);
  auto it = values.find_by_order(rank);
  values.erase(it);
}

using ll = long long;
using ull = unsigned long long;
using ld = long double;
using pii = pair<int, int>;
using pll = pair<ll, ll>;
#define gcd __gcd

#if defined(__LP64__) || defined(_WIN64)
using int128 = __int128;
using uint128 = __uint128_t;
#else
using int128 = ll;
using uint128 = ull;
#endif

template <typename T>
typename enable_if<is_same<T, int128>::value || is_same<T,
uint128>::value,
                   istream &>::type
operator>>(istream &is, T &value) {
  string input;
  is >> input;

  value = 0;

  for (char character : input)
    if (isdigit(character))
      value = 10 * value + character - '0';

  if (input[0] == '-')
    value *= -1;

  return is;
}

template <typename T>
typename enable_if<is_same<T, int128>::value || is_same<T,
uint128>::value,
                   ostream &>::type
operator<<(ostream &os, T value) {
  string output;

  bool isNegative = value < 0;

  if (isNegative) {
```

```
    value *= -1;
    output += '-';
  }

  do {
    output += value % 10 + '0';
    value /= 10;
  } while (value > 0);

  reverse(output.begin() + isNegative, output.end());

  return os << output;
}

template <typename T>
typename enable_if<is_integral<T>::value, T>::type lcm(T a,
T b) {
  return a / gcd(a, b) * b;
}

const ll MOD = 1e9 + 7;
const ll FFT_MOD = 119 << 23 | 1;
const ld PI = acos(-1);

void solve() {}

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

  //   freopen("input.txt", "r", stdin);
  //   freopen("output.txt", "w", stdout);

  int t = 1;
  //   cin >> t;

  while (t--)
    solve();
}
```