Binary Exponentiation

```
/* Given three unsigned 64-bit integers x, n, and m, powmod() returns x raised to the power of n (modulo
      m). mulmod() returns x multiplied by n (modulo m). Despite the fact that both functions use
      unsigned 64-bit integers for arguments and intermediate calculations, arguments x and n must not
      exceed 2^63 - 1 (the maximum value of a signed 64-bit integer) for the result to be correctly
      computed without overflow.  Binary exponentiation, also known as exponentiation by squaring,
      decomposes the exponentiation into a logarithmic number of multiplications while avoiding overflow
      . To further prevent overflow in the intermediate squaring computations, multiplication is
      performed using a similar principle of repeated addition.

Time Complexity:
- O(log n) per call to mulmod() and powmod(), where n is the second argument.

Space Complexity:
- O(1) auxiliary.*/

typedef unsigned long long uint64;

uint64 mulmod(uint64 x, uint64 n, uint64 m) {
  uint64 a = 0, b = x % m;
  for (; n > 0; n >>= 1) {
    if (n & 1) {
      a = (a + b) % m;
    }
    b = (b << 1) % m;
  }
  return a % m;
}

uint64 powmod(uint64 x, uint64 n, uint64 m) {
  uint64 a = 1, b = x;
  for (; n > 0; n >>= 1) {
    if (n & 1) {
      a = mulmod(a, b, m);
    }
    b = mulmod(b, b, m);
  }
  return a % m;
}
```

Shortest Path (Dijkstra’s)

```
/* Given a starting node in a weighted, directed graph with nonnegative weights only, visit every
      connected node and determine the minimum distance to each such node. Optionally, output the
      shortest path to a specific destination node using the shortest-path tree from the predecessor
      array pred[]. dijkstra() applies to a global, pre-populated adjacency list adj[] which must only
      consist of nodes numbered with integers between 0 (inclusive) and the total number of nodes (
      exclusive), as passed in the function argument.  Since priority_queue is by default a max-heap, we
       simulate a min-heap by negating node distances before pushing them and negating them again after
      popping them. Alternatively, the container can be declared with the following template arguments
      (#include <functional> to access greater): priority_queue<pair<int, int>, vector<pair<int, int> >,
       greater<pair<int, int> > > pq;  Dijkstra's algorithm may be modified to support negative edge
      weights by allowing nodes to be re-visited (removing the visited array check in the inner for-loop
      ). This is known as the Shortest Path Faster Algorithm (SPFA), which has a larger running time of
      O(n*m) on the number of nodes and edges respectively. While it is as slow in the worst case as the
       Bellman-Ford algorithm, the SPFA still tends to outperform in the average case.

Time Complexity:
- O(m log n) for dijkstra(), where m is the number of edges and n is the numberof nodes.

Space Complexity:
- O(max(n, m)) for storage of the graph, where n is the number of nodes and mis the number of edges. - O
      (n) auxiliary heap space for dijkstra().*/

#include <queue>
#include <utility>
#include <vector>
using namespace std;

const int MAXN = 100, INF = 0x3f3f3f3f;
vector<pair<int, int> > adj[MAXN];
int dist[MAXN], pred[MAXN];

void dijkstra(int nodes, int start) {
  vector<bool> visit(nodes, false);
  for (int i = 0; i < nodes; i++) {
    dist[i] = INF;
    pred[i] = -1;
  }
  dist[start] = 0;
  priority_queue<pair<int, int> > pq;
  pq.push(make_pair(0, start));
  while (!pq.empty()) {
    int u = pq.top().second;
    pq.pop();
    visit[u] = true;
    for (int j = 0; j < (int)adj[u].size(); j++) {
      int v = adj[u][j].first;
      if (visit[v]) {
        continue;
      }
      if (dist[v] > dist[u] + adj[u][j].second) {
        dist[v] = dist[u] + adj[u][j].second;
        pred[v] = u;
        pq.push(make_pair(-dist[v], v));
      }
    }
  }
}
```

Shortest Path (Bellman-Ford)

```
/* Given a starting node in a weighted, directed graph with possibly negative weights, visit every
      connected node and determine the minimum distance to each such node. Optionally, output the
      shortest path to a specific destination node using the shortest-path tree from the predecessor
      array pred[]. bellman_ford() applies to a global, pre-populated edge list which must only consist
      of nodes numbered with integers between 0 (inclusive) and the total number of nodes (exclusive),
      as passed in the function argument.  This function will also detect whether the graph contains
      negative-weighted cycles, in which case there is no shortest path and an error will be thrown.

Time Complexity:
- O(n*m) per call to bellman_ford(), where n is the number of nodes and m is thenumber of edges.

Space Complexity:
- O(max(n, m)) for storage of the graph, where n is the number of nodes and m isthe number of edges. - O
      (n) auxiliary heap space for bellman_ford(), where n is the number of nodes.*/

#include <stdexcept>
#include <vector>
using namespace std;

struct edge { int u, v, w; };  // Edge from u to v with weight w.

const int MAXN = 100, INF = 0x3f3f3f3f;
vector<edge> e;
int dist[MAXN], pred[MAXN];

void bellman_ford(int nodes, int start) {
  for (int i = 0; i < nodes; i++) {
    dist[i] = INF;
    pred[i] = -1;
  }
  dist[start] = 0;
  for (int i = 0; i < nodes; i++) {
    for (int j = 0; j < (int)e.size(); j++) {
      if (dist[e[j].v] > dist[e[j].u] + e[j].w) {
        dist[e[j].v] = dist[e[j].u] + e[j].w;
        pred[e[j].v] = e[j].u;
      }
    }
  }
  // Optional: Report negative-weighted cycles.
  for (int i = 0; i < (int)e.size(); i++) {
    if (dist[e[i].v] > dist[e[i].u] + e[i].w) {
      throw runtime_error("Negative-weight cycle found.");
    }
  }
}
```

Shortest Path (Floyd-Warshall)

```
/* Given a weighted, directed graph with possibly negative weights, determine the minimum distance
      between all pairs of start and destination nodes in the graph. Optionally, output the shortest
      path between two nodes using the shortest-path tree precomputed into the parent[][] array.
      floyd_warshall() applies to a global adjacency matrix dist[][], which must be initialized using
      initialize() and subsequently populated with weights. After the function call, dist[u][v] will
      have been modified to contain the shortest path from u to v, for all pairs of valid nodes u and v.
        This function will also detect whether the graph contains negative-weighted cycles, in which
      case there is no shortest path and an error will be thrown.

Time Complexity:
- O(n^2) per call to initialize(), where n is the number of nodes.- O(n^3) per call to floyd_warshall().

Space Complexity:
- O(n^2) for storage of the graph, where n is the number of nodes.- O(n^2) auxiliary heap space for
      initialize() and floyd_warshall().*/

#include <stdexcept>
using namespace std;

const int MAXN = 100, INF = 0x3f3f3f3f;
int dist[MAXN][MAXN], parent[MAXN][MAXN];

void initialize(int nodes) {
  for (int i = 0; i < nodes; i++) {
    for (int j = 0; j < nodes; j++) {
      dist[i][j] = (i == j) ? 0 : INF;
      parent[i][j] = j;
    }
  }
}

void floyd_warshall(int nodes) {
  for (int k = 0; k < nodes; k++) {
    for (int i = 0; i < nodes; i++) {
      for (int j = 0; j < nodes; j++) {
        if (dist[i][j] > dist[i][k] + dist[k][j]) {
          dist[i][j] = dist[i][k] + dist[k][j];
          parent[i][j] = parent[i][k];
        }
      }
    }
  }
  // Optional: Report negative-weighted cycles.
  for (int i = 0; i < nodes; i++) {
    if (dist[i][i] < 0) {
      throw runtime_error("Negative-weight cycle found.");
    }
  }
}
```

Bridges, Cut-points, and Biconnectivity

```
/* Given an undirected graph, compute the following properties of the graph using Tarjan's algorithm.
     tarjan() applies to a global, pre-populated adjacency list adj[] which satisfies the precondition
     that for every node v in adj[u], node u also exists in adj[v]. Nodes in adj[] must be numbered
     with integers between 0 (inclusive) and the total number of nodes (exclusive), as passed in the
     function arguments. get_block_forest() applies to the global vector of biconnected components bcc
     [] which must have already been precomputed by a call to tarjan().  A bridge is an edge such that
     when deleted, the number of connected components in the graph is increased. An edge is a bridge if
      and only if it is not part of any cycle.  A cut-point (i.e. cut-node, or articulation point) is
     any node whose removal increases the number of connected components in the graph.  A biconnected
     component of a graph is a maximally biconnected subgraph. A biconnected graph is a connected and "
     non-separable" graph, meaning that if any node were to be removed, the graph will remain connected
     . Thus, a biconnected graph has no articulation points.  Any connected graph decomposes into a
     tree of biconnected components called the "block tree" of the graph. An unconnected graph will
     thus decompose into a "block forest."

Time Complexity:
- O(max(n, m)) per call to tarjan() and get_block_forest(), where n is thenumber of nodes and m is the
     number of edges.

Space Complexity:
- O(max(n, m)) for storage of the graph, where n the number of nodes and m isthe number of edges - O(n)
     auxiliary stack space for tarjan().- O(1) auxiliary stack space for get_block_forest().*/

#include <algorithm>
#include <vector>
using namespace std;

const int MAXN = 100;
int timer, lowlink[MAXN], tin[MAXN], comp[MAXN];
vector<bool> visit(MAXN);
vector<int> adj[MAXN], bcc_forest[MAXN];
vector<int> stack, cutpoints;
vector<vector<int> > bcc;
vector<pair<int, int> > bridges;

void dfs(int u, int p) {
  visit[u] = true;
  lowlink[u] = tin[u] = timer++;
  stack.push_back(u);
  int v, children = 0;
  bool cutpoint = false;
  for (int j = 0; j < (int)adj[u].size(); j++) {
    v = adj[u][j];
    if (v == p) {
      continue;
    }
    if (visit[v]) {
      lowlink[u] = min(lowlink[u], tin[v]);
    } else {
      dfs(v, u);
      lowlink[u] = min(lowlink[u], lowlink[v]);
      cutpoint |= (lowlink[v] >= tin[u]);
      if (lowlink[v] > tin[u]) {
        bridges.push_back(make_pair(u, v));
      }
      children++;
    }
  }
  if (p == -1) {
    cutpoint = (children >= 2);
  }
  if (cutpoint) {
    cutpoints.push_back(u);
  }
  if (lowlink[u] == tin[u]) {
    vector<int> component;
    do {
      v = stack.back();
      stack.pop_back();
      component.push_back(v);
    } while (u != v);
    bcc.push_back(component);
  }
}

void tarjan(int nodes) {
  bcc.clear();
  bridges.clear();
  cutpoints.clear();
  stack.clear();
  fill(lowlink, lowlink + nodes, 0);
  fill(tin, tin + nodes, 0);
  fill(visit.begin(), visit.end(), false);
  timer = 0;
  for (int i = 0; i < nodes; i++) {
    if (!visit[i]) {
      dfs(i, -1);
    }
  }
}

void get_block_forest(int nodes) {
  fill(comp, comp + nodes, 0);
  for (int i = 0; i < nodes; i++) {
    bcc_forest[i].clear();
  }
  for (int i = 0; i < (int)bcc.size(); i++) {
    for (int j = 0; j < (int)bcc[i].size(); j++) {
      comp[bcc[i][j]] = i;
    }
  }
  for (int i = 0; i < nodes; i++) {
    for (int j = 0; j < (int)adj[i].size(); j++) {
      if (comp[i] != comp[adj[i][j]]) {
        bcc_forest[comp[i]].push_back(comp[adj[i][j]]);
      }
    }
  }
}
```

Minimal Spanning Tree (Kruskal's)

```
/* Given a connected, undirected, weighted graph with possibly negative weights, its minimum spanning
     tree is a subgraph which is a tree that connects all nodes with a subset of its edges such that
     their total weight is minimized. kruskal() applies to a global, pre-populated adjacency list adj[]
      which must only consist of nodes numbered with integers between 0 (inclusive) and the total
     number of nodes (exclusive), as passed in the function argument. If the input graph is not
     connected, then this implementation will find the minimum spanning forest.

Time Complexity:
- O(m log n) per call to kruskal(), where m is the number of edges and n is thenumber of nodes.

Space Complexity:
- O(max(n, m)) for storage of the graph, where n the number of nodes and m isthe number of edges - O(n)
     auxiliary stack space for kruskal().*/

#include <algorithm>
#include <utility>
#include <vector>
using namespace std;

const int MAXN = 100;
vector<pair<int, pair<int, int> > > edges;
int root[MAXN];
vector<pair<int, int> > mst;

int find_root(int x) {
  if (root[x] != x) {
    root[x] = find_root(root[x]);
  }
  return root[x];
}

int kruskal(int nodes) {
  mst.clear();
  sort(edges.begin(), edges.end());
  int total_dist = 0;
  for (int i = 0; i < nodes; i++) {
    root[i] = i;
  }
  for (int i = 0; i < (int)edges.size(); i++) {
    int u = find_root(edges[i].second.first);
    int v = find_root(edges[i].second.second);
    if (u != v) {
      root[u] = root[v];
      mst.push_back(edges[i].second);
      total_dist += edges[i].first;
    }
  }
  return total_dist;
}
```

Max Flow (Edmonds-Karp)

```
/* Given a flow network with integer capacities, find the maximum flow from a given source node to a
      given sink node. The flow of a given edge u -> v is defined as the minimum of its capacity and the
      sum of the flows of all incoming edges of u. edmonds_karp() applies to a global adjacency list
      adj[] that will be modified by the function call.  The Edmonds-Karp algorithm will also support
      real-valued flow capacities. As such, this implementation will work as intended upon changing the
      appropriate variables to doubles.

Time Complexity:
- O(min(n*m^2, m*f)) per call to edmonds_karp(), where n is the number of nodes,m is the number of edges
      , and f is the maximum flow.

Space Complexity:
- O(max(n, m)) for storage of the flow network, where n is the number of nodesand m is the number of
      edges. */

#include <algorithm>
#include <queue>
#include <vector>
using namespace std;

struct edge { int u, v, rev, cap, f; };

const int MAXN = 100, INF = 0x3f3f3f3f;
vector<edge> adj[MAXN];

void add_edge(int u, int v, int cap) {
  adj[u].push_back((edge){u, v, (int)adj[v].size(), cap, 0});
  adj[v].push_back((edge){v, u, (int)adj[u].size() - 1, 0, 0});
}

int edmonds_karp(int nodes, int source, int sink) {
  int max_flow = 0;
  for (;;) {
    vector<edge*> pred(nodes, (edge*)0);
    queue<int> q;
    q.push(source);
    while (!q.empty() && !pred[sink]) {
      int u = q.front();
      q.pop();
      for (int j = 0; j < (int)adj[u].size(); j++) {
        edge &e = adj[u][j];
        if (!pred[e.v] && e.cap > e.f) {
          pred[e.v] = &e;
          q.push(e.v);
        }
      }
    }
    if (!pred[sink]) {
      break;
    }
    int flow = INF;
    for (int u = sink; u != source; u = pred[u]->u) {
      flow = min(flow, pred[u]->cap - pred[u]->f);
    }
    for (int u = sink; u != source; u = pred[u]->u) {
      pred[u]->f += flow;
      adj[pred[u]->v][pred[u]->rev].f -= flow;
    }
    max_flow += flow;
  }
  return max_flow;
}
```

Maximum Bipartite Matching (Hopcroft-Karp)

```
/* Given two sets of nodes A = {0, 1, ..., n1} and B = {0, 1, ..., n2} such that n1 < n2, as well as a
      set of edges E mapping nodes from set A to set B, find the largest possible subset of E containing
      no edges that share the same node. hopcroft_karp() applies to a global, pre-populated adjacency
      list adj[] which must only consist of nodes numbered with integers between 0 (inclusive) and the
      total number of nodes (exclusive), as passed in the function argument.

Time Complexity:
- O(m*sqrt(n1 + n2)) per call to hopcroft_karp(), where m is the number ofedges.

Space Complexity:
- O(max(n, m)) for storage of the graph, where n the number of nodes and m isthe number of edges. - O(n1
      + n2) auxiliary stack and heap space for hopcroft_karp().*/

#include <algorithm>
#include <queue>
#include <vector>
using namespace std;

const int MAXN = 100;
vector<int> adj[MAXN];
vector<bool> used(MAXN), visit(MAXN);
int match[MAXN], dist[MAXN];

void bfs(int n1, int n2) {
  fill(dist, dist + n1, -1);
  queue<int> q;
  for (int u = 0; u < n1; u++) {
    if (!used[u]) {
      q.push(u);
      dist[u] = 0;
    }
  }
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int j = 0; j < (int)adj[u].size(); j++) {
      int v = match[adj[u][j]];
      if (v >= 0 && dist[v] < 0) {
        dist[v] = dist[u] + 1;
        q.push(v);
      }
    }
  }
}

bool dfs(int u) {
  visit[u] = true;
  for (int j = 0; j < (int)adj[u].size(); j++) {
    int v = match[adj[u][j]];
    if (v < 0 || (!visit[v] && dist[v] == dist[u] + 1 && dfs(v))) {
      match[adj[u][j]] = u;
      used[u] = true;
      return true;
    }
  }
  return false;
}

int hopcroft_karp(int n1, int n2) {
  fill(match, match + n2, -1);
  fill(used.begin(), used.end(), false);
  int res = 0;
  for (;;) {
    bfs(n1, n2);
    fill(visit.begin(), visit.end(), false);
    int f = 0;
    for (int u = 0; u < n1; u++) {
      if (!used[u] && dfs(u)) {
        f++;
      }
    }
    if (f == 0) {
      return res;
    }
    res += f;
  }
  return res;
}
```

Shortest Hamiltonian Cycle (TSP)

```
/* Given a weighted graph, determine a cycle of minimum total distance which visits each node exactly
      once and returns to the starting node. This is known as the traveling salesman problem (TSP).
      Since this implementation uses bitmasks with 32-bit integers, the maximum number of nodes must be
      less than 32. shortest_hamiltonian_cycle() applies to a global adjacency matrix adj[][], which
      must be populated with add_edge() before the function call.

Time Complexity:
- O(2^n * n^2) per call to shortest_hamiltonian_cycle(), where n is the numberof nodes.

Space Complexity:
```

```
- O(n^2) for storage of the graph, where n is the number of nodes.- O(2^n * n^2) auxiliary heap space
      for shortest_hamiltonian_cycle().*/

#include <algorithm>
using namespace std;

const int MAXN = 20, INF = 0x3f3f3f3f;
int adj[MAXN][MAXN], dp[1 << MAXN][MAXN], order[MAXN];

void add_edge(int u, int v, int w) {
  adj[u][v] = w;
  adj[v][u] = w;  // Remove this line if the graph is directed.
}

int shortest_hamiltonian_cycle(int nodes) {
  int max_mask = (1 << nodes) - 1;
  for (int i = 0; i <= max_mask; i++) {
    fill(dp[i], dp[i] + nodes, INF);
  }
  dp[1][0] = 0;
  for (int mask = 1; mask <= max_mask; mask += 2) {
    for (int i = 1; i < nodes; i++) {
      if ((mask & 1 << i) != 0) {
        for (int j = 0; j < nodes; j++) {
          if ((mask & 1 << j) != 0) {
            dp[mask][i] = min(dp[mask][i],
                                       dp[mask ^ (1 << i)][j] + adj[j][i]);
          }
        }
      }
    }
  }
  int res = INF + INF;
  for (int i = 1; i < nodes; i++) {
    res = min(res, dp[max_mask][i] + adj[i][0]);
  }
  int mask = max_mask, old = 0;
  for (int i = nodes - 1; i >= 1; i--) {
    int bj = -1;
    for (int j = 1; j < nodes; j++) {
      if ((mask & 1 << j) != 0 && (bj == -1 ||
              dp[mask][bj] + adj[bj][old] > dp[mask][j] + adj[j][old])) {
        bj = j;
      }
    }
    order[i] = bj;
    mask ^= 1 << bj;
    old = bj;
  }
  return res;
}
```

AVL Tree

```
/* Maintain a map, that is, a collection of key-value pairs such that each possible key appears at most
      once in the collection. This implementation requires an ordering on the set of possible keys
      defined by the < operator on the key type. An AVL tree is a binary search tree balanced by height,
       guaranteeing O(log n) worst-case running time in insertions and deletions by making sure that the
       heights of the left and right subtrees at every node differ by at most 1.  - avl_tree()
      constructs an empty map.- size() returns the size of the map.- empty() returns whether the map is
      empty.- insert(k, v) adds an entry with key k and value v to the map, returning trueif an new
      entry was added or false if the key already exists (in which case the map is unchanged and the old
       value associated with the key is preserved). - erase(k) removes the entry with key k from the map
      , returning true if theremoval was successful or false if the key to be removed was not found. -
      find(k) returns a pointer to a const value associated with key k, or NULL ifthe key was not found.
       - walk(f) calls the function f(k, v) on each entry of the map, in ascendingorder of keys.

Time Complexity:
- O(1) per call to the constructor, size(), and empty().- O(log n) per call to insert(), erase(), and
      find(), where n is the number ofentries currently in the map. - O(n) per call to walk().

Space Complexity:
- O(n) for storage of the map elements.- O(log n) auxiliary stack space for insert(), erase(), and walk
      ().- O(1) auxiliary for all other operations.*/

#include <algorithm>
#include <cstdlib>
using namespace std;

template<class K, class V>
class avl_tree {
  struct node_t {
    K key;
    V value;
    int height;
    node_t *left, *right;

    node_t(const K &k, const V &v)
        : key(k), value(v), height(1), left(NULL), right(NULL) {}
} *root;
int num_nodes;

static int height(node_t *n) {
  return (n != NULL) ? n->height : 0;
}

static void update_height(node_t *n) {
  if (n != NULL) {
    n->height = 1 + max(height(n->left), height(n->right));
  }
}

static void rotate_left(node_t *&n) {
  node_t *tmp = n;
  n = n->right;
  tmp->right = n->left;
  n->left = tmp;
  update_height(tmp);
  update_height(n);
}

static void rotate_right(node_t *&n) {
  node_t *tmp = n;
  n = n->left;
  tmp->left = n->right;
  n->right = tmp;
  update_height(tmp);
  update_height(n);
}

static int balance_factor(node_t *n) {
  return (n != NULL) ? (height(n->left) - height(n->right)) : 0;
}

static void rebalance(node_t *&n) {
  if (n == NULL) {
    return;
  }
  update_height(n);
  int bf = balance_factor(n);
  if (bf > 1 && balance_factor(n->left) >= 0) {
    rotate_right(n);
  } else if (bf > 1 && balance_factor(n->left) < 0) {
    rotate_left(n->left);
    rotate_right(n);
  } else if (bf < -1 && balance_factor(n->right) <= 0) {
    rotate_left(n);
  } else if (bf < -1 && balance_factor(n->right) > 0) {
    rotate_right(n->right);
    rotate_left(n);
  }
}

static bool insert(node_t *&n, const K &k, const V &v) {
  if (n == NULL) {
    n = new node_t(k, v);
    return true;
  }
  if ((k < n->key && insert(n->left, k, v)) ||
      (n->key < k && insert(n->right, k, v))) {
    rebalance(n);
    return true;
  }
  return false;
}

static bool erase(node_t *&n, const K &k) {
  if (n == NULL) {
    return false;
  }
  if (!(k < n->key || n->key < k)) {
    if (n->left != NULL && n->right != NULL) {
      node_t *tmp = n->right, *parent = NULL;
      while (tmp->left != NULL) {
        parent = tmp;
        tmp = tmp->left;
      }
      n->key = tmp->key;
      n->value = tmp->value;
      if (parent != NULL) {
        if (!erase(parent->left, parent->left->key)) {
          return false;
        }
      } else if (!erase(n->right, n->right->key)) {
        return false;
        }
      } else {
        node_t *tmp = (n->left != NULL) ? n->left : n->right;
        delete n;
        n = tmp;
      }
      rebalance(n);
      return true;
    }
    if ((k < n->key && erase(n->left, k)) ||
        (n->key < k && erase(n->right, k))) {
      rebalance(n);
      return true;
    }
    return false;
  }

  template<class KVFunction>
  static void walk(node_t *n, KVFunction f) {
    if (n != NULL) {
      walk(n->left, f);
      f(n->key, n->value);
      walk(n->right, f);
    }
  }

  static void clean_up(node_t *n) {
    if (n != NULL) {
      clean_up(n->left);
      clean_up(n->right);
      delete n;
    }
  }

 public:
  avl_tree() : root(NULL), num_nodes(0) {}

  ~avl_tree() {
    clean_up(root);
  }

  int size() const {
    return num_nodes;
  }

  bool empty() const {
    return root == NULL;
  }

  bool insert(const K &k, const V &v) {
    if (insert(root, k, v)) {
      num_nodes++;
      return true;
    }
    return false;
  }

  bool erase(const K &k) {
    if (erase(root, k)) {
      num_nodes--;
      return true;
    }
    return false;
  }

  const V* find(const K &k) const {
    node_t *n = root;
    while (n != NULL) {
      if (k < n->key) {
        n = n->left;
      } else if (n->key < k) {
        n = n->right;
      } else {
        return &(n->value);
      }
    }
    return NULL;
  }

  template<class KVFunction>
  void walk(KVFunction f) const {
    walk(root, f);
  }
};
```

K-d Tree (2D Range Query)

```
/* Maintain a set of two-dimensional points while supporting queries for all points that fall inside
     given rectangular regions. This implementation uses pair to represent points, requiring operators
      < and == to be defined on the numeric template type.  - kd_tree(lo, hi) constructs a set from two
      RandomAccessIterators to pairas a range [lo, hi) of points. - query(x1, y1, x2, y2, f) calls the
      function f(i, p) on each point in the setthat falls into the rectangular region consisting of rows
       from x1 to x2, inclusive, and columns from y1 to y2, inclusive. The first argument to f is the
      zero-based index of the point in the original range given to the constructor. The second argument
      is the point itself as an pair.

Time Complexity:
- O(n log n) per call to the constructor, where n is the number of points.- O(log(n) + m) on average per
       call to query(), where m is the number of pointsthat are reported by the query.

Space Complexity:
- O(n) for storage of the points.- O(log n) auxiliary stack space for query().*/

#include <algorithm>
#include <utility>
#include <vector>
using namespace std;

template<class T>
class kd_tree {
  typedef pair<T, T> point;

  static inline bool comp1(const point &a, const point &b) {
    return a.first < b.first;
  }

  static inline bool comp2(const point &a, const point &b) {
    return a.second < b.second;
  }

  vector<point> tree, minp, maxp;
  vector<int> l_index, h_index;

  void build(int lo, int hi, bool div_x) {
    if (lo >= hi) {
      return;
    }
    int mid = lo + (hi - lo)/2;
    nth_element(tree.begin() + lo, tree.begin() + mid, tree.begin() + hi,
                     div_x ? comp1 : comp2);
    l_index[mid] = lo;
    h_index[mid] = hi;
    minp[mid].first = maxp[mid].first = tree[lo].first;
    minp[mid].second = maxp[mid].second = tree[lo].second;
    for (int i = lo + 1; i < hi; i++) {
      minp[mid].first = min(minp[mid].first, tree[i].first);
      minp[mid].second = min(minp[mid].second, tree[i].second);
      maxp[mid].first = max(maxp[mid].first, tree[i].first);
      maxp[mid].second = max(maxp[mid].second, tree[i].second);
    }
    build(lo, mid, !div_x);
    build(mid + 1, hi, !div_x);
  }

  // Helper variables for query().
  T x1, y1, x2, y2;

  template<class ReportFunction>
  void query(int lo, int hi, ReportFunction f) {
    if (lo >= hi) {
      return;
    }
    int mid = lo + (hi - lo)/2;
    T ax = minp[mid].first, ay = minp[mid].second;
    T bx = maxp[mid].first, by = maxp[mid].second;
    if (x2 < ax || bx < x1 || y2 < ay || by < y1) {
      return;
    }
    if (!(ax < x1 || x2 < bx || ay < y1 || y2 < by)) {
      for (int i = l_index[mid]; i < h_index[mid]; i++) {
        f(tree[i]);
      }
      return;
    }
    query(lo, mid, f);
    query(mid + 1, hi, f);
    if (tree[mid].first < x1 || x2 < tree[mid].first ||
        tree[mid].second < y1 || y2 < tree[mid].second) {
      return;
    }
    f(tree[mid]);
  }

 public:
  template<class It>
  kd_tree(It lo, It hi) : tree(lo, hi) {
    int n = distance(lo, hi);
    l_index.resize(n);
    h_index.resize(n);
    minp.resize(n);
    maxp.resize(n);
    build(0, n, true);
  }

  template<class ReportFunction>
  void query(const T &x1, const T &y1, const T &x2, const T &y2,
             ReportFunction f) {
    this->x1 = x1;
    this->y1 = y1;
    this->x2 = x2;
    this->y2 = y2;
    query(0, tree.size(), f);
  }
};
```

K-d Tree (Nearest Neighbor)

```
/* Maintain a set of two-dimensional points while supporting queries for the closest point in the set to
      a given query point. This implementation uses pair to represent points, requiring operators <,
      ==, -, and long double casting to be defined on the numeric template type.  - kd_tree(lo, hi)
      constructs a set from two RandomAccessIterators to pairas a range [lo, hi) of points. - nearest(x,
      y, can_equal) returns a point in the set that is closest to (x, y)by Euclidean distance. This may
      be equal to (x, y) only if can_equal is true.

Time Complexity:
- O(n log n) per call to the constructor, where n is the number of points.- O(log n) on average per call
      to nearest().

Space Complexity:
- O(n) for storage of the points.- O(log n) auxiliary stack space for nearest().*/

#include <algorithm>
#include <limits>
#include <stdexcept>
#include <utility>
#include <vector>
using namespace std;

template<class T>
class kd_tree {
  typedef pair<T, T> point;

  static inline bool comp1(const point &a, const point &b) {
    return a.first < b.first;
  }

  static inline bool comp2(const point &a, const point &b) {
    return a.second < b.second;
  }

  vector<point> tree;
  vector<bool> div_x;

  void build(int lo, int hi) {
    if (lo >= hi) {
      return;
    }
    int mid = lo + (hi - lo)/2;
    T minx, maxx, miny, maxy;
    minx = maxx = tree[lo].first;
    miny = maxy = tree[lo].second;
    for (int i = lo + 1; i < hi; i++) {
      minx = min(minx, tree[i].first);
      miny = min(miny, tree[i].second);
      maxx = max(maxx, tree[i].first);
      maxy = max(maxy, tree[i].second);
    }
    div_x[mid] = !((maxx - minx) < (maxy - miny));
    nth_element(tree.begin() + lo, tree.begin() + mid, tree.begin() + hi,
                div_x[mid] ? comp1 : comp2);
    if (lo + 1 == hi) {
      return;
    }
    build(lo, mid);
    build(mid + 1, hi);
  }

  // Helper variables for nearest().
  long double min_dist;
  int id;

  void nearest(int lo, int hi, const T &x, const T &y, bool can_equal) {
    if (lo >= hi) {
      return;
    }
    int mid = lo + (hi - lo)/2;
    T dx = x - tree[mid].first, dy = y - tree[mid].second;
    long double d = dx*(long double)dx + dy*(long double)dy;
    if (d < min_dist && (can_equal || d != 0)) {
      min_dist = d;
      id = mid;
    }
    if (lo + 1 == hi) {
      return;
    }
    d = (long double)(div_x[mid] ? dx : dy);
    int l1 = lo, r1 = mid, l2 = mid + 1, r2 = hi;
    if (d > 0) {
      swap(l1, l2);
      swap(r1, r2);
    }
    nearest(l1, r1, x, y, can_equal);
    if (d*(long double)d < min_dist) {
      nearest(l2, r2, x, y, can_equal);
    }
  }

 public:
  template<class It>
  kd_tree(It lo, It hi) : tree(lo, hi) {
    int n = distance(lo, hi);
    if (n <= 1) {
      throw runtime_error("K-d tree must be have at least 2 points.");
    }
    div_x.resize(n);
    build(0, n);
  }

  point nearest(const T &x, const T &y, bool can_equal = true) {
    min_dist = numeric_limits<long double>::max();
    nearest(0, tree.size(), x, y, can_equal);
    return tree[id];
  }
};
```

Fenwick Tree (Compressed)

```
/* Maintain an array of numerical type, allowing for contiguous sub-arrays to be simultaneously
      incremented by arbitrary values (range update) and queries for the sum of contiguous sub-arrays (
      range query). This implementation uses map for coordinate compression, allowing for large indices
      to be accessed with efficient space complexity. That is, all array indices from 0 to MAXN,
      inclusive, are accessible.  - at(i) returns the value at index i.- add(i, x) adds x to the value
      at index i.- add(lo, hi, x) adds x to the values at all indices from lo to hi, inclusive.- set(i,
      x) assigns the value at index i to x.- sum(hi) returns the sum of all values at indices from 0 to
      hi, inclusive.- sum(lo, hi) returns the sum of all values at indices from lo to hi, inclusive.

Time Complexity:
- O(log^2 MAXN) per call to all member functions. If map is replaced withunordered_map, then the
      amortized running time will become O(log MAXN).

Space Complexity:
- O(n log MAXN) for storage of the array elements, where n is the number ofdistinct indices that have
      been accessed across all of the operations so far. - O(1) auxiliary for all operations.*/

#include <map>
using namespace std;

template<class T>
class fenwick_tree {
  static const int MAXN = 1000000001;
  map<int, T> tmul, tadd;

  void add_helper(int at, int mul, T add) {
    for (int i = at; i <= MAXN; i |= i + 1) {
      tmul[i] += mul;
      tadd[i] += add;
    }
  }

 public:
  void add(int lo, int hi, const T &x) {
    add_helper(lo, x, -x*(lo - 1));
    add_helper(hi, -x, x*hi);
  }

  void add(int i, const T &x) {
    return add(i, i, x);
  }

  void set(int i, const T &x) {
    add(i, x - at(i));
  }

  T sum(int hi) {
    T mul = 0, add = 0;
    for (int i = hi; i >= 0; i = (i & (i + 1)) - 1) {
      if (tmul.find(i) != tmul.end()) {
```

```cpp
        mul += tmul[i];
      }
      if (tadd.find(i) != tadd.end()) {
        add += tadd[i];
      }
    }
    return mul*hi + add;
  }

  T sum(int lo, int hi) {
    return sum(hi) - sum(lo - 1);
  }

  T at(int i) {
    return sum(i, i);
  }
};
```

Heavy Light Decomposition

```cpp
/* Maintain a tree with values associated with either its edges or nodes, while supporting both dynamic
      queries and dynamic updates of all values on a given path between two nodes in the tree. Heavy-
      light decomposition partitions the nodes of the tree into disjoint paths where all nodes have
      degree two, except the endpoints of a path which has degree one.  The query operation is defined
      by an associative join_values() function which satisfies join_values(x, join_values(y, z)) =
      join_values(join_values(x, y), z) for all values x, y, and z in the tree. The default code below
      assumes a numerical tree type, defining queries for the "min" of the target range. Another
      possible query operation is "sum", in which case the join_values() function should be defined to
      return "a + b".  The update operation is defined by the join_value_with_delta() and join_deltas()
      functions, which determines the change made to values. These must satisfy: - join_deltas(d1,
      join_deltas(d2, d3)) = join_deltas(join_deltas(d1, d2), d3).- join_value_with_delta(join_values(v,
       ...(m times)..., v), d, m)) should beequal to join_values(join_value_with_delta(v, d, 1), ...(m
      times)). - if a sequence d_1, ..., d_m of deltas is used to update a value v,
      thenjoin_value_with_delta(v, join_deltas(d_1, ..., d_m), 1) should be equivalent to m sequential
      calls to join_value_with_delta(v, d_i, 1) for i = 1..m. The default code below defines updates
      that "set" a path's edges or nodes to a new value. Another possible update operation is "increment
      ", in which case join_value_with_delta(v, d, len) should be defined to return "v + d*len" and
      join_deltas(d1, d2) should be defined to return "d1 + d2".  - heavy_light(n, adj[], v) constructs
      a new heavy light decomposition on a treewith n nodes defined by the adjacency list adj[], with
      all values initialized to v. The adjacency list must be a size n array of vectors consisting of
      only the integers from 0 to n - 1, inclusive. No duplicate edges should exist, and the graph must
      be connected. - query(u, v) returns the result of join_values() applied to all values on thepath
      from node u to node v. - update(u, v, d) modifies all values on the path from node u to node v
      byrespectively joining them with d using join_value_with_delta().

Time Complexity:
- O(n) per call to the constructor, where n is the number of nodes.- O(log n) per call to query() and
      update();

Space Complexity:
- O(n) for storage of the decomposition.- O(n) auxiliary stack space for the constructor.- O(1)
      auxiliary for query() and update().*/

#include <algorithm>
#include <stdexcept>
#include <vector>
using namespace std;

template<class T>
class heavy_light {
  // Set this to true to store values on edges, false to store values on nodes.
  static const bool VALUES_ON_EDGES = true;

  static T join_values(const T &a, const T &b) {
    return min(a, b);
  }

  static T join_value_with_delta(const T &v, const T &d, int len) {
    return d;
  }

  static T join_deltas(const T &d1, const T &d2) {
    return d2;  // For "set" updates, the more recent delta prevails.
  }

  int counter, paths;
  vector<vector<T> > value, delta;
  vector<vector<bool> > pending;
  vector<vector<int> > len;
  vector<int> size, parent, tin, tout, path, pathlen, pathpos, pathroot;
  vector<int> *adj;

  void dfs(int u, int p) {
    tin[u] = counter++;
    parent[u] = p;
    size[u] = 1;
    for (int j = 0; j < (int)adj[u].size(); j++) {
      int v = adj[u][j];
      if (v != p) {
        dfs(v, u);
        size[u] += size[v];
      }
    }
    tout[u] = counter++;
  }

  int new_path(int u) {
    pathroot[paths] = u;
    return paths++;
  }

  void build_paths(int u, int path) {
    this->path[u] = path;
    pathpos[u] = pathlen[path]++;
    for (int j = 0; j < (int)adj[u].size(); j++) {
      int v = adj[u][j];
      if (v != parent[u]) {
        build_paths(v, (2*size[v] >= size[u]) ? path : new_path(v));
      }
    }
  }

  inline T join_value_with_delta(int path, int i) {
    return pending[path][i]
        ? join_value_with_delta(value[path][i], delta[path][i], len[path][i])
        : value[path][i];
  }

  void push_delta(int path, int i) {
    int d = 0;
    while ((i >> d) > 0) {
      d++;
    }
    for (d -= 2; d >= 0; d--) {
      int l = (i >> d), r = (l ^ 1), n = l/2;
      if (pending[path][n]) {
        value[path][n] = join_value_with_delta(path, n);
        delta[path][l] =
            pending[path][l] ? join_deltas(delta[path][l], delta[path][n])
                             : delta[path][n];
        delta[path][r] =
            pending[path][r] ? join_deltas(delta[path][r], delta[path][n])
                             : delta[path][n];
        pending[path][l] = pending[path][r] = true;
        pending[path][n] = false;
      }
    }
  }

  bool query(int path, int u, int v, T *res) {
    push_delta(path, u += value[path].size()/2);
    push_delta(path, v += value[path].size()/2);
    bool found = false;
    for (; u <= v; u = (u + 1)/2, v = (v - 1)/2) {
      if ((u & 1) != 0) {
        T value = join_value_with_delta(path, u);
        *res = found ? join_values(*res, value) : value;
        found = true;
      }
      if ((v & 1) == 0) {
        T value = join_value_with_delta(path, v);
        *res = found ? join_values(*res, value) : value;
        found = true;
      }
    }
    return found;
  }

  void update(int path, int u, int v, const T &d) {
    push_delta(path, u += value[path].size()/2);
    push_delta(path, v += value[path].size()/2);
    int tu = -1, tv = -1;
    for (; u <= v; u = (u + 1)/2, v = (v - 1)/2) {
      if ((u & 1) != 0) {
        delta[path][u] = pending[path][u] ? join_deltas(delta[path][u], d) : d;
        pending[path][u] = true;
        if (tu == -1) {
          tu = u;
        }
      }
      if ((v & 1) == 0) {
        delta[path][v] = pending[path][v] ? join_deltas(delta[path][v], d) : d;
        pending[path][v] = true;
        if (tv == -1) {
          tv = v;
        }
      }
    }
    for (int i = tu; i > 1; i /= 2) {
      value[path][i/2] = join_values(join_value_with_delta(path, i),
                                     join_value_with_delta(path, i ^ 1));
    }
    for (int i = tv; i > 1; i /= 2) {
      value[path][i/2] = join_values(join_value_with_delta(path, i),
                                     join_value_with_delta(path, i ^ 1));
    }
  }

  inline bool is_ancestor(int parent, int child) {
    return (tin[parent] <= tin[child]) && (tout[child] <= tout[parent]);
  }

 public:
  heavy_light(int n, vector<int> adj[], const T &v = T())
      : counter(0), paths(0), size(n), parent(n), tin(n), tout(n), path(n),
        pathlen(n), pathpos(n), pathroot(n), adj(adj) {
    dfs(0, -1);
    build_paths(0, new_path(0));
    value.resize(paths);
    delta.resize(paths);
    pending.resize(paths);
    len.resize(paths);
    for (int i = 0; i < paths; i++) {
      int m = pathlen[i];
      value[i].assign(2*m, v);
      delta[i].resize(2*m);
      pending[i].assign(2*m, false);
      len[i].assign(2*m, 1);
      for (int j = 2*m - 1; j > 1; j -= 2) {
        value[i][j/2] = join_values(value[i][j], value[i][j ^ 1]);
        len[i][j/2] = len[i][j] + len[i][j ^ 1];
      }
    }
  }

  T query(int u, int v) {
    if (VALUES_ON_EDGES && u == v) {
      throw runtime_error("No edge between u and v to be queried.");
    }
    bool found = false;
    T res = T(), value;
    int root;
    while (!is_ancestor(root = pathroot[path[u]], v)) {
      if (query(path[u], 0, pathpos[u], &value)) {
        res = found ? join_values(res, value) : value;
        found = true;
      }
      u = parent[root];
    }
    while (!is_ancestor(root = pathroot[path[v]], u)) {
      if (query(path[v], 0, pathpos[v], &value)) {
        res = found ? join_values(res, value) : value;
        found = true;
      }
      v = parent[root];
    }
    if (query(path[u], min(pathpos[u], pathpos[v]) + (int)VALUES_ON_EDGES,
              max(pathpos[u], pathpos[v]), &value)) {
      res = found ? join_values(res, value) : value;
      found = true;
    }
    if (!found) {
      throw runtime_error("Unexpected error: No values found.");
    }
    return res;
  }

  void update(int u, int v, const T &d) {
    if (VALUES_ON_EDGES && u == v) {
      return;
    }
    int root;
    while (!is_ancestor(root = pathroot[path[u]], v)) {
      update(path[u], 0, pathpos[u], d);
      u = parent[root];
    }
    while (!is_ancestor(root = pathroot[path[v]], u)) {
      update(path[v], 0, pathpos[v], d);
      v = parent[root];
    }
    update(path[u], min(pathpos[u], pathpos[v]) + (int)VALUES_ON_EDGES,
           max(pathpos[u], pathpos[v]), d);
```

```
  }
};
```

Combinatorial Calculations

```
/* The following functions implement common operations in combinatorics. All input arguments must be non
      -negative. All return values and table entries are computed as 64-bit integers modulo an input
      argument m or p.  - factorial(n, m) returns n! mod m.- factorialp(n, p) returns n! mod p, where p
      is prime.- binomial_table(n, m) returns rows 0 to n of Pascal's triangle as a table tsuch that t[i
      ][j] is equal to (i choose j) mod m. - permute(n, k, m) returns (n permute k) mod m.- choose(n, k,
       p) returns (n choose k) mod p, where p is prime.- multichoose(n, k, p) returns (n multi-choose k)
       mod p, where p is prime.- catalan(n, p) returns the nth Catalan number mod p, where p is prime.-
      partitions(n, m) returns the number of partitions of n, mod m.- partitions(n, k, m) returns the
      number of partitions of n into k parts, mod m.- stirling1(n, k, m) returns the (n, k) unsigned
      Stirling number of the 1st kindmod m. - stirling2(n, k, m) returns the (n, k) Stirling number of
      the 2nd kind mod m.- eulerian1(n, k, m) returns the (n, k) Eulerian number of the 1st kind mod m,
      where n > k. - eulerian2(n, k, m) returns the (n, k) Eulerian number of the 2nd kind mod m,where n
       > k.

Time Complexity:
- O(n) for factorial(n, m).- O(p log n) for factorialp(n, p).- O(n^2) for binomial_table(n, m).- O(k)
      for permute(n, k, p).- O(min(k, n - k)) for choose(n, k, p).- O(k) for multichoose(n, k, p).- O(n)
       for catalan(n, p).- O(n^2) for partitions(n, m).- O(n*k) for partitions(n, k, m), stirling1(n, k,
       m), stirling2(n, k, m),eulerian1(n, k, m), and eulerian2(n, k, m).

Space Complexity:
- O(n^2) auxiliary heap space for binomial_table(n, m).- O(n*k) auxiliary heap space for partitions(n, k
      , m), stirling1(n, k, m),stirling2(n, k, m), eulerian1(n, k, m), and eulerian2(n, k, m). - O(1)
      auxiliary for all other operations.*/

#include <vector>
using namespace std;

typedef long long int64;
typedef vector<vector<int64> > table;

int64 factorial(int n, int m = 1000000007) {
  int64 res = 1;
  for (int i = 2; i <= n; i++) {
    res = (res*i) % m;
  }
  return res % m;
}

int64 factorialp(int64 n, int64 p = 1000000007) {
  int64 res = 1;
  while (n > 1) {
    if (n / p % 2 == 1) {
      res = res*(p - 1) % p;
    }
    int max = n % p;
    for (int i = 2; i <= max; i++) {
      res = (res*i) % p;
    }
    n /= p;
  }
  return res % p;
}

table binomial_table(int n, int64 m = 1000000007) {
  table t(n + 1);
  for (int i = 0; i <= n; i++) {
    for (int j = 0; j <= i; j++) {
      if (i < 2 || j == 0 || i == j) {
        t[i].push_back(1);
      } else {
        t[i].push_back((t[i - 1][j - 1] + t[i - 1][j]) % m);
      }
    }
  }
  return t;
}

int64 permute(int n, int k, int64 m = 1000000007) {
  if (n < k) {
    return 0;
  }
  int64 res = 1;
  for (int i = 0; i < k; i++) {
    res = res*(n - i) % m;
  }
  return res % m;
}

int64 mulmod(int64 x, int64 n, int64 m) {
  int64 a = 0, b = x % m;
  for (; n > 0; n >>= 1) {
    if (n & 1) {
      a = (a + b) % m;
    }
    b = (b << 1) % m;
  }
  return a % m;
}

int64 powmod(int64 x, int64 n, int64 m) {
  int64 a = 1, b = x;
  for (; n > 0; n >>= 1) {
    if (n & 1) {
      a = mulmod(a, b, m);
    }
    b = mulmod(b, b, m);
  }
  return a % m;
}

int64 choose(int n, int k, int64 p = 1000000007) {
  if (n < k) {
    return 0;
  }
  if (k > n - k) {
    k = n - k;
  }
  int64 num = 1, den = 1;
  for (int i = 0; i < k; i++) {
    num = num*(n - i) % p;
  }
  for (int i = 1; i <= k; i++) {
    den = den*i % p;
  }
  return num*powmod(den, p - 2, p) % p;
}

int64 multichoose(int n, int k, int64 p = 1000000007) {
  return choose(n + k - 1, k, p);
}

int64 catalan(int n, int64 p = 1000000007) {
  return choose(2*n, n, p)*powmod(n + 1, p - 2, p) % p;
}

int64 partitions(int n, int64 m = 1000000007) {
  vector<int64> t(n + 1, 0);
  t[0] = 1;
  for (int i = 1; i <= n; i++) {
    for (int j = i; j <= n; j++) {
      t[j] = (t[j] + t[j - i]) % m;
    }
  }
  return t[n] % m;
}

int64 partitions(int n, int k, int64 m = 1000000007) {
  table t(n + 1, vector<int64>(k + 1, 0));
  t[0][1] = 1;
  for (int i = 1; i <= n; i++) {
    for (int j = 1, h = k < i ? k : i; j <= h; j++) {
      t[i][j] = (t[i - 1][j - 1] + t[i - j][j]) % m;
    }
  }
  return t[n][k] % m;
}

int64 stirling1(int n, int k, int64 m = 1000000007) {
  table t(n + 1, vector<int64>(k + 1, 0));
  t[0][0] = 1;
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= k; j++) {
      t[i][j] = (i - 1)*t[i - 1][j] % m;
      t[i][j] = (t[i][j] + t[i - 1][j - 1]) % m;
    }
  }
  return t[n][k] % m;
}

int64 stirling2(int n, int k, int64 m = 1000000007) {
  table t(n + 1, vector<int64>(k + 1, 0));
  t[0][0] = 1;
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= k; j++) {
      t[i][j] = j*t[i - 1][j] % m;
      t[i][j] = (t[i][j] + t[i - 1][j - 1]) % m;
    }
  }
  return t[n][k] % m;
}

int64 eulerian1(int n, int k, int64 m = 1000000007) {
  if (k > n - 1 - k) {
    k = n - 1 - k;
  }
  table t(n + 1, vector<int64>(k + 1, 1));
  for (int j = 1; j <= k; j++) {
    t[0][j] = 0;
  }
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= k; j++) {
      t[i][j] = (i - j)*t[i - 1][j - 1] % m;
      t[i][j] = (t[i][j] + ((j + 1)*t[i - 1][j] % m)) % m;
    }
  }
  return t[n][k] % m;
}

int64 eulerian2(int n, int k, int64 m = 1000000007) {
  table t(n + 1, vector<int64>(k + 1, 1));
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= k; j++) {
      if (i == j) {
        t[i][j] = 0;
      } else {
        t[i][j] = (j + 1)*t[i - 1][j] % m;
        t[i][j] = ((2*i - 1 - j)*t[i - 1][j - 1] % m + t[i][j]) % m;
      }
    }
  }
  return t[n][k] % m;
}
```

Enumerating Partitions

```
/* A partition of a natural number n is a way to write n as a sum of positive integers where the order
      of the addends does not matter.  - next_partition(p) takes a reference to a vector p[] of positive
       integers as apartition of n for which the function will re-assign to become the next
      lexicographically greater partition. The function returns true if such a partition exists, or
      false if p[] already consists of the lexicographically greatest partition (i.e. the single integer
       n). - partition_by_rank(n, r) returns the partition of n that is lexicographicallyranked r if
      addends in each partition were sorted in non-increasing order, where r is a zero-based rank in the
       range [0, partitions(n)). - rank_by_partition(p) returns an integer representing the zero-based
      rank ofthe partition specified by vector p[], which must consist of positive integers sorted in
      non-increasing order. - generate_increasing_partitions(n, f) calls the function f(lo, hi) on
      strictlyincreasing partitions of n in lexicographically increasing order of partition, where lo
      and hi are two RandomAccessIterators to a range [lo, hi) of integers. Note that non-strictly
      increasing partitions like {1, 1, 1, 1} are skipped.

Time Complexity:
- O(n) per call to next_partition().- O(n^2) per call to partition_by_rank(n, r) and rank_by_partition(p
      ).- O(p(n)) per call to generate_increasing_partitions(n, f), where p(n) is thenumber of
      partitions of n.

Space Complexity:
- O(1) auxiliary for next_partition().- O(n^2) auxiliary heap space for partition_function(),
      partition_by_rank(), andrank_by_partition(). - O(n) auxiliary stack space for
      generate_increasing_partitions().*/

#include <vector>
using namespace std;

bool next_partition(vector<int> &p) {
  int n = p.size();
  if (n <= 1) {
    return false;
  }
  int s = p[n - 1] - 1, i = n - 2;
  p.pop_back();
  for (; i > 0 && p[i] == p[i - 1]; i--) {
    s += p[i];
    p.pop_back();
  }
  for (p[i]++; s > 0; s--) {
    p.push_back(1);
  }
  return true;
}

long long partition_function(int a, int b) {
  static vector<vector<long long> > p(
      1, vector<long long>(1, 1));
  if (a >= (int)p.size()) {
    int old = p.size();
    p.resize(a + 1);
    p[0].resize(a + 1);
    for (int i = 1; i <= a; i++) {
      p[i].resize(a + 1);
      for (int j = old; j <= i; j++) {
        p[i][j] = p[i - 1][j - 1] + p[i - j][j];
      }
    }
  }
  return p[a][b];
}

vector<int> partition_by_rank(int n, long long r) {
  vector<int> res;
  for (int i = n, j; i > 0; i -= j) {
    for (j = 1; ; j++) {
      long long count = partition_function(i, j);
      if (r < count) {
        break;
      }
      r -= count;
    }
    res.push_back(j);
  }
  return res;
}

long long rank_by_partition(const vector<int> &p) {
  long long res = 0;
  int sum = 0;
  for (int i = 0; i < (int)p.size(); i++) {
    sum += p[i];
  }
  for (int i = 0; i < (int)p.size(); i++) {
    for (int j = 0; j < p[i]; j++) {
      res += partition_function(sum, j);
    }
    sum -= p[i];
  }
  return res;
}

typedef void (*ReportFunction)(vector<int>::iterator,
                               vector<int>::iterator);

void generate_increasing_partitions(int left, int prev, int i,
                                    vector<int> &p, ReportFunction f) {
  if (left == 0) {
    f(p.begin(), p.begin() + i);
    return;
  }
  for (p[i] = prev + 1; p[i] <= left; p[i]++) {
    generate_increasing_partitions(left - p[i], p[i], i + 1, p, f);
  }
}

void generate_increasing_partitions(int n, ReportFunction f) {
  vector<int> p(n, 0);
  generate_increasing_partitions(n, 0, 0, p, f);
}
```

GCD, LCM, Mod Inverse, Chinese Remainder

```
/* Common number theory operations relating to modular arithmetic.  - gcd(a, b) and gcd2(a, b) both
      return the greatest common division of a and busing the Euclidean algorithm. - lcm(a, b) returns
      the lowest common multiple of a and b.- extended_euclid(a, b) and extended_euclid2(a, b) both
      return a pair (x, y) ofintegers such that gcd(a, b) = a*x + b*y. - mod(a, b) returns the value of
      a mod b under the true Euclidean definition ofmodulo, that is, the smallest nonnegative integer m
      satisfying a + b*n = m for some integer n. Note that this is identical to the remainder operator %
       in C++ for nonnegative operands a and b, but the result will differ when an operand is negative.
      - mod_inverse(a, m) and mod_inverse2(a, m) both return an integer x such thata*x = 1 (mod m),
      where the arguments must satisfy m > 0 and gcd(a, m) = 1. - generate_inverse(p) returns a vector
      of integers where for each index i inthe vector, i*v[i] = 1 (mod p), where the argument p is prime
      . - simple_restore(n, a, p) and garner_restore(n, a, p) both return the solution xfor the system
      of simultaneous congruences x = a[i] (mod p[i]) for all indices i in [0, n), where p[] consist of
      pairwise coprime integers. The solution x is guaranteed to be unique by the Chinese remainder
      theorem.

Time Complexity:
- O(log(a + b)) per call to gcd(a, b), gcd2(a, b), lcm(a, b),extended_euclid(a, b), extended_euclid2(a,
      b), mod_inverse(a, b), and mod_inverse2(a, b). - O(1) for mod(a, b).- O(p) for generate_inverse(p)
      .- Exponential for simple_restore(n, a, p).- O(n^2) for garner_restore(n, a, p).

Space Complexity:
- O(log(a + b)) auxiliary stack space for gcd2(a, b), extended_euclid2(a, b),and mod_inverse2(a, b). - O
      (p) auxiliary heap space for generate_inverse(p).- O(n) auxiliary heap space for garner_restore(n,
       a, p).- O(1) auxiliary space for all other operations.*/

#include <cstdlib>
#include <utility>
#include <vector>
using namespace std;

template<class Int>
Int gcd(Int a, Int b) {
  while (b != 0) {
    Int t = b;
    b = a % b;
    a = t;
  }
  return abs(a);
}

template<class Int>
Int gcd2(Int a, Int b) {
  return (b == 0) ? abs(a) : gcd(b, a % b);
}

template<class Int>
Int lcm(Int a, Int b) {
  return abs(a / gcd(a, b) * b);
}

template<class Int>
pair<Int, Int> extended_euclid(Int a, Int b) {
  Int x = 1, y = 0, x1 = 0, y1 = 1;
  while (b != 0) {
    Int q = a/b, prev_x1 = x1, prev_y1 = y1, prev_b = b;
    x1 = x - q*x1;
    y1 = y - q*y1;
    b = a - q*b;
    x = prev_x1;
    y = prev_y1;
    a = prev_b;
  }
  return (a > 0) ? make_pair(x, y) : make_pair(-x, -y);
}

template<class Int>
pair<Int, Int> extended_euclid2(Int a, Int b) {
  if (b == 0) {
    return (a > 0) ? make_pair(1, 0) : make_pair(-1, 0);
  }
  pair<Int, Int> r = extended_euclid2(b, a % b);
  return make_pair(r.second, r.first - a/b*r.second);
}

template<class Int>
Int mod(Int a, Int m) {
  Int r = a % m;
  return (r >= 0) ? r : (r + m);
}

template<class Int>
Int mod_inverse(Int a, Int m) {
  a = mod(a, m);
  return (a == 0) ? 0 : mod((1 - m*mod_inverse(m % a, a)) / a, m);
}

template<class Int>
Int mod_inverse2(Int a, Int m) {
  return mod(extended_euclid(a, m).first, m);
}

vector<int> generate_inverse(int p) {
  vector<int> res(p);
  res[1] = 1;
  for (int i = 2; i < p; i++) {
    res[i] = (p - (p / i)*res[p % i] % p) % p;
  }
  return res;
}

long long simple_restore(int n, int a[], int p[]) {
  long long res = 0, m = 1;
  for (int i = 0; i < n; i++) {
    while (res % p[i] != a[i]) {
      res += m;
    }
    m *= p[i];
  }
  return res;
}

long long garner_restore(int n, int a[], int p[]) {
  vector<int> x(a, a + n);
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
      x[i] = mod_inverse((long long)p[j], (long long)p[i])*(x[i] - x[j]);
    }
    x[i] = (x[i] % p[i] + p[i]) % p[i];
  }
  long long res = x[0], m = 1;
  for (int i = 1; i < n; i++) {
    m *= p[i - 1];
    res += x[i] * m;
  }
  return res;
}
```

Big Integers

```
/* Perform operations on arbitrary precision big integers internally represented as a vector of base
      -1000000000 digits in little-endian order. Typical arithmetic operations involving mixed numeric
      primitives and strings are supported using templates and operator overloading, as long as at least
       one operand is a bigint at any given level of evaluation.  - bigint(n) constructs a big integer
      from a long long (default = 0).- bigint(s) constructs a big integer from a C string or an string s
      .- operator = is defined to copy from another big integer or to assign from an64-bit integer
      primitive. - size() returns the number of digits in the base-10 representation.- operators >> and
      << are defined to support stream-based input and output.- v.to_string(), v.to_llong(), v.to_double
      (), and v.to_ldouble() return the biginteger v converted to an string, long long, double, and long
       double respectively. For the latter three data types, overflow behavior is based on that of
      inputting from istream. - v.abs() returns the absolute value of big integer v.- a.comp(b) returns
      -1, 0, or 1 depending on whether the big integers a and bcompare less, equal, or greater,
      respectively. - operators <, >, <=, >=, ==, !=, +, -, *, /, %, ++, --, +=, -=, *=, /=, and %=are
      defined analogous to those on integer primitives. Addition, subtraction, and comparisons are
      performed using the standard linear algorithms. Multiplication is performed using a combination of
       the grade school algorithm (for smaller inputs) and either the Karatsuba algorithm (if the
      USE_FFT_MULT flag is set to false) or the ö-SchnhageStrassen algorithm (if USE_FFT_MULT is set to
      true). Division and modulo are computed simultaneously using the grade school method. - a.div(b)
      returns a pair consisting of the quotient and remainder.- v.pow(n) returns v raised to the power
      of n.- v.sqrt() returns the integral part of the square root of big integer v.- v.nth_root(n)
      returns the integral part of the n-th root of big integer v.- rand(n) returns a random, positive
      big integer with n digits.

Time Complexity:
- O(n) per call to the constructors, size(), to_string(), to_llong(),to_double(), to_ldouble(), abs(),
      comp(), rand(), and all comparison and arithmetic operators except multiplication, division, and
      modulo, where n is total number of digits in the argument(s) and result for each operation. - O(n*
      log(n)*log(log(n))) or O(n^1.585) per call to multiplication operations,depending on whether
      USE_FFT_MULT is set to true or false. - O(n*m) per call to division and modulo operations, where n
       and m are thenumber of digits in the dividend and divisor, respectively. - O(M(m) log n) per call
       to pow(n), where m is the length of the big integer.

Space Complexity:
- O(n) for storage of the big integer.- O(n) auxiliary heap space for negation, addition, subtraction,
      multiplication,division, abs(), sqrt(), pow(), and nth_root(). - O(1) auxiliary space for all
      other operations.*/

#include <algorithm>
#include <cmath>
#include <complex>
#include <cstdlib>
#include <cstring>
#include <iomanip>
#include <istream>
#include <ostream>
#include <sstream>
#include <stdexcept>
#include <string>
#include <utility>
#include <vector>
using namespace std;

class bigint {
  static const int BASE = 1000000000, BASE_DIGITS = 9;
  static const bool USE_FFT_MULT = true;

  typedef vector<int> vint;
  typedef vector<long long> vll;
  typedef vector<complex<double> > vcd;

  vint digits;
  int sign;

  void normalize() {
    while (!digits.empty() && digits.back() == 0) {
      digits.pop_back();
    }
    if (digits.empty()) {
      sign = 1;
    }
  }

  void read(int n, const char *s) {
    sign = 1;
    digits.clear();
    int pos = 0;
    while (pos < n && (s[pos] == '-' || s[pos] == '+')) {
      if (s[pos] == '-') {
        sign = -sign;
      }
      pos++;
    }
    for (int i = n - 1; i >= pos; i -= BASE_DIGITS) {
      int x = 0;
      for (int j = max(pos, i - BASE_DIGITS + 1); j <= i; j++) {
        x = x*10 + s[j] - '0';
      }
      digits.push_back(x);
    }
    normalize();
  }

  static int comp(const vint &a, const vint &b, int asign, int bsign) {
    if (asign != bsign) {
      return asign < bsign ? -1 : 1;
    }
    if (a.size() != b.size()) {
      return a.size() < b.size() ? -asign : asign;
    }
    for (int i = (int)a.size() - 1; i >= 0; i--) {
      if (a[i] != b[i]) {
        return a[i] < b[i] ? -asign : asign;
      }
    }
    return 0;
  }

  static bigint add(const vint &a, const vint &b, int asign, int bsign) {
    if (asign != bsign) {
      return (asign == 1) ? sub(a, b, asign, 1) : sub(b, a, bsign, 1);
    }
    bigint res;
    res.digits = a;
    res.sign = asign;
    int carry = 0, size = (int)max(a.size(), b.size());
    for (int i = 0; i < size || carry; i++) {
      if (i == (int)res.digits.size()) {
        res.digits.push_back(0);
      }
      res.digits[i] += carry + (i < (int)b.size() ? b[i] : 0);
      carry = (res.digits[i] >= BASE) ? 1 : 0;
      if (carry) {
        res.digits[i] -= BASE;
      }
    }
    return res;
  }

  static bigint sub(const vint &a, const vint &b, int asign, int bsign) {
    if (asign == -1 || bsign == -1) {
      return add(a, b, asign, -bsign);
    }
    bigint res;
    if (comp(a, b, asign, bsign) < 0) {
      res = sub(b, a, bsign, asign);
      res.sign = -1;
      return res;
    }
    res.digits = a;
    res.sign = asign;
    for (int i = 0, borrow = 0; i < (int)a.size() || borrow; i++) {
      res.digits[i] -= borrow + (i < (int)b.size() ? b[i] : 0);
      borrow = res.digits[i] < 0;
      if (borrow) {
        res.digits[i] += BASE;
      }
    }
    res.normalize();
    return res;
  }

  static vint convert_base(const vint &digits, int l1, int l2) {
    vll p(max(l1, l2) + 1);
    p[0] = 1;
    for (int i = 1; i < (int)p.size(); i++) {
      p[i] = p[i - 1]*10;
    }
    vint res;
    long long curr = 0;
    for (int i = 0, curr_digits = 0; i < (int)digits.size(); i++) {
      curr += digits[i]*p[curr_digits];
      curr_digits += l1;
      while (curr_digits >= l2) {
        res.push_back((int)(curr % p[l2]));
        curr /= p[l2];
      curr_digits -= 12;
    }
  }
  res.push_back((int)curr);
  while (!res.empty() && res.back() == 0) {
    res.pop_back();
  }
  return res;
}

template<class It>
static vll karatsuba(It alo, It ahi, It blo, It bhi) {
  int n = distance(alo, ahi), k = n/2;
  vll res(n*2);
  if (n <= 32) {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        res[i + j] += alo[i]*blo[j];
      }
    }
    return res;
  }
  vll a1b1 = karatsuba(alo, alo + k, blo, blo + k);
  vll a2b2 = karatsuba(alo + k, ahi, blo + k, bhi);
  vll a2(alo + k, ahi), b2(blo + k, bhi);
  for (int i = 0; i < k; i++) {
    a2[i] += alo[i];
    b2[i] += blo[i];
  }
  vll r = karatsuba(a2.begin(), a2.end(), b2.begin(), b2.end());
  for (int i = 0; i < (int)a1b1.size(); i++) {
    r[i] -= a1b1[i];
    res[i] += a1b1[i];
  }
  for (int i = 0; i < (int)a2b2.size(); i++) {
    r[i] -= a2b2[i];
    res[i + n] += a2b2[i];
  }
  for (int i = 0; i < (int)r.size(); i++) {
    res[i + k] += r[i];
  }
  return res;
}

template<class It>
static vcd fft(It lo, It hi, bool invert = false) {
  int n = distance(lo, hi), k = 0, high1 = -1;
  while ((1 << k) < n) {
    k++;
  }
  vector<int> rev(n, 0);
  for (int i = 1; i < n; i++) {
    if (!(i & (i - 1))) {
      high1++;
    }
    rev[i] = rev[i ^ (1 << high1)];
    rev[i] |= (1 << (k - high1 - 1));
  }
  vcd roots(n), res(n);
  for (int i = 0; i < n; i++) {
    double alpha = 2*3.14159265358979323846*i/n;
    roots[i] = complex<double>(cos(alpha), sin(alpha));
    res[i] = *(lo + rev[i]);
  }
  for (int len = 1; len < n; len <<= 1) {
    vcd tmp(n);
    int rstep = roots.size()/(len << 1);
    for (int pdest = 0; pdest < n; pdest += len) {
      int p = pdest;
      for (int i = 0; i < len; i++) {
        complex<double> c = roots[i*rstep]*res[p + len];
        tmp[pdest] = res[p] + c;
        tmp[pdest + len] = res[p] - c;
        pdest++;
        p++;
      }
    }
    res.swap(tmp);
  }
  if (invert) {
    for (int i = 0; i < (int)res.size(); i++) {
      res[i] /= n;
    }
    reverse(res.begin() + 1, res.end());
  }
  return res;
```

```
  }
 public:
  bigint() : sign(1) {}
  bigint(int v) { *this = (long long)v; }
  bigint(long long v) { *this = v; }
  bigint(const char *s) { read(strlen(s), s); }
  bigint(const string &s) { read(s.size(), s.c_str()); }

  void operator=(const bigint &v) {
    sign = v.sign;
    digits = v.digits;
  }

  void operator=(long long v) {
    sign = 1;
    if (v < 0) {
      sign = -1;
      v = -v;
    }
    digits.clear();
    for (; v > 0; v /= BASE) {
      digits.push_back(v % BASE);
    }
  }

  int size() const {
    if (digits.empty()) {
      return 1;
    }
    ostringstream oss;
    oss << digits.back();
    return oss.str().length() + BASE_DIGITS*(digits.size() - 1);
  }

  friend istream& operator>>(istream &in, bigint &v) {
    string s;
    in >> s;
    v.read(s.size(), s.c_str());
    return in;
  }

  friend ostream& operator<<(ostream &out, const bigint &v) {
    if (v.sign == -1) {
      out << '-';
    }
    out << (v.digits.empty() ? 0 : v.digits.back());
    for (int i = (int)v.digits.size() - 2; i >= 0; i--) {
      out << setw(BASE_DIGITS) << setfill('0') << v.digits[i];
    }
    return out;
  }

  string to_string() const {
    ostringstream oss;
    if (sign == -1) {
      oss << '-';
    }
    oss << (digits.empty() ? 0 : digits.back());
    for (int i = (int)digits.size() - 2; i >= 0; i--) {
      oss << setw(BASE_DIGITS) << setfill('0') << digits[i];
    }
    return oss.str();
  }

  long long to_llong() const {
    long long res = 0;
    for (int i = (int)digits.size() - 1; i >= 0; i--) {
      res = res*BASE + digits[i];
    }
    return res*sign;
  }

  double to_double() const {
    stringstream ss(to_string());
    double res;
    ss >> res;
    return res;
  }

  long double to_ldouble() const {
    stringstream ss(to_string());
    long double res;
    ss >> res;
    return res;
  }

  int comp(const bigint &v) const {
    return comp(digits, v.digits, sign, v.sign);
  }
```

```cpp
bool operator<(const bigint &v) const { return comp(v) < 0; }
bool operator>(const bigint &v) const { return comp(v) > 0; }
bool operator<=(const bigint &v) const { return comp(v) <= 0; }
bool operator>=(const bigint &v) const { return comp(v) >= 0; }
bool operator==(const bigint &v) const { return comp(v) == 0; }
bool operator!=(const bigint &v) const { return comp(v) != 0; }

template<class T>
friend bool operator<(const T &a, const bigint &b) { return bigint(a) < b; }

template<class T>
friend bool operator>(const T &a, const bigint &b) { return bigint(a) > b; }

template<class T>
friend bool operator<=(const T &a, const bigint &b) { return bigint(a) <= b; }

template<class T>
friend bool operator>=(const T &a, const bigint &b) { return bigint(a) >= b; }

template<class T>
friend bool operator==(const T &a, const bigint &b) { return bigint(a) == b; }

template<class T>
friend bool operator!=(const T &a, const bigint &b) { return bigint(a) != b; }

bigint abs() const {
  bigint res(*this);
  res.sign = 1;
  return res;
}

bigint operator-() const {
  bigint res(*this);
  res.sign = -sign;
  return res;
}

bigint operator+(const bigint &v) const {
  return add(digits, v.digits, sign, v.sign);
}

bigint operator-(const bigint &v) const {
  return sub(digits, v.digits, sign, v.sign);
}

void operator*=(int v) {
  if (v < 0) {
    sign = -sign;
    v = -v;
  }
  for (int i = 0, carry = 0; i < (int)digits.size() || carry; i++) {
    if (i == (int)digits.size()) {
      digits.push_back(0);
    }
    long long curr = digits[i]*(long long)v + carry;
    carry = (int)(curr/BASE);
    digits[i] = (int)(curr % BASE);
  }
  normalize();
}

bigint operator*(int v) const {
  bigint res(*this);
  res *= v;
  return res;
}

bigint operator*(const bigint &v) const {
  static const int TEMP_BASE = 10000, TEMP_BASE_DIGITS = 4;
  vint a = convert_base(digits, BASE_DIGITS, TEMP_BASE_DIGITS);
  vint b = convert_base(v.digits, BASE_DIGITS, TEMP_BASE_DIGITS);
  int n = 1 << (33 - __builtin_clz(max(a.size(), b.size()) - 1));
  a.resize(n, 0);
  b.resize(n, 0);
  vll c;
  if (USE_FFT_MULT) {
    vcd at = fft(a.begin(), a.end()), bt = fft(b.begin(), b.end());
    for (int i = 0; i < n; i++) {
      at[i] *= bt[i];
    }
    at = fft(at.begin(), at.end(), true);
    c.resize(n);
    for (int i = 0; i < n; i++) {
      c[i] = at[i].real() + 0.5;
    }
  } else {
    c = karatsuba(a.begin(), a.end(), b.begin(), b.end());
  }
  bigint res;
  res.sign = sign*v.sign;
  for (int i = 0, carry = 0; i < (int)c.size(); i++) {
    long long d = c[i] + carry;
    res.digits.push_back(d % TEMP_BASE);
    carry = d/TEMP_BASE;
  }
  res.digits = convert_base(res.digits, TEMP_BASE_DIGITS, BASE_DIGITS);
  res.normalize();
  return res;
}

bigint& operator/=(int v) {
  if (v == 0) {
    throw runtime_error("Division by zero in bigint.");
  }
  if (v < 0) {
    sign = -sign;
    v = -v;
  }
  for (int i = (int)digits.size() - 1, rem = 0; i >= 0; i--) {
    long long curr = digits[i] + rem*(long long)BASE;
    digits[i] = (int)(curr/v);
    rem = (int)(curr % v);
  }
  normalize();
  return *this;
}

bigint operator/(int v) const {
  bigint res(*this);
  res /= v;
  return res;
}

int operator%(int v) const {
  if (v == 0) {
    throw runtime_error("Division by zero in bigint.");
  }
  if (v < 0) {
    v = -v;
  }
  int m = 0;
  for (int i = (int)digits.size() - 1; i >= 0; i--) {
    m = (digits[i] + m*(long long)BASE) % v;
  }
  return m*sign;
}

pair<bigint, bigint> div(const bigint &v) const {
  if (v == 0) {
    throw runtime_error("Division by zero in bigint.");
  }
  if (comp(digits, v.digits, 1, 1) < 0) {
    return make_pair(0, *this);
  }
  int norm = BASE/(v.digits.back() + 1);
  bigint an = abs()*norm, bn = v.abs()*norm, q, r;
  q.digits.resize(an.digits.size());
  for (int i = (int)an.digits.size() - 1; i >= 0; i--) {
    r *= BASE;
    r += an.digits[i];
    int s1 = (r.digits.size() <= bn.digits.size())
                ? 0 : r.digits[bn.digits.size()];
    int s2 = (r.digits.size() <= bn.digits.size() - 1)
                ? 0 : r.digits[bn.digits.size() - 1];
    int d = ((long long)s1*BASE + s2)/bn.digits.back();
    for (r -= bn*d; r < 0; r += bn) {
      d--;
    }
    q.digits[i] = d;
  }
  q.sign = sign*v.sign;
  r.sign = sign;
  q.normalize();
  r.normalize();
  return make_pair(q, r/norm);
}

bigint operator/(const bigint &v) const { return div(v).first; }
bigint operator%(const bigint &v) const { return div(v).second; }

bigint operator++(int) { bigint t(*this); operator++(); return t; }
bigint operator--(int) { bigint t(*this); operator--(); return t; }
bigint& operator++() { *this = *this + bigint(1); return *this; }
bigint& operator--() { *this = *this - bigint(1); return *this; }
bigint& operator+=(const bigint &v) { *this = *this + v; return *this; }
bigint& operator-=(const bigint &v) { *this = *this - v; return *this; }
bigint& operator*=(const bigint &v) { *this = *this * v; return *this; }
bigint& operator/=(const bigint &v) { *this = *this / v; return *this; }
```

```
  bigint& operator%=(const bigint &v) { *this = *this % v; return *this; }

  template<class T>
  friend bigint operator+(const T &a, const bigint &b) { return bigint(a) + b; }

  template<class T>
  friend bigint operator-(const T &a, const bigint &b) { return bigint(a) - b; }

  bigint pow(int n) const {
    if (n == 0) {
      return bigint(1);
    }
    if (*this == 0 || n < 0) {
      return bigint(0);
    }
    bigint x(*this), res(1);
    for (; n != 0; n >>= 1) {
      if (n & 1) {
        res *= x;
      }
      x *= x;
    }
    return res;
  }

  bigint sqrt() const {
    if (sign == -1) {
      throw runtime_error("Cannot take square root of a negative number.");
    }
    bigint v(*this);
    while (v.digits.empty() || v.digits.size() % 2 == 1) {
      v.digits.push_back(0);
    }
    int n = v.digits.size();
    int ldig = (int)::sqrt((double)v.digits[n - 1]*BASE + v.digits[n - 2]);
    int norm = BASE/(ldig + 1);
    v *= norm;
    v *= norm;
    while (v.digits.empty() || v.digits.size() % 2 == 1) {
      v.digits.push_back(0);
    }
    bigint r((long long)v.digits[n - 1]*BASE + v.digits[n - 2]);
    int q = ldig = (int)::sqrt((double)v.digits[n - 1]*BASE + v.digits[n - 2]);
    bigint res;
    for (int j = n/2 - 1; j >= 0; j--) {
      for (;; q--) {
        bigint r1 = (r - (res*2*BASE + q)*q)*BASE*BASE +
            (j > 0 ? (long long)v.digits[2*j - 1]*BASE + v.digits[2*j - 2] : 0);
        if (r1 >= 0) {
          r = r1;
          break;
        }
      }
      res = res*BASE + q;
      if (j > 0) {
        int sz1 = res.digits.size(), sz2 = r.digits.size();
        int d1 = (sz1 + 2 < sz2) ? r.digits[sz1 + 2] : 0;
        int d2 = (sz1 + 1 < sz2) ? r.digits[sz1 + 1] : 0;
        int d3 = (sz1 < sz2) ? r.digits[sz1] : 0;
        q = ((long long)d1*BASE*BASE + (long long)d2*BASE + d3)/(ldig*2);
      }
    }
    res.normalize();
    return res/norm;
  }

  bigint nth_root(int n) const {
    if (sign == -1 && n % 2 == 0) {
      throw runtime_error("Cannot take even root of a negative number.");
    }
    if (*this == 0 || n < 0) {
      return bigint(0);
    }
    if (n >= size()) {
      int p = 1;
      while (comp(bigint(p).pow(n)) > 0) {
        p++;
      }
      return comp(bigint(p).pow(n)) < 0 ? p - 1 : p;
    }
    bigint lo(bigint(10).pow((int)ceil((double)size()/n) - 1)), hi(lo*10), mid;
    while (lo < hi) {
      mid = (lo + hi)/2;
      int cmp = comp(digits, mid.pow(n).digits, 1, 1);
      if (lo < mid && cmp > 0) {
        lo = mid;
      } else if (mid < hi && cmp < 0) {
        hi = mid;
      } else {
        return (sign == -1) ? -mid : mid;
      }
    }
    return (sign == -1) ? -(mid + 1) : (mid + 1);
  }

  static bigint rand(int n) {
    if (n == 0) {
      return bigint(0);
    }
    string s(1, '1' + (::rand() % 9));
    for (int i = 1; i < n; i++) {
      s += '0' + (::rand() % 10);
    }
    return bigint(s);
  }

  friend int comp(const bigint &a, const bigint &b) { return a.comp(b); }
  friend bigint abs(const bigint &v) { return v.abs(); }
  friend bigint pow(const bigint &v, int n) { return v.pow(n); }
  friend bigint sqrt(const bigint &v) { return v.sqrt(); }
  friend bigint nth_root(const bigint &v, int n) { return v.nth_root(n); }
};
```

Simplex Algorithm

```
/* Solves a linear programming problem using Dantzig's simplex algorithm. The canonical form of a linear
      programming problem is to maximize (or minimize) the dot product c*x, subject to a*x <= b and x
     >= 0, where x is a vector of unknowns to be solved, c is a vector of coefficients, a is a matrix
     of linear equation coefficients, and b is a vector of boundary coefficients.  - simplex_solve(a, b
     , c, &x) solves the linear programming problem for an m by nmatrix a of real values, a length m
     vector b, a length n vector c, returning 0 if a solution was found or -1 if there is no solution.
     If a solution is found, then the vector pointed to by x is populated with the solution vector of
     length n.

Time Complexity:
- Polynomial (average) on the number of equations and unknowns, but exponentialin the worst case.

Space Complexity:
- O(m*n) auxiliary heap space.*/

#include <cmath>
#include <limits>
#include <vector>
using namespace std;

template<class Matrix>
int simplex_solve(const Matrix &a, const vector<double> &b,
                  const vector<double> &c, vector<double> *x,
                  const bool MAXIMIZE = true, const double EPS = 1e-10) {
  int m = a.size(), n = c.size();
  Matrix t(m + 2, vector<double>(n + 2));
  t[1][1] = 0;
  for (int j = 1; j <= n; j++) {
    t[1][j + 1] = MAXIMIZE ? c[j - 1] : -c[j - 1];
  }
  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
      t[i + 1][j + 1] = -a[i - 1][j - 1];
    }
    t[i + 1][1] = b[i - 1];
  }
  for (int j = 1; j <= n; j++) {
    t[0][j + 1] = j;
  }
  for (int i = n + 1; i <= m + n; i++) {
    t[i - n + 1][0] = i;
  }
  double p1 = 0, p2 = 0;
  bool done = true;
  do {
    double mn = numeric_limits<double>::max(), xmax = 0, v;
    for (int j = 2; j <= n + 1; j++) {
      if (t[1][j] > 0 && t[1][j] > xmax) {
        p2 = j;
        xmax = t[1][j];
      }
    }
    for (int i = 2; i <= m + 1; i++) {
      v = fabs(t[i][1] / t[i][p2]);
      if (t[i][p2] < 0 && mn > v) {
        mn = v;
        p1 = i;
      }
    }
    swap(t[p1][0], t[0][p2]);
    for (int i = 1; i <= m + 1; i++) {
      if (i != p1) {
        for (int j = 1; j <= n + 1; j++) {
          if (j != p2) {
            t[i][j] -= t[p1][j]*t[i][p2] / t[p1][p2];
          }
        }
      }
    }
    t[p1][p2] = 1.0 / t[p1][p2];
    for (int j = 1; j <= n + 1; j++) {
      if (j != p2) {
        t[p1][j] *= fabs(t[p1][p2]);
      }
    }
    for (int i = 1; i <= m + 1; i++) {
      if (i != p1) {
        t[i][p2] *= t[p1][p2];
      }
    }
    for (int i = 2; i <= m + 1; i++) {
      if (t[i][1] < 0) {
        return -1;
      }
    }
    done = true;
    for (int j = 2; j <= n + 1; j++) {
      if (t[1][j] > 0) {
        done = false;
      }
    }
  } while (!done);
  x->clear();
  for (int j = 1; j <= n; j++) {
    for (int i = 2; i <= m + 1; i++) {
      if (fabs(t[i][0] - j) < EPS) {
        x->push_back(t[i][1]);
      }
    }
  }
  return 0;
}
```

Convex Hull (Monotone Chain)

```
/* Given a list of points in two dimensions, determine its convex hull using the monotone chain
      algorithm. The convex hull is the smallest convex polygon (a polygon such that every line crossing
       through it will only do so once) that contains all of its points.  - convex_hull(lo, hi) returns
      the convex hull as a vector of polygon vertices inclockwise order, given a range [lo, hi) of
      points where lo and hi must be RandomAccessIterators. The input range will be sorted
      lexicographically (by x, then by y for equal x) after the function call. Note that to produce the
      hull points in counter-clockwise order, replace every GE() comparison with LE(). To have the first
       point on the hull repeated as the last in the resulting vector, the final res.resize(k - 1) may
      be changed to res.resize(k).

Time Complexity:
- O(n log n) per call to convex_hull(lo, hi), where n is the distance between loand hi.

Space Complexity:
- O(n) auxiliary for storage of the convex hull.*/

#include <algorithm>
#include <cmath>
#include <utility>
#include <vector>
using namespace std;

const double EPS = 1e-9;

#define GE(a, b) ((a) >= (b) - EPS)

typedef pair<double, double> point;
#define x first
#define y second

double cross(const point &a, const point &b, const point &o = point(0, 0)) {
  return (a.x - o.x)*(b.y - o.y) - (a.y - o.y)*(b.x - o.x);
}

template<class It>
vector<point> convex_hull(It lo, It hi) {
  int k = 0;
  if (hi - lo <= 1) {
    return vector<point>(lo, hi);
  }
  vector<point> res(2*(int)(hi - lo));
  sort(lo, hi);
  for (It it = lo; it != hi; ++it) {
    while (k >= 2 && GE(cross(res[k - 1], *it, res[k - 2]), 0)) {
      k--;
    }
    res[k++] = *it;
  }
  int t = k + 1;
  for (It it = hi - 2; it != lo - 1; --it) {
    while (k >= t && GE(cross(res[k - 1], *it, res[k - 2]), 0)) {
      k--;
    }
    res[k++] = *it;
  }
  res.resize(k - 1);
  return res;
}
```

Minimum Enclosing Circle

```
/* Given a list of points in two dimensions, find the circle with smallest area which contains all the
      given points using a randomized algorithm.  - minimum_enclosing_circle(lo, hi) returns the minimum
       enclosing circle given arange [lo, hi) of points, where lo and hi must be RandomAccessIterators.
      The input range will be shuffled after the function call, though this is only to avoid the worst-
      case running time and is not necessary for correctness.

Time Complexity:
- O(n) on average per call to minimum_enclosing_circle(lo, hi), where n is thedistance between lo and hi
      .

Space Complexity:
- O(1) auxiliary.*/

#include <algorithm>
#include <cmath>
#include <stdexcept>
#include <utility>
using namespace std;

const double EPS = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= EPS)
#define LE(a, b) ((a) <= (b) + EPS)

typedef pair<double, double> point;
#define x first
#define y second

double sqnorm(const point &a) { return a.x*a.x + a.y*a.y; }
double norm(const point &a) { return sqrt(sqnorm(a)); }

struct circle {
  double h, k, r;

  circle() : h(0), k(0), r(0) {}
  circle(double h, double k, double r) : h(h), k(k), r(fabs(r)) {}

  // Circle with the line segment ab as a diameter.
  circle(const point &a, const point &b) {
    h = (a.x + b.x)/2.0;
    k = (a.y + b.y)/2.0;
    r = norm(point(a.x - h, a.y - k));
  }

  // Circumcircle of three points.
  circle(const point &a, const point &b, const point &c) {
    double an = sqnorm(point(b.x - c.x, b.y - c.y));
    double bn = sqnorm(point(a.x - c.x, a.y - c.y));
    double cn = sqnorm(point(a.x - b.x, a.y - b.y));
    double wa = an*(bn + cn - an);
    double wb = bn*(an + cn - bn);
    double wc = cn*(an + bn - cn);
    double w = wa + wb + wc;
    if (EQ(w, 0)) {
      throw runtime_error("No circumcircle from collinear points.");
    }
    h = (wa*a.x + wb*b.x + wc*c.x)/w;
    k = (wa*a.y + wb*b.y + wc*c.y)/w;
    r = norm(point(a.x - h, a.y - k));
  }

  bool contains(const point &p) const {
    return LE(sqnorm(point(p.x - h, p.y - k)), r*r);
  }
};

template<class It>
```

```
circle minimum_enclosing_circle(It lo, It hi) {
  if (lo == hi) {
    return circle(0, 0, 0);
  }
  if (lo + 1 == hi) {
    return circle(lo->x, lo->y, 0);
  }
  random_shuffle(lo, hi);
  circle res(*lo, *(lo + 1));
  for (It i = lo + 2; i != hi; ++i) {
    if (res.contains(*i)) {
      continue;
    }
    res = circle(*lo, *i);
    for (It j = lo + 1; j != i; ++j) {
      if (res.contains(*j)) {
        continue;
      }
      res = circle(*i, *j);
      for (It k = lo; k != j; ++k) {
        if (!res.contains(*k)) {
          res = circle(*i, *j, *k);
        }
      }
    }
  }
  return res;
}
```

Diameter of Points

```
/* Determines the diametral pair of a range of points. The diamter of a set of points is the largest
      distance between any two points in the set. A diametral pair is a pair of points in the set whose
      distance is equal to the set's diameter. The following program uses rotating calipers method to
      find a solution.

Time Complexity: O(n log n) on the number of points in the set.
*/

#include <algorithm> /* sort() */
#include <cmath>     /* fabs(), sqrt() */
#include <utility>   /* pair */
#include <vector>
using namespace std;

typedef pair<double, double> point;
#define x first
#define y second

double sqdist(const point & a, const point & b) {
  double dx = a.x - b.x, dy = a.y - b.y;
  return sqrt(dx * dx + dy * dy);
}

double cross(const point & o, const point & a, const point & b) {
  return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}

bool cw(const point & o, const point & a, const point & b) {
  return cross(o, a, b) < 0;
}

double area(const point & o, const point & a, const point & b) {
  return fabs(cross(o, a, b));
}

template<class It> vector<point> convex_hull(It lo, It hi) {
  int k = 0;
  if (hi - lo <= 1) return vector<point>(lo, hi);
  vector<point> res(2 * (int)(hi - lo));
  sort(lo, hi); //compare by x, then by y if x-values are equal
  for (It it = lo; it != hi; ++it) {
    while (k >= 2 && !cw(res[k - 2], res[k - 1], *it)) k--;
    res[k++] = *it;
  }
  int t = k + 1;
  for (It it = hi - 2; it != lo - 1; --it) {
    while (k >= t && !cw(res[k - 2], res[k - 1], *it)) k--;
    res[k++] = *it;
  }
  res.resize(k - 1);
  return res;
}

template<class It> pair<point, point> diametral_pair(It lo, It hi) {
  vector<point> h = convex_hull(lo, hi);
  int m = h.size();
  if (m == 1) return make_pair(h[0], h[0]);
  if (m == 2) return make_pair(h[0], h[1]);
  int k = 1;
  while (area(h[m - 1], h[0], h[(k + 1) % m]) > area(h[m - 1], h[0], h[k]))
    k++;
  double maxdist = 0, d;
  pair<point, point> res;
  for (int i = 0, j = k; i <= k && j < m; i++) {
    d = sqdist(h[i], h[j]);
    if (d > maxdist) {
      maxdist = d;
      res = make_pair(h[i], h[j]);
    }
    while (j < m && area(h[i], h[(i + 1) % m], h[(j + 1) % m]) >
                    area(h[i], h[(i + 1) % m], h[j])) {
      d = sqdist(h[i], h[(j + 1) % m]);
      if (d > maxdist) {
        maxdist = d;
        res = make_pair(h[i], h[(j + 1) % m]);
      }
      j++;
    }
  }
  return res;
}
```

Closest Pair

```
/* Given a range containing distinct points on the Cartesian plane, determine two points which have the
      closest possible distance. A divide and conquer algorithm is used. Note that the ordering of
      points in the input range may be changed by the function.

Time Complexity: O(n log^2 n) where n is the number of points.
*/

#include <algorithm> /* min, sort */
#include <cfloat>    /* DBL_MAX */
#include <cmath>     /* fabs */
#include <utility>   /* pair */
using namespace std;

typedef pair<double, double> point;
#define x first
#define y second

double sqdist(const point & a, const point & b) {
  double dx = a.x - b.x, dy = a.y - b.y;
  return dx * dx + dy * dy;
}

bool cmp_x(const point & a, const point & b) { return a.x < b.x; }
bool cmp_y(const point & a, const point & b) { return a.y < b.y; }

template<class It>
double rec(It lo, It hi, pair<point, point> & res, double mindist) {
  if (lo == hi) return DBL_MAX;
  It mid = lo + (hi - lo) / 2;
  double midx = mid->x;
  double d1 = rec(lo, mid, res, mindist);
  mindist = min(mindist, d1);
  double d2 = rec(mid + 1, hi, res, mindist);
  mindist = min(mindist, d2);
  sort(lo, hi, cmp_y);
  int size = 0;
  It t[hi - lo];
  for (It it = lo; it != hi; ++it)
    if (fabs(it->x - midx) < mindist)
      t[size++] = it;
  for (int i = 0; i < size; i++) {
    for (int j = i + 1; j < size; j++) {
      point a = *t[i], b = *t[j];
      if (b.y - a.y >= mindist) break;
      double dist = sqdist(a, b);
      if (mindist > dist) {
        mindist = dist;
        res = make_pair(a, b);
      }
    }
  }
  return mindist;
}

template<class It> pair<point, point> closest_pair(It lo, It hi) {
  pair<point, point> res;
  sort(lo, hi, cmp_x);
  rec(lo, hi, res, DBL_MAX);
  return res;
}
```

Polygon Union and Intersection

```cpp
/* Given two ranges of points respectively denoting the vertices of two polygons, determine the
      intersection area of those polygons. Using this, we can easily calculate their union with the
      forumla: union_area(A, B) = area(A) + area(B) - intersection_area(A, B)

Time Complexity: O(n^2 log n), where n is the total number of vertices.
*/

#include <algorithm> /* sort() */
#include <cmath>     /* fabs(), sqrt() */
#include <set>
#include <utility>   /* pair */
#include <vector>
using namespace std;

const double eps = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
#define LT(a, b) ((a) < (b) - eps)        /* less than */
#define LE(a, b) ((a) <= (b) + eps)       /* less than or equal to */

typedef pair<double, double> point;
#define x first
#define y second

inline int sgn(const double & x) {
  return (0.0 < x) - (x < 0.0);
}

//Line and line segment intersection (see their own sections)

int line_intersection(const point & p1, const point & p2,
                      const point & p3, const point & p4, point * p = 0) {
  double a1 = p2.y - p1.y, b1 = p1.x - p2.x;
  double c1 = -(p1.x * p2.y - p2.x * p1.y);
  double a2 = p4.y - p3.y, b2 = p3.x - p4.x;
  double c2 = -(p3.x * p4.y - p4.x * p3.y);
  double x = -(c1 * b2 - c2 * b1), y = -(a1 * c2 - a2 * c1);
  double det = a1 * b2 - a2 * b1;
  if (EQ(det, 0))
    return (EQ(x, 0) && EQ(y, 0)) ? 1 : -1;
  if (p != 0) *p = point(x / det, y / det);
  return 0;
}

double norm(const point & a) { return a.x * a.x + a.y * a.y; }
double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }

const bool TOUCH_IS_INTERSECT = true;

bool contain(const double & l, const double & m, const double & h) {
  if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
  return LT(l, m) && LT(m, h);
}

bool overlap(const double & l1, const double & h1,
             const double & l2, const double & h2) {
  if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
  return LT(l1, h2) && LT(l2, h1);
}

int seg_intersection(const point & a, const point & b,
                     const point & c, const point & d,
                     point * p = 0, point * q = 0) {
  point ab(b.x - a.x, b.y - a.y);
  point ac(c.x - a.x, c.y - a.y);
  point cd(d.x - c.x, d.y - c.y);
  double c1 = cross(ab, cd), c2 = cross(ac, ab);
  if (EQ(c1, 0) && EQ(c2, 0)) { //collinear
    double t0 = dot(ac, ab) / norm(ab);
    double t1 = t0 + dot(cd, ab) / norm(ab);
    if (overlap(min(t0, t1), max(t0, t1), 0, 1)) {
      point res1 = max(min(a, b), min(c, d));
      point res2 = min(max(a, b), max(c, d));
      if (res1 == res2) {
        if (p != 0) *p = res1;
        return 0; //collinear, meeting at an endpoint
      }
      if (p != 0 && q != 0) *p = res1, *q = res2;
      return 1; //collinear and overlapping
    } else {
      return -1; //collinear and disjoint
    }
  }
  if (EQ(c1, 0)) return -1; //parallel and disjoint
  double t = cross(ac, cd) / c1, u = c2 / c1;
  if (contain(0, t, 1) && contain(0, u, 1)) {
    if (p != 0) *p = point(a.x + t * ab.x, a.y + t * ab.y);
    return 0; //non-parallel with one intersection
  }
  return -1; //non-parallel with no intersections
}

struct event {
  double y;
  int mask_delta;

  event(double y = 0, int mask_delta = 0) {
    this->y = y;
    this->mask_delta = mask_delta;
  }

  bool operator < (const event & e) const {
    if (y != e.y) return y < e.y;
    return mask_delta < e.mask_delta;
  }
};

template<class It>
double intersection_area(It lo1, It hi1, It lo2, It hi2) {
  It plo[2] = {lo1, lo2}, phi[] = {hi1, hi2};
  set<double> xs;
  for (It i1 = lo1; i1 != hi1; ++i1) xs.insert(i1->x);
  for (It i2 = lo2; i2 != hi2; ++i2) xs.insert(i2->x);
  for (It i1 = lo1, j1 = hi1 - 1; i1 != hi1; j1 = i1++) {
    for (It i2 = lo2, j2 = hi2 - 1; i2 != hi2; j2 = i2++) {
      point p;
      if (seg_intersection(*i1, *j1, *i2, *j2, &p) == 0)
        xs.insert(p.x);
    }
  }
  vector<double> xsa(xs.begin(), xs.end());
  double res = 0;
  for (int k = 0; k < (int)xsa.size() - 1; k++) {
    double x = (xsa[k] + xsa[k + 1]) / 2;
    point sweep0(x, 0), sweep1(x, 1);
    vector<event> events;
    for (int poly = 0; poly < 2; poly++) {
      It lo = plo[poly], hi = phi[poly];
      double area = 0;
      for (It i = lo, j = hi - 1; i != hi; j = i++)
        area += (j->x - i->x) * (j->y + i->y);
      for (It j = lo, i = hi - 1; j != hi; i = j++) {
        point p;
        if (line_intersection(*j, *i, sweep0, sweep1, &p) == 0) {
          double y = p.y, x0 = i->x, x1 = j->x;
          if (x0 < x && x1 > x) {
            events.push_back(event(y,  sgn(area) * (1 << poly)));
          } else if (x0 > x && x1 < x) {
            events.push_back(event(y, -sgn(area) * (1 << poly)));
          }
        }
      }
    }
    sort(events.begin(), events.end());
    double a = 0.0;
    int mask = 0;
    for (int j = 0; j < (int)events.size(); j++) {
      if (mask == 3)
        a += events[j].y - events[j - 1].y;
      mask += events[j].mask_delta;
    }
    res += a * (xsa[k + 1] - xsa[k]);
  }
  return res;
}

template<class It> double polygon_area(It lo, It hi) {
  if (lo == hi) return 0;
  double area = 0;
  if (*lo != *--hi)
    area += (lo->x - hi->x) * (lo->y + hi->y);
  for (It i = hi, j = hi - 1; i != lo; --i, --j)
    area += (i->x - j->x) * (i->y + j->y);
  return fabs(area / 2.0);
}

template<class It>
double union_area(It lo1, It hi1, It lo2, It hi2) {
  return polygon_area(lo1, hi1) + polygon_area(lo2, hi2) -
         intersection_area(lo1, hi1, lo2, hi2);
}
```

Delaunay Triangulation (Simple)

```
/* Given a range of points P on the Cartesian plane, the Delaunay Triangulation of said points is a set
     of non-overlapping triangles covering the entire convex hull of P, such that no point in P lies
     within the circumcircle of any of the resulting triangles. The triangulation maximizes the minimum
      angle of all the angles of the triangles in the triangulation. In addition, for any point p in
     the convex hull (not necessarily in P), the nearest point is guaranteed to be a vertex of the
     enclosing triangle from the triangulation. See: https://en.wikipedia.org/wiki/
     Delaunay_triangulation  The triangulation may not exist (e.g. for a set of collinear points) or it
      may not be unique (multiple possible triangulations may exist). The triangulation may not exist (
     e.g. for a set of collinear points) or it may not be unique (multiple possible triangulations may
     exist). The following program assumes that a triangulation exists, and produces one such valid
     result using one of the simplest algorithms to solve this problem. It involves encasing the
     simplex in a circle and rejecting the simplex if another point in the tessellation is within the
     generalized circle.

Time Complexity: O(n^4) on the number of input points.
*/

#include <algorithm> /* sort() */
#include <cmath>     /* fabs(), sqrt() */
#include <utility>   /* pair */
#include <vector>
using namespace std;

const double eps = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
#define LT(a, b) ((a) < (b) - eps)        /* less than */
#define GT(a, b) ((a) > (b) + eps)        /* greater than */
#define LE(a, b) ((a) <= (b) + eps)       /* less than or equal to */
#define GE(a, b) ((a) >= (b) - eps)       /* greater than or equal to */

typedef pair<double, double> point;
#define x first
#define y second

double norm(const point & a) { return a.x * a.x + a.y * a.y; }
double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }

const bool TOUCH_IS_INTERSECT = false;

bool contain(const double & l, const double & m, const double & h) {
  if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
  return LT(l, m) && LT(m, h);
}

bool overlap(const double & l1, const double & h1,
             const double & l2, const double & h2) {
  if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
  return LT(l1, h2) && LT(l2, h1);
}

int seg_intersection(const point & a, const point & b,
                     const point & c, const point & d) {
  point ab(b.x - a.x, b.y - a.y);
  point ac(c.x - a.x, c.y - a.y);
  point cd(d.x - c.x, d.y - c.y);
  double c1 = cross(ab, cd), c2 = cross(ac, ab);
  if (EQ(c1, 0) && EQ(c2, 0)) {
    double t0 = dot(ac, ab) / norm(ab);
    double t1 = t0 + dot(cd, ab) / norm(ab);
    if (overlap(min(t0, t1), max(t0, t1), 0, 1)) {
      point res1 = max(min(a, b), min(c, d));
      point res2 = min(max(a, b), max(c, d));
      return (res1 == res2) ? 0 : 1;
    }
    return -1;
  }
  if (EQ(c1, 0)) return -1;
  double t = cross(ac, cd) / c1, u = c2 / c1;
  if (contain(0, t, 1) && contain(0, u, 1)) return 0;
  return -1;
}

struct triangle { point a, b, c; };

template<class It>
vector<triangle> delaunay_triangulation(It lo, It hi) {
  int n = hi - lo;
  vector<double> x, y, z;
  for (It it = lo; it != hi; ++it) {
    x.push_back(it->x);
    y.push_back(it->y);
    z.push_back((it->x) * (it->x) + (it->y) * (it->y));
  }
  vector<triangle> res;
  for (int i = 0; i < n - 2; i++) {
    for (int j = i + 1; j < n; j++) {
      for (int k = i + 1; k < n; k++) {
        if (j == k) continue;
        double nx = (y[j] - y[i]) * (z[k] - z[i]) - (y[k] - y[i]) * (z[j] - z[i]);
        double ny = (x[k] - x[i]) * (z[j] - z[i]) - (x[j] - x[i]) * (z[k] - z[i]);
        double nz = (x[j] - x[i]) * (y[k] - y[i]) - (x[k] - x[i]) * (y[j] - y[i]);
        if (GE(nz, 0)) continue;
        bool done = false;
        for (int m = 0; m < n; m++)
          if (x[m] - x[i]) * nx + (y[m] - y[i]) * ny + (z[m] - z[i]) * nz > 0) {
            done = true;
            break;
          }
        if (!done) { //handle 4 points on a circle
          point s1[] = { *(lo + i), *(lo + j), *(lo + k), *(lo + i) };
          for (int t = 0; t < (int)res.size(); t++) {
            point s2[] = { res[t].a, res[t].b, res[t].c, res[t].a };
            for (int u = 0; u < 3; u++)
              for (int v = 0; v < 3; v++)
                if (seg_intersection(s1[u], s1[u + 1], s2[v], s2[v + 1]) == 0)
                  goto skip;
          }
          res.push_back((triangle){*(lo + i), *(lo + j), *(lo + k)});
        }
skip:;
      }
    }
  }
  return res;
}
```

String Searching (KMP)

```
/* Given an text and a pattern to be searched for within the text, determine the first position in which
      the pattern occurs in the text. The KMP algorithm is much faster than the naive, quadratic time,
     string searching algorithm that is found in string.find() in the C++ standard library.  KMP
     generates a table using a prefix function of the pattern. Then, the precomputed table of the
     pattern can be used indefinitely for any number of texts.

Time Complexity: O(n + m) where n is the length of the text
and m is the length of the pattern.

Space Complexity: O(m) auxiliary on the length of the pattern.
*/

#include <string>
#include <vector>
using namespace std;

int find(const string & text, const string & pattern) {
  if (pattern.empty()) return 0;
  //generate table using pattern
  vector<int> p(pattern.size());
  for (int i = 0, j = p[0] = -1; i < (int)pattern.size(); ) {
    while (j >= 0 && pattern[i] != pattern[j])
      j = p[j];
    i++;
    j++;
    p[i] = (pattern[i] == pattern[j]) ? p[j] : j;
  }
  //use the precomputed table to search within text
  //the following can be repeated on many different texts
  for (int i = 0, j = 0; j < (int)text.size(); ) {
    while (i >= 0 && pattern[i] != text[j])
      i = p[i];
    i++;
    j++;
    if (i >= (int)pattern.size())
      return j - i;
  }
  return string::npos;
}
```

Longest Common Subsequence

```
/* A subsequence is a sequence that can be derived from another sequence by deleting some elements
     without changing the order of the remaining elements (e.g. "ACE" is a subsequence of "ABCDE", but
     "BAE" is not). Using dynamic programming, determine the longest string which is a subsequence
     common to any two input strings.  In addition, the shortest common supersequence between two
     strings is a closely related problem, which involves finding the shortest string which has both
     input strings as subsequences (e.g. "ABBC" and "BCB" has the shortest common supersequence of "
     ABBCB"). The answer is simply: (sum of lengths of s1 and s2) - (length of LCS of s1 and s2)

Time Complexity: O(n * m) where n and m are the lengths of the two
input strings, respectively.

Space Complexity: O(n * m) auxiliary.
*/
```

```
#include <string>
#include <vector>
using namespace std;

string longest_common_subsequence
(const string & s1, const string & s2) {
  int n = s1.size(), m = s2.size();
  vector< vector<int> > dp;
  dp.resize(n + 1, vector<int>(m + 1, 0));
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (s1[i] == s2[j]) {
        dp[i + 1][j + 1] = dp[i][j] + 1;
      } else if (dp[i + 1][j] > dp[i][j + 1]) {
        dp[i + 1][j + 1] = dp[i + 1][j];
      } else {
        dp[i + 1][j + 1] = dp[i][j + 1];
      }
    }
  }
  string ret;
  for (int i = n, j = m; i > 0 && j > 0; ) {
    if (s1[i - 1] == s2[j - 1]) {
      ret = s1[i - 1] + ret;
      i--;
      j--;
    } else if (dp[i - 1][j] < dp[i][j - 1]) {
      j--;
    } else {
      i--;
    }
  }
  return ret;
}
```

Suffix and LCP Array (N log N)

```
/* A suffix array SA of a string S[1..n] is a sorted array of indices of all the suffixes of S ("abc"
      has suffixes "abc", "bc", and "c"). SA[i] contains the starting position of the i-th smallest
      suffix in S, ensuring that for all 1 < i <= n, S[SA[i - 1], n] < S[A[i], n] holds. It is a simple,
       space efficient alternative to suffix trees. By binary searching on a suffix array, one can
      determine whether a substring exists in a string in O(log n) time per query.  The longest common
      prefix array (LCP array) stores the lengths of the longest common prefixes between all pairs of
      consecutive suffixes in a sorted suffix array and can be found in O(n) given the suffix array.
      The following algorithm uses a "gap" partitioning algorithm explained here: http://stackoverflow.
      com/a/17763563, except that the O(n log n) comparison-based sort is substituted for an O(n)
      counting sort to reduce the running time by an order of log n.

Time Complexity: O(n log n) for suffix_array() and O(n) for
lcp_array(), where n is the length of the input string.

Space Complexity: O(n) auxiliary.
*/

#include <algorithm>
#include <string>
#include <vector>
using namespace std;

const string * str;

bool comp(const int & a, const int & b) {
  return (*str)[a] < (*str)[b];
}

vector<int> suffix_array(const string & s) {
  int n = s.size();
  vector<int> sa(n), order(n), rank(n);
  for (int i = 0; i < n; i++)
    order[i] = n - 1 - i;
  str = &s;
  stable_sort(order.begin(), order.end(), comp);
  for (int i = 0; i < n; i++) {
    sa[i] = order[i];
    rank[i] = (int)s[i];
  }
  vector<int> r(n), cnt(n), _sa(n);
  for (int len = 1; len < n; len *= 2) {
    r = rank;
    _sa = sa;
    for (int i = 0; i < n; i++)
      cnt[i] = i;
    for (int i = 0; i < n; i++) {
      if (i > 0 && r[sa[i - 1]] == r[sa[i]] && sa[i - 1] + len < n &&
          r[sa[i - 1] + len / 2] == r[sa[i] + len / 2]) {
        rank[sa[i]] = rank[sa[i - 1]];
      } else {
        rank[sa[i]] = i;
      }
    }
    for (int i = 0; i < n; i++) {
      int s1 = _sa[i] - len;
      if (s1 >= 0)
        sa[cnt[rank[s1]]++] = s1;
    }
  }
  return sa;
}

vector<int> lcp_array(const string & s,
                           const vector<int> & sa) {
  int n = sa.size();
  vector<int> rank(n), lcp(n - 1);
  for (int i = 0; i < n; i++)
    rank[sa[i]] = i;
  for (int i = 0, h = 0; i < n; i++) {
    if (rank[i] < n - 1) {
      int j = sa[rank[i] + 1];
      while (max(i, j) + h < n && s[i + h] == s[j + h])
        h++;
      lcp[rank[i]] = h;
      if (h > 0) h--;
    }
  }
  return lcp;
}
```

Andrea AVL

```
#include <cstdio>
#include <cassert>
#include <algorithm>
#include <cmath>
using namespace std;

//16.05
//16.43
//17.32

enum CHILD {
  LEFT,
  RIGHT
};

struct node {
  int size;
  int value;
  node* left;
  node* right;
} nodes[10000000];

node *freen;

#define SZ(n) ((n) ? (n)->size : 0)

inline void updateup(node *&n) {
  n->size = SZ(n->left) + SZ(n->right) + 1;
}

inline void rrotate(node *&n) {
  // assert(n && n->child[!mode]);

  node *oldroot = n;
  n = oldroot->left;
  oldroot->left = n->right;
  n->right = oldroot;

  updateup(oldroot);
  updateup(n);
}

inline void lrotate(node *&n) {
  // assert(n && n->child[!mode]);

  node *oldroot = n;
  n = oldroot->right;
  oldroot->right = n->left;
  n->left = oldroot;

  updateup(oldroot);
  updateup(n);
}
```

```
inline int coeff(node *&n) {
    return SZ(n->left) - SZ(n->right);
}

void balance(node *&n) {
  int cf = coeff(n);
  if (cf > 1) {
    // if (coeff(n->left) < 0)
    //   lrotate(n->left);
    rrotate(n);
  }
  else if (cf < -1) {
    // if (coeff(n->right) > 0)
    //   rrotate(n->right);
    lrotate(n);
  }
}

void insert(node *&n, int pos, int value) {
  if (!n) {
    n = freen;
    n = new node {1, value, NULL, NULL};
    return;
  }

  int ls = SZ(n->left);
  if (pos <= ls) {
    insert(n->left, pos, value);
  }
  else {
    insert(n->right, pos-ls-1, value);
  }
  updateup(n);
  balance(n);
}

int reml(node *&n) {
  if (!n->left) {
    int val = n->value;
    node *tmp = n;
    n = n->right;
    freen = tmp;
    return val;
  }
  int v = reml(n->left);
  updateup(n);
  balance(n);
  return v;
}

int del(node *&n, int pos) {
  if (!n) return -1;
  int val;

  int ls = SZ(n->left);
  if (pos < ls) {
    val = del(n->left, pos);
  }
  else if (pos > ls) {
    val = del(n->right, pos-ls-1);
  }
  else {
    val = n->value;
    if (!n->right) {
      freen = n;
      n = n->left;
    }
    else {
      n->value = reml(n->right);
    }
  }
  if (n) {
    updateup(n);
    balance(n);
  }
  return val;
}

int get(node *n, int pos) {
  if (!n) return -1;

  int ls = SZ(n->left);

  if (pos < ls) {
    return get(n->left, pos);
  }
  else if (pos > ls) {
    return get(n->right, pos-ls-1);
  }

  return n->value;
}

void print(node *n, int d = 0) {
  if (!n) return;
  // for (int i = 0; i < d; i++) printf("\t");
  print(n->left, d+1);
  printf("%d ", n->value);//, n->size, SZ(n->left), SZ(n->right));
  print(n->right, d+1);
}

void create(node *&root, int s, int e) {
  if (e < s) return;
  int m = (s + e) / 2;
  root = nodes + m;
  *root = node {1, m, NULL, NULL};
  if (s == e) {
    return;
  }
  create(root->left, s, m-1);
  create(root->right, m+1, e);
  updateup(root);
}


int main(int argc, char const *argv[]) {

  int N, Q;

  assert(freopen("input.txt","r",stdin) != NULL);
  assert(freopen("output.txt","w",stdout) != NULL);
  assert(scanf("%d%d", &N, &Q) == 2);


  node *root = NULL;
  create(root, 0, N);
  // dfs(tree.root, 0);
  // printf("Done.\n");
  for (int i = 0; i < Q; i++) {
    // for (int i = 0; i < N; i++) {printf("%d ", blk.getval(i));}
    // puts("");

    char op[2];
    int start;
    assert(scanf("%1s %d\n", op, &start) == 2);
    switch (op[0]) {
      case 's': {
        int end;
        assert(scanf("%d", &end) == 1);
        // printf("S: %c %d %d\n", op[0], start, end);
        // printf("***************** Removal.\n");
        int tmp = del(root, start);
        // dfs(tree.root, 0);
        // printf("Done.\n");
        // printf("***************** Insertion.\n");
        insert(root, end, tmp);
        // assert(root->height < (log2(N)+2)*2);
        // print(root);
        // puts("");
        // dfs(tree.root, 0);
        // printf("Done.\n");
        break;
      }
      case 'c':
        printf("%d ", get(root, start));
        break;
    }
  }
  puts("");

  // for (int i = 0; i < N; i++) {printf("%d ", blk.getval(i));}
  // puts("");


  return 0;
}

// int main() {
//   node *root = NULL;
//   create(root, 0, 60);
//   // for (int i = 0; i < 60; i++) {
//   //   insert(root, i, i);
//   //   // del(root, 0);
//   //
//   //   // print(root);
//   // }
//   del(root, 15);
//   // del(root, 5);
//   // del(root, 5);
//   printf("Root: %d\n", root->size);
//   for (int i = 0; i < 60; i++) {
```

```
//     printf("%d ", get(root, i));
//   }
//
//
//   puts("");
// }
```

Andrea kdtree

```
#include <cstdio>
#include <vector>
#include <algorithm>
#include <cassert>
using namespace std;

#define INF (1000000000)

typedef int ll;

struct point {
  ll x, y;
  inline bool operator ==(point const& other) const {
    return x == other.x && y == other.y;
  }
};

inline ll absll(ll x) {
  return x < 0 ? -x : x;
}

inline ll manh(ll x, ll y, point const& pt) {
  // printf("%lld %lld\n", x, y);
  return absll(pt.x-x) + absll(pt.y-y);
}

struct node {

  node(point *pts, int size) {
    x1 = x2 = pts[0].x;
    y1 = y2 = pts[0].y;
    for (int i = 1; i < size; i++) {
        x1 = min(x1, pts[i].x);
        x2 = max(x2, pts[i].x);
        y1 = min(y1, pts[i].y);
        y2 = max(y2, pts[i].y);
    }
    left = right = NULL;
    len = 0;
  }

  ll manhattan(point const& pt) {
    if (x1 <= pt.x && pt.x <= x2) {
      if (y1 <= pt.y && pt.y <= y2)
        return 0;
      return min(absll(pt.y-y1), absll(pt.y-y2));
    }
    if (y1 <= pt.y && pt.y <= y2)
      return min(absll(pt.x-x1), absll(pt.x-x2));

    ll dist = manh(x1, y1, pt);
    dist = min(dist, manh(x1, y2, pt));
    dist = min(dist, manh(x2, y1, pt));
    return min(dist, manh(x2, y2, pt));
  }

  // INTERSECTION contained(point const& pt, ll radsq) {
  //   ll d1 = manh(x1, y1, pt);
  //   ll d2 = manh(x1, y2, pt);
  //   ll d3 = manh(x2, y1, pt);
  //   ll d4 = manh(x2, y2, pt);
  //   if (d1 <= radsq &&
  //       d2 <= radsq &&
  //       d3 <= radsq &&
  //       d4 <= radsq)
  //           return CONTAINED;
  //
  //
  //   if (x1 <= pt.x && pt.x <= x2) {
  //     if (y1 <= pt.y && pt.y <= y2) {
  //       return INTERSECT;
  //     }
  //     if (min((pt.y-y1)*(pt.y-y1), (pt.y-y2)*(pt.y-y2)) <= radsq)
  //       return INTERSECT;
  //     return DISTINCT;
  //   }
  //   if (y1 <= pt.y && pt.y <= y2) {
  //     if (min((pt.x-x1)*(pt.x-x1), (pt.x-x2)*(pt.x-x2)) <= radsq)
  //       return INTERSECT;
  //     else
  //       return DISTINCT;
  //   }
  //   if (min(min(d1, d2), min(d3, d4)) > radsq) return DISTINCT;
  //   return INTERSECT;
  // }

  ll x1, y1;
  ll x2, y2;
  point *pts;
  int len;
  node *left;
  node *right;
};

struct xsort {
  inline bool operator()(point const& a, point const& b) {
    return a.x < b.x;
  }
};

struct ysort {
  inline bool operator()(point const& a, point const& b) {
    return a.y < b.y;
  }
};


node *create(point* pts, int len, int dim = 0) {
  node *n = new node(pts, len);
  if (len <= 96) {
    n->pts = pts;
    n->len = len;
    return n;
  }

  if (dim)
    nth_element(pts, pts+(len/2), pts+len, xsort());
  else
    nth_element(pts, pts+(len/2), pts+len, ysort());

  int cnt = len / 2;
  // while (cnt < len-1 && pts[cnt+1] == pts[cnt]) cnt++;

  n->left = create(pts, cnt, !dim);
  n->right = create(pts+cnt, len-cnt, !dim);
  return n;
}

ll nearest(node *root, point query, ll bsf) {
  if (root->len) {
    ll dst = INF;//manh(best.x, best.y, query);
    for (int i = 0; i < root->len; i++) {
      ll dst2 = manh(root->pts[i].x, root->pts[i].y, query);
      if (dst2 < dst && dst2>0) {
        dst = dst2;
      }
    }
    return dst;
  }

  ll ld = root->left ? root->left->manhattan(query) : INF;
  ll rd = root->right ? root->right->manhattan(query) : INF;
  bool right;

  if (ld <= rd && ld < bsf) {
    bsf = min(bsf, nearest(root->left, query, bsf));
    right = false;
  }
  else if (rd <= ld && rd < bsf) {
    bsf = min(bsf, nearest(root->right, query, bsf));
    right = true;
  }
  else
    return bsf;


  if (!right && rd < bsf) {
    bsf = min(bsf, nearest(root->right, query, bsf));
  }
  else if (right && ld < bsf) {
    bsf = min(bsf, nearest(root->left, query, bsf));
  }
  return bsf;
}

// ll countcircle(node *root, point center, ll sizesq) {
//   func++;
//
//   INTERSECTION status = root->contained(center, sizesq);
//
//    switch (status) {
//      case DISTINCT: return 0;
//      case CONTAINED: return root->cnt;
//    }
//
//    int count = 0;
//
//    if (root->pts.size()) {
//      for (int i = 0; i < root->pts.size(); i++) {
//        ll dst = manh(root->pts[i].x, root->pts[i].y, center);
//        if (dst <= sizesq) {
//          count++;
//        }
//      }
//      return count;
//    }
//
//    ll ld = root->left ? root->left->manhattan(center) : INF;
//    ll rd = root->right ? root->right->manhattan(center) : INF;
//
//    if (ld <= sizesq) {
//      count += countcircle(root->left, center, sizesq);
//    }
//    if (rd <= sizesq) {
//      count += countcircle(root->right, center, sizesq);
//    }
//    return count;
// }


ll naive(point* pts, int len, point query) {
  ll dist = max(manh(query.x, query.y, pts[0]), manh(query.x, query.y, pts[1]));
  for (int i = 0; i < len; i++) {
    ll d = manh(query.x, query.y, pts[i]);
    if (d < dist && d) {
      dist = d;
    }
  }
  return dist;
}

ll naivecnt(point* pts, int len, point center, ll radsq) {
  ll count = 0;
  for (int i = 0; i < len; i++) {
    ll d = manh(center.x, center.y, pts[i]);
    if (d <= radsq) {
      count++;
    }
  }
  return count;
}
```

```
void test() {
  vector<point> tmp;
  int N = 1000000;
  for (int i = 0; i < N; i++) tmp.push_back(point {rand()%1000000-N/2, rand()%1000000-N/2});
  node *kd = create(tmp.data(), tmp.size());

  int a = 0;
  for (int i = 0; i < N; i++) {
    point query = point {rand()%1000000, rand()%1000000};
    // func = 0;
    ll res = nearest(kd, query, INF);
    // ll radsq = rand()%1000000; radsq *= radsq;
    // ll res = countcircle(kd, query, radsq);
    // printf("CALL %d\n", func);

    // ll nai = naivecnt(tmp.data(), tmp.size(), query, radsq);

    ll nai = naive(tmp.data(), tmp.size(), query);
    a += res;
    // printf("KD %lld\n", res);
    // printf("NV %lld\n", nai);
    assert(res == nai);
    // printf("KD %lld %lld %f\n", res.x, res.y, sqrt(manh(res.x, res.y, query)));
    // printf("NV %lld %lld %f\n", nai.x, nai.y, sqrt(manh(nai.x, nai.y, query)));
    // assert(manh(res.x, res.y, query) == manh(nai.x, nai.y, query));
  }
  printf("A%d\n", a);
}

point pts[1000100];
point pts2[1000100];
int main(int argc, char const *argv[]) {
  // test();
  // return 0;
  int N;
  scanf("%d", &N);
  for (int i = 0; i < N; i++) {
    scanf("%d%d",&pts[i].x, &pts[i].y);
    pts2[i] = pts[i];
  }
  node *kd = create(pts, N);

  for (int j = 0; j < N; j++) {
    ll res = nearest(kd, pts2[j], INF);
    // assert(nai == res);

    printf("%d\n", res);
  }

  return 0;
}
```