Competitive Programming Library

Collection by Mitko Nikov

October 2, 2023

Intentionally left blank.

Contents

1 Foreword

Dear Reader, I collected this set of algorithms and data structures from various sources over the years and now is the time to give it all to the world. This collection of algorithms and data structures contains some very well known ones, but also some that are so specific, that they have a single use case only.

Some of the algorithms, for shorter code use the following template:

```
#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define all(c) ((c).begin()), ((c).end())
#define sz(x) (int)(x).size()
typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;
```

Always try to do the following checks before submitting a problem:

- Edge cases?
- Overflows?
- Memory allocation (MLE)?
- Out-of-bounds on array access?
- Did you escape all characters?
- Recursion depth, stack memory?

I dedicate this book to You - all of the people that work hard, even if they know that they will never be appreciated for what they have accomplished or what they tried to achieve.

2 Graphs

For every graph algorithm, the nodes will be numbered from 0 to $N-1$.
When talking about time complexity, we will interchangeably use N as the number of vertices in a graph and M as the number of edges.

2.1 Topological Sort

Time Complexity: $O(N)$

Given adjacency list of the graph, the following DFS creates a topological order of the graph nodes:

```
vector<bool> visited;
vector<int> sorted;

void dfs(int v) {
    visited[v] = true;
    for (auto u: adj[v]) {
        if (!visited[u]) {
            dfs(u);
        }
    }

    sorted.push_back(v);
}
```

```
void sort() {
  for (int i = 0; i < n; i++) {
      if (!visited[i]) dfs(i);
  }

  reverse(
    sorted.begin(),
    sorted.end()
  );
}
```

2.1.1 Consistent Topological Sort

Time Complexity: $O(N)$

If we have multiple components, this algorithm can jump between components due to the order of the nodes. Thus, we can extend this algorithm, by classifying the nodes by components and performing topological sort on each component separately.

```
vector<vector<int>> adj;
vector<vector<int>> tree_adj;

// All vectors should be resized to size N
vector<int> visited; // Topological Sort
vector<int> sorted; // Output
vector<bool> visited_comp;
vector<vector<int>> comps;

void dfs_comp(int u, int p, int c) {
    comps[c].push_back(u);
    visited_comp[u] = true;
    for (auto v: tree_adj[u]) {
        if (v == p || visited_comp[v]) continue;
        dfs_comp(v, u, c);
    }
}
```

```
void sort() {
  // Classify the nodes by
  // the component they are in.
  // Time Complexity is still O(N)
  int comp_id = 0;
  for (int i = 0; i < n; i++) {
      if (visited_comp[i]) continue;
      dfs_comp(i, -1, comp_id);
      comp_id++;
  }

  for (int i = 0; i < comp_id; i++) {
      for (auto u: comps[i]) {
          if (visited[u]) continue;
          // This is the same function
          // as in the ordinary Topo-Sort.
          dfs(u);
      }
  }

  // You may want to reverse
  // each component separately
  reverse(sorted.begin(), sorted.end());
}
```

2.2 Euler Tour

In graph theory, an Euler Tour is a tour in a graph that visits every edge exactly once (allowing for revisiting vertices). Similarly, an Eulerian circuit or Eulerian cycle is an Eulerian tour that starts and ends on the same vertex.

Let us denote an undirected connected graph as UCG and directed connected graph as DCG. Then the following properties hold:

- An UCG has an Eulerian cycle iff $deg(v) \equiv 0 \pmod 2, \forall v \in V$
- An UCG can be decomposed into edge-disjoint cycles iff has an Eulerian cycle.
- An UCG has an Eulerian trail iff exactly zero or two vertices have odd degree.
- A DCG has an Eulerian cycle iff $deg_{in}(v) = deg_{out}(v), \forall v \in V$, and all of its vertices with nonzero degree belong to a single strongly connected component.
- A DCG has an Eulerian trail iff at most one vertex has $deg_{out}(v) - deg_{in}(v) = 1$, for every other vertex holds $deg_{in}(v) = deg_{out}(v)$, and all of its vertices with nonzero degree belong to a single connected component of the underlying undirected graph.

The following algorithm finds an Euler Cycle or Tour if there exists. Otherwise, it returns an empty vector. Input should be a vector of (dest, global edge index), where for undirected graphs, forward/backward edges have the same index.

Time Complexity: $O(N)$
Implementation: Simon Lindholm

```
vi eulerWalk(vector<vector<pii>>& gr, int nedges, int src=0) {
   int n = sz(gr);
   vi D(n), its(n), eu(nedges), ret, s = {src};
   D[src]++; // to allow Euler paths, not just cycles
   while (!s.empty()) {
      int x = s.back(), y, e, &it = its[x], end = sz(gr[x]);
      if (it == end){ ret.push_back(x); s.pop_back(); continue; }
      tie(y, e) = gr[x][it++];
      if (!eu[e]) {
         D[x]--, D[y]++;
         eu[e] = 1; s.push_back(y);
      // To get edge indices back, add .second to s and ret.
      }}
   for (int x : D) if (x < 0 || sz(ret) != nedges+1) return {};
   return {ret.rbegin(), ret.rend()};
}
```

An example of how the adjacency list has to look like. In this case we have an undirected graph.

```
for (int i = 0; i < m; i++) {
  int u, v; cin >> u >> v; u--; v--;
  adj[u].emplace_back(v, i);
  adj[v].emplace_back(u, i);
}
```

2.3 Cycles

Time Complexity: $O(N)$

We can ask the following question about a *DCG*:
- How many nodes are a part of any cycle?

This problem can be answered in $O(N)$ time by performing a DFS and keeping track for each node whether or not it's on the current stack in our DFS. If the node is on the stack and we have hit it again, then we know we have reached a cycle. When exiting the DFS we update all of the "parent" nodes with this information.

```cpp
vector<vector<int>> adj; // directed graph
vector<bool> recStack, visited, is_cycle;

bool dfs(int u) {
    recStack[u] = visited[u] = true;
    for (auto v: adj[u]) {
        if (is_cycle[v] || (!visited[v] && dfs(v)) || recStack[v]) {
            // If you don't need to reset the recursion stack
            // you won't need to check is_cycle[v]
            recStack[u] = false;
            return is_cycle[v] = true;
        }
    }

    recStack[u] = false;
    return false;
}

// In the main function, call dfs() for each node
for (int i = 0; i < n; i++) {
    if (!visited[i] && dfs(i)) {
        is_cycle[i] = true;
    }
}
```

2.4 Bipartite Checker

Time Complexity: $O(N)$

This BFS-based algorithm checks if a given graph is bipartite. It finds out a bipartite coloring as well.

Remark 1 *If the graph is not connected, you will need to call this function separately on each component, while making the color array global to avoid square complexity.*

```
vector<vector<int>> adj;
vector<int> color;

bool isBipartite() {
    color.resize((int)adj.size(), -1);
    color[0] = 1;

    queue<int> q;
    q.push(0);
    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (auto v: adj[u]) {
            if (v == u) return false;
            if (color[v] == -1) {
                color[v] = 1 - color[u];
                q.push(v);
            } else if (color[v] == color[u]) {
                return false;
            }
        }
    }

    return true;
}
```

2.5 2SAT

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

struct TWOSAT {
    int n;
    vector<vector<int>> adj, adj_t;
    vector<bool> used;
    vector<int> order, comp;
    vector<bool> assignment;

    TWOSAT(int n) {
        this->n = n;
        adj.resize(n);
        adj_t.resize(n);
        used.resize(n);
        order.resize(n);
        comp.resize(n);
        assignment.resize(n);
    }

    void dfs1(int v) {
        used[v] = true;
        for (int u : adj[v]) {
            if (!used[u])
                dfs1(u);
        }
        order.push_back(v);
    }

    void dfs2(int v, int cl) {
        comp[v] = cl;
        for (int u : adj_t[v]) {
            if (comp[u] == -1)
                dfs2(u, cl);
        }
    }

    bool solve_2SAT() {
        order.clear();
        used.assign(n, false);
        for (int i = 0; i < n; ++i) {
            if (!used[i])
                dfs1(i);
        }

        comp.assign(n, -1);
        for (int i = 0, j = 0; i < n; ++i) {
            int v = order[n - i - 1];
            if (comp[v] == -1)
                dfs2(v, j++);
        }

        assignment.assign(n / 2, false);
        for (int i = 0; i < n; i += 2) {
            if (comp[i] == comp[i + 1])
                return false;
            assignment[i / 2] = comp[i] > comp[i + 1];
        }
        return true;
    }

    // (a or b) and ...
    void add_disjunction(int a, bool na, int b, bool nb) {
        // na and nb signify whether a and b are to be negated
        a = 2*a ^ na;
        b = 2*b ^ nb;
        int neg_a = a ^ 1;
        int neg_b = b ^ 1;
        adj[neg_a].push_back(b);
        adj[neg_b].push_back(a);
        adj_t[b].push_back(neg_a);
        adj_t[a].push_back(neg_b);
    }
};
```

2.6 Articulation Points

Time Complexity: $O(N + M)$
Space Complexity: $O(N)$

If you remove an articulation point (cut vertex) in a graph, the graph will split into more components than originally. They represent vulnerabilities in a connected network. The following implementation [9] calls the `process()` function when each articulation point is found.

```
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer = 0;
void process(int v) {
    // v is articulation point and process it
    // if v is cut down => the graph will be disconnected
}
void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    int children = 0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] >= tin[v] && p!=-1)
                process(v);
            ++children;
        }
    }
    if(p == -1 && children > 1)
        process(v);
}
void find_cutpoints() {
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i]) dfs (i);
    }
}
```

2.7 Bridges

Time Complexity: $O(N + M)$
Space Complexity: $O(N)$

Bridges are very similar to Articulation Points in a graph, except that bridges are edges for which it holds that their removal increases the number of connected components. The following implementation [3] finds bridges and articulation points with one DFS.

```
// adj[u] = adjacent nodes of u
// ap = articulation points (output)
// p = parent
// disc[u] = discovery time of u
// low[u] = 'low' node of u

int timer = 0;

int dfs(int u, int p) {
  int children = 0;
  low[u] = disc[u] = ++timer;
  for (int& v : adj[u]) {
    // we don't want to go back through the same path.
    // if we go back is because we found another way back
    if (v == p) continue;
    if (!disc[v]) { // if V has not been discovered before
      children++;
      dfs(v, u);
      if (disc[u] <= low[v])
        ap[u] = 1;
      low[u] = min(low[u], low[v]);
      // low[v] might be an ancestor of u
    } else
      // if v was already discovered means
      // that we found an ancestor
      // => finds the ancestor with the least discovery time
      low[u] = min(low[u], disc[v]);
  }
  return children;
}

void solve() {
  ap = low = disc = vector<int>(adj.size());
  for (int u = 0; u < adj.size(); u++)
    if (!disc[u])
      ap[u] = dfs(u, u) > 1;
}
```

Example Problems

Problem 1 - *Street Directions (UVa)* *[13]*

Given an undirected graph. Find a directed configuration of the same graph such that you convert as many undirected edges to directed edges as possible and the graph will still remain strongly connected.

2.8 Maximum Matchings

2.8.1 Unweighted Bipartite Graphs

Time Complexity: $O(M\sqrt{N})$
Space Complexity: $O(M + N)$

If we know that an unweighted graph is bipartite, we can solve the maximum matching problem fairly easy with the Hopcroft-Karp algorithm. [21] The algorithm takes an adjacency list as an input and produces a maximum cardinality matching as output in the `match` vector i.e. the matched node for each node is written in the `match` vector.

```
vector<vector<int>> adj;
vector<int> match;
vector<int> dist;
bool bfs() {
  queue<int> q;
  fill(dist.begin(), dist.end(), -1);
  for (int i = 0; i < n; i++) {
    if (match[i] == -1) {
      q.push(i);
      dist[i] = 0;
    }
  }
  bool reached = false;
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int v : adj[u]) {
      if (match[v] == -1) reached =
          true;
      else if (dist[match[v]] == -1) {
        dist[match[v]] = dist[u] + 1;
        q.push(match[v]);
      }
    }
  }
  return reached;
}
```

```
bool dfs(int u) {
  if (u == -1) return true;
  for (int v : adj[u]) {
    if (match[v] == -1 ||
        dist[match[v]] == dist[u] + 1) {
      if (dfs(match[v])) {
        match[v] = u, match[u] = v;
        return true;
      }
    }
  }
  return false;
}

int hopcroft_karp() {
  fill(match.begin(), match.end(), -1);
  int matching = 0;
  while (bfs()) {
    for (int i = 0; i < n; i++)
      if (match[i] == -1 && dfs(i))
        matching++;
  }
  return matching;
}
```

Implementation: Eric K. Zhang

Remark 2 *The knights' graph in a chessboard is a bipartite graph.*

Example Problems

Problem 2 - *Fast Maximum Matching (SPOJ)* *[12]*
There are $N \leq 5 \cdot 10^4$ cows and $M \leq 5 \cdot 10^4$ bulls. There are also $P \leq 1.5 \cdot 10^5$ compatible (cow, bull) pairs. Find the maximum number of (cow, bull) matches we can do.

Problem 3 - *Gambit (CODEFU)* *[2]*
There is a $N \times M$ chessboard with some squares which are occupied. Find the maximum number of knights you can place on non-occupied squares, so no two of them attack each other.

2.8.2 Kőnig's theorem

A **minimum node cover** of a graph is a minimum set of nodes such that each edge of the graph has at least one endpoint in the set. In a general graph, finding a minimum node cover is a NP-hard problem. However, if the graph is bipartite, Kőnig's theorem tells us that the size of a minimum node cover and the size of a maximum matching are always equal. The nodes that do not belong to a minimum node cover form a **maximum independent set**.

2.8.3 Weighted Bipartite Graphs

In weighted bipartite graphs, the matching problem can be solved with the Hungarian Algorithm or reduced to a Min Cost Max Flow problem where one of the bipartite sets is connected to s via 1 capacity, 0 cost edges and the other set is connected with t via 1 capacity, 0 cost edges. All of the other edges between the two sets should have 1 capacity with their original cost.

2.8.4 Max Flow - Edmonds-Karp

Worst-Case Time Complexity: $O(NM^2)$
In practice it's much faster.
Space Complexity: $O(N^2)$

Edmonds-Karp algorithm for finding a maximum flow in a graph uses consecutive BFS runs to fill up the network with flow.

```cpp
class MaxFlow {
 public:
  int n;
  vector<vector<int>> adj;
  // stores the residual flow
  vector<vector<int>> capacity;
  // stores the forward flow
  vector<vector<int>> flows;
  // stores the edges direction
  vector<vector<bool>> direction;

  struct Flow {
      int node;
      int flow;
  };

  MaxFlow(int n) {
      this->n = n;
      this->adj.resize(n, vector<int>());
      this->capacity.resize(n, vector<int>(n, 0));
      this->direction.resize(n, vector<bool>(n, 0));
      this->flows.resize(n, vector<int>(n, 0));
  }

  int bfs(int source, int sink, vector<int>& parent) {
      fill(parent.begin(), parent.end(), -1);
      parent[source] = -2; // NULL value
      queue<Flow> q;
      q.push({ source, INT_MAX });

      while(!q.empty()) {
          int currentNode = q.front().node;
          int flow = q.front().flow;
          q.pop();

          for (int nextNode: adj[currentNode]) {
              if (parent[nextNode] == -1 &&
                  capacity[currentNode][nextNode] > 0) {
                  parent[nextNode] = currentNode;
                  int new_flow = min(flow,
                        capacity[currentNode][nextNode]);
                  if (nextNode == sink) return new_flow;
                  q.push({ nextNode, new_flow });
                  assert(new_flow != INT_MAX);
              }
          }
      }

      return 0;
  }
```

```cpp
    int getMaxFlow(int source, int sink) {
        int flow = 0;
        vector<int> parent(n);
        int new_flow;

        while (new_flow = bfs(source, sink, parent)) {
            flow += new_flow;
            int current = sink;
            while (current != source) {
                int prev = parent[current];
                flows[prev][current] += new_flow;
                flows[current][prev] -= new_flow;
                capacity[prev][current] -= new_flow;
                capacity[current][prev] += new_flow;
                current = prev; // go back
            }
        }

        return flow;
    };

    map<pair<int, int>, int> getEdges() {
        // edge (u, v) mapped to flow
        map<pair<int, int>, int> result;
        for (int i = 0; i < (int)adj.size(); i++) {
            for (int j = 0; j < (int)adj[i].size(); j++) {
                int nextNode = adj[i][j];
                if (flows[i][nextNode] > 0 &&
                      direction[i][nextNode]) {
                    result[{ i, nextNode }] =
                          flows[i][nextNode];
                }
            }
        }

        return result;
    }
};
```

2.8.5 Max Flow - Push-Relabel

Worst-Case Time Complexity: $O(N^2\sqrt{M})$
In practice it's much faster.
Space Complexity: $O(N^2)$
Implementation: Simon Lindholm

```cpp
struct PushRelabel {
  struct Edge {
    int dest, back;
    ll f, c;
  };
  vector<vector<Edge>> g;
  vector<ll> ec;
  vector<Edge*> cur;
  vector<vi> hs; vi H;
  PushRelabel(int n) :
    g(n), ec(n), cur(n), hs(2*n), H(n) {}

  void addEdge(int s, int t, ll cap, ll rcap=0) {
    if (s == t) return;
    g[s].push_back({t, sz(g[t]), 0, cap});
    g[t].push_back({s, sz(g[s])-1, 0, rcap});
  }

  void addFlow(Edge& e, ll f) {
    Edge &back = g[e.dest][e.back];
    if (!ec[e.dest] && f)
         hs[H[e.dest]].push_back(e.dest);
    e.f += f; e.c -= f; ec[e.dest] += f;
    back.f -= f; back.c += f; ec[back.dest] -= f;
  }
```

```cpp
  ll calc(int s, int t) {
    int v = sz(g); H[s] = v; ec[t] = 1;
    vi co(2*v); co[0] = v-1;
    rep(i,0,v) cur[i] = g[i].data();
    for (Edge& e : g[s]) addFlow(e, e.c);

    for (int hi = 0;;) {
      while (hs[hi].empty()) if (!hi--) return -ec[s];
      int u = hs[hi].back(); hs[hi].pop_back();
      while (ec[u] > 0) // discharge u
        if (cur[u] == g[u].data() + sz(g[u])) {
          H[u] = 1e9;
          for (Edge& e : g[u]) if (e.c && H[u] >
              H[e.dest]+1)
            H[u] = H[e.dest]+1, cur[u] = &e;
          if (++co[H[u]], !--co[hi] && hi < v)
            rep(i,0,v) if (hi < H[i] && H[i] < v)
              --co[H[i]], H[i] = v + 1;
          hi = H[u];
        } else if (cur[u]->c && H[u] ==
            H[cur[u]->dest]+1)
          addFlow(*cur[u], min(ec[u], cur[u]->c));
        else ++cur[u];
    }
  }
  bool leftOfMinCut(int a) { return H[a] >= sz(g); }
}
```

2.8.6 Max Flow - Capacity Scaling

2.8.7 Min Cost Max Flow

The cost of the flow is defined as:

$$c(F) = \sum_{e \in E} flow(e) \cdot cost(e)$$

Duplicate or antiparallel edges with different costs are allowed, but **negative cycles are not allowed.**

```cpp
template<int V, class T=long long>
class mcmf {
  const T INF = numeric_limits<T>::max();

  struct edge {
    int t, rev;
    T cap, cost, f;
  };

  vector<edge> adj[V];
  T dist[V];
  int pre[V];
  bool vis[V];

  // void spfa(int s) {};

  priority_queue<pair<T, int>, vector<pair<T, int> >,
    greater<pair<T, int> > > pq; /* for dijkstra */

  void dijkstra(int s) {
    memset(pre, -1, sizeof pre);
    memset(vis, 0, sizeof vis);
    fill(dist, dist + V, INF);

    dist[s] = 0;
    pq.emplace(0, s);
    while (!pq.empty()) {
      int v = pq.top().second;
      pq.pop();
      if (vis[v]) continue;
      vis[v] = true;
      for (auto e : adj[v]) if (e.cap != e.f) {
        int u = e.t;
        T d = dist[v] + e.cost;
        if (d < dist[u]) {
          dist[u] = d, pre[u] = e.rev;
          pq.emplace(d, u);
        }
      }
    }
  }

  void reweight() {
    for (int v = 0; v < V; v++)
      for (auto& e : adj[v])
        e.cost += dist[v] - dist[e.t];
  }
```

```cpp
public:
  void add(int u, int v, T cap=1, T cost=0) {
    adj[u].push_back({ v, (int) adj[v].size(), cap,
        cost, 0 });
    adj[v].push_back({ u, (int) adj[u].size() - 1, 0,
        -cost, 0 });
  }

  pair<T, T> calc(int s, int t) {
    spfa(s); /* comment out if all costs are
        non-negative */
    T totalflow = 0, totalcost = 0;
    T fcost = dist[t];
    while (true) {
      reweight();
      dijkstra(s);
      if (~pre[t]) {
        fcost += dist[t];
        T flow = INF;
        for (int v = t; ~pre[v]; v = adj[v][pre[v]].t) {
          edge& r = adj[v][pre[v]];
          edge& e = adj[r.t][r.rev];
          flow = min(flow, e.cap - e.f);
        }
        for (int v = t; ~pre[v]; v = adj[v][pre[v]].t) {
          edge& r = adj[v][pre[v]];
          edge& e = adj[r.t][r.rev];
          e.f += flow;
          r.f -= flow;
        }
        totalflow += flow;
        totalcost += flow * fcost;
      }
      else break;
    }
    return { totalflow, totalcost };
  }

  void clear() {
    for (int i = 0; i < V; i++) {
      adj[i].clear();
      dist[i] = pre[i] = vis[i] = 0;
    }
  }
};
```

If the costs can be negative, we need to use shortest path finding algorithm which can handle negative costs. The Shortest Path Faster Algorithm is an improvement of the Bellman-Ford algorithm. The worst-case running time of the algorithm is $O(|V| \cdot |E|)$, just like the standard Bellman-Ford algorithm. Experiments suggest that the average running time is $O(|E|)$, and indeed this is true on random graphs, but it is possible to construct sparse graphs where SPFA runs in time $\Omega(|V| \cdot |E|)$ like the usual Bellman-Ford algorithm.

```
void spfa(int s) {
  list<int> q;

  memset(pre, -1, sizeof pre);
  memset(vis, 0, sizeof vis);
  fill(dist, dist + V, INF);

  dist[s] = 0;
  q.push_back(s);
  while (!q.empty()) {
    int v = q.front();
    q.pop_front();
    vis[v] = false;
    for (auto e : adj[v]) if (e.cap != e.f) {
      int u = e.t;
      T d = dist[v] + e.cost;
      if (d < dist[u]) {
        dist[u] = d, pre[u] = e.rev;
        if (!vis[u]) {
          if (q.size() && d < dist[q.front()]) q.push_front(u);
          else q.push_back(u);
          vis[u] = true;
        }
      }
    }
  }
}
```

2.8.8 Maximum Matching in General Graphs

Time Complexity: $O(NM \log N)$

[19]

```cpp
struct BlossomAlgorithm {
  int n;
  vector<vector<int>> adj;
  BlossomAlgorithm(int n) : n(n), adj(n){};
  void addEdge(int u, int v) {
      adj[u].push_back(v);
      adj[v].push_back(u);
  }

  vector<int> mate;
  int maximumMatching() {
      mate.assign(n + 1, n);
      vector<int> first(n + 1, n), que(n);
      vector<pair<int, int>> label
        (n + 1, make_pair(-1, -1));
      int head = 0, tail = 0;
      function<void(int, int)> rematch = [&](int v, int
           w) {
          int t = mate[v];
          mate[v] = w;
          if (mate[t] != v) return;
          if (label[v].snd == -1) {
              mate[t] = label[v].fst;
              rematch(mate[t], t);
          } else {
              int x, y;
              tie(x, y) = label[v];
              rematch(x, y);
              rematch(y, x);
          }
      };
      auto relabel = [&](int x, int y) {
          function<int(int)> findFirst = [&](int u) {
              return label[first[u]].fst < 0
                ? first[u]
                : first[u] = findFirst(first[u]);
          };
          int r = findFirst(x), s = findFirst(y);
          if (r == s) return;
          auto h = make_pair(~x, y);
          label[r] = label[s] = h;
          int join;
          while (1) {
              if (s != n) swap(r, s);
              r = findFirst(label[mate[r]].fst);
              if (label[r] == h) {
                  join = r;
                  break;
              } else {
                  label[r] = h;
              }
          }
          for (int v : {first[x], first[y]}) {
              for (; v != join; v =
                     first[label[mate[v]].fst]) {
                  label[v] = make_pair(x, y);
                  first[v] = join;
                  que[tail++] = v;
              }
          }
      };
      auto augment = [&](int u) {
          label[u] = make_pair(n, -1);
          first[u] = n;
          head = tail = 0;
          for (que[tail++] = u; head < tail;) {
              int x = que[head++];
              for (int y : adj[x]) {
                  if (mate[y] == n && y != u) {
                      mate[y] = x;
                      rematch(x, y);
                      return true;
                  } else if (label[y].fst >= 0) {
                      relabel(x, y);
                  } else if (label[mate[y]].fst == -1) {
                      label[mate[y]].fst = x;
                      first[mate[y]] = y;
                      que[tail++] = mate[y];
                  }
              }
          }
          return false;
      };
      int matching = 0;
      for (int u = 0; u < n; ++u) {
          if (mate[u] < n || !augment(u)) continue;
          ++matching;
          for (int i = 0; i < tail; ++i)
              label[que[i]] = label[mate[que[i]]] =
                    make_pair(-1, -1);
          label[n] = make_pair(-1, -1);
      }
      return matching;
  }
};
```

Problem 4 - *Ada and Bloom (SPOJ)* [16]

2.8.9 Stable Marriage Problem

```
vector<int> stable_matching(vector<vector<int>> prefer_m, vector<vector<int>> prefer_w) {
  int n = prefer_m.size();
  vector<int> pair_m(n, -1);
  vector<int> pair_w(n, -1);
  vector<int> p(n);
  for (int i = 0; i < n; i++) {
    while (pair_m[i] < 0) {
      int w = prefer_m[i][p[i]++];
      int m = pair_w[w];
      if (m == -1) {
        pair_m[i] = w;
        pair_w[w] = i;
      } else if (prefer_w[w][i] < prefer_w[w][m]) {
        pair_m[m] = -1;
        pair_m[i] = w;
        pair_w[w] = i;
        i = m;
      }
    }
  }
  return pair_m;
}

int main() {
  vector<vector<int>> prefer_m{{0, 1, 2}, {0, 2, 1}, {1, 0, 2}};
  vector<vector<int>> prefer_w{{0, 1, 2}, {2, 0, 1}, {2, 1, 0}};

  vector<int> matching = stable_matching(prefer_m, prefer_w);
  for (int x : matching) cout << x << " ";
}
```

2.8.10 Stable Roommate Problem

Not verified ... Implementation: Mitko Nikov

```
struct StableRoommateProblem {
    // N lists of N - 1 preferences
    vector<vector<int>> p;
    vector<int> proposed, accepted;

    // This is not guaranteed to be O(N^2)
    // To achieve O(N^2), the preference lists have to be actual lists
    // It's O(N^3) at max... But in practice it would be a lot faster.
    bool solve() {
        int N = p.size();
        proposed.resize(N, -1);
        accepted.resize(N, -1);

        auto wouldBreak = [&](int me, int pref) {
            int he = accepted[pref];
            assert(he != -1);
            // am I better than him?
            int he_index = find(p[pref].begin(), p[pref].end(), he) - p[pref].begin();
            int me_index = find(p[pref].begin(), p[pref].end(), me) - p[pref].begin();
            return me_index < he_index;
        };

        auto reject = [&](int me, int pref) {
            auto me_iter = find(p[pref].begin(), p[pref].end(), me);
            if (me_iter != p[pref].end()) p[pref].erase(me_iter);

            auto pref_iter = find(p[me].begin(), p[me].end(), pref);
            if (pref_iter != p[me].end()) p[me].erase(pref_iter);
        };
```

```
        // Phase 1
        vector<int> q(N); int id = 0;
        iota(q.begin(), q.end(), 0);
        while (id < q.size()) {
            int me = q[id];
            if (p[me].size() == 0) break;
            int preferred = p[me][0];

            if (accepted[preferred] == -1) { // The preferred is free
                proposed[me] = preferred;
                accepted[preferred] = me;
                id++;
            } else if (wouldBreak(me, preferred)) {
                // The preferred is willing to break up with his accepted
                proposed[me] = preferred;
                int oldAC = accepted[preferred];
                accepted[preferred] = me;
                reject(preferred, oldAC);
                q[id] = oldAC;
            } else {
                reject(me, preferred);
            }
        }

        auto index = [&](int me, int who) {
            auto it = find(p[me].begin(), p[me].end(), who);
            if (it == p[me].end()) return -1;
            return (int)(it - p[me].begin());
        };

        // Phase 2
        for (int i = 0; i < N; i++) {
            int idAC = index(i, accepted[i]);
            if (idAC == -1) continue;
            vector<int> to_remove(p[i].begin() + idAC + 1, p[i].end());
            for (auto j: to_remove) {
                reject(i, j);
            }
        }
        auto rotation = [&](int me) {
            vector<int> P, Q;
            map<int, int> first;
            while (true) {
                if (first.count(me)) { // cycle!
                    P.push_back(me);
                    for (int i = first[me] + 1; i < P.size(); i++) {
                        reject(P[i], Q[i-1]);
                    }
                    break;
                }
                P.push_back(me);
                Q.push_back(p[me][1]);
                first[me] = P.size() - 1;
                me = p[p[me][1]].back();
            }
            return true;
        };
        auto check = [&]() { // Check if there are valid preferences
            bool ok = true;
            for (int i = 0; i < N; i++) ok &= !p[i].empty();
            return ok;
        };
        // Phase 3
        for (int me = 0; me < N; me++) {
            if (p[me].size() == 1) continue;
            if (!rotation(me)) return false;
            if (!check()) return false;
            me = -1; // reset from the start
        }
      return true;
    }
};
```

```
// INPUT:          OUTPUT:
// 6               YES
// C D B F E       0 5
// F E D A C       1 3
// B D E A F       2 4
// E B C F A       3 1
// C A B D F       4 2
// E A C D B       5 0

int main() {
    int N;
    cin >> N;
    StableRoommateProblem SRP;
    SRP.p.resize(N, vector<int>(N - 1));
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N - 1; j++) {
            char ch; cin >> ch;
            SRP.p[i][j] = ch - 'A';
        }
    }
    auto ok = SRP.solve();
    cout << (ok ? "YES" : "NO") << endl;
    for (int i = 0; ok && i < N; i++) {
        cout << i << " " << SRP.p[i][0] << endl;
    }
    return 0;
}
```

2.9 Global Minimum Cut

A global minimum cut of an undirected graph is a cut of minimum size. The term global here is meant to connote that any cut of the graph is allowed - there is no source or sink. Thus the global min-cut is a natural “robustness” parameter. It is the smallest number of edges whose deletion disconnects the graph.

Time Complexity: $O(N^3)$

Implementation: Simon Lindholm

```
pair<int, vi> globalMinCut(vector<vi> mat) {
   pair<int, vi> best = {INT_MAX, {}};
   int n = sz(mat);
   vector<vi> co(n);
   rep(i,0,n) co[i] = {i};
   rep(ph,1,n) {
      vi w = mat[0];
      size_t s = 0, t = 0;
      rep(it,0,n-ph) { // O(V^2) -> O(E log V) with prio. queue
         w[t] = INT_MIN;
         s = t, t = max_element(all(w)) - w.begin();
         rep(i,0,n) w[i] += mat[t][i];
      }
      best = min(best, {w[t] - mat[t][t], co[t]});
      co[s].insert(co[s].end(), all(co[t]));
      rep(i,0,n) mat[s][i] += mat[t][i];
      rep(i,0,n) mat[i][s] = mat[s][i];
      mat[0][t] = INT_MIN;
   }
   return best;
}
```

2.10 Strongly Connected Components

Time Complexity: $O(N + M)$

```cpp
struct TarjanSCC {
  int N;
  vector<vector<int>> adj;
  vector<int> scc, in, low;
  stack<int> s;
  vector<bool> inStack;
  int scc_num = 0, timer = 0;

  TarjanSCC(int N) {
    this->N = N;
    scc.resize(N, -1);
    in.resize(N, -1);
    low.resize(N);
    inStack.resize(N, false);
  }

  // In the scc vector are
  // the IDs of the components for each node
  void run() {
    for (int i = 0; i < N; i++)
      if (scc[i] == -1) dfs(i);
  }
```

```cpp
  void dfs(int n) {
    low[n] = in[n] = timer++;
    s.push(n);
    inStack[n] = true;
    for (int m : adj[n]) {
      if (in[m] == -1) {
        dfs(m);
        low[n] = min(low[n], low[m]);
      } else if (inStack[m]) {
        low[n] = min(low[n], in[m]);
      }
    }

    if (low[n] == in[n]) {
      while (true) {
        int u = s.top();
        s.pop();
        scc[u] = scc_num;
        inStack[u] = false;
        if (u == n) break;
      }
      ++scc_num;
    }
  }
};
```

2.11 Dynamic Connectivity

The Dynamic Connectivity Problem [15] revolves around the idea that we will have a dynamically changing graph and we are asked at some points of time, whether some nodes are connected, or about the number of connected components and similar queries. This problem is really difficult, but we will offer a very fast offline solution for general graphs. Furthermore, in the section about trees, we will talk about solving this problem on trees, but with a very fast amortized online solution.

The idea behind solving such a problem is realizing that we can build a special segment tree type of data-structure on the queries as our leafs. Each query defines a unit of time. We can imagine that each edge will exist/live from time T_i to time T_j. So each edge contributes only in this range. When traversing the tree, we can book-keep the components i.e. their connectivity in a DSU data-structure. One thing we need to be able to do, is to rollback the changes on the DSU when we backtrack with the DFS or when we get to the point where we need to remove an edge.

2.12 Dynamic Reachability for DAG

It is a data structure that admits the following operations:

- `add_edge(s, t)` ... insert edge (s, t) to the network if it does not make a cycle
- `is_reachable(s, t)` ... return true iff there is a path there is a path from s to t

We maintain reachability trees $T(u)$ for all u in V. Then $is_reachable(s, t)$ is solved by checking $t \in T(u)$. For $add_edge(s, t)$, if $is_reachable(s, t)$ or $is_reachable(t, s)$ then no update is performed. Otherwise, we meld $T(s)$ and $T(t)$.

Time Complexity (update): Amortized $O(N)$
Time Complexity (query): $O(1)$

```
struct dag_reachability {
  int n;
  vector<vector<int>> parent;
  vector<vector<vector<int>>> child;
  dag_reachability(int n)
      : n(n),
        parent(n, vector<int>(n, -1)),
        child(n, vector<vector<int>>(n)) {}
  bool is_reachable(int src, int dst) {
      return src == dst || parent[src][dst] >= 0;
  }
  bool add_edge(int src, int dst) {
      if (is_reachable(dst, src)) return false; // break DAG condition
      if (is_reachable(src, dst)) return true; // no-modification performed
      for (int p = 0; p < n; ++p)
          if (is_reachable(p, src) && !is_reachable(p, dst))
              meld(p, dst, src, dst);
      return true;
  }
  void meld(int root, int sub, int u, int v) {
      parent[root][v] = u;
      child[root][u].push_back(v);
      for (int c : child[sub][v])
          if (!is_reachable(root, c)) meld(root, sub, v, c);
  }
};
```

2.13 Minimum Cost Arborescence

Time Complexity: $O(NM)$

Let $G = (V, E)$ be a weighted directed graph. For a vertex r, an edge-set T is called r-arborescense if

- T is a spanning tree (with forgetting directions),
- for each u in V, $indeg_T(u) \leq 1$, $indeg_T(r) = 0$.

The program finds the minimum weight of r-arborescence.

Algorithm: Chu-Liu/Edmonds' recursive shrinking. At first, it finds a minimum incomming edge for each v in V. Then, if it forms a arborescence, it is a solution, and otherwise, it contracts a cycle and iterates the procedure.

```
const int INF = 1e9 + 1000;
struct graph {
    int n;
    graph(int n) : n(n) {}
    struct edge {
        int src, dst;
        int weight;
    };
    vector<edge> edges;
    void add_edge(int u, int v, int w) { edges.push_back({u, v, w}); }
    int arborescence(int r) {
        int N = n;
        for (int res = 0;;) {
            vector<edge> in(N, {-1, -1, (int)INF});
            vector<int> C(N, -1);
            for (auto e : edges) // cheapest comming edges
                if (in[e.dst].weight > e.weight) in[e.dst] = e;
            in[r] = {r, r, 0};

            for (int u = 0; u < N; ++u) { // no comming edge ==> no aborescense
                if (in[u].src < 0) return -1;
                res += in[u].weight;
            }
            vector<int> mark(N, -1); // contract cycles
            int index = 0;
            for (int i = 0; i < N; ++i) {
                if (mark[i] != -1) continue;
                int u = i;
                while (mark[u] == -1) {
                    mark[u] = i;
                    u = in[u].src;
                }
                if (mark[u] != i || u == r) continue;
                for (int v = in[u].src; u != v; v = in[v].src) C[v] = index;
                C[u] = index++;
            }
            if (index == 0) return res; // found arborescence
            for (int i = 0; i < N; ++i) // contract
                if (C[i] == -1) C[i] = index++;

            vector<edge> next;
            for (auto &e : edges)
                if (C[e.src] != C[e.dst] && C[e.dst] != C[r])
                    next.push_back(
                        {C[e.src], C[e.dst], e.weight - in[e.dst].weight});
            edges.swap(next);
            N = index;
            r = C[r];
        }
    }
};
```

2.14 Minimum Mean Cycle

Time Complexity: $O(NM)$
Space Complexity: $O(N^2)$

Given a directed graph $G = (V, E)$ with edge weight $w(e), \forall e \in E$.
Find a minimum mean cycle C, i.e., $min \frac{w(C)}{|C|}\}$.

Karp's Algorithm [22] starts by fixing some vertex s. Using dynamic programming, we can compute the shortest path from s to every possible v, with "exactly" k edges. We write $d(s, u; k)$ for this value. Then, we can show that

$$\min_{u \in V} \max_{k \in [|V|]} \frac{d(s, u; n) - d(s, u; k)}{n - k}$$

is the length of minimum mean cycle.

Proof 1 *Note that $d(s, u; n)$ consists of a cycle and a path. Subtract the path from s to u, we obtain a length of cycle.*

Remark 3 *For an undirected graph, the minimum mean cycle problem can be solved by b-matching/T-join. See Korte and Vygen, Ch. 12.*

```
struct graph {
  typedef int weight_type;
  const weight_type INF = 99999999;
  struct edge {
    int src, dst;
    weight_type weight;
  };
  int n;
  vector<vector<edge>> adj;
  graph(int n) : n(n), adj(n) { }
  void add_edge(int src, int dst, weight_type weight) {
    adj[src].push_back({src, dst, weight});
  }
```

```
typedef pair<weight_type, int> fraction;
fraction min_mean_cycle() {
  vector<vector<weight_type>> dist(n+1,
        vector<weight_type>(n));
  vector<vector<int>> prev(n+1, vector<int>(n, -1));
  fill(all(prev[0]), 0);

  for (int k = 0; k < n; ++k) {
    for (int u = 0; u < n; ++u) {
      if (prev[k][u] < 0) continue;
      for (auto e: adj[u]) {
        if (prev[k+1][e.dst] < 0 ||
            dist[k+1][e.dst] > dist[k][e.src] +
                  e.weight) {
          dist[k+1][e.dst] = dist[k][e.src] + e.weight;
          prev[k+1][e.dst] = e.src;
        }
      }
    }
  }
  int v = -1;
  fraction opt = {1, 0}; // +infty
  for (int u = 0; u < n; ++u) {
    fraction f = {-1, 0}; // -infty
    for (int k = n-1; k >= 0; --k) {
      if (prev[k][u] < 0) continue;
      fraction g = {dist[n][u] - dist[k][u], n - k};
      if (f.fst * g.snd < f.snd * g.fst) f = g;
    }
    if (opt.fst * f.snd > f.fst * opt.snd) { opt = f;
          v = u; }
  }
  if (v >= 0) { // found a loop
    vector<int> p; // path
    for (int k = n; p.size() < 2 || p[0] != p.back();
          v = prev[k--][v])
      p.push_back(v);
    reverse(all(p));
  }
  return opt;
 }
};
```

2.15 Max Cut

2.16 Max Clique

```
#define vb vector<bitset<101>>
struct Maxclique {
  double limit = 0.025, pk = 0;
  struct Vertex { int i, d = 0; };
  typedef vector<Vertex> vv;
  vb e;
  vv V;
  vector<vector<int>> C;
  vector<int> qmax, q, S, old;

  void init(vv& r) {
     for (auto& v : r) v.d = 0;
     for (auto& v : r) for (auto j : r) v.d += e[v.i][j.i];
     sort(all(r), [](auto a, auto b) { return a.d > b.d; });

    // maximum_color(vertex)
     int mxD = r[0].d;
     for (int i = 0; i < r.size(); i++) r[i].d = min(i, mxD) + 1;
  }

  void expand(vv& R, int lev = 1) {
     S[lev] += S[lev - 1] - old[lev];
     old[lev] = S[lev - 1];
     while (sz(R)) {
        if (sz(q) + R.back().d <= sz(qmax)) return;
        q.push_back(R.back().i);
        vv T;
        for(auto v:R) if (e[R.back().i][v.i]) T.push_back({v.i});
        if (sz(T)) {
           if (S[lev]++ / ++pk < limit) init(T);
           int j = 0, mxk = 1, mnk = max(sz(qmax) - sz(q) + 1, 1);
           C[1].clear(), C[2].clear();
           for (auto v : T) {
              int k = 1;
              auto f = [&](int i) { return e[v.i][i]; };
              while (any_of(all(C[k]), f)) k++;
              if (k > mxk) mxk = k, C[mxk + 1].clear();
              if (k < mnk) T[j++].i = v.i;
              C[k].push_back(v.i);
           }
           if (j > 0) T[j - 1].d = 0;
           for (int k = mnk; k < mxk + 1; ++k) for (int i : C[k])
              T[j].i = i, T[j++].d = k;
           expand(T, lev + 1);
        } else if (sz(q) > sz(qmax)) qmax = q;
        q.pop_back(), R.pop_back();
     }
  }

  vector<int> maxClique() { init(V), expand(V); return qmax; }

  Maxclique(vb conn) : e(conn), C(sz(e)+1), S(sz(C)), old(S) {
     for (int i = 0; i < e.size(); i++) V.push_back({i});
  }
};
```

2.17 Chromatic Number

A vertex coloring is an assignment of colors to the vertices such that no adjacent vertices have a same color. The smallest number of colors for a vertex coloring is called the chromatic number. Computing the chromatic number is NP-hard.

We can compute the chromatic number by the inclusion-exlusion principle. The complexity is $O(poly(n)2^n)$. The following implementation runs in $O(n2^n)$ but is a Monte-Carlo algorithm since it takes modulos to avoid multiprecision numbers. [17]

Complexity: $O(n2^n)$

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

#define fst first
#define snd second
#define all(c) ((c).begin()), ((c).end())

struct Graph {
  int n;
  vector<vector<int>> adj;
  Graph(int n) : n(n), adj(n) { }
  void addEdge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
  }
};

int chromaticNumber(Graph g) {
  const int N = 1 << g.n;
  vector<int> nbh(g.n);
  for (int u = 0; u < g.n; ++u)
    for (int v: g.adj[u])
      nbh[u] |= (1 << v);

  int ans = g.n;
  for (int d: {7}) { // ,11,21,33,87,93}) {
    long long mod = 1e9 + d;
    vector<long long> ind(N), aux(N, 1);
    ind[0] = 1;
    for (int S = 1; S < N; ++S) {
      int u = __builtin_ctz(S);
      ind[S] = ind[S^(1<<u)] + ind[(S^(1<<u))&~nbh[u]];
    }
    for (int k = 1; k < ans; ++k) {
      long long chi = 0;
      for (int i = 0; i < N; ++i) {
        int S = i ^ (i >> 1); // gray-code
        aux[S] = (aux[S] * ind[S]) % mod;
        chi += (i & 1) ? aux[S] : -aux[S];
      }
      if (chi % mod) ans = k;
    }
  }
  return ans;
}

int main() {
  int n = 6; Graph g(n);
  g.addEdge(0,1); g.addEdge(1,2);
  g.addEdge(2,3); g.addEdge(0,2);
  g.addEdge(3,4); g.addEdge(4,5);
  g.addEdge(5,0);
  cout << chromaticNumber(g) << endl;
}
```

3 Trees

3.1 Least Common Ancestor

```
struct LCA {
    int n, l;
    vector<vector<int>> adj;

    int timer;
    vector<int> tin, tout;
    vector<vector<int>> up;
    vector<int> depth;

    LCA(vector<vector<int>> adj, int root) {
        this->n = adj.size();
        this->adj = adj;
        tin.resize(n);
        tout.resize(n);
        depth.resize(n);
        timer = 0;
        l = ceil(log2(n));
        up.assign(n, vector<int>(l + 1));
        dfs(root, root);
    }

    ~LCA() {
        tin.clear();
        tout.clear();
        up.clear();
        adj.clear();
    }

    void dfs(int v, int p, int d = 0) {
        tin[v] = ++timer;
        up[v][0] = p;
        depth[v] = d;
        for (int i = 1; i <= l; ++i)
            up[v][i] = up[up[v][i-1]][i-1];

        for (int u : adj[v]) {
            if (u != p)
                dfs(u, v, d + 1);
        }

        tout[v] = ++timer;
    }

    // Is u an ancestor of v
    bool is_ancestor(int u, int v) {
        return tin[u] <= tin[v] && tout[u] >= tout[v];
    }

    int lca(int u, int v) {
        if (is_ancestor(u, v))
            return u;
        if (is_ancestor(v, u))
            return v;
        for (int i = l; i >= 0; --i) {
            if (!is_ancestor(up[u][i], v))
                u = up[u][i];
        }
        return up[u][0];
    }

    // Jump from u, k steps up
    int jump(int u, int k) {
        for (int i = l; i >= 0; i--) {
            if ((1 << i) & k) {
                u = up[u][i];
            }
        }
        return u;
    }

    bool on_path2(int down, int up, int x) {
        // up <= x <= down
        if (is_ancestor(up, x) && is_ancestor(x, down))
            return true;
        return false;
    }

    // Is x on the path from u to v
    bool on_path(int u, int v, int x) {
        int lc = lca(u, v);
        return (on_path2(u, lc, x) || on_path2(v, lc, x));
    }

    // Distance from u to v given their lca
    int dist(int u, int v, int l) {
        return depth[u] + depth[v] - 2 * depth[l];
    }

    // k-th node on the path from u to v (including)
    int kth(int u, int v, int k) {
        int l = lca(u, v);
        if (k > dist(u, v, l)) return -1;
        if (dist(u, l, l) >= k) return jump(u, k);
        return jump(v, dist(u, v, l) - k);
    }
};
```

3.2 Heavy-Light Decomposition

```cpp
/**
 * Author: Benjamin Qi, Oleksandr Kulkov, chilli
 * Date: 2020-01-12
 * License: CC0
 * Time: O((\log N)^2)
 */
#pragma once
#include "../data-structures/LazySegmentTree.h"

template <bool VALS_EDGES> struct HLD {
  int N, tim = 0;
  vector<vi> adj;
  vi par, siz, depth, rt, pos;
  Node *tree;
  HLD(vector<vi> adj_)
    : N(sz(adj_)), adj(adj_), par(N, -1),
    siz(N, 1), depth(N),
      rt(N), pos(N), tree(new Node(0, N))
    {
      dfsSz(0);
      dfsHld(0);
    }
  void dfsSz(int v) {
    if (par[v] != -1) {
      adj[v].erase(find(all(adj[v]), par[v]));
    }
    for (int& u : adj[v]) {
      par[u] = v, depth[u] = depth[v] + 1;
      dfsSz(u);
      siz[v] += siz[u];
      if (siz[u] > siz[adj[v][0]]) swap(u, adj[v][0]);
    }
  }
  void dfsHld(int v) {
    pos[v] = tim++;
    for (int u : adj[v]) {
      rt[u] = (u == adj[v][0] ? rt[v] : u);
      dfsHld(u);
    }
  }
  void process(int u, int v, function<void(int, int)> op) {
    for (; rt[u] != rt[v]; v = par[rt[v]]) {
      if (depth[rt[u]] > depth[rt[v]]) swap(u, v);
      op(pos[rt[v]], pos[v] + 1);
    }
    if (depth[u] > depth[v]) swap(u, v);
    op(pos[u] + VALS_EDGES, pos[v] + 1);
  }
  void modifyPath(int u, int v, int val) {
    process(u, v, [&](int l, int r) {
      tree->add(l, r, val);
    });
  }
  int queryPath(int u, int v) { // Modify depending on problem
    int res = -1e9;
    process(u, v, [&](int l, int r) {
      res = max(res, tree->query(l, r));
    });
    return res;
  }
  int querySubtree(int v) { // modifySubtree is similar
    return tree->query(pos[v] + VALS_EDGES, pos[v] + siz[v]);
  }
};
```

3.3 Centroid Decomposition

```cpp
struct CentroidDecomposition {
    vector<vector<int>> G;
    vector<map<int, int>> dis;
    vector<int> sz, pa, ans;
    vector<bool> checked;


    CentroidDecomposition(int N) {
        N += 10;
        G.resize(N);
        dis.resize(N);
        sz.resize(N);
        pa.resize(N);
        ans.resize(N);
        checked.resize(N);
        for (int i = 0; i < N; ++i) {
            G[i].clear();
            dis[i].clear();
            ans[i] = inf;
        }
    }


    // PAY ATTENTION: 1 INDEXED
    void addEdge(int u, int v) {
        G[u].push_back(v);
        G[v].push_back(u);
    }
    int dfs(int u, int p) {
        sz[u] = 1;
        for (auto v : G[u]) {
            if (v == p || checked[v]) continue;
            sz[u] += dfs(v, u);
        }
        return sz[u];
    }
    int centroid(int u, int p, int n) {
        for (auto v : G[u]) {
            if (v == p || checked[v]) continue;
            if (sz[v] > n / 2) return centroid(v, u, n);
        }
        return u;
    }
    void dfs2(int u, int p, int c, int d) {
        dis[c][u] = d; // distance from centroid to me
        for (auto v : G[u]) {
            if (v == p || checked[v]) continue;
            dfs2(v, u, c, d + 1);
        }
    }
    void build(int u, int p) {
        int n = dfs(u, p);
```

```
    int c = centroid(u, p, n);
    if (p == -1) p = c;
    pa[c] = p;
    dfs2(c, p, c, 0);
    checked[c] = true;

    for (auto v : G[c]) {
        if (v == p || checked[v]) continue;
        build(v, c);
    }
}
void modify(int u) {
    for (int v = u; v != 0; v = pa[v]) {
        ans[v] = min(ans[v], dis[v][u]);
    }
}
int query(int u) {
    int mn = inf;
    for (int v = u; v != 0; v = pa[v]) {
        mn = min(mn, ans[v] + dis[v][u]);
    }
    return mn;
}
};
```

3.4 Kruskal Reconstruction Trees

3.5 Minimum Diameter Spanning Tree

```
//
// Minimum Diameter Spanning Tree
//
// Description:
//   Find a minimum diameter spanning tree
//   (<=> absolute center of a graph)
//
// Algorithm:
//   Based on Kariv-Hakimi or CunninghameGreen,
//   with Halpern bound.
//
// Complexity:
//   O(APSP + n^2 log n + n m) time,
//   O(n^2) space.
#include <iostream>
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;

#define ALL(c) c.begin(), c.end()
#define FOR(i,c) for(typeof(c.begin())i=c.begin();i!=c.end();++i)
#define REP(i,n) for(int i=0;i<n;++i)

typedef pair<int,int> PII;
#define fst first
#define snd second

typedef int Weight;
const Weight INF = 1 << 28;
struct Graph {
  struct Edge {
    int s, t;
    Weight w;
    Edge(int s, int t, Weight w) : s(s), t(t), w(w) { }
  };
  int n;
  vector<Edge> edge; // edge list
  vector< vector<Weight> > A; // adj matrix

  Graph(int n) : n(n), A(n, vector<int>(n, INF)) {
    REP(u, n) A[u][u] = 0;
  }
  void addEdge(int s, int t, Weight w) {
    w *= 2; // because of half integrality
    if (w >= A[s][t]) return;
    A[s][t] = A[t][s] = w;
    edge.push_back(Edge(s, t, w));
  }

  vector<int> R; // radius from the absolute center
  vector<bool> visited;
  void traverse(int u, int sh = 0) { // explicit construction of sp-tree
    visited[u] = true;
    REP(v,n) if (R[v] == R[u] + A[u][v]) {
      if (!visited[v]) {
        printf("%d %d\n", u+1, v+1);
        traverse(v, sh+2);
      }
    }
  }
  vector< vector<Weight> > d;
  vector< vector<int> > L;
  void farthestOrdering () {
    L.assign(n, vector<int>(n)); d = A;
    REP(z,n) REP(x,n) REP(y,n)
      d[x][y] = min(d[x][y], d[x][z] + d[z][y]);
    REP(x,n) {
      vector<PII> aux;
      REP(y,n) aux.push_back(PII(-d[x][y], y));
      sort(ALL(aux));
      REP(k,n) L[x][k] = aux[k].snd; // farthest ordering
    }
  }
  Weight minimumDiameterSpanningTree() {
    farthestOrdering(); // preprocessing
    Weight h, D = INF;
    Edge &e = edge[0];
    FOR(it, edge) {
      int s = it->s, t = it->t; // for simplicity
      Weight w = it->w;
      if (d[s][L[t][0]] + d[t][L[s][0]] + w > 2*D) continue; // Halpern bound
      if (L[s][0] == L[t][0]) continue; // no-coincide condition

      vector<int> &v = L[s];
      int k = 0; // last active constraint
      Weight x = 0, y = min(d[s][v[0]], d[t][v[0]] + w), xi, yi;
      for (int i = 1; i < n; ++i) {
        if (d[t][v[k]] < d[t][v[i]]) {
          xi = (d[t][v[k]] - d[s][v[i]] + w) / 2;
          yi = xi + d[s][v[i]];
          if (yi < y) { y = yi; x = xi; }
          k = i;
        }
      }
      yi = min(d[s][v[k]]+w, d[t][v[k]]);
      if (yi < y) { y = yi; x = 1; }
      if (y < D) { D = y; h = x; e = *it; }
    }
    printf("%d\n", D);
    R.resize(n); // explicit reconstruction by DFS
    REP(u, n) R[u] = min(d[e.s][u]+h, d[e.t][u]+e.w-h);
    visited.assign(n, false);
    if (h > 0) traverse(e.t);
    if (e.w > h) {
      traverse(e.s);
      if (h > 0) printf("%d %d\n", e.s+1, e.t+1);
    }
  }
};

int main () {
  int n, m; scanf("%d%d", &n, &m);
  Graph G(n);

  while (m--) {
    int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    --a, --b;
    G.addEdge(a, b, c);
  }
  G.minimumDiameterSpanningTree();
}
```

3.6 Gomory Hu Tree

3.7 Dominator Tree (Lengauer-Tarjan)

Let $G = (V, E)$ be a directed graph and fix r in V. v is a dominator of u if all paths from r to u go through v. The set of dominators of u forms a total order, and the closest dominator is called the immediate dominator. The set $\{(u, v) : v \text{ is the immediate dominator } \}$ forms a tree, which is called the dominator tree. [23]

Time Complexity: $O(MlogN)$

```
struct edge { int src, dst; };
struct graph {
  int n;
  vector<vector<edge>> adj, rdj;
  graph(int n = 0) : n(0) { }
  void add_edge(int src, int dst) {
    n = max(n, max(src, dst)+1);
    adj.resize(n); rdj.resize(n);
    adj[src].push_back({src, dst});
    rdj[dst].push_back({dst, src});
  }
  vector<int> rank, semi, low, anc;
  int eval(int v) {
    if (anc[v] < n && anc[anc[v]] < n) {
      int x = eval(anc[v]);
      if (rank[semi[low[v]]] > rank[semi[x]]) low[v] = x;
      anc[v] = anc[anc[v]];
    }
    return low[v];
  }
  vector<int> prev, ord;
  void dfs(int u) {
    rank[u] = ord.size();
    ord.push_back(u);
    for (auto e: adj[u]) {
      if (rank[e.dst] < n) continue;
      dfs(e.dst);
      prev[e.dst] = u;
    }
  }
  vector<int> idom; // idom[u] is an immediate dominator of u
  void dominator_tree(int r) {
    idom.assign(n, n); prev = rank = anc = idom;
    semi.resize(n); iota(all(semi), 0); low = semi;
    ord.clear(); dfs(r);

    vector<vector<int>> dom(n);
    for (int i = ord.size()-1; i >= 1; --i) {
      int w = ord[i];
      for (auto e: rdj[w]) {
        int u = eval(e.dst);
        if (rank[semi[w]] > rank[semi[u]]) semi[w] = semi[u];
      }
      dom[semi[w]].push_back(w);
      anc[w] = prev[w];
      for (int v: dom[prev[w]]) {
        int u = eval(v);
        idom[v] = (rank[prev[w]] > rank[semi[u]] ? u : prev[w]);
      }
      dom[prev[w]].clear();
    }
    for (int i = 1; i < ord.size(); ++i) {
      int w = ord[i];
      if (idom[w] != semi[w]) idom[w] = idom[idom[w]];
    }
  }
  vector<int> dominators(int u) {
    vector<int> S;
    for (; u < n; u = idom[u]) S.push_back(u);
    return S;
  }
};
```

4 Algorithms

4.1 Binary and Ternary Search Function

```
template<typename T>
T ternSearch(T a, T b, function<T(T)> f) {
	assert(a <= b);
	while (b - a >= 1e-8) {
		T mid = (b - a) / 3;
		if (f(a + mid) > f(a + mid * 2)) b = a + mid * 2;
		else a = a + mid;
	}
	return a;
}
```

4.2 Index Compression

```
auto comp = a;
sort(comp.begin(), comp.end());
for (int i = 0; i < n; i++) {
    a[i] = lower_bound(comp.begin(), comp.end(), a[i]) - comp.begin() + 1;
}
```

4.3 Fast Eratosthenes Sieve

```
struct Sieve {
  static const int N = 1e5 + 1000;
  int lp[N+1], pr[N+1];
  int counter = 0;

  Sieve() {
    for (int i = 0; i < N; i++) {
      pr[i] = -1;
      lp[i] = 0;
    }

    for (int i = 2; i <= N; ++i) {
      if (lp[i] == 0) {
        lp[i] = i;
        pr[counter++] = i;
      }
      for (int j = 0; j < counter && pr[j] <= lp[i] && i * pr[j] <= N; ++j)
        lp[i * pr[j]] = pr[j];
    }
  }
};
```

4.4 Mo's Algorithm

```
struct Query {
    int id;
    int l, r;

    bool operator<(const Query &rhs) const {
        if (l / MAGIC == rhs.l / MAGIC) return r < rhs.r;
        return (l / MAGIC < rhs.l / MAGIC);
    }
};

vector<int> ans_vec(q);
int cur_l = 0, cur_r = -1;
for (int i = 0; i < q; i++) {
    while (cur_l > queries[i].l) {
        cur_l--;
        add(cur_l);
    }
    while (cur_r < queries[i].r) {
        cur_r++;
        add(cur_r);
    }

    while (cur_l < queries[i].l) {
        remove(cur_l);
        cur_l++;
    }

    while (cur_r > queries[i].r) {
        remove(cur_r);
        cur_r--;
    }

    ans_vec[queries[i].id] = ans;
}
```

4.5 Merge Sort

4.6 K-th Order Statistic

4.7 K-th Shortest Paths

We are given a weighted graph. The k-shortest walks problem seeks k different s-t walks (paths allowing repeated vertices) in the increasing order of the lengths.

If we maintain each walks explicitly, it must costs $O(k^2 m)$ time. To avoid this complexity, we maintain the walks in a compact format. Let us fix a reverse shortest path tree from t. A deviation is an edge that is not on the tree. Any walk is represented by a concatenation of deviations and paths on the tree. We enumerate all possible deviations and use the best-first search to find the k-th solution.

The Eppstein's algorithm maintains the set of deviations by the augmented persistent heaps and emurates the best-first search. Here, we implemented a simplified version of the Eppstein's algorithm, which uses the simple persistent heaps instead of the augmented ones. It increases the space from $O(m + n \log n)$ to $O(m \log n)$. [1]

Complexity: $O(m \log m)$ construction and $O(k \log k)$ for k-th search

```
// Verified:
//   UTPC2013_10 J K-th Cycle
#include <bits/stdc++.h>

using namespace std;

#define fst first
#define snd second
#define all(c) ((c).begin()), ((c).end())

struct Graph {
  int n, m = 0;
  vector<int> head;                // Vertex
  vector<int> src, dst, next, prev; // Edge

  using Weight = long long;
  vector<Weight> weight;
  Graph(int n) : n(n), head(n, -1) { }
  int addEdge(int u, int v, Weight w) {
    next.push_back(head[u]);
    src.push_back(u);
    dst.push_back(v);
    weight.push_back(w);
    return head[u] = m++;
  }
};
constexpr Graph::Weight INF = 1e15;

struct KShortestWalks {
  Graph g;
  vector<Graph::Weight> dist;
  vector<int> tree, order;
  void reverseDijkstra(int t) {
    vector<vector<int>> adj(g.n);
    for (int u = 0; u < g.n; ++u)
      for (int e = g.head[u]; e >= 0; e = g.next[e])
        adj[g.dst[e]].push_back(e);
    dist.assign(g.n, INF);
    tree.assign(g.n, ~g.m);
    using Node = tuple<Graph::Weight,int>;
    priority_queue<Node, vector<Node>, greater<Node>> que;
    que.push(make_tuple(0, t));
    dist[t] = 0;
    while (!que.empty()) {
      int u = get<1>(que.top()); que.pop();
      if (tree[u] >= 0) continue;
      tree[u] = ~tree[u];
      order.push_back(u);
      for (int e: adj[u]) {
        int v = g.src[e];
        if (dist[v] > dist[u] + g.weight[e]) {
          tree[v] = ~e;
          dist[v] = dist[u] + g.weight[e];
          que.push(Node(dist[v], v));
        }
      }
    }
  }
  struct Node { // Persistent Heap (Leftist Heap)
    int e;
    Graph::Weight delta;
    Node *left = 0, *right = 0;
    int rnk = 0;
  } *root = 0;
  static Node *merge(Node *x, Node *y) {
    if (!x) return y;
    if (!y) return x;
    if (x->delta > y->delta) swap(x, y);
    x = new Node(*x);
    x->right = merge(x->right, y);
```

```
    if (!x->left || x->left->rnk < x->rnk) swap(x->left, x->right);
    x->rnk = (x->right ? x->right->rnk : 0) + 1;
    return x;
  }
  vector<Node*> deviation;
  void buildHeap() {
    deviation.resize(g.n);
    for (int u: order) {
      int v = -1;
      for (int e = g.head[u]; e >= 0; e = g.next[e]) {
        if (tree[u] == e) v = g.dst[e];
        else if (dist[g.dst[e]] < INF) {
          auto delta = g.weight[e] - dist[g.src[e]] + dist[g.dst[e]];
          deviation[u] = merge(deviation[u], new Node({e, delta}));
        }
      }
      if (v >= 0) deviation[u] = merge(deviation[u], deviation[v]);
    }
  }
  KShortestWalks(Graph g_, int t) : g(g_) {
    reverseDijkstra(t);
    buildHeap();
  }
  void enumerate(int s, int kth) {
    int k = 0;
    Node *x = deviation[s];
    Graph::Weight len = dist[s];
    ++k;
    using SearchNode = tuple<Node*, Graph::Weight>;
    auto comp = [](SearchNode x, SearchNode y) { return get<1>(x) > get<1>(y); };
    priority_queue<SearchNode, vector<SearchNode>, decltype(comp)> que(comp);
    if (x) que.push(SearchNode(x, len + x->delta));
    while (!que.empty() && k < kth) {
      tie(x, len) = que.top(); que.pop();
      int e = x->e, u = g.src[e], v = g.dst[e];
      cout << len << endl; ++k;
      if (deviation[v]) que.push(SearchNode(deviation[v], len+deviation[v]->delta));
      for (Node *y: {x->left, x->right})
        if (y) que.push(SearchNode(y, len + y->delta-x->delta));
    }
    while (k < kth) { cout << -1 << endl; ++k; }
  }
};

void KSH_test() {
  int n = 4;
  Graph g(n);
  g.addEdge(0, 1, 2);
  g.addEdge(0, 2, 2);
  g.addEdge(1, 3, 4);
  g.addEdge(2, 3, 2);
  g.addEdge(1, 2, 1);
  g.addEdge(2, 1, 1);
  KShortestWalks ksh(g, 3);
  ksh.enumerate(0, 10);
}

// Your task is to write a program that finds the lengths
// of the 1,2,...,K-th shortest paths
// starting from vertex 0 and back to vertex 0, given a graph.
void UTPC2013_10() {
  int n, m, k;
  scanf("%d %d %d", &n, &m, &k);
  Graph g(n);
  for (int i = 0; i < m; ++i) {
    int u, v;
    long long w;
    scanf("%d %d %lld", &u, &v, &w);
    g.addEdge(u, v, w);
  }
  int ending_vertex = 0;
  KShortestWalks ksh(g, ending_vertex);
  ksh.enumerate(0, k+1);
}

int main() {
  UTPC2013_10();
  // KSH_test();
}
```

4.8 Shunting Yard

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

bool delim(char c) {
    return c == ' ';
}

bool is_op(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/';
}

int priority (char op) {
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return -1;
}

void process_op(stack<int>& st, char op) {
    int r = st.top(); st.pop();
    int l = st.top(); st.pop();
    switch (op) {
        case '+': st.push(l + r); break;
        case '-': st.push(l - r); break;
        case '*': st.push(l * r); break;
        case '/': st.push(l / r); break;
    }
}

int evaluate(string& s) {
    stack<int> st;
    stack<char> op;
    for (int i = 0; i < (int)s.size(); i++) {
        if (delim(s[i]))
            continue;

        if (s[i] == '(') {
            op.push('(');
        } else if (s[i] == ')') {
            while (op.top() != '(') {
                process_op(st, op.top());
                op.pop();
            }
            op.pop();
        } else if (is_op(s[i])) {
            char cur_op = s[i];
            while (!op.empty() && priority(op.top()) >= priority(cur_op)) {
                process_op(st, op.top());
                op.pop();
            }
            op.push(cur_op);
        } else {
            int number = 0;
            while (i < (int)s.size() && isalnum(s[i]))
                number = number * 10 + s[i++] - '0';
            --i;
            st.push(number);
        }
    }

    while (!op.empty()) {
        process_op(st, op.top());
        op.pop();
    }
    return st.top();
}

int main() {
    string s; cin >> s;
    cout << evaluate(s) << endl;
    return 0;
}
```

4.9 Nearest Smaller

```
//  Compute nearest smaller values for all i:
//    nearest_smaller[i] = argmax { j : j < i, a[j] < a[i] }.

template <class T>
vector<int> nearest_smallers(const vector<T> &x) {
  vector<int> z;
  for (int i = 0; i < x.size(); ++i) {
    int j = i-1;
    while (j >= 0 && x[j] >= x[i]) j = z[j];
    z.push_back(j);
  }
  return z;
}
```

5 String Algorithms

5.1 Minimum Lexicographical String Rotation

5.2 Manacher Algorithm

5.3 Z Function

5.4 KMP Algorithm

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

vector<int> comp_shifts(string P) {
	int p = P.length();
	vector<int> shifts(p);
	for (int q = 1; q < p; q++) {
		int k = shifts[q - 1];
		while (k > 0 && P[k] != P[q])
			k = shifts[k - 1];
		if (P[k] == P[q])
			k++;
		shifts[q] = k;
	}
	return shifts;
}

int kmp(string P, string T) {
	vector<int> shifts = comp_shifts(P);
	int n = T.length();
	int m = P.length();

	int occurrences = 0;
	int q = 0;
	for (int i = 0; i < n; i++) {
		while (q && P[q] != T[i])
			q = shifts[q - 1];
		if (P[q] == T[i])
			q++;
		if (q == m) {
			occurrences++;
			q = shifts[q - 1];
		}
	}
	return occurrences;
}

int main() {
	string T; cin >> T; // text to search
	int Q; cin >> Q;    // number of queries

	for (int i = 0; i < Q; i++) {
		string P; cin >> P;        // pattern to search for in text T
		cout << kmp(P, T) << endl; // prints number of occurrences of pattern P in text T
	}

	return 0;
}
```

5.5 Offline Aho Corassick

```
// Problem: Given a text string and a dictionary
// of search strings, find all occurences of a search
// string in the text (overlap included).

// Bad solution: Run KMP a lot, giving O(N * |S|)

// Good solution: Aho-Corasick gives O(N + sum|S|) by
// essentially extending KMP, creating a trie with
// "failure" functions. Basically a FSM.

// Note: The automation generated by Aho-Corasick can
// also be used as a first step in many problems. This
// is often how it'll be used (e.g. sparse table) in
// competitions, since Rabin-Karp is easier to code
// for the simple version of the problem.

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

typedef long long LL;

#define MAXN 100013
int N; // size of dictionary
string dict[MAXN];
string text;

#define MAXM 100013
int M; // number of states in the automation
int g[MAXM][26]; // the normal edges in the trie
int f[MAXM]; // failure function
LL out[MAXM]; // output function

int aho_corasick() {
	memset(g, -1, sizeof g);
	memset(out, 0, sizeof out);

	int nodes = 1;

	// build the trie
	for (int i = 0; i < N; i++) {
		string& s = dict[i];
		int cur = 0;

		for (int j = 0; j < s.size(); j++) {
			if (g[cur][s[j] - 'a'] == -1) {
				g[cur][s[j] - 'a'] = nodes++;
			}
			cur = g[cur][s[j] - 'a'];
		}
		out[cur]++;
	}

	for (int ch = 0; ch < 26; ch++) {
		if (g[0][ch] == -1) {
			g[0][ch] = 0;
		}
	}

	// BFS to calculate out and failure functions
	memset(f, -1, sizeof f);
	queue<int> q;
	for (int ch = 0; ch < 26; ch++) {
		if (g[0][ch] != 0) {
			f[g[0][ch]] = 0;
			q.push(g[0][ch]);
		}
	}

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

		for (int ch = 0; ch < 26; ch++) {
			if (g[state][ch] == -1) continue;

			int fail = f[state];
			while (g[fail][ch] == -1) {
				fail = f[fail];
			}

			f[g[state][ch]] = g[fail][ch];
			out[g[state][ch]] += out[g[fail][ch]];

			q.push(g[state][ch]);
		}
	}

	return nodes;
}

LL search() {
	// Using the Aho-Corasick automation, search the text.
	int state = 0;
	LL ret = 0;
	for (char c : text) {
		while (g[state][c - 'a'] == -1) {
			state = f[state];
		}
		state = g[state][c - 'a'];
		ret += out[state];
	}
	// It's that simple!
	return ret;
}

int main() {
```

```cpp
    // make I/O faster
    ios_base::sync_with_stdio(false);
    cin.tie(0);

    freopen("aho_corasick.in", "r", stdin);

    cin >> N;
    for (int i = 0; i < N; i++) {
        cin >> dict[i];
    }
    cin >> text;

    M = aho_corasick();
    LL result = search();

    cout << result << endl;
    return 0;
}
```

5.6 Online Aho Corassick

```cpp
// Educational Round 16 Problem F
// Add a string s to the set D. It is guaranteed that the string s was not added before.
// Delete a string s from the set D. It is guaranteed that the string s is in the set D.
// For the given string s find the number of occurrences of the strings from the set D.
//   If some string p from D has several occurrences in s you should count all of them.
#include <bits/stdc++.h>
using namespace std;

// This version of aho_corasick uses a bitmask of size ALPHABET,
// so it must be modified for ALPHABET > 26.
template<char MIN_CHAR = 'a'>
struct aho_corasick {
    // suff = the index of the node of the longest strict suffix of
    //                the current node that's also in the tree.
    // dict = the index of the node of the longest strict suffix of
    //                the current node that's in the word list.
    // depth = normal trie depth (root is 0). Can be removed to save memory.
    // word_index = the index of the *first* word ending at this node. -1 if none.
    // word_count = the total number of words ending at this node.
    //                    Used in count_total_matches().
    // first_child = the first child of this node
    //                    (children are sequential due to BFS order), -1 if none.
    // child_mask = the bitmask of child keys available from this node.
    //   IMPORTANT:  If ALPHABET > 26, change the type.

    struct node {
        int suff = -1, dict = -1, depth = 0;
        int word_index = -1, word_count = 0;
        int first_child = -1;
        unsigned child_mask = 0;

        int get_child(char c) const {
            int bit = c - MIN_CHAR;

            if ((child_mask >> bit & 1) == 0)
                return -1;

            assert(first_child >= 0);
            return first_child + __builtin_popcount(child_mask & ((1 << bit) - 1));
        }
    };

    vector<node> nodes;
    int W = 0;
    vector<int> word_location;
    vector<int> word_indices_by_depth;
    vector<int> defer;

    aho_corasick(const vector<pair<string, int>> &words = {}) {
        build(words);
    }

    // Builds the adj list based on suffix parents.
    // Often we want to perform DP and/or queries on this tree.
    vector<vector<int>> build_suffix_adj() const {
        vector<vector<int>> adj(nodes.size());

        for (int i = 1; i < int(nodes.size()); i++)
            adj[nodes[i].suff].push_back(i);

        return adj;
    }

    int get_or_add_child(int current, char c) {
        int bit = c - MIN_CHAR;

        if (nodes[current].child_mask >> bit & 1)
            return nodes[current].get_child(c);

        assert(nodes[current].child_mask >> bit == 0);
        int index = int(nodes.size());
        nodes[current].child_mask |= 1 << bit;

        if (nodes[current].first_child < 0)
            nodes[current].first_child = index;

        nodes.emplace_back();
        nodes.back().depth = nodes[current].depth + 1;
        return index;
    }

    // Where in the trie we end up after starting at `location` and adding char `c`.
    // Runs in worst case O(depth) but amortizes to O(1) in most situations.
    int get_suffix_link(int location, char c) const {
        int child;

        while (location >= 0 && (child = nodes[location].get_child(c)) < 0)
            location = nodes[location].suff;

        return location < 0 ? 0 : child;
    }

    void build(const vector<pair<string, int>> &words) {
        nodes = {node()};
        W = int(words.size());
        vector<int> indices(W);
        iota(indices.begin(), indices.end(), 0);
        stable_sort(indices.begin(), indices.end(), [&](int a, int b) {
            return words[a].first < words[b].first;
        });
        word_location.assign(W, 0);
        vector<int> remaining = indices;
        int rem = W;

        for (int depth = 0; rem > 0; depth++) {
            int nrem = 0;

            for (int i = 0; i < rem; i++) {
                int word = remaining[i];
                int &location = word_location[word];

                if (depth >= int(words[word].first.size())) {
                    if (nodes[location].word_index < 0)
                        nodes[location].word_index = word;

                    nodes[location].word_count += words[word].second;
                } else {
                    location = get_or_add_child(location, words[word].first[depth]);
                    remaining[nrem++] = word;
                }
            }

            rem = nrem;
        }

        int max_depth = 0;
        defer.resize(W);

        for (int i = 0; i < W; i++) {
            max_depth = max(max_depth, int(words[i].first.size()));
            defer[i] = nodes[word_location[i]].word_index;
        }

        // Create a list of word indices in decreasing order of depth,
        // in linear time via counting sort.
        word_indices_by_depth.resize(W);
        vector<int> depth_freq(max_depth + 1, 0);

        for (int i = 0; i < W; i++)
            depth_freq[words[i].first.size()]++;

        for (int i = max_depth - 1; i >= 0; i--)
            depth_freq[i] += depth_freq[i + 1];

        for (int i = W - 1; i >= 0; i--)
            word_indices_by_depth[--depth_freq[words[i].first.size()]] = i;

        // Solve suffix parents by traversing in order of depth (BFS order).
        for (int i = 0; i < int(nodes.size()); i++) {
            unsigned child_mask = nodes[i].child_mask;

            while (child_mask != 0) {
                int bit = __builtin_ctz(child_mask);
                char c = char(MIN_CHAR + bit);
                int index = nodes[i].get_child(c);
                child_mask ^= 1 << bit;

                // Find index's suffix parent by traversing suffix parents
                // of i until one of them has a child c.
                int suffix_parent = get_suffix_link(nodes[i].suff, c);
                nodes[index].suff = suffix_parent;
                nodes[index].word_count += nodes[suffix_parent].word_count;
                nodes[index].dict = nodes[suffix_parent].word_index < 0 ?
                    nodes[suffix_parent].dict : suffix_parent;
            }
        }
    }

    // Counts the number of matches of each word in O(text length + num words).
    vector<int> count_matches(const string &text) const {
        vector<int> matches(W, 0);
        int current = 0;

        for (char c : text) {
```

```cpp
            current = get_suffix_link(current, c);
            int dict_node = nodes[current].word_index < 0
                ? nodes[current].dict : current;

            if (dict_node >= 0)
                matches[nodes[dict_node].word_index]++;
        }

        // Iterate in decreasing order of depth.
        for (int word_index : word_indices_by_depth) {
            int location = word_location[word_index];
            int dict_node = nodes[location].dict;

            if (dict_node >= 0)
                matches[nodes[dict_node].word_index] += matches[word_index];
        }

        for (int i = 0; i < W; i++)
            matches[i] = matches[defer[i]];

        return matches;
    }

    // Counts the number of matches over all words at
    // each ending position in `text` in O(text length).
    vector<int> count_matches_by_position(const string &text) const {
        vector<int> matches(text.size());
        int current = 0;

        for (int i = 0; i < int(text.size()); i++) {
            current = get_suffix_link(current, text[i]);
            matches[i] = nodes[current].word_count;
        }

        return matches;
    }

    // Counts the total number of matches of all words
    // within `text` in O(text length).
    int64_t count_total_matches(const string &text) const {
        int64_t matches = 0;
        int current = 0;

        for (char c : text) {
            current = get_suffix_link(current, c);
            matches += nodes[current].word_count;
        }

        return matches;
    }
};

// Enables online insertion of words into Aho-Corasick
// by adding an extra log factor to the runtime.
// The main idea is to have a distinct Aho-Corasick trie
// for each 1-bit of n, where n is the current number of words.
template<char MIN_CHAR = 'a'>
struct online_aho_corasick {
    vector<pair<string, int>> words;
    vector<aho_corasick<MIN_CHAR>> ACs;

    void add_word(const string &word, int delta) {
        words.emplace_back(word, delta);
        int rebuild = __builtin_ctz(int(words.size()));

        for (int i = 0; i < rebuild; i++)
            ACs[i] = aho_corasick<MIN_CHAR>();

        if (int(ACs.size()) <= rebuild)
            ACs.emplace_back();

        int start = int(words.size()) - (1 << rebuild);
        ACs[rebuild].build(vector<pair<string, int>>(words.begin() + start, words.end()));
    }

    vector<int> count_matches(const string &text) const {
        vector<int> matches;

        for (int i = int(ACs.size()) - 1; i >= 0; i--) {
            vector<int> ac_matches = ACs[i].count_matches(text);
            matches.insert(matches.end(), ac_matches.begin(), ac_matches.end());
        }

        return matches;
    }

    // Counts the number of matches over all words at
    // each ending position in `text` in O(text length).
    vector<int> count_matches_by_position(const string &text) const {
        vector<int> matches(text.size(), 0);

        for (int i = int(ACs.size()) - 1; i >= 0; i--) {
            vector<int> ac_matches = ACs[i].count_matches_by_position(text);

            for (int t = 0; t < int(text.size()); t++)
                matches[t] += ac_matches[t];
        }

        return matches;
    }

    int64_t count_total_matches(const string &text) const {
        int64_t matches = 0;

        for (const auto &ac : ACs)
            matches += ac.count_total_matches(text);

        return matches;
    }
};


int main() {
    int Q;
    cin >> Q;
    online_aho_corasick AC;

    for (int q = 0; q < Q; q++) {
        int t;
        string str;
        cin >> t >> str;

        if (t == 1)
            AC.add_word(str, +1);
        else if (t == 2)
            AC.add_word(str, -1);
        else
            cout << AC.count_total_matches(str) << endl;
    }
}
```

6 Data Structures

6.1 Disjoint Set Union

A Disjoint Set Union (DSU) is a data structure capable of storing sets of vertices, merging sets and checking if two elements belong in the same set in almost $O(1)$ time. [7]

The following implementation of a DSU includes path compression, union by size and set sizes.

```
struct dsu {
  vector<int> parent;
  dsu(int n) : parent(n, -1) {}

  int find_set(int a) {
    if (parent[a] < 0) return a;
    return parent[a] = find_set(parent[a]);
  }

  int merge(int a, int b) {
    int x = find_set(a), y = find_set(b);
    if (x == y) return x;
    if (-parent[x] < -parent[y]) swap(x, y);
    parent[x] += parent[y];
    parent[y] = x;
    return x;
  }

  bool are_same(int a, int b) {
    return find_set(a) == find_set(b);
  }

  int size(int a) {
    return -parent[find_set(a)];
  }
};
```

In some DSU applications [7], we need to keep track of the distance from u to its root. If we don't use path compression, the distance is just the number of recursive calls. But this will be inefficient. However, we can store the distance to the root in each node and modify the distances when doing the path compression.

```
// the parent vector has to be
// modified to store { v, 0 } by default
// returns { root, distance }
pair<int, int> find_set(int v) {
  if (v != parent[v].first) {
      int len = parent[v].second;
      parent[v] = find_set(parent[v].first);
      parent[v].second += len;
  }
  return parent[v];
}
```

When extending the data structure to allow roll-backs, we can't use path compression, but we will still get $O(log(n))$ time per query due to the implementation of union by size.

6.1.1 Eval DSU

Implementation of Evaluation-capable DSU.

```
struct EvalDSU {
  vector<int> parent;
  vector<ll> lazy;
  function<ll(int,int)> f;

  EvalDSU(int N, function<ll(int,int)> operation) {
    parent.resize(N);
    lazy.resize(N);
    iota(parent.begin(), parent.end(), 0);
    f = operation;
  }

  // Path compression to get the O(log(N))
  int find_set(int u) {
    if (parent[u] != u) {
      int leader = find_set(parent[u]);
      lazy[u] += lazy[parent[u]];
      parent[u] = leader;
    }
    return parent[u];
  }

  bool same(int u, int v) {
    return (find_set(u) == find_set(v));
  }

  // u - is the parent
  // v - is a leader of some other set
  // Merge in O(log(N))
  void merge(int u, int v) {
    if (same(u, v)) return;
    lazy[v] += f(u, v);
    parent[v] = u;
  }

  // Evaluation of f(u, leader(u)) in O(log(N))
  ll eval(int u) {
    find_set(u);
    return lazy[u];
  }
};
```

6.1.2 Disjoint Set Union with Queue-like Undo

```
struct DSU {
  vector<int> rank, link;
  vector<int> stk, chkp;

  DSU(int n) : rank(2 * n, 0), link(2 * n, -1) {}

  int find(int x) {
    while (link[x] != -1)
      x = link[x];
    return x;
  }
  void unite(int a, int b) {
    a = find(a); b = find(b);
    if (a == b) return;
    if (rank[a] > rank[b]) swap(a, b);
    stk.push_back(a);
    link[a] = b;
    rank[b] += (rank[a] == rank[b]);
  }

  bool Try(int a, int b) {
    if (find(2 * a + 1) == find(2 * b + 1))
      return false;
    return true;
  }

  void Unite(int a, int b) {
    chkp.push_back(stk.size());
    unite(2 * a, 2 * b + 1);
    unite(2 * a + 1, 2 * b);
    assert(find(2 * a) != find(2 * a + 1));
  }

  void Undo() {
    for (int i = chkp.back(); i < (int)stk.size(); ++i)
      link[stk[i]] = -1;
    stk.resize(chkp.back());
    chkp.pop_back();
  }
};

struct Upd {
  int type, a, b;
};
```

```
int main() {
  DSU dsu(n);
  vector<Upd> upds, tmp[2];
  int rem_a = 0;
  auto pop = [&]() {
    if (rem_a == 0) {
      reverse(upds.begin(), upds.end());
      for (int i = 0; i < (int)upds.size(); ++i)
        dsu.Undo();
      for (auto& upd : upds) {
        upd.type = 0;
        dsu.Unite(upd.a, upd.b);
      }
      rem_a = upds.size();
    }
    while (upds.back().type == 1) {
      tmp[1].push_back(upds.back());
      dsu.Undo();
      upds.pop_back();
    }
    int sz = (rem_a & (-rem_a));
    for (int i = 0; i < sz; ++i) {
      assert(upds.back().type == 0);
      tmp[0].push_back(upds.back());
      dsu.Undo();
      upds.pop_back();
    }
    for (int z : {1, 0}) {
      for (; tmp[z].size(); tmp[z].pop_back()) {
        auto upd = tmp[z].back();
        dsu.Unite(upd.a, upd.b);
        upds.push_back(upd);
      }
    }
    assert(upds.back().type == 0);
    upds.pop_back();
    dsu.Undo();
    --rem_a;
  };
  auto push = [&](int a, int b) {
    upds.push_back(Upd{1, a, b});
    dsu.Unite(a, b);
  };

  vector<int> dp(2 * m);
  int lbound = 0;
  for (int i = 0; i < 2 * m; ++i) {
    auto [a, b] = edges[i % m];
    while (!dsu.Try(a, b)) {
      pop();
      ++lbound;
    }
    push(a, b);
    dp[i] = lbound;
  }

  for (int i = 0; i < q; ++i) {
    int a, b; cin >> a >> b; --a; --b;
    if (dp[a + m - 1] <= b + 1) cout << "NO\n";
    else cout << "YES\n";
  }
}
```

6.2 Sparse Table

Good implementation in KACTL.

6.3 Disjoint Sparse Table

Time Complexity (preprocessing): $O(NlogN)$
Time Complexity (query): $O(1)$

```
template <typename T>
struct DisjointSparseTable {
  vector<vector<T>> ys;
  function<T(T, T)> f;
  DisjointSparseTable(vector<T> xs, function<T(T, T)> f_) : f(f_) {
    int n = 1;
    while (n <= xs.size()) n *= 2;
    xs.resize(n);
    ys.push_back(xs);
    for (int h = 1; ; ++h) {
      int range = (2 << h), half = (range /= 2);
      if (range > n) break;
      ys.push_back(xs);
      for (int i = half; i < n; i += range) {
        for (int j = i-2; j >= i-half; --j)
          ys[h][j] = f(ys[h][j], ys[h][j+1]);
        for (int j = i+1; j < min(n, i+half); ++j)
          ys[h][j] = f(ys[h][j-1], ys[h][j]);
      }
    }
  }
  T prod(int i, int j) { // [i, j) query
    --j;          // __CHAR_BIT__ is usually 8
    int h = sizeof(int)*__CHAR_BIT__-1-__builtin_clz(i ^ j);
    return f(ys[h][i], ys[h][j]);
  }
};
```

6.4 Fenwick Tree

```
struct FenwickTree {
    vector<ll> fwt;
    int LOGN = 20;

    FenwickTree(int n) {
        fwt.resize(n, 0);
        LOGN = log2(n) + 1;
    }

    void add(int ind, ll val = 1) {
        for (ind++; ind < fwt.size(); ind+=ind&-ind)
            fwt[ind]+=val;
    }

    ll query(int ind) {
        ll s = 0;
        for (ind++; ind > 0; ind-=ind&-ind)
            s += fwt[ind];
        return s;
    }

    ll lower_bound(int v) {
        v++;
        ll sum = 0;
        int pos = 0;
        for (int i = LOGN; i >= 0; i--) {
            if (pos + (1 << i) < fwt.size() &&
                sum + fwt[pos + (1 << i)] < v) {
                    sum += fwt[pos + (1 << i)];
                    pos += (1 << i);
            }
        }
        return pos - 1;
    }
};
```

6.5 Segment Tree

```
struct SegmentTree {
    int N = 0;

    struct Node {
        ll value = def;
        Node operator+(const Node &other) {
            return { this->value + other.value };
        }
    };

    vector<Node> seg;
    SegmentTree(int N) { this->N = N; seg.resize(4 * N); }

    void build(vector<ll>& v, int ind = 0, int l = 0, int r = -1) {
        if (r == -1) r = N;
        if (r - l == 1) { seg[ind] = v[l]; return; }
        int m = (l + r) / 2;
        build(v, ind * 2 + 1, l, m);
        build(v, ind * 2 + 2, m, r);
        seg[ind] = seg[ind * 2 + 1] + seg[ind * 2 + 2];
    }

    void update(int i, Node v, int ind = 0, int l = 0, int r = -1) {
        if (r == -1) r = N;
        if (r - l == 1) { seg[ind] = v; return; }
        int m = (l + r) / 2;
        if (i < m) update(i, v, ind * 2 + 1, l, m);
        else update(i, v, ind * 2 + 2, m, r);
        seg[ind] = seg[ind * 2 + 1] + seg[ind * 2 + 2];
    }

    // result in [b,e) of node that covers [l,r)
    Node query(int b, int e, int ind = 0, int l = 0, int r = -1) {
        if (r == -1) r = N;
        if (l>=e || r<=b) return { };
        if (l>=b && r<=e) return seg[ind];
        int m = (l + r) / 2;
        return query(b, e, ind * 2 + 1, l, m) + query(b, e, ind * 2 + 2, m, r);
    }
};
```

6.6 Lazy Segment Tree

```
struct LazySegmentTree {
    int N;

    struct Node {
        int ones = 0;
        int zeros = 0;
        bool lazy = 0;
        ll inversions1 = 0;
        ll inversions2 = 0;

        Node operator+(const Node &other) const {
            return {
                ones + other.ones,
                zeros + other.zeros,
                0,
                other.inversions1 + inversions1 + (ll)other.zeros * ones,
                other.inversions2 + inversions2 + (ll)other.ones * zeros
            };
        }
    };

    vector<Node> seg;
    LazySegmentTree(int n) {
        N = n;
        seg.resize(4 * N);
    }

    void push(int ind, int l, int r) {
        if (seg[ind].lazy == 0) return;
        swap(seg[ind].ones, seg[ind].zeros);
        swap(seg[ind].inversions1, seg[ind].inversions2);
        if (r - l != 1) {
            seg[2*ind+1].lazy ^= seg[ind].lazy;
            seg[2*ind+2].lazy ^= seg[ind].lazy;
        }
        seg[ind].lazy = 0;
    }

    void updateRange(int b, int e, int ind = 0, int l = 0, int r = -1) {
        if (r == -1) r = N;
        push(ind, l, r);
        if (b>=r || e<=l) return;
        if (b<=l && r<=e) {
            seg[ind].lazy ^= 1;
            push(ind, l, r);
            return;
        }
        int m = (l + r) / 2;
        updateRange(b, e, 2 * ind + 1, l, m);
        updateRange(b, e, 2 * ind + 2, m, r);
        seg[ind] = seg[2*ind+1] + seg[2*ind+2];
    }

    Node askRange(int b, int e, int ind = 0, int l = 0, int r = -1) {
        if (r == -1) r = N;
        push(ind, l, r);
        if (e<=l || b>=r) return { 0, 0, 0, 0, 0 };
        if (b<=l && r<=e) return seg[ind];
        int m = (l + r) / 2;
        auto res = askRange(b, e, 2 * ind + 1, l, m) + askRange(b, e, 2 * ind + 2, m, r);
        return res;
    }

    void put(int i, int what, int ind = 0, int l = 0, int r = -1) {
        if (r == -1) r = N;
        if (r - l == 1) {
            if (what) seg[ind].ones = 1;
            else seg[ind].zeros = 1;
            return;
        }
        int m = (l + r) / 2;
        if (i < m) put(i, what, 2 * ind + 1, l, m);
        else put(i, what, 2 * ind + 2, m, r);
        seg[ind] = seg[2*ind+1] + seg[2*ind+2];
    }
};
```

6.7 Sparse Lazy Segment Tree

Good implementation in KACTL.

6.8 Persistent Segment Tree

```
// Persistant Segment Tree
// O(log(N)) for update and query
// Eric K. Zhang; Nov. 22, 2017

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

struct PersistantSegTree {
  int N, t = 0;
  struct node {
    node *l = nullptr, *r = nullptr;
    int x = 0;
  };

  vector<node> vals;
  vector<node*> tree;

  PersistantSegTree(int n) {
    this->N = n;
      // PAY ATTENTION TO THE MEMORY
    vals.resize(2 * 100015 * 18);
    tree.resize(100015, nullptr);
    tree[0] = build_tree();
  }

  node* build_tree(int lo=0, int hi=-1) {
    if (hi == -1) hi = N - 1;
    node* cur = &vals[t++];
    if (lo != hi) {
      int mid = (lo + hi) / 2;
```

```cpp
      cur->l = build_tree(lo, mid);
      cur->r = build_tree(mid + 1, hi);
    }
    return cur;
  }

  node* update(node* n, int i, int x, int lo=0, int hi=-1) {
    if (hi == -1) hi = N - 1;
    if (hi < i || lo > i) return n;
    node* v = &vals[t++];
    if (lo == hi) { v->x = n->x + x; return v; }
    int mid = (lo + hi) / 2;
    v->l = update(n->l, i, x, lo, mid);
    v->r = update(n->r, i, x, mid + 1, hi);
    v->x = v->l->x + v->r->x;
    return v;
  }

  int query(node* n, int s, int e, int lo=0, int hi=-1) {
    if (hi == -1) hi = N - 1;
    if (hi < s || lo > e) return 0;
    if (lo >= s && hi <= e) return n->x;
    int mid = (lo + hi) / 2;
    return query(n->l, s, e, lo, mid) +
        query(n->r, s, e, mid + 1, hi);
  }
};

int main() {
  int N = 100;
  PersistantSegTree segtree(N);
  for (int i = 0; i < N; i++) {
    segtree.tree[i + 1] = segtree.update(segtree.tree[i], i, 1);
  }

  // how many of the numbers in indices [0..4]
  // are in the range [3..5]?
  cout << segtree.query(segtree.tree[5], 3, 5) << endl;
  return 0;
}
```

6.9 Persistent Fenwick Tree

```cpp
struct PersistentFenwickTree {
  struct change {
    int data, id;
    change(int d = 0, int i = 0) : data(d), id(i) { }
    inline friend bool operator<(const change &a, const change &b){
      return a.id < b.id;
    }
  };

  int n, now = 0;
  vector<vector<change>> tree;

  PersistentFenwickTree(int n) {
    tree.resize(n + 100);
    this->n = n;
    for (int i = 0; i <= n; i++) {
      tree[i].push_back(change());
    }
  }

  void modify(int i, int x) {
    for (i++; i <= n; i += i&(-i)) {
      int new_data = max(tree[i].back().data, x);
      tree[i].push_back(change(new_data, now));
    }
    now++;
  }

  int query(int i, int tree_id) {
    int ans = 0;
    vector<change>::iterator a;
    for (i++; i; i -= i&(-i)){
      a = upper_bound(tree[i].begin(), tree[i].end(), change(0, tree_id)) - 1;
      ans = max(ans, a->data);
    }
    return ans;
  }
};
```

6.10 2D Sparse Table

[25]

```cpp
// Watch out for memory limit
// Sparse table's build and memory complexity are
// O(N * M * log2(N) * log2(M))
// Check for when M = N
// Don't forget that you need to call the read and build function!
struct SparseTable2D {
    int LGN = 10, LGM = 10;
    vector<int> lg1, lg2;
    vector<vector<int>> a;
    int N, M;

    int* st_internal;

    inline int& st(int x, int y, int a, int b) {
        return st_internal[x * M * LGN * LGM + y * LGN * LGM + a * LGM + b];
    }

    SparseTable2D(int n, int m) {
        N = n;
        M = m;

        LGN = log2(N) + 1;
        LGM = log2(M) + 1;

        lg1.resize(N + 1);
        lg2.resize(M + 1);

        st_internal = new int[N * M * LGN * LGM + 10];
        memset(st_internal, INT_MAX, sizeof st_internal);
        // Don't forget to change INT_MAX if it's not a MIN query.
    }

    ~SparseTable2D() {
        delete[] st_internal;
    }

    void read() {
        a.resize(N, vector<int>(M, 0));
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < M; j++) {
                cin >> a[i][j];
            }
        }
    }

    void build() {
        for (int i = 2; i <= N; i++) lg1[i] = lg1[i >> 1] + 1;
        for (int i = 2; i <= M; i++) lg2[i] = lg2[i >> 1] + 1;
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < M; j++) {
                st(i, j, 0, 0) = a[i][j];
            }
        }
        for (int a = 0; a < LGN; a++) {
            for (int b = 0; b < LGM; b++) {
                if (a + b == 0) continue;
                for (int i = 0; i + (1 << a) <= N; i++) {
                    for (int j = 0; j + (1 << b) <= M; j++) {
                        if (!a) {
                            st(i, j, a, b) = min(
                                st(i, j, a, b - 1),
                                st(i, j + (1 << (b - 1)), a, b - 1));
                        } else {
                            st(i, j, a, b) = min(
                                st(i, j, a - 1, b),
                                st(i + (1 << (a - 1)), j, a - 1, b));
                        }
                    }
                }
            }
        }
    }

    int query(int x1, int y1, int x2, int y2) {
        x2++; y2++;
        int a = lg1[x2 - x1], b = lg2[y2 - y1];
        return min(
            min(st(x1, y1, a, b), st(x2 - (1 << a), y1, a, b)),
            min(st(x1, y2 - (1 << b), a, b), st(x2 - (1 << a), y2 - (1 << b), a, b))
        );
    }
};
```

6.11 2D Sparse Segment Tree

```cpp
// segtree_sparse.cpp
// Eric K. Zhang; Dec. 31, 2017

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

typedef long long LL;
#define MAXN 100013
#define MAXLGN 18
#define MAXSEG 262144
#define MAXSEG2 (2 * MAXN * MAXLGN * MAXLGN)

int N;
struct node {
        node *l, *r;
        LL x;
} vals[MAXSEG2]; int t = 0;
node* st[MAXSEG];

void update2(node*& n, int i, int x, int lo=0, int hi=-1) {
        if (hi == -1) hi = N - 1;
        if (hi < i || lo > i) return;
        if (!n) n = &vals[t++];
        if (lo == hi) {
                n->x += x;
                return;
        }
        int mid = (lo + hi) / 2;
        if (i <= mid) update2(n->l, i, x, lo, mid);
        else update2(n->r, i, x, mid + 1, hi);
        n->x += x;
}

void update(int i, int j, int x, int lo=0, int hi=-1, int node=0) {
        if (hi == -1) hi = N - 1;
        if (hi < i || lo > i) return;
        if (lo == hi) {
                update2(st[node], j, x);
                return;
        }
        int mid = (lo + hi) / 2;
        update(i, j, x, lo, mid, 2 * node + 1);
        update(i, j, x, mid + 1, hi, 2 * node + 2);
        update2(st[node], j, x);
}

LL query2(node* n, int s, int e, int lo=0, int hi=-1) {
        if (hi == -1) hi = N - 1;
        if (hi < s || lo > e || !n) return 0;
        if (s <= lo && hi <= e) return n->x;
        int mid = (lo + hi) / 2;
        return query2(n->l, s, e, lo, mid) + query2(n->r, s, e, mid + 1, hi);
}

LL query(int s, int e, int s2, int e2, int lo=0, int hi=-1, int node=0) {
        if (hi == -1) hi = N - 1;
        if (hi < s || lo > e) return 0;
        if (s <= lo && hi <= e) return query2(st[node], s2, e2);
        int mid = (lo + hi) / 2;
        return query(s, e, s2, e2, lo, mid, 2 * node + 1)
                + query(s, e, s2, e2, mid + 1, hi, 2 * node + 2);
}

int main() {
        N = 100;                                        // matrix:
                                                                 //       .   .   .
        update(0, 0, 5);                                //   [  5  0  0  0 ]
        update(2, 2, 30);                               // > [  0  0 10  0 ]
        update(3, 1, 10);                               // > [  0  0 30  0 ]
        update(1, 2, 10);                               // > [  0 10  0  0 ]
        cout << query(1, 3, 0, 2) << endl;
        //            >  >  .  .
}
```

6.12 K-th Minimum in Segment Tree

6.13 Treap

```cpp
/**
 * Author: someone on Codeforces
 * Date: 2017-03-14
 * Source: folklore
 * Description: A short self-balancing tree. It acts as a
 *  sequential container with log-time splits/joins, and
 *  is easy to augment with additional data.
 * Time: £O(\log N)£
 * Status: stress-tested
 */
#include <bits/stdc++.h>
#include <random>
#include <chrono>
#define ll long long
using namespace std;

mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
auto unif = uniform_int_distribution<int>(0, 1000000);

struct Node {
	Node *l = 0, *r = 0;
	// change to ll if necessary
	int val, y, c = 1; // c - subtree size
	int lazy = 0; // to propagate
	ll sum = 0;
	Node(int val) {
		this->val = val;
		this->sum = val;
		this->y = unif(rng);
	}

	void prop() {
		if (this->lazy == 0) return;
		this->val += lazy;

		if (l) {
			this->l->lazy += this->lazy;
		}

		if (r) {
			this->r->lazy += this->lazy;
		}

		this->lazy = 0;
		recalc();
	}

	void recalc() {
		this->c = 1;
		this->sum = this->val + this->lazy;

		if (l) {
			this->c += l->c;
			this->sum += l->lazy * l->c + l->sum;
		}

		if (r) {
			this->c += r->c;
			this->sum += r->lazy * r->c + r->sum;
		}
	}
};

struct Treap {
	int cnt(Node* n) { return n ? n->c : 0; }

	template<class F> void each(Node* n, F f) {
		if (n) {
			n->prop();
			each(n->l, f);
			f(n->val);
			each(n->r, f);
		}
	}

	pair<Node*, Node*> split(Node* n, int k) {
		if (!n) return {};
		n->prop();
		if (cnt(n->l) >= k) { // "n->val >= k" for lower_bound(k)
			auto pa = split(n->l, k);
			n->l = pa.second;
			n->recalc();
			return {pa.first, n};
		} else {
			auto pa = split(n->r, k - cnt(n->l) - 1); // and just "k"
			n->r = pa.first;
			n->recalc();
			return {n, pa.second};
		}
	}

	Node* merge(Node* l, Node* r) {
		if (!l) return r;
		if (!r) return l;
		l->prop();
		r->prop();
		if (l->y > r->y) {
			l->r = merge(l->r, r);
			l->recalc();
			return l;
		} else {
			r->l = merge(l, r->l);
			r->recalc();
			return r;
		}
	}

	Node* insert(Node* t, Node* n, int pos) {
		auto pa = split(t, pos);
		return merge(merge(pa.first, n), pa.second);
	}

	// Move the range [l, r) to index k
	void move(Node*& t, int l, int r, int k) {
		Node *a, *b, *c;
		tie(a,b) = split(t, l); tie(b,c) = split(b, r - l);
		if (k <= l) t = merge(insert(a, b, k), c);
		else t = merge(a, insert(c, b, k - r));
	}

	void rangeAdd(Node*& root, int l, int r, int add) {
		assert(l <= r);
		auto [L, mid] = split(root, l);
		auto [mid2, R] = split(mid, r-l+1);
		mid2->lazy += add;
		root = merge(L, merge(mid2, R));
	}

	int rangeSum(Node*& root, int l, int r) {
		assert(l <= r);
		auto [L, mid] = split(root, l);
		auto [mid2, R] = split(mid, r-l+1);
		int res = mid2->sum;
		root = merge(L, merge(mid2, R));
		return res;
	}

	Node* createTreap(const vector<int> &v) {
		assert(!v.empty());
		Node* root = new Node(v[0]);
		for (int i = 1; i < v.size(); i++) {
			Node* right = new Node(v[i]);
			root = merge(root, right);
		}
		return root;
	}
};

int main() {
	int n, q; cin >> n >> q;
	vector<int> a(n);
	for (int i = 0; i < n; i++) {
		cin >> a[i];
	}
	Treap t;
	Node* root = t.createTreap(a);
	for (int i = 0; i < q; i++) {
		int type;
		cin >> type;
		if (type == 0) {
			int x, y, z;
			cin >> x >> y >> z;
			x--; y--;
			t.rangeAdd(root, x, y, z);
		} else {
			int x, y;
			cin >> x >> y;
			x--; y--;
			cout << t.rangeSum(root, x, y) << endl;
		}
	}

	return 0;
}
```

6.14 Min-Max Heap

```cpp
//
// Min-Map Heap (four heaps technique)
//
// Description:
//   A data structure for push/min/max/popmin/popmax.
//
// Algorithm:
//   Maintain two min heaps (minh, minp) an two max heaps (maxh, maxp).
//   The actual elements in the heap is (minh \cup maxh) \setminus (\minp \cup maxp).
//
// Complexity:
//   Amortized O(1) for push and top, and O(log n) for pop.
//

template <class T, class L = less<T>, class G = greater<T>>
struct minmax_heap {
  priority_queue<T, vector<T>, G> minh, minp;
  priority_queue<T, vector<T>, L> maxh, maxp;
  void normalize() {
    while (!minp.empty() && minp.top() == minh.top()) {
      minp.pop();
      minh.pop();
    }
    while (!maxp.empty() && maxp.top() == maxh.top()) {
      maxp.pop();
      maxh.pop();
    }
  }
  void push(T x) { minh.push(x); maxh.push(x); }
  T min() { normalize(); return minh.top(); }
  T max() { normalize(); return maxh.top(); }
  void pop_min() { normalize(); maxp.push(minh.top()); minh.pop(); }
  void pop_max() { normalize(); minp.push(maxh.top()); maxh.pop(); }
};
```

6.15 Indexed Binary Heap

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

using namespace std;

template <class T>
struct binary_heap_indexed {
    vector<T> heap;
    vector<int> pos2Id;
    vector<int> id2Pos;
    int size;

    binary_heap_indexed() : size(0) {}

    binary_heap_indexed(int n) : heap(n), pos2Id(n), id2Pos(n), size(0) {}

    void add(int id, T value) {
        heap[size] = value;
        pos2Id[size] = id;
        id2Pos[id] = size;
        up(size++);
    }

    int remove_min() {
        int removedId = pos2Id[0];
        heap[0] = heap[--size];
        pos2Id[0] = pos2Id[size];
        id2Pos[pos2Id[0]] = 0;
        down(0);
        return removedId;
    }

    void remove(int id) {
        int pos = id2Pos[id];
        pos2Id[pos] = pos2Id[--size];
        id2Pos[pos2Id[pos]] = pos;
        change_value(pos2Id[pos], heap[size]);
    }

    void change_value(int id, T value) {
        int pos = id2Pos[id];
        if (heap[pos] < value) {
            heap[pos] = value;
            down(pos);
        } else if (heap[pos] > value) {
            heap[pos] = value;
            up(pos);
        }
    }

    void up(int pos) {
        while (pos > 0) {
            int parent = (pos - 1) / 2;
            if (heap[pos] >= heap[parent])
                break;
            exchange(pos, parent);
            pos = parent;
        }
    }

    void down(int pos) {
        while (true) {
            int child = 2 * pos + 1;
            if (child >= size)
                break;
            if (child + 1 < size && heap[child + 1] < heap[child])
                ++child;
            if (heap[pos] <= heap[child])
                break;
            exchange(pos, child);
            pos = child;
        }
    }

    void exchange(int i, int j) {
        swap(heap[i], heap[j]);
        swap(pos2Id[i], pos2Id[j]);
        id2Pos[pos2Id[i]] = i;
        id2Pos[pos2Id[j]] = j;
    }
};
```

6.16 Persistent Heap

```cpp
//
// Leftist Heap
//
// Description:
//
//   Leftist heap is a heap data structure that allows
//   the meld (merge) operation in O(log n) time.
//   Use this for persistent heaps.
//
// Complexity:
//
//   O(1) for top, O(log n) for push/pop/meld
//
// g++ -std=c++17 -O3 -fmax-errors=1 -fsanitize=undefined
#include <bits/stdc++.h>

using namespace std;

#define fst first
#define snd second
#define all(c) ((c).begin()), ((c).end())

template <class T>
struct LeftistHeap {
  struct Node {
    T key;
    Node *left = 0, *right = 0;
    int dist = 0;
  } *root = 0;
  static Node *merge(Node *x, Node *y) {
    if (!x) return y;
    if (!y) return x;
    if (x->key > y->key) swap(x, y);
    x->right = merge(x->right, y);
    if (!x->left || x->left->dist < x->dist) swap(x->left, x->right);
    x->dist = (x->right ? x->right->dist : 0) + 1;
    return x;
  }
  void push(T key) { root = merge(root, new Node({key})); }
  void pop() { root = merge(root->left, root->right); }
  T top() { return root->key; }
};

//
// Persistent Implementaiton. (allow copy)
//
template <class T>
struct PersistentLeftistHeap {
```

```
  struct Node {
    T key;
    Node *left = 0, *right = 0;
    int dist = 0;
  } *root = 0;
  static Node *merge(Node *x, Node *y) {
    if (!x) return y;
    if (!y) return x;
    if (x->key > y->key) swap(x, y);
    x = new Node(*x);
    x->right = merge(x->right, y);
    if (!x->left || x->left->dist < x->dist) swap(x->left, x->right);
    x->dist = (x->right ? x->right->dist : 0) + 1;
    return x;
  }
  void push(T key) { root = merge(root, new Node({key})); }
  void pop() { root = merge(root->left, root->right); }
  T top() { return root->key; }
};

int main() {
  PersistentLeftistHeap<int> heap;
  heap.push(3);
  heap.push(1);
  heap.push(4);
  heap.push(1);
  heap.push(5);
  cout << heap.top() << endl; heap.pop();
  cout << heap.top() << endl; heap.pop();
  auto temp = heap;
  cout << heap.top() << endl; heap.pop();
  cout << heap.top() << endl; heap.pop();
  cout << temp.top() << endl; temp.pop();
  cout << temp.top() << endl; temp.pop();
}
```

6.17 Skew Heap

```
// Skew Heap
//
// Description:
//   Heap data structure with the following operations.
//
//   1. push a value O(log n)
//   2. pop the smallest value O(log n)
//   3. merge two heaps O(log n + log m)
//   4. add a value to all elements O(1)
//

struct skew_heap {
  struct node {
    node *ch[2];
    int key;
    int delta;
  } *root;
  skew_heap() : root(0) { }
  void propagate(node *a) {
    a->key += a->delta;
    if (a->ch[0]) a->ch[0]->delta += a->delta;
    if (a->ch[1]) a->ch[1]->delta += a->delta;
    a->delta = 0;
  }
  node *merge(node *a, node *b) {
    if (!a || !b) return a ? a : b;
    propagate(a); propagate(b);
    if (a->key > b->key) swap(a, b); // min heap
    a->ch[1] = merge(b, a->ch[1]);
    swap(a->ch[0], a->ch[1]);
    return a;
  }
  void push(int key) {
    node *n = new node();
    n->ch[0] = n->ch[1] = 0;
    n->key = key; n->delta = 0;
    root = merge(root, n);
  }
  void pop() {
    propagate(root);
    node *temp = root;
    root = merge(root->ch[0], root->ch[1]);
  }
  int top() {
    propagate(root);
    return root->key;
  }
  bool empty() {
    return !root;
  }
  void add(int delta) {
    if (root) root->delta += delta;
  }
  void merge(skew_heap x) { // destroy x
    root = merge(root, x.root);
  }
};
```

6.18 Persistent Array

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

struct Node {
	int val;
	Node *l, *r;

	Node(ll x) : val(x), l(nullptr), r(nullptr) {}
	Node(Node *L, Node *R) : val(0), l(L), r(R) {}
};

int n, init[100001]; // The initial array and its size
Node* roots[100001]; // The persistent array's roots

Node* build(int l = 0, int r = n - 1) {
	if (l == r) return new Node(init[l]);
	int mid = (l + r) / 2;
	return new Node(build(l, mid), build(mid + 1, r));
}

Node* update(Node* node, int val, int pos, int l = 0, int r = n - 1) {
	if (l == r) return new Node(val);
	int mid = (l + r) / 2;
	if (pos > mid) return new Node(node->l, update(node->r, val, pos, mid + 1, r));
	else return new Node(update(node->l, val, pos, l, mid), node->r);
}

int query(Node* node, int pos, int l = 0, int r = n - 1) {
	if (l == r) return node->val;
	int mid = (l + r) / 2;
	if (pos > mid) return query(node->r, pos, mid + 1, r);
	return query(node->l, pos, l, mid);
}

// Gets the array item at a given index and time
int get_item(int time, int index) {
	return query(roots[time], index);
}

// Updates the array item at a given index and time
void update_item(int prev_time, int curr_time, int index, int value) {
	roots[curr_time] = update(roots[prev_time], index, value);
}

// Initializes the persistent array, given an input array
void init_arr(int nn, vector<int> initial_array) {
	n = nn;
	for (int i = 0; i < n; i++)
		init[i] = initial_array[i];
	roots[0] = build();
}

int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    for (int i = 0; i < n; i++) {
        cin >> v[i];
    }

    init_arr(n, v);

    for (int i = 0; i < 5; i++) {
        update_item(i, i+1, i, i);
    }

    for (int time = 0; time < 5; time++) {
        for (int i = 0; i < n; i++) {
            cout << get_item(time, i) << " ";
        }
        cout << endl;
    }
    return 0;
}
```

6.19 Sparse Bitset

6.20 Interval Container

6.21 Stream Median

TODO: Needs testing...

```
template<typename T>
struct Median {
  priority_queue<T, vector<T>, greater<T>> right;
  priority_queue<T, vector<T>, less<T>> left;

  T get() {
    if (left.empty() && right.empty()) return -1;
    if (left.empty()) return right.top();
    if (right.empty()) return left.top();
    if (left.size() == right.size()) return left.top();
    return left.size() > right.size() ? left.top() : right.top();
  }

  void insert(T num) {
    T median = get();
    if (num < median) {
      if (left.size() > right.size()) {
        right.push(left.top());
        left.pop();
      }
      left.push(num);
    } else {
      if (right.size() > left.size()) {
        left.push(right.top());
        right.pop();
      }
      right.push(num);
    }
  }
};
```

6.22 Palindromic Trie

```
struct palindromic_tree {
  vector<vector<int>> next;
  vector<int> suf, len;
  int new_node() {
    next.push_back(vector<int>(256,-1));
    suf.push_back(0);
    len.push_back(0);
    return next.size() - 1;
  }
  palindromic_tree(char *s) {
    len[new_node()] = -1;
    len[new_node()] = 0;
    int t = 1;
    for (int i = 0; s[i]; ++i) {
      int p = t;
      for (; i-1-len[p] < 0 ||
             s[i-1-len[p]] != s[i];
             p = suf[p]);

      if ((t = next[p][s[i]]) >= 0) continue;
      t = new_node();
      len[t] = len[p] + 2;
      next[p][s[i]] = t;
      if (len[t] == 1) {
        suf[t] = 1; // EMPTY
      } else {
        p = suf[p];
        for (; i-1-len[p] < 0 || s[i-1-len[p]] != s[i];
              p = suf[p]);
        suf[t] = next[p][s[i]];
      }
    }
  }
  void display() {
    vector<char> buf;
    function<void (int, string)> rec =
      [&](int p, string depth) {
        if (len[p] > 0) {
          cout << depth;
          for (int i = buf.size()-1; i >= 0; --i)
            cout << buf[i];
          for (int i = len[p] % 2; i < buf.size(); ++i)
            cout << buf[i];
          cout << endl;
        }
        for (int a = 0; a < 256; ++a) {
          if (next[p][a] >= 0) {
            buf.push_back(a);
            rec(next[p][a], depth + " ");
            buf.pop_back();
          }
        }
      };

    cout << "---" << endl;
    rec(0, "");
    cout << "---" << endl;
    rec(1, "");
  }
};
```

6.23 Eval-Link-Update Tree

6.24 Euler Tour Tree

The Euler Tour Tree is a dynamic connectivity based data structure [20] [24] capable of the following operations in $O(log(N))$ time:

- `make_node(x)` ... return singleton with value x
- `link(u,v)` ... add link between u and v
- `cut(uv)` ... remove edge uv
- `sum_in_component(u)` ... return sum of all values in the component

Note that when adding a link between u and v, they can't be already in the same component. If the problem requires the sum of a component in a graph instead of a tree, we have explain how you can achieve this using the Dynamic Connectivity.

The sum query can be interchanged with another query type as long as the operation is in a associative field.

```
struct euler_tour_tree {
  struct node {
    int x, s; // value, sum
    node *ch[2], *p, *r;
  };
  int sum(node *t) { return t ? t->s : 0; }
  node *update(node *t) {
    if (t) t->s = t->x + sum(t->ch[0]) + sum(t->ch[1]);
    return t;
  }
  int dir(node *t) { return t != t->p->ch[0]; }
  void connect(node *p, node *t, int d) {
    p->ch[d] = t; if (t) t->p = p;
    update(p);
  }
  node *disconnect(node *t, int d) {
    node *c = t->ch[d]; t->ch[d] = 0; if (c) c->p = 0;
    update(t);
    return c;
  }
  void rot(node *t) {
    node *p = t->p;
    int d = dir(t);
    if (p->p) connect(p->p, t, dir(p));
    else      t->p = p->p;
    connect(p, t->ch[!d], d);
    connect(t, p, !d);
  }
  void splay(node *t) {
    for (; t->p; rot(t))
      if (t->p->p) rot(dir(t) == dir(t->p) ? t->p : t);
  }
  void join(node *s, node *t) {
    if (!s || !t) return;
    for (; s->ch[1]; s = s->ch[1]); splay(s);
    for (; t->ch[0]; t = t->ch[0]); connect(t, s, 0);
  }
  node *make_node(int x, node *l = 0, node *r = 0) {
    node *t = new node({x});
    connect(t, l, 0); connect(t, r, 1);
    return t;
  }
  node *link(node *u, node *v, int x = 0) {
    splay(u); splay(v);
    node *uv = make_node(x, u, disconnect(v,1));
    node *vu = make_node(0, v, disconnect(u,1));
    uv->r = vu; vu->r = uv;
    join(uv, vu);
    return uv;
  }
  void cut(node *uv) {
    splay(uv); disconnect(uv,1); splay(uv->r);
    join(disconnect(uv,0), disconnect(uv->r,1));
    delete uv, uv->r;
  }
  int sum_in_component(node *u) {
    splay(u);
    return u->s;
  }
};
```

6.25 Link-Cut Tree

The Link-Cut Tree is one of the most powerful tree data structures. It can support the following queries in amortized $O(log(n))$ time:

- `link(u, v)` ... add an edge (u, v)
- `cut(u)` ... cut the edge $(parent[u], u)$
- `lca(u, v)` ... returns the least common ancestor of u and v
- `connected(u, v)` ... checks if u and v are part of the same tree
- `find_root(u)` ... returns the root of u
- `component_size(u)` ... returns the size of the component containing u
- `subtree_size(u)` ... returns the subtree size of u
- `depth(u)` ... returns the depth of u
- `subtree_query(u, root)` ... subtree query with a given root
- `query(u, v)` ... path from u to v - sum query
- `update(u, x)` ... update the value of node u to x
- `update(u, v, x)` ... update the path from u to v with x

The following implementation includes lazy propagation which allows updating of paths. It is possible to change the sum function to any associative function.

```cpp
struct node {
  int p = 0, c[2] = {0, 0}, pp = 0;
  bool flip = 0;
  int sz = 0, ssz = 0, vsz = 0;
  // sz -> aux tree size
  // ssz -> subtree size in rep tree
  // vsz -> virtual tree size
  long long val = 0, sum = 0, lazy = 0;
  long long subsum = 0, vsum = 0;
  node() {}
  node(int x) {
    val = x; sum = x;
    sz = 1; lazy = 0;
    ssz = 1; vsz = 0;
    subsum = x; vsum = 0;
  }
};

struct LCT {
  vector<node> t;
  LCT(int n) : t(n + 1) {}

  // <independant splay tree code>
  int dir(int x, int y) { return t[x].c[1] == y; }
  void set(int x, int d, int y) {
    if (x) t[x].c[d] = y, pull(x);
    if (y) t[y].p = x;
  }
  void pull(int x) {
    if (!x) return;
    int &l = t[x].c[0], &r = t[x].c[1];
    t[x].sum = t[l].sum + t[r].sum + t[x].val;
    t[x].sz = t[l].sz + t[r].sz + 1;
    t[x].ssz = t[l].ssz + t[r].ssz + t[x].vsz + 1;
    t[x].subsum = t[l].subsum + t[r].subsum + t[x].vsum
        + t[x].val;
  }
  void push(int x) {
    if (!x) return;
    int &l = t[x].c[0], &r = t[x].c[1];
    if (t[x].flip) {
      swap(l, r);
      if (l) t[l].flip ^= 1;
      if (r) t[r].flip ^= 1;
      t[x].flip = 0;
    }
    if (t[x].lazy) {
      t[x].val += t[x].lazy;
      t[x].sum += t[x].lazy * t[x].sz;
      t[x].subsum += t[x].lazy * t[x].ssz;
      t[x].vsum += t[x].lazy * t[x].vsz;
      if (l) t[l].lazy += t[x].lazy;
      if (r) t[r].lazy += t[x].lazy;
      t[x].lazy = 0;
    }
  }
  void rotate(int x, int d) {
    int y = t[x].p, z = t[y].p, w = t[x].c[d];
    swap(t[x].pp, t[y].pp);
    set(y, !d, w);
    set(x, d, y);
    set(z, dir(z, y), x);
  }
  void splay(int x) {
    for (push(x); t[x].p;) {
      int y = t[x].p, z = t[y].p;
      push(z); push(y); push(x);
      int dx = dir(y, x), dy = dir(z, y);
      if (!z) rotate(x, !dx);
      else if (dx == dy) rotate(y, !dx), rotate(x, !dx);
      else rotate(x, dy), rotate(x, dx);
    }
  }
  // </independant splay tree code>
```

```cpp
  // making it a root in the rep. tree
  void make_root(int u) {
    access(u);
    int l = t[u].c[0];
    t[l].flip ^= 1;
    swap(t[l].p, t[l].pp);
    t[u].vsz += t[l].ssz;
    t[u].vsum += t[l].subsum;
    set(u, 0, 0);
  }
  // make the path from root to u a preferred path
  // returns last path-parent of a node as it moves up
       the tree
  int access(int _u) {
    int last = _u;
    for (int v = 0, u = _u; u; u = t[v = u].pp) {
      splay(u); splay(v);
      t[u].vsz -= t[v].ssz;
      t[u].vsum -= t[v].subsum;
      int r = t[u].c[1];
      t[u].vsz += t[r].ssz;
      t[u].vsum += t[r].subsum;
      t[v].pp = 0;
      swap(t[r].p, t[r].pp);
      set(u, 1, v);
      last = u;
    }
    splay(_u);
    return last;
  }
```

```cpp
void link(int u, int v) { // u -> v
  // assert(!connected(u, v));
  make_root(v);
  access(u); splay(u);
  t[v].pp = u;
  t[u].vsz += t[v].ssz;
  t[u].vsum += t[v].subsum;
}
// cut par[u] -> u, u is non root vertex
void cut(int u) {
  access(u);
  assert(t[u].c[0] != 0);
  t[t[u].c[0]].p = 0;
  t[u].c[0] = 0;
  pull(u);
}
// parent of u in the rep. tree
int get_parent(int u) {
  access(u); splay(u); push(u);
  u = t[u].c[0]; push(u);
  while (t[u].c[1]) {
    u = t[u].c[1]; push(u);
  }
  splay(u);
  return u;
}
// root of the rep. tree containing this node
int find_root(int u) {
  access(u); splay(u); push(u);
  while (t[u].c[0]) {
    u = t[u].c[0]; push(u);
  }
  splay(u);
  return u;
}
bool connected(int u, int v) {
  return find_root(u) == find_root(v);
}
// depth in the rep. tree
int depth(int u) {
  access(u); splay(u);
  return t[u].sz;
}
int lca(int u, int v) {
  // assert(connected(u, v));
  if (u == v) return u;
  if (depth(u) > depth(v)) swap(u, v);
  access(v);
  return access(u);
}
int is_root(int u) {
  return get_parent(u) == 0;
}
```

```cpp
int component_size(int u) {
  return t[find_root(u)].ssz;
}
int subtree_size(int u) {
  int p = get_parent(u);
  if (p == 0) {
    return component_size(u);
  }
  cut(u);
  int ans = component_size(u);
  link(p, u);
  return ans;
}
long long component_sum(int u) {
  return t[find_root(u)].subsum;
}
long long subtree_sum(int u) {
  int p = get_parent(u);
  if (p == 0) {
    return component_sum(u);
  }
  cut(u);
  long long ans = component_sum(u);
  link(p, u);
  return ans;
}
// sum of the subtree of u when root is specified
long long subtree_query(int u, int root) {
  int cur = find_root(u);
  make_root(root);
  long long ans = subtree_sum(u);
  make_root(cur);
  return ans;
}
// path sum
long long query(int u, int v) {
  int cur = find_root(u);
  make_root(u); access(v);
  long long ans = t[v].sum;
  make_root(cur);
  return ans;
}
void update(int u, int x) {
  access(u); splay(u);
  t[u].val += x;
}
// add x to the nodes on the path from u to v
void update(int u, int v, int x) {
  int cur = find_root(u);
  make_root(u); access(v);
  t[v].lazy += x;
  make_root(cur);
}
```

Example Problems

Problem 5 - *SPOJ - Dynamic Tree Connectivity* [11]

Problem 6 - *CODECHEF - Query on a tree VI* [4]

Given a tree, all the nodes are black initially. There are two types of queries:

- `Query 1`: How many nodes are connected to u, such that two nodes are connected iff all of the nodes on the path from u to v have the same color.
- `Query 2`: Toggle the color of u.

6.26 Kinetic Tournament Data Structure

```cpp
// kinetic_tournament.cpp
// Eric K. Zhang; Aug. 29, 2020
//
// This is an implementation of a _kinetic tournament_, which I originally
// learned about from Daniel Zhang in this Codeforces blog comment:
// https://codeforces.com/blog/entry/68534#comment-530381
//
// The functionality of the data structure is a mix between a line container,
// i.e., "convex hull trick", and a segment tree.
//
// Suppose that you have an array containing pairs of nonnegative integers,
// A[i] and B[i]. You also have a global parameter T, corresponding to the
// "temperature" of the data structure. Your goal is to support the following
// queries on this data:
//
//   - update(i, a, b): set A[i] = a and B[i] = b
//   - query(s, e): return min{s <= i <= e} A[i] * T + B[i]
//   - heaten(new_temp): set T = new_temp
//       [precondition: new_temp >= current value of T]
//
// (For simplicity, we set A[i] = 0 and B[i] = LLONG_MAX for uninitialized
// entries, which should not change the query results.)
//
// This allows you to essentially do arbitrary lower convex hull queries on a
// collection of lines, as well as any contiguous subcollection of those lines.
// This is more powerful than standard convex hull tricks and related data
// structures (Li-Chao Segment Tree) for three reasons:
//
//   - You can arbitrarily remove/edit lines, not just add them.
//   - Dynamic access to any subinterval of lines, which lets you avoid costly
//     merge small-to-large operations in some cases.
//   - Easy to reason about and implement from scratch, unlike dynamic CHT.
//
// The tradeoff is that you can only query sequential values (temperature is
// only allowed to increase) for amortization reasons, but this happens to be
// a fairly common case in many problems.
//
// Time complexity:
//
//   - query: O(log n)
//   - update: O(log n)
//   - heaten: O(log^2 n)  [amortized]
//
// Verification: FBHC 2020, Round 2, Problem D "Log Drivin' Hirin'"

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

template <typename T = int64_t>
class kinetic_tournament {
        const T INF = numeric_limits<T>::max();
        typedef pair<T, T> line;

        size_t n;          // size of the underlying array
        T temp;            // current temperature
        vector<line> st;   // tournament tree
        vector<T> melt;    // melting temperature of each subtree

        inline T eval(const line& ln, T t) {
                return ln.first * t + ln.second;
        }

        inline bool cmp(const line& line1, const line& line2) {
                auto x = eval(line1, temp);
                auto y = eval(line2, temp);
                if (x != y) return x < y;
                return line1.first < line2.first;
        }

        T next_isect(const line& line1, const line& line2) {
                if (line1.first > line2.first) {
                        T delta = eval(line2, temp) - eval(line1, temp);
                        T delta_slope = line1.first - line2.first;
                        assert(delta > 0);
                        T mint = temp + (delta - 1) / delta_slope + 1;
                        return mint > temp ? mint : INF;  // prevent overflow
                }
                return INF;
        }

        void recompute(size_t lo, size_t hi, size_t node) {
                if (lo == hi || melt[node] > temp) return;

                size_t mid = (lo + hi) / 2;
                recompute(lo, mid, 2 * node + 1);
                recompute(mid + 1, hi, 2 * node + 2);

                auto line1 = st[2 * node + 1];
                auto line2 = st[2 * node + 2];
                if (!cmp(line1, line2))
                        swap(line1, line2);
                st[node] = line1;

                melt[node] = min(melt[2 * node + 1], melt[2 * node + 2]);
                if (line1 != line2) {
                        T t = next_isect(line1, line2);
                        assert(t > temp);
                        melt[node] = min(melt[node], t);
                }
        }

        void update(size_t i, T a, T b, size_t lo, size_t hi, size_t node) {
                if (i < lo || i > hi) return;
                if (lo == hi) {
                        st[node] = {a, b};
                        return;
                }
                size_t mid = (lo + hi) / 2;
                update(i, a, b, lo, mid, 2 * node + 1);
                update(i, a, b, mid + 1, hi, 2 * node + 2);
                melt[node] = 0;
                recompute(lo, hi, node);
        }

        T query(size_t s, size_t e, size_t lo, size_t hi, size_t node) {
                if (hi < s || lo > e) return INF;
                if (s <= lo && hi <= e) return eval(st[node], temp);
                size_t mid = (lo + hi) / 2;
                return min(query(s, e, lo, mid, 2 * node + 1),
                        query(s, e, mid + 1, hi, 2 * node + 2));
        }
public:
        // Constructor for a kinetic tournament, takes in the size n of the
        // underlying arrays a[..], b[..] as input.
        kinetic_tournament(size_t size) : n(size), temp(0) {
                assert(size > 0);
                size_t seg_size = ((size_t) 2) << (64 - __builtin_clzll(n - 1));
                st.resize(seg_size, {0, INF});
                melt.resize(seg_size, INF);
        }

        // Sets A[i] = a, B[i] = b.
        void update(size_t i, T a, T b) {
                update(i, a, b, 0, n - 1, 0);
        }

        // Returns min{s <= i <= e} A[i] * T + B[i].
        T query(size_t s, size_t e) {
                return query(s, e, 0, n - 1, 0);
        }

        // Increases the internal temperature to new_temp.
        void heaten(T new_temp) {
                assert(new_temp >= temp);
                temp = new_temp;
                recompute(0, n - 1, 0);
        }
};
```

7 Maths

7.1 Basic Algorithms

7.1.1 Binary Exponentiation

```cpp
ll expo(ll base, ll power) {
    ll result = 1;

    while (power > 0) {
        if (power % 2 == 1) {
            result = result * base;
            power--;
        }

        base = base * base;
        power /= 2;
    }

    return result;
}
```

7.1.2 Sum of Geometric Progression

Given integers A, N and M, find $\sum_{i=0}^{N-1} A^i$, mod M.

```cpp
int geometric(int N, int A, int M) {
  ll T = 1;
  ll E = A % M;
  ll total = 0;
  while (N > 0) {
    if (N & 1) {
      total = (E * total + T) % M;
    }
    T = ((E + 1) * T) % M;
    E = (E * E) % M;
    N = N / 2;
  }
  return total;
}
```

7.2 Primes

List of primes close to 10^9:

- 997927277
- 997929287
- 999999599
- 1000001333
- 1000000123

```cpp
// Primes less than 1000:
//      2     3     5     7    11    13    17    19    23    29    31    37
//     41    43    47    53    59    61    67    71    73    79    83    89
//     97   101   103   107   109   113   127   131   137   139   149   151
//    157   163   167   173   179   181   191   193   197   199   211   223
//    227   229   233   239   241   251   257   263   269   271   277   281
//    283   293   307   311   313   317   331   337   347   349   353   359
//    367   373   379   383   389   397   401   409   419   421   431   433
//    439   443   449   457   461   463   467   479   487   491   499   503
//    509   521   523   541   547   557   563   569   571   577   587   593
//    599   601   607   613   617   619   631   641   643   647   653   659
//    661   673   677   683   691   701   709   719   727   733   739   743
//    751   757   761   769   773   787   797   809   811   821   823   827
//    829   839   853   857   859   863   877   881   883   887   907   911
//    919   929   937   941   947   953   967   971   977   983   991   997
```

```cpp
// Other primes:
//    The largest prime smaller than 10 is 7.
//    The largest prime smaller than 100 is 97.
//    The largest prime smaller than 1000 is 997.
//    The largest prime smaller than 10000 is 9973.
//    The largest prime smaller than 100000 is 99991.
//    The largest prime smaller than 1000000 is 999983.
//    The largest prime smaller than 10000000 is 9999991.
//    The largest prime smaller than 100000000 is 99999989.
//    The largest prime smaller than 1000000000 is 999999937.
//    The largest prime smaller than 10000000000 is 9999999967.
//    The largest prime smaller than 100000000000 is 99999999977.
//    The largest prime smaller than 1000000000000 is 999999999989.
//    The largest prime smaller than 10000000000000 is 9999999999971.
//    The largest prime smaller than 100000000000000 is 99999999999973.
//    The largest prime smaller than 1000000000000000 is 999999999999989.
//    The largest prime smaller than 10000000000000000 is 9999999999999937.
//    The largest prime smaller than 100000000000000000 is 99999999999999997.
//    The largest prime smaller than 1000000000000000000 is 999999999999999989.
```

7.2.1 Carmichael Lambda (Universal Totient Function)

$\lambda(n)$ is a smallest number that satisfies $a^{\lambda(n)} \equiv 1 \pmod{n}$ for all a coprime with n. This is also known as an universal totien tunction $\psi(n)$.

Complexity:

- `carmichael_lambda(n)` ... $O(\sqrt{n})$ by trial division.
- `carmichael_lambda(lo,hi)` ... $O((H-L)loglog(H))$ by prime sieve.

Note. Required GCD and LCM.

```
ll carmichael_lambda(ll n) {
  ll lambda = 1;
  if (n % 8 == 0) n /= 2;
  for (ll d = 2; d*d <= n; ++d) {
    if (n % d == 0) {
      n /= d;
      ll y = d - 1;
      while (n % d == 0) {
        n /= d;
        y *= d;
      }
      lambda = lcm(lambda, y);
    }
  }
  if (n > 1) lambda = lcm(lambda, n-1);
  return lambda;
}
```

```
  vector<ll> primes(ll lo, ll hi) { // primes in [lo, hi)
  const ll M = 1 << 14, SQR = 1 << 16;
  vector<bool> composite(M), small_composite(SQR);

  vector<pair<ll,ll>> sieve;
  for (ll i = 3; i < SQR; i+=2) {
    if (!small_composite[i]) {
      ll k = i*i + 2*i*max(0.0, ceil((lo - i*i)/(2.0*i)));
      sieve.push_back({2*i, k});
      for (ll j = i*i; j < SQR; j += 2*i)
        small_composite[j] = 1;
    }
  }
  vector<ll> ps;
  if (lo <= 2) { ps.push_back(2); lo = 3; }
  for (ll k = lo|1, low = lo; low < hi; low += M) {
    ll high = min(low + M, hi);
    fill(all(composite), 0);
    for (auto &z: sieve)
      for (; z.snd < high; z.snd += z.fst)
        composite[z.snd - low] = 1;
    for (; k < high; k+=2)
      if (!composite[k - low]) ps.push_back(k);
  }
  return ps;
}
vector<ll> primes(ll n) { // primes in [0,n)
  return primes(0,n);
}
vector<ll> carmichael_lambda(ll lo, ll hi) { // lambda(n)
    for all n in [lo, hi)
  vector<ll> ps = primes(sqrt(hi)+1);
  vector<ll> res(hi-lo), lambda(hi-lo, 1);
  iota(all(res), lo);

  for (ll k = ((lo+7)/8)*8; k < hi; k += 8) res[k-lo] /= 2;
  for (ll p: ps) {
    for (ll k = ((lo+(p-1))/p)*p; k < hi; k += p) {
      if (res[k-lo] < p) continue;
      ll t = p - 1;
      res[k-lo] /= p;
      while (res[k-lo] > 1 && res[k-lo] % p == 0) {
        t *= p;
        res[k-lo] /= p;
      }
      lambda[k-lo] = lcm(lambda[k-lo], t);
    }
  }
  for (ll k = lo; k < hi; ++k) {
    if (res[k-lo] > 1)
      lambda[k-lo] = lcm(lambda[k-lo], res[k-lo]-1);
  }
  return lambda; // lambda[k-lo] = lambda(k)
}
```

7.2.2 Miller Rabin Primality Test and Pollard Factorization Algorithm

```
#define ll long long
typedef unsigned long long ull;
ull modmul(ull a, ull b, ull M) {
    ll ret = a * b - M * ull(1.L / M * a * b);
    return ret + M * (ret < 0) - M * (ret >= (ll)M);
}
ull modpow(ull b, ull e, ull mod) {
    ull ans = 1;
    for (; e; b = modmul(b, b, mod), e /= 2)
        if (e & 1) ans = modmul(ans, b, mod);
    return ans;
}

bool isPrime(ull n) {
    if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
    ull A[] = {
        2, 325, 9375,
        28178, 450775,
        9780504, 1795265022 },
        s = __builtin_ctzll(n - 1), d = n >> s;
    for (ull a : A) {  // ^ count t ra i l in g zeroes
        ull p = modpow(a % n, d, n), i = s;
        while (p != 1 && p != n - 1 && a % n && i--)
            p = modmul(p, p, n);
        if (p != n - 1 && i != s) return 0;
    }
    return 1;
}

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

// factors of very large N
vector<ull> factor(ull n) {
    if (n == 1) return {};
    if (isPrime(n)) return {n};
    ull x = pollard(n);
    auto l = factor(x), r = factor(n / x);
    l.insert(l.end(), all(r));
    return l;
}
```

7.3 Modular Arithmetic

7.3.1 Discrete Log

The following function returns any x such that $a^x \equiv b\ (mod\ m)$ in $O(\sqrt{m})$ time. It returns -1 if such x can not be found. [5]

```
int solve(int a, int b, int m) {
  // This reduction step is not needed
  // If a and m are relatively prime.
  a %= m, b %= m;
  int k = 1, add = 0, g;
  while ((g = gcd(a, m)) > 1) {
    if (b == k)
      return add;
    if (b % g)
      return -1;
    b /= g, m /= g, ++add;
    k = (k * 1ll * a / g) % m;
  }

  // Here the algorithm starts
  int n = sqrt(m) + 1;
  int an = 1;
  for (int i = 0; i < n; ++i)
    an = (an * 1ll * a) % m;

  unordered_map<int, int> vals;
  for (int q = 0, cur = b; q <= n; ++q) {
    vals[cur] = q;
    cur = (cur * 1ll * a) % m;
  }

  // if the gcd(a, m) = 1
  //   => cur = 1 AND add = 0
  for (int p = 1, cur = k; p <= n; ++p) {
    cur = (cur * 1ll * an) % m;
    if (vals.count(cur)) {
      int ans = n * p - vals[cur] + add;
      return ans;
    }
  }
  return -1;
}
```

7.3.2 Discrete Root

The problem of finding a discrete root is defined as follows. Given a prime n and two integers and a, k, find all x for which: $x^k \equiv a\ (mod\ n)$ [6]
Required functions:

- `powmod(a, x, m)` ... $a^x\ (mod\ m)$ in $O(log(n))$
- `gcd(a, b)` ... GCD of a and b
- `primitive_root(m)` ... The primitive root of m

```
// Find all integers x such that x^k = a (mod n)
void solve(int n, int k, int a) {
  if (a == 0) {
    puts("1\n0");
    return;
  }

  int g = primitive_root(n);

  // Baby-step giant-step discrete logarithm algorithm
  int sq = (int) sqrt (n + .0) + 1;
  vector<pair<int, int>> dec(sq);
  for (int i = 1; i <= sq; ++i)
    dec[i-1] = {powmod(g, i * sq * k % (n - 1), n), i};
  sort(dec.begin(), dec.end());
  int any_ans = -1;
  for (int i = 0; i < sq; ++i) {
    int my = powmod(g, i * k % (n - 1), n) * a % n;
    auto it = lower_bound(dec.begin(), dec.end(),
          make_pair(my, 0));
    if (it != dec.end() && it->first == my) {
      any_ans = it->second * sq - i;
      break;
    }
  }
  if (any_ans == -1) {
    puts("0");
    return;
  }

  // Print all possible answers
  int delta = (n-1) / gcd(k, n-1);
  vector<int> ans;
  for (int cur = any_ans % delta; cur < n-1; cur +=
        delta)
    ans.push_back(powmod(g, cur, n));
  sort(ans.begin(), ans.end());
  printf("%d\n", ans.size());
  for (int answer : ans)
    printf("%d ", answer);
}
```

7.3.3 Primitive Root

```
// Finds the primitive root modulo p
int primitive_root(int p) {
  vector<int> fact;
  int phi = p-1, n = phi;
  for (int i = 2; i * i <= n; ++i) {
    if (n % i == 0) {
      fact.push_back(i);
      while (n % i == 0)
        n /= i;
    }
  }
  if (n > 1)
    fact.push_back(n);

  for (int res = 2; res <= p; ++res) {
    bool ok = true;
    for (int factor : fact) {
      if (powmod(res, phi / factor, p) == 1) {
        ok = false;
        break;
      }
    }
    if (ok) return res;
  }
  return -1;
}
```

7.3.4 Factorial Mod Linear P

```
int factmod(int n, int p) {
  // Precompute in O(p)
  vector<int> f(p);
  f[0] = 1;
  for (int i = 1; i < p; i++)
    f[i] = f[i-1] * i % p;

  // Answer in O(log(n))
  int res = 1;
  while (n > 1) {
    if ((n/p) % 2)
      res = p - res;
    res = res * f[n%p] % p;
    n /= p;
  }
  return res;
}
```

7.3.5 Legendre's Formula

The exponent of p in the prime factorization [8] of $n!$ is:

$$\nu_p(n!) = \sum_{i=1}^{\infty} \left\lfloor \frac{n}{p^i} \right\rfloor$$

```
int multiplicity_factorial(int n, int p) {
  int count = 0;
  do { n /= p; count += n; } while (n);
  return count;
}
```

7.3.6 ModInt Structure

```
const int mod=998244353;
struct mi {
    int v;
    mi(){v=0;}
    mi(ll _v){v=int(-mod<=_v&&_v<mod?_v:_v%mod); if(v<0)v+=mod;}
    explicit operator int()const{return v;}
    friend bool operator==(const mi &a,const mi &b){return (a.v==b.v);}
    friend bool operator!=(const mi &a,const mi &b){return (a.v!=b.v);}
    friend bool operator<(const mi &a,const mi &b){return (a.v<b.v);}
    mi& operator+=(const mi &m){if((v+=m.v)>=mod)v-=mod; return *this;}
    mi& operator-=(const mi &m){if((v-=m.v)<0)v+=mod; return *this;}
    mi& operator*=(const mi &m){v=((ll)(v)*m.v)%mod; return *this;}
    mi& operator/=(const mi &m){return (*this)*=inv(m);}
    friend mi pow(mi a,ll e){mi r=1; for(;e;a*=a,e/=2)if(e&1)r*=a; return r;}
    friend mi inv(mi a){return pow(a,mod-2);}
    mi operator-()const{return mi(-v);}
    mi& operator++(){return (*this)+=1;}
    mi& operator--(){return (*this)-=1;}
    friend mi operator++(mi &a,int){mi t=a; ++a; return t;}
    friend mi operator--(mi &a,int){mi t=a; --a; return t;}
    friend mi operator+(mi a,const mi &b){return a+=b;}
    friend mi operator-(mi a,const mi &b){return a-=b;}
    friend mi operator*(mi a,const mi &b){return a*=b;}
    friend mi operator/(mi a,const mi &b){return a/=b;}
    friend istream& operator>>(istream &is,mi &m){ll _v; is >> _v; m=mi(_v); return is;}
    friend ostream& operator<<(ostream &os,const mi &m){os << m.v; return os;}
};
```

7.4 Binomial Coefficient

Precomputing binomial coefficients in $O(N^2)$.

```
const int maxn = ...;
int C[maxn + 1][maxn + 1];
C[0][0] = 1;
for (int n = 1; n <= maxn; ++n) {
    C[n][0] = C[n][n] = 1;
    for (int k = 1; k < n; ++k)
        C[n][k] = C[n - 1][k - 1] + C[n - 1][k];
}
```

Computing binomial coefficients in $O(\log M)$, where M is the modulo.

```
mi C(mi n, mi k) {
  return fact[n] / (fact[k] * fact[n-k]);
}
```

7.5 Inclusion-Exclusion Principle

A general example code to showcase the inclusion-exclusion principle.

```
for (int i = 1; i < (1 << n); i++) {
  vector<int> current;
  int bit_count = 0;
  for (int bit = 0; bit < n; bit++) {
    if ((1 << bit) & i) {
      // add the i-th element to
      // the data structure
      add(current, element[i])
      bit_count++;
    }
  }

  // depending on the parity of the bitcount,
  // calculate and modify the answer
  if (bit_count % 2 == 0) {
    ans -= calc(current);
  } else {
    ans += calc(current);
  }
}
```

7.6 Dynamic Programming - Sum over subsets

$$F_{mask} = \sum_{i \subseteq mask} A_i$$

Time Complexity: $O(N \cdot 2^N)$

```
for (int i = 0; i < (1 << N); ++i) F[i] = A[i];
for (int i = 0; i < N; ++i) {
  for (int mask = 0; mask < (1 << N); ++mask) {
   if(mask & (1<<i)) F[mask] += F[mask^(1<<i)];
  }
}
```

If we want for each subset to sum up the functions with inclusion-exclusion in mind, we can use the Mobius Transform.

$$\mu(f(s)) = \sum_{s' \subseteq s} (-1)^{|s \setminus s'|} f(s')$$

Time Complexity: $O(N \cdot 2^N)$

```
for (int i = 0; i < N; i++) {
  for (int mask = 0; mask < (1 << N); mask++) {
    if ((mask & (1 << i)) != 0) {
      f[mask] -= f[mask ^ (1 << i)];
    }
  }
}
for(int mask = 0; mask < (1 << N); mask++) {
  zinv[mask] = mu[mask] = f[mask];
}
```

7.7 Fractional Binary Search

7.8 Matrix Exponentiation

7.9 Phi function

7.10 Gaussian Elimination

7.11 Determinant

7.12 K-th Permutation

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

struct KthPermutation {
    #define ull unsigned long long
    ull factorial[21];

    KthPermutation() {
        factorial[0] = 1;
        for (int i = 1; i <= 20; i++) {
            factorial[i] = factorial[i-1] * i;
        }
    }

    // O(n^2)
    vector<int> kth(int n, ull k) {
        vector<int> v(n);
        iota(v.begin(), v.end(), 1);
        vector<int> result;

        do {
            ull count = factorial[v.size() - 1];
            ull selected = (k - 1) / count;
            result.push_back(v[selected]);
            v.erase(v.begin() + selected);
            k -= (count * selected);
        } while (!v.empty());

        return result;
    }
};

int main() {
    KthPermutation util;

    for (int i = 1; i <= 24; ++i) {
        vector<int> permutation = util.kth(4, i);
        for (auto element: permutation) {
            cout << element << " ";
        } cout << endl;
    }
}
```

7.13 Berlekamp-Messay Algorithm

```
/*
        Finds the smallest linear recurrence for the sequence given as input.
    For example, for the sequence
        0 1 1 2 3 5 8 13
        The recursion will be: f(x) = 1 * f(x-1) + 1 * f(x-2)
*/

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

struct BerlekampMassey {
    const int M = 1e9 + 7;
    ll fast_pow(ll a, ll b) {
        ll ans = 1;

        while (b) {
            if (b & 1) ans = ans * a % M;
            b >>= 1;
            a = a * a % M;
        }
        return ans;
    }
    inline ll inv(ll x) { return fast_pow(x, M - 2); }
    // 'vec' is the array with the first vec.size() recurrence numbers
    // 'cur' is the array of multiplied coefficients,
    //  and must be interpreted as
    //  vec[i] = cur [0] * vec [i-1] + cur [1] * vec [i-2] + ...
    vector<ll> solve(vector<ll> vec) {
        const int n = vec.size();

        // S[i] = calculated recurrence for the prefix i,
        // which evaluates to 0 for all indices from 0 to i
        // and evaluates to 1 for i+1
        vector<vector<ll> > S(n, vector<ll>());

        // current recurrence
        vector<ll> cur = {0};
        S[0] = {1};
        for (int i = 1; i < n; ++i) {
            ll res = 0;
            assert((int)cur.size() <= i);
            for (int j = 0; j < cur.size(); ++j) {
                res = (res + cur[j] * vec[i - 1 - j] % M) % M;
            }

            if (vec[i] == res) continue;    // it worked out
            ll v = (res - vec[i] + M) % M;  // v such that vec[i] + v = res

            // Calculate S[i]
            S[i].push_back(M - 1 * inv(v) % M);
            for (ll x : cur) S[i].push_back(x * inv(v) % M);

            // Recalculate cur
            int k = 0;
            for (int j = 0; j < i; ++j) {
                if (S[j].size() == 0) continue;
                if (S[j].size() + i - 1 - j <= S[k].size() + i - 1 - k) k = j;
            }

            vector<ll> aux(max(int(S[k].size()) + i - 1 - k, (int)cur.size()));
            for (int j = 0; j < aux.size(); ++j) {
                ll x = j < (i - 1 - k) ? 0 : S[k][j - (i - 1 - k)];
                aux[j] = ((vec[i] - res + M) * x % M) +
                         (j < cur.size() ? cur[j] : 0);
                aux[j] %= M;
            }

            cur = aux;
        }

        return cur;
    }
};

int main() {
    int n;
    cin >> n;
    vector<ll> v(n);
    for (int i = 0; i < n; i++) cin >> v[i];
    BerlekampMassey bm;              // for 0 1 3 10 33 returns 3 1 0
    vector<ll> coeff = bm.solve(v);  //   f(n) = 3 * f(n-1) + f(n-2)
    for (int i = 0; i < coeff.size(); i++) cout << coeff[i] << " ";
    cout << endl;
    return 0;
}
```

7.14 Chinese Remainder Theorem

```cpp
// Chinese Remainder Theorem with
// generalization on non-coprime moduli
// https://www.programmersought.com/article/34462894954/

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

template<typename T>
T gcd(T a, T b) {
    return (a == 0) ? b : gcd(b % a, a);
}

template<typename T>
pair<T, pair<T,T> > reverseGCD(T a, T b) {
    // returns (g,(x,y))
    pair<T,pair<T,T>> ret;
    if (a == 0) {
        ret.first = b;
        ret.second.first = 0;
        ret.second.second = 1;
    } else {
        T g, x, y;
        pair<T,pair<T, T>> temp = reverseGCD(b%a, a);
        g = temp.first;
        x = temp.second.first;
        y = temp.second.second;
        ret.first = g;
        ret.second.first = y - (b/a)*x;
        ret.second.second = x;
    }
    return ret;
}

template<typename T>
T CRT(vector<T> n, vector<T> a) {
    // n periods, a is the remainder

    T Di = n[0], remainder = a[0];
    int k = n.size();

    for (int i = 1; i < k; ++i) {
        T di = n[i];
        T bi = a[i];
        auto rGCD = reverseGCD(Di, di);
        T x = rGCD.second.first, y = rGCD.second.second;

        T c = bi - remainder;

        if (c % rGCD.first) // indicates no solution
            return (T)(-1);

        T D = di / rGCD.first;
        remainder += Di * (((c / rGCD.first * x)%D+D)%D);

        Di *= D;
    }

    // represents the remainder of all zeros
    if (!remainder) {
        remainder = 1;
        for(int i = 0; i < k; ++i) {
            remainder = remainder * n[i] / gcd(remainder, n[i]);
        }
    }

    return remainder;
}
```

7.15 Mobius Inversion

7.16 GCD & LCM Convolution

```cpp
// GCD/LCM Convolution
// Given seq a[i] and b[i], find c[i] such that
// c[k] = sum(a[i] * b[j]) where gcd(i, j) = k
// all MOD, N <= 10^6, a[i], b[i] <= MOD

// For LCM follow comments

#include <bits/stdc++.h>
#define ll long long
using namespace std;
constexpr int MOD = 998'244'353;

void zeta(vector<ll>& a) {
  int n = a.size() - 1;
  for (int i = 1; i <= n; ++i) { // reverse loop
    for (int j = 2; j <= n / i; ++j) {
      a[i] += a[i * j];             // swap(i, i*j)
      if (a[i] >= MOD) a[i] -= MOD; // swap(i, i*j)
    }
  }
}

void mobius(vector<ll>& a) {
  int n = a.size() - 1;
  for (int i = n; i >= 1; --i) { // reverse loop
    for (int j = 2; j <= n / i; ++j) {
      a[i] -= a[i * j];             // swap(i, i*j)
      if (a[i] < 0) a[i] += MOD;    // swap(i, i*j)
    }
  }
}

int main() {
  int n;
  scanf("%d", &n);

  vector<ll> a(n + 1), b(n + 1);
  for (int i = 1; i <= n; ++i) scanf("%lld", &a[i]);
  for (int i = 1; i <= n; ++i) scanf("%lld", &b[i]);

  zeta(a);
  zeta(b);
  vector<ll> c(n + 1);
  for (int i = 1; i <= n; ++i) c[i] = a[i] * b[i] % MOD;
  mobius(c);

  for (int i = 1; i <= n; ++i) {
    if (i > 1) printf(" ");
    printf("%lld", c[i]);
  }
  printf("\n");
  return 0;
}
```

7.17 Fast Fourier Transform

```cpp
#include <bits/stdc++.h>
using namespace std;                // A = 1 2 3
typedef long long ll;               // B = 1 2 3
typedef complex<double> cd;         // A * B = result = 1 4 10 12 9
                                    // (1 + 2x + 3x^2) * (1 + 2x + 3x^2)
const ll Max = 1e6 + 10;            //  1 + 2x + 3x^2 +
ll bound, logBound;                 //      2x + 4x^2 + 6x^3
const double pi = 4 * atan(1.0);    //           3x^2 + 6x^3 + 9x^2
cd root[Max], arrA[Max], arrB[Max]; //  1   4    10     12      9
ll perm[Max], prod[Max];

void fft(cd *arr) {
    for (ll i = 0; i < bound; i++) {
        if (i < perm[i]) {
            swap(arr[i], arr[perm[i]]);
        }
    }
    for (ll len = 1; len < bound; len *= 2) {
        for (ll pos = 0; pos < bound; pos += 2 * len) {
            for (ll i = 0; i < len; i++) {
                cd x = arr[pos + i],
                   y = arr[pos + i + len] * root[bound / len / 2 * i];
                arr[pos + i] = x + y;
                arr[pos + i + len] = x - y;
            }
        }
    }
}

void preCalc() {
    ll hb = -1;
    root[0] = 1;
```

```
    double angle = 2 * pi / bound;
    for (ll i = 1; i < bound; i++) {
        if ((i & (i - 1)) == 0) hb++;
        root[i] = cd(cos(angle * i), sin(angle * i));
        perm[i] = perm[i ^ (1 << hb)] + (1 << (logBound - hb - 1));
    }
}

void mult(vector<ll> &a, vector<ll> &b, vector<ll> &c) {
    logBound = 0;
    while ((1 << logBound) < a.size() || (1 << logBound) < b.size()) logBound++;
    logBound++;
    bound = (1 << logBound);
    preCalc();
    for (ll i = 0; i < a.size(); i++) {
        arrA[i] = cd(a[i], 0);
    }
    for (ll i = a.size(); i < bound; i++) {
        arrA[i] = cd(0, 0);
    }
    for (ll i = 0; i < b.size(); i++) {
        arrB[i] = cd(b[i], 0);
    }
    for (ll i = b.size(); i < bound; i++) {
        arrB[i] = cd(0, 0);
    }
    fft(arrA);
    fft(arrB);
    for (ll i = 0; i < bound; i++) {
        arrA[i] *= arrB[i];
    }
    fft(arrA);
    reverse(arrA + 1, arrA + bound);
    c.resize(bound);
    for (ll i = 0; i < bound; i++) {
        arrA[i] /= bound;
        ll temp =
            (arrA[i].real() > 0 ? arrA[i].real() + .5 : arrA[i].real() - .5);
        c[i] = temp;
    }
    while (c.size() && c.back() == 0) c.pop_back();
}

int main() {
    int n;
    cin >> n;
    vector<ll> a(n), b(n), result(2 * n + 100);
    for (int i = 0; i < n; i++) cin >> a[i];
    for (int i = 0; i < n; i++) cin >> b[i];
    mult(a, b, result);
    for (int i = 0; i < result.size(); i++) {
        cout << result[i] << " ";
    }
    cout << endl;
    return 0;
}
```

7.18 Number Theoretic Transform

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

const int mod=998244353;
const int primitive_root=3;

struct mi {};

void ntt(vector<mi> &a) {
    int n=a.size(),L=31-__builtin_clz(n);
    vector<int> rev(n);
    for(int i=0;i<n;i++) rev[i]=(rev[i/2]+((i&1)<<L))/2;
    for(int i=0;i<n;i++) if(i<rev[i]) swap(a[i],a[rev[i]]);
    static vector<mi> rt(2,1);
    for(static int k=2,s=2;k<n;k*=2,s++) {
        rt.resize(n);
        mi z=pow(mi(primitive_root),mod>>s);
        for(int i=k;i<2*k;i++) rt[i]=rt[i/2]*((i&1)?z:1);
    }
    for(int k=1;k<n;k*=2) {
        for(int i=0;i<n;i+=2*k) {
            for(int j=0;j<k;j++) {
                mi z=rt[j+k]*a[i+j+k];
                a[i+j+k]=a[i+j]-z;
                a[i+j]+=z;
            }
        }
    }
}

vector<mi> conv(vector<mi> &a, vector<mi> &b) {
    int sa=a.size(),sb=b.size();
    if(sa==0||sb==0) return {};
    int n=1<<(32-__builtin_clz(sa+sb-2));
    mi inv=1/mi(n);
    vector<mi> f(a),g(b),h(n);
    f.resize(n); g.resize(n);
    ntt(f); ntt(g);
    for(int i=0;i<n;i++) h[(-i)&(n-1)]=(f[i]*g[i]*inv);
    ntt(h);
    h.resize(sa+sb-1);
    return h;
}

vector<mi> multipoly(vector<vector<mi>> v) {
    auto cmp=[&](const vector<mi> &a,const vector<mi> &b){return (a.size()>b.size());};
    priority_queue<vector<mi>,vector<vector<mi>>,decltype(cmp)> q(cmp);
    for(auto &u:v) q.push(u);
    while(q.size()>=2) {
        auto a=q.top();
        q.pop();
        auto b=q.top();
        q.pop();
        q.push(conv(a,b));
    }
    return q.top();
}
```

7.19 Fast Subset Transform

7.20 More Operations on Finite Polynomials

```
//
// Polynomial with integer coefficient (mod M)
//
// Implemented routines:
//   1) addition
//   2) subtraction
//   3) multiplication (naive O(n^2), Karatsuba O(n^1.5..), FFT O(n log n))
//   4) division (naive O(n^2), Newton O(M(n)))
//   5) gcd
//   6) multipoint evaluation (divide conquer: O(M(n) log |X|))
//   7) interpolation (naive O(n^2), divide conquer O(M(n) log n))
//   8) polynomial shift (naive, fast)
//
//   *) n! mod M in O(n^{1/2} log n) time
// >>> p(x) = p[0] + p[1] x + ... + p[n-1] x^n-1
//
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef vector<ll> poly;

#define fst first
#define snd second
#define all(c) ((c).begin()), ((c).end())
#define TEST(s) if (!(s)) { cout << __LINE__ << " " << #s << endl; exit(-1); }

ll add(ll a, ll b, ll M) { // a + b (mod M)
  return (a += b) >= M ? a - M : a;
}
ll sub(ll a, ll b, ll M) { // a - b (mod M)
  return (a -= b) < 0 ? a + M : a;
}
ll mul(ll a, ll b, ll M) { // a * b (mod M)
  ll r = a*b - (ll)((long double)(a)*b/M+.5)*M;
  return r < 0 ? r + M: r;
}
ll div(ll a, ll b, ll M) { // solve b x == a (mod M)
  ll u = 1, x = 0, s = b, t = M;
  while (s) { // extgcd for b x + M s = t
    ll q = t / s;
    swap(x -= u * q, u);
    swap(t -= s * q, s);
  }
  if (a % t) return -1; // infeasible
  return mul(x < 0 ? x + M : x, a / t, M); // b (xa/t) == a (mod M)
}
ll pow(ll a, ll b, ll M) {
  ll x = 1;
  for (; b > 0; b >>= 1) {
    if (b & 1) x = (a * x) % M;
```

```
    a = (a * a) % M;
  }
  return x;
}

poly add(poly p, const poly &q, ll M) {
  if (p.size() < q.size()) p.resize(q.size());
  for (int i = 0; i < q.size(); ++i)
    p[i] = add(p[i], q[i], M);
  while (!p.empty() && !p.back()) p.pop_back();
  return p;
}
poly sub(poly p, const poly &q, ll M) {
  if (p.size() < q.size()) p.resize(q.size());
  for (int i = 0; i < q.size(); ++i)
    p[i] = sub(p[i], q[i], M);
  while (!p.empty() && !p.back()) p.pop_back();
  return p;
}

// naive multiplication in O(n^2)
poly mul_n(const poly &p, const poly &q, ll M) {
  if (p.empty() || q.empty()) return {};
  poly r(p.size() + q.size() - 1);
  for (int i = 0; i < p.size(); ++i)
    for (int j = 0; j < q.size(); ++j)
      r[i+j] = add(r[i+j], mul(p[i], q[j], M), M);
  while (!r.empty() && !r.back()) r.pop_back();
  return r;
}
// naive division (long division) in O(n^2)
pair<poly,poly> divmod_n(poly p, poly q, ll M) {
  poly u(p.size() - q.size() + 1);
  ll inv = div(1, q.back(), M);
  for (int i = u.size()-1; i >= 0; --i) {
    u[i] = mul(p.back(), inv, M);
    for (int j = 0; j < q.size(); ++j)
      p[j+p.size()-q.size()] = sub(p[j+p.size()-q.size()], mul(q[j], u[i], M), M);
    p.pop_back();
  }
  return {u, p};
}
// Karatsuba multiplication; this works correctly for M in [long long]
poly mul_k(poly p, poly q, ll M) {
  int n = max(p.size(), q.size()), m = p.size() + q.size() - 1;
  for (int k: {1,2,4,8,16}) n |= (n >> k); ++n; // n is power of two
  p.resize(n); q.resize(n);
  poly r(6*n);
  function<void(ll*, ll*, int, ll*)> rec = [&](ll *p0, ll *q0, int n, ll *r0) {
    if (n <= 4) { // 4 is the best threshold
      fill_n(r0, 2*n, 0);
      for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
          r0[i+j] = add(r0[i+j], mul(p0[i], q0[j], M), M);
      return;
    }
    ll *p1=p0+n/2,*q1=q0+n/2,*r1=r0+n/2,*r2=r0+n,*u=r0+5*n,*v=u+n/2,*w=r0+2*n;
    for (int i = 0; i < n/2; ++i) {
      u[i] = add(p0[i], p1[i], M);
      v[i] = add(q0[i], q1[i], M);
    }
    rec(p0, q0, n/2, r0);
    rec(p1, q1, n/2, r2);
    rec( u,  v, n/2, w);
    for (int i = 0; i < n; ++i) w[i] = sub(w[i], add(r0[i], r2[i], M), M);
    for (int i = 0; i < n; ++i) r1[i] = add(r1[i], w[i], M);
  }; rec(&p[0], &q[0], n, &r[0]);
  r.resize(m);
  return r;
}

// FFT-based multiplication: this works correctly for M in [int]
// assume: size of a/b is power of two, mod is predetermined
template <int mod, int sign>
void fmt(vector<ll>& x) {
  const int n = x.size();
  int h = pow(3, (mod-1)/n, mod);
  if (sign < 0) h = div(1, h, mod);
  for (int i = 0, j = 1; j < n-1; ++j) {
    for (int k = n >> 1; k > (i ^= k); k >>= 1);
    if (j < i) swap(x[i], x[j]);
  }
  for (int m = 1; m < n; m *= 2) {
    ll w = 1, wk = pow(h, n / (2*m), mod);
    for (int i = 0; i < m; ++i) {
      for (int s = i; s < n; s += 2*m) {
        ll u = x[s], d = x[s + m] * w % mod;
        if ((x[s] = u + d) >= mod) x[s] -= mod;
        if ((x[s + m] = u - d) < 0) x[s + m] += mod;
      }
      w = w * wk % mod;
    }
  }
  if (sign < 0) {
    ll inv = div(1, n, mod);
    for (auto &a: x)
      a = a * inv % mod;
  }
}
// assume: size of a/b is power of two, mod is predetermined
template <int mod>
vector<ll> conv(vector<ll> a, vector<ll> b){
  fmt<mod,+1>(a); fmt<mod,+1>(b);
  for (int i = 0; i < a.size(); ++i)
    a[i] = a[i] * b[i] % mod;
  fmt<mod,-1>(a);
  return a;
}
// general convolution where mod < 2^31.
vector<ll> conv(vector<ll> a, vector<ll> b, ll mod){
  int n = a.size() + b.size() - 1;
  for (int k: {1,2,4,8,16}) n |= (n >> k); ++n;
  a.resize(n); b.resize(n);
  const int A = 167772161, B = 469762049, C = 1224736769, D = (ll)(A) * B % mod;
  vector<ll> x = conv<A>(a,b), y = conv<B>(a,b), z = conv<C>(a,b);
  for (int i = 0; i < x.size(); ++i) {
    ll X = (y[i] - x[i]) * 104391568;
    if ((X %= B) < 0) X += B;
    ll Y = (z[i] - (x[i] + A * X) % C) * 721017874;
    if ((Y %= C) < 0) Y += C;
    x[i] += A * X + D * Y;
    if ((x[i] %= mod) < 0) x[i] += mod;
  }
  x.resize(n);
  return x;
}
poly mul(poly p, poly q, ll M) {
  poly pq = conv(p, q, M);
  pq.resize(p.size() + q.size() - 1);
  while (!pq.empty() && !pq.back()) pq.pop_back();
  return pq;
}

// Newton division: O(M(n)); M is the complexity of multiplication
// fast when FFT multiplication is used
//
// Note: complexity = M(n) + M(n/2) + M(n/4) + ... <= 2 M(n).
pair<poly,poly> divmod(poly p, poly q, ll M) {
  if (p.size() < q.size()) return { {}, p };
  reverse(all(p)); reverse(all(q));
  poly t = {div(1, q[0], M)};
  if (t[0] < 0) return { {}, {} }; // infeasible
  for (int k = 1; k <= 2*(p.size()-q.size()+1); k *= 2) {
    poly s = mul(mul(t, q, M), t, M);
    t.resize(k);
    for (int i = 0; i < k; ++i)
      t[i] = sub(2*t[i], s[i], M);
  }
  t.resize(p.size() - q.size() + 1);
  t = mul(t, p, M);
  t.resize(p.size() - q.size() + 1);
  reverse(all(t)); reverse(all(p)); reverse(all(q));
  while (!t.empty() && !t.back()) t.pop_back();
  return {t, sub(p, mul(q, t, M), M) };
}
// polynomial GCD: O(M(n) log n);
poly gcd(poly p, poly q, ll M) {
  for (; !p.empty(); swap(p, q = divmod(q, p, M).snd));
  return p;
}

// value of p(x)
ll eval(poly p, ll x, ll M) {
  ll ans = 0;
  for (int i = p.size()-1; i >= 0; --i)
    ans = add(mul(ans, x, M), p[i], M);
  return ans;
};
//
// faster multipoint evaluation
// fast if |x| >= 10000.
//
// algo:
//   evaluate(p, {x[0], ..., x[n-1]})
//   = evaluate(p mod (X-x[0])...(X-x[n/2-1]), {x[0], ..., x[n/2-1]}),
//   + evaluate(p mod (X-x[n/2])...(X-x[n-1]), {x[n/2], ..., x[n-1]}),
//
// f(n) = 2 f(n/2) + M(n) ==> O(M(n) log n)
//
vector<ll> evaluate(poly p, vector<ll> x, ll M) {
  vector<poly> prod(8*x.size()); // segment tree
  function<poly(int,int,int)> run = [&](int i, int j, int k) {
    if (i   == j) return prod[k] = (poly){1};
    if (i+1 == j) return prod[k] = (poly){M-x[i], 1};
    return prod[k] = mul(run(i,(i+j)/2,2*k+1), run((i+j)/2,j,2*k+2), M);
  }; run(0, x.size(), 0);
  vector<ll> y(x.size());
  function<void(int,int,int,poly)> rec = [&](int i, int j, int k, poly p) {
    if (j - i <= 8) {
      for (; i < j; ++i) y[i] = eval(p, x[i], M);
    } else {
      rec(i, (i+j)/2, 2*k+1, divmod(p, prod[2*k+1], M).snd);
      rec((i+j)/2, j, 2*k+2, divmod(p, prod[2*k+2], M).snd);
    }
  }; rec(0, x.size(), 0, p);
  return y;
}

poly interpolate_n(vector<ll> x, vector<ll> y, ll M) {
  int n = x.size();
  vector<ll> dp(n+1);
  dp[0] = 1;
  for (int i = 0; i < n; ++i) {
    for (int j = i; j >= 0; --j) {
      dp[j+1] = add(dp[j+1], dp[j], M);
      dp[j] = mul(dp[j], M - x[i], M);
    }
  }
  poly r(n);
  for (int i = 0; i < n; ++i) {
    ll den = 1, res = 0;
    for (int j = 0; j < n; ++j)
      if (i != j) den = mul(den, sub(x[i], x[j], M), M);
    den = div(1, den, M);
```

```
    for (int j = n-1; j >= 0; --j) {
      res = add(dp[j+1], mul(res, x[i], M), M);
      r[j] = add(r[j], mul(res, mul(den, y[i], M), M), M);
    }
  }
  while (!r.empty() && !r.back()) r.pop_back();
  return r;
}
//
// faster algo to find a poly p such that
//   p(x[i]) = y[i]  for each i
//
// see http://people.mpi-inf.mpg.de/~csaha/lectures/lec6.pdf
//
poly interpolate(vector<ll> x, vector<ll> y, ll M) {
  vector<poly> prod(8*x.size()); // segment tree
  function<poly(int,int,int)> run = [&](int i, int j, int k) {
    if (i   == j) return prod[k] = (poly){1};
    if (i+1 == j) return prod[k] = (poly){M-x[i], 1};
    return prod[k] = mul(run(i,(i+j)/2,2*k+1), run((i+j)/2,j,2*k+2), M);
  }; run(0, x.size(), 0); // preprocessing in O(n log n) time

  poly H = prod[0]; // newton polynomial
  for (int i = 1; i < H.size(); ++i) H[i-1] = mul(H[i], i, M);
  do H.pop_back(); while (!H.empty() && !H.back());

  vector<ll> u(x.size());
  function<void(int,int,int,poly)> rec = [&](int i, int j, int k, poly p) {
    if (j - i <= 8) {
      for (; i < j; ++i) u[i] = eval(p, x[i], M);
    } else {
      rec(i, (i+j)/2, 2*k+1, divmod(p, prod[2*k+1], M).snd);
      rec((i+j)/2, j, 2*k+2, divmod(p, prod[2*k+2], M).snd);
    }
  }; rec(0, x.size(), 0, H); // multipoint evaluation

  for (int i = 0; i < x.size(); ++i) u[i] = div(y[i], u[i], M);

  function<poly(int,int,int)> f = [&](int i, int j, int k) {
    if (i   >= j) return poly();
    if (i+1 == j) return (poly){u[i]};
    return add(mul(f(i,(i+j)/2,2*k+1), prod[2*k+2], M),
               mul(f((i+j)/2,j,2*k+2), prod[2*k+1], M), M);
  };
  return f(0, x.size(), 0);
}

//
// return p(x+a)
//
poly shift_n(poly p, ll a, ll M) {
  poly q(p.size());
  for (int i = p.size()-1; i >= 0; --i) {
    for (int j = p.size()-i-1; j >= 1; --j)
      q[j] = add(mul(q[j], a, M), q[j-1], M);
    q[0] = add(mul(q[0], a, M), p[i], M);
  }
  return q;
}

//
// faster algorithm for computing p(x + a)
//
// fast if n >= 4096
// algo: p(x+a) = p_h(x) (x+a)^m + q_h(x)
// cplx: preproc: O(M(n))
//       div-con: O(M(n) log n)
//
poly shift(poly p, ll a, ll M) {
  vector<poly> pow(p.size());
  pow[0] = {1}; pow[1] = {a,1};
  int m = 2;
  for (; m < p.size(); m *= 2)
    pow[m] = mul(pow[m/2], pow[m/2], M);
  function<poly(poly,int)> rec = [&](poly p, int m) {
    if (p.size() <= 1) return p;
    while (m >= p.size()) m /= 2;
    poly q(p.begin() + m, p.end());
    p.resize(m);
    return add(mul(rec(q, m), pow[m], M), rec(p, m), M);
  };
  return rec(p, m);
}

//
// overperform when n >= 134217728 lol
//
ll factmod(ll n, ll M) {
  if (n <= 1) return 1;
  ll m = sqrt(n);
  function<poly(int,int)> get = [&](int i, int j) {
    if (i   == j) return poly();
    if (i+1 == j) return (poly){i,1};
    return mul(get(i, (i+j)/2), get((i+j)/2, j), M);
  };
  poly p = get(0, m); // = x (x+1) (x+2) ... (x+(m-1))
  vector<ll> x(m);
  for (int i = 0; i < m; ++i) x[i] = 1 + i * m;
  vector<ll> y = evaluate(p, x, M);
  ll fac = 1;
  for (int i = 0; i < m; ++i)
    fac = mul(fac, y[i], M);
  for (ll i = m*m+1; i <= n; ++i)
    fac = mul(fac, i, M);
  return fac;
}
ll factmod_n(ll n, ll M) {
  ll fac = 1;
  for (ll k = 1; k <= n; ++k)
    fac = mul(k, fac, M);
  return fac;
}
ll factmod_p(ll n, ll M) { // only works for prime M
  ll fac = 1;
  for (; n > 1; n /= M) {
    fac = mul(fac, (n / M) % 2 ? M - 1 : 1, M);
    for (ll i = 2; i <= n % M; ++i)
      fac = mul(fac, i, M);
  }
  return fac;
}

bool TEST_DIVMOD() {
    ll MOD = 1000000007;
    int n;
    cin >> n;
    int b;
    cin >> b;
    int m = b + n;
    n++;
    m++;
    poly p(m), q(n);
    for (int i = 0; i < q.size(); ++i) cin >> q[i];
    for (int i = 0; i < p.size(); ++i) cin >> p[i];
    auto pq2 = divmod_n(p, q, MOD);
    for (int i = 0; i < pq2.first.size(); i++) {
        if (pq2.first[i] > 1e6) {
            pq2.first[i] -= MOD;
        }
        cout << pq2.first[i] << " ";
    }
    cout << endl;

    return true;
}

int main() {
  TEST_DIVMOD();
}
```

7.21 Game Theory, Nim Game

7.22 Simplex

7.23 Miscellaneous Stuff

7.23.1 Gray Code

Gray code is a binary numeral system where two successive values differ in only one bit. [10] For example, the sequence of Gray codes for 3-bit numbers is: 000, 001, 011, 010, 110, 111, 101, 100...

```
int g(int n) {
  return n ^ (n >> 1);
}

int rev_g(int g) {
  int n = 0;
  for (; g; g >>= 1)
    n ^= g;
  return n;
}
```

7.23.2 Lemmas

Lemma 1 $1 + 3 + 5 + \cdots + (2n - 1) = n^2$

Lemma 2 $\displaystyle\sum_{i=1}^{n} i \cdot i! = (n+1)! - 1$

Lemma 3 $2^n < n!\ ,\ n > 3$

Lemma 4 $|a_1 + a_2 + \cdots + a_n| \leq |a_1| + |a_2| + \cdots + |a_n|$, *for any real numbers* $a_1, a_2, \ldots a_n$

Lemma 5 *For any array* $(a_1, a_2, \ldots a_n)$*, there exist* l *and* r*, such that* $1 \leq l < r \leq n$ *and* $\displaystyle\sum_{i=l}^{r} a_i \equiv 0$ *(mod n).*

8 Hashing

8.1 Polynomial Hashing

8.2 Fenwick Tree on Hashes

8.3 Hashing Rooted Trees for Isomorphism

In a Chinese blog [14] the following technique for checking rooted trees for isomorphism using hashing is described. Let $s(a)$ be the rooted subtree at vertex a and $v_1, v_2 \ldots v_k$ are children of vertex a. Then we will define a isomorphic hash function as follows:

$$h(s(a)) = 1 + \sum_{i=1}^{k} f(h(s(v_i)))$$

The function f is defined as follows:

```
const ll HB = 1237123, HS = 19260817;
ll h(ll x) {
  return x * x * x * HB + HS;
}
ll f(ll x) {
  return h(x & ((1 << 31) - 1)) + h(x >> 31);
}
```

Note. HB and HS are constants, which can be changed if the hash is seen to collide.

It can be proved that if f is a random function, the expected number of collisions of such a hash under natural overflow is no more than $O(\frac{n^2}{2^w})$. TODO: Proof this.

8.4 Nimber Field

The Nim product $a \otimes b$ is an operation defined as follows:

$$a \otimes b = mex\{(a\prime \otimes b) \otimes (a \otimes b\prime) \otimes (a\prime \otimes b\prime) | a\prime < a, b\prime < b\}$$

If ordinarily in rolling hashes, we have the following structure:

$$R(A) = \left(\sum_{i=1}^{N} A_i \cdot x^{(N-i)} \right) \pmod p$$

In the Nimber field the structure remains the same, but the operations change.

$$R(A) = \left(XOR_{i=1}^{N} \left[A_i \otimes x^{(N-i)} \right] \right)$$

Each query gives you integers a, b, c, d, e, and f, each between 1 and N, inclusive. These integers satisfy $a \leq b$, $c \leq d$, $e \leq f$, and $b - a = d - c$. If $S(A(a,b), A(c,d))$ is strictly lexicographically smaller than $A(e,f)$, print Yes; otherwise, print No.

```
using u64 = unsigned long long;
constexpr int N_MAX = 1e6 + 10;
int N, Q;
u64 A[N_MAX], pw[N_MAX], hs[N_MAX], basis, small[256][256];
template <bool is_pre = false>
u64 nim_product(u64 a, u64 b, int p = 64) {
  if (min(a, b) <= 1) return a * b;
  if (!is_pre and p <= 8) return small[a][b];
  p >>= 1;
  u64 a1 = a >> p, a2 = a & ((1ull << p) - 1);
  u64 b1 = b >> p, b2 = b & ((1ull << p) - 1);
  u64 c = nim_product<is_pre>(a1, b1, p);
  u64 d = nim_product<is_pre>(a2, b2, p);
  u64 e = nim_product<is_pre>(a1 ^ a2, b1 ^ b2, p);
  return nim_product<is_pre>(c, 1uLL << (p - 1), p) ^ d ^ ((d ^ e) << p);
}
void init() {
  for (int i = 0; i < 256; i++)
    for (int j = 0; j < 256; j++) small[i][j] = nim_product<true>(i, j, 8);
  pw[0] = 1, hs[0] = basis = 0;
  mt19937_64 rng(time(NULL));
  basis = rng();
  for (int i = 1; i <= N; i++) {
    pw[i] = nim_product(pw[i - 1], basis);
    hs[i] = nim_product(hs[i - 1], basis) ^ A[i - 1];
  }
}
u64 get(int l, int r) { return nim_product(hs[l], pw[r - l]) ^ hs[r]; }
void send(int flag) { printf(flag ? "Yes\n" : "No\n"); }
int main() {
  scanf("%d %d", &N, &Q);
  for (int i = 0; i < N; i++) scanf("%llu", &(A[i]));
  init();
  while (Q--) {
    int a, b, c, d, e, f;
    scanf("%d %d %d %d %d %d", &a, &b, &c, &d, &e, &f);
    --a, --c, --e;
    int l = 0, h = min(f - e, b - a) + 1;
    while (l + 1 < h) {
      int m = (l + h) / 2;
      ((get(a, a + m) ^ get(c, c + m) ^ get(e, e + m)) ? h : l) = m;
    }
    send(e + l != f and (a + l == b or (A[a + l] ^ A[c + l]) < A[e + l]));
  }
}
```

9 Geometry

9.1 2D Geometry

9.1.1 Helper Functions

```cpp
const double PI = acos(-1.0);

// implementation note: use EPS only in this function
// usage note: check sign(x) < 0, sign(x) > 0, or sign(x) == 0
const double EPS = 1e-8;
int sign(double x) {
  if (x < -EPS) return -1;
  if (x > +EPS) return +1;
  return 0;
}

using real = long double;
struct point {
  real x, y;
  point &operator+=(point p) { x += p.x; y += p.y; return *this; }
  point &operator-=(point p) { x -= p.x; y -= p.y; return *this; }
  point &operator*=(real a)     { x *= a;   y *= a;   return *this; }
  point &operator/=(real a)     { return *this *= (1.0/a); }
  point operator-() const    { return {-x, -y}; }
  bool operator<(point p) const {
    int s = sign(x - p.x);
    return s ? s < 0 : sign(y - p.y) < 0;
  }
};
bool operator==(point p, point q) { return !(p < q) && !(q < p); }
bool operator!=(point p, point q) { return p < q || q < p; }
bool operator<=(point p, point q) { return !(q < p); }
point operator+(point p, point q) { return p += q; }
point operator-(point p, point q) { return p -= q; }
point operator*(real a, point p) { return p *= a; }
point operator*(point p, real a) { return p *= a; }
point operator/(point p, real a) { return p /= a; }
real dot(point p, point q) { return p.x*q.x+p.y*q.y; }
real cross(point p, point q) { return p.x*q.y-p.y*q.x; } // left turn > 0
real norm2(point p) { return dot(p,p); }
point orth(point p) { return {-p.y, p.x}; }
real norm(point p) { return sqrt(dot(p,p)); }
real arg(point p) { return atan2(p.y, p.x); }
real arg(point p, point q){ return atan2(cross(p,q), dot(p,q)); }

istream &operator>>(istream &is, point &p) { is>>p.x>>p.y;return is; }
ostream &operator<<(ostream &os, const point &p) { os<<"("<<p.x<<","<<p.y<<")"; return os; }
typedef vector<point> polygon;


// exact comparison by polar angle
// usage: sort(all(ps), polar_angle(origin, direction));
struct polar_angle {
  const point o;
  const int s; // +1 for ccw, -1 for cw
  polar_angle(point p = {0,0}, int s = +1) : o(p), s(s) { }
  int quad(point p) const {
    for (int i = 1; i <= 4; ++i, swap(p.x = -p.x, p.y))
      if (p.x > 0 && p.y >= 0) return i;
    return 0;
  }
  bool operator()(point p, point q) const {
    p = p - o; q = q - o;
    if (quad(p) != quad(q)) return s*quad(p) < s*quad(q);
    if (cross(p, q)) return s*cross(p, q) > 0;
    return norm2(p) < norm2(q); // closer first
  }
};

struct line { point p, q; };
bool operator==(line l, line m) {
  return !sign(cross(l.p-l.q,m.p-m.q)) && !sign(cross(l.p-l.q,m.p-l.p));
}

struct segment { point p, q; };
bool operator==(segment l, line m) {
  return (l.p==m.p && l.q==m.q) || (l.p==m.q && l.q==m.p); // do not consider the direction
}
struct circle { point p; real r; };
bool operator==(circle c, circle d) { return c.p == d.p && !sign(c.r - d.r); }


//--------------------------------------------------------------------------
// triangulate simple polygon in O(n) time.
//
// [non-verified]; future work for polygonal overlay
//--------------------------------------------------------------------------
real triangulate(vector<point> ps) {
  int n = ps.size();
  vector<int> next(n);
  for (int i = 0; i < n-1; ++i) next[i] = i+1;
  auto is_ear = [&](int i, int j, int k) {
    if (sign(cross(ps[j]-ps[i], ps[k]-ps[i])) <= 0) return false;
    for (int l = next[k]; l != i; l = next[l])
      if (sign(cross(ps[i]-ps[l], ps[j]-ps[l])) >= 0
       && sign(cross(ps[j]-ps[l], ps[k]-ps[l])) >= 0
       && sign(cross(ps[k]-ps[l], ps[i]-ps[l])) >= 0) return false;
    return true;
  };
  real area = 0;
  for (int i = 0; next[next[i]] != i; ) {
    if (is_ear(i, next[i], next[next[i]])) {
      area  += abs(cross(ps[next[i]]-ps[i], ps[next[next[i]]] - ps[i])) / 2;
      next[i] = next[next[i]];
    } else i = next[i];
  }
  return area;
}

//--------------------------------------------------------------------------
// area of intersection of two circles
// [verified]
//--------------------------------------------------------------------------
real intersection_area(circle c, circle d) {
  if (c.r < d.r) swap(c, d);
  auto A = [&](real r, real h) {
    return r*r*acos(h/r)-h*sqrt(r*r-h*h);
  };
  auto l = norm(c.p - d.p), a = (l*l + c.r*c.r - d.r*d.r)/(2*l);
  if (sign(l - c.r - d.r) >= 0) return 0; // far away
  if (sign(l - c.r + d.r) <= 0) return PI*d.r*d.r;
  if (sign(l - c.r) >= 0) return A(c.r, a) + A(d.r, l-a);
  else return A(c.r, a) + PI*d.r*d.r - A(d.r, a-l);
}

//--------------------------------------------------------------------------
// circle-polygon intersection area
// [verified]
//--------------------------------------------------------------------------
real intersection_area(vector<point> ps, circle c) {
  auto tri = [&](point p, point q){
    point d = q - p;
    auto a = dot(d,p)/dot(d,d), b = (dot(p,p)-c.r*c.r)/dot(d,d);
    auto det = a*a - b;
    if (det <= 0) return arg(p,q) * c.r*c.r / 2;
    auto s = max(0.1, -a-sqrt(det)), t = min(1.1, -a+sqrt(det));
    if (t < 0 || 1 <= s) return c.r*c.r*arg(p,q)/2;
    point u = p + s*d, v = p + t*d;
    return arg(p,u)*c.r*c.r/2 + cross(u,v)/2 + arg(v,q)*c.r*c.r/2;
  };
  auto sum = 0.0;
  for (int i = 0; i < ps.size(); ++i)
    sum += tri(ps[i] - c.p, ps[(i+1)%ps.size()] - c.p);
  return sum;
}
real intersection_area(circle c, vector<point> ps) {
  return intersection_area(ps, c);
}

//--------------------------------------------------------------------------
// find the closest pair of points by sweepline
// [verified]
//--------------------------------------------------------------------------
pair<point,point> closest_pair(vector<point> ps) {
  sort(all(ps), [](point p, point q) { return p.y < q.y; });
  auto u = ps[0], v = ps[1];
  auto best = dot(u-v, u-v);
  auto update = [&](point p, point q) {
    auto dist = dot(p-q, p-q);
    if (best > dist) { best = dist; u = p; v = q; }
  };
  set<point> S; S.insert(u); S.insert(v);
  for (int l = 0, r = 2; r < ps.size(); ++r) {
    if (S.count(ps[r])) return {ps[r], ps[r]};
    if ((ps[l].y-ps[r].y)*(ps[l].y-ps[r].y) > best) S.erase(ps[l++]);
    auto i = S.insert(ps[r]).fst;
    for (auto j = i; ; ++j) {
      if (j == S.end() || (i->x-j->x)*(i->x-j->x) > best) break;
      if (i != j) update(*i, *j);
    }
    for (auto j = i; ; --j) {
      if (i != j) update(*i, *j);
      if (j == S.begin() || (i->x-j->x)*(i->x-j->x) > best) break;
    }
  }
  return {u, v};
}


//--------------------------------------------------------------------------
// find a circle of radius r that contains many points as possible
//
// quad-tree search (this is faster than the next sweepline solution)
// [verified: AOJ 1132]
//--------------------------------------------------------------------------
int maximum_circle_cover(vector<point> ps, double r) {
  const double dx[] = {1,-1,-1,1}, dy[] = {1,1,-1,-1};
  point best_p;
  int best = 0;
  function<void(point,double,vector<point>)>
    rec = [&](point p, double w, vector<point> ps) {
    w /= 2;
    point qs[4];
    vector<point> pss[4];
    for (int i = 0; i < 4; ++i) {
```

```
      qs[i] = p + w * point({dx[i], dy[i]});
      int lo = 0;
      for (point q: ps) {
        auto d = dist(qs[i], q);
        if (sign(d - r) <= 0) ++lo;
        if (sign(d - w*sqrt(2) - r) <= 0) pss[i].push_back(q);
      }
      if (lo > best) { best = lo; best_p = qs[i]; }
    }
    for (int i = 0, j; i < 4; ++i) {
      for (int j = i+1; j < 4; ++j)
        if (pss[i].size() < pss[j].size())
          swap(pss[i], pss[j]), swap(qs[i], qs[j]);
      if (pss[i].size() <= best) break;
      rec(qs[i], w, pss[i]);
    }
  };
  real w = 0;
  for (point p: ps) w = max(w, max(abs(p.x), abs(p.y)));
  rec({0,0}, w, ps);
  return best; //best_p;
}

//-----------------------------------------------------------------------------
// area of union of rectangles
// Bentley's sweepline with segment tree.
//
// [accepted, LightOJ 1120 Rectangle Union]
//-----------------------------------------------------------------------------
struct rectangle { point p, q; }; // lower-left and upper-right
real rectangle_union(vector<rectangle> rs) {
  vector<real> ys; // plane sweep with coordinate compression
  struct event {
    real x, ymin, ymax;
    int add;
    bool operator<(event e) const { return x < e.x; }
  };
  vector<event> es;
  for (auto r: rs) {
    ys.push_back(r.p.y);
    ys.push_back(r.q.y);
    es.push_back({r.p.x, r.p.y, r.q.y, +1});
    es.push_back({r.q.x, r.p.y, r.q.y, -1});
  }
  sort(all(es));
  sort(all(ys));
  ys.erase(unique(all(ys)), ys.end());

  vector<real> len(4 * ys.size()); // segment tree on sweepline
  vector<int> sum(4 * ys.size());
  function<void(real, real, int, int,int,int)> update
    = [&](real ymin, real ymax, int add, int i, int j, int k) {
    ymin = max(ymin, ys[i]); ymax = min(ymax, ys[j]);
    if (ymin >= ymax) return;
    if (ys[i] == ymin && ys[j] == ymax) sum[k] += add;
    else {
      update(ymin, ymax, add, i, (i+j)/2, 2*k+1);
      update(ymin, ymax, add, (i+j)/2, j, 2*k+2);
    }
    if (sum[k]) len[k] = ys[j] - ys[i];
    else        len[k] = len[2*k+1] + len[2*k+2];
  };
  real area = 0;
  for (int i = 0; i+1 < es.size(); ++i) {
    update(es[i].ymin, es[i].ymax, es[i].add, 0, ys.size()-1, 0);
    area += len[0] * (es[i+1].x - es[i].x);
  }
  return area;
}
```

9.1.2 Segment - Segment Intersection

9.1.3 Angle Struct

9.1.4 Center of Mass

9.1.5 Barycentric Coordinates

9.1.6 Circle

```
typedef complex<double> point;
namespace std {
  bool operator<(point p, point q) {
    if (real(p) != real(q)) return real(p) < real(q);
    return imag(p) < imag(q);
  }
};  // namespace std
double dot(point p, point q) { return real(conj(p) * q); }
double cross(point p, point q) { return imag(conj(p) * q); }
double EPS = 1e-8;
int sign(double x) {
  if (x < -EPS) return -1;
  if (x > +EPS) return +1;
  return 0;
}

struct circle {
  point p;
  double r;
};
struct line {
  point p, q;
};

vector<point> intersect(circle C, circle D) {
  double d = abs(C.p - D.p);
  if (sign(d - C.r + D.r) > 0) return {};        // too far
  if (sign(d - abs(C.r - D.r)) <= 0) return {};  // too close
  double a = (C.r * C.r - D.r * D.r + d * d) / (2 * d);
  double h = sqrt(C.r * C.r - a * a);
  point v = (C.p - D.p) / d;
  if (sign(h) == 0) return {C.p + v * a};     // touch
  return {C.p + v * a + point(0, 1) * v * h,  // intersect
          C.p + v * a - point(0, 1) * v * h};
}
// Intersection of line L and circle C.
//
// Let p(t) = L.p + t (L.q - L.p). We solve
//   |p(t) - C.p|^2 = C.r^2
// By letting u = L.p - L.q, v = L.p - C.p, the above is
//   t^2 (u,u) + 2 t (u,v) + (v,v) = C.r^2
//       ~~a~~       ~~b~~   ~~c~~
// Thus
//   det = b^2 - ac,
//   t in { (b + sqrt(det))/a, c/(b + sqrt(det)) }
//
vector<point> intersect(line L, circle C) {
  point u = L.p - L.q, v = L.p - C.p;
  double a = norm(u), b = dot(u, v), c = norm(v), det = b * b - a * c,
         r = b + sqrt(max(det, 0.0));
  if (sign(det) < 0) return {};                   // no solution
  if (sign(det) == 0) return {L.p - b / a * u};  // touch
  return {L.p - (b + sqrt(det)) / a * u, L.p - c / (b + sqrt(det)) * u};
}

//
// Tangent point(s) of point p and circle C
//
// Let q be a tangent point.
// The angle between q-p-c.p is
//   sin(t) = r/|p - c.p|.
// and the solution is
//   p + (c.p - p) * exp(\pm it).
//
// Verified: SPOJ18531
//
vector<point> tangent(point p, circle c) {
  double sin2 = c.r * c.r / norm(p - c.p);
  if (sign(1 - sin2) < 0) return {};
  if (sign(1 - sin2) == 0) return {p};
  point z(sqrt(1 - sin2), sqrt(sin2));
  return {p + (c.p - p) * conj(z), p + (c.p - p) * z};
}
```

9.1.7 Point in Hull and Closest Pair of Points

9.1.8 Convex Hull

```cpp
// graham_scan.cpp
// Eric K. Zhang; Nov. 22, 2017

// Reads a number N (1 <= N <= 3 * 10^5), then N pairs
// of integers (X, Y). Outputs the convex hull of these
// points in O(N log N) time, without including middle
// points that are collinear.

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

typedef long long LL;
typedef pair<int, int> point;

#define MAXN 300000

int N;
point points[MAXN];

LL ccw(point a, point b, point c) {
  return (LL) (b.first - a.first) * (c.second - a.second)
  - (LL) (b.second - a.second) * (c.first - a.first);
}

vector<point> graham_scan(vector<point> points) {
  int N = points.size();
  sort(points.begin(), points.end());
  vector<point> hull(N + 1);
  int idx = 0;
  for (int i = 0; i < N; i++) {
    while (idx >= 2 && ccw(hull[idx - 2], hull[idx - 1], points[i]) >= 0)
      idx--;
    hull[idx++] = points[i];
  }
  int half = idx;
  for (int i = N - 2; i >= 0; i--) {
    while (idx > half && ccw(hull[idx - 2], hull[idx - 1], points[i]) >= 0)
      idx--;
    hull[idx++] = points[i];
  }
  idx--;
  hull.resize(idx);
  return hull;
}

int main() {
  vector<point> points = {
    {44, 140}, {67, 153}, {69, 128}
  };

  vector<point> hull = graham_scan(points);
  for (int i = 0; i < hull.size(); i++) {
    cout << '(' << hull[i].first << ", "
         << hull[i].second << ')' << endl;
  }

  return 0;
}
```

```python
class Graham:
    def __init__(self) -> None:
        pass

    def solve(self, points: List[Point]) -> List[Point]:
        n = len(points)
        average = Point(0, 0)
        for point in points:
            average = average.add(point)

        average = average.div(Point(n, n))

        points.sort(key = lambda point: point.sub(average).angle())

        left = 0
        for i in range(n):
            if points[i].x < points[left].x:
                left = i

        stack = []
        for i in range(n + 1):
            id = (i + left) % n
            while (len(stack) >= 2 and stack[-2].triangle(stack[-1], points[id]) < 0):
                stack.pop()
            stack.append(points[id])

        return stack
```

9.1.9 Online Convex Hull Merger

```cpp
#include <algorithm>
#include <climits>
#include <cmath>
#include <iostream>
#include <vector>
using namespace std;
using ll = long long;
using Point = pair<ll, ll>;
const ll INF = LLONG_MAX / 4;
const double inv_phi = (sqrt(5) - 1) / 2;
void chmax(ll& a, ll b){ if(a < b) a = b; }

ll ccw(Point a, Point b, Point c){
    ll cross = (b.first - a.first) * (c.second - b.second) - (b.second - a.second) * (c.first - b.first);
    if (cross > 0) return 1;
    else if (cross == 0) return 0;
    else return -1;
}

struct ConvexHull{
    vector<Point> lower, upper;
    ConvexHull(Point p): lower{p}, upper{p}{}
    ConvexHull(const ConvexHull& a, const ConvexHull& b){
        vector<Point> v;
        merge(a.lower.begin(), a.lower.end(),
              b.lower.begin(), b.lower.end(), back_inserter(v));
        for (Point p : v){
            while(lower.size() >= 2 && ccw(lower.rbegin()[1], lower.back(), p) <= 0){
                lower.pop_back();
            }
            lower.push_back(p);
        }
        v.clear();
        merge(a.upper.begin(), a.upper.end(),
              b.upper.begin(), b.upper.end(), back_inserter(v));
        for (Point p : v) {
            while(upper.size() >= 2 && ccw(upper.rbegin()[1], upper.back(), p) >= 0){
                upper.pop_back();
            }
            upper.push_back(p);
        }
    }

    ll get(ll A, ll B) const {
        auto& s = B < 0 ? lower : upper;
        // here's the eval function
        auto f = [&](ll i){ return A * s[i].first + B * s[i].second; };
        // golden-section search
        ll l = 0, r = s.size() - 1, r2 = round(r * inv_phi), f_r2 = f(r2);
        while(abs(l - r) >= 6){
            ll l2 = r + llround((l - r) * inv_phi), f_l2 = f(l2);
            if(f_l2 < f_r2) tie(l, r) = tuple{r, l2};
            else tie(r, r2, f_r2) = tuple{r2, l2, f_l2};
        }
        ll ans = -INF;
        if(l > r) swap(l, r);
        for(ll i = l; i <= r; i++) chmax(ans, f(i));
        return ans;
    }
};

int main(){
    cin.tie(nullptr);
    ios::sync_with_stdio(false);
    ll Q;
    cin >> Q;
    vector<ConvexHull> S;
    for(ll i = 1; i <= Q; i++){
        ll X, Y, A, B;
        cin >> X >> Y >> A >> B;
        S.emplace_back(pair{X, Y});
        for(ll j = 1; (i & j) == 0; j <<= 1){
            S.rbegin()[1] = ConvexHull(S.rbegin()[1], S.back());
            S.pop_back();
        }
        ll ans = -INF;
        for(auto& s : S) chmax(ans, s.get(A, B));
        cout << ans << '\n';
    }
}
```

9.1.10 Convex Hull Container of Lines

9.1.11 Polygon Union

9.1.12 Minimum Circle that encloses all points

9.1.13 Convex Layers

```cpp
// Computes convex layers by repeatedly removing convex hull
//"segment tree"-style implementation of online decremental dynamic convex hull
// Based on paper "Maintenance of configurations in the plane" by Overmars and
// van Leeuwen, with some modifications This implementation only supports
// efficient (O(log^2n)) deletion of points, which is enough to compute nested
// convex hulls in O(nlog^2n). This problem can also be solved in O(nlogn)

// Assumes all points are distinct
// Assumes coordinates are at most 10^6
// Can handle 200000 points in a few seconds.

#include <stdint.h>
#include <algorithm>
#include <cassert>
#include <cstdio>
#include <map>
#include <set>
#include <vector>

struct Point {
    int64_t x, y;
    Point operator-(Point p) const { return {x - p.x, y - p.y}; }
    int64_t cross(Point p) const { return x * p.y - y * p.x; }
    int64_t dot(Point p) const { return x * p.x + y * p.y; }
    bool operator<(Point p) const {
        if (y != p.y) return y < p.y;
        return x < p.x;
    }
    bool operator==(Point p) const { return x == p.x && y == p.y; }
    Point operator-() const { return {-x, -y}; }
};

int64_t cross(Point a, Point b, Point c) { return (b - a).cross(c - a); }

class LeftHull {
    std::vector<Point> ps;
    struct Node {
        int bl, br;
        int L, R;
        int lchd, rchd;
    };
    std::vector<Node> nodes;
    int root;
    bool isleaf(int w) { return nodes[w].lchd == -1 && nodes[w].rchd == -1; }
    void pull(int w) {
        assert(!isleaf(w));
        int l = nodes[w].lchd, r = nodes[w].rchd;
        int64_t split_y = ps[nodes[r].L].y;
        while (!isleaf(l) || !isleaf(r)) {
            int a = nodes[l].bl, b = nodes[l].br, c = nodes[r].bl,
                d = nodes[r].br;
            if (a != b && cross(ps[a], ps[b], ps[c]) > 0) {
                l = nodes[l].lchd;
            } else if (c != d && cross(ps[b], ps[c], ps[d]) > 0) {
                r = nodes[r].rchd;
            } else if (a == b) {
                r = nodes[r].lchd;
            } else if (c == d) {
                l = nodes[l].rchd;
            } else {
                int64_t s1 = cross(ps[a], ps[b], ps[c]);
                int64_t s2 = cross(ps[b], ps[a], ps[d]);
                assert(s1 + s2 >= 0);
                if (s1 + s2 == 0 ||
                    s1 * ps[d].y + s2 * ps[c].y < split_y * (s1 + s2)) {
                    l = nodes[l].rchd;
                } else {
                    r = nodes[r].lchd;
                }
            }
        }
        nodes[w].bl = nodes[l].L;
        nodes[w].br = nodes[r].L;
    }
    void build(int w, int L, int R) {
        nodes[w].L = L;
        nodes[w].R = R;
        if (R - L == 1) {
            nodes[w].lchd = nodes[w].rchd = -1;
            nodes[w].bl = nodes[w].br = L;
        } else {
            int M = (L + R) / 2;
            nodes[w].lchd = w + 1;
            nodes[w].rchd = w + 2 * (M - L);
            build(nodes[w].lchd, L, M);
            build(nodes[w].rchd, M, R);
            pull(w);
        }
    }
    int erase(int w, int L, int R) {
        if (R <= nodes[w].L || L >= nodes[w].R) return w;
        if (L <= nodes[w].L && R >= nodes[w].R) return -1;
        nodes[w].lchd = erase(nodes[w].lchd, L, R);
        nodes[w].rchd = erase(nodes[w].rchd, L, R);
        if (nodes[w].lchd == -1) return nodes[w].rchd;
        if (nodes[w].rchd == -1) return nodes[w].lchd;
        pull(w);
        return w;
    }
    // only works for whole hull
    void get_hull(int w, int l, int r, std::vector<int>& res) {
        if (isleaf(w)) {
            res.push_back(nodes[w].L);
        } else if (r <= nodes[w].bl) {
            get_hull(nodes[w].lchd, l, r, res);
        } else if (l >= nodes[w].br) {
            get_hull(nodes[w].rchd, l, r, res);
        } else {
            assert(l <= nodes[w].bl && nodes[w].br <= r);
            get_hull(nodes[w].lchd, l, nodes[w].bl, res);
            get_hull(nodes[w].rchd, nodes[w].br, r, res);
        }
    }
public:
    LeftHull(const std::vector<Point>& ps)
        : ps(ps), nodes(ps.size() * 2), root(0) {
        build(0, 0, ps.size());
    }
    std::vector<int> get_hull() {
        if (root == -1) return {};
        std::vector<int> res;
        get_hull(root, 0, ps.size() - 1, res);
        return res;
    }
    void erase(int L) { root = erase(root, L, L + 1); }
};

std::vector<Point> ps;
std::map<Point, int> id;
int layer[1000005];
int ans[1000005];

int main() {
    int N;
    scanf("%d", &N);
    for (int i = 0; i < N; i++) {
        int X, Y;
        scanf("%d %d", &X, &Y);
        ps.push_back({X, Y});
        id[{X, Y}] = i;
    }

    std::sort(ps.begin(), ps.end());
    LeftHull left(ps);
    std::reverse(ps.begin(), ps.end());
    for (auto& p : ps) {
        p = -p;
    }
    LeftHull right(ps);
    for (auto& p : ps) {
        p = -p;
    }
    std::reverse(ps.begin(), ps.end());
    for (int l = 1, cnt = 0; cnt < N; l++) {
        std::set<int> hull;
        for (int i : left.get_hull()) {
            hull.insert(i);
        }
        for (int i : right.get_hull()) {
            hull.insert(N - 1 - i);
        }
        for (int i : hull) {
            assert(!layer[i]);
            cnt++;
            layer[i] = l;
            left.erase(i);
            right.erase(N - 1 - i);
        }
    }
    for (int i = 0; i < N; i++) {
        ans[id[ps[i]]] = layer[i];
    }
    for (int i = 0; i < N; i++) {
        printf("%d\n", ans[i]);
    }
}
```

9.1.14 Check for segment pair intersection

The following algorithm checks if any two of the n segments intersect in $O(n \log n)$ time. Returns the indices of an intersecting pair.

```
const double EPS = 1E-9;

struct pt { double x, y; };

struct seg {
  pt p, q;
  int id;

  double get_y(double x) const {
    if (abs(p.x - q.x) < EPS) return p.y;
    return p.y + (q.y - p.y) * (x - p.x) / (q.x - p.x);
  }
};

bool intersect1d(double l1, double r1, double l2,
     double r2) {
  if (l1 > r1) swap(l1, r1);
  if (l2 > r2) swap(l2, r2);
  return max(l1, l2) <= min(r1, r2) + EPS;
}

int vec(const pt& a, const pt& b, const pt& c) {
  double s = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) *
      (c.x - a.x);
  return abs(s) < EPS ? 0 : s > 0 ? +1 : -1;
}

bool intersect(const seg& a, const seg& b) {
  return intersect1d(a.p.x, a.q.x, b.p.x, b.q.x) &&
         intersect1d(a.p.y, a.q.y, b.p.y, b.q.y) &&
         vec(a.p, a.q, b.p) * vec(a.p, a.q, b.q) <= 0 &&
         vec(b.p, b.q, a.p) * vec(b.p, b.q, a.q) <= 0;
}

bool operator<(const seg& a, const seg& b) {
  double x = max(min(a.p.x, a.q.x), min(b.p.x, b.q.x));
  return a.get_y(x) < b.get_y(x) - EPS;
}

struct event {
  double x;
  int tp, id;

  event() {}
  event(double x, int tp, int id) : x(x), tp(tp), id(id)
      {}

  bool operator<(const event& e) const {
    if (abs(x - e.x) > EPS) return x < e.x;
    return tp > e.tp;
  }
};
```

```
set<seg> s;
vector<set<seg>::iterator> where;

set<seg>::iterator prev(set<seg>::iterator it) {
  return it == s.begin() ? s.end() : --it;
}

set<seg>::iterator next(set<seg>::iterator it) {
  return ++it;
}

pair<int, int> solve(const vector<seg>& a) {
  int n = (int)a.size();
  vector<event> e;
  for (int i = 0; i < n; ++i) {
    e.push_back(event(min(a[i].p.x, a[i].q.x), +1, i));
    e.push_back(event(max(a[i].p.x, a[i].q.x), -1, i));
  }
  sort(e.begin(), e.end());

  s.clear();
  where.resize(a.size());
  for (size_t i = 0; i < e.size(); ++i) {
    int id = e[i].id;
    if (e[i].tp == +1) {
      set<seg>::iterator nxt = s.lower_bound(a[id]), prv =
          prev(nxt);
      if (nxt != s.end() && intersect(*nxt, a[id]))
        return make_pair(nxt->id, id);
      if (prv != s.end() && intersect(*prv, a[id]))
        return make_pair(prv->id, id);
      where[id] = s.insert(nxt, a[id]);
    } else {
      set<seg>::iterator nxt = next(where[id]), prv =
          prev(where[id]);
      if (nxt != s.end() && prv != s.end() && intersect(*nxt,
          *prv))
        return make_pair(prv->id, nxt->id);
      s.erase(where[id]);
    }
  }

  return make_pair(-1, -1);
}
```

9.1.15 Rectangle Union

9.1.16 Rotating Calipers

9.1.17 KD Tree

9.1.18 Burkhard-Keller Tree

Burkhard-Keller Tree [18] (also known as metric tree) is a flexible data structure created to support the following queries:

- `insert(p)` ... insert a point p in $O(log^2(n))$
- `traverse(p, d)` ... enumerate all points q with $dist(p, q) \leq d$

Note that the distance function in the following implementation uses the Chebyshev distance metric. To change the distance metric, you need to redefine the distance function and redefine the check inside the traverse function. To delete elements and/or rebalance the tree, we can use the same technique as the scapegoat tree.

```
typedef pair<int,int> PII;
int dist(PII a, PII b) { return max(abs(a.first - b.first), abs(a.second - b.second)); }
void process(PII a) { printf("%d %d\n", a.first, a.second); }

template <class T>
struct bk_tree {
  typedef int dist_type;
  struct node {
    T p;
    unordered_map<dist_type, node*> ch;
  } *root;
  bk_tree() : root(0) { }

  node *insert(node *n, T p) {
    if (!n) { n = new node(); n->p = p; return n; }
    dist_type d = dist(n->p, p);
    n->ch[d] = insert(n->ch[d], p);
    return n;
  }
  void traverse(node *n, T p, dist_type dmax) {
    if (!n) return;
    dist_type d = dist(n->p, p);
    if (d < dmax) {
      process(n->p); // write your process
    }
    for (auto i: n->ch)
      if (-dmax <= i.first - d && i.first - d <= dmax)
        traverse(i.second, p, dmax);
  }

  // Wrapper functions
  void insert(T p) { root = insert(root, p); }
  void traverse(T p, dist_type dmax) { traverse(root, p, dmax); }
};
```

9.1.19 Manhatten Minimum Spanning Tree

9.1.20 GJK Algorithm

Are two convex polygons intersecting?

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

#define ld long double
#define vpt vector<Point<T>>

template<class T>
struct Point {
    typedef Point P;
    T x, y;
    explicit Point(T x=0, T y=0) : x(x), y(y) {}
    P operator+(P p) const { return P(x+p.x, y+p.y); }
    P operator-(P p) const { return P(x-p.x, y-p.y); }
    P operator*(T d) const { return P(x*d, y*d); }
    P operator/(T d) const { return P(x/d, y/d); }
    P negate() const { return P(-x, -y); }
    T dot(P p) const { return x*p.x + y*p.y; }
    T dist2() const { return x*x + y*y; }

    P perp() const { return P(-y, x); } // rotates +90 degrees
    P perpCW() const { return P(y, -x); } // rotates -90 degrees <<<

    static P tripleProduct(P a, P b, P c) {
        return b * (a.dot(c)) - a * (b.dot(c));
    }

    static P average(const vector<P> &points) {
        P avg(0, 0);
        for (int i = 0; i < points.size(); i++) {
            avg = avg + points[i];
        }

        return avg / points.size();
    }
};

template<typename T>
int furthestPoint(const vpt &points, Point<T> d) {
    T maxProduct = d.dot(points[0]); // it could be negative!
    int index = 0;
    for (int i = 1; i < points.size(); i++) {
        T product = d.dot(points[i]);
        if (product > maxProduct) {
            maxProduct = product;
            index = i;
        }
    }
    return index;
}

template<typename T>
Point<T> support(const vpt &v1, const vpt &v2, Point<T> d) {
    int i = furthestPoint<T>(v1, d);
    int j = furthestPoint<T>(v2, d.negate());
    return v1[i] - v2[j];
}

int iter_count = 0;

template<typename T>
bool GJK(const vpt &v1, const vpt &v2) {
    int index = 0;

    Point<T> a, b, c, d, ao, perp_prod, simplex[3];

    Point<T> pos1 = Point<T>::average(v1);
    Point<T> pos2 = Point<T>::average(v2);
    d = pos1 - pos2;

    if (d.x == 0 && d.y == 0) {
        d.x = 1;
    }

    a = simplex[0] = support(v1, v2, d);
    if (a.dot(d) <= 0) return false;
    d = a.negate();

    while (true) {
        iter_count++;

        a = simplex[++index] = support(v1, v2, d);
        if (a.dot(d) <= 0) return false;
        ao = a.negate();

        if (index < 2) {
            b = simplex[0];
            d = Point<T>::tripleProduct(b-a, ao, b-a);
            if (d.dist2() == 0) d = (b-a).perpCW();
            continue;
        }

        b = simplex[1];
        c = simplex[0];

        perp_prod = Point<T>::tripleProduct(b-a, c-a, c-a);
        if (perp_prod.dot(ao) >= 0) {
            d = perp_prod;
        } else {
            perp_prod = Point<T>::tripleProduct(c-a, b-a, b-a);
            if (perp_prod.dot(ao) < 0) return true;
            simplex[0] = simplex[1];
            d = perp_prod;
        }

        simplex[1] = simplex[2];
        --index;
    }

    return false;
}

int main() {
    // Point<ld> vertices1[] = {
    //     Point<ld>(4.0f, 11.0f),
    //     Point<ld>(5.0f, 5.0f),
    //     Point<ld>(9.0f, 9.0f)
    // };

    // Point<ld> vertices2[] = {
    //     Point<ld>(4.0f, 11.0f),
    //     Point<ld>(5.0f, 5.0f),
    //     Point<ld>(9.0f, 9.0f)
    // };

    // // NO COLLISION
    // Point<ld> vertices1[] = {
    //     Point<ld>(4.0f, 8.0f),
    //     Point<ld>(5.0f, 5.0f),
    //     Point<ld>(1.0f, 2.0f)
    // };

    // Point<ld> vertices2[] = {
    //     Point<ld>(2.0f, 2.0f),
    //     Point<ld>(4.0f, 4.0f),
    //     Point<ld>(8.0f, 6.0f)
    // };

    int n, m;
    cin >> n >> m;

    vector<Point<ld>> v1(n), v2(m);
    for (int i = 0; i < n; i++) {
        cin >> v1[i].x >> v1[i].y;
    }

    for (int i = 0; i < n; i++) {
        cin >> v2[i].x >> v2[i].y;
    }

    int collisionDetected = GJK<ld>(v1, v2);
    cout << (collisionDetected ? "YES" : "NO") << endl;
    return 0;
}
```

9.1.21 Half-Plane intersection

Calculates the intersection (convex polygon) of a set of half-planes in $O(N \log N)$ time.

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

// Redefine epsilon and infinity as necessary. Be mindful of precision errors.
const long double eps = 1e-9, inf = 1e9;

// Basic point/vector struct.
struct Point {

    long double x, y;
    explicit Point(long double x = 0, long double y = 0) : x(x), y(y) {}

    // Addition, substraction, multiply by constant, cross product.
    friend Point operator + (const Point& p, const Point& q) {
        return Point(p.x + q.x, p.y + q.y);
    }
```

```cpp
    friend Point operator - (const Point& p, const Point& q) {
        return Point(p.x - q.x, p.y - q.y);
    }

    friend Point operator * (const Point& p, const long double& k) {
        return Point(p.x * k, p.y * k);
    }

    friend long double cross(const Point& p, const Point& q) {
        return p.x * q.y - p.y * q.x;
    }
};

// Basic half-plane struct.
struct Halfplane {
    // 'p' is a passing point of the line and 'pq' is the direction vector of the line.
    Point p, pq;
    long double angle;

    Halfplane() {}
    Halfplane(const Point& a, const Point& b) : p(a), pq(b - a) {
        angle = atan2l(pq.y, pq.x);
    }

    // Check if point 'r' is outside this half-plane.
    // Every half-plane allows the region to the LEFT of its line.
    bool out(const Point& r) {
        return cross(pq, r - p) < -eps;
    }

    // Comparator for sorting.
    // If the angle of both half-planes is equal, the leftmost one should go first.
    bool operator < (const Halfplane& e) const {
        if (fabsl(angle - e.angle) < eps) return cross(pq, e.p - p) < 0;
        return angle < e.angle;
    }

    // We use equal comparator for std::unique to easily remove parallel half-planes.
    bool operator == (const Halfplane& e) const {
        return fabsl(angle - e.angle) < eps;
    }

    // Intersection point of the lines of two half-planes. It is assumed they're never parallel.
    friend Point inter(const Halfplane& s, const Halfplane& t) {
        long double alpha = cross((t.p - s.p), t.pq) / cross(s.pq, t.pq);
        return s.p + (s.pq * alpha);
    }
};

// Actual algorithm
vector<Point> hp_intersect(vector<Halfplane>& H) {
    Point box[4] = {  // Bounding box in CCW order
        Point(inf, inf),
        Point(-inf, inf),
        Point(-inf, -inf),
        Point(inf, -inf)
    };

    for(int i = 0; i < 4; i++) { // Add bounding box half-planes.
        Halfplane aux(box[i], box[(i+1) % 4]);
        H.push_back(aux);
    }

    // Sort and remove duplicates
    sort(H.begin(), H.end());
    H.erase(unique(H.begin(), H.end()), H.end());

    deque<Halfplane> dq;
    int len = 0;
    for(int i = 0; i < int(H.size()); i++) {
        // Remove from the back of the deque while last half-plane is redundant
        while (len > 1 && H[i].out(inter(dq[len-1], dq[len-2]))) {
            dq.pop_back();
            --len;
        }

        // Remove from the front of the deque while first half-plane is redundant
        while (len > 1 && H[i].out(inter(dq[0], dq[1]))) {
            dq.pop_front();
            --len;
        }

        // Add new half-plane
        dq.push_back(H[i]);
        ++len;
    }

    // Final cleanup: Check half-planes at the front against the back and vice-versa
    while (len > 2 && dq[0].out(inter(dq[len-1], dq[len-2]))) {
        dq.pop_back();
        --len;
    }

    while (len > 2 && dq[len-1].out(inter(dq[0], dq[1]))) {
        dq.pop_front();
        --len;
    }

    // Report empty intersection if necessary
    if (len < 3) return vector<Point>();

    // Reconstruct the convex polygon from the remaining half-planes.
    vector<Point> ret(len);
    for(int i = 0; i+1 < len; i++) {
        ret[i] = inter(dq[i], dq[i+1]);
    }
    ret.back() = inter(dq[len-1], dq[0]);
    return ret;
}
```

9.2 3D Geometry

9.2.1 Point3D

9.2.2 3D Geometry

```cpp
#define LINE 0
#define SEGMENT 1
#define RAY 2

struct point{
    double x, y, z;
    point(){};
    point(double _x, double _y, double _z){ x=_x; y=_y; z=_z; }
    point operator+ (point p) { return point(x+p.x, y+p.y, z+p.z); }
    point operator- (point p) { return point(x-p.x, y-p.y, z-p.z); }
    point operator* (double c) { return point(x*c, y*c, z*c); }
};

double dot(point a, point b){
    return a.x*b.x + a.y*b.y + a.z*b.z;
}

point cross(point a, point b) {
    return point(a.y*b.z-a.z*b.y, a.z*b.x-a.x*b.z, a.x*b.y-a.y*b.x);
}

double distSq(point a, point b){
    return dot(a-b, a-b);
}

// compute a, b, c, d such that all points lie on ax + by + cz = d. TODO: test this
double planeFromPts(point p1, point p2, point p3,
    double& a, double& b, double& c, double& d) {
        point normal = cross(p2-p1, p3-p1);
        a = normal.x; b = normal.y; c = normal.z;
        d = -a*p1.x-b*p1.y-c*p1.z;
}

// project point onto plane. TODO: test this
point ptPlaneProj(point p, double a, double b, double c, double d) {
        double l = (a*p.x+b*p.y+c*p.z+d)/(a*a+b*b+c*c);
        return point(p.x-a*l, p.y-b*l, p.z-c*l);
}

// distance from point p to plane aX + bY + cZ + d = 0
double ptPlaneDist(point p, double a, double b, double c, double d){
    return fabs(a*p.x + b*p.y + c*p.z + d) / sqrt(a*a + b*b + c*c);
}

// distance between parallel planes aX + bY + cZ + d1 = 0 and
// aX + bY + cZ + d2 = 0
double planePlaneDist(double a, double b, double c, double d1, double d2){
    return fabs(d1 - d2) / sqrt(a*a + b*b + c*c);
}

// square distance between point and line, ray or segment
double ptLineDistSq(point s1, point s2, point p, int type){
    double pd2 = distSq(s1, s2);
    point r;
    if(pd2 == 0) r = s1;
    else {
        double u = dot(p-s1, s2-s1) / pd2;
        r = s1 + (s2 - s1)*u;
        if(type != LINE && u < 0.0)
            r = s1;
        if(type == SEGMENT && u > 1.0)
            r = s2;
    }
    return distSq(r, p);
}

// Distance between lines ab and cd. TODO: Test this
double lineLineDistance(point a, point b, point c, point d) {
        point v1 = b-a;
        point v2 = d-c;
```

```
	point cr = cross(v1, v2);
	if (dot(cr, cr) < EPS) {
		point proj = v1*(dot(v1, c-a)/dot(v1, v1));
		return sqrt(dot(c-a-proj, c-a-proj));
	} else {
		point n = cr/sqrt(dot(cr, cr));
		point p = dot(n, c - a);
		return sqrt(dot(p, p));
	}
}

// Distance between line segments ab and cd (translated from Java)
double segmentSegmentDistance(point a, point b, point c, point d) {
	point u = b - a, v = d - c, w = a - c;
	double a = dot(u, u), b = dot(u, v), c = dot(v, v), d = dot(u, w), e = dot(v, w);
	double D = a*c-b*b;
	double sc, sN, sD = D;
	double tc, tN, tD = D;

    // compute the line parameters of the two closest points
    if (D < EPS) { // the lines are almost parallel
        sN = 0.0;        // force using point P0 on segment S1
        sD = 1.0;        // to prevent possible division by 0.0 later
        tN = e;
        tD = c;
    } else {                // get the closest points on the infinite lines
        sN = (b*e - c*d);
        tN = (a*e - b*d);
        if (sN < 0.0) {       // sc < 0 => the s=0 edge is visible
            sN = 0.0;
            tN = e;
            tD = c;
        }
        else if (sN > sD) {  // sc > 1 => the s=1 edge is visible
            sN = sD;
            tN = e + b;
            tD = c;
        }
    }

    if (tN < 0.0) {            // tc < 0 => the t=0 edge is visible
        tN = 0.0;
        // recompute sc for this edge
        if (-d < 0.0)
            sN = 0.0;
        else if (-d > a)
            sN = sD;
        else {
            sN = -d;
            sD = a;
        }
    }
    else if (tN > tD) {      // tc > 1 => the t=1 edge is visible
        tN = tD;
        // recompute sc for this edge
        if ((-d + b) < 0.0)
            sN = 0;
        else if ((-d + b) > a)
            sN = sD;
        else {
            sN = (-d + b);
            sD = a;
        }
    }
    // finally do the division to get sc and tc
    sc = (abs(sN) < EPS ? 0.0 : sN / sD);
    tc = (abs(tN) < EPS ? 0.0 : tN / tD);

    // get the difference of the two closest points
    point dP = w + (sc * u) - (tc * v);  // = S1(sc) - S2(tc)
    return sqrt(dot(dP, dP));   // return the closest distance
}

double signedTetrahedronVol(point A, point B, point C, point D) {
	double A11 = A.x - B.x;
	double A12 = A.x - C.x;
	double A13 = A.x - D.x;
	double A21 = A.y - B.y;
	double A22 = A.y - C.y;
	double A23 = A.y - D.y;
	double A31 = A.z - B.z;
	double A32 = A.z - C.z;
	double A33 = A.z - D.z;
	double det =
		A11*A22*A33 + A12*A23*A31 +
		A13*A21*A32 - A11*A23*A32 -
		A12*A21*A33 - A13*A22*A31;
	return det / 6;
}

// Parameter is a vector of vectors of points - each interior vector
// represents the 3 points that make up 1 face, in any order.
// Note: The polyhedron must be convex, with all faces given as triangles.
double polyhedronVol(vector<vector<point> > poly) {
	int i,j;
	point cent(0,0,0);
	for (i=0; i<poly.size(); i++)
		for (j=0; j<3; j++)
			cent=cent+poly[i][j];
	cent=cent*(1.0/(poly.size()*3));
	double v=0;
	for (i=0; i<poly.size(); i++)
		v+=fabs(signedTetrahedronVol(cent,poly[i][0],poly[i][1],poly[i][2]));
	return v;
}
```

9.2.3 3D Convex Hull

9.2.4 Delaunay Triangulation

9.2.5 Voronoi Diagram with Euclidean Metric

9.2.6 Voronoi Diagram with Manhattan Metric

9.2.7 3D Coordinate-Wise Domination

```cpp
//
// 3D Coordinate-Wise Domination
//
// Description:
//   Point (x,y,z) dominates (x',y',z') if
//     x < x', y < y', and z < z'
//   holds. Kung-Luccio-Preparata proposed an algorithm to compute
//   the all set of dominating points in O(n log n) time.
//
// Complexity:
//   O(n log n). By using this method recursively,
//   we can solve d-dimensional domination in O(n log^{d-2} n).
//
// Reference:
//   Hsiang-Tsung Kung, Fabrizio Luccio, Franco P. Preparata (1975):
//   "On finding the maxima of a set of vectors." Journal of the ACM,
//   vol.22, no.4, pp.469-476.
//
// Implementation Author:
//   Mitko Nikov
//
// Tested:
//   https://mendo.mk/Task.do?id=912 & stress-tested
//

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

struct Point {
    int x, y, z;
};

int domination(vector<Point> v) {
    int n = v.size();
    int ans = 0;
    sort(v.begin(), v.end(), [](const Point& a, const Point& b) {
        if (a.x != b.x) return (a.x > b.x);
        if (a.y != b.y) return (a.y < b.y); // this is very important
        return (a.z < b.z);
    });

    auto cond = [&](pair<int, int> a, pair<int, int> b) {
        return (a.first > b.first && a.second > b.second);
    };

    // Maintains a data structure, where x is mapped to
    // the max(y) seen with this x while holding that
    // if x2 > x1 then y2 < y1 for each (x[i], y[i])
    map<int, int> frontier; // { x -> max(y) }
    auto update = [&](Point p) {
        auto it = frontier.find(p.y);
        if (it != frontier.end()) {
            it->second = max(it->second, p.z);
        } else {
            frontier[p.y] = p.z;
            it = frontier.find(p.y);
        }

        while (it != frontier.begin()
            && prev(it)->second <= it->second) {
            frontier.erase(prev(it));
        }

        if (next(it) != frontier.end() && cond(*next(it), { p.y, p.z })) {
            if (it->second <= p.z) frontier.erase(it);
            return true;
        } else {
            return false;
        }
    };

    for (int i = 0; i < n; i++) {
        if (update(v[i])) {
            ans++;
        }
    }
    return ans;
}

int main() {
    int n;
    scanf("%d", &n);
    vector<Point> ps(n);
    for (int i = 0; i < n; ++i) {
        scanf("%d %d %d", &ps[i].x, &ps[i].y, &ps[i].z);
    }
    printf("%d\n", domination(ps));
}
```

9.2.8 Maximum Circle Cover

10 Miscellaneous Stuff

10.1 Gosper's Hack

```
//
//  Iterate through the bit masks by increasing number of ON bits.
//  For every C(N, K) bits the function f() is called.
//

#define ll long long
void GospersHack(int n, int k, function<void(ll)> f) {
    ll set = (1 << k) - 1;
    ll limit = (1 << n);
    while (set < limit) {
        f(set);
        ll c = set & - set;
        ll r = set + c;
        set = (((r ^ set) >> 2) / c) | r;
    }
}
```

10.2 Matrix Flips

```
template<typename T>
vector<vector<T>> rotate(const vector<vector<T>>& matrix, int x, int y) {
    vector<vector<T>> result(x, vector<T>(y, 0));
    int newColumn, newRow = 0;
    for (int oldColumn = x - 1; oldColumn >= 0; oldColumn--) {
        newColumn = 0;
        for (int oldRow = 0; oldRow < y; oldRow++) {
            result[newRow][newColumn] = matrix[oldRow][oldColumn];
            newColumn++;
        }
        newRow++;
    }

    return result;
}

template<typename T>
vector<vector<T>> flipH(const vector<vector<T>> &matrix, int n) {
    vector<vector<T>> result(n, vector<T>(n, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            result[i][j] = matrix[i][n-j-1];
        }
    }

    return result;
}

template<typename T>
vector<vector<T>> flipV(const vector<vector<T>> &matrix, int n) {
    vector<vector<T>> result(n, vector<T>(n, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            result[i][j] = matrix[n-i-1][j];
        }
    }

    return result;
}
```

10.3 Calendar Conversions

```
// Routines for performing computations on dates.  In these routines,
// months are expressed as integers from 1 to 12, days are expressed
// as integers from 1 to 31, and years are expressed as 4-digit
// integers.

#include <iostream>
#include <string>

using namespace std;

string dayOfWeek[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};

// converts Gregorian date to integer (Julian day number)
int dateToInt (int m, int d, int y){
  return
    1461 * (y + 4800 + (m - 14) / 12) / 4 +
    367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
    3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 +
    d - 32075;
}

// converts integer (Julian day number) to Gregorian date: month/day/year
void intToDate (int jd, int &m, int &d, int &y){
  int x, n, i, j;

  x = jd + 68569;
  n = 4 * x / 146097;
  x -= (146097 * n + 3) / 4;
  i = (4000 * (x + 1)) / 1461001;
  x -= 1461 * i / 4 - 31;
  j = 80 * x / 2447;
  d = x - 2447 * j / 80;
  x = j / 11;
  m = j + 2 - 12 * x;
  y = 100 * (n - 49) + i + x;
}

// converts integer (Julian day number) to day of week
string intToDay (int jd){
  return dayOfWeek[jd % 7];
}

int main (int argc, char **argv){
  int jd = dateToInt (3, 24, 2004);
  int m, d, y;
  intToDate (jd, m, d, y);
  string day = intToDay (jd);

  // expected output:
  //    2453089
  //    3/24/2004
  //    Wed
  cout << jd << endl
    << m << "/" << d << "/" << y << endl
    << day << endl;
}
```

10.4 Hamilton Cycle with Ore Condition

10.5 Exact Cover

```
//
// Exact Cover
//
// Description:
//   We are given a family of sets F on [0,n).
//   The exact cover problem is to find a subfamily of F
//   such that each k in [0,n) is covered exactly once.
//   For example, if F consists from
//     {1,2}, {1,2,3}, {3,4}, {5,6}, {3,5,6},
//   then the exact cover is
//     {1,2}, {3,4}, {5,6}.
//
// Algorithm:
//   Knuth's algorithm X is the following recursive algorithm:
//
//     select some k in [0,n)
//     for each subset S that covers k
//       select S and remove all conflicting sets
//       recursion
//
//    To implement this algorithm efficiently, we can use
//    a data structure, which is called dancing links.
//
// Verified:
//   SPOJ 1428: EASUDOKU
//   SPOJ 1110: SUDOKU
//
#include <iostream>
#include <vector>
#include <cstdio>
#include <algorithm>
#include <functional>

using namespace std;

#define fst first
#define snd second
#define all(c) ((c).begin()), ((c).end())

vector<int> exact_cover(vector<vector<int>> sets) {
  int m = 0, M = 10;
  for (auto &v: sets) {
    m = max(m, *max_element(all(v))+1);
  }
  M += (1 + sets.size()) * m;
  vector<int> L(M), R(M), U(M), D(M), S(M), C(M), A(M);
  for (int i = 0; i <= m; ++i) {
    L[i] = i-1; R[i] = i+1;
    D[i] = U[i] = C[i] = i;
  }
  L[0] = m; R[m] = 0;
  int p = m+1;
  for (int row = 0; row < sets.size(); ++row) { // add sets
    for (int i = 0; i < sets[row].size(); ++i) {
      int col = sets[row][i];
      C[p] = col; A[p] = row; ++S[col];
      D[p] = D[col]; U[p] = col; D[col] = U[D[p]] = p;
      if (i == 0) { L[p] = R[p] = p; }
      else { L[p] = p-1; R[p] = R[p-1]; R[p-1] = L[R[p]] = p; }
      ++p;
    }
  }
  auto remove = [&](int x) {
    L[R[x]] = L[x]; R[L[x]] = R[x];
    for (int i = D[x]; i != x; i = D[i])
      for (int j = R[i]; j != i; j = R[j])
      { U[D[j]] = U[j], D[U[j]] = D[j], --S[C[j]]; }
  };
  auto resume = [&](int x) {
    for (int i = U[x]; i != x; i = U[i])
      for (int j = L[i]; j != i; j = L[j])
      { U[D[j]] = j, D[U[j]] = j, ++S[C[j]]; }
    L[R[x]] = x; R[L[x]] = x;
  };

  vector<int> solution;
  function<bool(void)> rec = [&]() {
    if (R[m] == m) return true; // found
    int col = R[m];
    for (int i = R[m]; i != m; i = R[i])
      if (S[i] < S[col]) col = i;
    if (S[col] == 0) return false;
    remove(col);
    for (int i = D[col]; i != col; i = D[i]) {
      solution.push_back(A[i]);
      for (int j = R[i]; j != i; j = R[j]) remove(C[j]);
      if (rec()) return true;
      for (int j = L[i]; j != i; j = L[j]) resume(C[j]);
      solution.pop_back();
    }
    resume(col);
    return false;
  };
  rec();
  return solution;
}



int w = 3;
bool sudoku(vector<vector<int>> &b) {
  vector<vector<int>> sets;
  vector<tuple<int,int,int>> ns;

  auto id = [](int a, int b, int c) { return w*w*w*w*a + w*w*b + c; };
  auto add_set =[&](int i, int j, int k) {
    sets.push_back({});
    sets.back().push_back(id(0, i, j));
    sets.back().push_back(id(1, i, k));
    sets.back().push_back(id(2, j, k));
    sets.back().push_back(id(3, w*(i/w)+(j/w), k));
    ns.push_back(make_tuple(i, j, k));
  };
  for (int i = 0; i < w*w; ++i) {
    for (int j = 0; j < w*w; ++j) {
      if (b[i][j] == 0) {
        for (int k = 0; k < w*w; ++k)
          add_set(i, j, k);
      } else {
        add_set(i, j, b[i][j]-1);
      }
    }
  }
  auto x = exact_cover(sets);
  if (x.empty()) return false;
  for (auto a: x) {
    int i = get<0>(ns[a]);
    int j = get<1>(ns[a]);
    int k = get<2>(ns[a]);
    b[i][j] = k;
  }
  return true;
}

void SPOJ_EASUDOKU() {
  w = 3;
  int ncase; scanf("%d", &ncase);
  for (int icase = 0; icase < ncase; ++icase) {
    vector<vector<int>> b(w*w, vector<int>(w*w));
    for (int i = 0; i < w*w; ++i)
      for (int j = 0; j < w*w; ++j)
        scanf("%d", &b[i][j]);
    if (sudoku(b)) {
      for (int i = 0; i < w*w; ++i) {
        for (int j = 0; j < w*w; ++j) {
          if (j > 0) printf(" ");
          printf("%d", b[i][j]+1);
        }
        printf("\n");
      }
      printf("\n");
    } else {
      printf("No solution\n");
    }
  }
}
void SPOJ_SUDOKU() {
  w = 4;
  int ncase; scanf("%d", &ncase);
  for (int icase = 0; icase < ncase; ++icase) {
    if (icase > 0) printf("\n");
    vector<vector<int>> b(w*w, vector<int>(w*w));
    for (int i = 0; i < w*w; ++i) {
      char s[1024];
      scanf("%s", s);
      for (int j = 0; j < w*w; ++j) {
        char c = s[j];
        if (c == '-') b[i][j] = 0;
        else          b[i][j] = c - 'A' + 1;
      }
    }
    if (sudoku(b)) {
      for (int i = 0; i < w*w; ++i) {
        for (int j = 0; j < w*w; ++j) {
          printf("%c", b[i][j]+'A');
        }
        printf("\n");
      }
    }
  }

}

int main() {
  // SPOJ_EASUDOKU();
   SPOJ_SUDOKU();
}
```

10.6 Roman Numerals

```
#include <iostream>
#include <string>

std::string to_roman(int value) {
    struct romandata_t {
        int value;
        char const* numeral;
    };
    static romandata_t const romandata[] = {
        1000, "M",  900, "CM", 500, "D",  400, "CD", 100, "C",
        90,   "XC", 50,  "L",  40,  "XL", 10,  "X",  9,   "IX",
        5,    "V",  4,   "IV", 1,   "I",  0,   NULL}; // end marker

    std::string result;
    for (romandata_t const* current = romandata; current->value > 0;
         ++current) {
        while (value >= current->value) {
            result += current->numeral;
            value -= current->value;
        }
    }
    return result;
}

int main() {
    for (int i = 1; i <= 4000; ++i) {
        std::cout << to_roman(i) << std::endl;
    }
}
```

10.7 Group Dynamics

10.8 Graph Isomorphism

10.9 Integer coordinates on a line

10.10 Bradley-Terry Model for Pairwise Comparison

Consider pairwise comparisons between N players. This model assumes that each player i has a strength w_i, and player i beats player j with probability $\frac{w_i}{w_i+w_j}$. The algorithm estimates the strengths from a comparison data.

Time Complexity: $O(N^2)$ per iteration. The number of iterations needed are usually small.

```
struct bradley_terry {
  int n;
  vector<double> w;
  vector<vector<int>> a;
  bradley_terry(int n) : n(n), w(n,1) { regularize(); }

  // reguralization avoids no-match pairs
  void regularize() {
    a.assign(n, vector<int>(n, 1));
    for (int i = 0; i < n; ++i)
      a[i][i] = n-1;
  }

  // win beats lose num times
  void add_match(int win, int lose, int num = 1) {
    a[win][lose] += num;
    a[win][win] += num;
  }
  // estimate the strengths
  void learning() {
    for (int iter = 0; iter < 100; ++iter) {
      double norm = 0;
      vector<double> z(n);
      for (int i = 0; i < n; ++i) {
        double sum = 0;
        for (int j = 0; j < n; ++j)
          if (i != j) sum += (a[i][j] + a[j][i]) / (w[i] +
              w[j]);
        z[i] = a[i][i] / sum;
        norm += z[i];
      }
      double err = 0;
      for (int i = 0; i < n; ++i) {
        err += abs(w[i] - z[i] / norm);
        w[i] = z[i] / norm;
      }
      if (err < 1e-6) break;
    }
  }
};
```

11 Picked Solutions

11.1 Range Harvest

```cpp
#include <bits/stdc++.h>
#define ll long long
#define pii pair<ll, ll>
using namespace std;
ll mod = 998244353;
struct mint; // this struct is used
struct interval {
    pii range;
    ll day;

    bool operator<(const interval &other) const {
        return this->range < other.range;
    }
};
struct cmp {
    bool operator()(const interval &t, const interval &other) const {
        return t.range < other.range;
    }
};

mint harvest = 0;

mint calc(ll L, ll R, mint dayNow, mint dayPrev) {
    if (L > R) return (mint)0;
    ll Rm = (R - 1);
    ll Lm = (L - 1);
    if (R % 2 == 0) R /= 2;
    else if (Rm % 2 == 0) Rm /= 2;
    if (L % 2 == 0) L /= 2;
    else if (Lm % 2 == 0) Lm /= 2;

    mint Rs = (mint)R * (mint)Rm;
    mint Ls = (mint)L * (mint)Lm;
    return (Rs - Ls) * (dayNow - dayPrev);
}

set<interval, cmp> sets;
void removeInterval(ll L, ll R, ll currentDay) {
    if (L == R) return;
        auto it = sets.lower_bound(interval{ pii{L, 0}, -1 });
    it--;
    // L and R should not be changed
    ll left_day = it->day, right_day = it->day;
    ll curL = it->range.first;
    ll curR = it->range.second;
    while (it != sets.end() && it->range.first < R) {
        right_day = it->day;
        harvest += calc(max(L, it->range.first), min(R, it->range.second), currentDay, it->day);
        curR = max(curR, it->range.second);
        it = sets.erase(it);
    }
    if (curL != L) sets.insert({{curL, L}, left_day});
    if (curR != R) sets.insert({{R, curR}, right_day});
}

int main() {
    ll N;
    int Q;
    cin >> N >> Q;
    sets.insert({ {0, LLONG_MAX - 10}, 0 });
    for (int i = 0; i < Q; i++) {
        ll day, l, r;
        cin >> day >> l >> r;
        harvest = 0;
        removeInterval(l, r+1, day);
        sets.insert({ {l, r+1}, day });
        cout << harvest.value << endl;
    }
    ll prev = 0;
    for (auto s: sets) {
        if (s.range.first != prev) assert(false);
        prev = s.range.second;
    }
    cout << flush;
    return 0;
}
```

11.2 Vinjete

```cpp
// "Vinjete" Task from COI2022
// Given a tree where each path adds an interval of numbers,
// calculate the interval cost from to root to each of the cities

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

struct edge {
    int u, v, l, r;
    void read() {
        cin >> u >> v >> l >> r;
        u--; v--; l--; r--;
    }
};

struct change {
    pair<int, int> interval; bool added = false; int effect = 0;
    int calc() { return effect = interval.second - interval.first; }
};

vector<vector<edge>> adj;
set<pair<int, int>> dsu;
vector<change> changes;
int current_ans = 0;
// [inclusive, exclusive).
int addInterval(int L, int R) {
    int change_in_ans = 0;
    if (L == R) return 0;
    auto it = dsu.lower_bound({L, R}), before = it;
    while (it != dsu.end() && it->first <= R) {
        R = max(R, it->second);
        changes.push_back({*it, false});
        change_in_ans -= changes.back().calc();
        before = it = dsu.erase(it);
    }
    if (it != dsu.begin() && (--it)->second >= L) {
        L = min(L, it->first);
        R = max(R, it->second);
        changes.push_back({*it, false});
        change_in_ans -= changes.back().calc();
        dsu.erase(it);
    }
    changes.push_back({{L, R}, true});
    change_in_ans += changes.back().calc();
    dsu.insert(before, {L, R});
    return change_in_ans;
}

int insert(int l, int r) {
    int sz = changes.size();
    current_ans += addInterval(l, r + 1);
    return sz;
}

void rollback(int size) {
    while (changes.size() > size) {
        change c = changes.back();
        changes.pop_back();
        if (c.added) {
            current_ans -= c.effect;
            dsu.erase(c.interval);
        } else {
            current_ans += c.effect;
            dsu.insert(c.interval);
        }
    }
}

vector<int> ans;
void dfs(int u, int p) {
    ans[u] = current_ans;
    for (edge e : adj[u]) {
        if (e.v == p) continue;
        int size = insert(e.l, e.r);
        dfs(e.v, u);
        rollback(size);
    }
}

int main() {
    int n, m;
    cin >> n >> m; adj.resize(n); ans.resize(n, 0); changes.reserve(3e5);
    for (int i = 0; i < n - 1; i++) {
        edge e; e.read();
        adj[e.u].push_back(e); swap(e.u, e.v);
        adj[e.u].push_back(e);
    }
    dfs(0, -1);
    for (int i = 1; i < n; i++) cout << ans[i] << endl; return 0;
}
```

12 References

References

[1] September 1998. [Online; accessed 1. Oct. 2023].

[2] CodeFu - 2021 Autumn - Problem GAMBIT, November 2021. [Online; accessed 3. Nov. 2022].

[3] Articulation points and bridges (Tarjan's Algorithm) - Codeforces, November 2022. [Online; accessed 4. Nov. 2022].

[4] CodeChef - Query on a tree VI, November 2022. [Online; accessed 12. Nov. 2022].

[5] Discrete Log - Algorithms for Competitive Programming, November 2022. [Online; accessed 23. Nov. 2022].

[6] Discrete Root - Algorithms for Competitive Programming, November 2022. [Online; accessed 23. Nov. 2022].

[7] Disjoint Set Union - Algorithms for Competitive Programming, November 2022. [Online; accessed 13. Nov. 2022].

[8] Factorial modulo p - Algorithms for Competitive Programming, November 2022. [Online; accessed 23. Nov. 2022].

[9] Finding articulation points in a graph in O(N+M) - Algorithms for Competitive Programming, October 2022. [Online; accessed 4. Nov. 2022].

[10] Gray code - Algorithms for Competitive Programming, November 2022. [Online; accessed 24. Nov. 2022].

[11] SPOJ.com - Problem DYNACON1, November 2022. [Online; accessed 4. Nov. 2022].

[12] SPOJ.com - Problem MATCHING, November 2022. [Online; accessed 3. Nov. 2022].

[13] UVa - Street Directions, November 2022. [Online; accessed 4. Nov. 2022].

[14] An easy-to-write tree hash that won't collide easy - peehs-moorhsum's blog, March 2023. [Online; accessed 20. Mar. 2023].

[15] Dynamic connectivity problem - Codeforces, March 2023. [Online; accessed 8. Mar. 2023].

[16] SPOJ.com - Problem ADABLOOM, March 2023. [Online; accessed 8. Mar. 2023].

[17] Andreas Bjorklund and Thore Husfeldt. Inclusion–Exclusion Algorithms for Counting Set Partitions. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 21–24. IEEE.

[18] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, April 1973.

[19] Harold N. Gabow. An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs. *J. ACM*, 23(2):221–234, April 1976.

[20] M. R. Henzinger and M. L. Fredman. Lower Bounds for Fully Dynamic Connectivity Problems in Graphs. *Algorithmica*, 22(3):351–362, November 1998.

[21] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM J. Comput.*, July 2006.

[22] Richard M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Math.*, 23(3):309–311, January 1978.

[23] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, January 1979.

[24] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theoret. Comput. Sci.*, 130(1):203–236, August 1994.

[25] ShahjalalShohag. code-library, December 2022. [Online; accessed 27. Dec. 2022].