Competitive Programming Library

July 9, 2023

Intentionally left blank.

Contents

1 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. 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;
```

1.1 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()
    );
}
```

1.1.1 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());
}
```

1.2 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);
}
```

1.3 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.

```
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;
    }
}
```

1.4 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;
}
```

1.5 2SAT

1.6 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 [8] 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);
    }
}
```

1.7 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 [2] 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)* *[12]*

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.

1.8 Maximum Matchings

1.8.1 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. [19] 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)* *[11]*
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)* *[1]*
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.

1.8.2 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**.

1.8.3 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.

1.8.4 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.

```
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;
    }
```

```
    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;
    }
};
```

1.8.5 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

```
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;
  }
```

```
  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); }
}
```

1.8.6 Max Flow - Capacity Scaling

1.8.7 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.**

```
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];
  }
```

```
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;
        }
      }
    }
  }
}
```

1.8.8 Maximum Matching in General Graphs

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

[17]

```
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)* *[15]*

1.8.9 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 << " ";
}
```

1.8.10 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;
}
```

1.9 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;
}
```

1.10 Strongly Connected Components

Time Complexity: $O(N + M)$

```
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);
  }

  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;
    }
  }
};
```

1.11 Dynamic Connectivity

The Dynamic Connectivity Problem [14] 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.

1.12 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);
  }
};
```

1.13 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];
        }
    }
};
```

1.14 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 [20] 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;
  }
};
```

1.15 Max Cut

1.16 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});
  }
};
```

1.17 Chromatic Number

2 Trees

2.7 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. [21]

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;
  }
};
```

3 Algorithms

3.1 Binary and Ternary Search Function

3.2 Index Compression

3.3 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];
    }
  }
};
```

3.4 Mo's Algorithm

3.5 Merge Sort

3.6 K-th Order Statistic

3.7 K-th Shortest Paths

3.8 Shunting Yard

4 String Algorithms

4.1 Minimum Lexicographical String Rotation

4.2 Manacher Algorithm

4.3 Z Function

4.4 KMP Algorithm

4.5 Offline Aho Corassick

4.6 Online Aho Corassick

5 Data Structures

5.1 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. [6]

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)];
  }
};
```

When extending the data structure to allow rollbacks, 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.

In some DSU applications [6], 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];
}
```

5.2 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";
  }
}
```

5.3 Sparse Table

5.4 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]);
  }
};
```

5.5 Fenwick Tree

5.6 Segment Tree

5.7 Sparse Lazy Segment Tree

5.8 Persistent Segment Tree

5.9 Persistent Fenwick Tree

```
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;
  }
};
```

5.10 2D Sparse Table

[23]

```
// 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))
            );
    };
};
```

5.11 2D Sparse Segment Tree

5.12 K-th Minimum in Segment Tree

5.13 Treap

5.14 Indexed Binary Heap

5.15 Persistent Array

5.16 Sparse Bitset

5.17 Interval Container

5.18 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);
    }
  }
};
```

5.19 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, "");
  }
};
```

5.20 Eval-Link-Update Tree

5.21 Euler Tour Tree

The Euler Tour Tree is a dynamic connectivity based data structure [18] [22] 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;
  }
};
```

5.22 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.

```
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>
```

```
  // 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;
  }
```

```
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;
}
```

```
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* *[10]*

Problem 6 - *CODECHEF - Query on a tree VI* *[3]*

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 Maths

6.1 Basic Algorithms

6.1.1 Binary Exponentiation

6.1.2 Sum of Geometric Progression

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

```
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;
}
```

6.2 Primes

6.2.1 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)
}
```

6.3 Modular Arithmetic

6.3.1 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. [4]

```
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;
}
```

6.3.2 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: $x^k \equiv a \ (mod\ n)$ [5]
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);
}
```

6.3.3 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;
}
```

6.3.4 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;
}
```

6.3.5 Legendre's Formula

The exponent of p in the prime factorization [7] 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;
}
```

6.3.6 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;}
};
```

6.4 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]);
}
```

6.5 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);
  }
}
```

6.6 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];
}
```

6.24 Miscellaneous Stuff

6.24.1 Gray Code

Gray code is a binary numeral system where two successive values differ in only one bit. [9] 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;
}
```

6.24.2 Lemmas

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

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

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* $\sum_{i=l}^{r} a_i \equiv 0 \pmod{n}$.

7 Hashing

7.1 Polynomial Hashing

7.2 Fenwick Tree on Hashes

7.3 Hashing Rooted Trees for Isomorphism

In a Chinese blog [13] the following technique for checking rooted trees for isomorphism using hashing is described. Let $s(a)$ be the rooted subtree at vertex a and $v_1, v_2 \dots v_k$ 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.

7.4 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]));
  }
}
```

8 Geometry

8.1 2D Geometry

8.1.1 Helper Functions

8.1.2 Segment - Segment Intersection

8.1.3 Angle Struct

8.1.4 Center of Mass

8.1.5 Barycentric Coordinates

8.1.6 Point in Hull and Closest Pair of Points

8.1.7 Convex Hull

8.1.8 Online Convex Hull Merger

8.1.9 Convex Hull Container of Lines

8.1.10 Polygon Union

8.1.11 Minimum Circle that encloses all points

8.1.12 Convex Layers

8.1.13 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);
}
```

8.1.14 Rectangle Union

8.1.15 Rotating Calipers

8.1.16 KD Tree

8.1.17 Burkhard-Keller Tree

Burkhard-Keller Tree [16] (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 Miscellaneous Stuff

9.1 Gosper's Hack

9.2 Matrix Flips

9.3 Calendar Conversions

9.4 Hamilton Cycle with Ore Condition

9.5 Exact Cover

9.6 Roman Numerals

9.7 Group Dynamics

9.8 Graph Isomorphism

9.9 Integer coordinates on a line

9.10 Circles

9.11 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;
    }
  }
};
```

10 References

References

[1] CodeFu - 2021 Autumn - Problem GAMBIT, November 2021. [Online; accessed 3. Nov. 2022].

[2] Articulation points and bridges (Tarjan's Algorithm) - Codeforces, November 2022. [Online; accessed 4. Nov. 2022].

[3] CodeChef - Query on a tree VI, November 2022. [Online; accessed 12. Nov. 2022].

[4] Discrete Log - Algorithms for Competitive Programming, November 2022. [Online; accessed 23. Nov. 2022].

[5] Discrete Root - Algorithms for Competitive Programming, November 2022. [Online; accessed 23. Nov. 2022].

[6] Disjoint Set Union - Algorithms for Competitive Programming, November 2022. [Online; accessed 13. Nov. 2022].

[7] Factorial modulo p - Algorithms for Competitive Programming, November 2022. [Online; accessed 23. Nov. 2022].

[8] Finding articulation points in a graph in O(N+M) - Algorithms for Competitive Programming, October 2022. [Online; accessed 4. Nov. 2022].

[9] Gray code - Algorithms for Competitive Programming, November 2022. [Online; accessed 24. Nov. 2022].

[10] SPOJ.com - Problem DYNACON1, November 2022. [Online; accessed 4. Nov. 2022].

[11] SPOJ.com - Problem MATCHING, November 2022. [Online; accessed 3. Nov. 2022].

[12] UVa - Street Directions, November 2022. [Online; accessed 4. Nov. 2022].

[13] An easy-to-write tree hash that won't collide easy - peehs-moorhsum's blog, March 2023. [Online; accessed 20. Mar. 2023].

[14] Dynamic connectivity problem - Codeforces, March 2023. [Online; accessed 8. Mar. 2023].

[15] SPOJ.com - Problem ADABLOOM, March 2023. [Online; accessed 8. Mar. 2023].

[16] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, April 1973.

[17] Harold N. Gabow. An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs. *J. ACM*, 23(2):221–234, April 1976.

[18] M. R. Henzinger and M. L. Fredman. Lower Bounds for Fully Dynamic Connectivity Problems in Graphs. *Algorithmica*, 22(3):351–362, November 1998.

[19] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM J. Comput.*, July 2006.

[20] Richard M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Math.*, 23(3):309–311, January 1978.

[21] 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.

[22] 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.

[23] ShahjalalShohag. code-library, December 2022. [Online; accessed 27. Dec. 2022].