Adventure Plan

You are preparing for an adventure. The adventure is a journey from the start vertex to the end vertex, hosted on a directed acyclic graph (DAG). The DAG has n vertices numbered from 0 to $n-1$ and m edges. The starting vertex is vertex 0, and all vertices are reachable from the start.

Each directed edge from u_i to v_i has two attributes: l_i and r_i. It indicates the minimum and maximum time required to safely pass through the edge. Therefore, you may set the planned time for this edge to any integer t_i in the interval $[l_i, r_i]$.

Lots of your friends are also preparing for the adventure. Each of them will take a different route from the start to the end. To ensure safety, you hope that friends taking different routes can meet at each vertex simultaneously. That is, you want to set the planned time for each edge such that for every vertex u, all paths from the start to u have the same total time.

We call a graph safe if it satisfies the requirement. It is guaranteed that the initial graph is safe.

Task 1

You want to add some new edges to the graph to make it more interesting. You will perform q operations of "adding new edges." Each operation provides a new directed edge (u_i, v_i, l_i, r_i). You know that after adding this edge, the graph remains a directed acyclic graph, but you are not sure whether the graph is still safe. Please determine whether the graph is safe after adding this edge. If it is, add the edge to the graph. If not, you should ignore this operation.

Task 2

After all operations, you need to output the planned time for each edge (including the newly added ones) to prove that you are sure the new graph is safe.

Implementation Details

You need to implement two procedures.

The first procedure you need to implement is `add_roads`:

```
boolean[] add_roads(int32 N, int32 M, int32 Q,
    int32[] U, int32[] V,
    int32[] L, int32[] R,
    int32[] U2, int32[] V2,
    int32[] L2, int32[] R2);
```

- N: the number of vertices;
- M: the number of initial edges;
- Q: the number of operations;
- U, V: arrays of length M, where $(U[i], V[i])$ represents the i-th directed edge;
- L, R: arrays of length M, where $[L[i], R[i]]$ is the feasible interval of times for edge i;
- $U2$, $V2$, $L2$, $R2$: arrays of length Q, describing the new edges;
- This procedure is called exactly once for each test case at the beginning of the program.

The procedure should return a vector of length Q, where the i-th element is `true` if the i-th operation edge is added, or `false` otherwise.

The second procedure you need to implement is `assign_times`:

```
int32[] assign_times();
```

- This procedure is called exactly once for each test case after calling `add_roads`.

The procedure should return a vector containing the planned time t_i for each edge that is present in the final graph (in any valid order).

Constraints

- $3 \leq n \leq 500$
- $n - 1 \leq m \leq 10^5$
- $0 \leq q \leq 500$
- $0 \leq u_i < v_i < n$
- $1 \leq l_i \leq r_i \leq 10^9$

Scoring

1. Subtask 1 (7 points): $n \leq 3$
2. Subtask 2 (21 points): $q = 0$
3. Subtask 3 (12 points): $v_i = u_i + 1$
4. Subtask 4 (11 points): $l_i = r_i$
5. Subtask 5 (24 points): $n \leq 100, m \leq 100, q \leq 100$
6. Subtask 6 (25 points): No additional constraints

Examples

Example 1:

Consider the following call.

```
add_roads(4, 4, 2,
      [0, 1, 0, 0],
      [1, 3, 3, 2],
      [1, 3, 9, 6],
      [5, 7, 14, 8],
      [2, 2],
      [3, 3],
      [7, 5],
      [11, 7]);
```

The procedure should return `[false, true]`.

Consider the following input after `add_roads` is called.

```
assign_times();
```

The procedure should return `[5, 7, 12, 6, 6]`.

Sample Grader

The sample grader reads the input in the following format:

- Line 1: Three integers n, m, and q
- Next m lines: four integers u_i, v_i, l_i, r_i describing each edge
- Next q lines: four integers u_i, v_i, l_i, r_i describing each operation