PROBLEM 2 : Not The Best.

Problem Link:

https://lightoj.com/problem/not-the-best

Problem Statement:

- Given a graph with n nodes and m edges. Edges are undirected and weighted.
- Find the shortest path and also the second-shortest path from node **1** to node **n**.
- The "second-shortest" must be strictly longer than the shortest path — not equal.
- Paths may revisit nodes / edges (i.e., not necessarily simple).

Hint / Key Idea:

- Use a **modified Dijkstra** to keep track of **two best distances** for each node:
    - dist[u][0] = the shortest distance to u
    - dist[u][1] = the second-shortest distance to u
- Use a **priority queue** where entries also store whether it corresponds to "first best" or "second best" distance.
- When relaxing edges from u → v with weight w:
    - Compute alt = dist[u][k] + w for both k = 0 and k = 1 (first / second best of u).
    - If alt < dist[v][0]:
        - Update second-best of v to old dist[v][0]
        - Update dist[v][0] = alt
        - Push both new distances into the queue.
    - Else if dist[v][0] < alt < dist[v][1]:
        - Update dist[v][1] = alt
        - Push (v, second-best) into queue.

Solution Approach (Step-by-Step)

1. **Graph Representation**
    - Use adjacency list: for each node u, store (v, w) for its neighbors.
2. **Distance Arrays**
    - dist[nodes][2], where dist[u][0] = best, dist[u][1] = second-best.
    - Initialize both to "infinite" (a large value).

- Set dist[1][0] = 0 (start at node 1 with zero).

3. **Visited / Process Arrays**
   - Maintain vis[u][0] and vis[u][1] to mark whether that best / second-best state has been finalized.

4. **Priority Queue**
   - Store entries (u, state, d) where u = node, state = 0 or 1 (best or second), and d = distance.
   - The queue is sorted by d (min-heap).

5. **Modified Dijkstra Loop**
   - While queue is not empty:
     - Pop (u, state, d).
     - Skip if vis[u][state] is true.
     - Mark vis[u][state] = true.
     - For each neighbor (v, w) of u:
       - Let alt = d + w (d is dist[u][state]).
       - **Case A – New shortest for v**:
         - If alt < dist[v][0]:
           - dist[v][1] = dist[v][0] (downgrade old shortest)
           - dist[v][0] = alt
           - Push (v, 0, dist[v][0]) and (v, 1, dist[v][1]) into queue.
       - **Case B – Between shortest and second**:
         - Else if dist[v][0] < alt < dist[v][1]:
           - dist[v][1] = alt
           - Push (v, 1, alt) in queue.

6. **Answer**
   - After algorithm ends, check dist[n][1]: that is the second-shortest distance to node n.
   - Print that value.

Complexity

- Time complexity: **O((n + m) log(n + m))**, because each node has two states, and edges are relaxed possibly twice.
- Memory: O(n + m) for graph + O(n) for distance/state arrays.

Pseudocode :

```
FUNCTION solve():

  READ N, R
  IF input invalid:
    RETURN -1

  CREATE adjacency list adj[1..N]

  FOR i = 1 to R:
    READ u, v, w
    ADD (v, w) to adj[u]
    ADD (u, w) to adj[v]

  INITIALIZE dist1[1..N] = INF
  INITIALIZE dist2[1..N] = INF

  CREATE min-heap priority queue pq

  dist1[1] = 0
  PUSH (0, 1) INTO pq       // (distance, node)

  WHILE pq is NOT empty:

    (d, u) = pq.pop()

    IF d > dist2[u]:
      CONTINUE

    FOR each edge (u → v) with weight w in adj[u]:

      d_new = d + w

      // Case 1: Found a better shortest path
      IF d_new < dist1[v]:
```