**🏩 Leetcode 1976: Number of Ways to Arrive at Destination**

**Difficulty:** Medium  
**Topics:** Graph, Dijkstra’s Algorithm, Shortest Path, Priority Queue

**📌 Problem Statement**

* Given **n** intersections labeled 0 to n-1 and **bidirectional roads**, each with a **travel time**.
* Find the **shortest time** from **intersection 0 to n-1**.
* Count the **number of ways** to reach n-1 in that shortest time.
* Return the result **modulo 10^9 + 7**.

**Example**

**Input:**

n = 7, roads = [[0,6,7],[0,1,2],[1,2,3],[1,3,3],[6,3,3],

[3,5,1],[6,5,1],[2,5,1],[0,4,5],[4,6,2]]

**Output:**

4

**Explanation:**

The **shortest time** from 0 to 6 is **7 minutes**, and there are **4 ways** to achieve this.

**🔹 Approach**

**1️⃣ Graph Representation**

* Convert roads into an **adjacency list** storing **neighbors and travel time**.

**2️⃣ Dijkstra’s Algorithm (Min-Heap)**

* Use **Dijkstra’s Algorithm** to find the **shortest time**.
* Maintain a **priority queue (min-heap)** to process **nodes with the shortest distance first**.

**3️⃣ Tracking Shortest Paths**

Maintain **two arrays**:

* dist[]: Stores the **shortest time** to reach each node.
* ways[]: Stores the **number of ways** to reach each node using the **shortest path**.

**4️⃣ Update Conditions**

* If a **shorter path** is found → **Update dist[] and reset ways[]**.
* If an **equal shortest path** is found → **Add to ways[]**.

**✅ Java Solution**

import java.util.\*;

class Solution {

public int countPaths(int n, int[][] roads) {

int MOD = 1\_000\_000\_007;

// Step 1: Create adjacency list

List<List<int[]>> graph = new ArrayList<>();

for (int i = 0; i < n; i++) {

graph.add(new ArrayList<>());

}

for (int[] road : roads) {

int u = road[0], v = road[1], time = road[2];

graph.get(u).add(new int[]{v, time});

graph.get(v).add(new int[]{u, time});

}

// Step 2: Min-Heap for Dijkstra's Algorithm

PriorityQueue<long[]> pq = new PriorityQueue<>(Comparator.comparingLong(a -> a[0]));

pq.offer(new long[]{0, 0}); // (travel\_time, node)

// Step 3: Distance array to track shortest time

long[] dist = new long[n];

Arrays.fill(dist, Long.MAX\_VALUE);

dist[0] = 0;

// Step 4: Ways array to count shortest paths

int[] ways = new int[n];

ways[0] = 1;

while (!pq.isEmpty()) {

long[] curr = pq.poll();

long currTime = curr[0];

int node = (int) curr[1];

if (currTime > dist[node]) continue; // Ignore outdated paths

for (int[] neighbor : graph.get(node)) {

int nextNode = neighbor[0];

long newTime = currTime + neighbor[1];

if (newTime < dist[nextNode]) {

dist[nextNode] = newTime;

ways[nextNode] = ways[node]; // Reset count

pq.offer(new long[]{newTime, nextNode});

} else if (newTime == dist[nextNode]) {

ways[nextNode] = (ways[nextNode] + ways[node]) % MOD; // Add count for equal shortest path

}

}

}

return ways[n - 1]; // Return number of ways to reach destination

}

}

**📊 Complexity Analysis**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| **Graph Construction** | **O(E)** | E is the number of edges. |
| **Dijkstra’s Algorithm** | **O((V + E) log V)** | Min-heap optimization. |
| **Overall Complexity** | **O(E log V)** | Efficient for n ≤ 200. |

**📌 Edge Cases Considered**

✅ **Single road scenario:** n = 2, roads = [[1, 0, 10]]  
✅ **Multiple shortest paths exist**  
✅ **Large values of timei**  
✅ **Graph with max constraints (n = 200, E ≈ 20,000)**

**🚀 Key Takeaways**

✔ **Dijkstra’s Algorithm** is ideal for **finding the shortest path** in weighted graphs.  
✔ **Tracking ways[] alongside dist[]** helps count **number of shortest paths**.  
✔ **Using a Min-Heap (Priority Queue)** ensures **efficiency**.  
✔ **Modulo (10^9 + 7)** prevents **integer overflow**.