Solution 2:

Understanding the question

Compute shortest distances from city 1 in a graph with roads and "rail" links from 1→v. Remove as many rails as possible **without changing any** distances.

Approach

- Run **Dijkstra** on the full graph (roads + rails).
- For each city v, track `minRail[v]` (cheapest rail to v) and `cntRail[v]` (how many).
- Post-scan only **road** edges to mark `hasRoadPred[v]` if some road `(u→v,w)` satisfies `dist[u]+w==dist[v]`.

What went wrong

- Used sets/maps to infer "is rail" by endpoints, which hid **parallel road edges** between {1,v}.
- Assumed `dist[v]==rail_w` ⇒ rail needed, missing **road-only ties**.
- Risked overflow with `int` distances in PQ.

Fixes

- Single adjacency list with `rail` flag per edge.
- Standard Dijkstra with `(long long dist, node)` and relax-on-better.
- After Dijkstra, mark `hasRoadPred` using only road edges.

Decision rule (per v)

1. `minRail[v] > dist[v]` → remove **all** rails to v.
2. `minRail[v] == dist[v]` & `hasRoadPred[v]` → remove **all** rails to v.
3. Else (rail required) → keep **one** minimal rail, remove `cntRail[v]-1`.

What would have prevented issues

- Write the three-case rule first; model data to test it directly.
- Tag edges with `rail` from the start; avoid endpoint heuristics.

- Use `long long`; add tiny tests for worse/equal rails and parallel road+rail.

Outcome

One Dijkstra + linear post-scan yields correct removals, handles duplicates/ties, and preserves all shortest distances.