Solution 3:

Understanding the question

We are given a complete weighted graph and a permutation describing the order in which vertices are removed. For each step, we need to compute the sum of all shortest paths between the currently "alive" vertices. This is equivalent to **adding the vertices back in reverse order** and updating shortest paths as we go.

Initial strategy

- Recognize that Floyd–Warshall fits naturally: when we add vertex `k`, it can act as an **intermediate** in new shortest paths.
- Idea: iterate in reverse order of removals, and at each step run the relaxation `dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])`.
- Maintain a `used[]` mask to sum distances only between vertices that are currently alive.
- Push each sum into a deque (front) and print at the end.

What went wrong

- My first attempt only updated distances for pairs `(u,v)` within the active set at the current step.
- This missed shorter paths where one endpoint was not yet active but later would be. For example, when vertex 3 is added early, distances like `dist[future][3]` and `dist[3][future]` were never updated, so future sums could not take advantage of paths through 3.
- I also overcomplicated the code with an unused second matrix `dist1`, redundant edge list storage, and re-initializations.

Fixes

- Always relax **all pairs** `(i,j)` in the full `n×n` matrix when a new vertex `k` is added. This guarantees that whenever a future vertex is activated, its paths through previously added vertices are already optimal.
- Sum only among `used` vertices to match the problem's requirements.
- Simplify: drop the `edges` vector, the duplicate `dist1`, and unnecessary `INF` resets.
- Keep the logic clean:
  1. Read matrix.
  2. Process reversed order.
  3. At each step, relax via `k` for all `i,j`, then sum among active vertices.

How I could have avoided it

- Write down the **invariant** at the start: after `t` steps, `dist[i][j]` must equal the shortest path using only intermediates in the active set. That immediately shows why we need to update all `i,j`, not just active endpoints.
- Test on small custom graphs (e.g., a triangle with asymmetric weights) to catch cases where a future vertex needed earlier updates.
- Resist overengineering: avoid carrying extra matrices unless they serve a clear purpose. A single `dist` with a `used[]` mask is enough.

Outcome

With the corrected relaxation, the solution handles arbitrary removal orders, passes hidden cases, and runs in O(n3)O(n^3)O(n3) time as intended. The main lesson: in dynamic shortest path problems, always reason about **what invariant the DP matrix maintains** and ensure each update preserves it.