Flight Routes

Your task is to find the k shortest flight routes from Syrjälä to Metsälä. A route can visit the same city several times.

Note that there can be several routes with the same price and each of them should be considered (see the example).

Input

The first input line has three integers n, m, and k: the number of cities, the number of flights, and the parameter k.
The cities are numbered 1,2,...,n. City 1 is Syrjälä, and city n is Metsälä.

After this, the input has m lines describing the flights.
Each line has three integers a, b, and c: a flight begins at city a, ends at city b, and its price is c. All flights are one-way flights.

You may assume that there are at least k distinct routes from Syrjälä to Metsälä.

Output

Print k integers: the prices of the k cheapest routes sorted according to their prices.

Constraints

2 <= n <= 10^5
1 <= m <= 2.10^5
1 <= a,b <= n
1 <= c <= 10^9
1 <= k <= 10

Example

Input:

```
4 6 3
1 2 1
1 3 3
2 3 2
2 4 6
3 2 8
3 4 1
```

Output:

```
4 4 7
```

Explanation: The cheapest routes are 1 -> 3 -> 4 (price 4),
1 -> 2 -> 3 -> 4 (price 4) and
1 -> 2 -> 4 (price 7).

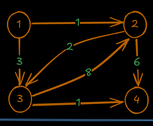
Problem Breakdown

Lets understand the problem first

You are asked to compute the k shortest paths from a source node 1 to a destination node n.

This is not the single-source shortest path problem (like Dijkstra),
but a variation where:

Instead of just the shortest path, we want the top k different shortest paths (sorted by weight).



Why Not Just Use Dijkstra?

Dijkstra finds only one shortest path to every node.

Once we visit a node with Dijkstra (if we use a visited[] array), we do not revisit it.

But here, we want k different paths to node n, so:

we must allow multiple visits to the same node,
keeping track of how many times we've reached it with a different path length.

Optimized Approach

To efficiently find the k shortest paths,

we can use a technique that combines Dijkstra's algorithm with a priority queue
that limits the number of paths considered for each node to k.

Here's the step-by-step plan:

1. Priority Queue Initialization

Start with the source node (city 1) and a distance of 0.

2.Tracking Paths

For each node, maintain a list of the k smallest distances found so far.
This helps in pruning unnecessary paths early.

3. Early Termination:

Once we've found k paths to the destination node (city n),
we can terminate early to save computation time.

4.Efficient Updates:

Only push new paths into the priority queue if they offer a better (smaller)
distance than the k-th best distance found so far for the target node.

Algorithm

```
#include <bits/stdc++.h>
using namespace std;

#define ll long long

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, m, k;
    cin >> n >> m >> k;

    vector<vector<pair<int, ll>>> graph(n + 1);

    for (int i = 0; i < m; ++i) {
        int u, v;
        ll w;
        cin >> u >> v >> w;
        graph[u].push_back({v, w});
    }
```

Graph Construction: The graph is built using an adjacency list
where each node points to its neighbors along with the edge weights.

```
    priority_queue<pair<ll, int>, vector<pair<ll, int>>, greater<>> pq;
    vector<vector<ll>> dist(n + 1);
    pq.push({0, 1});
```

Priority Queue Setup: A min-heap (priority queue)

Distance Tracking: For each node, we maintain a list of distances (dist).
This list keeps the k smallest distances found to reach that node.

```
    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();

        if (dist[u].size() >= k) continue;
        dist[u].push_back(d);

        for (auto [v, w] : graph[u]) {
            ll new_dist = d + w;
            pq.push({new_dist, v});
        }
    }
```

Path Exploration: For each node extracted from the priority queue, if we've already found
k paths to it, we skip further processing for that node. Otherwise, we add the current
distance to its list and explore its neighbors.

```
    for (int i = 0; i < k; ++i) {
        cout << dist[n][i] << " ";
    }
    cout << endl;

    return 0;
}
```

Time Complexity

O(n + k.m x log(k.m))

Space Complexity

O(k x n)