Shortest Routes I

There are n cities and m flight connections between them. Your task is to determine the length of the shortest route from Syrjälä to every city.

Input

The first input line has two integers n and m: the number of cities and flight connections.
The cities are numbered 1,2,...,n, and city 1 is Syrjälä.

After that, there are m lines describing the flight connections.
Each line has three integers a, b and c: a flight begins at city a, ends at city b, and its length is c. Each flight is a one-way flight.

You can assume that it is possible to travel from Syrjälä to all other cities.

Output

Print n integers: the shortest route lengths from Syrjälä to cities 1,2,...,n.

Constraints

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

Example

Input:
```
3 4
1 2 6
1 3 2
3 2 3
1 3 4
```

Output:
```
0 5 2
```

Problem Breakdown

Lets understand the problem first

You're given n cities and m one-way roads.

Each road goes from city a to city b and has a cost c.

Find the minimum total cost to travel from city 1 to every other city (1 to n).

Recognize the Pattern

This is a classic single-source shortest path problem on a weighted directed graph.

- Graph is weighted ✅
- No negative weights (implied) ✅
- Need shortest path from node 1 to all others ✅

Dijkstra's Algorithm is the perfect match.

Naive Approach — Why Not Use BFS?

BFS works great for unweighted graphs. If every road had equal cost (like all cost = 1), we could just do BFS and track the number of steps.

But here, each road has a different cost. For example:

- 1 → 2 → 4 might cost 100
- 1 → 3 → 4 might cost 10

BFS won't prioritize the lower-cost path, it only looks at step count. So:

❌ BFS is invalid here

Key Insight — Use Dijkstra's Algorithm

Dijkstra helps us find the minimum-cost path in a weighted graph, when all edge weights are non-negative (like tolls, fuel, time).

It always expands the cheapest city reachable at each step and uses that to discover new paths.

How Dijkstra's Algorithm Works (Intuition)

Imagine you're in city 1 and you want to go to all other cities.

You begin by saying:

"I know that reaching city 1 costs 0. All others? I have no idea — let's say it's infinity for now."

You then say:

"From city 1, where can I go? Let's update those cities' costs."

Then always pick the city with the lowest cost so far, and explore from there.

Repeat this until you've updated distances to all cities.

Data Structures You'll Need

Structure	Why?
vector<vector<pair<int,int>>> adj	To store roads (graph) — from city to neighbors
vector<long long> dist(n+1, INF)	Distance to each city from city 1
priority_queue (min-heap)	Always pick the city with the smallest current cost

Algorithm

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

#define int long long
const long long INF = 1e18;

int32_t main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

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

    // Graph: adjacency list of (neighbor, cost)
    vector<vector<pair<int, int>>> graph(n + 1);

    for (int i = 0; i < m; i++) {
        int u, v, cost;
        cin >> u >> v >> cost;
        graph[u].push_back({v, cost}); // directed edge
    }

    // Distance from node 1 to all others
    vector<long long> dist(n + 1, INF);
    dist[1] = 0;

    // Min-heap of (current_dist, node)
    priority_queue<pair<long long, int>, vector<pair<long long, int>>, greater<>> pq;
    pq.push({0, 1});

    while (!pq.empty()) {
        auto [currDist, node] = pq.top();
        pq.pop();

        // Skip if we already have a better distance
        if (currDist > dist[node]) continue;

        for (auto [neighbor, weight] : graph[node]) {
            if (dist[node] + weight < dist[neighbor]) {
                dist[neighbor] = dist[node] + weight;
                pq.push({dist[neighbor], neighbor});
            }
        }
    }

    for (int i = 1; i <= n; i++) {
        cout << dist[i] << " ";
    }

    return 0;
}
```

Mark distance to start city (1) as 0, all others as INF

Push (0, 1) into min-heap + cost 0, city 1

While heap is not empty:
  a. Pop the city with smallest current cost
  b. For each neighbor:
    - If going through current city gives shorter path
      - update neighbor's cost
      - push (new_cost, neighbor) into heap

Time Complexity

O((N + M) x log N)

- Initialize distance array -> O(N)
- Push/Pop in heap -> O(log(N))
- Process all edges -> O(M log N)

Space Complexity

O(N + M)