Shortest Routes II

There are n cities and m roads between them.
Your task is to process q queries where you have to determine the length of the shortest route between two given cities.

Input

The first input line has three integers n, m and q: the number of cities, roads, and queries.

Then, there are m lines describing the roads.
Each line has three integers a, b and c: there is a road between cities a and b whose length is c. All roads are two-way roads.

Finally, there are q lines describing the queries.
Each line has two integers a and b: determine the length of the shortest route between cities a and b.

Output

Print the length of the shortest route for each query. If there is no route, print -1 instead.

Constraints

1 <= n <= 500
1 <= m <= n^2
1 <= q <= 10^5
1 <= a,b <= n
1 <= c <= 10^9

Example

Input:

```
4 3 5
1 2 5
1 3 9
2 3 3
1 2
2 1
1 3
1 4
3 2
```

Output:

```
5
5
8
-1
3
```

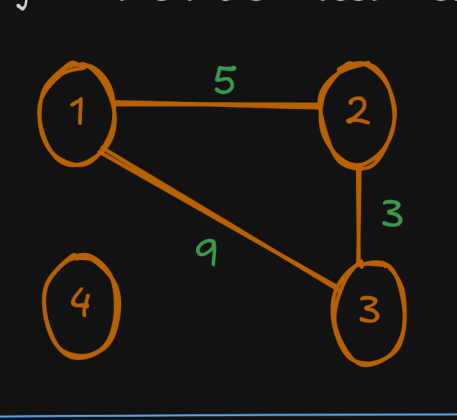
Problem Breakdown

Lets understand the problem first

You're given n cities(nodes) and m bidirectional roads(edges).

Each road has a cost c.

q queries asking: What is the shortest cost between city a and city b?



Why not use Dijkstra?

There are two ways to try using Dijkstra here — both fail in this scenario:

1. Run Dijkstra for Every Query

For each query (u → v), run a fresh Dijkstra.

Time Complexity:
$10^5 \times (E + V \log V)$ → ❌ Too slow, leads to TLE

2. Precompute from Every Node

Run Dijkstra V times (once from every node)

Store shortest distances in a matrix

Time Complexity:
$V \times (E + V \log V)$
For dense graphs → $O(V^3 \log V)$

🚫 Drawbacks:

Cannot handle negative weights

Still not fast enough for $n = 500$, $m \approx 10^5$

Requires priority queue and repeated logic — more complex

Floyd-Warshall Algorithm

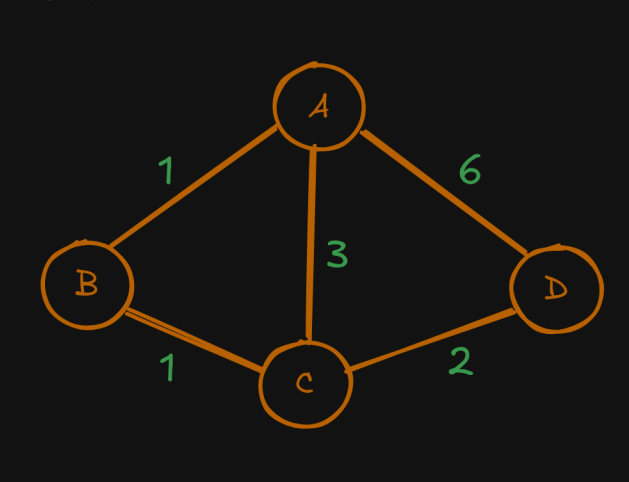
The Floyd-Warshall algorithm is used to find the shortest paths between all pairs of vertices in a weighted graph.

It works for both directed and undirected graphs, and can handle negative weights (but not negative cycles).

Uses Dynamic Programming to build up solution step by step

How It Works - Step by Step

Let's use a simple example graph with 4 vertices (A, B, C, D):



Step 1: Create the Initial Distance Matrix

	A	B	C	D
A	0	1	3	6
B	1	0	1	∞
C	3	1	0	2
D	6	∞	2	0

We'll represent the graph as a 4x4 matrix where:

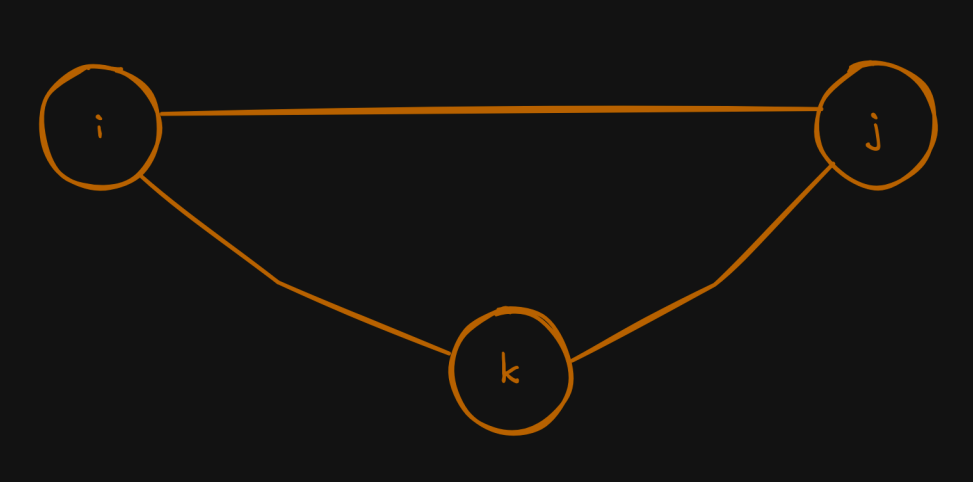
Diagonal elements (distance from a vertex to itself) are 0

Direct edges have their weight

No direct connection is represented by ∞ (infinity)

Step 2: The Main Idea

The algorithm says that if there is a path from i -> j then we can also reach to j from some intermediate nodes like here i -> k then k -> j , if this path is smaller we take this this is called relaxing of edges.



The algorithm works by considering each vertex as an intermediate point (k)
and checking if going through k gives a shorter path between every pair of vertices (i, j).

```
D[i][j] = min(D[i][j], D[i][k] + D[k][j])
```

Step 3: Iterations

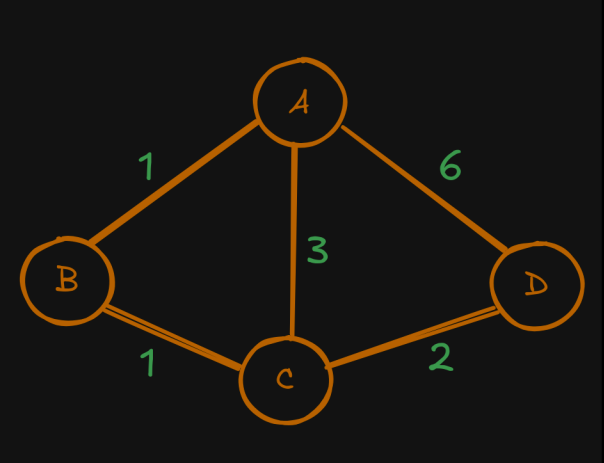
We'll do 4 iterations (one for each vertex as the intermediate):

Iteration 1 (k = A - using A as intermediate):

	A	B	C	D
A	0	1	3	6
B	1	0	1	∞
C	3	1	0	2
D	6	∞	2	0

→

	A	B	C	D
A	0	1	3	6
B	1	0	1	7
C	3	1	0	2
D	6	7	2	0

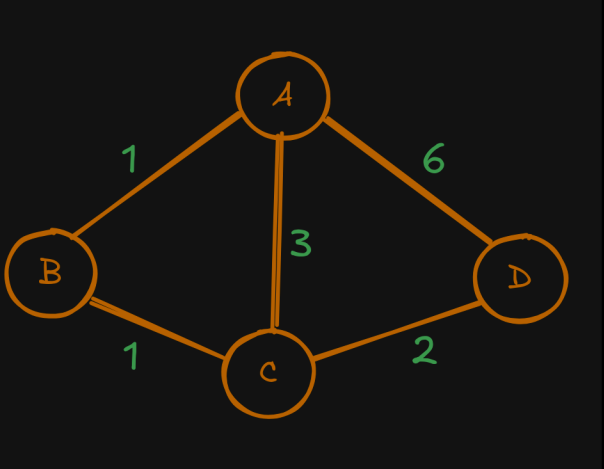


Iteration 2 (k = B - using B as intermediate):

	A	B	C	D
A	0	1	3	6
B	1	0	1	7
C	3	1	0	2
D	6	7	2	0

→

	A	B	C	D
A	0	1	2	6
B	1	0	1	7
C	2	1	0	2
D	6	7	2	0

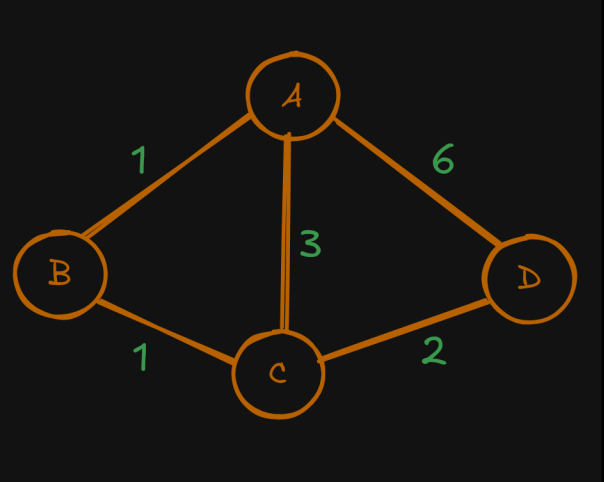


Iteration 3 (k = C - using C as intermediate):

	A	B	C	D
A	0	1	2	6
B	1	0	1	7
C	2	1	0	2
D	6	7	2	0

→

	A	B	C	D
A	0	1	2	4
B	1	0	1	3
C	2	1	0	2
D	4	3	2	0

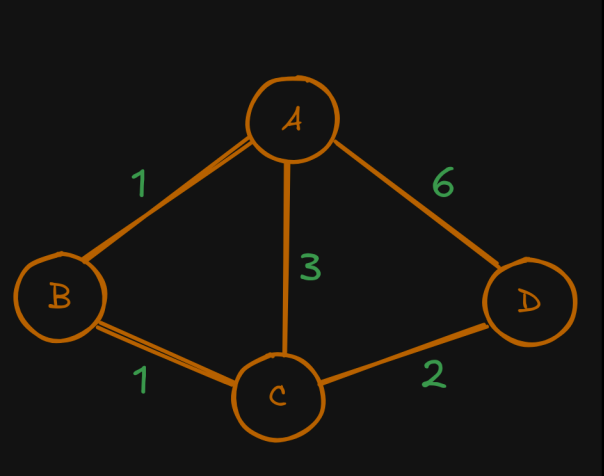


Iteration 4 (k = D - using D as intermediate):

	A	B	C	D
A	0	1	2	4
B	1	0	1	3
C	2	1	0	2
D	4	3	2	0

→

	A	B	C	D
A	0	1	2	4
B	1	0	1	3
C	2	1	0	2
D	4	3	2	0



Algorithm

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

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

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

    vector<vector<long long>> dist(n, vector<long long>(n, LLONG_MAX));

    for (int i = 0; i < n; i++) {
        dist[i][i] = 0;  // Distance from a node to itself is zero
    }

    for (int i = 0; i < m; i++) {
        int u, v;
        long long wt;
        cin >> u >> v >> wt;
        u--; v--;  // Adjusting for 0-based indexing
        dist[u][v] = min(dist[u][v], wt);
        dist[v][u] = min(dist[v][u], wt);  // If the graph is bidirectional
    }

    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (dist[i][k] != LLONG_MAX && dist[k][j] != LLONG_MAX) {
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }
    }

    for (int i = 0; i < q; i++) {
        int x, y;
        cin >> x >> y;
        x--; y--;  // Adjusting for 0-based indexing
        cout << (dist[x][y] == LLONG_MAX ? -1 : dist[x][y]) << endl;
    }

   return 0;
}
```

→ Initialize the Distance Matrix

→ Reading graph input

→ Run Floyd-Warshall

Time Complexity

The algorithm runs in $O(V^3)$ time because of the three nested loops (each running |V| times).

Space Complexity

$O(V^2)$

When to Use Floyd-Warshall

When you need shortest paths between all pairs of vertices

When the graph may contain negative weights (but no negative cycles)

When the graph isn't too large (since it's $O(V^3)$)

Limitations

Doesn't work with negative cycles (cycles where the sum of weights is negative)

Not as efficient as Dijkstra's for single-source shortest path problems

$O(V^3)$ time makes it impractical for very large graphs