**Problem Statement**

**Shortest Path in Directed Acyclic Graph (DAG)**

Given a Directed Acyclic Graph (DAG) with NNN vertices numbered from 0 to N−1N-1N−1 and a 2D Integer array (or vector) edges of length MMM, where there is a directed edge from edges[i][0] to edges[i][1] with a distance of edges[i][2] for all iii.

Find the shortest path from the source vertex (0) to all the vertices. If it is impossible to reach any vertex, then return -1 for that vertex.

**Input Format**

1. The first line contains an integer NNN denoting the number of vertices.
2. The second line contains an integer MMM denoting the number of edges.
3. The next MMM lines each contain three integers uuu, vvv, and www, indicating there is a directed edge from vertex uuu to vertex vvv with weight www.

**Output Format**

Return an integer array (or vector), denoting the list of distances from the source (0) to all nodes.

**Constraints**

* 1<= N<=100
* 1 <= M <= min(2N(N−1)​,4000)
* 0 <=edge[i][0],edge[i][1]<N
* 0<=edge[i][2]<=10^5

**Sample Input and Output**

**Example 1:**

**Input:**

4

2

0 1 2

0 2 1

**Output:**

0 2 1 -1

**Explanation:**

* Shortest path from 0 to 1 is 0 -> 1 with edge weight 2.
* Shortest path from 0 to 2 is 0 -> 2 with edge weight 1.
* There is no way to reach vertex 3, so it's -1 for 3.

**Example 2:**

**Input:**

6

7

0 1 2

0 4 1

4 5 4

4 2 2

1 2 3

2 3 6

5 3 1

**Output:**

0 2 3 6 1 5

**Explanation:**

* Shortest path from 0 to 1 is 0 -> 1 with edge weight 2.
* Shortest path from 0 to 2 is 0 -> 4 -> 2 with edge weight 1 + 2 = 3.
* Shortest path from 0 to 3 is 0 -> 4 -> 5 -> 3 with edge weight 1 + 4 + 1 = 6.
* Shortest path from 0 to 4 is 0 -> 4 with edge weight 1.
* Shortest path from 0 to 5 is 0 -> 4 -> 5 with edge weight 1 + 4 = 5.

**Solution**

To solve this problem, we use topological sorting followed by a shortest path calculation. Here's the implementation:

python

from collections import defaultdict, deque

import sys

def shortest\_path\_in\_dag(N, M, edges):

# Create adjacency list and in-degree array

graph = defaultdict(list)

in\_degree = [0] \* N

for u, v, w in edges:

graph[u].append((v, w))

in\_degree[v] += 1

# Topological sort using Kahn's algorithm

topo\_order = []

queue = deque([i for i in range(N) if in\_degree[i] == 0])

while queue:

node = queue.popleft()

topo\_order.append(node)

for neighbor, \_ in graph[node]:

in\_degree[neighbor] -= 1

if in\_degree[neighbor] == 0:

queue.append(neighbor)

# Initialize distances

distances = [sys.maxsize] \* N

distances[0] = 0

# Relax edges according to topological order

for u in topo\_order:

if distances[u] != sys.maxsize:

for v, w in graph[u]:

if distances[u] + w < distances[v]:

distances[v] = distances[u] + w

# Replace sys.maxsize with -1 for unreachable nodes

return [d if d != sys.maxsize else -1 for d in distances]

# Input reading

N = int(input())

M = int(input())

edges = [tuple(map(int, input().split())) for \_ in range(M)]

# Processing and Output

result = shortest\_path\_in\_dag(N, M, edges)

print(" ".join(map(str, result)))

**Test Cases**

**Test Case 1:**

makefile

Input:

6

7

0 1 2

0 4 1

4 5 4

4 2 2

1 2 3

2 3 6

5 3 1

Output:

0 2 3 6 1 5

**Test Case 2:**

makefile

Input:

4

2

0 1 2

0 2 1

Output:

0 2 1 -1

**Test Case 3:**

makefile

Input:

6

6

0 4 1

4 5 4

4 2 2

1 2 3

2 3 6

5 3 1

Output:

0 -1 3 6 1 5

**Test Case 4:**

makefile

Input:

6

5

0 4 6

4 5 6

1 2 6

2 3 6

5 3 6

Output:

0 -1 -1 18 6 12

**Test Case 5:**

makefile

Input:

4

2

2 3 6

5 3 6

Output:

0 -1 -1 -1