Lab 9

Problem 1

You are helping to design a **navigation system** for a small city. The city has **5 important locations**, connected by **5 one-way roads**. Your task is to calculate the **shortest time (in minutes)** it takes to reach each location from the **Central Station** (Location 1).

Each road goes **from one location to another**, and has a certain **travel time**.

Here is the road map:

```
Number of locations: 5
Number of roads: 5
Roads:
1 → 2 (5 minutes)
1 → 3 (2 minutes)
3 → 4 (1 minute)
1 → 4 (6 minutes)
3 → 5 (5 minutes)
```

Question:

Use **Dijkstra's Algorithm** to find the **shortest travel time** from the **Central Station (Location 1)** to each of the other locations (Locations 2 to 5).
Show step-by-step how the algorithm updates distances.
At the end, write the final travel times separated by spaces. If a location can't be reached, write `"INF"`.

Answer

Step 0: Initialization

We apply **Dijkstra's Algorithm** starting from **Location 1 (Central Station)**.

We initialize the shortest distances from Location 1 as follows:

```
Distance[1] = 0       (starting point)
Distance[2..5] = ∞    (unknown at the beginning)
```

We also maintain a **priority queue (min-heap)** that always selects the next closest location to process.

Road Map:

```
1 → 2 (5)
1 → 3 (2)
3 → 4 (1)
1 → 4 (6)
3 → 5 (5)
```

Step-by-Step Execution

Step 1: Start at Location 1

- Distance[1] = 0
- Explore neighbors from Location 1:
    - 1 → 2: 0 + 5 = 5 → update Distance[2] = 5
    - 1 → 3: 0 + 2 = 2 → update Distance[3] = 2
    - 1 → 4: 0 + 6 = 6 → update Distance[4] = 6

Priority queue: [(2, 3), (5, 2), (6, 4)]

Current distances:

```
[0, 5, 2, 6, ∞]
```

Step 2: Visit Location 3 (distance 2)

- Explore neighbors from Location 3:
    - 3 → 4: 2 + 1 = 3 → update Distance[4] = 3 (better than previous 6)
    - 3 → 5: 2 + 5 = 7 → update Distance[5] = 7

Priority queue: [(3, 4), (5, 2), (7, 5)]

Current distances:

```
[0, 5, 2, 3, 7]
```

Step 3: Visit Location 4 (distance 3)

- Location 4 has no outgoing roads → nothing to update

Priority queue: [(5, 2), (7, 5)]

Step 4: Visit Location 2 (distance 5)

- Location 2 has no outgoing roads → nothing to update

Priority queue: [(7, 5)]

Step 5: Visit Location 5 (distance 7)

- Location 5 has no outgoing roads → nothing to update

Priority queue: []

Final Shortest Travel Times:

From Location 1 to:

- Location 2: 5 minutes
- Location 3: 2 minutes
- Location 4: 3 minutes
- Location 5: 7 minutes

Final Output: `5 2 3 7`

Problem 2

You are managing a project that consists of **5 tasks**, and some tasks can only be started **after others are completed**. These dependencies form a **Directed Acyclic Graph (DAG)** — meaning the tasks must follow a one-way, non-circular sequence.

Each task is represented by a number from `1` to `5`. If task `X` must be completed before task `Y`, it is written as a directed edge `X → Y`.

Below is the structure of your project:

```
Number of tasks: 5
Number of dependencies: 6
Dependencies:
1 → 2
1 → 3
2 → 3
2 → 4
3 → 4
3 → 5
```

Your goal is to determine a **valid order to complete all tasks** such that every task is started **only after all its prerequisite tasks** have been completed.
If there are **multiple valid orders**, print the one that is **lexicographically smallest** (i.e., appears first if you sort all valid results like words in a dictionary).

Question:

Simulate the **topological sort step by step**.
At each step, choose the available task with the **smallest number** (lowest task ID) among those with no remaining prerequisites.

At the end, print the topological order — a single line of task numbers in the required order.

Answer

Why Topological Sort?

We are given a list of **tasks with dependencies**, meaning certain tasks must be completed before others. This kind of dependency structure forms a **Directed Acyclic Graph (DAG)**. Our goal is to find an **ordering of tasks** such that **every task is scheduled only after all its prerequisites are done**.

This is exactly what **topological sort** is designed to do:

> A topological sort of a DAG is a linear ordering of its vertices such that for every directed edge `u → v`, `u` comes before `v` in the ordering.

Why Use a Min-Heap?

In this problem, **multiple valid topological orders** may exist. For example, if both Task 2 and Task 3 are available at the same time, either could be chosen.

But the question specifically asks for the **lexicographically smallest** order — this means we should always choose the **lowest-numbered available task** at each step.

To achieve this, we use a **min-heap (priority queue)**:

- It allows us to always access the **smallest available task number** in O(log N) time.
- This ensures that at every step, among all the available tasks with no prerequisites, we choose the one with the **smallest ID**.

Step-by-Step Execution

Given:

```
Number of tasks: 5
Dependencies:
1 → 2
1 → 3
2 → 3
2 → 4
3 → 4
3 → 5
```

Step 1: Build Graph and In-Degree Array

We represent the graph using an adjacency list and track the number of prerequisites (**in-degree**) for each task.

```
Adjacency list:
1: [2, 3]
2: [3, 4]
3: [4, 5]

In-degree:
Task 1: 0
Task 2: 1 (from 1)
Task 3: 2 (from 1 and 2)
Task 4: 2 (from 2 and 3)
Task 5: 1 (from 3)
```

Step 2: Initialize the Min-Heap

We start with all tasks that have in-degree = 0 → only Task 1
→ Min-Heap = `[1]`

The Core Idea: Why We Decrease In-Degree and Push to Heap

Each task's **in-degree** shows how many tasks must be completed **before** it can start.
When we **process a task** (i.e., pop it from the heap), we treat it as "completed."

For each neighbor `v` of that task (each task that depends on it), we do:

```
in_degree[v] -= 1
```

This reflects that **one prerequisite has been completed**.
If `in_degree[v]` becomes 0, that means **all prerequisites are done**, and it's now **ready to be scheduled** — so we **push it to the heap**.

Using the **min-heap** ensures we always choose the **smallest task ID** among available options, which guarantees the **lexicographically smallest topological order**.

Step-by-Step Topological Sort

1. **Pop 1** from heap → result: `[1]`
    - Task 2: `in_degree[2] -= 1` → becomes 0 → push 2 to heap
    - Task 3: `in_degree[3] -= 1` → becomes 1
      → Heap = `[2]`
2. **Pop 2** → result: `[1, 2]`
    - Task 3: `in_degree[3] -= 1` → becomes 0 → push 3
    - Task 4: `in_degree[4] -= 1` → becomes 1
      → Heap = `[3]`
3. **Pop 3** → result: `[1, 2, 3]`
    - Task 4: `in_degree[4] -= 1` → becomes 0 → push 4

  - Task 5: `in_degree[5] -= 1` → becomes 0 → push 5
    → Heap = `[4, 5]`

4. **Pop 4** (smallest of 4 and 5) → result: `[1, 2, 3, 4]`

   - No neighbors to process
     → Heap = `[5]`

5. **Pop 5** → result: `[1, 2, 3, 4, 5]`

   - No neighbors to process
     → Heap is empty

Final Output: `1 2 3 4 5`

This is a **valid topological order**, and because we used a **min-heap**, it's also the **lexicographically smallest** possible.

Problem 3

In the near future, a **smart city** is equipped with **automated roads** that **disappear after a certain time**. These roads are used for transportation, and their existence is time-dependent — they automatically become **inactive** at certain times.

Your task is to determine the **minimum time required to reach each district** from the **central hub (District 0)** before the **road to that district disappears**. If a district is unreachable before the road disappears, return `-1`.

You are given:

1. A **city map** consisting of **districts** connected by **automated roads** (edges) with traversal times.
2. An array indicating when each **district becomes inactive** (its corresponding road disappears).

The problem is to compute the **minimum time to reach each district** from the central hub (District 0) before the roads to them become unavailable.

Example Input:

```
Number of districts: 3
Number of roads: 3

Starting district: 1

Target district: 3


Roads:

0 → 1 (2 minutes)

1 → 2 (1 minutes)

0 → 2 (4 minute)


Disappear times:

District 0: 1

District 1: 3

District 2: 5
```

Questions:

1. **Describe the algorithm** used to find the minimum time required to reach each district from the Central District (node 0) before the road disappears.
2. **Simulate the process step-by-step** for the given example, showing how time is updated and how the disappearance times of the roads affect the journey.

Answer

Why Use Dijkstra's Algorithm?

In this problem, we need to find the minimum time required to reach each district from the central district (District 0) before the road to that district disappears. This introduces a time-dependent constraint: roads become inactive after certain times, so we need to ensure that we reach each district before its road becomes unavailable.

We can solve this problem using **Dijkstra's algorithm** because:

1. **Shortest Path**: Dijkstra's algorithm is designed to find the shortest path in a graph with non-negative edge weights. In this case, the edge weights are the travel times for the roads.

2. **Priority Queue**: The algorithm uses a priority queue (min-heap) to always process the node with the smallest accumulated time first. This is ideal for ensuring we always find the quickest path to each district.

3. **Time Constraints**: We can modify Dijkstra's algorithm to take the road disappearance times into account by skipping nodes whose current time exceeds their disappearance time.

In short, Dijkstra's algorithm is efficient for finding the shortest path in graphs, and with slight modification, it can handle the time-sensitive nature of this problem.

Step-by-Step Solution

1. **Graph Representation**:
    - We represent the city's districts as a graph, where nodes represent the districts and edges represent the roads between them, with the associated travel times.

2. **Priority Queue (Min-Heap)**:
    - We use a priority queue to process nodes starting from the central district (District 0), ensuring that we always process the district with the smallest time required to reach it.

3. **Visited and Disappearance Check**:
    - We track which districts have been visited to avoid processing the same district multiple times.
    - For each district, we also check if the current time to reach it exceeds its disappearance time. If it does, we skip that district.

4. **Neighbor Exploration**:
    - For each district, we explore its neighbors (connected districts), calculate the time it takes to reach them, and update the minimum time if a shorter path is found.

5. **Final Answer**:

- After running the algorithm, we return the minimum time to reach each district. If a district is unreachable before the road disappears, we return `-1`.

Example Walkthrough

1. **Initialization**:
   - We start with **District 0** at time `0` in the priority queue.
   - The `dist` array (minimum time to reach each district) is initialized as `[0, inf, inf]`, where `inf` represents that the district is initially unreachable, and `0` is the time to reach District 0.
   - The `disappear` times are `[1, 3, 5]`.
2. **First Step - Process District 0 (Starting Point)**:
   - We pop District 0 from the priority queue with a time of `0`.
   - We explore its neighbors:
     - **To reach District 1**: The travel time is 2 minutes, so the new time is `0 + 2 = 2`. Since District 1 disappears at time `3`, we can reach it before it disappears. We update the time for District 1 to 2.
     - **To reach District 2**: The travel time is 4 minutes, so the new time is `0 + 4 = 4`. Since District 2 disappears at time `5`, we can reach it before it disappears. We update the time for District 2 to 4.
3. After this step, the `dist` array is `[0, 2, 4]`, meaning:
   - District 0 (starting district) has a time of `0`.
   - District 1 can be reached in 2 minutes.
   - District 2 can be reached in 4 minutes.
4. **Second Step - Process District 1**:
   - We now pop District 1 from the priority queue with a time of `2`.

    - We explore its neighbors:
        - **To reach District 0**: The travel time is 2 minutes, so the new time would be `2 + 2 = 4`, which is greater than the current time to reach District 0 (which is 0). Therefore, no update is made for District 0.
        - **To reach District 2**: The travel time is 1 minute, so the new time is `2 + 1 = 3`. Since District 2 disappears at time `5`, we can reach District 2 before it disappears. We update the time for District 2 to `3`.

5. After this step, the `dist` array is updated to `[0, 2, 3]`.
6. **Final Step - Process District 2**:
    - We now pop District 2 from the priority queue with a time of `3`.
    - We explore its neighbors:
        - **To reach District 0**: The travel time is 4 minutes, so the new time would be `3 + 4 = 7`, which is greater than the current time to reach District 0 (which is 0). Therefore, no update is made for District 0.
        - **To reach District 1**: The travel time is 1 minute, so the new time would be `3 + 1 = 4`, which is greater than the current time to reach District 1 (which is 2). Therefore, no update is made for District 1.
7. After this step, the `dist` array remains `[0, 2, 3]`.

Final Answer:

- **District 0 (Starting point)**: We are already at District 0, so the time is `0`.
- **District 1**: We can reach District 1 in `2` minutes before it disappears at time `3`, so the answer is `2`.
- **District 2**: We can reach District 2 in `3` minutes before it disappears at time `5`, so the answer is `3`.

Thus, the minimum times to reach each district are `[0, 2, 3]`.

Problem 4

You are exploring an archipelago of **5 islands**, each connected by a network of **one-way bridges**. These bridges let you travel from one island to another — but **only in the forward direction**. Once you leave an island, **you cannot come back**.

You start on **Island 1**. From there, at every step, you choose **uniformly and randomly** one of the islands directly reachable via a bridge and move to it. You repeat this process until you land on an island with **no outgoing bridges** — in which case you're **stuck there forever**.

You are given the number of islands, the list of bridges (each from island `X` to island `Y`), and the island you start on. Your task is to **simulate the process** and determine:

> Which island(s) you are **most likely** to end up stuck on.

If there are multiple islands with the **same maximum probability**, return them in **increasing order**.

Example Input:

```
Number of islands: 5
Number of one-way bridges: 7
Start island: 1

Bridges:
1 → 2
1 → 3
1 → 4
1 → 5
2 → 4
2 → 5
3 → 4
```

Question:

- Describe the algorithm used to find the island(s) where you are most likely to get stuck.
- Simulate the process step by step using the example and show how probabilities are distributed.

<u>Answer</u>

The **problem** is that we need to simulate the movement across islands where the probability of ending up at each island depends on the probability of reaching the **previous islands** that can move into it.

However, some islands are **dependent** on others — for example, you can't calculate the probability of being at island 4 until you know the probabilities from island 1, 2, and 3 that contribute to it.

So the challenge is: if we update islands in the wrong order, we might calculate a probability **before** knowing all the possible sources that flow into it — leading to **incomplete or incorrect values**.

So we need to use Topological Traversal because:

- It processes each island **only after all its incoming sources** have already been processed.
- It guarantees that for any directed edge `u → v`, we process `u` **before** `v`.
- Since the graph is a **DAG**, topological ordering is always possible and reliable for forward-only propagation.
- This solves the **dependency issue**, ensuring that each probability value is fully updated before it's passed on.

Once we've computed probabilities in topological order, we simply check the **sink islands** (with no outgoing bridges) and return the one(s) with the **highest final probability**.

How Do We Calculate Probability?

- We start by assigning `prob[start] = 1.0`, meaning you are 100% sure to start from the given island.
- For every island `u`, if it has `k` outgoing bridges (neighbors), then it distributes its current probability equally: `prob[u] / k` to each neighbor.
- We repeat this for every island, in topological order.
- The final probabilities at the **sink islands** (those with no outgoing bridges) represent the **likelihood of being stuck there**.
- We return the island(s) with the **highest probability** among the sinks.

Simulate the process step by step

Step 1: Topological Traversal

We apply **Kahn's Algorithm** to generate a valid topological order:

1. Calculate **in-degree** (number of incoming edges) for each island:

   ```
   in-degree[1] = 0
   in-degree[2] = 1 (from 1)
   in-degree[3] = 1 (from 1)
   in-degree[4] = 3 (from 1, 2, 3)
   in-degree[5] = 2 (from 1, 2)
   ```

2. Start with nodes having in-degree 0 → queue = `[1]`
3. Process node 1:
   - Add to topo order: `[1]`
   - Decrease in-degrees:
     ```
     in-degree[2] = 0 → enqueue
     in-degree[3] = 0 → enqueue
     in-degree[4] = 2
     in-degree[5] = 1
     ```
4. Process node 2:
   - Topo order: `[1, 2]`
   - Update in-degrees:
     ```
     in-degree[4] = 1
     in-degree[5] = 0 → enqueue
     ```
5. Process node 3:
   - Topo order: `[1, 2, 3]`
   - Update in-degree:
     ```
     in-degree[4] = 0 → enqueue
     ```
6. Process node 5 → `[1, 2, 3, 5]`

7. Process node 4 → `[1, 2, 3, 5, 4]`

Final topological order: `[1, 2, 3, 5, 4]`

Step 2: Probability Propagation (based on topo order)

1. **Initialize probabilities**:

   ```
   prob[1] = 1.0  # starting point
   prob[2..5] = 0.0
   ```

2. **Process island 1**
   Has 4 outgoing edges → each neighbor gets `1.0 / 4 = 0.25`

   ```
   prob[2] = 0.25
   prob[3] = 0.25
   prob[4] = 0.25
   prob[5] = 0.25
   ```

3. **Process island 2**
   Has 2 outgoing edges → each gets `0.25 / 2 = 0.125`

   ```
   prob[4] += 0.125 → prob[4] = 0.375
   prob[5] += 0.125 → prob[5] = 0.375
   ```

4. **Process island 3**
   Has 1 outgoing edge → all `0.25` goes to 4

   ```
   prob[4] += 0.25 → prob[4] = 0.625
   ```

5. **Process islands 5 and 4**
   Both have no outgoing edges (sink nodes) → we do not propagate further.

Final Answer: `4`

Island 4 has the **highest probability** of being the final destination, so it is the island you're most likely to get stuck on.

Problem 5

You are leading a rescue operation in a **smart building** where each room is equipped with **automated fire suppression systems**. The building is laid out as an `n × m` grid, and you're given a 2D list `moveTime`, where `moveTime[i][j]` represents the **earliest time (in seconds)** at which the room `(i, j)` becomes safe to enter after the fire suppression is activated.

You start at the **emergency control room** located at the **top-left corner (0, 0)** at **time 0**. You can move to **adjacent rooms** (up, down, left, or right), and **each move takes exactly 1 second**. However, you **cannot enter a room before its safety system finishes** (i.e., current time must be ≥ `moveTime[i][j]`).

Your mission is to reach the **trapped survivors** at the **bottom-right room (n-1, m-1)** in the **minimum time possible**.

Example Inputs:

Example 1:

```
moveTime = [
  [0, 4],
  [4, 4]
]
```

Example 2:

```
moveTime = [
  [0, 0, 0],
  [0, 0, 0]
]
```

Questions:

1. Describe the algorithm used to find the minimum time to reach the survivors in the bottom-right room.
2. Simulate the path step-by-step for each example and show how time is updated at each move.

<u>Answer</u>

Question 1: Describe the algorithm used to find the minimum time to reach the survivors in the bottom-right room.

You are in a smart building where **each room has a safety lock**, and you are **not allowed to enter** until the room is **ready** (`moveTime[i][j]` seconds after the start).
You start from `(0, 0)` and want to reach the bottom-right room `(n-1, m-1)` as **quickly as possible**, moving one cell at a time (up, down, left, right) — each move takes exactly **1 second**.

But there's a twist: *Even if a room is next to you, you **must wait** if it's not yet safe to enter.*

Why use Dijkstra's Algorithm?

This is a **shortest-path problem with dynamic movement costs**:

- In a normal grid, you'd use **BFS** (uniform cost).
- But here, **some rooms make you wait longer than others**, so the cost to enter depends on the **current time** and the room's **unlock time**.

That makes it a **weighted shortest path problem**, where:

- Nodes = rooms
- Edge weight = `max(current time, room unlock time) + 1`
- Goal = minimize time to reach `(n-1, m-1)`

Dijkstra's Algorithm is the best tool when:

- You want the **fastest route**.
- Travel cost depends on **where you are now**, and **where you go next**.

How the algorithm works:

1. **Use a priority queue (min-heap)** to always explore the room that can be reached the **earliest**.
2. Keep a `minTime[i][j]` matrix to record the **earliest time you can reach each room**.
3. Start from `(0, 0)` at time `0` and push it into the heap.
4. While the heap is not empty:
    - Pop the room with the **lowest current time**.
    - Try to move to all 4 adjacent rooms:
        - You can only move into `(ni, nj)` **at or after** `moveTime[ni][nj]`
        - Wait if needed: `nextTime = max(currTime, moveTime[ni][nj]) + 1`
    - If `nextTime` is better than the recorded time, update it and push it to the heap.
5. When you reach `(n-1, m-1)`, return the current time.

Question 2: Step-by-step simulation

Example 1

```
moveTime = [
  [0, 4],
  [4, 4]
]
```

Start at room (0, 0) at **time 0**.

Step 1:

Try moving to adjacent rooms:

- Room (0, 1): it is locked until time **4**, so you must wait. You can enter it at `max(0, 4) + 1 = 5`.
- Room (1, 0): also locked until time **4**, so you must wait. You can enter it at `max(0, 4) + 1 = 5`.

Push both of these into the heap:

`(5, 0, 1), (5, 1, 0)`

Step 2:

Pop the room with the lowest time → (5, 0, 1)

From here, try moving to adjacent rooms:

- Room (1, 1): it is available at time 4, and you're already at time 5, so you can enter immediately after 1 second: `5 + 1 = 6`.

Push into the heap:

`(6, 1, 1)`

Step 3:

Pop the next room → (5, 1, 0)

From here, try moving to room (1, 1) again. But we already plan to reach room (1, 1) at time 6, so we **skip it** (no improvement).

Step 4:

Pop room (6, 1, 1) → this is the destination.
You have reached the bottom-right room at **time 6**.

Output: **6**

Example 2

```
moveTime = [

  [0, 0, 0],

  [0, 0, 0]

]
```

Start at room (0, 0) at **time 0**.

Step 1:

Try moving to adjacent rooms:

- Room (1, 0): it is already available (time 0), so you can enter at `max(0, 0) + 1 = 1`
- Room (0, 1): also available, enter at `max(0, 0) + 1 = 1`

Push both:

```
(1, 1, 0), (1, 0, 1)
```

Step 2:

Pop room (1, 0, 1)

From here, move to:

- Room (0, 2): available, reach at `1 + 1 = 2`

Push:

```
(2, 0, 2)
```

Step 3:

Pop room (1, 1, 0)

From here, move to:

- Room (1, 1): available, reach at `1 + 1 = 2`

Push:

```
(2, 1, 1)
```

Step 4:

Pop room (2, 0, 2)

From here, move to:

- Room (1, 2): available, reach at `2 + 1 = 3`

Push:

```
(3, 1, 2)
```

Step 5:

Pop room (2, 1, 1) — from here, we could also reach (1, 2), but it's already scheduled at time 3, so we skip.

Step 6:

Pop room (3, 1, 2) → this is the destination.
You have reached the bottom-right room at **time 3**.

Output: 3