Assignment Solutions : Topological Sort

Q1 There is a group of n people labeled from 0 to n - 1 where each person has a different amount of money and a different level of quietness.
You are given an array richer where richer[i] = [ai, bi] indicates that ai has more money than bi and an integer array quiet where quiet[i] is the quietness of the ith person. All the given data in richer are logically correct (i.e., the data will not lead you to a situation where x is richer than y and y is richer than x at the same time).
Return an integer array answer where answer[x] = y if y is the least quiet person (that is, the person y with the smallest value of quiet[y]) among all people who definitely have equal to or more money than the person x.

Code link: https://pastebin.com/eNkpFbhS

Explanation:

- **Create an adjacency list adj to represent the richer relationships between people based on the given richer array. Each element adj[i] contains a list of people who are richer than person i.**
- **Create an indegree array of size n to store the indegree (number of incoming edges) of each person. Initialize all elements of indegree to 0.**
- **Iterate through the richer array and populate the adjacency list adj and the indegree array indegree. For each richer relationship [ai, bi], add bi to the adjacency list of ai and increment the indegree of bi by 1.**
- **Create a queue q to perform a topological sorting of the people based on their indegrees.**
- **Create an answer array of size n to store the answer for each person. Initialize all elements of answer to the person index itself, indicating that each person is the least quiet person among themselves.**
- **Iterate through all people from 0 to n-1 and enqueue the people with an indegree of 0 into the queue q.**
- **Perform a breadth-first search (BFS) on the graph using the topological sorting order:**
    - **Remove a person node from the front of the queue q.**
    - **Iterate through each person adjNode in the adjacency list of node:**

- Compare the quietness levels of answer[adjNode] and answer[node]. If the quietness level of answer[adjNode] is greater than the quietness level of answer[node], update answer[adjNode] with answer[node].
- Decrement the indegree of adjNode by 1.
- If the indegree of adjNode becomes 0, enqueue adjNode into the queue q.

- After the BFS, the answer array will contain the least quiet person for each person who has equal to or more money.
- Return the answer array.

The provided code uses a topological sorting and BFS approach to determine the least quiet person among those who have equal to or more money. It starts by building the adjacency list and indegree array, then performs a BFS to update the answer for each person based on the quietness levels.

Output:

```
Least quiet people among those who have equal to or more money:
Person 0: 5
Person 1: 5
Person 2: 2
Person 3: 5
Person 4: 4
Person 5: 5
Person 6: 6
Person 7: 7


...Program finished with exit code 0
Press ENTER to exit console.
```

Q2 There is a directed graph of n nodes with each node labeled from 0 to n - 1. The graph is represented by a 0-indexed 2D integer array graph where graph[i] is an integer array of nodes adjacent to node i, meaning there is an edge from node i to each node in graph[i]. A node is a terminal node if there are no outgoing edges. A node is a safe node if every possible path starting from that node leads to a terminal node (or another safe node). Return an array containing all the safe nodes of the graph. The answer should be sorted in ascending order.

Code link: https://pastebin.com/zTsBjfjA

Explanation:
The provided code uses a reverse graph approach combined with topological sorting to find the safe nodes in a directed graph. Here's an explanation of the approach:

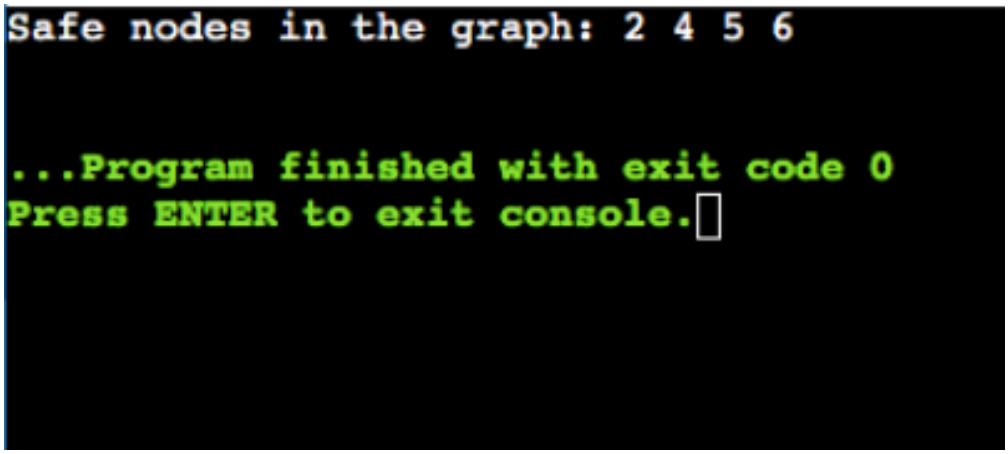
- **Create a reversed adjacency list adjrev to represent the reverse graph of the given adjacency list adj. Each element adjrev[i] contains a list of nodes that have an outgoing edge to node i.**
- **Create an indegree array indegree of size V (the number of nodes) to store the indegree (number of incoming edges) of each node. Initialize all elements of indegree to 0.**
- **Iterate through each node i from 0 to V-1:**
    - **For each neighbor it of node i in the original adjacency list adj, do the following:**
        - **Add node i to the reversed adjacency list adjrev as an incoming neighbor of it.**
        - **Increment the indegree of node i in the indegree array.**
- **Create a queue q and a vector safe to store the safe nodes.**
- **Enqueue all nodes with an indegree of 0 into the queue q.**
- **Perform a topological sorting using a modified breadth-first search (BFS) approach:**
    - **While the queue q is not empty, do the following:**
        - **Dequeue a node node from the front of the queue q.**
        - **Add node to the safe vector, as it is a safe node.**
        - **Iterate through each incoming neighbor it of node in the reversed adjacency list adjrev:**
            - **Decrement the indegree of neighbor it in the indegree array.**
            - **If the indegree of neighbor it becomes 0, enqueue it into the queue q.**
- **Sort the safe vector in ascending order to obtain the safe nodes in the graph.**
- **Return the safe vector containing all the safe nodes.**

Here's a breakdown of the code:

- **The eventualSafeNodes function takes the adjacency list adj as input and returns a vector safeNodes containing the safe nodes in the graph.**
- **The reverse adjacency list adjrev and the indegree array indegree are created based on the given adj list.**
- **The q queue and the safe vector are initialized.**
- **All nodes with an indegree of 0 are enqueued into the queue q.**
- **The modified BFS algorithm is performed:**
    - **Nodes are dequeued from the queue q, added to the safe vector, and their incoming neighbors' indegrees are decremented.**
    - **If an incoming neighbor's indegree becomes 0, it is enqueued into the queue q.**
- **The safe vector is sorted in ascending order.**
- **The safeNodes vector is returned, containing all the safe nodes.**

Note: The code assumes that V represents the number of nodes in the graph.
The provided code efficiently finds the safe nodes in the graph by using the reverse graph and topological sorting.

Output:



Q3 You are in a city that consists of n intersections numbered from 0 to n - 1 with bi-directional roads between some intersections. The inputs are generated such that you can reach any intersection from any other intersection and that there is at most one road between any two intersections.
You are given an integer n and a 2D integer array roads where roads[i] = [ui, vi, timei] means that there is a road between intersections ui and vi that takes timei minutes to travel. You want to know in how many ways you can travel from intersection 0 to intersection n - 1 in the shortest amount of time.
Return the number of ways you can arrive at your destination in the shortest amount of time. Since the answer may be large, return it modulo 109 + 7.

Code link: https://pastebin.com/A4Ei19jK

Explanation:

- **The code defines a custom struct pll (pair of long long) to represent the graph edges with the destination node and the travel time.**
- **The function dijkstra takes the graph, the number of nodes n, and the source node src as input and returns the number of ways to reach the destination in the shortest time.**
- **The dijkstra function initializes the dist and ways arrays with appropriate values. It uses a minHeap priority queue to store the nodes with their respective distances.**
- **The priority queue is initialized with the source node and distance (0, 0). The distance 0 represents the initial distance to the source node.**
- **The while loop continues until the minHeap is empty. In each iteration, it pops the node with the smallest distance from the priority queue.**
- **It checks if the popped distance is greater than the stored distance for that node (d > dist[u]). If it is, the node has already been visited with a shorter distance, so it skips the current iteration.**
- **For each neighbor v of the current node u, it calculates the new distance d + time and checks if it is smaller than the current distance stored in dist[v]. If it is smaller, it updates dist[v] and resets ways[v] to the number of ways to reach u since a new shortest path is found.**

- If the new distance d + time is equal to the current distance stored in dist[v], it increments ways[v] by the number of ways to reach u. This counts the number of ways to reach v in the shortest time.
- Finally, the countPaths function initializes the graph using the input roads and calls the dijkstra function to get the number of ways to reach the destination in the shortest time.

The updated code improves efficiency by reducing unnecessary updates to the dist array, resulting in faster execution.

Output:

```
Number of ways to reach the destination in the shortest time: 1


...Program finished with exit code 0
Press ENTER to exit console.
```

Q4 There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course ai first if you want to take course bi.

- For example, the pair [0, 1] indicates that you have to take course 0 before you can take course 1.

Prerequisites can also be indirect. If course a is a prerequisite of course b, and course b is a prerequisite of course c, then course a is a prerequisite of course c.

You are also given an array queries where queries[j] = [uj, vj]. For the jth query, you should answer whether course uj is a prerequisite of course vj or not.

Return a boolean array answer, where answer[j] is the answer to the jth query.

Code link: https://pastebin.com/G9vHwH6T

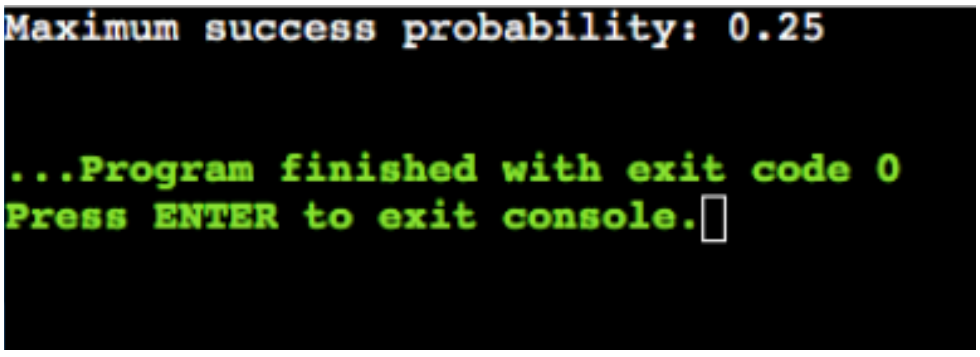
Explanation:
The approach used in the code is based on Dijkstra's algorithm with some modifications to find the path with the maximum success probability from the start node to the end node in an undirected weighted graph.

- Creating the Adjacency List:
  - The first step is to create an adjacency list representation of the graph. This is done by iterating through the edges and success probabilities and adding the target node and its success probability to the adjacency list of the source node, and vice versa.

- **Initializing Data Structures:**
  - **Initialize a probability vector prob of size n (number of nodes) with all elements set to 0. This vector will store the maximum success probabilities for each node.**
  - **Create a priority queue pq to store nodes based on their probabilities in descending order.**
- **Dijkstra's Algorithm:**
  - **Push the start node with a probability of 1.0 to the priority queue and set its probability in the prob vector to 1.**
  - **While the priority queue is not empty:**
    - **Extract the node with the highest probability (cur_prob) from the priority queue.**
    - **If the current node is the end node, return its probability (cur_prob) as the maximum success probability.**
    - **Iterate through the adjacent nodes (next_prob, j) of the current node:**
      - **Calculate the new probability (new_prob) by multiplying the current probability (cur_prob) with the success probability of the edge (next_prob).**
      - **If the new probability is higher than the existing probability of the adjacent node (prob[j]), update the probability of the adjacent node with the new probability (prob[j] = new_prob).**
      - **Push the adjacent node with its new probability (new_prob) to the priority queue for further exploration.**
- **Return the Result:**
  - **If the end node is not reachable, the function returns 0.0 as the maximum success probability.**

The algorithm iterates through the graph using Dijkstra's algorithm, updating the probabilities for each node as it explores the graph. The priority queue ensures that the nodes with higher probabilities are explored first, allowing the algorithm to find the path with the maximum success probability from the start node to the end node.
Overall, this approach efficiently finds the path with the maximum success probability by utilizing Dijkstra's algorithm and a priority queue to prioritize nodes with higher probabilities.

Output:



Q5 You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time it takes for a signal to travel from source to target.
We will send a signal from a given node k. Return the minimum time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1.

Code link: https://pastebin.com/AeAw5F06

Explanation:

- **Creating the Adjacency List:**
  - **The code initializes an adjacency list, adj, to represent the directed graph. Each entry adj[i] represents the neighbors of node i.**
  - **It iterates through the given times list and populates the adjacency list with the target node and the time taken to reach it from the source node.**

- **Initializing data structures:**
  - **The code initializes a distance array, dist, with a large value (INF) for all nodes except the source node. The distance array will store the minimum time to reach each node from the source node.**
  - **It initializes a visited array, vis, to track visited nodes.**
  - **It initializes a set, st, to store nodes based on their minimum distance in ascending order. Each entry in the set is a pair of distance and node.**

- **Applying Dijkstra's algorithm:**
  - **The code inserts the source node with a distance of 0 into the set, st.**
  - **While the set is not empty, the code performs the following steps:**
    - **Extracts the node with the minimum distance from the set.**
    - **If the node is already visited, continue to the next iteration.**
    - **Marks the node as visited.**
    - **Iterates through the neighbors of the current node.**
    - **If the new distance (current distance + edge weight) is smaller than the current distance stored in dist, it updates the distance and inserts the node into the set.**

- **Calculating the result:**
  - **The code checks if there are any nodes with a distance of INF, indicating that they are not reachable from the source node. If such a node exists, it returns -1.**
  - **Otherwise, it calculates the maximum time in the dist array, which represents the minimum time for all nodes to receive the signal.**

The time complexity of the code is O(V^2 log V), where V is the number of nodes. This is because for each node, the code potentially performs V-1 iterations in the worst case, and each iteration involves inserting or extracting a node from the set, which takes O(log V) time.
Overall, the code correctly calculates the minimum time for all nodes to receive the signal using Dijkstra's algorithm.

Output:

```
Minimum time for all nodes to receive the signal: 2


...Program finished with exit code 0
Press ENTER to exit console.
```