9. We claim that in a breadth-first search (BFS) from node `s` to node `t` in a graph `G`, if `t` is found in the `d`-th layer of BFS (the shortest path from `s` to `t` has the length of `d`), there must be at least one "unbreakable" layer between the 1st and (d-1)-th layers that contains exactly one node, and removing this node would disconnect `s` from `t`. We will proof this by way of contradiction:

1. Assume that every layer from the 1st to the (d-1)-th contains at least two nodes.
2. If every such layer has at least two nodes, then the total number of nodes from the 1st layer to the (d-1)-th layer would be at least `2(d - 1)`.
3. We would have at least `2(d - 1) + 2` nodes including s and t
4. From the problem set, we know that `d` > `n/2`, multiplying `d` by 2 gives us a value greater than `n`, which means `2d > n`.
5. The assumption that every layer between 1st and (d-1)-th has at least two nodes leads to the conclusion that there would be more than `n` nodes in the graph (`2d > n`), which is impossible since the graph only has `n` nodes.

Therefore, we reach our contradiction.

BFS (Breadth-First Search) is used to identify a node v whose removal would disconnect s from t. The time complexity of the algorithm is O(m + n) :

- BFS traverses every node once with o(n) time.
- BFS goes through every edge once, which takes O(m) time.

11. We will use a directed graph and traverse it to solve the problem. The nodes are computers and edges are the possible transmission of the virus at specific times. The key to the algorithm is to check if there is a path from the infected computer `Ca` to another computer `Cb` with correct timestamp. Specifically, the steps are:

1. Create a graph with each node representing a computer and no edges.
2. Add directed edges between nodes where there is communication. Each edge is annotated with the time of communication.
3. To see if a virus could have spread from `Ca` to `Cb`, we do a Depth-First Search (DFS) starting from `Ca` if the time is greater than or equal to the time at which `Ca` was infected.
4. If `Cb` is reached during the traversal, then `Cb` could have been infected by `Ca`.

Proof by contradiction:

Suppose the algorithm concluded that `C_b` could be infected by `C_a` but there is no sequence of communications moving forward in time where the virus could spread from `C_a` to `C_b`. This would mean there is a path from `C_a` to `C_b` in the graph where at least one edge was traversed before `C_a` was infected. However, by construction of the graph, such a path cannot exist because we only traverse edges that represent communication after the source node has been infected. This contradiction proves that if the algorithm concludes `C_b` can be infected, then it is because there exists a valid sequence of communications.

Time Complexity:

1. Building the graph: If there are `m` triples, then the time complexity for building the graph is O(m) .
2. Graph traversal (DFS): In the worst case, we visit every vertex and edge once, which has a time complexity of O(n + m), where `n` is the number of nodes (computers) and `m` is the number of edges (communications).

Therefore, the total time complexity is o(m)+o(n+m) =o(m+n).

12.

The algorithm involves creating a flowchart of sorts to map out how different lifetimes are connected. For every individual, we mark two events – their arrival and their departure. We then draw a line from their arrival to their departure, since everyone is born and later dies.

If we come across a piece of information indicating that one individual's departure happened before another's arrival, we draw an arrow from the end of the former's lifespan to the beginning of the latter's lifespan. And if we find that two individuals were alive during the same period, we draw arrows in both directions to show that their lifetimes crossed paths.

Proof:

Assume that the algorithm has produced a set of birth and death dates for each person, but there exists a pair of facts that contradict each other. This would mean that there is a directed cycle in the graph that went undetected during the cycle detection phase, which is impossible because our cycle detection step is exhaustive. Thus, if a cycle existed, the algorithm would have reported an inconsistency, contradicting our initial assumption.

Time Complexity:

O(m+n), including constructing the graph(o(m)), performing DFS which takes o(m+n) and topological sorting which take o(n+m).

6. We will use hashmap to keep track of indices and find indices j when `arr[i] == arr[j] + 2`.

1. Initialize an array `jumps` to store the minimum number of jumps to each index, initializing each value to infinity except for `jumps[0]` which is `0` since we start from the first index.
2. Create a hashmap `value_to_indices` to map each `arr[i] + 2` value to a list of indices where this value occurs.
3. Loop over each index `i` from `0` to `N-1`. For each index:
    - If `i > 0`, update `jumps[i]` to be the minimum of its current value or `jumps[i-1] + 1`.
    - If `i < N-1`, update `jumps[i+1]` to be the minimum of its current value or `jumps[i] + 1`.
    - Use the hashmap to find all indices `j` where `arr[i] == arr[j] + 2` and update `jumps[j]` to be the minimum of its current value or `jumps[i] + 1`.
4. Our final answer is the value in `jumps[N-1]`, the minimum number of jumps to reach the end.

Proof by contradiction:

Assume the algorithm does not provide the minimum number of jumps to reach the last index, which means there is a path with fewer jumps that the algorithm did not find. However, our algorithm explores all possible paths to each index using one step forward, one step backward, and jumps to indices with `arr[i] == arr[j] + 2`. Since we update the number of jumps at each step to the minimum found so far, there cannot be a shorter path that is not explored by the algorithm. Therefore, by contradiction, the algorithm must find the minimum number of jumps.

Runtime complexity:

Let `N` be the size of the array.

1. Initializing the `jumps` array takes `O(N)`.
2. Creating the hashmap by iterating through the array takes `O(N)`.
3. The main loop also takes `O(N)` since we use the hashmap to find and update jumps, which takes `O(1)` for each lookup and update.
4. Therefore, the overall time complexity is `O(N)`.