CS180 HW 2.

1.

3. The algorithm described in Section 3.6 for computing a topological ordering of a DAG repeatedly finds a node with no incoming edges and deletes it. This will eventually produce a topological ordering, provided that the input graph really is a DAG.

But suppose that we're given an arbitrary graph that may or may not be a DAG. Extend the topological ordering algorithm so that, given an input directed graph G, it outputs one of two things: (a) a topological ordering, thus establishing that G is a DAG; or (b) a cycle in G, thus establishing that G is not a DAG. The running time of your algorithm should be $O(m+n)$ for a directed graph with n nodes and m edges.

① First we create a set for visited nodes. and another for topological order.

② Starting from a node (n) with no edges pointing to it and add it to topological ordering list.

③. If this node has been visited, it means we have a cycle, and we can output the cycle with nodes from the start of the topological order list till when node n occurs.

④. Elese, if the node has not been visited, we will delete node n and its edges.

⑤. We will continue our next iteration

by finding the next node without incoming edges and repeat the same process above (②③④) again.

⑥. After the iterations, we will output the topo order list as case (a) and we have a DAG. (visited all notes).

⑦. We would've found the cycle before visiting all nodes (before step ⑥). So that's case (b).

Proof: The runtime is $O(m+n)$, since we check each edge m and each node n once during the algorithm. at most

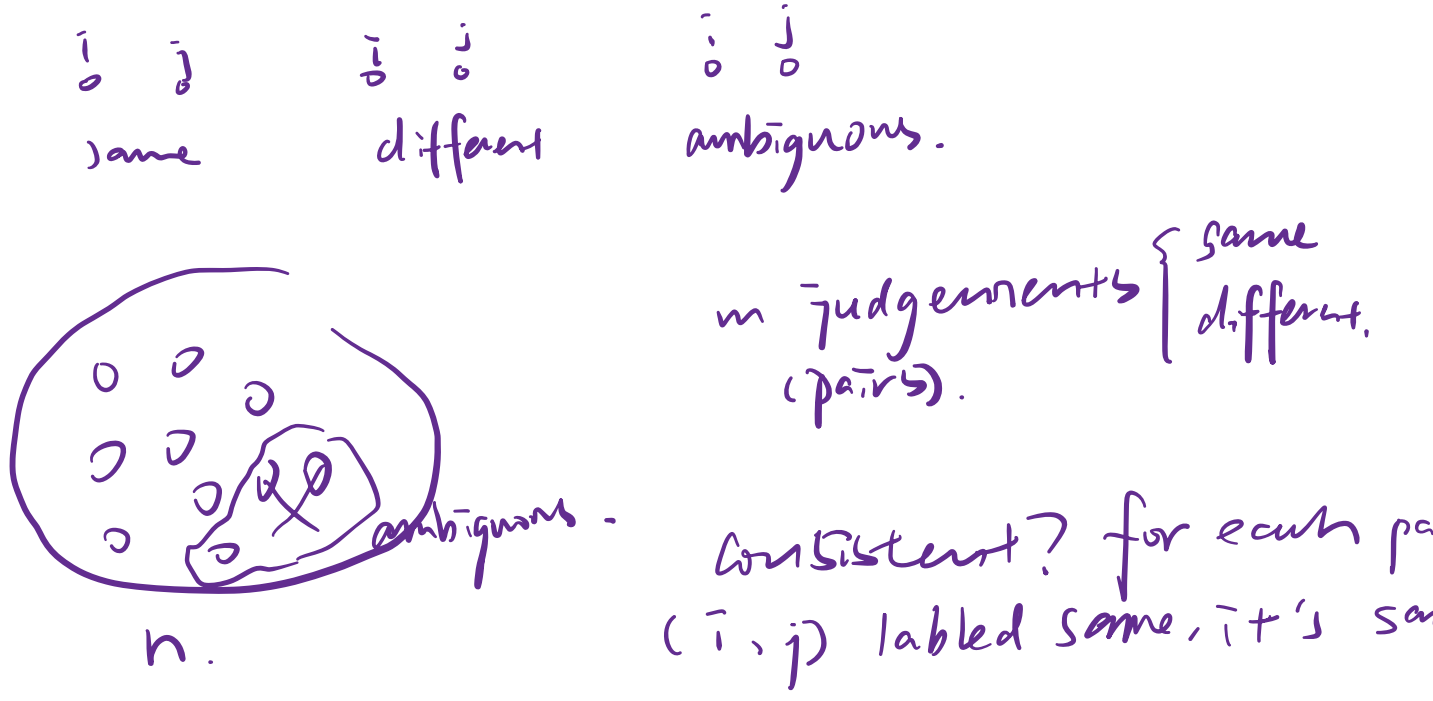
4. Inspired by the example of that great Cornellian, Vladimir Nabokov, some of your friends have become amateur lepidopterists (they study butterflies). Often when they return from a trip with specimens of butterflies, it is very difficult for them to tell how many distinct species they've caught—thanks to the fact that many species look very similar to one another.

One day they return with n butterflies, and they believe that each belongs to one of two different species, which we'll call A and B for purposes of this discussion. They'd like to divide the n specimens into two groups—those that belong to A and those that belong to B—but it's very hard for them to directly label any one specimen. So they decide to adopt the following approach.

For each pair of specimens i and j, they study them carefully side by side. If they're confident enough in their judgment, then they label the pair (i, j) either "same" (meaning they believe them both to come from the same species) or "different" (meaning they believe them to come from different species). They also have the option of rendering no judgment on a given pair, in which case we'll call the pair *ambiguous*.

So now they have the collection of n specimens, as well as a collection of m judgments (either "same" or "different") for the pairs that were not declared to be ambiguous. They'd like to know if this data is consistent with the idea that each butterfly is from one of species A or B. So more concretely, we'll declare the m judgments to be *consistent* if it is possible to label each specimen either A or B in such a way that for each pair (i, j) labeled "same," it is the case that i and j have the same label; and for each pair (i, j) labeled "different," it is the case that i and j have different labels. They're in the middle of tediously working out whether their judgments are consistent, when one of them realizes that you probably have an algorithm that would answer this question right away.

Give an algorithm with running time $O(m + n)$ that determines whether the m judgments are consistent.



To solve the problem, we'll use Breadth-First Search (BFS) and use graph as the way to conceptualize the algorithm.

We represent butterfly specimen as nodes in the graph, and the judgment that two specimens are "same" as edges between two nodes. If two specimens are "different", there's no edge between them. ("ambiguous" judgment is not under our consideration)

① Creat a graph G with n nodes and m edges.

② Start from an arbitrary node, mark it as visited and assign it to group A.

③ We will continue doing BFS, go over each node and

assign each one to a group.

- If we visit a node connected to the current node by the "same" edge, it should be put to the same group as the current node.
- If it's a "different" edge, the node should be put to a different group that the current node.

④. If we get to a node that has been visited and the group assignment contradicts what we had before, we'll know if there is inconsistency.

Proof: If we put two connected specimens in "different" group (group B) when they are actually the same, This contradicts the assumption that all "same" connected specimens are in the same group.

Run-Time Complexity: It's $O(m+n)$ because $O(n)$ is the time it takes to visit each node (specimen) once. and $O(m)$ is the time it takes to visit every edge (judgment) once during the BFS.

9. There's a natural intuition that two nodes that are far apart in a communication network—separated by many hops—have a more tenuous connection than two nodes that are close together. There are a number of algorithmic results that are based to some extent on different ways of making this notion precise. Here's one that involves the susceptibility of paths to the deletion of nodes.

Suppose that an n-node undirected graph $G = (V, E)$ contains two nodes s and t such that the distance between s and t is strictly greater than $n/2$. Show that there must exist some node v, not equal to either s or t, such that deleting v from G destroys all s-t paths. (In other words, the graph obtained from G by deleting v contains no path from s to t.) Give an algorithm with running time $O(m + n)$ to find such a node v.

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. You're helping a group of ethnographers analyze some oral history data they've collected by interviewing members of a village to learn about the lives of people who've lived there over the past two hundred years.

From these interviews, they've learned about a set of n people (all of them now deceased), whom we'll denote $P_1, P_2, \ldots, P_n$. They've also collected facts about when these people lived relative to one another. Each fact has one of the following two forms:

- For some i and j, person P_i died before person P_j was born; or
- for some i and j, the life spans of P_i and P_j overlapped at least partially.

Naturally, they're not sure that all these facts are correct; memories are not so good, and a lot of this was passed down by word of mouth. So what they'd like you to determine is whether the data they've collected is at least internally consistent, in the sense that there could have existed a set of people for which all the facts they've learned simultaneously hold.

Give an efficient algorithm to do this: either it should produce proposed dates of birth and death for each of the n people so that all the facts hold true, or it should report (correctly) that no such dates can exist—that is, the facts collected by the ethnographers are not internally consistent.

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. Given an array arr of size N , the task is to find the minimum number of jumps to reach the last index of the array starting from index 0. In one jump, you can jump from current index i to index (i + 1) or index (i − 1); or you can move from index i to index j ($i \neq j$) if arr[i] is equal to arr[j] + 2.
Note: You can not jump outside of the array at any time.
Examples:
Input: arr = {100, −23, −23, 404, 98, 23, 23, 23, 3, 402}
Output: 3
Explanation: Valid jump indices are $0 \rightarrow 4 \rightarrow 3 \rightarrow 9$.

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)`.