Charles Zhang

Disc. 1D
Fri. 10am - 12pm
Song

1)

- Algorithm:
  - Construct a BFS tree starting from node *s* as the root
  - Set up an array of size *n* + 1, initializing each element to 0
  - Iterate through the BFS tree, calculating the level of each node by keeping an iterator to count how many edges have passed by
    - For each node, use this level as an index into the array and increment the accessed value by 1 (this keeps track of how many nodes there are per level)
  - Iterate through the array, checking for a value that is exactly 1
  - The index of this value, *i*, represents the level that the single node is on, so iterate through the BFS tree once more on an arbitrary path, passing exactly *i* edges
  - Return the node you land on
- Proof:
  - Construct a BFS tree from *s* to *t*
  - The first claim is that there is a level in this BFS tree that contains only 1 node
    - By the problem statement, we know there will be at least *n* / 2 levels between the root *s* and the level containing *t*
    - Let's say there are at least 2 nodes per level
    - If this were true, the levels between *s* and *t* would contain a total of 2(*n* / 2) = *n* nodes
    - This is impossible since *s* and *t* are not found in the levels between *s* and *t*, therefore there would have to be *n* + 2 nodes in the graph
    - By the problem statement, we know there are only *n* nodes in the graph, so we have reached a contradiction, there must be at least 1 level containing only 1 node
  - The second claim is that the existence of a level with only 1 node implies that a node *v* exists that can break any *s*-*t* path
    - We now know there exists some level that only contains node *v*, and that this level lies between the levels containing *s* and *t*
    - Let's say this node *v* is at some level *n*
    - Deleting node *v* means we cannot access any level from *n* onwards
    - By the definition of a BFS tree, the path to *t* must contain an edge leading through level *n*, otherwise *t* would have been in an earlier level of the BFS tree
    - This means that the path from *s* to *t* must contain an edge from level *n* - 1 to *n*

        - However, the only node in n is v, which is now deleted, so any possible path from s to t is now broken
- Time Complexity: $O(n + m)$
- Time Complexity Proof:
    - Construction of a BFS tree requires visiting each node and edge in the graph once and performing a constant time operation on each, resulting in an $O(n + m)$ runtime
    - Allocating and initializing the array requires $O(n)$ time
    - Iterating through the BFS tree and calculating each level also requires an $O(n + m)$ runtime for the same reasons listed above
    - Iterating through the array has a worst case runtime of $O(n)$, as that is the maximum number of levels we will have to check
    - Iterating through the BFS for the third time once again has a runtime of $O(n + m)$
    - Overall, this algorithm has a runtime of $O(5n + 3m)$, which can simply be reduced to $O(n + m)$

2)*

- Proof:
  - Assume that G is not the same graph as T
  - In order for this to be true, there must be some edge *e* = (*i*, *j*) in G that is not in T
  - There are 2 possibilities for this edge:
    - *e* would be located within a level of T
      - Let T be the result of the BFS and T' be the result of the DFS
      - Assuming this is the case, this means that *i* is an ancestor of *j*
      - By the definition of a DFS, if *i* was indeed an ancestor of *j*, *j* would be located on a lower level than *i*, since all ancestors in a DFS tree are guaranteed to be on a higher level than their child nodes
      - This implies T' is not equal to T, which is a contradiction of the problem statement
      - As a result, such an edge *e* could not be present in the graph G
    - *e* would skip over a level of T
      - Let T be the result of the DFS and T' be the result of the BFS
      - Assuming this is the case, this means that *i* and *j* differ by more than exactly 1 level in T
      - By the definition of BFS, this means that, in T', *i* and *j* are exactly 1 level apart or on the same level
        - This is because in a BFS, if there is an edge between 2 nodes, either one will discover the other or they will both be discovered in the same level
      - This implies that T is not equal to T', which is a contradiction of the problem statement
      - As a result, such an edge *e* could not be present in the graph G
  - Since both possibilities of edge *e* cannot exist, we have proven by contradiction that G = T if T is the BFS tree and DFS tree of G

3)

- Algorithm:
    - Iterate through each of the triples, creating an undirected edge between each computer pair C_i and C_j, assigning the edge a weight of t_k to create a graph G
    - Starting from the root C_i with time $t = 0$
    - While there are still nodes to search in G:
        - For the children of the current node:
            - If the weight of the edge leading to the child node is greater than or equal to the node's time value:
                - If the child node is explored:
                    - If the new edge's weight is less than the child node's time pairing:
                        - Replace the time pairing with the edge weight
                    - Else:
                        - Skip the node
                - Else:
                    - Push the node into the list of nodes to search, paired with the weight of the edge as its time value
                    - Create an edge between the current node and the child node
                    - Mark the node as explored
            - Else:
                - Skip the node
    - For all nodes in the BFS tree:
        - If the current node is C_j:
            - Return true
    - Return false
- Proof:
    - The construction of the graph is straightforward
        - We define our graph as a collection of computers (nodes) that are connected by an edge if they communicate at some time t
        - This graph can be undirected, since the virus can be passed as long as the computers communicate, it doesn't matter which communicates with which
        - The time t at which they connected is stored for later use
    - Now, we must prove that the algorithm's next step finds all computers that C_i communicates with, both directly and indirectly
        - Assume there is a computer C_x that communicates with C_i, but is ignored by the algorithm
        - There are 2 cases:
            - C_x was in direct communication with C_i:
                - For this to be true, C_i must be the parent of C_x
                - Since the algorithm initializes C_i's time as 0, all other time

values are greater than or equal to C_i's
        - This means that an edge leading from C_i to C_x would automatically be valid, and C_x would be analyzed by the algorithm
        - The case has been contradicted
    - C_x was in indirect communication with C_i:
        - The direct communication case was the base case
        - Assume the graph is correctly constructed up until C_x is analyzed
        - By definition, the edge leading to C_x must have a greater weight than its parent node's time value, otherwise C_x would not have received the virus based on our problem setup
        - If this were true, the algorithm would push C_x onto the list of nodes to be analyzed
        - This case has been proven incorrect by induction
    - Both cases have been contradicted, meaning it is impossible for there to be a computer C_x that should've been accessed but wasn't
- Time Complexity: O(m + n)
- Time Complexity Proof:
    - The creation of the graph G involves performing multiple constant time operations on each triple, which reduces down to an O(m) time since there are m triples
    - The construction of a BFS tree requires visiting each node and edge in the worst-case, resulting in an O(m + n) runtime
    - The search of the BFS tree requires a worst-case time of O(m + n) to analyze each node
    - The overall runtime can then be reduced to O(m + n)

4b)*

- Counterexample:
  - This statement is false
  - Assume a simple case:
    - All weights are positive, the graph is undirected, and the graph is connected
    - The path *P* leads from *s* to *t* and has weight 4
    - An alternate path exists that goes through some node *x* with edge weights of 3 and 3
    - Assuming these are the only nodes and edges in the graph, by definition, *P* is a minimum-cost path, since *P* has a weight of 4 and the path through *x* is of total weight 5
    - However, when squared the path *P* takes on a weight of 16, while the path through *x* takes on a weight of 13
    - *P* is clearly no longer the minimum length path, as 16 > 13
    - The statement must be false since it clearly doesn't apply to all possible cases

5)*

- Algorithm:
    - Initialize a counter variable to 0 and an element variable
    - For all elements in the array:
        - If the current counter is 0:
            - Set the element variable to the current element
            - Set counter to 1
        - Else if the current element is equal to the element variable:
            - Increment the counter variable by 1
        - Else:
            - Decrement the counter variable by 1
    - Set the counter variable to 0
    - For all elements in the array:
        - If the current element is equal to the element variable:
            - Increment counter by 1
    - Return if the counter is strictly greater than n / 2
- Proof:
    - The second for loop is self-explanatory
        - We're simply checking to see if the result of the first for loop is a majority through brute force iteration
    - Therefore, we must simply prove that the result returned from the first for loop is the best possible option for a majority element
    - There are 3 cases that occur during the algorithm's runtime:
        - There's no majority vote:
            - There is no "best" element to be returned, since all of them are equally not the majority
            - The absence of a majority is caught by the second for loop, as the element returned from the first for loop is not a majority
            - The correct result is returned
        - The counter never returned to 0
            - In order for the counter to not return to 0, at any given point in the array traversal, it must have incremented more times than it has decremented
            - For this to be true, the first element encountered in the algorithm's runtime must be the majority element
            - Since the counter never returns to 0, the element variable never changes from the first element
            - The first element is therefore returned from the first for loop
            - The correct result is returned
        - The counter returned to 0 at some point
            - Each time the counter returns to 0, the algorithm essentially resets
            - This means that when the counter returns to 0, the majority element of the whole array must still be the majority element of the elements in the remaining subarray

        - If we can prove this to be true, we've inductively proved that the algorithm will return the correct value for this case
    - Assume there are *x* occurrences of the majority element in the array
    - By the definition of a majority element $x > n / 2$
    - Imagine we have searched *i* elements when the counter reaches 0
    - Assuming a lower bound case for the number of majority elements remaining, there are $x - i$ majority elements left in the subarray and $n / 2 - i$ total elements left in the subarray
    - Since $x > n / 2$, it is implied that $x - i > n / 2 - i$, which means the majority element is still the majority element in the subarray
    - In the base case where there is a single element left in the array (the majority element by definition), the algorithm increments counter to 1 and returns the correct value, so the base case is satisfied
    - The correct value is guaranteed to be returned
- Time Complexity: $O(v)$
- Time Complexity Proof:
    - The first pass iterates through all elements of the array once, performing constant time operations on each of them, resulting in an $O(v)$ complexity for the first for loop
    - The second pass iterates through all elements of the array once, performing a check and possible iteration, both of which are constant time, resulting in an $O(v)$ complexity for the second for loop
    - This reduces down to an $O(v)$ runtime for the entire algorithm

6a)*

- Algorithm:
  - If there are no edges:
    - Return 0
  - Arbitrarily order all edges in a list *x*
  - Initialize a size variable to *e*
  - While the size variable is greater than 0:
    - For all elements in *x*:
      - While new permutations beginning with *x* of size *e* still exist:
        - Recursively find an unused permutation of *x* beginning with the current element
        - If the permutation of *x* represents a valid path in the DG
          - Return the current size variable
        - Else:
          - Mark the current permutation as used
    - Decrement the size variable
  - Return 0
- Proof (informal since the previous problem explicitly asked for a proof and this problem doesn't ask for it):
  - The interior of the algorithm finds each possible permutation of the edges presented in the problem
  - The algorithm therefore checks every possible path starting from longest possible (*e*) to shortest (0)
  - Therefore, the first permutation that represents a valid path must be the longest length path in the entire graph
- Time Complexity: O(*e*!)
- Time Complexity Proof:
  - The algorithm generates every permutation of *e* edges, and checks if the path generated by each permutation is valid
  - The check of each path takes at most O(*e*) time
  - The check of every permutation takes O(*e*!) time
  - If the first check fails, the algorithm goes to check (*e* - 1)! elements by the definition of an r-permutation
    - This continues from *n*, *n* - 1, *n* - 2, …, 1
  - The *e*! term dominates the time complexity, so the algorithm is overall O(*e*!)

6b)

- Algorithm:
  - Perform a topological ordering of the given graph
  - Starting from the beginning of the topological ordering:
    - If the node has an indegree of 0:
      - Pair the node and out-edges with the value 0
    - Else:
      - Pair the node with the maximum value of its in-edges + 1

    - Initialize a max value variable to 0
    - For each node in the topological ordering:
        - If the node's paired value is greater than the max value:
            - Update the max value to the current node's value
    - Return the max value variable
- Proof (informal since the previous problem explicitly asked for a proof and this problem doesn't ask for it):
    - The guarantee that this graph is acyclic tells us that the longest path must begin at a source and end at a sink, as any other start/end would have an indegree/outdegree that could be used to extend the path
    - The use of topological sort allows us to generate all possible paths from source(s) to sink(s)
    - By virtue of topological sort's design, it is then possible to iterate through the sorted list of nodes and track local maximum path lengths
        - This is because in a topological sort, we don't have to worry about anything prior to the current node, we know for a fact everything until the current point is updated
    - By tracking lengths, we can then reach the sink(s) and save maximum values obtained from them, leading to the right output
- Time Complexity: $O(n + e)$
- Time Complexity Proof:
    - Topological sort takes $O(n + e)$ time
    - The first loop iterates through the graph, accessing each node once and most edges twice, resulting in an $O(n + e)$ runtime
    - The final loop runs through each node, resulting in an $O(n)$ runtime
    - The overall runtime of the algorithm can be reduced to $O(n + e)$