Hannah Kwon
[redacted]
Professor Hsieh

Midterm

Computer Science 180 - Introduction to Algorithms and Complexity

Question 1

(1) True
(2) True
(3) True
(4) False
(5) False

Question 2

```
isConnected(nonAdjList, n):
    choose random vertex v
    let seen = boolean array of size n (number of vertices) set to all false
    dfs(v, seen, nonAdjList)

    return true if seen array is all true, else false

dfs(v, seen, nonAdjList):
    if seen[v] is true:
        nothing
    else:
        seen[v] = true
        let s = 0
        for v1 in nonAdjList[v]:
            call dfs(v2, seen, nonAdjList) for each v2 in (s ... v1]
# (these are nodes that are adjacent to v, calculated using the non-adjacent nodes.
# Nodes between the previous node and current node in the non-adjacency list are
# adjacent nodes because the list is sorted)
            s = v1
```

Algorithm is O(m) because for each node in the non-adjacency list, we process the node once, calling dfs on its children. The graph is connected if every node is seen by processing connected nodes starting at the initial vertex. If a vertex is missing, then that means it is not connected to the initial node.

Question 3

```
V = number of total vertexes
T = [] #T stores the topological sort
Z = [] #Z stores vertices with indegree 0
indegree #stores current indegree of each vertex in dictionary

def topologicalSort :
  for each v in V : #initialize indegree
      for each u adjacent to v : #initialize in
          indegree[v]++

  for each v in V : #initialize Z
      if indegree[v] = 0 :
          Z.append(v)

  while Z is not empty do :
      v = Z.pop()
      T.append(v)
      for each u adjacent to v :
          indegree[u]--
          if indegree[u] = 0:
              Z.append(u)
  return T
```

```
path = []
visited = [false, false, false , ... ] (V amount of false elements in list)
start = 0
visited[start] = true
path.append(start)
def ifNodeVisitingAllNodes(path, v) :
    if path.size() == V :
        return True
    for each node of v : #that is directed from v
      if visited(node) == false :
          visited[node] = true
          path.append(node) #add node to path
          ifNodeVisitingAllNodes(path, node) :
              return True
          visited[node] = False   #backtrack
          path.pop(node)

    return False
```

This algorithm is a O(|E| + |V|) since the topological sorting takes O(|E| + |V|) as proven by in class and the second recursion function takes O(|V|) time since it iterates all the vertices. Thus, O(|E| + |V|) + O(|V|) = O(|E| + |V|).

This algorithm is right because after doing the topological sort, you know that edges go from lower vertices to higher which means there exists a path that can visit every node, if there are edges between consecutive vertices.

Question 4

```
partitionElements(arr):
    Step 1: x = query(arr) # this gives us the total number of unique elements
        # complexity: O(1) query calls

    Step 2:  #take sample 0
        # Overview: There are x unique integers.
        # We are creating x subsets, each containing a single unique integer index
        s = x initially empty subsets
        numNonEmptySubsets = 0 # represents number of subsets in s that are not empty
        remainder = []
        for each index "i" in the array "arr":
            q = query(i, indices in s)
            if q is 1 + numNonEmptySubsets:
                add i to its own subset in s
                numNonEmptySubsets += 1
            else:
                append i to remainder

        # this will populate s with x subsets, each containing a single index.
        #Each subset will contain distinct indices to distinct integers
        # complexity: O(n) query calls

    Step 3:
        for each index "i" in array "remainder":
            call helper(i, s)

        # Step 3 complexity: O(nlogn): for each remaining element in remainder,
        #we call helper() which is an O(logn) operation
        # Overall algorithm complexity: O(1 + n + nlogn) => O(nlogn)
```

>> continued

```
# Puts index i into its proper subset by cutting the total number of subsets in half
# until it finds the right subset
# Complexity: O(logn) because by cutting the size of subsets in half
# with each recursive call, there can be a total of logn calls to helper()
helper(i, subsets):
    # basecase: if there is only 1 subset, and query() tells us
    # the the integer at index i belongs to the subset, put it in
    if length of subsets == 1:
        if query(i, indices in the single subset) == 1:
            put i into the subset
        else: nothing
    else:
        divide subsets into two halves ("left" and "right"),
        each containing half of the remaining subsets
        l = query(i, left indices) #query returns the number of unique int
        r = query(i, right indices)

        # one of these query calls will equal the length of that subset,
        # meaning that the integer at index i belongs to a subset within that group
        if l == length(left):
            helper(i, left)
        else:
            helper(i, right)
```

The algorithm is built by creating subset lists of all the indices of the unique integers in the inputted array. Once we have all types of subsets, we iterate over all the elements of the inputted array and find out using recursion if that particular element belongs to properly. Total time complexity would be O(1 + n + nlogn) = O(nlogn).