COMP2123- Assignment 5

UID: 500557727

Problem 1.

a) The ModifiedPrim algorithm is correct since it returns null when there is no monotone spanning tree and the correct monotone spanning tree if there is. I will prove this by induction. The base case is graphs of size 1, which is a trivial example in which the algorithm will simply return the single node as it is the minimum of the graph and will have no other elements to recurse over.

For some graph of size n, we assume that the algorithm has correctly either returned null after finding that there can be no minimum spanning tree or has correctly created a minimum spanning tree of the k elements in G that it has recursed on. For the k+1'th element, the algorithm will choose this as the minimum vertex that we have visited but not yet iterated over its incident edges. If it finds that all the incident unvisited vertices have integer values greater than it, it will continue creating the minimum spanning tree, otherwise it will return null. These choices are correct because, for a minimum spanning tree to be possible, all paths from the root to the leaves must follow vertices that have integer values greater than the ones before them. The algorithm allows for that by iterating over the incident edges of the smallest vertex in H, which will have the smallest difference between its integer value and its unvisited incident vertices when compared to the other vertices in the graph. If the k+1'th element has a higher integer value than one of its incident vertices, then there is no other edge which would have a smaller value than that vertex, meaning that a monotone spanning tree is not possible. So if the algorithm returns a monotone spanning tree then it has found the correct spanning tree and if it has returned null then no monotone spanning tree is possible.

b) The ModifiedDFS algorithm is also correct as it always attempts to choose and attach the incident vertex that will maintain T as a monotone spanning tree. If the tree T created by the recursion is not the monotone spanning tree of the given graph G then the algorithm will return null.

The latter is simple to prove by contradiction if we assume that T is always a monotone spanning tree: if the algorithm has returned null but there is a monotone spanning tree for the graph, then the assumption must be incorrect and T must not have been a monotone spanning tree of graph G. This is true for the converse as well.

To prove that T is always a monotone spanning tree I will use that as the invariant. At first, T is the vertex s with the smallest integer value, which of course is a monotone spanning tree. At each recursion of the algorithm, the DFS attaches a vertex that is greater than the current. If a given vertex is not greater than the current, then that

vertex can only be visited from another connected vertex that is less than it or can never be visited at all. This ensures that all paths created by the monotone spanning tree are non-decreasing as each recursion from the root attaches a vertex with a higher integer value to each one of its children.

Problem 2.

a) This problem is quite similar to the task scheduling/interval partitioning problem, with the added requirement that we have to save the points at which the partitions should be placed. Like the slides, my algorithm will begin by merge-sorting the customers' ranges by starting range 'l' in ascending order. I initialise the towers' positions in a hashmap/dictionary 'towers_customers' where the key is the tower's position and the value is an array of all the customers that the tower is covering. Then I will iterate over the customers' intervals and, if a customer's range does not contain the towers' position then I have to check if moving that tower to the current customer's starting position will still cover this tower's original customers AND this one. If it can then I can just move it to that position. If it can't and no other tower can, then we have to add a new tower. Otherwise, the tower can cover that customer's needs and I just add that customer to the tower's value in the 'towers_customers' dictionary. At the end, I return the array 'towers' which is the smallest set of locations feasible.

```
def in_range(num, range_start, range_end):
    if num >=range_start and num<=range_end then
        return true
    return false

def place_towers():
    sort intervals in increasing starting time order using merge-sort
    towers ← ∅ #dict containing tower positions as the key and customer's ranges as the value
    for customer in customers_increasing do
        customer_assigned = False
        for t in towers do
            if in_range(t, customer[l], customer[r]) then
                add customer to towers[t]
                customer_assigned = True
```

```
                    break
            else
                    if for customer in t in_range(customers[l], cust_range[l], cust_range[r])) then
                            move this tower to customers[l]
                            add this customer to the tower's value in 'towers'
                            customer_assigned = True

        if not customer_assigned:
                add a new tower to 'towers' at position customers[l]

    return keys of 'towers' dictionary
```

b) The proof of correctness for my algorithm is similar to that of the slides: showing that any new towers that are placed means that any new customers are incompatible with the previous towers. When a new tower X is placed, that means that no other tower could have covered that customer AND the previous customers they've been covering i.e. the current customer's starting range is greater than the position of tower X - 1. So, there must be at least X towers at that number of customers.

Of course, when a tower is placed or updated its position is the starting position of its newest customer, meaning that it is guaranteed to be at the correct position to cover its customers. The algorithm then returns these positions.

c) My algorithm starts with merge-sort and as shown in the slides, this takes O(nlogn) time. Next, I make n iterations over the n customers. I then check if the customer can be assigned to any of the towers in 'towers'. In the worst case of each iteration, each tower in 'towers' will not have been in range of the customer and will have tried to change to that customer's starting position to see if it's going to be in range of all the customers now. It will have iterated over all of the towers and all their customers and then created a new tower.

I will begin by analysing the iterations of the loops and then the time complexities within the loops themselves. The maximum towers and customers at customer k are inverse to each other; there can only ever be a maximum of k-1 towers in the iteration of the kth customer, in which case there will only be one customer in each tower. And if there is the maximum of k-1 customers in one tower in the iteration of the kth customer, then there is only one tower to iterate over. This is because there are k-1 customers that have been assigned when assigning the kth customer. If the customer can't be assigned to a tower straight away, then there will be either k-1 loops over the towers and 1 loop over the customer of that tower, or k -2 loops over the towers with one tower having 2 customers and the rest with 1 etc. This means that in each loop over the towers, the algorithm will do O(k-1) for $k < n$.

After sorting the customers in increasing order of starting time, the algorithm initialises an empty dictionary in O(1). Then it iterates over each customer in that dictionary, beginning by initialising a boolean to false in O(1) then iterating over the worst-case k -1 towers where k is the number of customers visited. It makes a call to the in_range function which simply makes an O(1) check then returns a Boolean in O(1), meaning that the function is O(1). In the worst case, this Boolean will be false and the algorithm will iterate over the customers in the tower t, which is a single customer in the worst-case of k-1 towers. In this iteration, it will perform the O(1) function check and then make more O(1) assignments within the 'towers' dictionary. If no tower can be moved to accommodate this customer, then the algorithm will perform another O(1) check and another O(1) addition to 'towers'. Each loop over the customers is therefore $O(k - 1) + O(1) = O(1)$.

So, since each loop in the customers' iteration takes O(1), this amounts to O(n) so the entire algorithm is $O(n\log n) + O(n) = O(n\log n)$.

Problem 3.

a) My algorithm attempts to find the envy-free location/index in O(logn) time by performing a modification of the binary-search algorithm. Like the binary-search algorithm, it begins by examining the middle element of the arrays and then reducing the options in half each recursion by choosing the right or left half of the array for the next array.

By using the fact that Alice and Bob's arrays are in increasing order and decreasing order respectively, I can check the middle of both arrays at each recursion for equality. If they are equal, then I can simply return the 'mid_index' value I've been using in the recursive function as the cake position. If they are not equal, then I must recurse over the right or left half of the arrays. I make this choice by recursing over the right half if A[mid_index] < B[mid_index], and over the left half if A[mid_index] > B[mid_index]. At each recursion, I check the middle element of the array half and make this choice again until we either find an equal element and return it or reach the empty set and return null.

b) Similar to the slides' proof of the binary search algorithm, I will use an invariant to prove my algorithm's correctness. That invariant is that if an envy-free location exists in A and B before the recursions, then it will remain in A and B after the recursions. A and B are copies of the halves of the previous arrays.

If A[mid_index] is not equal to B[mid_index] then we need to look for an element that is between the range created by A[mid_index] and B[mid_index]. When A[mid_index] < B[mid_index], we can try to find this in the right halves of A and B since every element in the right half of A is greater than A[mid_index] and every element in the right half of B is less than B[mid_index]. We want to look for an element in A that is greater than A[mid_index] so we can get closer to B[mid_index], and we want to look for an element in B that is lower than B[mid_index] so we can get closer to the new element of A. The opposite is true for the case where we recurse over the left half.

c) I will use unrolling to analyse my time complexity:

At each recursion, my algorithm makes one recursive call on arrays half the size of the previous arrays, meaning that there will be a maximum of logn recursive calls. In each of these recursions, there are 4 if-statements: one O(1) check if the arrays are empty where it will return null, one O(1) check to see if A[mid_index] = B[mid_index] where it will return A[mid_index], one O(1) check to see if A[mid_index] < B[mid_index] where it will recurse on the right half, and one O(1) check to see if A[mid_index] > B[mid_index] where it will recurse on the left half. The worst case for each recursion is that it will bypass the first three if statements and fall into the last one, meaning 4O(1) checks and O(1) copying of A and B halves and recursing on them. Each recursion is then 5O(1) = O(1) so logn recursions will amount to O(logn).