Hannah Kwon
705182275
Professor Hsieh

# Homework #3

**Computer Science 180 - Introduction to Algorithms and Complexity**

## Question 1

We have a Minimum Spanning Tree (MST), of a graph G, and we are asked to find whether the MST will still be an MST, if we change the weight w, of an edge (u, v), to w'. We can solve this problem by splitting MST into two components and checking the edges that connect these two components. An MST is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together without any cycles. Because all nodes are connected to each other, if we remove (u,v), the MST will be split into two components. Let us call these separate components A and B, where u is part of A and v is part of B. Then, if we run BFS, on both of these nodes, we can see which component each node belongs to. Now, we can loop through all edges of the G, and get the edges that connect A and B. This is because only these edges will create a spanning tree by connecting itself with A and B. The edge with the lowest cost is the one that belongs to the new MST. If the lowest edge is still (u, v), then the new MST is the same as the old one and if not, then it isn't.

```
Algorithm :
Let weight(x, y) = the weight of edge (x, y)
Let M = MST of G with edge (u,v)
Remove (u, v) from M
# run BFS starting from u and v separately and store all nodes that are visited
Let A = the component of M such that includes u
Let B = the component of M such that includes v
Let all nodes in A
for each edge (s, t) in E :
    if label[s] != label[t] and weight(s,t) < weight(u, v) :
        return false //means that
return true
```

The algorithm is a total of O(m) time, because we traverse all the edges of a total of 3 times. The first two iterations are by running BFS to check the array that has each node's component. The last traversal comes from searching through the edges in G for all the edges that connect the two components.

## Question 2

Let shortest path p between s and t contain edges with edge weight $e_1, e_2, \cdots, e_n$.
And let's say there exists a path q between s and t that contain edges with edge weight $e_1', e_2', \cdots e_n'$.
Currently, $(e_1+e_2+e_3+\cdots+e_n) < (e_1'+e_2'+e_3'+\cdots e_n')$.
If we replace every edge weight with its square, p would be $(e_1^2+e_2^2+\cdots+e_n^2)$ and p' would be $(e_1'^2+e_2'^2+e_3'^2+\cdots+e_n'^2)$. However, $(e_1+\cdots+e_n) < (e_1'+\cdots+e_n')$ doesn't guarantee $(e_1^2+e_2^2+\cdots+e_n^2) < (e_1'^2+e_2'^2+\cdots+e_n'^2)$ when all edges are associated with a positive edges as stated in the question.
An example to support this would be $5 < 1+2+3$, 5 is smaller than 6.
However, if we square it, $5^2 > 1^2+2^2+3^2$.

(b) Squaring the weights of each edges does not change the minimum spanning tree, so we get the same tree. The minimum spanning tree only depends on the ordering among the edges. This is because the only thing we do with edges are comparing them. Let's say there is initially the MST is T. After each edge is squared, let's say the MST chan es to T'. Therefore, there would be at least one edge $(u,v) \in T$ but $(u,v) \notin T'$. (If all edges are weights of 1, no need to check) Suppose we add edge (u,v) into tree T' and make a tree $T'' = T' + (u,v)$. Since (u,v) is not in T' therefore (u,v) must be the longest edge and when we square root each weight by 1 (u,v) will still be the longest edge in T. However, this is a contradiction. Since, longest edge (u,v) cannot be contained in MST T, therefore $(u,v) \notin T$, this proves that trees T and T' are the same.

## Question 3

For this algorithm we define the overlap (pair1, pair2) between two intervals sorted by their starting Si values. A case where the intervals overlap would be when pair1.si <= pair2.si, compare the pair1_fi to pair2_si. If pair1_fi > pair2_si, then there is an overlap between the two intervals. However, if pair1_fi< pair2_si then there are no overlaps. To solve this problem, we use a divide and conquer approach and we are using two pointers left, right that each point to the first starting point of the interval and the last finishing point of the interval of the sorted array.

```
int longest_overlap(int left, int right) :
    if left == right :
        return 0 #base case for recursions

    int mid = (left + right) / 2 #compute the mid point for divide and conquer
    #recurse through the sorted array using recursion
    int left_overlap = longest_overalp(left, mid)
    int right_overlap = longest_overalp(mid+1, right)

    #To check whethere there is a maximum overlap in between the intervals
    #of the two halves. Since we sorted the array before the recursion,
    #the max is possible between the max fi of the first half and smallest
    #si of the second half.

    for interval in intervals :
        lastFinishingTime = interval with the max ending time fi

    firstStartingTime = the first starting interval which is
    the first element of the second half
    int overlapIncludingMid = overlap(lastFinishingTime, firstStartingTime)

    #This divide and conquer algorithm works by dividing the array into
    #two halves and finding the max overlap that are overlapping the two halves.
    #However, there might be a case where the longest overlap doesn't cross between
    #the two halves but either be in the left side or the right side.
    #This algorithm also takes care of that case while merging the recursion.
    return max(max(left_overlap, right_overlap), overlapIncludingMid)
```

Sorting the array before the algorithm takes an O(nlogn) time complexity. The interval that finishes the last (fi) can be approached in O(n) time and with the interval that starts the first si, can be found in O(1) since the array is already sorted and it’s the first element of the right half. The overall time-complexity would be O(nlogn) since both sorting and divide and conquer takes O(nlogn).

The algorithm can be proved using induction. The base case for the induction would be, for any 2 intervals, the maximum overlap would be the overlap between the two

intervals which can simply be the first element’s finishing time subtracted by the second element’s starting time. Only if this is greater or equal to 0.
Then let’s say this algorithm has calculated the maximum overlap for all intervals with length 1, … n - 1 and we want to find the maximum overlap in an array with n intervals. As we did in the above algorithm, we first split the array into halves, one from the first element to the mid element, and two from the mid + 1 element to the last element of the sorted array. We already have the maximum overlap of the first half using the inductive hypothesis so we calculate the maximum overlap of the second half recursively. Finally to compare the maximum overlap of the two halves as described on the above algorithm and then return the maximum maximum of all these sub-maximums.

## Question 4

Put the cards in a list with random orders and divide the list into two parts, one including the lower half of the card and the other the higher half of the cards. Let us assume that on each of the smaller lists, we have a function that runs on both of these two sub-lists, F returns a pair (num,c) such that account number *num* occurs *c* times in the list and c is greater than half the list, and if no such *num* exists, then F returns (-1, 0).

Let's say F returns (num1, c1) from the first list and (num2, c2) from the second list. If num1 == num2, then we return (num1, c1 + c2). If num1 != num2, we search list 2 for num1 updating c1 and for search list 1 for num2 updating c2. If c1 > c2, we return (num1, c1), if c2 > c1, then we return (num2, c2), and if c1 == c2, then we return (-1,0). If we have returned a positive number c, then a set of more than n/2 cards is equivalent to each other, and otherwise there is no such set.

The description above shows a recursive procedure for computing F function. The first paragraph, F needs bases cases for an empty list by returning (-1,0), a list of one by returning (one_card,1), and a list of two cards one_card, two_card returning (one_card, 2) if one_card == two_card, and else (-1, 0). If the list is larger than 2, then a divide and conquer algorithm can be used. So divide the list into two sub-lists and run F on both of the sub-lists. After F returns from each of these runs, the output will come out as mentioned in the second paragraph, it combines and returns the following pair.
The computation in the second paragraph would be a total of n iterations. Thus, the cost of running F is T(0) = 0, T(1) = 0, T(2) = 2, …. T(n) = 2T(n/2) + n for n > 2 (T(n/2) for each of the sub lists and a total of 2T(n/2)). Overall, this is a divide and conquer algorithm and has an O(nlogn) time complexity.