Assignment 2

Vaibhav Agarwal(B15139)

Q1. I implemented Huffman Coding algorithm with and without using priority Queue. The runtime complexity on using the priority queue is **O(n*log(n))** where n is the number of letters that is to be encoded using Huffman Coding. This is because 2*(n-1) extract _min operations take place where each extract min takes O(log(n)) time. On the other hand , without using priority queue, the runtime complexity was **O(n^2)** , it was because the entire array had to be searched to find the two lowest frequency characters and then they deleted from the array for each of the n-1 iterations. Each of the above operations take O(n) in a unordered array. Hence, the priority queue gave quicker result as compared to the unordered array when using Huffman Coding algorithm.

Psuedocode:

HuffmanCode(C):

```
n = size(C)
Q = C
for i = 1 to n−1 do
    allocate a new node z
    z.left = x = Extract_Min(Q)
    z.right = y = Extract_Min(Q)
    z.freq = x.freq + y.freq;
    Insert(Q,z)
end for
printCode(Q)
```

Q2. I made a |V| * |E| matrix M , such that such that M_{ve} = -1 if edge e leaves vertex v, Mve = 1 if edge e enters vertex v, and M_{ve} =0, then on checking a given set of edges if all those edges constitute a cycle than the columns for these edges would be linearly dependent. Otherwise, the columns are independent. The time complexity for this algorithm was **O(V*E)**. This is because at most we will have to check all the columns in the matrix and each columns has entries for all V vertices.

Q3. I implemented this problem using modifications in Kruskal Algorithm. The runtime complexity for Kruskal's algorithm is **O(ElogE)** or **O(ElogV)** as it requires sorting of all the edges based on their weights. I sorted the edge weight in descending order and hence found the maximum spanning tree of the graph, that is also the maximum weight acyclic subset.

Pseudo Code

KruskalModified(G):

```
A = ∅
```

```
For each vertex v ∈ G.V:
   Make_Set(v)


For each edge (u, v) ∈ G.E ordered by decreasing order by weight(u, v):
   if Find_Set(u) ≠ Find_Set(v):
   A = A ∪ {(u, v)}
   Union(u, v)
return A
```

Q4. I sorted the individual task time in a ascending order. If the tasks are performed in that order the average completion time taken for all tasks is minimum. The runtime complexity for this algorithm is **O(nlogn)** , that is time taken to sort the values of task times.

Pseudo Code:

```
MinimumAverageCompletionTime(List Of Tasks):

Sort all the tasks time taken in ascending order.
While list of tasks is not empty :
         Pick a task from list of task and and run it.
Calculate average completion time.
```

Q5. I used graph coloring for this problem. Initially , we are provided with the start and finish times of activities so i made a graph where each vertex is a activity and edge between two activity denotes there is a clash in the time of activities. We need to color each vertex of the graph, such that no adjacent vertex receive the same color. This means , all the incompatible activities get different classrooms. The runtime complexity for the solution of this problem is O(n^2).

Pseudo Code:

```
ActivityRoom(G(V,E)) :
For every vertex v ∈ V :
    color(classroom) ← 1
    For every vertex u ∈ V such that (u,v) ∈ E  :
         If color(classroom) matches adjacent node
             color(classroom) ← color(classroom) + 1
   classroom [v] ← color(classroom)
```

Q6. I implemented this problem using Kruskal Algorithm. The runtime complexity for Kruskal's Minimum Spanning tree algorithm is **O(ElogE)** or **O(ElogV)** as it requires sorting of all the edges based on their weights. So replacing edge weights with their square won't affect the minimum spanning tree, since order of weights doesn't change on squaring (edge weights are positive and distinct), the same set of edges will be part of MST as before.

Pseudo Code

Kruskal(G):

```
A = $\varnothing$

For each vertex v $\in$ G.V:
    Make_Set(v)


For each edge (u, v) $\in$ G.E ordered by increasing order by weight(u, v):
    if Find_Set(u) $\neq$ Find_Set(v):
    A = A $\cup$ {(u, v)}
    Union(u, v)
return A
```