Name: Viswa Nihar Nukala UBID: 50414392

Q.① Given n sorted arrays $A_1, A_2, A_3, A_4 \ldots A_n$. Need to merge all the arrays with cost of merging A_i and A_j is $k_i + k_j$. Find minimum total cost to merge all arrays. We can do this using 2 different queues in the following ways:-

```
1.  Total_minimum_cost(A):        ← array consisting number of elements
2.      Sort(A)
3.      Queue Q1
4.      Queue Q2
5.      for i=0 to len(A):
            Q1.add(A[i])
6.
7.      cost = 0
8.      while !(Q1.size == 1 && Q2.size == 0) ||
            (Q1.size == 0 && Q2.size == 1):
9.          C1, C2 = MAX_INT
10.         if Q1.size > 0 && (Q2.size == 0 ||
                Q1.peek ≤ Q2.peek)
11.             C1 = Q1.pop
12.         else
13.             C1 = Q2.pop
```

```
14.     if Q1.size >0 and (Q2.size ==0 or
                           Q1.peek <= Q2.peek):
15.         C2 = Q1.pop
16.     else
17.         C2 = Q2.pop
18.     Cost = cost + C1 + C2
19.     Q2.add (C1 + C2)
20. return cost
```

The above algorithm first sorts the array containing the length of all arrays. We then maintain 2 queues. The first queue contains all the lengths. The second contains the cost costs by minimising cost after every iteration.

We can ignore line 5 and 7 and use the same array.

Runtime analysis:-

Sorting the array takes $n\log(n)$ time in efficient way.

In worst case, we pop only one element from the first queue every iteration and hence the "while" loop takes $O(n)$ time.

Therefore the runtime for the whole algorithm is $O(n\log n)$.

Q.2) Given a set of jobs $J = \{job_1, job_2, \ldots\}$, deadline for all the jobs, profit for all the jobs.
Need to return valid job schedule with maximum profit.

Solution:-
We first sort all the jobs in decreasing order of the profit. We then keep checking the job, find a time slot for the job such that it is completed before the deadline and is closest to the deadline. If no such time slot is found, we ignore it.
Following is the algorithm:

```
1. Max_profit_job_scheduling(J, d, p):
       Sort J, d and p in decreasing order of profit.
2.     n := length(J)
3.     result = [False] * length(J)
4.     job = [-1] * n
5.
```

To keep track of the time slots filled.

To keep track of all valid jobs.

```
6.      for i=0 to n:
7.          for j = n-1 to min(n-1, d[i]-1):
8.              if result[j] = False:
9.                  result[j] = True:
10.                 job[j] = J[i]
                    break
11.
12.     return job without -1s.
```

In line 7-9, we check for a time slot which is before it's deadline and also which is filled.

Hence doing this we make sure that the jobs with higher profits are scheduled and then we try to fit in the jobs with lower profit.

Runtime analysis :-

The line 2 takes $O(n \log n)$.

The lines 6 - 11 take n^2 in worst case where if has to iterate over all the time slots.

Hence the runtime analysis is $O(n^2)$.

(3) Given a set of jobs $J = \{J_1, J_2, J_3 \ldots J_4\}$ and also associated processing time T and importance factor W for each job. Completion time of J_i is given by when it is completed and is denoted by C_i. The dissatisfaction rate is given by $w_i * C_i$. Goal is to minimize the total dissatisfaction

$$\sum_{i=1}^{n} w_i C_i$$

Solution:-

We can start scheduling jobs based on the difference of the weight and the processing time.

Hence following is the greedy algorithm:

1. Minimize_Total_Dissatisfaction (J, T, W):
2. GS = T - W ← Greedy score
3. Sort T, W and GS in decreasing order of GS
4. time = 0
5. weighted_sum = 0

```
6.      for i=0 to length(T):
7.          time = time + T[i]
8.          weighted_sum = weighted sum + [w[i] x time]
9.      return weighted_sum
```

Note :-

In line 3, if there is a tie between the greedy scores, then you can break the tie by selecting the job with highest weight as in this case, completing the job with highest weight early reduces the weighted sum.

Runtime analysis :-

Line 2 takes $O(n)$ time if we consider writing a for loop.

Sorting the list would take $O(n \log n)$

Lines 6-8, to calculate the weighted sum, we need to iterate over all jobs, hence the runtime for that would be $O(n)$.

Hence the runtime for the whole algorithm is $O(n \log n)$.

Proof of correctness :-

Greedy Choice property :-

The activity with highest weight and least processing time is picked up first by the greedy algorithm.

Hence we can say that the optimal solution needs to have this job also as the first job to be executed. If not then the first job to be executed in the optimal solution should have the highest weight.

Optimal Substructure property :-

We can say that if we execute the job with the highest weight and also the highest time first, then even though the weight for the 2nd job may be low but the cost for executing the 2nd job will be higher and hence the dissatisfaction increases.

Again if we prioritise the jobs based on processing time, then there will be cases when higher weight jobs are executed later. Hence it is feasible and optimal to take the job with higher greedy score (i.e., difference between weight and processing time) to be executed first.

(7) Let $G = (V, E)$ be a directed graph. Transpose of G is denoted by G^T and $G^T = (V, E^T)$ where:-

$$E^T = \{ v \to u \mid u \to \in E \}$$

(a) M_G is adjacency matrix. Describe to construct M_{G^T} of G^T in $\theta(n^2)$.

Solution :-

We traverse over each element of the graph and then change the is to 0's if there is no edge in M_{G^T} and vice-versa.

```
1. Transpose_of_Graph(M):
2.         n = len(M)
           M_G = Matrix of n x n with all 0's.
3.         for i=0 to n:
4.             for j=0 to n:
5.                 if M[j][i] == 1
6.                     M_G[i][j] = 1
7.
8.         return M_G
```

Since there are 2 for loops with size n, the runtime is $O(n^2)$ where n is the number of edges.

(b) Suppose G is represented by adjacency list. Describe how to construct the adjacency list representation of G^T.

We have an array with list corresponding to the all the neighbours. Following is the algorithm.

```
1.  Tranpose_of_Graph_List(M):
2.      n = len(M)
3.      M_G = array of list with size n.
4.      for i=0 to n:
5.          list_neighbours = M[i]
6.          list_len = len(list_neighbours)
7.          for j=0 to list_len:
8.              vertex = list_neighbours[j]
9.              M_G[vertex].append(i)
10.     return M_G
```

First we tranverse the array and then traverse list wise and add the corresponding to the edge in the transpose array.

Runtime analysis :-

Traversing the first loop means traversing the vertices and traversing the list means traversing the neighbouring edges of vertex.

Hence we need to traverse all the vertices and edges.

Hence if we n is the number of vertices and m is the number of edges then runtime is $O(nm)$.