CSC373H1 - Assignment 1

Question 3

a) The algorithm increments in employees and assigns the first available task with skill demand $\leq$ the current employee's skill, if it exists. If not, the algorithm moves to the next employee.

Algorithm 1 Task Assignment

```
Sort the employees in a non-decreasing order of their skill level so that s_1 ≤ ... ≤ s_n
Sort the tasks in a non-decreasing order of their minimum skill requirement so that d_1 ≤ ... ≤ d_m
j ← 1                                                    ▷ Increments the tasks
for Employee i = 1, ..., n do
    if d_j ≤ s_i then
        Assign task j to employee i
        j ← j + 1
    end if
    if j > m then                                        ▷ No more tasks left
        Break the for loop
    end if
end for
```

b) The worst-case runtime is $O(n \log n + m \log m)$. Sorting the employees takes $O(n \log n)$ time, while sorting the tasks takes $O(m \log m)$ time. For each of the n employees, a constant amount of work is done.

c) Define G_i and O_i to be the sets of tasks assigned to employees $1, \ldots, i$ by the greedy (GRD) and optimal (OPT) approaches respectively. Since GRD considers the employees in order, assume WLOG that OPT also does this (see Lemma). We proceed by induction to show that $\forall i \in \{1, \ldots, n\}, |G_i| = |O_i|$, where $|\cdot|$ denotes set size. This implies $|G_n| = |O_n|$, so GRD is as optimal as OPT.

Base case ($i = 1$): Assume both OPT and GRD has not considered any employee after 1 at this point. Consider two cases:

- Case ($\exists$ task j s.t. $d_j \leq s_1$): OPT assigns employee 1 to such a task. For GRD, this case implies that $d_1 \leq s_1$, so it assigns employee 1 to task 1. Thus, $|G_i| = |O_i| = 1$.
- Case ($\forall$ tasks j, $d_j > s_1$): Both OPT and GRD do not assign employee 1 to a task, so $|G_i| = |O_i| = 0$.

Induction step ($i > 1$): Assume $|O_{i-1}| = |G_{i-1}|$ and both OPT and GRD has not considered any employee after i at this point. Suppose for contradiction that $|O_i| > |G_i|$, which means O_i includes a task j assigned to employee i, but that G_i does not assign employee i to any task. Define $A_i = \{\text{tasks } j : d_j \leq s_i\} \setminus O_{i-1}$ to be the set of available tasks with demands $\leq s_i$, and similarly $B_i = \{\text{tasks } j : d_j \leq s_i\} \setminus G_{i-1}$. Task $j \in A_i$ but $\notin B_i$ since GRD must have assigned it to some other employee before employee i. Note that $|A_i| = |B_i|$ since $|O_{i-1}| = |G_{i-1}|$ and both OPT and GRD has not considered any employee after i. This means B_i must contain an additional available

task with demand $\leq s_i$ for the sets to be equal in size. Then, GRD can just assign employee i to this task, yielding a contradiction. We conclude $|O_i| = |G_i|$, as needed.

Lemma: Define OPT as the optimal approach that considers the employee in order and A as another approach that does not follow this order but is otherwise as similar to OPT as possible. We show OPT is at least as optimal as A. Since A does not follow the order, there must be ≥ 1 pair of employees x, y s.t. it considers y before considering x and $x < y$. Note $x < y \implies s_x \leq s_y$.

- If A assigns y to task m and x to task n, this implies either
  - d_m, d_n both $\leq s_x, s_y$, in which case OPT might make the same or opposite assignments depending on the order of m and n, or
  - $d_n \leq s_x$ and $d_m \in (s_x, s_y]$, in which case OPT would make the same assignments.
- If A assigns y to task m and leaves x unassigned, this implies either
  - $d_m \leq s_x$, in which case OPT would assign x to m and leave y unassigned instead, or
  - $d_m \in (s_x, s_y]$, in which case OPT would make the same choice.

  Notice however that for the first sub-case, if there is an available task n s.t. $s_x < d_n \leq s_y$, OPT would also assign y to n, meaning it has 1 more task assigned than A.
- If A leaves both x, y unassigned, this implies there are no available tasks m s.t. $d_m \leq s_y$, in which case OPT would make the same choice.

In all cases, the number of tasks assigned to each pair of employees by A is $\leq$ that of OPT.

Question 4

a) After each iteration d, the corresponding project d should not have any employee who feels inferior since the algorithm should have moved all such employees to project $d + 1$. Since the algorithm stops when the inferiority-freeness constraint holds for a newly-created project k, it must hold for all previously created projects as well.

b) Base case: Suppose $k = 2$. Since i_2 feels inferior to i_1, any solution must assign i_2 to project 2 while keeping i_1 in project 1.

Inductive step: Let $k > 2$, and assume any solution assigns $(i_1, \ldots, i_{k-1})$ to $k-1$ different projects since $s_1 > \ldots > s_{k-1}$ and $t_1 < \ldots < t_{k-1}$. Note that i_{k-1} feels inferior to i_k implies any employee before i_{k-1} must also feel inferior to i_k, since $s_1 > \ldots > s_{k-1} > s_k$ and $t_1 < \ldots < t_{k-1} < t_k$. Thus, i_k cannot be assigned to the same project as any other employee before it. This means any solution must assign i_k to a new project k, and so any solution must assign $(i_1, \ldots, i_k)$ to k different projects.

c) Consider an arbitrary set of employees, and assume it is sorted as in part d. If it has ≥ 1 sequence s.t. each employee feels inferior to the previous, it must have a longest such sequence. (Notice the sequence does not have to be contiguous.) Define this longest sequence as $S = (i_1, \ldots, i_k)$ for some $k \leq n$. By part b, any solution must have at least k projects. Notice that the greedy solution (GRD) uses exactly k projects for this set. This is because for every employee:

- By the Lemma, the employee does not belong to S $\implies$ there must exist ≥ 1 employee in S where neither employee feels inferior to the other. So, GRD can assign it to the same project as ≥ 1 employee in S without it or the other feeling inferior. This does not affect the overall number of projects.
- If GRD cannot assign the employee in this manner, by the contrapositive of the Lemma, it must belong to S. GRD, which produces a feasible solution by part a, already splits S up between k projects by part b.

We conclude that # of projects needed by GRD $= k \leq$ # of projects needed by any solution.

Lemma: We show that an employee $i \notin S \implies \exists j \in S$ where neither i nor j feels inferior to the other. Suppose for contradiction that $i \notin S$ but $\forall j \in S$, either i feels inferior to j or vice versa.

- If all j's feel inferior to i or i feels inferior to all j's, i must be at the beginning or end of S by definition, a contradiction.
- If $\exists j \in S, s_j < s_i \wedge t_i < t_j$ and $\exists j' \in S, s_{j'} > s_i \wedge t_i > t_{j'}$, then $s_j < s_i < s_{j'} \wedge t_{j'} < t_i < t_j$, and i must be nested within S by definition, a contradiction.
- If the previous cases cannot be reached, then there must exist an employee $j \in S$ where neither i nor j feels inferior to the other, a contradiction.

d) For the algorithm, since it sorts the employee twice, it only needs to compare each adjacent pair of employees.

Algorithm 2 Project Division

Sort the employees in order of their start time so that $s_1 \leq \ldots \leq s_n$
If there are employees sharing a start time, sort them in non-decreasing order of their finish time
Initialize an empty set Project 1 as a linked list and add the employees in order
CurrentProject $\leftarrow$ Project 1
InferiorityExists $\leftarrow True$
$d \leftarrow 2$
while InferiorityExists is $True$ **do**
    Initialize an empty set Project d as a linked list
    PreviousStart $\leftarrow$ start time of first employee in CurrentProject
    PreviousEnd $\leftarrow$ finish time of first employee in CurrentProject
    for Employee e_i in CurrentProject **do** ▷ Iterating through the linked list
        if PreviousStart $< s_i$ and PreviousEnd $> t_i$ **then**
            Add $\{e_i\}$ to Project d
            Remove $\{e_i\}$ from CurrentProject
        else
            PreviousStart $\leftarrow s_i$
            PreviousEnd $\leftarrow t_i$
        end if
    end for
    if Project d is $\emptyset$ **then** ▷ Only happens when CurrentProject is inferiority-free
        InferiorityExists $\leftarrow False$ ▷ Breaks the while loop
    else
        CurrentProject $\leftarrow$ Project d
        $d \leftarrow d + 1$
    end if
end while

The algorithm runs in $O(n^2)$ time since there can be at most n projects and n iterations in the while loop. For each project, the algorithm loops over all $\leq n$ employees assigned to it and performs a constant amount of work. Also, sorting the employees takes $O(n \log n)$ time, and moving employees from one project to another takes $O(n)$ time overall due to the linked list implementation.