# CMSC 27230

Owen Fahey

January 2023

## Problem 1.a

| $v$ | $d_v$ | Priority Queue: $\{(v_1 : d_{v_1}$, last vertex in path of $d_v)$, ...$\}$ |
|---|---|---|
| $s$ | 0 | $\{(a:3,s),(c:8,s),(e:10,s)\}$ |
| $a$ | 3 | $\{(b:5,a),(c:8,s),(e:10,s)\}$ |
| $b$ | 5 | $\{(c:7,b),(e:10,s),(d:15,b),(t:17,b)\}$ |
| $c$ | 7 | $\{(e:10,s),(d:13,c),(t:17,b)\}$ |
| $e$ | 10 | $\{(f:11,e),(d:13,c),(t:17,b)\}$ |
| $f$ | 11 | $\{(d:13,c),(t:17,b)\}$ |
| $d$ | 13 | $\{(t:16,d)\}$ |
| $t$ | 16 | $\varnothing$ |

The two shortest paths from $s$ to $t$ are:

$$\{(s,e),(e,f),(f,d),(d,t)\},$$

$$\{(s,a),(a,b),(b,c),(c,d),(d,t)\}$$

.

## Problem 1.b

**Kruskal's Algorithm:**

Order in which edges are considered: $(1, 3, 4, 5, 8, \underline{10}, 12, 14, \underline{15}, \underline{17}, 18)$

Underlined edges were not accepted.

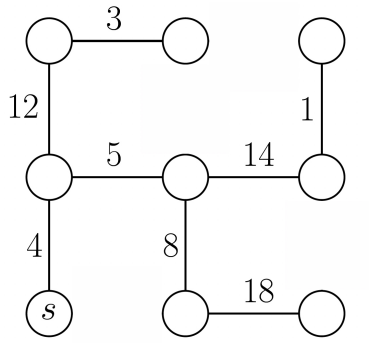


Figure 1: $T$ of $G$

**Prim's algorithm**:

Order in which edges are found: (4, 5, 8, 12, 3, 14, 1, 18)

## Problem 2

**Solution:** Use the follow greedy algorithm: serve tables based on the value of $\frac{u_i}{l_i}$ in descending order. The time complexity will be $\Theta(nlog(n))$.

**Proof Of Optimality:**

*Approach:* Use an exchange argument to show that this is the optimal strategy

*Setup:* We want to show that the total unhappiness that occurs from serving tables according to the ordering given by this algorithm is less than or equal to the total unhappiness that occurs when serving tables according to any other ordering. Let $T$ be a function corresponding to an ordering of tables. $T$ takes a table and returns the position that table gets served (e.g., if $T(i) = 1$, table $i$ would be the first table to be served).

*Claim:* Switching two tables adjacent in an ordering does not affect the time at which other tables get served. This means a switch between adjacent tables does not affect the unhappiness contributed by tables not involved in the switch.

*Proof:* Consider two tables $a$ and $b$ involved in a switch where before the switch: $T(a) = p$ and $T(b) = p + 1$ and after the switch: $T(a) = p + 1$ and $T(b) = p$. Let $i$ be some other table. We know that $t_i = l_i + \sum_{j \in J} l_j$ where $J$ is the set of tables served before $i$. Since $t_i$ only depends on tables served before $i$, $t_i$ is unaffected if $T(i) < p$. If $T(i) \geq p + 2$, $a$ and $b$ are still served before $i$. Since $l_a$ and $l_b$ do not change following the switch, $t_i$ does not change after the switch.

*Claim:* Given two tables $a$ and $b$ where $T(a) = p$ and $T(b) = p+1$, switching $a$ and $b$ will decrease unhappiness if $\frac{u_b}{l_b} > \frac{u_a}{l_a}$. In other words, given two tables adjacent in an ordering, it is optimal for the table with the higher $\frac{u}{l}$ to be served first.

*Proof:* We know from the previous claim that when a switch between adjacent tables occurs, the total unhappiness contributed by tables that are not involved in the switch remains unaffected. Hence, we are interested the unhappiness contributed by $a$ and $b$ before and after the switch. Before the switch this quantity is equal to:

$$u_a t_a + u_b t_b = u_a(l_a + \sum_{j \in J} l_j) + u_b(l_b + l_a + \sum_{j \in J} l_j) = u_a l_a + u_b l_b + u_b l_a + (u_a + u_b) \sum_{j \in J} l_j$$

where $J$ is the set of tables served before $a$. After the switch this quantity is equal to:

$$u_a t_a + u_b t_b = u_a(l_a + l_b + \sum_{j \in J} l_j) + u_b(l_b + \sum_{j \in J} l_j) = u_a l_a + u_a l_b + u_b l_b + (u_a + u_b) \sum_{j \in J} l_j$$

We are interested in finding out whether $u_a t_a + u_b t_b$ is greater before or after the switch. Let us set up an unknown inequality with before on the left and after on the right:

$$u_a l_a + u_b l_b + u_b l_a + (u_a + u_b) \sum_{j \in J} l_j \stackrel{?}{>} u_a l_a + u_a l_b + u_b l_b + (u_a + u_b) \sum_{j \in J}$$

$$u_b l_a \stackrel{?}{>} u_a l_b$$

$$\frac{u_b}{l_b} \stackrel{?}{>} \frac{u_a}{l_a}$$

A switch decreases total unhappiness (and is therefore optimal) if the unhappiness contributed by $a$ and $b$ is greater beforehand than afterwards. In other words, if $\frac{u_b}{l_b} > \frac{u_a}{l_a}$.

We can apply this argument repeatedly on adjacent tables that are not ordered from largest to smallest $\frac{u_i}{l_i}$ to justify swaps. If we keep doing this we will eventually arrive at an ordering where tables are sorted in descending order by $\frac{u_i}{l_i}$. It is worth pointing out that this may not be the same ordering as given by the algorithm because of ties between tables in their $\frac{u_i}{l_i}$; however, the total unhappiness will be the same. □

**Run time analysis:**

The run time of the algorithm that finds the optimal ordering is $O(nlog(n))$.

*Proof:* First, we must find the value of $\frac{u_i}{l_i}$ for each table. This will take $n$ steps. Then we must sort these values. This will take $\Theta(nlog(n))$ if we use merge sort. Putting this all together we get that the run time of our algorithm will be $\Theta(nlog(n))$.

## Problem 3

**Solution:** Use the follow greedy algorithm:

1. Sort the employees by smallest to largest end time
2. Promote the employee with the latest ending time of the employees that overlap with the employee with earliest end time (this includes the employee with earliest end time).
3. If all employees overlap with a manager then the process is finished. Otherwise, find the employee with the earliest end time who does not have a manager.
4. Promote the employee with the latest ending time of the employees that overlap with the employee with earliest end time who does not have a manager (this includes the employee with earliest end time who does not have a manager).
5. Go back to 3.

The run time of this algorithm will be $\Theta(nlog(n))$.

**Proof of Optimality:**

*Claim:* This process assigns everyone a manager.

*Proof:* The process only terminates when everyone has a manager. It is also always possible for the algorithm to assign a manager to an employee without a manager as that employee can always be promoted to manager. Therefore, everyone gets assigned a manager.

*Setup:* Let us say the $k^{th}$ step assigns the $k^{th}$ manager. Let us define the set of workers ordered by end time as $U = \{w_1, ..., w_n\}$. Let the set of managers given by the algorithm be $M = \{w_{i_1}, ..., w_{i_r}\}$ where $r$ is the final number of managers. Let $w_{i_j}$ correspond to $[a_{i_j}, b_{i_j}]$.

*Claim:* $b_{i_1} < b_{i_2} < ... < b_{i_r}$

*Proof:* Assume $k \geq 2$. As part of each step, we find the worker, $w_e$, with the earliest end time who does not have a manager. We know that $w_{i_{k-1}}$ is the last assigned manager and $w_{i_k}$ will be assigned this step. We can conclude $b_{i_k-1} < b_e$, as if this was not true, $w_e$ would have a manager. Since $w_e$ must overlap with the next manager $w_{i_k}$, it follows that $b_e \leq b_{i_k}$. Therefore, $b_{i_{k-1}} < b_{i_k}$.

*Claim:* $a_{i_1} < a_{i_2} < ... < a_{i_r}$

*Proof:* Let us compare the starting time of the managers assigned at step $k$ and $k+1$. As part of step $k$, we find the worker, $w_e$, with the earliest end time who does not have a manager. We know that $a_{i_k} \leq b_e \leq b_{i_k}$ because the $k^{th}$ manager must overlap with $w_e$. We also know that $b_{i_k} < b_{i_{k+1}}$ from the previous claim. Suppose $a_{i_k} \geq a_{i_{k+1}}$. This would be mean that: $a_{i_{k+1}} \leq a_{i_k} \leq b_e \leq b_{i_k} \leq b_{i_{k+1}}$. In words, $w_{i_{k+1}}$ would overlap with $w_e$ and end later than $w_{i_k}$. If this were true, then we would have chosen $w_{i_{k+1}}$ as the $k^{th}$ manager. However, $w_{i_{k+1}}$ is the $(k+1)^{th}$ manager. Hence, $a_{i_k} < a_{i_{k+1}}$ by contradiction.

*Claim:* No one starts after $b_{i_r}$

*Proof:* This is quite straightforward since anyone who starts after the last manager ends would not have a manager. Since, the algorithm assigns everyone a manager, this can never happen.

*Setup:* Suppose that $M'$ is an alternate final solution to the problem sorted least to greatest by end time. Let $p_j$ be a function that takes some $M$ and returns the $b$ value of the $j^{th}$ element of $M$. If $|M| < j$, then $p_j(M)$ returns $\infty$.

*Claim:* $p_i(M)$ is always $\geq p_i(M')$ for any $i$.

*Proof:* At each step, we are choosing to promote the worker $w_{i_k}$ with the latest possible end time $b_{i_k}$ such that all workers without a manager before $b_{i_k}$ get a manager. A key observation is that if we were try to designate any other manager with a later $b_i$ as the $k^{th}$ manager, then at least one employee before this manager's start time would be without a manager. Hence, at each point, we are choosing the manager that maximizes $p_i(M)$.

*Claim:* This strategy minimizes $|M|$ and is therefore optimal

*Proof:* From the previous claim, we know $p_r(M) = b_{i_r} \geq p_r(M')$ and $p_{r+1}(M) = \infty \geq p_{r+1}(M')$. In other words, the ending time of the last manager according to $M$ will be as late or later than the ending time of the manager in the equivalent position according to $M'$. This means that there is no case in which $|M| > |M'|$. □

**Detailed Implementation Example:**

1. Make a list of employees and sort them least to greatest by their start time using merge sort. Let us call this list $L_a$. Make another list of employees and sort them by least to greatest by their end time using merge sort. Let us call this list $L_b$. Create two pointers, $P_a$ and $P_b$, pointing to the first employee in each list.

2. Let us say the employee currently being pointed at in $L_b$ is $i$ and that $i$ has a start time of $a_i$ and an end time of $b_i$. Store this information as follows:

   $end = a_i$ ($end$ will track the earliest end time of currently unmatched employees)

   $max = b_i$ ($max$ will track the latest end time of employees who start at or before $end$)

   $num = i$ ($num$ will track the employee whose end time is stored $max$)[1]

3. Let us say the employee currently being pointed at in $L_a$ is $k$ and that $k$ has a start time of $a_k$ and an end time of $b_k$. Check if $a_k$ is less than or equal to $end$.

   I. If *true*: check if $b_k$ is greater than $max$.

      i. If *true*: set $num$ to $k$ and $max$ to $b_k$. Point $P_a$ to the next employee in $L_a$. If this employee exists, return to 3. Otherwise, add $num$ to a list of managers $L_m$ and terminate the process.

      ii. If *false*: point $P_a$ to the next employee in $L_a$. If this employee exists, return to 3. Otherwise, add $num$ to a list of managers $L_m$ and terminate the process.

   II. If *false*: then add $num$ to a list of managers $L_m$. Point $P_b$ to the next employee in $L_b$ and check if they exist.

      i. If *they exist*, check if their start time is greater than $max$.

[1] Storing both $num$ and $max$ may be redundant but it is convenient for explanation

A. If *true*: go back to 2

B. If *false*: point $P_b$ to the next employee in $L_b$ and check if they exist.

If *they exist*: go to 3.II.i

If *they do not exist*: go to 3.II.ii

ii. If *they do not exist*: terminate the process.

**Run Time Analysis:**

Now let us look at the run time of different sections. Sorting twice takes $\Theta(nlog(n))$.

Unpacking the run time of the rest of the algorithm is a bit trickier. We are stepping through two lists. In the worst case, ever employee in each list is visited once. In the best case, we visit one employee in $L_b$ and every employee in $L_a$. During a visit, the algorithm might do one or both of the following: store the employee's information OR make at most two comparison. If we consider a visit and one or both of these actions to be a step, then during this phase of the algorithm, $2n$ steps are taken in the worst case and $n + 1$ steps are taken in the best case. Hence, this part of the process is $\Theta(n)$.

Putting this all together, we can conclude the run time is $\Theta(nlog(n))$.

## Problem 4

**Solution:** In order to solve this problem, we will think about taking steps in reverse order. This makes the problem much easier to think about as it easily deals with the complex trade offs between treasure value and longevity. To illustrate what the problem looks like in reverse: we start with whatever treasure is available at $t = T$ (i.e. any treasure that will not be available after $T$). We make a choice from the available treasure and then proceed to $t = T - 1$. At each new time, new treasure may become available. We keep stepping backwards until we make our last choice at $t = 1$. Consider the following greedy algorithm which will tell you what treasure to take at each time:

1. At $t = T$, choose whatever available treasure has the greatest value
2. Move backwards to $t - 1$. If $t = 0$, terminate the process. Otherwise, choose whatever available treasure has the greatest value. Repeat this step.

The run time of this algorithm will be $\Theta(nlog(n))$.

**Proof of Optimality:**

*Approach:* Stays ahead

*Claim:* This algorithm is guaranteed to obtain the maximum possible total value

*Proof:* Assume there exists a different treasure selection strategy that results in a greater total value. Let's call the total value obtained by this strategy $S'$ and the total value obtained by the greedy algorithm strategy $S$. Since the greedy algorithm selects the treasure with the highest value at each time step, we know that the total value obtained by the greedy algorithm $S \geq S'$. Hence, our initial assumption must be false. □

**Run time analysis**[2]:

The run time of this algorithm is $\Theta(nlog(n))$. In order to run this algorithm, we must know what treasure that comes into availability (still thinking about the problem backwards) at each $t$ and which treasure has the greatest value at time $t$. One way to do this is to use a priority queue. This specific implementation will describe using a binary max heap. The run time of relevant operations are:

Build: $\Theta(n)$

Insert: $\Theta(log(n))$

Delete-Max: $\Theta(log(n))$

First, we need to know when treasures become available. This can be done by creating a list of treasures, $L_s$, and sorting them with merge sort from largest to smallest $t_i$. This will take $\Theta(nlog(n))$.

Next, we must build the priority queue starting with whatever treasure is available at $t = T$. This requires stepping through however many elements in $L_s$ have a $t_i = T$. In the best case, this will

[2]It may be possible to implement this algorithm with a better run time. I think using a Fibonacci heap gives a better amortized run time

be 1 element and in the worst case, this will be $n$ elements. Once, we have found the treasures available at $t = T$ we will build a the priority queue which will take $\Theta$(# of treasures at $t = T$) or in other words, linear time.

Then we must repeatedly step through each time, $t_k$, from $T - 1$ to 1. At each $t_k$, we must (A) insert whatever treasure becomes available at $t_k$ and then (B) access and delete the treasure with the maximum value:

(A) In order to know what treasure becomes available at $t_k$, we must repeatedly step through $L_s$ and perform an insertion if a treasure becomes available at $t_k$. We do this until we come across a treasure that does not become available at $t_k$. If no treasure becomes available at $t_k$, we skip insertion. The run time of each insertion depends on the number of elements in the priority queue. In the best case, we perform no insertions throughout the process. This occurs if all elements become available at $t = T$. In the worst case we, perform $n - 1$ insertions. This occurs if only one treasure becomes available at $T$. The number of elements in the priority queue will vary quite a bit. However, we can confidently say that an upper bound on the process of insertion is $(n-1)*\Theta(log(n))$ since the priority queue will never have more than $n$ elements in it. In actuality, the usual number of elements in the priority queue at any given time will be quite a bit less. We can conclude from these observations that the process of inserting is $O(nlog(n))$ and $\Omega(1)$.

(B) The process of deleting and returning the treasure with the maximum value from the priority queue will happen at most $n$ times. Hence, this process is $O(nlog(n))$.

Putting this all together, we have that the run time is $\Theta(nlog(n))$