Time Spent: 2 hours

Collaborators and Resources:

Problem 2

It looks like in problem 2 we are given a situation in which n tasks need to be completed and each task i takes an expert t_i time to complete. With respect to the problem my interpretation was that since each task requires unique skills, exactly one unique expert is assigned to each individual problem. Since all workers arrive at the same time and only one person works at a time, we have to find the best ordering of tasks (in this context best refers to minimizing total cost to you). Each person gets paid for the amount of time they spent at the house. This value is given to be k $\times$ minutes where k is a constant. So they only get paid for the time they spent in total. The cost does not change based on the type of job itself or who the laborer for that job is. Basically we want to get as many jobs done as fast as possible, that is to get as many laborers out as soon as we can. So the focus is on ordering the tasks so that $t_1 < t_2 < t_3 < \ldots < t_{n-1} < t_n$. That way, the cumulative amount of time over all workers is minimized. And thus the total cost is as well. My algorithm, which is based on the Earliest Deadline First algorithm referenced in K-T Section 4.2, demonstrates a greedy implementation ...

1. We want to order the tasks t in increasing order of their times t_i. That is, from least to greatest.
2. Given an array A of tasks {1, 2, ..., n}
   (a) Let i = A[0] # i stores the index of the value we look at
   (b) Let j = A[0] # j stores our progress through the array
   (c) While j < n-1 {
   # The while loop finds the least value in the remaining array
      i. For k in i : n-1

A. If $t_{A[i]} > t_{A[k]}$, then set i = k
that is, check if the task we're looking at takes the least time
ii. Swap A[j] with A[i]
iii. j++ } # So now we've sorted the first j tasks in the array

3. Return the array A

Claim 1. The algorithm is relatively efficient in $O(n^2)$ worst-case time and determines the order of jobs such that the total cost is minimized

Proof. The reason that the algorithm runs in $O(n^2)$ time is that it effectively has a for loop nested in a while loop. The for loop within the while loop essentially is going to iterate through the entirety of the currently unsorted portion of the array and do a series of linear operations (swapping and incrementing the index j (which marks the portion of A that has already been sorted)) within the for loop. Since the array is originally of size n, we can give this for loop a generous upper bound of n iterations every time it runs. Now, the while loop itself is going to check if j has not yet reached n - 1 (that is, whether we have sorted everything yet). j increases by 1 each time the greedy algorithm (greedy because it selects the task based on smallest required time out of the whole unsorted portion of the array of tasks) runs. So the while loop is essentially going to run n times, which brings us to the worst-case running time of $O(n^2)$.

Now, the total cost is minimized by this ordering of jobs. We've already established the fact that the algorithm orders all the tasks from least amount of time to most amount of time required, the only determinant of cost.

Let's say we select a base case in which we have two tasks, one which takes 30 minutes and another which takes an hour. If we select a laborer to do the hour-long task first then he will be paid 60 and then the second will be paid 60+30 = 90 because the time is cumulative. Thus the total cost will be 150, whereas if we select the shorter task then we will have to pay 30 to the first laborer and then 90 to the second laborer. Thus, this case demonstrates what we're trying to minimize, which is the sum of each time t that each worker w spends at the house. This sum is dependent entirely on when a worker leaves, and so we want to have as many workers leave as soon as we can.

□