CSE 101 Homework 4 Solutions

Winter 2021

This homework is due on gradescope Friday February 19th at 11:59pm pacific time. Remember to justify your work even if the problem does not explicitly say so. Writing your solutions in LaTeXis recommend though not required.

Question 1 (Dropping Lowest Grades, 35 points)**.** *In a class Ronaldo had to complete n assignments. On the i^{th} assignment he scored a_i points out of b_i possible (with $b_i > 0$). At the end of the quarter his teacher told him that he could drop the scores on* any *k assignments that he liked. In particular, if he kept the grades on a set S of $n-k$ assignments, his final grade would be $\frac{\sum_{i\in S} a_i}{\sum_{i\in S} b_i}$.*

(a) Give an example where dropping the k assignments in which a_i is smallest does not give Ronaldo the best possible grade. [5 points]

(b) Give an example where dropping the k assignments in which a_i/b_i is the smallest does not give Ronaldo the best possible grade. [5 points]

(c) Show that there is a polynomial time algorithm that given a target grade g determines whether or not there is a set of assignments that Ronaldo can drop in order to attain grade g or better.
Hint: Ronaldo's grade is at least g so long as $\sum_{i\in S} a_i - gb_i \geq 0$. [25 points]

Solution 1.

(a) Let $n = 2,\ k = 1$. Suppose $(a_1, b_1) = (1, 5)$ and $(a_2, b_2) = (2, 20)$. Observe that $a_1 \leq a_2$, yet it would be preferred for Ronaldo to drop assignment 2.

(b) Let $n = 3,\ k = 1$. Suppose $(a_1, b_1) = (1, 7)$, $(a_2, b_2) = (20, 100)$ and $(a_3, b_3) = (1, 1)$. Observe that $a_1/b_1 \leq a_2/b_2$, yet dropping assignment 1 results in grade $21/101$ while dropping assignment 2 results in grade $2/8$ and so it would be preferred for Ronaldo to drop assignment 2.

(c) Define $v_i \triangleq a_i - gb_i$. As indicated in the hint, observe as for any non-empty subset S, since $\sum_{i\in S} b_i > 0$, it follows that $\sum_{i\in S} a_i / \sum_{i\in S} b_i \geq g \iff \sum_{i\in S} a_i - gb_i \geq 0$, or equivalently $\sum_{i\in S} v_i \geq 0$. The largest sum of v_is is attained by selecting the set S with the largest values. Thus sort the grades by $v_i \triangleq a_i - gb_i$ and check if the $n-k$ largest v_i sum to ≥ 0. Computing the n values of v_i, sorting them, and evaluating the sum takes $\mathcal{O}(n \log n)$ time.

Question 2 (Job Scheduling, 50 points)**.** *Kiki works as a freelancer. At the start of each day, she is given a list of potential jobs to do for that day. Each job has a time at which it becomes available and an amount of time that it will take to accomplish. She can work on a job at any time after it becomes available and may even split up the time spent working on it (for example doing other jobs in between), however she will only be paid for the job if she completes it by the end of the day. As each job pays the same amount, Kiki's goal is to find a way to complete as many jobs as possible by the end of the day.*

Our goal for this problem is to devise an algorithm that given a schedule of n jobs, their start times and total work times and the time that the business day ends to come up with such an optimal schedule in time polynomial in n.

(a) Show that given a collection C of jobs, that it is possible for Kiki to complete all of the jobs in C if and only if she can do so by always working on the remaining job that was first made available (if there are any that are currently available). Hint: If this strategy doesn't work, show that there is a collection of jobs near the end of the day that there is not time to complete all of. [15 points]

(b) *Let job J be the shortest job that is available far enough ahead of the end of the day that it is possible to complete on time. Show that there is a schedule that completes as many jobs as possible so that J is one of the jobs completed. [20 points]*

(c) *Provide a polynomial time algorithm for coming up with an optimal schedule for Kiki. [15 points]*

Solution 2.

(a) We proceed by contradiction. If this strategy does not complete all jobs then let t be the earliest time so that this strategy has Kiki working continuously from time t onwards. It is clear that during this time Kiki is working continuously on jobs from C assigned at or after time t. On the other hand, there is not enough time from t until the end of the day to complete all of these jobs. Therefore, there is not enough time for any strategy to complete all of these jobs.

(b) Assume we have a valid strategy OS that does not include the shortest job J as one of the jobs completed. Suppose that OS contains some job J' that is assigned after J. Then Kiki can complete J in place of J', using the same time period to work on the job (this is enough time since J is shorter than J'. Otherwise, if all jobs in OS are assigned before J, suppose that Kiki completes these jobs using the strategy from part (a) and assume that J' is the last assigned job. We claim that once again, Kiki can complete J in place of J'. Now this is because either Kiki began work on J' after J was assigned (in which case she can use the time that would have been spent on J' to complete J), or J' was started before J was assigned (in which case other than J', Kiki doesn't work on any job after J is assigned and by assumption there is time to complete J). In either case, replacing J' by J gives another equally large set of jobs that can be completed and now include J. Thus if OS was initially a maximal set of completable jobs, we end up with a maximal set that contains J.

Since the assumption for J is that it will be available far enough ahead of the end of the day, then we can substitute any os_i with job J such that job J's available time is before the start time of os_i, and since job J has the shortest time, we can guarantee that job J won't have any overlap with os_{i+1}. The new strategy will finish the same number of jobs by the end of the day. Thus, we proved that for any strategy that doesn't include the shortest job, by exchanging the shortest job J with any available decision os_i, the number of jobs that the new strategy completes won't get any worse. As a result, there exists a strategy that completes as many jobs as possible such that job J is included.

(c) **Algorithm:**

We have already proved in (b) that there exists a strategy that completes as many jobs as possible such that the shortest job J is included. The idea of this algorithm is, we keep looking for the shortest job in C such that we can finish it by the end of the day. Assume we have a collection of jobs $C = \{C_0, C_1, ..., C_{n-1}\}$ where C_i is the i_{th} job in this collection.

(1) Sort each job in $C = \{C_0, C_1, ..., C_{n-1}\}$ by the amount of time (length) it takes to finish it.

(2) Denote job J as the shortest job in C and $|J|$ as the amount of time to finish J. If there isn't enough time to finish J starting from the available time of J, we remove J from C and start the next iteration, otherwise we will commit this time period $[J.start, J.start + |J|]$ to finish it. After that we will try to remove J and this time period from Kiki's timeline. For each job that is available before time $J.start + |J|$, if its start time is before $J.start$, then push back its available time by $|J|$, otherwise if its available time is after $J.start$ but before $J.start + |J|$, push back its available time to $J.start + |J|$. Finally, remove J from C.

(3) We will repeat (2) and keep track of the number of jobs Kiki can finish until there are no remaining jobs left.

Proof:

For each iteration in the algorithm, the algorithm picks the shortest job then removes the job and the time period used to finish this job from the timeline. We have already proved in (b) that at least one of the optimal strategies that finish as many jobs as possible will include the shortest job J in C. So in other words, for each iteration, the algorithm will always pick the job that is included in the optimal strategy for all the remaining jobs in C. As a result, the algorithm makes the optimal decision during every iteration and will eventually becomes one of the global optimal solutions.

Time Complexity Analysis:

The sorting will take time $\mathcal{O}(n * log(n))$. For each iteration the algorithm will potentially push back $\mathcal{O}(n)$ jobs. Then for n iterations, the time complexity should be $\mathcal{O}(n^2) + \mathcal{O}(n * log(n)) = \mathcal{O}(n^2)$.

Question 3 (Huffman Code Depths, 15 points)**.** *Suppose that we build an optimal Huffman code for an alphabet with given letter frequencies using the algorithm discussed in class. Suppose that there is some letter X which accounts for a p-fraction of the total letter occurrences. Show that the depth of X in the resulting Huffman tree is at most* $O(\log(1/p))$.

Solution 3.

Note that the weight of nodes merged by the Huffman tree algorithm only increase as the algorithm progresses. This means that when X is merged into its parent, the parent has weight at least p. When the parent is merged with another node, that other node must have weight at least p as well, and so the grandparent of X must have weight at least 2p. We apply an induction to prove the upper bound for height of the tree is $2 * log_2(1/p) + 2$. The base case is that for any $1 \geq p \geq 1/2$ the tree has at most height 2. Suppose that this upper bound holds for some $p = k \leq 1$ and we want to show that this upper bound is true when $p = k/2$. Since we know that the depth of the node with weight k is at most $2 * log_2(1/k) + 2$ and the grandparent of the node with weight $k/2$ has weight at least k, we know the depth for node with weight $k/2$ is at most $2 * log_2(1/k) + 2 + 2 = 2 * log_2(1/(k/2)) + 2$. The induction works and the upper bound for the depth is $O(log_2(1/k))$.