Homework 3

Due: Fri, Sep 19, 11.59 PT

1. Suppose you want to drive from USC to Santa Monica. Your gas tank, when full, holds enough gas to drive p miles. Suppose there are n gas stations along the route at distances $d_1 \leq d_2 \leq \ldots \leq d_n$ from USC. Assume that the distance between any neighboring gas stations, and the distance between USC and the first gas station, as well as the distance between the last gas station and Santa Monica, are all at most p miles. Assume you start from USC with the tank full. Your goal is to make as few gas stops as possible along the way. Design an efficient algorithm for determining the minimum number of gas stations you must stop at to drive from USC to Santa Monica. Prove that your algorithm is correct. Analyze the time complexity of your algorithm. (25 points)

   Solution:

   The greedy strategy we adopt is to go as far as possible before stopping for gas. That is, when you are at the i-th gas station, if you have enough gas to go to the (i+1)-th gas station, then skip the i-th gas station. Otherwise, stop at the i-th station and fill up the tank.

   Proof of optimality: The proof is similar to that for the interval scheduling solution we did in the lecture. We first show (using mathematical induction) that our gas station stops are always to the right of (or not to the left of) the corresponding stops in any optimal solution. Using this fact, we can then easily show that our solution is optimal using proof by induction.

   Proof:

   (a) First, we show our gas stations are never 'earlier' than the corresponding gas station in any optimal solution: Let $g_1, g_2, \ldots, g_m$ be the set of gas stations at which our algorithm made us refuel. Let $h_1, h_2, \ldots, h_k$ be an optimal solution. We first prove that for any indices $i \leq m$: $h_i \leq g_i$.

   Base case: Since it is not possible to get to the (g_1+1)-th gas station without stopping, any solution should stop at either g_1 or a gas station before g1, thus $h_1 \leq g_1$.

   Induction hypothesis: Assume that for some $1 \leq c$, the c-th gas stops satisfy $h_c \leq g_c$.

   Inductive step: We want to show that $h_c+1 \leq g_c+1$. Note that it is not possible to get to $g_{c+1}+1$ from g_c without any stop (otherwise our solution would not stop at g_{c+1}). Now, since h_c is no later than g_c (as per IH), it cannot be possible to reach $g_{c+1}+1$ from h_c as well. Thus, any solution should stop next at g_{c+1} or before, i.e., $h_{c+1} \leq g_{c+1}$

From the *stay-ahead* property above, it follows that if our solution makes m stops. Any optimal solution must have its m-th stop at or before g_m, thus, it must make m or more stops, implying that our greedy solution is optimal as well.

The running time is O(n) since we, at most, make O(1) computations at each gas station (update remaining fuel and compare with distance to next station).

2. Suppose you are given two sequences, A and B, of n positive integers each. You are allowed to permute the numbers in A and B (rearrange the numbers within each sequence, but not swap numbers between sequences). After permuting, let a_i be the i-th number in A, and let b_i be the i-th number in B. Given this arrangement, you receive a score of $\prod_{i=1}^{n} a_i^{b_i}$

   Design an efficient algorithm for permuting the numbers so as to maximize the resulting score. Prove its correctness and analyze its running time. (25 points).

   Solution:

   Algorithm: First, we observe that it suffices to fix any order for A and only permute B, since only the relative order matters, i.e., any same permutation can be applied to a given order of A and B without changing the score. Hence, without loss of generality, assume A is sorted in increasing order. Now, we claim that sorting B in increasing order results in a maximum score.
   Consider an optimal solution where B is not sorted in increasing order. Then there exists an *inversion*, i.e., a pair of indices i and j where $b_i > b_j$ but $i < j$. Since A is sorted, we know $a_i \leq a_j$. Then, the ratio between the score achieved when bi and bj are swapped versus the current choice of B is

   $$(a_i \text{ ^ } b_j \times a_j \text{ ^ } b_i) / (a_i \text{ ^ } b_i \times a_j \text{ ^ } b_j) = (a_j / a_i) \text{ ^ } (b_i - b_j) \geq 1$$

   The final inequality is due to the fact that $a_i \leq a_j$, so $(a_j / a_i) \geq 1$, and $b_i > b_j$, so $b_i - b_j > 0$ (a number that is ≥ 1 and taken to a positive power is ≥ 1). When b_i and b_j are swapped, the score is multiplied by this ratio, and thus, can only increase. By iteratively removing all inversions this way, we obtain the solution where B is sorted in increasing order - this must be as good as the initial optimal solution since removing the inversion at each step can only increase the score.

   Runtime Complexity: O(n log n) for sorting A and B.

3. You have n heavy iron chains of weights W_1, W_2, …, W_n respectively. You want to connect them together and form one long chain. You decide that everyday you will connect two existing chains into one (so that you have one less chain than the previous day), and repeat this for n-1 days. To connect two chains of weights W and W' into one chain of weight $W + W'$, you need to first carry the chains to the welding station and then bring back the bigger chain to the pile of chains - all this heavy lifting makes you burn 100 * ($W + W'$) calories. You want to finish the whole chain-connecting task (two at a time as described above) while exerting the least effort, i.e., burning the fewest calories in

total. Design an efficient algorithm to decide the order in which to connect all the chains and determine the minimum calories burnt as a result. Analyze the time complexity of your algorithm. You do not need to prove that your algorithm is correct. For full credit, your algorithm should run asymptotically faster than $\boldsymbol{\Theta}(n^2)$. (20 points)

Example: Suppose you have 3 chains with weights 2, 5, 6. Combining 5 and 2 first leads to 700 calories burnt, leaving us with chains of weight 7 and 6. Combining these two leads to 1300 more calories, 2000 in total. This turns out to be the lowest possible among all possible orders.

Solution:

The solution is to connect the least-weight chains available at every point. The intuition is that combining any chains early on, causes them to (indirectly) contribute to the effort more number of times in general, so we save the heaviest for last as much as possible.

Algorithm: Enter all $W_1, W_2, \ldots, W_n$ into one min-heap. Iteratively: Pop the 2 lightest chains, say W and W′ (and connect them). Add 100 * (W + W′) to the calories burnt. Then insert the new chain's weight W + W′ back into the heap. Continue until you are left with only 1 chain.

Runtime Analysis:
Building initial min-heap → O(n)
Removing the 2 minimum elements at each iteration → O(log n)
Inserting a new element in each iteration after connecting two ropes → O(log n)
Total runtime after repeating the above n-1 times → O(n log n)

Rubric:
Choosing the minimum length as greedy selection criteria (5 points)
The iterative process of moving back the connected ropes and updating the data structure (5 points)
Using heap (or any data structure) for efficiency and design algorithm with O(n log n) (5 points)
Correct runtime analysis (5 points)

Ungraded Problems:

1. You plan to have an awesome long weekend. Suppose that there are n activities with where each activity has degree of enjoyment $e_1, e_2, \ldots, e_n$ and it takes you t_k time to complete the k -th activity. If you finish an activity you will receive its full degree of enjoyment, and if you give up on an activity without finishing it you will still get enjoyment from it proportional to the time you spent on it. Assuming you have T time during your long weekend, find a way to maximize your enjoyment. (10 points)
   (a) Design a greedy algorithm to decide on the optimal set of activities to complete. (4 points)
   (b) Prove the correctness of your algorithm. (4 points)

(c) Justify the time complexity of your algorithm (2 points)

Solution:

(a) Calculate $\frac{e_i}{t_i}$ for each $1 \leq i \leq n$ and sort the activity in descending order of $\frac{e_i}{t_i}$. Let the sorted order of activities denoted by $s(1), s(2), \ldots, s(n)$. Do activities in this order until $s(j)$ such that $\sum_{k=1}^{j} t_{s(k)} \leq T$ and $\sum_{k=1}^{j+1} t_{s(k)} > T$, then stop, otherwise partially do the activity $s(j+1)$ in the time remaining.

(b) Correctness proof:
Greedy-choice property: Suppose some optimal solution doesn't pick the activity with the highest $\frac{e_i}{t_i}$ first. Then, swapping in the higher-density activity earlier yields a better or equal score. So an optimal solution always starts with the best density.
Optimal substructure: After choosing the best available activity, the remaining subproblem is of the same form — maximize score with remaining time. The greedy strategy applies recursively.
Thus, the greedy algorithm gives an optimal solution.

(c) The algorithm's time complexity is dominated by the sorting step, which is $O(n \log log\, n)$

Rubrics:

- Part a:
  - 3 points for sorting based on $\frac{e_i}{t_i}$ and picking on that order
  - 1 point for mentioning partial step
- Part b: 4 points for showing that the greedy solution is as good or better than any optimal solution
- Part c: 2 points for the correct time complexity

2. Given a positive integer n we would like to find non-negative integers x, y, z, w such that $n = 15x + 10y + 5z + w$ and $x + y + z + w$ is minimum possible. (10 points)

   (a) Show that in an optimal solution where $x + y + z + w$ is minimum possible, we must have $w < 5$, $z < 2$, and $y < 3$. (4 points)

   (b) Give an efficient algorithm that finds an optimal solution, and justify its running time (i.e., the number of arithmetic operations (additions, multiplications, and divisions)). (6 points)

Solution:
Part a:
If in the optimal solution (x_0, y_0, z_0, w_0), we have $w_0 \geq 5$, then consider the following assignment $(x_1, y_1, z_1, w_1) = (x_0, y_0, z_0 + 1, w_0 - 5)$. In this assignment, we have

$15x_1 + 10y_1 + 5z_1 + w_1 = 15x_0 + 10y_0 + 5(z_0 + 1) + w_0 - 5 = 15x_0 + 10y_0 + 5z_0 + w_0 = n$, $x_1 + y_1 + z_1 + w_1 = x_0 + y_0 + z_0 + w_0 - 4 < x_0 + y_0 + z_0 + w_0$, which contradicts with the optimality of (x_0, y_0, z_0, w_0).
Similarly, if $z_0 \geq 2$, consider $(x_2, y_2, z_2, w_2) = (x_0, y_0 + 1, z_0 - 2, w_0)$.
If $y_0 \leq 3$, consider $(x_3, y_3, z_3, w_3) = (x_0 - 3, y_0 + 2, z_0, w_0)$.

Part b:
Let (x^*, y^*, z^*, w^*) be the optimal solution. From the above argument, we know that $y^* \in \{0, 1, 2\} \triangleq Y$, $z^* \in \{0, 1\} \triangleq Z$, and $w^* \in \{0, 1, 2, 3, 4\} \triangleq W$. Our algorithm is as follows.

- Let $OPT = n$.
- For all $(y, z, w) \in Y \times Z \times W$,
  - Let $x \leftarrow n - 10y - 5z - w$
  - If $x \geq 0$ and $x \bmod 15 = 0$ and $\frac{x}{15} + y + z + w \leq OPT$
    - Set $(x^*, y^*, z^*, w^*) = (\frac{x}{15}, y, z, w)$ and $OPT = \frac{x}{15} + y + z + w$
- Return (x^*, y^*, z^*, w^*) and OPT.

The optimality is proven by the argument in the first question as optimal solution (x_0, y_0, z_0, w_0) guarantees that $(y_0, z_0, w_0) \in Y \times Z \times W$. The runtime is constant as the size of $Y \times Z \times W$ is constant and the number of operations per $(y, z, w) \in Y \times Z \times W$ is also constant.

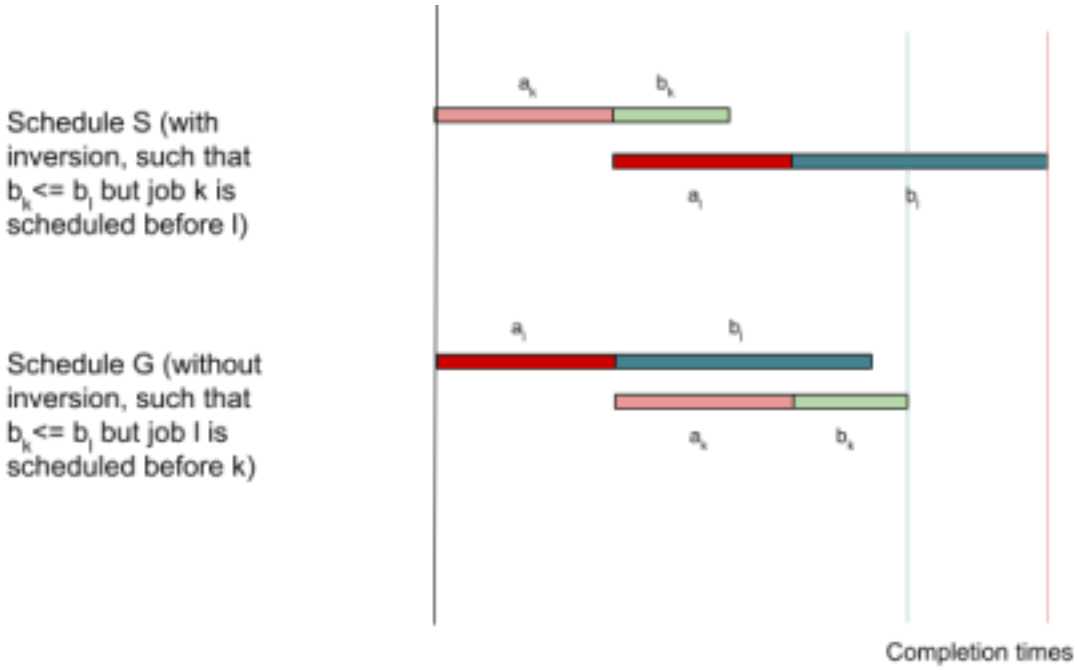
Rubrics:

- Part a: 4 points for finding the correct bounds
- Part b: 6 points for the correct algorithm.

3. There are N tasks that need to be completed by 2 computers A and B. Each task "i" has 2 parts that take time: a_i (first part) and b_i (second part) to be completed. The first part must be completed before starting the second part. Computer A does the first part of all the tasks while computer B does the second part of all the tasks. Computer A can only do one task at a time, while computer B can do any amount of tasks at the same time. Find an O(nlogn) algorithm that minimizes the time to complete all the tasks, and give a proof of why the solution is optimal. (15 points)

   Sort the tasks in decreasing order of bi. Perform the tasks in that order. Basically computer A does the first parts in that order, and computer B starts every second part after computer A finishes the first part.
   Proof: We show that given solution G is actually the optimal schedule, using an exchange argument. We define an inversion to be a pair of jobs whose order in the schedule does not agree with the order of their finishing times, i.e. job k and l form an inversion if $b_k <= b_l$ but job k is scheduled before job l. We will show that for any

given optimal schedule S≠G, we can repeatedly swap adjacent jobs with inversion between them so as to convert S into G without increasing the completion time.



1. Consider any optimal schedule S, and suppose it does not use the order of G. Then this schedule must contain an inversion, i.e. two jobs J_k and J_l so that J_l runs directly after J_k but the finishing time for the first job is less than the finishing time for the second one, i.e. $b_k <= b_l$. We can remove this inversion without affecting the optimality of the solution. Let S' be the schedule S where we swap only the order of J_k and J_l. It is clear that the finishing times for all jobs except J_k and J_l do not change. The job J_l now schedules earlier, thus this job will finish earlier than in the original schedule. The job J_k schedules later, but computer A hands off $J_k$to computer B in the new schedule S' at the same time as it would handed off J_l in the original schedule S. Since the finishing time for $J_k$is less than the finishing time for J_l, the job J_k
   will finish earlier in the new schedule than $J_l$would finish in the original one. Hence our swapped schedule does not have a greater completion time.
2. Since we know that removing inversions will not affect the completion time negatively, if we are given an optimal solution that has any inversions in it, we can remove these inversions one by one without affecting the optimality of the solution. When there are no more inversions, this solution will be the same as ours. Therefore the completion time for G is not greater than the completion time for any other optimal schedule S. Thus G is optimal.

7 points for the correct algorithm.

-3 for not mentioning sorting in descending order of B.

-2 for not mentioning Computer A follows the sorted order.

Proof rubrics: 8 points for the correct proof.

-2 points for not defining what the inversions in this case are -2 points for not generalizing the proof

-1 points for not showing that the algorithm is now the same as our greedy algorithm

4. The United States Commission of Southern California Universities (USCSCU) is researching the impact of class rank on student performance. For this research, they want to find a list of students ordered by GPA containing every student in California. However, each school only has an ordered list of its own students by GPA and the commission needs an algorithm to combine all the lists. There are a few hundred colleges of interest, but each college can have thousands of students, and the USCSCU is terribly underfunded so the only computer they have on hand is an old vacuum tube computer that can do about a thousand operations per second. They are also on a deadline to produce a report so every second counts. Find the fastest algorithm for yielding the combined list and give its runtime in terms of the total number of students (m) and the number of colleges (n).(15 points)

Use a min heap H of size n.
Insert the first elements of each sorted array into the heap. The objects entered into the heap will consist of the pair (GPA, college ID) with GPA as the key value.
Set pointers into all n arrays to the second element of the array CP(j) = 2 for j=1 to n
Loop over all students (i = 1 to m)
    S = Extract_min(H)
    CombinedSort (i) = S.GPA
    j = S.college_ID
    Insert element at CP(j) from college j into the heap
    Increment CP(j)
End loop

Rubrics:
Build min heap (5 points)
Extract the min element (2 points)
Keep pointer to insert the next element in array for extracted element. (3 points)
Recursively/ In Loop do it for all the students (2 points)
Runtime complexity - O(m log n) (3 points)

5. The array A below holds a max-heap. What will be the order of elements in array A after a new entry with value 19 is inserted into this heap? Show all your work.
   A = {26, 14, 10, 8, 11, 9, 3, 2, 4, 1} (10 points)

Solution:

Initial Array

Index	1	2	3	4	5	6	7	8	9	10

Element	26	14	10	8	11	9	3	2	4	1

Array after inserting 19

Index	1	2	3	4	5	6	7	8	9	10	11
Element	26	14	10	8	11	9	3	2	4	1	19

19 is greater than 7(the element at index 11/2=5), so swap

Index	1	2	3	4	5	6	7	8	9	10	11
Element	26	14	10	8	19	9	3	2	4	1	11

19 is greater than 14(the element at index 5/2=2), so swap

Index	1	2	3	4	5	6	7	8	9	10	11
Element	26	19	10	8	14	9	3	2	4	1	11

19 is not greater than 26(the element at index 2/2=1), so no swap

Final Array = {26,19,10,8,14,9,3,2,4,1,11}
Rubrics: 3 points for each pass, 1 point for final answer.