CSCI 570 - Fall 2021 - HW 3

Due February 2, 2022

1. You have N ropes each with length L1, L2, . . . , LN, and we want to connect the ropes into one rope. Each time, we can connect 2 ropes, and the cost is the sum of the lengths of the 2 ropes. Develop an algorithm such that we minimize the cost of connecting all the ropes. No proof is required.(10 points)

   Enter all rope segments into a min-heap with the length of the rope segment being its key value and pop the 2 shortest ropes each time and connect them. Then insert the new resulting rope segment (with key value being the sum of the lengths of the ropes that are connected together) back into the heap. Continue until you are left with only 1 rope segment in the heap.

   10 points for the correct algorithm.

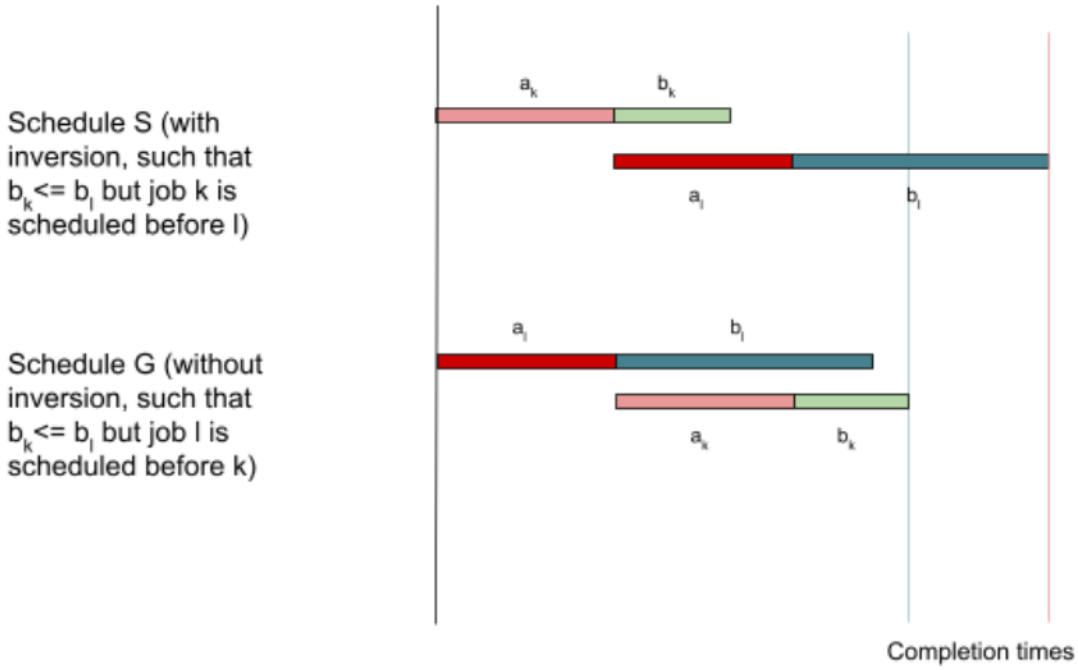
   -1 point if it doesn't clarify that each step involves picking TWO shortest ropes

   -2 points if it's not clearly mentioned that the resultant rope length after joining goes back into the set of candidates.

2. 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_1 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

3. Suppose you were to drive from USC to Santa Monica along I-10. Your gas tank, when full, holds enough gas to go p miles, and you have a map that contains the information on the distances between gas stations along the route. Let $d_1 < d_2 < .... < d_n$ be the locations of all the gas stations along the route where d_i is the distance from USC to the gas station. We assume that the distance between neighboring gas stations is at most $p$miles. Your goal is to make as few gas stops as possible along the way. Give the most efficient algorithm to determine at which gas stations you should stop and prove that your strategy yields an optimal solution. Give the time complexity of your algorithm as a function of n. (15 points)

Solution.

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

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

1. Our gas stations are always to the right (or not to the left of) 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 < 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 g_1, thus $h_1 \leq g_1$.
   Induction hypothesis: Assume that From the greedy strategy taken by our algorithm, $h_c \leq g_c$.
   Inductive step: We want to show that $h_{c+1} \leq g_{c+1}$. It follows from the same reasoning as above. If we start from h_c, we first get to g_c (IH) and , when leaving g_c, we now have at least as much fuel as we did if we had refilled at g_c. Since it is not possible to get to g_{c+1} without any stopping, any solution should stop at either g_{c+1} or a gas station before g_{c+1}, thus $h_{c+1} \leq g_{c+1}$.

2. Now assume that our solution requires m gas stations and the optimal solution requires fewer gas stations. We now look at our last gas station. The reason we needed this gas station in our solution was that there is a point on I-10 after this gas station that cannot be reached with the amount of gas when we left gas station m-1. Therefore we would not have enough gas if we left gas station m-1 in any optimal solution. Therefore, any optimal solution also would require another gas station.

The running time is O(n) since we at most make one computation/decision at each gas station.

Rubrics:

- Greedy algorithm 6 pts
- Proof 10 pts
  - Induction base: 3 pts
  - Induction hypothesis: 3pts
  - Induction step: 3 pts

4. (a) Consider the problem of making change for n cents using the fewest number of coins. Describe a greedy algorithm to make change consisting of quarters(25 cents), dimes(10 cents), nickels(5 cents) and pennies(1 cents). Prove that your algorithm yields an optimal solution. (Hints: consider how many pennies, nickels, dimes and dime plus nickels are taken by an optimal solution at most.) (10 pts)
   (b) For the previous problem, give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Assume that each coin's value is an integer. Your set should include a penny so that there is a solution for every value of n. (5 pts)

Solution.
(a) Denote the coins values as c1 = 1, c2 = 5, c3 = 10, c4 = 25.
1) if n = 0, do nothing but return.
2) Otherwise, find the largest coin c_k, $1 \le k \le 4$, such that $c_k \le n$. Add

the coin into the solution coin set S.
3) Subtract x from n, and repeat the steps 1) and 2) for n − c_k.

For the proof of optimality, we first prove the following claim: Any optimal solution must take the largest ck, such that $c_k \leq n$. Here we have the following observations for an optimal solution:
1. Must have at most 2 dimes; otherwise we can replace 3 dimes with quarter and nickel.
2. If 2 dimes, no nickels; otherwise we can replace 2 dimes and 1 nickel with a quarter.
3. At most 1 nickel; otherwise we can replace 2 nickels with a dime.
4. At most 4 pennies; otherwise can replace 5 pennies with a nickel.
Correspondingly, an optimal solution must have
• Total value of pennies: ≤ 4 cents.
• Total value of pennies and nickels: ≤ 4 + 5 = 9 cents.
• Total value of pennies, nickels and dimes: ≤ 2×10+4 = 24 cents.
Therefore,
• If 1 ≤ n < 5, the optimal solution must take a penny.
• If 5 ≤ n < 10, the optimal solution must take a nickel; otherwise, the total value of pennies exceeds 4 cents.
• If 10 ≤ n < 25, the optimal solution must take a dime; otherwise, the total value of pennies and nickels exceeds 9 cents.
• If n ≥ 25, the optimal solution must take a quarter; otherwise, the total value of pennies, nickels and dimes exceeds 24 cents.
Compared with the greedy algorithm and the optimal algorithm, since both algorithms take the largest value coin c_k from n cents, then the problem reduces to the coin changing of n - c_k cents, which, by induction, is optimally solved by the greedy algorithm.

Rubrics:
- Algorithm: 4 pts
- Claim and proof: 3 pts
- Final proof with induction: 3 pts

(b) Coin combinations = {1, 15, 20} cents coins.Consider this example n = 30 cents. According to the greedy algorithm, we need 11 coins: 30 = 1×20+10×1; but the optimal solution is 2 coins 30 = 2×15.

5. Suppose you are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the i-th element of set A, and let b_i be the i-th element of set B. You then receive a payoff on $\prod_{i=0}^{n} a_i^{b_i}$. Give an algorithm that will maximize your payoff (6 points). Prove that your algorithm maximizes the payoff(10 points) and state its running time (4 points).

Algorithm(6 points):

Let the set A be sorted such that $a_1 < a_2 < a_3 < \ldots\ldots a_m \ldots < a_n$ and B also be sorted such that $b_1 < b_2 < b_3 < \ldots\ldots b_m \ldots\ldots < b_n$

The payoff $\prod_{i=1}^{n} a_i^{b_i}$ would be maximum when a_i *and* b_i are sorted in the same order and then paired together.

Proof by Contradiction(10 points):
Let's assume that the optimal payoff is not obtained by the above sorted solution. Let *R* be the optimal solution, and let m be the highest index where $a_m^{b_m}$ does not appear in R and where a_m is paired with b_r and a_s is paired with b_m.
We know that $a_m > a_s$ *and* $b_m > b_r$

Consider another solution R_1 where a_m is paired with b_m and a_s is paired with b_r and all other pairs are same as in R

$$\frac{Payoff(R)}{Payoff(R_1)} = \frac{\prod_{R} a_i^{b_i}}{\prod_{R_1} a_i^{b_i}} = \frac{a_m^{b_r} a_s^{b_m}}{a_m^{b_m} a_s^{b_r}}$$

$$= \left(\frac{a_m}{a_s}\right)^{b_r - b_m}$$

Since $a_m \geq a_s$ and $b_m \geq b_r$

$$\frac{Payoff(R)}{Payoff(R_1)} \leq 1$$

This contradicts our assumption that R is the optimal solution.
Hence a_m should be paired with b_m.And the same argument holds true for all the other elements in A and B.

Therefore, the sets A and B sorted in the same order gives the maximum Payoff

Runtime Complexity:

- If the two sets A and B are already sorted then the Runtime complexity is O(n).
- If the sets are not sorted, then sort them first and then traverse so the Runtime complexity is O(n log n).

6. 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)
endloop

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(mlog n) (3 points)

SOLUTION 2: Divide & Conquer works too:

https://leetcode.com/problems/merge-k-sorted-lists/solution/#approach-5-merge-with-divide-and-conquer

7. 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 = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1} (10 points)

Initial Array

Index	1	2	3	4	5	6	7	8	9	10
Element	16	14	10	8	7	9	3	2	4	1

Array after inserting 19

Index	1	2	3	4	5	6	7	8	9	10	11
Element	16	14	10	8	7	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	16	14	10	8	19	9	3	2	4	1	7

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	16	19	10	8	14	9	3	2	4	1	7

19 is greater than 16(the element at index 2/2=1), so swap

Index	1	2	3	4	5	6	7	8	9	10	11
Element	19	16	10	8	14	9	3	2	4	1	7

Final Array = {19,16,10,8,14,9,3,2,4,1,7}

Rubrics: 2 points for each pass