CSCI 570 - Summer 2023 - HW 2 Solutions + Rubrics

Graded Problems

1. We have N ropes having lengths $L_1, L_2, \ldots, L_N$. We can connect two ropes at a time: Connecting ropes of length L and L' gives a single rope of length $L + L'$ and doing so has a cost of $L + L'$. We want to repeatedly perform such connections to finally obtain one single rope from the given N ropes. Develop an algorithm to do so, while minimizing the total cost of connecting. 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.

   Rubric:

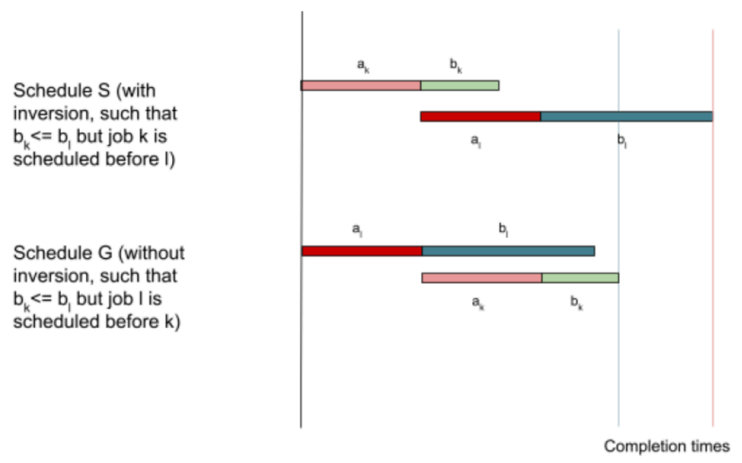
   (a) 10 points for the correct algorithm.

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

   (c) -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 using 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 $O(n \log n)$ algorithm that minimizes the time to complete all the tasks, and give a proof of why the solution obtained by the algorithm is optimal. (15 points)

   Sort the tasks in decreasing order of b_i. Perform the tasks in that order. Basically computer A does the first parts in that order, and computer B


Schedule S (with inversion, such that b_k<= b_l but job k is scheduled before l)
Schedule G (without inversion, such that b_k<= b_l but job l is scheduled before k)
a_k
b_k
a_l
b_l
Completion times

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 $b_k <= b_1$ but job k is scheduled before job 1. We will show that for any given optimal schedule $S \neq G$, 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_1 so that J_1 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. $b_k <= b_1$. We can remove this inversion without affecting the optimality of the solution. Let S' be the schedule obtained from S where we swap only the order of J_k and J_1. It is clear that the finishing times for all jobs except J_k and J_1 do not change. The job J_1 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_1 in the original schedule S. Since the finishing time for J_k is less than the finishing time for J_1, 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.

Rubric:

7 points for the correct algorithm.

(a) -3 for not mentioning sorting in descending order of B.
(b) -2 for not mentioning Computer A follows the sorted order.

Proof rubrics: 8 points for the correct proof.

(a) -2 points for not defining what the inversions in this case are
(b) -2 points for not generalizing the proof
(c) -1 points for not showing that the algorithm is now the same as our greedy algorithm

3. Suppose you want to drive from USC to Santa Monica. Your gas tank, when full, holds enough gas to go 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. Give the most efficient algorithm to determine which gas stations you should stop at and prove that your algorithm yields an optimal solution (i.e., the minimum number of gas stops). Give the time complexity of your algorithm as a function of n. (15 points)

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 $(i+1)^{th}$ 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.

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 $i < m$, $h_i \leq g_i$

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 for 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 $h_{c+1} \leq g_{c+1}$

b 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
- "Stay-ahead argument" of the proof (part a): 8 pts
  - Induction base: 2 pts
  - Induction hypothesis: 3pts
  - Induction step: 3 pts
- Completing the argument of optimality in the proof (part b): 2 points

4. Suppose you are given two sets A and B, each containing n positive integers. You can choose to order the numbers in each set however you like. After you order them, let a_i be the i^{th} number in set A, and let b_i be the i^{th} element of set B. You then receive a payoff of $\prod_{i=0}^{n} a_i^{b_i}$. Give an algorithm to decide the ordering of the numbers so as to maximize your resultant payoff (6 points). Prove that your algorithm maximizes the payoff (10 points) and state its running time (2 points).

Algorithm(6 points):

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

The payoff $\prod_{i=0}^{n} a_i^{b_i}$ 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^{bm} does not appear in R (i.e., $a_{m+1}^{b_{m+1}}$ . . . $a_n^{b_n}$ all do appear). Let a_m be paired with some b_r and some as be 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 while all other pairs are same as in R. Then,

$$\begin{aligned}
\frac{\text{Payoff}(R)}{\text{Payoff}(R_1)} &= \frac{a_m^{b_r} a_s^{b_m}}{a_m^{b_m} a_s^{b_r}} && \text{(since the other pairs cancel out)} \\
&= \left(\frac{a_m}{a_s}\right)^{b_r - b_m} \\
&< 1 && \text{(Since } a_m/a_s > 1\text{, and } b_m - b_r < 0\text{)}
\end{aligned}$$

Thus R_1 has a higher payoff than R, which is a contradiction to R is the optimal solution. Thus, the optimal solution cannot have a mismatch: sets A and B must be sorted in the same order to give the maximum Payoff.

Runtime Complexity (2 points): O(n log n) for sorting each of the sets.

5. 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. Find the fastest algorithm for yielding the combined list and give its runtime in terms of m, the total number of students across all colleges, and n, the number of colleges. (12 points)

Suppose the combined sorted list is L. We use a min heap H of size n. The algorithm (psudocode) is as follows:

```
Insert the first elements of each sorted array into the heap H.

Let $CP(j)$ be a 'pointer' for $j^th$ that denotes the next element to consider.
Set it to 2 for all the arrays (the second element), as the first elements are
already put in the heap.

Loop (i = 1 to m)          // adding one student to L in each iteration
  S = Extractmin(H)
  L(i) = S
  j = S.CollegeID
  Insert element at $CP(j)$ from $j^{th}$ array into H and Increment $CP(j)$
endloop
```

The runtime complexity is O(mlog n).

Rubrics:

- Build min heap (5 points)
- Extract the min element (2 points)
- Insert the next element in array for extracted element. (3 points)
- Recursively/ In Loop do it for all the students (2 points)
- The runtime complexity (3 points)

SOLUTION 2: Divide & Conquer works too Refer to this link

6. Design a data structure that has the following properties (assume n elements in the data structure, and that the data structure properties need to be preserved at the end of each operation):

   - Finding the median of all the n elements takes $O(1)$ time.
   - Inserting the element takes $O(\log n)$ time.

   Describe your data structure. Give the algorithms for Find-Median() and Insert() functions. (12 points)

   We use the $\lceil \frac{n}{2} \rceil$ smallest elements to build a max-heap and use the remaining $\lfloor \frac{n}{2} \rfloor$ elements to build a min-heap. The median will be at the root of the max-heap and hence accessible in time O(1) (For simplicity, in case of even n, assume the median to be the $n/2^{th}$ element. For the actual definition of the median, taking the average of the middle two elements for even n, only a couple of tiny details need to be adjusted, which we leave as exercise).

   Insert() algorithm: For a new element x

   - Initialize len of maxheap (maxlen) and len of minheap (minlen) to 0.
   - Compare x to the current root of max-heap.
   - If x < median, we insert x into the max-heap.Maintain length of max-heap say maxlen, and every time you insert element into maxheap increase maxlen by 1. Otherwise, we insert x into the min-heap, and increase length of minheap say minlen by 1. This takes O(log n) time in the worst case.
   - If size $(maxHeap) > size(minHeap)$+1, then we call Extract-Max() on maxheap, and decrease maxlen by 1 of and insert the extracted value into the min-heap and increase minlen by 1. This takes O(log n) time in the worst case.
   - Also, if size $(minHeap) > size(maxHeap)$, we call Extract-Min() on min-heap, decrease minlen by 1 and insert the extracted value into the maxheap and increase maxlen by 1. This takes O(log n) time in the worst case.

Find-Median() algorithm:

- If (maxlen + minlen) is even: return (sum of roots of max heap and min heap)/2 as median
- Else if (maxlen > minlen): return root of max heap as median
- Else: return root of min heap as median

Rubric

- 3 pt: Using max heap and min heap to store first half and second half of elements.
- 3 pt: Comparing element to the root and proper if conditions for inserting in appropriate heap.
- 3 pt: Proper if conditions for returning the median.
- 3 pt: Correct Time Complexity

7. Given a connected graph $G = (V, E)$ with positive edge weights. Let s and t be two given nodes for shortest path computation, prove or disprove with explanations (5 points each):

   (a) If all edge weights are unique, then there is a single shortest path between any two nodes in V. False. Counter example: (s, a) with weight 1, (a, t) with weight 2 and (s, t) with weight 3. There are two shortest path from s to t though the edge weights are unique. Rubric.

   - 2 pt: Correct T/F claim
   - 3 pt: Provides a correct counterexample as explanation

   (b) If each edge's weight is increased by k, the shortest path cost between s and t will increase by a multiple of k. False. Counter example: suppose the shortest path s $\rightarrow$ t consist of two edges, each with cost 1, and there is also an edge e = (s, t) in G with cost(e)=3. If now we increase the cost of each edge by 2, e will become the shortest path (with the total cost of 5).

   - 2 pt: Correct T/F claim
   - 3 pt: Provides a correct counterexample as explanation

   (c) If the weight of some edge e decreases by k, then the shortest path cost between s and t will decrease by at most k. False. Under the assumption that the new weight of the edge is not negative after decreasing, this statement is in fact true. This is because for any two nodes s, t, suppose that $P_i, ..., P_k$ are all the paths from s to t. If e whose cost decreased, belongs to some P_i then P_i's path cost decreases by k, otherwise, it is unchanged. Hence all paths from s to t will decrease by at most k, and in turn, the shortest path cost will decrease by at most k. However, since we are not given this assumption, the following case results in a counter-example. If there is a cycle C in the graph with the edge e (whose cost was decreased) on it, and if there is a path from s to t that goes through that cycle, and if the cost was decreased so much that the C is now a negative cycle, then a path containing C can loop through the cycle infinitely many times, making the shortest path cost $-\infty$, and thus not decreasing "by at most k" as claimed.

Rubric

- 2 pt: Correct T/F claim
- 3 pt: Justification by considering the negative cycle case

(d) If each edge's weight is replaced by its square, i.e., changed w to w^2, then the shortest path between s and t will be the same as before (though possibly with a different cost.) False. Counter example: Suppose the original graph G composed of $V = \{A, B, C, D\}$ and E : $(A \to B) = 100$, $(A \to C) = 51$, $(B \to D) = 1$, $(C \to D) = 51$, then the shortest path from A to D is $A \to B \to D$ with length 101. After squaring this path length become $100^2 + 1^2 = 10001$. However, $A \to C \to D$ has path length $51^2 + 51^2 = 5202 < 10001$. Thus $A \to C \to D$ becomes the new shortest path from A to D. Rubric

- 2 pt: Correct T/F claim
- 3 pt: Provides a correct counterexample as explanation

8. Consider a directed, weighted graph G where all edge weights are positive. You are allowed to change the weight of any one edge to zero. Propose an efficient method based on Dijkstra's algorithm to find the lowest-cost path from node s to node t, given that you may set one edge weight to zero. Your algorithm must have the same running time complexity as the Dijkstra's algorithm. (15 points)

Use Dijkstra's algorithm to find the shortest paths from s to all other vertices. Reverse all edges and use Dijkstra to find the shortest paths from all vertices to t. Denote the shortest path from u to v by $u \rightsquigarrow v$, and its length by $\delta\ (u,\ v)$. Now, try setting each edge to zero. For each edge $(u,\ v) \in E$, consider the path $s \to u \rightsquigarrow v \to t$. If we set w$(u,\ v)$ to zero, the path length is $\delta(s,\ u) + \delta(v,\ t)$. Find the edge for which this length is minimized and set it to zero; the corresponding path $s \to u \rightsquigarrow v \to t$ is the desired path. The algorithm requires two invocations of Dijkstra, and an additional $O(|E|)$ time to iterate through the edges and find the optimal edge to take for free, which is asymptotically less than Dijkstra. Thus the total running time is the same as that of Dijkstra. (For time complexity analysis, any implementation for Dijkstra's, such as with Binary heap or with Fibonacci heap works just fine.)

Rubric

For the algorithm:

- 12 pt if the approach is of the same complexity as Dijkstra's algorithm
- 4 pt if the approach is more efficient than naive but not has the same complexity as Dijkstra's algorithm
- 0 pt if the naive approach or any other approach with larger complexity than naive is proposed
- 3 pt for time complexity analysis.

Ungraded Problems

9. (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.)

   (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.

   (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 ck , 1 ≤ k ≤ 4, such that $c_k \leq n$. 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 c_k, 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: $\leq 4 + 5 = 9$ cents.
   - Total value of pennies, nickels and dimes: $\leq 2 \times 10 + 4 = 24$ cents.

   Therefore,

   - If $1 \leq n < 5$, the optimal solution must take a penny.
   - If $5 \leq n < 10$, the optimal solution must take a nickel; otherwise, the total value of pennies exceeds 4 cents.
   - If $10 \leq n < 25$, the optimal solution must take a dime; otherwise, the total value of pennies and nickels exceeds 9 cents.
   - If $n \geq 25$, 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.

10. 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. $A = \{16, 14, 10, 8, 7, 9, 3, 2, 4, 1\}$

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	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