CSE 222 (ADA) Model Solution of Mid Semester Examination

March 9, 2022

Duration of Examination : 2 hours.

Problem 1. Solve the following recurrence relation.

$$T(n) = \begin{cases} 3T(n/9) + \sqrt{n} & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

Solution: Observe that $\sqrt{n}$ is $\Theta(\sqrt{n})$. This recurrence relation is solvable by Master's Theorem. Using master's theorem, we see that $T(n) = \Theta(\sqrt{n}\log_9 n)$.

Rubrics: If Master's Theorem is used with proper justification (similar to above), then award 10 marks. No need to deduct any marks if the base of the log is missing in the answer. Also, no need to deduct marks if the answer is $O(\sqrt{n}\log n)$.

If recursion tree is used instead, then also check the justification about work done at each level of the recursion tree. If the justification is correct, then award 10 marks. Penalize 2 marks if there are some minor mistakes on the powers of calculation in case of recursion tree.

Problem 2. Suppose that there are k sorted arrays, $A_1, A_2, \ldots, A_k$ each of size n. Give an order $O(nk\log k)$-time algorithm to count the size of $\{(x,y) \mid x \in A_i, y \in A_j \text{ such that } i < j \text{ but } x > y\}$. Give the pseudocode and argue the runtime. Nothing else is required. But the pseudocode should be clear enough. Use English sentences withing the pseudocode if something is too complicated.

Solution (sketch): Observe that if $k = 1$, then there no such pair (x,y) is possible satisfying $x \in A_i, y \in A_j$ such that $i < j$ but $x > y$. On the other hand, if $k = 2$, then the problem is counting the number of pairs (x,y) such that $x \in A_1, y \in A_2$ such that $x > y$. This is same as counting the number of *split inversions* in two sorted arrays. This can be done in $O(n)$-time as both $|A_1| = |A_2| = n$ as discussed in the class (also available in KT book). This procedure for *Count-Split-Inversion* merges two sorted arrays A_1 and A_2, and correctly outputs the number of pairs (x,y) such that $x \in A_1, y \in A_2$ satisfying $x > y$ along with the merged array that contains the elements of $A_1 \cup A_2$.

Let the initial set of k arrays is $\mathcal{A}$. If $k = 1$, then return $(\mathcal{A}, 0)$ as the output. If $k = 2$, then invoke the algorithm for *Count-Split-Inversion* with $\mathcal{A}_1$ and $\mathcal{A}_2$ that outputs the total number of split inversions. The idea is to break $\mathcal{A}$ into two groups. The first group $\mathcal{A}_L$ has first $\lceil k/2 \rceil$ arrays and the second group $\mathcal{A}_R$ has the rest of $\lfloor k/2 \rfloor$ arrays. Recursively count the number of pairs within $\mathcal{A}_L$ and recursively count the number of pairs within $\mathcal{A}_R$. Each time while counting, output the merged sorted array. Then, finally count the number of split inversions and output the overall merged array. The pseudocde of the algorithm will is described in Algorithm 1.

Algorithm 1 Count-Merged-Inversion((Arrays $\mathcal{A} = [A_1, A_2, \cdots A_k]$), k)

1: **if** $k = 1$ **then**
2: Return $(0, A_1)$;
3: **end if**
4: $\mathcal{A}_L = [A_1, \ldots, A_{\lceil k/2 \rceil}]$; /* first $\lceil k/2 \rceil$ sorted arrays */
5: $\mathcal{A}_R = [A_{\lceil k/2 \rceil + 1}, \ldots, A_k]$; /*last $\lfloor k/2 \rfloor$ sorted arrays*/
6: $\left(b_L, \bigcup_{X \in \mathcal{A}_L} X\right) =$ Count-Merged-Inversion$(\mathcal{A}_L, \lceil k/2 \rceil)$;
7: $\left(b_R, \bigcup_{X \in \mathcal{A}_R} X\right) =$ Count-Merged-Inversion$(\mathcal{A}_R, \lfloor k/2 \rfloor)$;
8: $\left(b_{SP}, \bigcup_{X \in \mathcal{A}} X\right) =$ Count-Split-Inversion$\left(\bigcup_{X \in \mathcal{A}_L} X, \bigcup_{X \in \mathcal{A}_R} X\right)$;
9: Return $\left(b_{SP} + b_L + b_R, \text{the merged sorted array } \bigcup_{X \in \mathcal{A}} X\right)$;

Running time analysis (details not needed for grading, but some justification is needed): Observe that the process of Count-Split-Inversion will always run in $O(|\cup_{X \in \mathcal{A}_L} X| + |\cup_{X \in \mathcal{A}_R} X|)$-time. We have to do the analysis based on how the recursive approach works. Initially, we have k sorted arrays given. They are broken into 2 groups of approximately $k/2$ sorted arrays each. In the next stage of recursion, each of the two groups is broken into two groups of $k/4$ sorted arrays each, and so on. In this process, after $\lceil \log_2 k \rceil$ many recursive calls, every subroutine have only one sorted A_i array. In such case, it just returns that sorted array with $(0, A_i)$ as the number of such pairs. After that, when the subroutine gets there are two different sorted arrays, A_i and A_{i+1}, it invokes the $O(|A_i| + |A_{i+1}|)$-time algorithm that outputs the number of pairs (x, y) such that $x \in A_i, y \in A_{i+1}$ and $x > y$ along with the merged array containing the elements of $A_i \cup A_{i+1}$. In this process, at every stage whenever Count-Split-Inversion is invoked, an output array is sorted along with the total number of inversions outputted. Hence, the number of recursive calls here is $\lceil \log_2 k \rceil$ and the Count-Split-Inversion subroutine (outside recursive call) takes $O(nk)$-time. Hence, the total time taken is $O(nk \log k)$.

Alternative solution sketch (an equivalent interpretation of the above mentioned recursive algorithm): First, count the number split inversions (i.e. number of pairs (x, y) such that $x \in A_1, y \in A_2$ and $x > y$) using idea in the class or KT book between A_1 and A_2. This process outputs merged sorted array $A_{1,2}$. Then, repeat this process with the arrays $(A_3, A_4), (A_5, A_6), \ldots, (A_{k-1}, A_k)$. This process constructs merged sorted array $A_{3,4}, \ldots, A_{k-1,k}$. Then repeat the same process again with $(A_{1,2}, A_{3,4}), \ldots, (A_{k-3,k-2}, A_{k-1,k})$. Observe that in the first round, the algorithm invokes *count-split-inversion* $k/2$ times. Then, in the second round, it again invokes the same subroutine $k/4$ times, and so on. Finally, in the last round, it invokes the subroutine when there are only two sorted arrays. Hence, the number of rounds is $\log_2 k$. At every round, the total running time of all *count-split-inversions* is $O(nk)$. Hence, $O(nk \log k)$ is the total running time.

Rubrics: If the pseudocode is clear enough and consistent either with the Algorithm 1 or the alternative solution sketch, then award 7 marks. 3 marks for correct justification of the running time. Some minor mistake in the pseudocode should be penalized by 1 mark. Some minor mistake(s) at running time justification should be penalized by 2 marks at each case.

Problem 3. Suppose you are at the $n/2$th floor (this was corrected during the exam) of a building with n floors inside an elevator (assuming n even). Your friends are hiding at different floors of

the building (two of your friends can hide in the same floor). Now, let us say a game starts exactly at time $t = 0$. At time $t = i, i = 1, 2, \cdots n-1$, your friend will reveal himself/herself momentarily. If the elevator is on the particular floor where friend i is located at that precise time point, your friend has to get in to the elevator. Otherwise, you have missed him/her. The final goal is to take as many of your friends as possible to floor n exactly at $t = n$. Throughout the game you can take the elevator wherever you want. Assume the elevator moves one floor in 1 unit of time.

Design a polynomial time algorithm to catch as many of your friends as possible and take them to the top floor at $t = n$.

Subproblems. For every $t = 0, 1, 2 \cdots n$ and $i = 0, 1, 2, \cdots n$, define $\mathsf{OPT}(t, i)$ as the optimal solution (that is the maximum number of friends caught) given that the elevator is at floor i at time t and the game stops here. **Rubric: +1 for defining the subproblems correctly with all indices specified with correct range**

Final Answer. $\mathsf{OPT}(n, n)$

Rubric: +1 / 0

Recurrence.

$$\mathsf{OPT}(t, i) = \max\{\mathsf{OPT}(t-1, i-1), \mathsf{OPT}(t-1, i+1), \mathsf{OPT}(t-1, i)\} + \lambda,$$

$$\mathsf{OPT}(0, n/2) = 0$$

$$\mathsf{OPT}(0, i) = -\infty, \forall i \neq n/2$$

where $\lambda = 1$ if $floor(t) = i$, 0 otherwise

Rubric : Total +3. -1 for missing base case. -1 for minor errors (like indices), -2 for major errors

Explanation. (Not required for grading). I am interested in computing $\mathsf{OPT}(t, i)$. There are 3 choices really - either I am at floor $i-1, i+1$ or i at time $t-1$. So we recurse on those subproblems and take the max as usual. Depending on friend t is on floor i or not, we add 1. Note that the boundary conditions $-\infty$ are very important here to encode 'invalid' states, that is $\mathsf{OPT}(t, i) = -\infty$ should indicate that the lift cannot be at floor i at time point t under any feasible set of movements.

Pseudocode. Quite straightforward with a 2-D array. One important thing to note is the outer loop should run for $t = 0, 1, \cdots n$ and the inner loop should compute all $\mathsf{OPT}(i, j)$ for all values of $j = 0, 1, 2, \cdots$

Rubric : Total +3. -1 for wrong indexes in loops. -2 for wrong order of loops. -1 for missing final return

Runtime. Since we are filling up n^2 entries and each entry looks at 3 different values, the runtime is $\mathcal{O}(n^2)$.

Rubric : Total +2. Should be consistent with the pseudocode

Other correct solutions are acceptable. I do not think this problem can be solved using anything else other than a DP. So any random solutions get no credit.

Problem 4. For the following two problems, you need to find *examples which show that the given greedy algorithms* fail to find the optimal solution.

a) Consider the interval coloring problem. You are given a set of n intervals $I_1, I_2, \cdots I_n$ with s_i, e_i being the start and end time of interval $i = 1, 2, \cdots n$. The task is to assign a color (or a number

for simplicity) to each interval so that two overlapping intervals do not get the same color. The objective is to use the minimum possible number of colors.

Suppose the list is sorted according to **non-decreasing end-time**. We maintain a palette of colors. We start by giving color 1 to the first interval and adding it to the palette. At any iteration $1 < i \leq n$, we assign the **lowest index** color admissible from the current palette. If none of the colors in the palette is admissible for interval i, then we add a new color to the palette and assign this color to i. Find an example where this algorithm will not give you the optimal solution.

b) Suppose you are auctioning a set of n indivisible items and you have a set of n buyers. Every buyer $i = 1, 2, \cdots n$ bids a price p_{ij} for every item $j = 1, 2, \cdots n$. Your goal is to sell exactly one item to each buyer so as to maximize the total money you earn. A natural greedy approach is the following - assign item j to buyer i such that p_{ij} is the maximum among all possible pairings i, j. Remove item j and buyer i from the list and continue. Find an example where this does not give you the optimal solution.

Solution: (a) Consider the following example of intervals. $I_1 = (0, 3), I_2 = (2, 5), I_3 = (4, 9)$, and $I_4 = (6, 8)$. If we sort them at the nondecreasing order of 'finishing time', then it is $I_1 < I_2 < I_4 < I_3$. The algorithm will first **blue** color to I_1, then **red** color to I_2. Then, it will assign **blue** color to I_4 again. Observe that I_3 intersects both I_4 (colored blue) as well as I_2 (colored red). Hence, I_3 must be assigned a different color **green**. Hence, 3 colors are used.

But observe that it is possible to assign **blue** to I_1, I_3 and **red** to I_2, I_4.

Hence, the approach construct a *suboptimal* solution.

(b) Consider the following set of 2 buyers and 2 items and bids. $p_{1,1} = 1 + \epsilon, p_{1,2} = 1, p_{2,1} = 1, p_{2.2} = 0$ (ϵ is a very small number). Greedy will first assign item 1 to buyer 1. But then the only choice it has is to assign item 2 to buyer 2. So total profit is $1 + \epsilon$. Clearly, the optimal solution is 2 here.

Rubric. This is pretty much binary. +5 for correct and 0 for wrong.

Problem 5. Describe an implementation of the Interval Partitioning (also known as Interval Coloring) algorithm done in the lecture in $O(n \log_2 n)$-time. More precisely, there are two parts to this algorithm - first you need to do some sorting and then you run a greedy loop. Design your algorithm in a way that the main loop takes only $\mathcal{O}(n)$ time. You can use additional data structures.

(First write the pseudocode for the naive algorithm done in lecture. This will fetch you 20%. Next describe how you modify the naive implementation to achieve the required running time).

Solution Sketch. There are several ways to implement the main loop in $\mathcal{O}(n)$ time. Let $\mathcal{I}$ be a linked list where the ordering is according to the start times - this can be built in $\mathcal{O}(n \log n)$ time. Each node of this linked list will contains the start time and the color assigned to the particular interval.

We need to do another pre-processing. Sort *all* the start times and finish times in to a single array in increasing order - call this array τ. For each entry of τ, maintain an additional field that points to the correct interval in $\mathcal{I}$ depending on whether this entry is either the start or finish time of that interval.

Now, we maintain a set of 'inactive' colors - $\mathcal{C}$ initially empty. As usual, we traverse $\mathcal{I}$ from left to right (intervals are considered according to increasing start times) - let $s_1, s_2, \cdots s_n$ be the start

times. We also use a variable j to traverse τ, initialized to 0.

Here are the main operations at iteration i

1. We increment the variable j from s_{i-1} to s_i in the array τ and at each increment step do the following
2. Fetch the pointer $\tau[j]$ and add the color of the corresponding interval in $\mathcal{I}$ to $\mathcal{C}$ (intuitively, this indicates that the corresponding interval has ended and its color can be used again)
3. Fetch a color from $\mathcal{C}$. If $\mathcal{C}$ is empty, introduce a new color and assign that to interval i by suitably updating the list $\mathcal{I}$

It is easy to check that the total number of times the variable j is incremented in at most $2n$ and for each increment, we do only a constant number of updates in the data structures.

Rubrics. +2 for writing the pseudocode for the vanilla implementation .

+6 for a correct implementation of the inner loop in $\mathcal{O}(n)$ or $\mathcal{O}(n \log n)$ time. Some people have used priority queue/min-heap to fetch the color. Although this is an overkill, it is correct. The important thing is to make sure that in your implementation, you are not assigning a color which is already assigned to an overlapping interval. If this is not clear then only +3 out of 6.

+2 for correctly arguing the runtime consistent with the implementation. However, 0 for this part if the algorithm itself is completely wrong.