CSE 222 (ADA) Homework Assignment 2 (Theory)

Deadline : March 4 (Friday) 10am.

The theory assignment has to be done in a team of at most two members, as already selected by you. The solutions are to be typed either as a word document or latex-ed and uploaded as pdf on GC. We shall strictly not accept solutions written in any other form. Remember that both team members need to upload the HW solution on GC. Collaboration across teams or seeking help from any sources other than the lectures, notes and texts mentioned on the homepage will be considered an act of plagiarism

General instructions:

Each question is worth 15 points. For each question we need you to provide the following:

1. (3 points) A precise description of the subproblems you want to solve in the dynamic program. *Notice this is just the subproblem, not how to solve it, or how it was obtained.*
2. (3 points) A recurrence which relates a subproblem to "smaller" (Whatever you define "smaller" to mean) subproblems. *Notice this is just the recurrence, not the algorithm or why the recurrence is correct.*
3. (1 points) Identify the subproblem that solves the original problem *Often but NOT always, this is the answer of the"largest" subproblem.*
4. (3 points) A clear proof for why the recurrence is correct. Specifically, prove that an optimal solution for your subproblem can be obtained using optimal solutions for "smaller" subproblems via your recurrence. *Notice this is the proof of why the recurrence is correct. Often this involves contradiction arguments involving the optimal solution. This is NOT(!) to prove correctness of the final algorithm.*
5. (3 points) A pseudo-code for a dynamic program which solves the recurrence efficiently *You do not need to prove correctness of the pseudocode itself*
6. (2 points) An argument for the running time of your dynamic program

We are only interested in the *value* of the optimal solution, not the optimal solution itself. So you do not need to give the reconstruction algorithm.

If your solution does not have the structure above, you will be awarded *a zero* (yes, you read that right). There will be *no* concessions on this.

The first question below is from the text by Jeff Erickson. You can find an online copy here.

Problem 1. Remember Mr. Monkey from Skull Island ? Well, he is back to King Kong's garden, but this time he stands in front of a row of magical banana trees that Kong has grown. The bananas give Mr. Monkey an amazing power to jump over a certain number of trees. Eager to relive the jumping days of his youth, Mr. Monkey chalks the following plan. He will start walking from some point, say s which is to the left of the leftmost tree. Once he reaches a tree, he has two choices - either to walk past it or climb it and eat exactly one banana. Eating a banana will magically make him jump a certain distance (even if he doesn't want to !), possibly over a few trees. In case he lands on the ground, he can continue the above exercise from that point. In case the first jump lands him on a tree, he can either climb down and continue the exercise or he can eat the bananas from there and continue jumping.

Let $dist[i]$ be an array that contains the distance of t_i from the starting point s and $jump[i]$ be an array that contains the distance that Mr. Monkey can jump if he eats a banana from t_i. Write an algorithm that determines the maximum distance that Mr. Monkey can **jump**. Note that this does not include the distance he walks or climbs.

Solution: We are going to look at this problem as selecting a set of non-overlapping intervals, as done in the lectures of greedy algorithm. For each tree $t_i, i = 1, 2, 3, \cdots n$, we define an interval where start time $s(i) = dist[i]$ (imagine this is the starting time of interval i with $s = 0$) and finish time $f(i) = s(i) + jump[i]$. Now the problem is essentially picking a set of *non-overlapping* intervals so as to maximize the total length (note that in the problem discussed in class, the objective was to maximize the number of intervals). With this viewpoint, we design the DP.

We assume the intervals sorted by non-decreasing finish times in the following solution. This will take an additional $\mathcal{O}(n \log n)$ time. Assume that an array $f[]$ stores these finish times in this sorted order.

Subproblems. For all $i = 1, 2, \cdots n$, let $opt(i)$ be the set of non-overlapping intervals of maximum possible length considering the intervals $\{1, 2, \cdots i\}$.

Recurrence.

$$opt(i) = \max \begin{cases} opt(i-1) \\ opt(k) + jump[i], \quad \text{k is the largest index } 1 \leq k < i, f(k) \leq s(i) \\ opt(1) = jump[1] \end{cases} \tag{1}$$

Explanation (not required for credit). Its very similar to the ball selection problem. Essentially $opt(i)$ can have two cases - either you do not select interval i or you select it. In the second case, the remaining subproblem would be $opt(k)$ where k is the interval with largest finish time that does not overlap with i.

Pseudocode. Its a simple for-loop which implements the above recurrence. You need a 1-d array to store the values of $opt[i]$ for every $i = 1, 2, \cdots n$. Remember to initialize the base case.

Runtime. There are n iterations of the for-loop. Insider each iteration, you might need to find the suitable k. Done naively, this will take a linear search through a sub-array of size $i - 1$. Hence the overall runtime would be $\mathcal{O}(\sum_{i=1}^{n} i = \mathcal{O}(n^2)$.

But since $f[]$ is sorted in non-decreasing order, this can be done using a simple binary search through $f[1 \cdots i - 1]$. Hence, the overall time complexity would be $\mathcal{O}(n \log n)$ (together with the initial sorting). (it is enough to do $\mathcal{O}(n^2)$ for full credit).

Problem 2. Mr. Monkey is so excited with his last adventure that this time he brings his partner

Miss Chimp so that they can jump together. However, to their utter bewilderment, they find that Kong has removed most of the magical bananas and each tree has just one banana left. Assuming they both start at s, design an algorithm which determines the maximum length that the two of them together can jump.

Solution. The problem is quite similar to the above. However, here, we need to output two different sets of intervals S_1 (for Mr. Monkey) and S_2 (for Ms. Chimp) such that S_1 and S_2 are *disjoint* set of non-overlapping intervals. Very crucially, note that there can be overlaps between an interval $i \in S_1$ and $j \in S_2$. We need to find S_1 and S_2 which minimizes the total length of intervals in these sets.

Again, we assume there is an array $f[]$ which contains the finish times all the intervals in non-decreasing order.

Subproblems. We are going to define the subproblems in a slightly different way compared to Problem 1now. Let $opt(i, j)$ denote the optimal solution to the problem where the last interval in S_1 is i and the last interval in S_2 is j, where $0 \leq i < j \leq n$. Note that its enough to solve subproblems for $i < j$ since - (1) it does not really matter which set belongs to Mr. Monkey and which to Ms. Chimp and (2) $i = j$ is not allowed since they cannot have the same interval. Also $i = 0$ captures the case when S_1 is empty.

Final Solution. The final solution is given by

$$\max_{0 \leq i < j \leq n} opt(i, j)$$

Note that this is crucial given the way we have defined the subproblems.

Recurrence. For any $0 \leq i < j \leq n$,

$$opt(i) = \begin{cases} \max_{0 \leq k < i, k \neq j} opt(k, j) + jump[i], & \text{if } s_i \geq s_j \\ \max_{0 \leq k < j, k \neq i} opt(i, k) + jump[j], & \text{if } s_i > s_j \\ opt(0, 1) = jump[1] \end{cases} \tag{2}$$

Explanation (not required for credit). This is slightly trickier than Problem 1. We make the following observation. Suppose $s_i \geq s_j$. Then the last interval before i in S_i can be any interval such that $f(k) \leq s(i)$. Also note that we are making sure the interval j is not selected as the last interval before i. But importantly, it can happen that k overlaps with j, which is totally cool since j is in S_2. The other case $s(i) < s(j)$ is analogous.

Pseudocode. Again this is quite straightforward. First sort the intervals by finish time. Maintain a 2-D array for the subproblems. The table needs to be filled row-wise from top to bottom and for each row column-wise from left to right. Note that you only need to fill entries in the upper triangle and not even the diagonal.

Runtime. The runtime here is $\mathcal{O}(n^3)$. This is because you are roughly filling up half the table, so $\Theta(n^2)$ entries and for each you might potentially look up $\mathcal{O}(n)$ entries to find the right value of k. (I am being sloppy here but not too much).

Remark : I think this problem might have a solution that runs in $\mathcal{O}(n^2)$. I need to check the details for that.

Problem 3. The IIIT-Delhi robotics club has organized a Robothon. n robots are placed along

the edge of a circular area at the middle of the OAT. Each robot will move along arbitrary tracks inside the circle while leaving behind a heat signature along its trail. However, they have been programmed not to cross their own trail or the trail of another robot, neither will they ever move out of the circle. In case a pair of robots i and j meet at any point, they are removed from the scene and the club will pay a reward sum of $M[i, j]$ to the owners of these robots. Note that some robots can keep moving infinitely without ever meeting another one. Given the reward matrix M where $M[i, j] = M[j, i]$, design a polynomial time algorithm that determines the maximum money the club might end up spending. For this particular problem, give a very brief justification of the recurrence.

Solution: This problem is almost same as 'matrix-chain multiplication' problem. The only difference here is the that the objective function is different here. Also, there would be a small difference in formulation of the recurrence relation. Note that when there is only one robot, then the club does not have to spend anything at all. So, we define the subproblems as follows.

Subproblems: For every $i \in \{1, \ldots, n\}$, we define $OPT(i, i) = 0$ as a single robot cannot meet any other robot. For every $i < j$, consider the set of robots $\{i, i+1, \ldots, j\}$. The subproblem $OPT(i, j)$ denotes the maximum money the club can spend when these sets of robots $\{i, i+1, \ldots, j-1, j\}$ meet among themselves. If $i > j$, then also we define $OPT(i, j) = 0$.

Otherwise, $j \geq i + 1$ and the following three possibilities can arise. First possibility is that there is some k such that $i < k \leq j$ such that robot i and robot k meet among each other. In that case, the set of robots $\{i+1, \ldots, k-1\}$ meet among themselves. In addition, the rest of robots $\{k+1, \ldots, j\}$ meet among themselves. In other case, the k is such that $i \leq k$. Then, i does not meet any robot. In that case, only $\{k+1, \ldots, j\}$ meet among themselves.

The optimal solution to the original problem is $OPT(1, n)$.

Recurrence relation: We develop a recurrence relation for $OPT(i, j)$ as follows.

$$OPT(i,j) = \begin{cases} 0 & \text{if } i \geq j \\ \max\limits_{i \leq k \leq j} \left\{ OPT(i+1, k-1) + OPT(k+1, j) + M[i,k] \right\} & \text{otherwise} \end{cases}$$

Correctness arguments: Observe that the recurrence relation covers all possible cases as $OPT(i, j) = 0$ when $i \geq j$. The case of $i = j$ is obvious. When, $j = i + 1$, then the two possibilities guess is that $k = i + 1$. In that case, the recursive calls are on $OPT(i + 1, i)$ and $OPT(i + 2, i + 1)$ both of whom are 0. There is unique way in which the robots i and $i + 1$ can meet. The cost will be $M[i, i + 1]$. Otherwise, $j \geq i + 2$ and the recurrence relation guesses a $i < k < j$. If $j > k \geq i + 2$, then clearly, the robots i and k meet. Among the rest, robots $\{i + 1, \ldots, k - 1\}$ meet. Finally, robots $\{k + 1, \ldots, j\}$ meet among themselves. Otherwise in case $i = k$ when i does not meet any robot. Then, the recurrence $OPT(i + 1, k - 1) = 0$ and only the set of robots $\{k + 1, \ldots, j\}$ meet.

Algorithm (constructing table): It is clear that the algorithm is exactly same as matrix chain except some minor difference in the recurrence relations. For every $i \geq j$, initialize $Tab[i, j] = 0$. Then, for every $j > i$ (in chronological order), fill up the values of $Tab[i, j]$ as follows.

$$Tab[i,j] = \max_{i \leq k \leq j} \left\{ Tab[i+1, k] + Tab[k+1, j] + M[i,k] \right\}$$