CSE 101 Homework 5

Winter 2023

This homework is due on gradescope Friday March 3rd at 11:59pm on gradescope. Remember to justify your work even if the problem does not explicitly say so. Writing your solutions in LaTeXis recommend though not required.

Question 1 (Gameshow Again, 30 points)**.** *Dirk's gameshow from Homework 4 decides to change their rules again. Now Dirk can attempt at most k challenges but* cannot *attempt the same challenge more than once. Give an $O(n\log(n) + kn)$ algorithm to find the strategy that optimizes Dirk's expected winnings.*

Hint: Based on the solution to part (a) from the previous problem you can note that whatever challenges Dirk attempts to try, he should always attempt them in decreasing order of $p_iR_i/(1-p_i)$ (with the order not mattering if there are ties). You can assume this without proof.

We first sort the challenges in decreasing order of $p_iR_i/(1-p_i)$. Let the ordered challenges be represented as $c_1, c_2, ..c_n$. Then consider the sub-problem parametrized by i, j : the maximum reward possible using challenges $c_i, ...c_n$ with at most j challenges that can be attempted. We will construct our DP table to be of the size $n \times k$.

Here, i can take values starting from n to 1, and j can take values starting from 0 to k. The answer to the problem would be present at cell $DP[1][k]$. We have two base cases, first, when $j = 0$, then $DP[i][0] = 0$ for all values of i. Second, if $i = n$, then $DP[n][j] = p_n.R_n$ (where j ranges from 1 to k).

For the recursive relation, Dirk has two options at any sub-problem $DP[i][j]$, either play the i^{th} challenge or skip the challenge. If Dirk decides to play the i^{th} challenge and assuming Dirk passes the i^{th} challenge, he should play the best sequence for the remaining j-1 spots from the remaining challenges, that is essentially the sub-problem DP[i+1][j-1]. Thus, the total expected reward for the described scenario is $p_iR_i + p_i.DP[i+1][j-1]$. If Dirk decides to skip the challenge we can simply refer to the sub-problem DP[i+1][j] which holds the maximum expected reward possible when considering challenges $c_{i+1}, ...c_n$ with at most j challenges that can be attempted. This gives us the following recursive relation.

$$\text{DP}[i][j] = \max\left(DP[i+1][j]),p_i.R_i + p_i.DP[i+1][j-1]\right), \tag{1}$$

```
Initialize the array DP[n][k].
For i=n:1
    For j=0:k
        If j = 0
            DP[i][j] = 0.
        Else If i = n
            DP[i][j] = p_i.R_i
        Else
            DP[i][j] = MAX(p_i.R_i + p_i . DP[i+1][j-1], DP[i+1][j] ).
Return DP[1][k]
```

To construct the solution initialize a list StrategyOrder of size k having the list of challenges in the correct order which leads to the maximum total expected reward. We start backtracking from entry DP[1][k]. For $i = 1$ and $j = k$, if the $p_i.R_i + p_i.DP[i+1][j-1] > DP[i+1][j]$ then add c_i to the list at index i, update $i = i + 1$ and $j = j - 1$. Else, we don't include c_i in the list and just update $i = i + 1$. We keep repeating this process as long as $i \leq n$ and $j \geq 0$.

```
Initialize StrategyOrder = [], i = 1, j = k.
While(i <= n and j >= 0)
    If p_i.R_i + p_i . DP[i+1][j-1] > DP[i+1][j]
        Add c_i to StrategyOrder
        i = i + 1
        j = j - 1
    Else
       i = i + 1
Return StrategyOrder
```

Proof: We need to show that for all values of i $(1 \leq i \leq n)$ and j $(0 \leq j \leq k)$, $DP[i][j]$ holds the maximum expected reward possible using challenges $c_i, ...c_n$ with at most j challenges that can be attempted. We approach the proof using strong induction on the parameters i, and j. Clearly, when j = 0, then DP[i][0] = 0 for all challenges i and when i = n, then DP[n][j] = $p_n.R_n$ (where j ranges from 1 to k), so our base cases are correct. Now, assume in our inductive hypothesis that $DP[i'][j']$ has been filled out correctly for all $i' > i$, $j' < j$. Then from our recurrence, we know that DP[i][j] is always assigned the correct value since our recurrence relation only relies on DP[i+1][j], DP[i+1][j-1], which are all assigned correctly by inductive hypothesis. Thus, DP[1][k] would contain the maximum expected reward possible for n challenges given Dirk can attempt at most k challenges.

Runtime: Sorting takes $O(n \log(n))$. There are a total of $O(nk)$ many sub-problems and computing each entry takes time $O(1)$. Hence, the total runtime is $O(n \log(n)) + nk)$.

Question 2 (Interval Cover Redux, 30 points)**.** *Consider the interval cover from before, but with weights assigned to each interval. You are given a collection C of n intervals in the real line, each with an associated positive weight. Your goal is to find a set S of intervals from C so that no two elements of S overlap and so that subject to this, the sum of the weights of these intervals is as large as possible. Give an $O(n \log(n))$ time algorithm for this problem.*

We first sort the intervals according to their start time in ascending order. Let the sorted intervals be represented as $I_1, I_2, ..I_n$. Then consider the sub-problem parametrized by i : the maximum weight possible when considering intervals $I_i, ...I_n$. We will construct our DP array to be of size n.

Here, i can take values starting from n to 1. The answer to the problem would be present at cell $DP[1]$. We have one base case, when i = n, then DP[n] = weight(I_n).

For the recursive relation, we have two options at any sub-problem $DP[i]$, either include the current interval in the set or exclude it. If we include I_i, then the total weight for our sub-problem DP[i] should include the weight of I_i and the maximum weight possible when considering intervals $I_j, ...I_n$, which is the sub-problem $DP[j]$. Here, j is the index of the first interval after i^{th} interval having starting time greater than the finish time of the i^{th} interval. Thus, we have $DP[i] = weight(I_i) + DP[j]$. We can find j by using binary search on the end time of I_i in the list consisting of the start time of intervals $I_{i+1}, ..I_n$. If we exclude I_i we can simply refer to the sub-problem DP[i+1] which holds the value for the maximum weight possible for intervals $I_{i+1}, ...I_n$. This gives us the following recursive relation.

$$\text{DP}[i] = \max\left((weight(I_i) + DP[j]) \, , DP[i+1] \right), \tag{2}$$

```
Initialize the array DP[n].
For i=n:1
    If i = n
        DP[i] =  weight(I_i)
    Else
        j = BinarySerach((StartTimeI_{i+1}, .. , StartTimeI_n), EndTimeI_i)
        DP[i] = MAX( weight(I_i) + DP[j], DP[i+1] ).
Return DP[1]
```

To construct the set of intervals (S) having the maximum total weight, we backtrack from entry DP[1]. If the $weight(I_1) + DP[j] > DP[2]$ (where j is the index of the first interval after I_1 having the start time greater than the finish time of I_1) then add I_1 in the set S and update i = j. Else, we don't include I_1 in the set S and update i = i+1. We keep repeating this process until we $i < n + 1$.

```
Initialize set S, i = 1.
While(i < n+1)
    j = BinarySerach((StartTimeI_{i+1}, .. , StartTimeI_n), EndTimeI_i)
    If weight(I_i) + DP[j] > DP[i+1]
        Add I_i to S
        i = j
    Else
       i = i +1
Return S
```

Proof: We need to show for all values of i ($1 \leq i \leq n$), $DP[i]$ holds the maximum weight possible for intervals $I_i, ...I_n$. We approach the proof using strong induction on the parameter i. Clearly, when $i = n$, then $DP[n] = weight(I_n)$ since we only have one interval, so our base case is correct. Now, assume in our inductive hypothesis that $DP[i']$ has been filled out correctly for all $i' > i$. Then from our recurrence, we know that $DP[i]$ is always assigned the correct value since our recurrence relation only relies on $DP[j]$ (for $j > i$), and $DP[i + 1]$, which are all assigned correctly by the inductive hypothesis. Thus, $DP[1]$ would contain the maximum sum weights of the given intervals without any overlap between any of the intervals.

Runtime: Sorting takes $O(n \log(n))$. There are a total of $O(n)$ many sub-problems and computing each entry takes time $O(\log(n))$ due to binary search. Hence, the total runtime is $O(n \log(n))$.

Question 3 (Pitstop Planning, 40 points)**.** *Jake is a racecar driver. In the current race, he needs to take n laps around the course. Unfortunately, his car's performance slowly gets worse each lap he performs without taking a pitstop. In particular, a lap will take time T_ℓ if Jake has gone ℓ laps since his last one. Unfortunately, Jake's pit crew has only materials to let him perform at most k pitstops over the course of the race.*

Give an $O(kn^2)$ time algorithm to determine which laps Jake should take pitstops in so as to minimize his total time to complete all n laps.

Consider the sub-problem parametrized by m, p, ℓ: the minimum time taken for Jake to finish m laps using p pitstops if he has already gone ℓ laps since his last pitstop. Here, m can take values from $\{0, \cdots, n\}$, p can take valeus from $\{0, \cdots k\}$ and ℓ can take values from $\{0, \cdots, n\}$. We will denote the answer to the subproblem as $\text{DP}(m, p, \ell)$.

Let's first consider the case $m > 0, p > 0$. Then, Jake can choose to either take a pitstop now or wait until future laps. If he takes the pitstop now, it will take him T_0 time to finish the current lap and then he will enter the subproblem $\mathrm{DP}(m-1, p-1, 1)$. Otherwise, it takes him T_ℓ time to finish the current lap and then he will enter the subproblem $\mathrm{DP}(m-1, p, \ell+1)$. Hence, we have the following recurrence

$$\mathrm{DP}(m, p, \ell) = \min\left(T_0 + \mathrm{DP}(m-1, p-1, 1)\,, T_\ell + \mathrm{DP}(m-1, p, \ell+1)\right), \tag{3}$$

when $m > 0, p > 0$.

Next, we discuss the base cases. If $m = 0$, it means that there are no more remaining laps. Hence, we always have $\mathrm{DP}(0, p, \ell) = 0$. If $p = 0$, it means we cannot take no more pitstops. Then, we have

$$\mathrm{DP}(m, 0, \ell) = T_\ell + \mathrm{DP}(m-1, 0, \ell+1). \tag{4}$$

Combining the two cases then gives the following algorithm for computing the minimum time used.

```
Initialize the array DP[n][k][n].
For m=0:n
    For p=0:k
        For L=0:n
            If m = 0
                DP[m][p][L] = 0
            Else If p = 0
                DP[m][p][L] = T_L + DP[m-1, 0, L+1].
            Else
                DP[m][p][L] = MIN( T_0 + DP(m-1, p-1, 1), T_L + DP[m-1, p, L+1] ).
Return DP[n][k][0]
```

When one computes one entry $\mathrm{DP}(m, p, \ell)$, the other entries it depends on have already been computed. Hence, it follows from our recurrences and mathematic induction that each entry in the table is computed correctly. $\mathrm{DP}(n, k, 0)$ corresponds exactly to the full problem, and hence we have correctly computed the minimum time of the race.

To retrieve the solution, we start backtracking from the entry $\mathrm{DP}(n, k, 0)$. If $T_0 + \mathrm{DP}(m-1, p-1, 1)$ is a valid option ($p > 0$) and larger than $T_L + \mathrm{DP}(m-1, p, L+1)$, we will take pitstop now and enter the entry $\mathrm{DP}(m-1, p-1, 1)$. Otherwise, we do not take the pitstop and enter the entry $\mathrm{DP}(m-1, p, L+1)$. Then, we repeat the process until we have constructed the decisions of whether we should take the pitstop after each lap.

```
Initialize m = n, p = k, L = 0.
Initialize Sol = [].
While(m > 0)
    If p > 0 AND T_0 + DP[m-1][p-1][1] < T_L + DP[m-1][p][L+1]
        Add "Take Pitstop" to Sol.
        m = m-1
        p = p-1
        L = 1
    Else
        Add "Do Not Take Pitstop" to Sol.
        m = m-1
        p = p
        L = L+1
Return Sol
```

Finally, there are altogether $O(n^2k)$ many sub-problems and computing each entry takes time $O(1)$. Hence, the total runtime is $O(n^2k)$.

Question 4 (Extra credit, 1 point). *Where have I been getting the character names for assignments this quarter from?*

Homestuck