CIT 596 Online **Summer 2020**

Module 9–10

Name: Solutions

Questions requiring written answers.

1. Two players play a game where they start with a row of n piles of varied amounts of money. The players take turns and in each turn a player can pocket either the money in the first pile or the last pile in the row of piles that remains. Design an efficient algorithm, which on any given sequence of amounts, determines the maximum amount of money that player 1 can win.

   If n is even, prove that player 1 wins at least half the money available. If n is odd, player 1 actually gets one more pile than player 2. In spite of that, show with a simple example that player 1 can be left with far less than half the total amount.

   Solution:

   Algorithm: We solve the problem using dynamic programming. Let $T[i, j]$ be defined as follows: if we start with the row of piles from position i to position j (inclusive), then $T[i, j]$ returns the maximum amount of money that the *current* player can win. Also, let $v[k]$ denote the value of pile k. Then

$$T[i,j] = \begin{cases} v[k] & j-i=0 \\ \max(v[k], v[j]) & j-i=1 \\ \max(v[i] + \min(T[i+2,j], T[i+1,j-1]), & \\ \quad v[j] + \min(T[i,j-2], T[i+1,j-1])) & j-i>1 \end{cases}$$

   We first fill in all the base cases and then iteratively compute cells in order of non-decreasing $j - i$ value. Finally, return $T[1, n]$.

   Correctness: Notice the problem has optimal substructure such that for any sequence of piles, the optimal strategy value does not depend on how we pick piles before arriving to this sequence. The correctness of the base cases is easy to see: when there is only 1 pile, there is only one choice; when there are 2 piles, we choose the greater of the two piles. The min terms within the max expression correspond to the fact that the opponent will play optimally on their next turn, thus minimizing your money. The two arguments of the max operation correspond to your 2 choices on each turn.

   Running Time: Both i and j range from 1 to n and thus there are n^2 subproblems. Each subproblem is solved in a constant number of steps, i.e. $O(1)$. Therefore in total the problem takes $O(n^2)$.

   n **is even:** First, note that our algorithm is optimal. Thus, if we can show another strategy that guarantees at least half of the money for player 1, then we know that our algorithm also at least achieves this. Here is a strategy: at the beginning of the game, player 1 sums all piles that have odd index, getting a number S_{odd}, and also sums all piles that have even index, getting a number S_{even}. If $S_{odd} \geq S_{even}$, then player 1 should always choose the odd-indexed pile, and vice versa. Note that this is possible because while player 1 can always choose either an odd or even-index pile, player 2 will always only have one choice because after player 1 selects a pile, both ends will be odd or both will be even.

   n **is odd:** Player 1 will get far less money than player 2 in the following example: $[1, 100, 1]$. Assuming both play optimally, player 1 will only get 2.

2. Given an array of integers A, $a_1, a_2, \ldots, a_n$, design an efficient algorithm that finds the length of the longest increasing subsequence of the array. A subsequence of A is a (not necessarily consecutive)

sequence of elements from A that can be obtained by deleting, but not rearranging, elements of A. An increasing sequence is one where $a_i < a_j$ for all $1 \leq i < j \leq n$.

Hint: One method of solving this problem is find, for each valid i and j, the "best" increasing subsequence of length i among the integers $a_1, a_2, \ldots, a_j$, where the notion of "best" is for you to define. These can be your subproblems.

Solution:

Algorithm: Let $T[i]$ be the length of a longest increasing subsequence of A whose last element is a_i, for $1 \leq i \leq n$. Then

$$T[i] = \begin{cases} 1 & i = 1 \\ 1 + \max\left(\{T[j] \mid 1 \leq j < i,\ a_j < a_i\} \cup \{0\}\right) & i > 1 \end{cases}$$

Return max(T[1], T[2],..., T[n]).

Correctness: The longest increasing subsequence may end in $T[1]$, $T[2]$, or even $T[n]$, which we consider. To find $T[i]$, $i > 1$, our algorithm considers extending all possible sequences before a_i with a_i where the result is also an increasing subsequence. It ensures that only sequences whose values are less than a_i will be added. It then takes the maximum (best) value. Note that our algorithm guarantees that $T[i] \geq 1$ for all $1 \leq i \leq n$. This is correct because a singleton sequence is trivially an increasing subsequence with length 1.

Running time: There are n subproblems, and each takes n time. Hence, running time is $O(n^2)$

3. Consider an assignment with n questions. For each i ($1 \leq i \leq n$), the ith question has p_i points and time t_i associated with it. Find an efficient algorithm to compute the minimum time required to obtain a target of P points. (Assume that you either get full points or 0 on each question. Of course, this is not true in our course!)

   Solution:

   Algorithm: Let $T[i, j]$ denote the minimum time needed to obtain j points if you are restricted to choosing from the first i questions. Then compute as:

$$T[i,j] = \begin{cases} 0 & j = 0 \\ \infty & i = 0, j > 0 \\ \min(t_i + T(i-1, j-p_i), T(i-1, j)) & i > 0, j > 0 \end{cases}$$

   Note that for the general recurrence case (both i, $j > 1$), we assume that any out-of-bounds entry in T has value ∞. We compute using i in the outer loop and j in the inner loop. Return $T[n, P]$.

   Correctness: First, observe that it's clear that we should return $T[n, P]$. Now we explain the recurrence. (i) we can clearly achieve $j = 0$ points with 0 time since we don't need to choose any problems at all. (ii) it's clearly impossible to achieve any points without using any questions. (iii) for the general case, the terms in the min function correspond to using the ith problem and not using it, respectively.

   Running Time: There are $O(nP)$ subproblem, and each clearly takes $O(1)$ time. The total runtime is $O(nP)$.

4. Give the run-time of the algorithm taught in the lecture video to find a piece-wise linear regression fit for n points using dynamic programming so as to minimize the cost defined in the video:

$$Ck + \sum_{j=1}^{k} \sum_{i=i_j+1}^{i_{j+1}} (y_i - m_j x_i - b_j)^2$$

   Solution:

From lecture, we have the recurrence as

$$\text{OPT}[j] = \begin{cases} 0 & j = 0 \\ C & j = 1 \vee j = 2 \\ \min\limits_{i<j} \left(\text{OPT}[i] + C + \text{REG}(i,j)\right) & j > 2 \end{cases}$$

As stated in lecture, we can precompute each of the $O(n^2)$ REG values in $O(1)$, for a total of $O(n^2)$. In addition, there are n subproblems, each taking $O(n)$ time. Thus, the total running time is $O(n^2)$.

5. (Chapter 6, problem 13 on page 324 from CLRS)
The problem of searching for cycles in graphs arises naturally in financial trading applications. Consider a firm that trades shares in n different companies. For each pair $i \neq j$, they maintain a trade ratio r_{ij}, meaning that one share of i trades for r_{ij} shares of j. Here we allow the rate r to be fractional; that is, $r_{ij} = \frac{2}{3}$ means that you can trade three shares of i to get two shares of j.

A trading cycle for a sequence of shares $i_1, i_2, \cdots, i_k$ consists of successively trading shares in company i_1 for shares in company i_2, then shares in company i_2 for shares i_3, and so on, finally trading shares in i_k back to shares in company i_1. After such a sequence of trades, one ends up with shares in the same company i_1 that one starts with. Trading around a cycle is usually a bad idea, as you tend to end up with fewer shares than you started with. But occasionally, for short periods of time, there are opportunities to increase shares. We will call such a cycle an opportunity cycle, if trading along the cycle increases the number of shares. This happens exactly if the product of the ratios along the cycle is above 1. In analyzing the state of the market, a firm engaged in trading would like to know if there are any opportunity cycles.

Give a $O(n^3)$ algorithm that finds such an opportunity cycle, if one exists.

Solution:

Algorithm: Let R represent an adjacency matrix for a complete graph, where each currency c_i is represented as a node and each entry in the matrix $R[i,j]$ represents the weight of an edge in the graph. Create a new adjacency matrix R' and let $R'[i,j] = -\log R[i,j]$. Then run the Bellman-Ford algorithm on R' from any vertex to detect if there is a negative cycle. If there is, output TRUE, else output FALSE.

Correctness: We are looking for an opportunity cycle, which occurs when

$$\left(\prod_{i=1}^{k-1} R[c_i, c_{i+1}]\right) \cdot R[c_k, c_1] > 1$$

This desired inequality can be transformed:

$$\begin{aligned} &\implies \left(\sum_{i=1}^{k-1} \log R[c_i, c_{i+1}]\right) + \log R[c_k, c_1] > 0 \\ &\implies \left(\sum_{i=1}^{k-1} -\log R[c_i, c_{i+1}]\right) - \log R[c_k, c_1] < 0 \\ &\implies \left(\sum_{i=1}^{k-1} R'[c_i, c_{i+1}]\right) + R'[c_k, c_1] < 0 \end{aligned}$$

This last expression corresponds exactly to a negative cycle in the graph representation of R'. Thus, finding a negative cycle in R' is equivalent to finding an opportunity cycle in R. We can use Bellman-Ford to find negative cycles by simply inspecting all paths from each vertex to itself.

Correctness: Since we use an adjacency matrix, $O(m) = O(n^2)$. Creating R' takes $O(n^2)$ time, and running Bellman-Ford takes $O(mn) = O(n^3)$ time. Thus, the entire algorithm takes $O(n^3)$.

6. You want to go from station 1 to station n by rail. The train fare from station i to station j is $F(i, j)$ and can be found in $O(1)$ time. You want to break your trip up into a series of segments in order to minimize the cost, but each segment must go in the forward direction, i.e., from station i to station j for $i < j$. Design an efficient dynamic programming algorithm for doing this and analyze its running time.

   Solution:

   Algorithm: For $j \in [1..n]$, define $T[j]$ to be the minimum cost to travel from station 1 to station j. Then we have the recurrence

$$T[j] = \begin{cases} 0 & j = 1 \\ \min\limits_{i<j} (T[i] + F(i, j)) & j > 1 \end{cases}$$

   Compute $T[j]$ for all values from 1 to n and then return $T[n]$.

   Correctness: The base case is trivial. For the general case of reaching the jth station, note that we are trying every possible "last" segment and taking the minimum over all of the possibilities. Inductively, all values of $T[i]$ for $i < j$ are correct (optimal), so this recurrence is correct.

   Runtime: There are clearly $O(n)$ subproblems, each taking $O(n)$ time. Thus the total runtime is $O(n^2)$.