CS3000 Algorithms *Out: 23 July 2023*

Problem Set 3

Akshar Varma *Due: 30 July 2023*

The maximum score for this problem set is 20 **(that'll be A), we'll use the best possible score out of all attempted problems in your solution.**

Dynamic Programming Solution Guidelines

Dynamic programming can be very tricky and this template will help guide you through solving new problems. Get in the habit of going through the list and filling everything out step by step. We will grade harshly on missing items. And if there's no english description of the two items mentioned below then the solution will get an **AUTOMATIC 0** on homeworks and exams with no exceptions.

1. (**AUTOMATIC 0 IF MISSING**) English description of the
   (a) recursion you use/the memoization table.
   (b) logic behind your recursion.
2. The actual recursion. And don't forget the base cases!
3. Final value we have to return.
4. Brief description of how an iterative algorithm would loop through the memoization array and fill the values (make sure your order makes sense, and that values are already filled when you call them). Pseudocode isn't required, just a couple sentences.
5. The number of subproblems you have to solve.
6. How much time each subproblem takes to solve (usually constant or linear).
7. Final running time.

Problem 1 (Counting Coin Changes) 4

Given a denomination of positive coins $c_1, c_2, \ldots, c_m$ and a value n as input how many ways can you make change for n. For example, with coins being $\{1, 2, 3\}$ the number of ways to get the value 4 is 4 as follows: $\{1, 1, 1, 1\}, \{1, 1, 2\}, \{2, 2\}, \{3, 1\}$. With coins being $\{2, 5, 3, 6\}$ the number of ways to get the value 10 is 5 as follows: $\{2, 2, 2, 2, 2\}, \{2, 2, 3, 3\}, \{2, 2, 6\}, \{2, 3, 5\}, \{5, 5\}$.

English Description: The recursion that I would use for this algorithm would include using a memoization table to include the different ways that change can be made and store the result of the sub-problems. Our function should return the number of ways that change can be made, so we can look at the base cases for this. If the target value of change that you are trying to achieve is 0, then there is only 1 way of making change. If the number of coins at hand is all used, then we can return 0. For each coin, we have two choices to make to either choose the coin or to not choose the coin.

Bellman Equation :

$$\text{T(i, n)} = \begin{cases} 1 & n = 0 \\ 0 & i = 0 n > 0 \\ T(i-1, n) + T(i, n - c_i) & else \end{cases}$$

Final Value to Return:

The final value that will be returned here will be the result of the recursive call, which will be the sum of the number of ways that the change can be achieved with the coins at hand. This will happen through the recursion.

Iterative Step:

The iterative step here will be a for loop that will traverse through the array of given coins. It will loop through the memoization table from the bottom up, filling in the values from the previous subproblems.

Number of Subproblems: The number of subproblems will be the length of how many coins there are, times the target that is trying to achieve.

Time To Solve: The time to solve each of these sub-problems is roughly $O(1)$ as each of the sub-problems is doing very nominal operations like addition and recursive calls.

Runtime: The total runtime for this will be just the length of the coins times the target value, this is due to the fact that each problem will take O(1) to complete, so it will be constant to the amount of subproblems.

Problem 2 (Alice and Bob and Alternating Games) 4

Suppose Alice and Bob are given an array $A[1...n]$ of integers. They are playing a game where they alternate turns, and at each turn a player chooses one of the two integers at the end of the array. After they choose that number, the number gets deleted from the array and it's the next person's turn. The winner of the game is the one who has the larger sum from the numbers they have chosen. Give a dynamic programming solution for Alice to *maximize the sum of the integers she chooses*, assuming that Bob plays optimally.

For example, suppose we have the array $A = [8, 7, 8, 9]$. Alice can choose 9 in the first turn. Then we have the array $[8, 7, 8]$ and Bob can choose either 8. Then Alice has a choice between picking 7 and 8 so she chooses 8 which leaves Bob with 7. Alternatively, if Alice had initially chosen 8, then Bob would have chosen the 9 and Alice would choose the remaining 8. Thus, for this specific configuration, with optimal play, Alice and Bob will both have the same totals.

English Description: For this algorithm, Alice will either choose the first or the last element in the array, whichever one is the maximum, and then simulate Bob's play with him choosing the best from the new array, and then choosing the best play for Alice from the newest array. This is so we can make sure that the decision Alice makes accounts for Bob's next turn to make sure our next turn is also maximized.

Bellman Equation:

$$T[i] = \begin{cases} i[1] & len(i) = 1 \\ \max(i[1], i[j]) & j = 2 \\ \max(i[1] + T[i[2...j] - T(i[2...j])], i[j] + T[i[1...j-1] - T(i[1...j-1]]) & else \end{cases} \quad (1)$$

Final Output Value: The final output value for our algorithm will be the maximum sum that Alice will end up with in the end. This is due to the fact that our algorithm recursively adds up the simulated moves Alice can make while accounting for the moves that Bob will make. The algorithm will calculate the sum of the best possible choices made by Alice.

Iterative: As the iterative step is basically a summation operation, it will keep adding the selected value of Alice to the running total of the sum of Alice's moves. This means that our iteration will start at $T[0]$ with a value of 0. As the algorithm progresses, it will add each of Alice's selected values to get a final iteration of $T[A[1...n]]$.

Number of Subproblems: As there are n elements in the array that we are starting with, there will be n sub-problems to solve in this case. We are choosing the maximum from a given array n times, as there are n elements.

Time To Solve: The time to solve each of these sub-problems is roughly $O(1)$, as they will be in constant time in finding the max of two numbers, as well as the time to add the values together. If we were to account for the array deletion in our algorithm, then it would cause our time complexity to increase to linear time, $O(n)$. This is due to the fact that when an item is deleted from an array all of the other elements following must be copied and moved up.

Final Runtime: The final run-time for this algorithm would be simply $O(n)$, if we are to not account for the array deletion, but if we did, it would be $O(n^2)$.

Problem 3 (PalindromemordnilaP) 4

Given an array $A[1...n]$ of letters, compute the length of the longest palindrome subsequence of A. A sequence $P[1...m]$ is a palindrome if $P[i] = P[m - i + 1]$ for every i. For example, RACECAR is a palindrome, while REAR is not a palindrome. And the longest palindrome subsequence in the string ABETEB is BEEB.

English Description: For this problem, the algorithm will initialize two loops, with one starting at the beginning of the array and the other at the end. This way we can check for palindrome, by seeing if the letters, in the beginning, are the same as those at the end. We can recurse through this array, from the front and the back to find the length of the longest palindrome subsequence. We can write the recursive equation to be $P(i, j)$, where i and j are the indexes that will be outputted by the for loops, respectively.

Bellman Equation:

$$P[i,j] = \begin{cases} 0 & i > j \\ 1 & i = j \\ P(i+1, j-1) + 2 & A[i] = A[j] \\ \max(P(i+1, j), P(i, j-1)) & else \end{cases} \tag{2}$$

Final Output Value: The final output value for this algorithm will be a number that will be the length of the longest palindromic subsequence. This is because once the function will reach the base case, it will simply return a number, which will build together will all of the cases to get your overall length.

Iterative: The iterative operation here is the for loops that will allow the function to traverse through the array. This means that it will start iterating with i being 1, and j being the length of the array we are given. Starting from there, the i and j are dynamically updated based on the cases, until

the value of i becomes greater than j, which would mean that there are no more palindromes as the values have crossed each other now.

Number of Subproblems: As there are two for-loops that are traversing through an array of length n, this means that there will be a total of n^2 sub-problems that will be solved.

Time To Solve: The time to solve each of these subproblems will be constant as there is just a simple arithmetic that is done at each step. The time complexity of an arithmetic operation is 1, so at each of the steps it will take constant time.

Final Runtime: The final run-time for this algorithm would be simply $O(n^2)$ as each of n^2 problems will take constant time to complete, meaning $O(n * 1) = O(n)$.

Problem 4 (No Liars) 4

Given an $m \times n$ binary matrix, where each 1 represents that a knight who only speak the truth is standing there and each 0 represents that a knave who only speaks falsehoods is standing there, find size of largest square sub-matrix that does not have any liars in it. In the following example, the biggest square sub-matrix without liars (without 0s) is of size 3 (using the middle rows in the rightmost columns as highlighted).

0	0	1	0	1	1
0	1	1	1	0	0
0	0	1	**1**	**1**	**1**
1	1	0	**1**	**1**	**1**
1	1	1	**1**	**1**	**1**
1	1	0	1	1	1
1	0	1	1	1	1
1	1	1	0	1	1

English Description: We can use a two-dimensional array that is the same size as the binary matrix. We can store the size of the largest square matrix with no 0s in this array with $L[i][j]$ saying the size of the subsquare with no 0s. The function will go to each of these cells are check for the neighbors to make sure that it is not surrounded by 0s, making it a subsquare. Traversing form the top of the array and checking for the cells, we can go through each row and column to calculate the size of the largest sub-square and store it in the memoization array.

Bellman Equation :

$$\text{T(i, j)} = \begin{cases} 0 & L[i][j] = 0 \\ 1 & i = 0 or j = 0 \\ min(T[i-1][j-1], T[i-1][j], T[i][j-1]) + 1 & L[i][j] = 1 \end{cases}$$

Final Value to Return: The final value that this will return will be the size of the largest submatrix that are all ones.

Iterative Step: The iterative step here would be using two for loops to traverse through the matrix's rows and columns. It will start at the cell in the top left and work its way to the bottom right. At each step, the idea is to check the neighbors of the cell you are currently iterating. If the value of the cell we are checking is 0, then we return zero. However, if it is 1, we want to add one, and recursively check the neighbors of that cell. The memoization table will store the value of the largest submatrix. This way by the end, we will be given the size of the largest submatrix that will contain all 1s.

Number of Subproblems: The number of subproblems here will be the number of cells, which is $n\dot{m}$.

Time To Solve: The time to solve each of the sub-problems here would be minimal as it is simply checking the values of the cells in a matrix. It would be roughly $O(1)$.

Runtime: The total runtime for this algorithm would be simply $O(n\dot{m})$.

Problem 5 (All about Trees) 2+2+2+2+4

Let T be a binary search tree (BST) with n nodes. Each node stores a key which is used to compare nodes for the invariance property of BSTs.[1] Apart from the key, the nodes can store any arbitrary data. You are a data analytics intern tasked with maintaining the following extra metadata about the nodes of the BST.

Depth of a node: The number of edges needed to traverse starting from the root until this node is reached. The depth of the root node is 0.

Height of a node: The maximum number of edges traversed among all paths from this node to a leaf node. The height of a leaf node is 0.

Subtree size of a node: The total number of nodes that can be reached from the node including itself. The leaf node can only reach itself, so its subtree size is 1.

Rank of a node: The rank/order/index of this node's key if all keys are sorted (the order statistic value associated with this node's key). The leftmost leaf node will have rank 1 and the rightmost leaf node will have rank n.

1. Describe algorithms that will allow you to find and store the required metadata quickly given a n node BST T as input.
2. **(∗)** BST usage involves updating nodes; describe how each of these metadata values can be kept updated as elements are added or removed from T.

[1] Every node's key is at least as large as the largest key in the left subtree and at most as large as the smallest key in the right subtree.