CS 3510: Design & Analysis of Algorithms (Sections A & B)

Homework 3: Divide & Conquer, Intro to Dynamic Programming

Professors Dana Randall and Gerandy Brito **Due: 9/20/2022 11:59pm**

Name: Anthony Wong

1.) Let us consider fútbol players (or what Americans call "soccerers") who are lined up in a row. We can refer to the players by where they are standing in the line - positions numbered $1, \ldots, n$. Each player is wearing a *unique* numbered jersey which has an integer (which can be positive, negative, or zero) printed on it, and all the players are lined up such that the numbers on their jerseys are *increasing.* **For this question, your solutions must be in plain English and mathematical expressions (do not use pseudocode).**

(a.) (15 points) Design a Divide & Conquer algorithm to determine if there is a position, i, such that the player standing in position i is wearing the jersey with the number i on it. Your algorithm must run in $\mathcal{O}(\log n)$, and you must use the Master Theorem to prove your algorithm's time complexity.

Solution:

A design that could accomplish this task could be done with the following. First, we take what we can assume to be the array of players lined up and grab the middle element. We can then check if the index of this middle element matches the player number at that index. There are 3 cases we can hit:

Case 1: If the index and player number are equal, simply return true as we've found a true case
Case 2: If the index is greater than the player number, we should recurse on everything to the right of the middle index
Case 3: If the index is less than the player number, we should recurse on everything to the left of the middle index

We should continue going through this recursion process (continually checking middle indices and going through the above cases) until we either reach a match (at which point we return true) or we run out of elements to recurse on (at which point we return false). The time complexity can be described as $T(n) = T(n/2) + O(1)$. Using the Master Theorem, we can simplify this to $T(n) = O(\log n)$.

(b.) (15 points) Modify your algorithm from part (a.) to find the *smallest* value for i such that the player standing in position i is wearing the jersey with the number i (the modified algorithm should still use a Divide & Conquer approach). If this modified algorithm has the same time complexity as the one from part (a.), explain why; otherwise, use the Master Theorem to prove what the modified algorithm's time complexity is.

Solution:

In order to ensure that we return the smallest value i that has a matching position and jersey number, we need to modify part (a) by changing the cases that we will hit. They will be as such:

Case 1: Of the index and player number are equal, we should store the index as our smallest i and then recurse on everything to the left of the middle index
Case 2: If the index is greater than the player number, we should recurse on everything to the right of the middle index
Case 3: If the index is less than the player number, we should recurse on everything to the left of the middle index

The only change here in algorithm is that we do not stop when we find a match as there could be smaller match. We then rely on the base case where we run out of elements to recurse on. Additionally, we should store every match we find, as the last match we find will be the smallest i. This algorithm has the same time complexity as part (a). This is because in both part (a) and part (b), we can envision a case where the match in index and player number is the first element. In both parts, we go through the same recursive cases, meaning that in our worst cases, we still have the same runtime.

(c.) (10 points) Provide a Divide & Conquer algorithm to find the total number of players standing in a position that matches their jersey's number. You must prove why your algorithm finds the correct count and provide runtime analysis.

Solution:

We can again mimic our part (a) and part (b) solutions. Again, we first take what we can assume to be the array of players lined up and grab the middle element. We can then check if the index of this middle element matches the player number at that index. Assume we have a running count of matches. There are 3 cases we can hit:

Case 1: If the index and player number are equal, we increment our running tally of matches and recurse on both elements to the left and right of the middle index
Case 2: If the index is greater than the player number, we should recurse on everything to the right of the middle index
Case 3: If the index is less than the player, we should recurse on everything to the left of the middle index

We should continue going through this recursion process (continually checking middle indices and going through the above cases) until we either reach a match (at which point we increment the count) or we run out of elements to recurse on (at which point we return).

In proving why this algorithm finds the correct count, we can first discuss Case 2 and

Case 3. In Case 2 when the index is greater than the player number, we can disregard everything to the left because the player number cannot decrease slow enough such that it catches up to the index in value. The reverse is true for Case 3 where the index is less than the player number. In this case, we disregard everything to the right because the player number cannot increase slow enough such that it catches up to the index in value. As for Case 1, we ensure to tally our match. Because matches can occur on both sides of the index we recurse on both sides. This ensures we get the correct count. In analysing the runtime, let's consider the worst case such that every index matches the corresponding player number. This means that we recurse both sides of every middle index we check. Intuitively we know this to be $O(n)$. We can prove this with the Master Theorem where we can describe the time complexity as $T(n) = 2T(n/2) + O(1) = O(n^{\log_2 2}) = O(n)$.

2.) (30 points) Solve the following dynamic programming problems by creating and filling out tables using the algorithms shown in lecture.

(a.) (15 points) Find the longest increasing subsequence in the array $[5, 6, 9, 7, 2, 13, 11, 8, 10, 17, 1, 3]$. Your response must include: (1) the filled out DP table and (2) the longest increasing subsequence in the array and a brief explanation of how you used the table to find this result.

Solution:

(1)

Input	5	6	9	7	2	13	11	8	10	17	1	3
T[k]	1	2	3	3	1	4	4	4	5	6	1	2

(2) The longest increasing subsequence in the array is $[5, 6, 7, 8, 10, 17]$. To find this, we start by finding the max length of the LIS. Looking in the T[k] row we see this is 6. Starting at 6, we backtrack in the T[k] row from 6 to 1, passing through last (from the back) entries with those corresponding T[k] values. This means we start from 6 which maps to 17, 5 which maps to 10, the last 4 which maps to 8, etc.

(b.) (15 points) Find the longest common subsequence between the strings "NEWSLANG" and "SIMPLESONG". Your response must include: (1) the filled out DP table and (2) the longest common subsequence between the strings and a brief explanation of how you used the table to find this result.

Solution:

(1)

	N	E	W	S	L	A	N	G
S	0	0	0	1	1	1	1	1
I	0	0	0	1	1	1	1	1
M	0	0	0	1	1	1	1	1
P	0	0	0	1	1	1	1	1
L	0	0	0	1	2	2	2	2
E	0	1	1	1	2	2	2	2
S	0	1	1	2	2	2	2	2
O	0	1	1	2	2	2	2	2
N	1	1	1	2	2	2	3	3
G	1	1	1	2	2	2	3	4

(2) The longest common subsequence is "ESNG". To find this, we backtrack starting in the bottom right corner with the max length being 4. For every number from 4 to 0, we find the corresponding match that brought us that number. This usually occurs at the corners of each number block. In other words these are diagonal updates as outlined by the algorithm.

3.) (30 points) Given a list A of positive integers, you want to find the maximum sum of numbers such that no numbers in the sum are adjacent in the list A. Design a dynamic programming algorithm to solve this problem. For example, if the maximum sum contains the number at position 2 in the list, then the sum cannot contain the number at position 1 nor the number at position 3. The following are some examples of inputs and expected outputs:

- Example 1: Input = [1, 10, 8], Output = 10. We couldn't add 10 to either 1 or 8 since both of those are next to 10, so the list of all possible options is (1), (8), (1 + 8), and (10), of which 10 is the largest.
- Example 2: Input = [3, 2, 1, 4], Output = 7. Notice here that the gap between numbers in the maximum sum is not necessarily 1 index apart.
- Example 3: Input = [2, 1, 2, 3, 2], Output = 6. Here, the maximum sum is 2 + 2 + 2. Since none of the 2's are adjacent, our sum in this case contains three numbers.

(a.) Define the entries of your table in words, e.g. $T[i]$ or $T[i, j]$ is ...

Solution:

$T[i]$ is the maximum sum of non-adjacent numbers of $[A_1, A_2, ..., A_k]$ for $1 \le k \le n$ ending at A_k (With n being the length of A).

(b.) State your base cases and the recurrence for entries of your table in terms of smaller subproblems. Briefly explain in words why it is correct.

Solution:

We need to first state our bases cases, which are

$T[0] = 0$, $T[1] = A[1]$

In defining the recurrence for smaller subproblems, we have the following:

$T[i] = \max\{T[i-1], T[i-2] + A[i]\}$ where $2 \le i \le n$

We know this propagates the correct answer because each previous entry in the constructed table T is the maximum sum of non-adjacent numbers until that index in the array A. When we populate T, at every new element we iterate through, we know we cannot add to the sum of the previous index given they are adjacent. The best we can do is add to the sum of out index - 2. Taking the max of these two figures ensures that we store the maximum sum up to the current index.

(c.) Write pseudocode for your algorithm to solve this problem.

Solution:

MaxNonAdjacentSum(A=[$A_1, A_2, ..., A_n$])

```
T[0] = 0
T[1] = A[1]

for i=2 to n:
    T[i] = max(T[i-1], T[i-2] + A[i])

return T[n]
```

(d.) Analyze the running time of your algorithm.

Solution:

In analysing the above algorithm, we can see that we make one pass through the input array A. It follows that the time complexity can be described as $O(n)$.