CSE321

INTRODUCTION TO ALGORITHM DESIGN

HOMEWORK 5

Burcu Sultan ORHAN

1901042667

Question 1:

For this algorithm, I used a recursive method to implement divide & conquer strategy. Method initially takes 3 inputs: string array, bottom value which is 0, top value which is the length of the array. Since method will recursively check every value in the string array, bottom and top values will change and when local array has only 1 value, these bottom and top values will match. And this is our base condition. When this happens, there's no need to compare two strings because there's only one. In this condition, method returns the only element of the array. Other than that, if top value is greater than bottom value, method recursively calls itself with 2 different sub-arrays: one is from bottom to middle, other is from middle to top. Method gets two substrings from these calls, and checks their values from 0 to the length, char by char, and stores matching chars in another string called buildString. When method finds a non-match, it stops searching and returns buildString.

Worst Time Complexity: O(log n) for the recursive calls, O(n) for the operations inside, overall O(n log n).

Question 2:

In this question for dynamic programming algorithm:

This part reminded me of the closest pair algorithm. For that, we would draw a line in the middle and recursively check left and right sides of this line. On each side, we would check every two value and find the closest ones, and lastly, we would check the rightmost of the left side and leftmost of the right side to see how close they are. We compare these 3 values and return the minimum one. For this algorithm, I implemented a very similar method. Method parts the array into two halves, check two halves recursively (base condition is when the array has 1 value, array returns that value), and compares substraction of right array's max value and left array's min value with the values it gets from the two halves. Returns maximum of these values.

Worst time complexity: O(log n) for the recursive calls, O(n) for the operations inside each call, overall O(n log n).

In this question for non-dynamic algorithm:

For this part, method initializes min and tempMin as the first value of the array, gap as 0. I implemented a simple for loop to iterate through the array. On each iteration, it checks new gap with min value, if new gap is bigger than the previous one, gap is updated. And if current value of the array is smaller than min value, method stores this value in tempMin to see if these would be a successor element greater than tempMin. If there is, min becomes tempMin.

Worst time complexity: O(n) since there's a simple for loop and basic operations within.

Comparison: Second algorithm is better in terms of worst time complexity because its is O(n) whereas first's is O(n log n).

Question 3:

For this algorithm, to turn it into a dynamic algorithm, I firstly thought of a recursive method. And turned those recursive calls into an array which is a characteristic of dynamic programming. Method takes an array and in one for loop, it iterates through the array. On each iteration it checks whether the successor element is bigger than the current one. If so, dynamicValues array's current element is incremented. And lastly, method return max value in the dynamicValues array.

Worst time complexity: O(n) because it iterates through the array no matter what.

Question 4:

For this question, (aside from the previous report which explains the algorithm), I'll explain differences with 2 new ones.

Dynamic Programming:

For this algorithm, I turned the recursive calls from the previous HW into an array which stores max values along the way. Which bases dynamic programming.

Worst time complexity: O(n) because method iterates through map once.

Greedy Algorithm:

For this algorithm, I turned the previous HW's algorithm from recursive to iterative to make it greedy algorithm.

Worst time complexity: O(n) because method iterates through map once.

Comparison: Previous HW's algorithm has O(n) for the recursive calls, O(1) for the operations inside, overall O(n) worst time complexity. Which makes these three algorithms same in terms of worst time complexity.

OUTPUTS:

```
\.vscode\extensions\ms-python.python-2022.20.1\pythonFiles\lib\python\debugpy\adapter/../..\debugpy\launcher'
\cse321-hw5-1901042667.py'
['burcu', 'burcusultan', 'burcusultanorhan', 'burciga']
Longest common substring: burc


[4, 20, 0, 14, 4, 15, 2, 11]
Maximum profit found with Dynamic Programming:  16
Maximum profit found without Dynamic Programming:  16
Longest increasing subarray:  2


Input: n = 8
Input m = 4
Game map:

[33, 47, 48, 65]
[30, 82, 72, 3]
[62, 21, 44, 39]
[79, 13, 16, 98]
[98, 82, 71, 56]
[76, 44, 26, 43]
[69, 47, 97, 90]
[44, 63, 80, 53]
Max points found with Dynamic Programming:  657
Max points found with Greedy algorithm:  657
PS C:\Users\burcu\Desktop>
```