GTU Department of Computer Engineering
CSE 321 – Fall 2022
Homework #5

Hasan Mutlu
1801042673

Question 1: Longest Common Substring at the Beginning

This divide and conquer algorithm to find the longest common substring at the beginning of each word consists of two methods. The first method *longestCommonSubstring* takes the string array as input and recursively halves the array and calls itself again for both half arrays. The base case of this recursion is when the length of the passed array is 1.

The algorithm continues to divide the input array into smaller subarrays and find the longest common prefix of the subarrays until it reaches the base case, at which point it starts working its way back up the recursion tree, combining the substrings returned by the recursive calls to find the final answer.

In order to combine the substrings returned by each half of the array, the *findLongestCommonSubstring* method is called. It takes two strings as input and simply compares the characters of the two strings and returns the common substring so far when it encounters an uncommon character.

Here are the outputs of the algorithm. The two examples from the PDF and an example where there is no common subarray at the beginning is used:

```
Part 1: Longest Common Substring

strings: ['programmable', 'programming', 'programmer', 'programmatic', 'programmability']
Longest common substring: programm


strings: ['compute', 'compatible', 'computer', 'compare', 'compactness']
Longest common substring: comp


strings: ['abcd', 'bcde', 'cdef', 'defg', 'efgh']
Longest common substring:
```

Worst Case Time Complexity

In any case, since the input array is divided into equal-sized subarrays at each step, the depth of the recursion is logn. It takes O(n) time to find the longest common sequence of characters at each step. Therefore, the time complexity of the algorithm is O(n log n).

Question 2.a: Maximum Profit by Divide and Conquer

The divide and conquer algorithm to find the maximum profit consists of one method called *maxProfitDaC*. It works by dividing the input list of prices into two equal halves at each step. In the base case, when the array of prices only have one element, it returns 0, 0, 0 for profit amount, buy day and sell day, as there is no profit to be made. After the base step is reached, it starts calculating the max profit that can be made by working its way back up to the calling methods.

At each step, the maximum profit that can be obtained by buying and selling in each half of the list are obtained by the recursive calls. The profit that comes from buying on first half of the list and selling on the second half of the list (cross profit) is calculated by the returnee values from these recursive calls. Finally,

the maximum amount of profit that can be obtained from either one of these three possibilities is returned along with buy and sell days.

Here are the outputs of examples. The first two examples are the ones from the PDF.

```
Part2.1: Maximum Profit by Divide and Conquer

Prices: [10, 11, 10, 9, 8, 7, 9, 11]
Buy on Day5 for $7 and sell on Day7 for $11.

Prices: [100, 110, 80, 90, 110, 70, 80, 80, 90]
Buy on Day2 for $80 and sell on Day4 for $110.

Prices: [10, 7, 10, 11, 8, 10, 9, 11]
Buy on Day1 for $7 and sell on Day3 for $11.
```

Worst Case Time Complexity:

In any case, since the input array is divided into equal-sized subarrays at each step, the depth of the recursion is logn. In each step, there are expressions to find the minimums, maximums or indices of elements in the array. These expressions take O(n) time. So, in conclusion, it takes O(nlogn) time.

Question 2.b: Maximum Profit in Linear Time

The algorithm to find the maximum profit in linear time consists of one method called *maxProfitLin.* It first initializes the profit, buy and sell days and min_index as 0 and min_price as the price of first element. Then, it iterates the list. If it encounters a price lower than the current min_price, it assigns it as the new min_price, but doesn't use it to find the maximum profit right away. Else, it calculates the profit from buying in the day of current min_price and selling it on the current day [i]. If it is higher than the current max_profit, it saves it as the new best profit. This way, in each step, the profit comes from buying on min_price of the days before and selling on current day is calculated and selling before buying situation never happens.

Here are the outputs of examples. It is the same as the previous one.

```
Part2.2: Maximum Profit in Linear Time

Prices: [10, 11, 10, 9, 8, 7, 9, 11]
Buy on Day5 for $7 and sell on Day7 for $11.

Prices: [100, 110, 80, 90, 110, 70, 80, 80, 90]
Buy on Day2 for $80 and sell on Day4 for $110.

Prices: [10, 7, 10, 11, 8, 10, 9, 11]
Buy on Day1 for $7 and sell on Day3 for $11.
```

Worst Case Time Complexity

In any case, it iterates the list once so the time complexity is O(n)

Question 2.c: Comparison

The first solution which uses divide and conquer approach has O(nlogn) time complexity in any case and the second solution which uses iteration has O(n) time complexity. Also, the first solution has the overhead of recursion space complexity. So, in any circumstances, the second solution is a more viable option.

Question 3: Longest Increasing Sequential Subarray with Dynamic Programming

The algorithm to find the longest increasing sequential subarray consists of one method called *longestIncreasingSubarray*. It first creates an array called memo with same length as the array and initializes it with 1s. Then, it iterates the list. If the current element is greater than the previous element, its value in the memo array is assigned to be one more than the previous element's. Finally, the maximum value in the memo array is the length of the longest increasing sequential subarray and returned.

Here are the outputs of the examples from PDF:

```
Part 3: Longest Increasing Sequential Subarray

Array: [1, 4, 5, 2, 4, 3, 6, 7, 1, 2, 3, 4, 7]
Longest Increasing Sequential Subarray Length: 5

Array: [1, 2, 3, 4, 1, 2, 3, 5, 2, 3, 4]
Longest Increasing Sequential Subarray Length: 4
```

Worst Case Time Complexity

In any case, it iterates the list once and finds the maximum element in the memo array, both of them taking linear time. So, the time complexity is O(n).

Question 4.a: Maximum Score with Dynamic Programming

It consists of a method called *maxScoreDynamic* which takes a 2D array as grid as input. It first creates two 2D arrays with the same size of the original grid. maximumScore[i][j] will store the maximum score that can be achieved when arriving at coordinate (i, j) and the path[i][j] will store the coordinates of the previous step in the path when arriving at coordinate (i, j).

The value of the starting cell is assigned as the maximum score of that cell. Then, it iterates through all the cells in the grid. In each iteration, it compares what would the maximum score arriving at that cell would be if it comes to that cell from left or above. It chooses the one that gives the greater result and updates the maximumScore and path arrays' cells accordingly.

After the loop is finished, it backtraces the scores and the optimal path, starting from the finish cell and returns the maximum score, the path and the points gained along the path.

Here are the example outputs. First example is from the PDF and the second example is randomly generated.

```
Part 4: Maximum Score
[25, 30, 25]
[45, 15, 11]
[1, 88, 15]
[9, 4, 23]
Solution 4.1: Dynamic Programming
Path: [(0, 0), (1, 0), (1, 1), (2, 1), (2, 2), (3, 2)]
Points: [25, 45, 15, 88, 15, 23]
Score: 211
```

```
[6, 1, 1, 8, 8, 9]
[3, 6, 7, 0, 4, 7]
[7, 1, 3, 1, 1, 8]
[7, 7, 4, 2, 4, 1]
[6, 3, 0, 6, 4, 2]
[2, 8, 2, 3, 7, 0]
Solution 4.1: Dynamic Programming
Path: [(0, 0), (1, 0), (2, 0), (3, 0), (3, 1), (3, 2), (3, 3), (4, 3), (4, 4), (5, 4), (5, 5)]
Points: [6, 3, 7, 7, 7, 4, 2, 6, 4, 7, 0]
Score: 53
```

Worst Case Time Complexity

In any case, assuming the grid has a size of nxm, the method first iterates each cell on the grid, which takes O(nm) time. Then, it backtraces to gather scores and the path, which takes O(n+m) time. So, the time complexity of the algorithm is O(nm).

Question 4.b: Maximum Score Using Greedy Algorithm

It consists of a method called *maxScoreGreedy*, which takes a 2D grid array as input. It initializes the score as 0. After that, in a loop until it reaches the finishing cell, it compares the next two cell that it can visit, and chooses the one with the greater value. If it can reached the border, it compulsorily chooses the only next available cell. The score, path and point lists are updated accordingly in each step and returned when the finishing cell is reached.

Here are the outputs with comparison to the previous solution:

```
Part 4: Maximum Score
[25, 30, 25]
[45, 15, 11]
[1, 88, 15]
[9, 4, 23]
Solution 4.1: Dynamic Programming
Path: [(0, 0), (1, 0), (1, 1), (2, 1), (2, 2), (3, 2)]
Points: [25, 45, 15, 88, 15, 23]
Score: 211

Solution 4.2: Greedy Algorithm
Path: [(0, 0), (1, 0), (1, 1), (2, 1), (2, 2), (3, 2)]
Points: [25, 45, 15, 88, 15, 23]
Score: 211


[4, 2, 10, 0, 8, 13, 6, 9, 6, 1]
[11, 13, 12, 12, 3, 9, 2, 8, 13, 5]
[14, 8, 8, 5, 11, 13, 8, 2, 14, 11]
[7, 6, 3, 13, 8, 6, 3, 14, 10, 7]
[5, 1, 2, 6, 0, 4, 2, 1, 5, 13]
[5, 4, 11, 6, 0, 13, 4, 2, 9, 10]
[14, 1, 9, 11, 13, 3, 5, 9, 5, 9]
[3, 5, 12, 0, 13, 14, 14, 4, 10, 7]
[12, 1, 4, 14, 4, 5, 2, 12, 7, 5]
[5, 4, 6, 4, 5, 7, 1, 14, 4, 8]
Solution 4.1: Dynamic Programming
Path: [(0, 0), (1, 0), (1, 1), (1, 2), (1, 3), (2, 3), (3, 3), (4, 3), (5, 3), (6, 3), (6, 4), (7, 4), (7, 5), (7, 6), (7, 7), (8, 7), (9, 7), (9, 8), (9, 9)]
Points: [4, 11, 13, 12, 12, 5, 13, 6, 6, 11, 13, 13, 14, 14, 4, 12, 14, 4, 8]
Score: 189

Solution 4.2: Greedy Algorithm
Path: [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (2, 3), (3, 3), (3, 4), (3, 5), (4, 5), (5, 5), (5, 6), (6, 6), (7, 6), (7, 7), (8, 7), (9, 7), (9, 8), (9, 9)]
Points: [4, 11, 14, 8, 8, 5, 13, 8, 6, 4, 13, 4, 5, 14, 4, 12, 14, 4, 8]
Score: 159
```

Worst Case Time Complexity

In any case, it either goes to the cell on the right or below in each step until it reaches the last cell. Assuming the grid has a size of nxm, that makes a time complexity of O(n+m).

Question 4.c: Comparison of the Solutions

Comparison of Correctness:

The brute-force solution and dynamic programming solutions always produce the correct output. They always find the absolute maximum score that can be obtainable.

However, the greedy algorithm solution doesn't always produce the absolute maximum score. There are 2^((n-1)*(m-1)) possible paths from start to finish but it only takes 2^(n+m) of them into account. So, as the grid size get larger, its possibility to find the absolute maximum score gets smaller. In the examples below, it found a correct maximum score for a grid of 3x4 but it couldn't reach the maximum score in a grid of 10x10.

Comparison of Time Complexities:

The brute force approach has a time complexity of O(n^2m^2) while the dynamic programming approach has a time complexity of O(nm). The reason why brute force approach grows so fast is because in each step of calculation, it calculates all the cells coming before that cell. In other words, it ends up calculating the same values over and over again.

However, in the dynamic programming approach, the maximum score that can be obtained at each cell is saved once calculated and when this value is needed again, the saved value is simply obtained rather than calculating again. This situation speeds up the process with the cost of using additional memory space.

The greedy algorithm looks to be the fastest solution among the three, with a time complexity of O(n+m). This is because it doesn't take all the possible paths into account. It just chooses one of the two options in each step. There is a probability that there are even better options at some other parts of the grid, and this probability increases as the grid size gets larger.