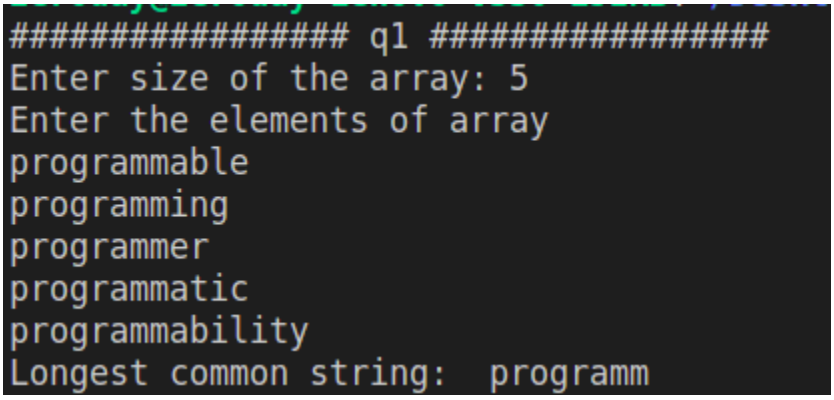
CSE 321 INTRODUCTION TO ALGORITHM DESIGN

Q1) The given algorithm is an implementation of the divide and conquer approach to finding the longest common substring of a list of strings. It does this by dividing the list of strings into two halves, finding the longest common substring of each half, and then finding the longest common substring of the two longest common substrings of the halves. This process is then repeated recursively on the halves until the base case of a list of one string is reached, in which case the only string in the list is returned as the longest common substring.

To analyze the worst-case time complexity of this algorithm, we can consider the number of recursive calls made. At each level of recursion, the list of strings is halved, so the number of recursive calls made at each level is halved. Therefore, the number of recursive calls made at each level can be represented by the

$n + n/2 + n/4 + n/8 + \ldots \Rightarrow n(1 + 1/2 + 1/4 + 1/8 + ...) \Rightarrow n(2) = 2n$

Therefore, the worst-case time complexity of this algorithm is O(n).



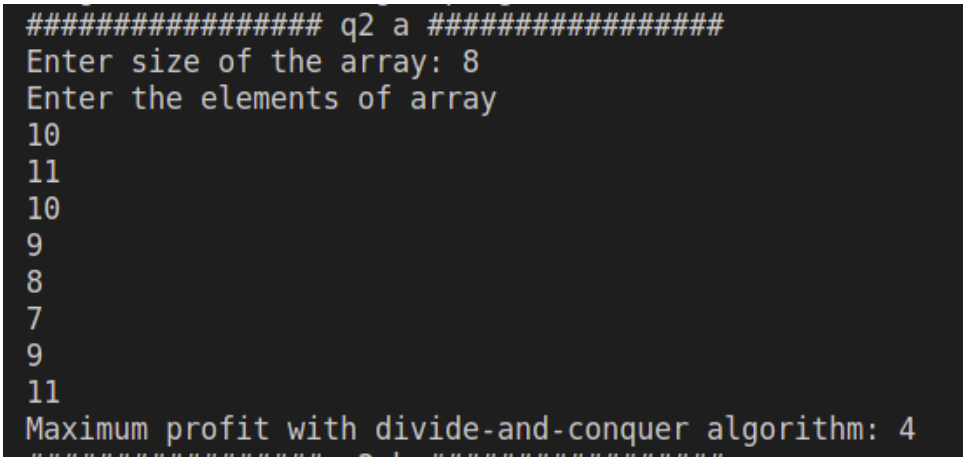
Q2)

a) The given algorithm is an implementation of the divide and conquer approach to finding the maximum profit that can be obtained from a list of stock prices. It does this by dividing the list of prices into two halves, finding the maximum profit of each half, and then finding the maximum profit that can be obtained by buying a stock in the left half and selling it in the right half. The maximum profit is then returned as the maximum of these three profits.

To analyze the worst-case time complexity of this algorithm, we can consider the number of recursive calls made. At each level of recursion, the list of prices is halved, so the number of recursive calls made at each level is halved. Therefore, the number of recursive calls made at each level can be represented by the

$n + n/2 + n/4 + n/8 + \ldots \Rightarrow n(1 + 1/2 + 1/4 + 1/8 + ...) \Rightarrow n(2) = 2n$

Therefore, the worst-case time complexity of this algorithm is O(n).



b) The given algorithm is an iterative approach to finding the maximum profit that can be obtained from a list of stock prices. It does this by keeping track of the minimum price seen so far and the maximum profit seen so far. It iterates through the list of prices and, at each step, updates the minimum price and the maximum profit if necessary.

To analyze the worst-case time complexity of this algorithm, we can consider the number of iterations performed. Since the algorithm iterates through every element in the list of prices, the number of iterations is directly proportional to the length of the list, n. Therefore, the worst-case time complexity of this algorithm is O(n).

```
################## q2 a ##################
Enter size of the array: 8
Enter the elements of array
10
11
10
9
8
7
9
11
Maximum profit with divide-and-conquer algorithm: 4
################## q2 b ##################
Maximum profit with not divide-and-conquer algorithm: 4
```

c) Both of the algorithms have a worst-case time complexity of O(n). This means that, in the worst case, both algorithms will take approximately the same amount of time to run, regardless of the specific inputs.

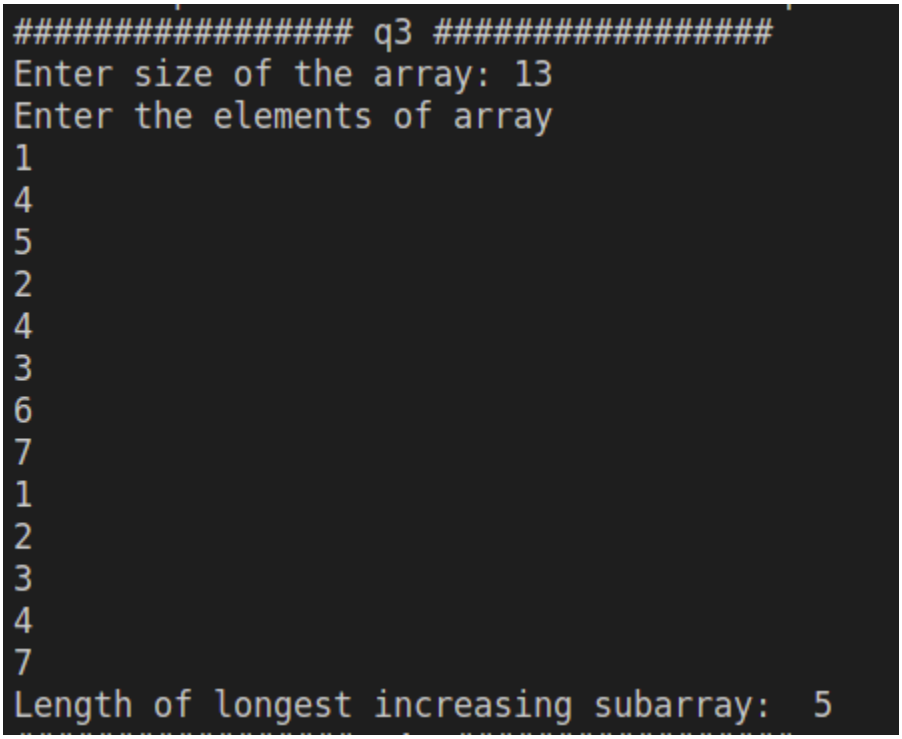
Q3) The given algorithm is an implementation of dynamic programming to find the length of the longest increasing subarray of a given array. It does this by iterating through the array and maintaining an array dp of the lengths of the longest increasing subarrays ending at each index.

To analyze the worst-case time complexity of this algorithm, we can consider the number of iterations performed. The inner loop iterates through all previous indices for each index i, which means that it performs a total of i iterations for each iteration of the outer loop.

Since the outer loop iterates through every element in the array, the number of iterations performed by the inner loop is equal to the sum of the first n positive integers:

n(n+1)/2

Therefore, the worst-case time complexity of this algorithm is O(n^2).



Q4)

a) Initialize a 2D array scores to store the maximum score that can be obtained from each position in the map. The array is initialized to have the same dimensions as the map, and all of its elements are set to 0.

Check if the map has only one row or only one column. If either of these conditions is true, then the maximum score can be obtained by simply moving right or down and collecting the scores along the way. In this case, the scores array is updated accordingly.

If the map has more than one row and more than one column, then update the first row and the first column of the scores array by moving right or down and collecting the scores along the way.

Iterate through the remaining elements of the map. For each element at position (i, j), update the corresponding element in the scores array.

Return the last element of the scores array as the maximum score. This element represents the maximum score that can be obtained by starting at the top left corner of the map and moving right and down until the bottom right corner is reached.

We can consider the number of iterations performed. The inner loop iterates through all elements in the jth column for each element in the ith row, and the outer loop iterates through all rows. Since the map is a 2D array with m rows and n columns, the number of iterations performed by the inner loop is n and the number of iterations performed by the outer loop is m. Therefore, the worst-case time complexity of this algorithm is O(mn).

b) The greedy approach is taken because, at each step, the algorithm chooses the path with the highest score, without considering the long-term effects of this choice. As for the reasoning behind the algorithm, the idea is to iterate through the map, starting at the top left corner, and choose the path with the highest score at each step. The down_score and right_score variables represent the scores that would be obtained by moving down or right, respectively. If the score from moving down is higher, then the algorithm moves down and updates the i and score variables accordingly. If the score from moving right is higher, then the algorithm moves right and updates the j and score variables accordingly. The loop continues until the bottom right corner is reached, at which point the score variable holds the maximum score that can be obtained by starting at the top left corner and moving right and down until the bottom right corner is reached.

To analyze the worst-case time complexity of this algorithm, we can consider the number of iterations performed. The loop continues until the bottom right corner of the map is reached, which means that it will perform a total of m+n-1 iterations, where m is the number of rows in the map and n is the number of columns. Therefore, the worst-case time complexity of this algorithm is O(m+n).

c) In terms of correctness, the first solution is guaranteed to find the maximum score, as it considers all possible paths that can be taken. However, the time complexity of this solution is very high(O(2^n)), making it impractical for large maps. The second solution is also guaranteed to find the maximum score. The time complexity of this solution is much better than the first solution(O(nm)), making it a more practical choice for large maps. The third solution is not guaranteed to find the maximum score, as it takes a greedy approach and does not consider the long-term effects of its choices. However, the time complexity of this solution is the best of the three(O(n+m)), making it the most efficient choice for finding the maximum score.

```
################## q4 a##################
Enter first dimension of the board: 4
Enter the board
25 30 25
45 15 11
1 88 15
9 4 23

maximum score with dynamic programming:  211
################## q4 b ##################

maximum score with greedy algorithm:  211
```