# CSE 321 - Homework 5

Due date: 08/01/2023, 23:59

1. Since a divide-and-conquer based solution is asked, we should divide the problem into sub-problems and solve them first. So we split the array into 2 parts and obtain the solutions for each. Then we merge these solutions and get the final result.

   At some point, the sub-problem becomes undividable, so we start from the beginning of the words and go until we see a mismatch. Then we return the part of the word (doesn't matter *which word* it is, because they have that substring in common) until the index that we stopped looking at (since we have seen a mismatch).

   In the worst case, all of the input strings are the same, so we have to check all characters of all strings to make sure. Therefore the worst-case time complexity is $O(n * m)$ where $n$ is the number of strings and $m$ is the length of the shortest word (because if the shortest length is $m$, we won't check the $(m + 1)^{th}$ character of any string even though they are long enough).

2. (a) A divide and conquer solution to this problem is to divide the array into halves and then solve them respectively.
   - The solution of the left half of the array gives us the maximum profit if we buy and sell in the first half of the days.
   - The solution of the right half of the array gives us the maximum profit if we buy and sell in the second half of the days.
   - But these solutions are not enough because we may get a larger profit by buying in the first half and selling in the second half. So we calculate this too.

   Finally, the maximum of these 3 values is the solution to the problem.

   The worst-case time complexity of this algorithm is $O(n * logn)$. $logn$ comes from dividing into parts (just as in binary search) and $n$ represents the traversing of each subarray.

   (b) This problem was solved during the PS in the class. Step-by-step solution and complexity analysis can be found in this file (page 37).

   (c) As seen, the divide-and-conquer solution is much slower than the other solution. Because the structure of the problem is not optimal for divide-and-conquer approach since the value we are looking at might be in the whole array (without dividing it). In such case, solving the halves is nothing but a waste of time because we cannot benefit from them.

3. Since we are asked to come up with a dynamic programming based solution, we should use memoization. In this problem, the memoization can be implemented as explained below:

- Generate an empty array (its length is the same as the length of the input array). This array will save the length of the maximal increasing subarray for each index. For example, if we call this array $dp\_map$, $dp\_map[4]$ should save the length of the maximal increasing subarray of the $input\_array[:4]$.
- To fill this map, we start from the left side. The first element of the map should be 1 ($dp\_map[0] = 1$) because $input\_array[:1]$ is an increasing array by itself.
- If the second element of the $input\_array$ is larger than the first one, then $dp\_map[1] = 2$ because we have an increasing subarray: $\{input\_array[0], input\_array[1]\}$ If $input\_array[1] \leq input\_array[0]$, then $dp\_map[1] = 1$ because we have 2 maximal subarray(one is $\{input\_array[0]\}$ and the other one is $\{input\_array[1]\}$ and their sizes are the same, 1.
- We keep doing this until we reach the end of $input\_array$.

Finally, the maximum element in $dp\_map$ is the solution.

4. (a) As in the previous question, we should keep a map to solve the problem by using dynamic programming. The map has the same size as the input. Since we get a $map$ as input let's call this second table that we will use to save some information $dp\_table$.

   $dp\_table$ will save the maximum score until a previous position. For example: $dp\_table[4][3]$ should save the maximum possible score from $map[0][0]$ to $map[4][3]$. To solve the problem, we will start from the bottom right and go to the top left.

   To calculate $dp\_table[n-1][m-1]$, where $n$ and $m$ are the size of the maps, we should know $dp\_table[n-2][m-1]$ and $dp\_table[n-1][m-2]$. Because if we are at the bottom right cell of the table, we either reached there from the left side or from the upper side (diagonal move is forbidden). So we calculate these two values. As it is seen, these values require some other values. So we recursively go until the top left of the map. In this way, the $dp\_table$ will be filled and the bottom right element of it will be the solution. Please check the .py file for a detailed explanation of the algorithm.

   The time complexity of such an algorithm is $O(n * m)$ because we are filling a table with a size of $nxm$, and we use previous solutions to calculate each cell. Thus, we don't make any unnecessary computations.

   (b) A greedy solution to this problem is pretty simple. At each step, we have 2 options for the next move. We compare the scores of these 2 options and go with the maximum one. The complexity of such algorithm is $O(n + m - 2) = O(n + m)$ because we have to take $(n - 1)$ steps towards the right side and $(m - 1)$ steps towards the downside to reach the bottom right from the top left.

   (c) In terms of time complexity, greedy solution is the best while brute force is the worst. But greedy algorithm will not always give the correct solution (there are two test cases in the .py file, please check them to see the difference). Therefore, the dynamic programming solution becomes prominent since it guarantees the correct solution and is much faster than the brute force solution.