CSE 321 - Homework 5

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

1. **15 pts.** Given an array of strings, design a divide-and-conquer algorithm that finds the longest common substring that is present at the beginning of all these strings.

   Example:
   Input: ["programmable", "programming", "programmer", "programmatic", "programmability"]
   Output: "programm"

   Example:
   Input: ["compute", "compatible", "computer", "compare", "compactness"]
   Output: "comp"

2. Assume you are a merchant. Your aim is to maximize your profit by buying some goods preferably at a low price and then selling them preferably at a higher price afterwards. Assuming that you know the market prices for a given period of time, your goal is to decide when to buy and when to sell in order to maximize your profit. Do not forget that you cannot sell before you buy.

   (a) **15 pts.** Design a divide-and-conquer algorithm to solve this problem.

   (b) **5 pts.** Design an algorithm that is not based on the divide-and-conquer approach to solve this problem in linear time.

   (c) **5 pts.** Compare these two algorithms in terms of their worst-case time complexity.

   Example:
   Input: [10, 11, 10, 9, 8, 7, 9, 11]
   (It means that the market price is 10₺ on Day_0, 11₺ on Day_1, ..., 11₺ on Day_7)
   Output: Buy on Day_5 for 7₺ and sell on Day_{11} for 11₺.

   Example:
   Input: [100, 110, 80, 90, 110, 70, 80, 80, 90]
   (It means that the market price is 100₺ on Day_0, 110₺ on Day_1, ..., 90₺ on Day_8)
   Output: Buy on Day_2 for 80₺ and sell on Day_4 for 110₺.

3. **20 pts.** Given an integer array, design a dynamic programming algorithm to find the length of the longest increasing sub-array. The elements of the sub-array should be consecutive in the main array. Make sure your algorithm does not compute the sub-problems which had been computed before.

Example:
Input: [1, 4, 5, 2, 4, 3, 6, 7, 1, 2, 3, 4, 7]
Output: 5 (the sub-array is [1, 2, 3, 4, 7])

Example:
Input: [1, 2, 3, 4, 1, 2, 3, 5, 2, 3, 4]
Output: 4 (the sub-array is [1, 2, 3, 4] or [1, 2, 3, 5])

PS: Printing the sub-array is not mandatory.

4. Remember the first problem of Homework 4.

Consider a computer game with a 2D map with axes $A(A_1, A_2, ..., A_n)$ and $B(B_1, B_2, ..., B_m)$. The goal is to start from A_1B_1, move step by step to arrive at A_nB_m, and reach the highest possible score. At each coordinate the player arrives, they gain a (positive) number of game points. Additionally, there is a rule that restricts the movements. If the player is at A_iB_j, their next move should be either A_iB_{j+1} or $A_{i+1}B_j$, and no other movement is possible.

Example:

Input: $n = 4, m = 3$
Game map:

	B_1	B_2	B_3
A_1	25	30	25
A_2	45	15	11
A_3	1	88	15
A_4	9	4	23

Output:
Route: $A_1B_1 \rightarrow A_2B_1 \rightarrow A_2B_2 \rightarrow A_3B_2 \rightarrow A_3B_3 \rightarrow A_4B_3$
Points: 25 + 45 + 15 + 88 + 15 + 23 = 211

(a) **15 pts.** Design a dynamic programming algorithm to solve this problem.

(b) **15 pts.** Design a greedy algorithm to solve this problem.

(c) **10 pts.** Consider the brute-force solution given in the solution file of Homework 4. Compare these three solutions in terms of their correctness and worst-case time complexity.

Important Notes

- Whenever you are asked to design an algorithm, implement your solution in Python3. Write a driver function to test each of these algorithms. Inputs should be randomly generated (by using *random* library) or taken from the user (you may assume that the inputs are proper). Gather all of the python code in a single .py file. Do not use external libraries or functions to implement a part of the solution. Pay attention to clean coding.

- Write a report explaining the reasoning behind the algorithms you coded and analyze the worst-case time complexity of each of them. This report should also include your answers to Question 2-c and Question 4-c. Write your report by using a program like MS Office and then convert it to a single PDF file. Pictures of handwritten works are **not accepted**.

- Upload two files only, a .py file and a .pdf file, **not a .zip or a .rar file**.

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.