QUESTION 1

In this question, first I have determined my base cases which are array has 0, 1 or 2 elements. If array has no elements, then common substring is empty; if array has 1 element, then common substring is that 1 element; if array has 2 elements, I compared all characters one by one to find the longest common substring.

If array has 3 or more elements, then I divide the array into 2 and recursively try to find the longest common substring in each 2 halves. After found these 2 substrings, I have compared them character by character to find the longest common substring.

The worst case time complexity of this algorithm is θ(n * m):

- n: number of strings
- m: length of the longest string

It is because all the strings in the array has been examined character by character so product of n and m gives us the worst case complexity (where all the strings are same).

QUESTION 2

(a)

In this question, I need to divide problem into sub-problems. For this, I have divided array into two and find the max difference in both sides. Then result is maximum of these two result or maximum of right minus minimum of left (for instance if we have array [1, 2, 3, 4, 5], then last scenario occurs where maximum element is at the right side of the array which is 5 and minimum element is at the left side of the array which is 1 then maximum difference is 4). Then we have two results again and maximum of these two results gives us the maximum difference in the array where smaller element occurs before the larger element.

While I am assigning maximum difference, I keep tuple and keep the index of both operands of subtraction. With this, I am able to find days of buy & sell.

(b)

At our first PS with Sibel Gülmez, we have solved the very similar question. The algorithm goes like this:

- Assign maximum element to last element
- Start from second last element of the array and goes till the beginning of the array
- If we found a new maximum element:
  - Assign maximum element to it
- If current element is not bigger than maximum element:
  - Subtract current element from maximum element and if it is bigger than the difference that is calculated before, we have our new maximum difference

I do it like this because from end to beginning, I keep the large numbers and whenever I see an element that is smaller than this large element, I subtract it from that large element. Then choose between the maximum difference that is calculated formerly and this new result of subtraction according to their values. With this way, I have calculated all the subtractions that is greater element comes after the smaller element and choose maximum of them.

Again, while I am assigning maximum difference and maximum element, I keep tuple and keep the index of both operands of subtraction. With this, I am able to find days of buy & sell.

(c)

For the part a:

- I am dividing the array into two equal halves each of size n/2
- I am finding min in the left side and max in the right side then find the max difference
- With these two, time complexity becomes:
  - $T(n) = 2T(n/2) + cn$
  - It is very similar to merge sort so worst case time complexity becomes:
    - $\theta(n\log n)$

- Each function call takes linear time while finding minimum and maximum, that is where n comes from
- Dividing array into two and calling function is where logn comes from

Part b's time complexity is θ(n-1) = θ(n) which is linear because we visit every element of the array except last element in the for loop.

In terms of worst case time complexity, part b seems better than divide & conquer solution.

QUESTION 3

For this problem, I have tried to prevent solving overlapping subproblems again and again and solve each of the subproblems only once and record the results. For this purpose, I have kept an array to record results for each of the elements. This array keeps the length of the sequence that element at the same index in the original array has. So for array [1, 2, 3, 4], my array for dynamic programming becomes [1, 2, 3, 4]. Each element in DP array keeps the length of the sequence number at that index has. I am doing this by traversing the array from beginning to end and examining each element if it is bigger than the element that comes 1 before. If so, I increment the DP array's relevant value. Finally, I return the biggest number in this array with its index to find the sub-array.

I have only traversed the array once so time complexity of this algorithm is linear.

QUESTION 4

(a)

For this problem, again I think how to design a dynamic programming algorithm by trying to prevent solving overlapping subproblems again and again and solve each of the subproblems only once and record the results. We have solved the knapsack problem at lectures with dynamic programming. I have been inspired by that solution. I am keeping a 2D table for the results whose elements are all 0 except the up-left corner of the given map which is A1 B1. This table keeps the corresponding maximum score for the map in the same index. At first, I have filled first row and first column by summing up the values. For the array below:

1	4	3
3	5	1
2	4	8

My table becomes like this at first:

1	5	8
4	0	0
6	0	0

Then I have filled the rest of the array by using a logic like this: At each iteration, I have determined the maximum score by looking one above and one left element, take the bigger one, add with the current element. So at the end, my table becomes like this:

1	5	8
4	10	11
6	14	22

Value at the bottom-right corner is the answer. Then I found the path by going backwards from 22 and find the previous element by subtracting the current value from 22.

Time complexity of this algorithm is determined by nested for loop which is θ(n*m) for both worst and best cases---> # of rows times # of columns.

(b)

While designing the greedy algorithm, I have tried to make locally best choice at each step without worrying about the future steps that I am going to do. As a nature of greedy algorithm, my algorithm does not consider long-term impact of given decisions. At each step, I determine if below element is bigger or right element is bigger and go to the direction that bigger element belongs to. Since we didn't think about long-term impacts, this algorithm may not always give correct results. For example, if next step occurs in a path that has maximum score but that next element is smaller than the other choice (right or below) then this algorithm isn't the best choice. On the other hand this greedy algorithm has the best time complexity which is θ(n+m) ---> # of rows plus # of columns. This complexity is because we follow a path by choosing maximum element at each step and this path's length is n+m.

(c)

Correctness

For example, let's say our 2D map is this:

1	2	5
3	2	6
5	4	8

BRUTE FORCE: Since I examined all the results and since I was going to both of the directions (right & below) and then choose the path that has maximum score, this brute force approach gives correct results. This algorithm choose this path that results in maximum score:

- 1 → 2 → 5 → 6 → 8 = 22

This is because every path has been tried and the path with maximum score has been chosen.

DYNAMIC PROGRAMMING: I was keeping a table for dynamic programming according to one previous step, which is either up or left, and filling the table in a way that results in getting maximum path with choosing the maximum of up and left elements for each of the index. As a result of this, bottom right cornerth element of the table gives me the correct result since it has the maximum score up to that element in the original array. This algorithm choose this path that results in maximum score:

- 1 → 2 → 5 → 6 → 8 = 22

This is because algorithm finds the path with the maximum score up to specific index for each element with the table it uses.

GREEDY: My steps at this algorithm can be considered as small. Long-term impacts hasn't been considered and at each step maximum of right and bottom elements is chosen. As a result of this, this algorithm cannot always give correct results. This algorithm choose this path that results in maximum score:

- 1 → 3 → 5 → 4 → 8 = 21

This is because while choosing second element, greedy algorithm chooses 3 since it is bigger than 2.

Worst-case Time Complexity

BRUTE FORCE: Its time complexity was $\theta(2^{n*m})$ which is very high. It is because game array's size is n*m. In each recursive call, the function can either go down or right, so, at each step, the number of possible paths doubles, leading to this complexity.

DYNAMIC PROGRAMMING: Itss time complexity was θ(n*m) which is better than brute force. Here, I didn't check every possible path. Instead, I have gone through the original array and keep a table to keep scores up to each index by looking possible one previous steps which are up and left and choose the maximum of them and add it to current index's value at original array. This improvement comes with a downside which is using extra space for table and it's complexity is also θ(n*m).

GREEDY: Its time complexity is best among three of them which is θ(n+m) because this algorithm follows only one path whose length is n+m but as I mentioned, this algorithm's correctness is not always guaranteed.