

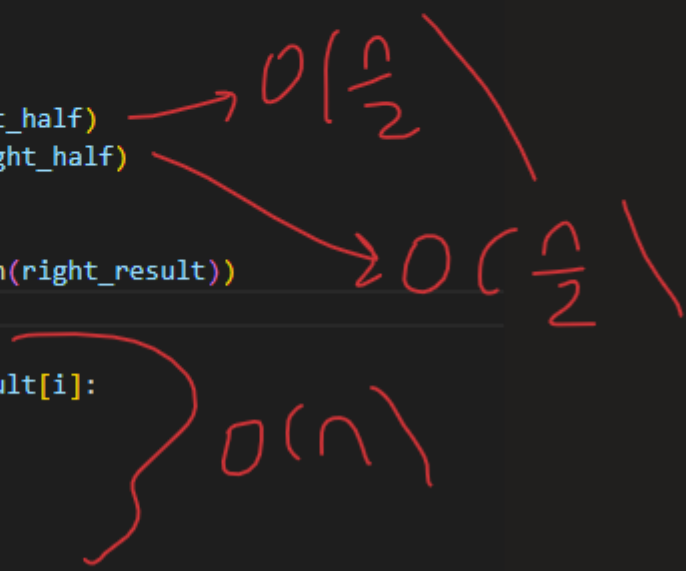
## Q-1

### Algorithm:

We will divide and conquer. We have arrays with strings in them. We go down by dividing this array. When there is 1 element left in each array, we start to go up. In the first step of the going up, we compare the characters of the 2 strings with each other to obtain a common string. In the next step, we get a new common string by comparing this common string with the common string from other split strings. this is the main idea.

### Worst Case Analysis:

```
def common_substring(strings):  
    if not strings:  
        return ""  
  
    # if we have only one string in the array, return it to compare  
    if len(strings) == 1:  
        return strings[0]  
  
    # split the array into 2 part  
    mid = len(strings) // 2  
    left_half = strings[:mid]  
    right_half = strings[mid:]  
  
    # get the right and left results  
    left_result = common_substring(left_half)  
    right_result = common_substring(right_half)  
  
    result = ""  
    min_len = min(len(left_result), len(right_result))  
    i = 0  
    while i < min_len:  
        if left_result[i] == right_result[i]:  
            result += left_result[i]  
            i += 1  
        else:  
            break  
  
    return result
```



Asymptotic notation is  $T[n] = 2 \cdot T[n/2] + n$

When we solve this recurrence relation, we will see that the result is:  $T(n) = n \log(n)$

So, the complexity is  $O(n \log n)$ .

It is always  $n \log n$  since there is no stop condition.

## Q-2.a

### *Algorithm:*

This question can be exploited. And, I am sorry, I did it. We have only 1 integer array. There is a similar problem in the book, and I think it was inspired by it. This problem is Closest-Pair problem. But, in that question we have 2 array and each element of arrays has (x,y) pairs. However, we have only one integer array in this question. You have asked us to find furthest pair. But there is only one array. If we sort this array, then only we must do is get first and last element of sorted array. Moreover, we must keep the index of the furthest pairs. So, I convert array to a dictionary. Keys will be indexes and values will be values (This operation linear). After that, I convert the map to a tuple.

The solution must be divide and conquer algorithm as you wanted. So, I use Merge Sort to sort this tuple. I modify the Merge Sort so that it sorts according to values of tuple. Merge Sort sorted the tuple. That's it! I use divide and conquer algorithm and I found the furthest pair in the list. Regards.

### *Worst Case Analysis:*

As I told, converting list to map and map to tuple is linear. Merge sort is  $O(n \log n)$ . So, total complexity of this algorithm is  $O(n \log n)$

## Q-2.b

### *Algorithm*

It is very simple algorithm. I iterate the array's elements in a single loop and found the maximum and minimum element's indexes and return them. So, I found the targets!

### *Worst Case Analysis:*

```
def max_difference(arr):
    min_element = arr[0]
    max_element = arr[0]
    min_index = 0
    max_index = 0
    for i in range(1, len(arr)):
        if arr[i] < min_element:
            min_element = arr[i]
            min_index = i
        if arr[i] > max_element:
            max_element = arr[i]
            max_index = i
    return (min_index, max_index)
```

There is a single loop and it runs as many as element of array. So, the complexity is linear =  $O(n)$

It is always linear since there is no stop condition to stop the loop.

## Q-2.c

The second solution is faster since it is linear. The first solution uses Merge Sort and it is  $O(n \log n)$

The problem is simple. All we must do is find max and min elements. So, second solution is more appropriate for this problem. This problem is so simple to apply divide and conquer algorithm. No need.

## Q-3

### *Algorithm*

Dynamic programming is a technique for solving problems by breaking them down into smaller subproblems and storing the solutions to these subproblems in an array or similar data structure, so that they can be reused (rather than recomputed) when needed.

In this problem, we have an array initial 0's named *length* for storing. We are moving along the main array trail. If there is an increasing process, we increase the index value corresponding to the index in the main array we are looking at by 1 in the *length* array. For example, if our main array is [6, 8, 10, 0], our *length* array is [1, 2, 3, 1]. In summary, the function stores the length of the longest increasing subarray ending at each index in the *length* array.

End the end of the function, we find the maximum element of length array to find the longest subarray.

### *Worst Case Analysis:*

This problem is perfect for dynamic programming. It is only linear. We iterate a single loop as many as element number of array. So, the complexity is  $O(n)$ .

There is no stop condition to stop the array. Complexity is always linear.  $O(n)$

## Q-4.a

### *Algorithm*

It is showed in the textbook Ananay Levitin – Introduction to Algorithm Design and Analysis of Algorithms page 289. There is a pseudo-code there. I implement the python by help of it.

It creates an array named F and stores there the maximum points. Calculating the maximum points is simply :

$$F(i, j) = \max\{F(i - 1, j), F(i, j - 1)\} + c_{ij} \quad \text{for } 1 \leq i \leq n, \quad 1 \leq j \leq m$$
$$F(0, j) = 0 \quad \text{for } 1 \leq j \leq m \quad \text{and} \quad F(i, 0) = 0 \quad \text{for } 1 \leq i \leq n,$$

Firstly, it initializes the row 0 and column 0. After that it iterates all table and find maximum points by looking up and left cells. It makes an addition  $\max(\text{left}, \text{up}) + \text{cell}$ .

It is a clear dynamic programming algorithm. It uses an array to store all maximum points. It goes from the beginning [0][0] to end [n][m] by makes additions in that store array. At the end, result is in the index [n][m]

### *Worst Case Analysis:*

It has a nested loop. First loop runs n time, the second one runs m time. So, the time complexity is  $O(n*m) = O(n^2)$

Algorithm is always quadratic.  $\Theta(n^2)$

## Q-4.b

### Algorithm

As the name suggests (greedy), this algorithm constantly goes to the place where it thinks it will give more points. Isn't very smart. The algorithm is very simple, at each step we look to the right or down and go to the one with an excess of points. obviously, it doesn't seem like a suitable algorithm for this problem.

### Worst Case Analysis:

```
def max_score(arr):  
    n = len(arr)  
    m = len(arr[0])  
    x = 0  
    y = 0  
    max_score = arr[0][0]  
    while x < n-1 or y < m-1:  
        if x < n-1 and (y == m-1 or arr[x+1][y] > arr[x][y+1]):  
            max_score += arr[x+1][y]  
            x += 1  
        elif y < m-1:  
            max_score += arr[x][y+1]  
            y += 1  
    return max_score
```

It traverses all table. It is  $O(n*m) = O(n^2)$

Quadratic Algorithm. It is always  $O(n^2)$

## Q-4.c

This problem is not suitable for *greedy* algorithms. The example in the PDF turns out to be correct by chance. In this problem, it will often give the wrong output. It works with Quadratic Complexity.

*Dynamic programming* is quite suitable for this problem. It gives correct result. The coding is simple, understandable. It works with Quadratic Complexity.

We can say *that Brute-Force* will definitely find it right. But it's very difficult to understand. Complex. Since path is also requested in that problem, this complicates the problem even more. Uses recursion. Although it seems to be in the same complexity as the others, it is more inefficient.