Problem4 (Programming Assignment)

1.Divide and Conquer algorithm

I have an array which contains a sequence of interest of rates and my goal is to find a maximum contiguous subarray from the given array. And result is the sum of maximum subarray

I use same algorithm with homework3's problem 1. I divide the array by three parts and compare them to find the maximum part. Then there will be three cases:

Case1. Maximum sum array can come from left half
Case2. Maximum sum array can come from right half
Case3. Maximum sum array can come from central part, which cross check both left and right part of subarray.

Thus, the procedure of divide and conquer algorithm is:

Step1. Find middle point and separate array into two by this middle point
Step2. Recursively split the left half of the array until we reach the base case and achieve maximum sum of left half.
Step3. Recursively split the right half of the array until we reach the base case and achieve maximum sum of right half.
Step4. Find the maximum sum of central part, which cross check both right and left part of array
Step5. Find the maximum sum among left, right, and central sub array

Time Complexity: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$, I can get $O(n\log(n))$ by master theorem.
Space Complexity: No extra space required apart from making recursive stack calls, so from a computation perspective it is O(1).

Code for Divide and Conquer

(1).Cross check (Central subarray's maximum sum)

```
def maxCrossingSum(self,arr,l,m,h) :
        # Include elements on left of mid.
        s_l = 0
        left_sum = float('-inf')
        for i in range(m,l-1,-1) :
            s_l = s_l + arr[i]
            if (s_l > left_sum):
                left_sum = s_l
                start_ind =i
        # Include elements on right of mid
        s_r = 0; right_sum = float('-inf')
        for i in range(m + 1, h + 1) :
            s_r = s_r + arr[i]
            if (s_r > right_sum) :
                right_sum = s_r
                end_ind = i
```

```
        # Return sum of elements on left and right of mid
        return left_sum + right_sum, start_ind+1, end_ind+1
```

Here, python's index start from 0. Thus, I add one to both start_ind and end_ind (start_ind +1, end_ind+1)

(2).Find maximum subarray (Recursive function)

```
def max_sub_array(self,arr,l,h):
        if l == h:
            return arr[h], l,h
        m = (l+h)//2
        left_sum,left_start,left_end = self.max_sub_array(arr,l,m)
        right_sum,right_start,right_end = self.max_sub_array(arr,m+1,h)
        overlapping_sum, cross_start, cross_end = self.maxCrossingSum(arr,l,m,h)

        i,j,max_sum = 0,0,0

        if left_sum > right_sum:
            i = left_start
            j = left_end
            max_sum = left_sum
        else:
            i = right_start
            j = right_end
            max_sum = right_sum
        if overlapping_sum > max_sum:
            i = cross_start
            j = cross_end
            max_sum = overlapping_sum
        return max_sum,i,j
```

I use almost same algorithm (code) with HW3's problem1

2.Dynamic Programming algorithm

I just follow the algorithm in homework. For a given day j, maximum interest rate that Amrita can obtain up till that day is either 0 i.e. sum of interest rates up till j was negative or sum of interest rates till j-1 day plus interest rate for j. Then Here is recursive relation:

$$S[j] = \begin{cases} 0 & if\ j = 0 \\ \max\{S[j-1] + a_j & otherwise \end{cases}$$

In order to implement algorithm, I used a bottom-up method rather than top-down. I calculated the 'maximum_sum_ending_here' for each index and use this value for calculating the 'maximum_sum_so_far' at each iteration. In this way, I need only two variables which I can use throughout my iterations and can get the result in one pass.

A minor difficulty for the algorithm is required to find the indices for the contiguous array which gives me the maximum sum. My temp_starting index will be updated each time current element is more than the maximum_sum_ending_here + nums[i], which means for any element

if my maximum_sum_ending_here becomes negative then I updated temp_starting index. Thus, I can set actual starting index to temp_starting index and end index to current index, whenever maximum_sum_ending becomes larger than or equal to maximum_sum_so_far,

Time Complexity: Time complexity is O(n)

Space Complexity: I think constant space, O(1)

Code for Dynamic programming

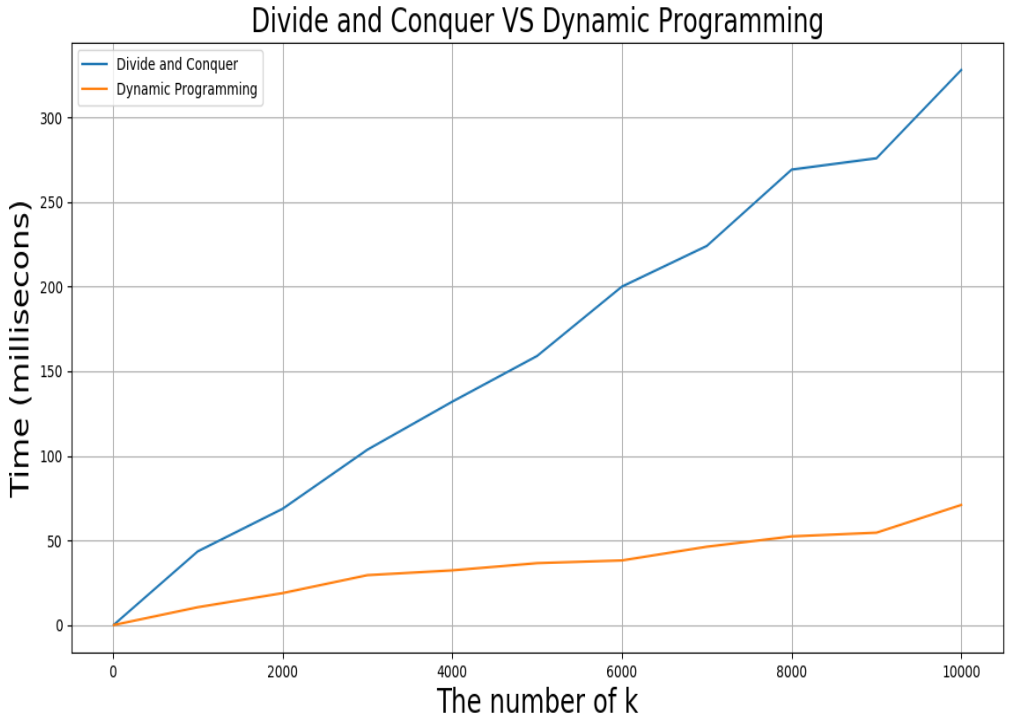
```
def get_max_arr(self,nums):
        max_till_here = nums[0]
        max_total = nums[0]
        temp_start = 0
        start = 0
        end = 0

        for i in range(1,len(nums)):
            if nums[i] >= max_till_here+nums[i]:
                temp_start = i

            max_till_here = max(nums[i],max_till_here+nums[i])

            if max_till_here >= max_total:
                start = temp_start
                end = i

            max_total = max(max_total,max_till_here)

        return max_total,start+1,end+1
```

Here, python's index start from 0. Thus, I add one both start and end (start +1, end +1)

<Next Page for Plot graph>

3.Plotting result (Divide and Conquer and Dynamic programming)

Above section, I mentioned the time complexity, for example the time complexity of divide and conquer is O(nlog(n)) and the time complexity of dynamic programming is O(n). Following graph's trend is same with theoretical time complexity. That is, theoretical and empirical trends are same:



<Figure 1. Time complexity of divide & conquer, dynamic programming>

Here, blue line is divide & conquer and orange line is dynamic programming. As you can see, as the number of k increases, both algorithms' running time also increase. In addition, the running time of divide & conquer (O(nlog(n))) is larger than that of dynamic programming (O(n)). That is, the running time of divide and conquer << the running time of dynamic programming.

My zip file has graph.py named Plot_divide_dynamic.py. Thus, if you want, you can check the graph

4.Refrence

Hongsup OH hw3 problem1 for divide and conquer algorithm (I used almost same algorithm).

CSE6140 lecture slide 06 for divide and conquer.

CSE6140 lecture slide 07, and 08 for Dynamic programming.