Homework 3

Bhanu Verma

Question 2: Help Amrita in buying the car

***Divide & Conquer Algorithm***: Before we start looking at the solution, let’s describe the problem first. We are given an array which contains a sequence of interest rates and we need to find a maximum contiguous sub-array from the given array that would give us the maximum sum.

If we think intuitively, there can only be three cases that will give us the result while we are splitting up our array into equal parts:

Case 1: Maximum sum sub-array will come from left half

Case 2: Maximum sum sub-array will come from right half

Case 3: A piece of maximum sum sub-array will be come from the left and other piece will come from the right. This means that middle point will be contained in the final result.

So our divide and conquer algorithm can be broken down into following steps:

* Find middle point and split array into two halves using that middle point.
* Recursively split left half till we reach the base case and get the final sum for left half, called left\_sum.
* Recursively split right half till we reach the base case and get the final sum for right half, called right\_sum.
* Find the maximum sum from the sub-array overlapping both the halves, called overlapping sum.
* Find the maximum sum from these three values and return that sum while moving up.

A minor addition is required to this approach to return indices; we’ll need to return indices as well along with sum while we are moving up in our recursive algorithm.

***Time Complexity***: T(n) = 2T(n/2) + O(n), using master theorem this comes out to be O(n\*logn).

***Space Complexity***: No extra space required apart from making recursive stack calls, so from a computation perspective it is O(1).

***Dynamic Programming Algorithm***: a DP approach for the given problem has already been provided in the homework itself. Let’s have a look at this approach.

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. Let’s write that down in terms of recurrence relation:

S[j] = 0 , if j=0

= max(S[j-1]+aj, 0) , otherwise

From an implementation perspective, I have used a bottom-up approach instead of top-down. For the bottom-up approach, we would calculate the ‘*maximum\_sum\_ending\_here’* for each index and use this value for calculating the ‘*maximum\_sum\_so\_far’* at each iteration. This way we would need only two variables that we can use throughout our iterations and can get the result in one pass.

A minor tweak to the algorithm is required to find the indices for the contiguous array which gives us the maximum sum. Our *temp\_starting* index will be update each time our current element is more than the *maximum\_sum\_ending\_here + nums[i]*, which means for any element if our *maximum\_sum\_ending\_here* becomes negative then we update our *temp\_starting* index.

Now, whenever our *maximum\_sum\_ending\_here* becomes greater than or equal to *maximum\_sum\_so\_far*, then we set our actual starting index to temp\_starting index and end index to current index.

***Time Complexity***: As we are getting the result in one pass, our time complexity is O(n).

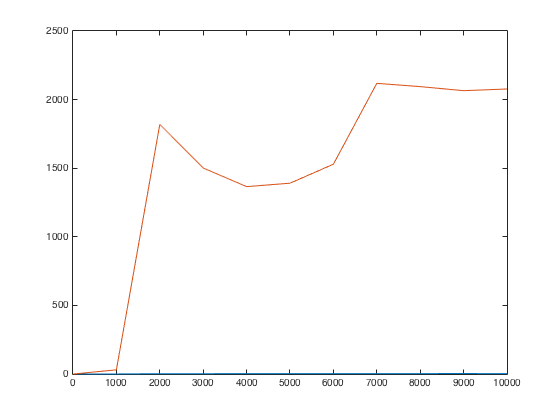
**Space Complexity**: Constant space, hence O(1).

**GRAPH PLOTTING**: Dynamic Programming (Blue) vs Divide & Conquer(Orange)

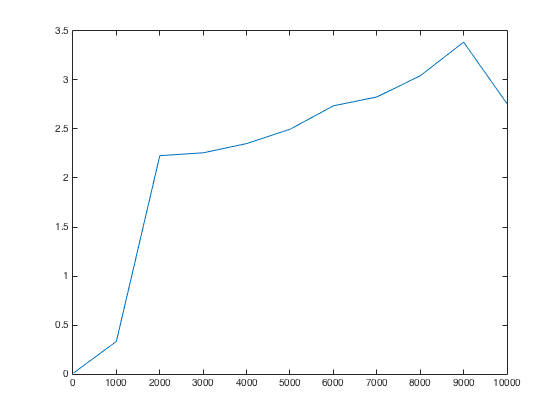
As you will see in the graph below, plots for the graphs for both the algorithms are expected. As divide and conquer algorithm is O(n\*logn), it’s time complexity increases as the value of n increases and that increase is more than the increase for dynamic programming algorithm’s time as time for DP algorithm is linear. This is the reason we are not able to see the time for DP clearly on the graph as time for dp << time for dc.

Y Axis – Time in milliseconds

X Axis – Number of instances



Here is the graph plot of time for dynamic programming algorithm vs number of instance:



As we can see from the plots above that DP algorithm is a lot faster in comparison to Divide & Conquer approach and the difference in time keeps on increasing as we keep on increasing the problem size.