Lab 2

Cache Angus

20000629

CMPE 362

September 28, 2018

## Question 1:

When implementing a solution that uses less than the ‘obvious’ O(n^2) complexity, I was able to utilize the combination of divide and conquer with the return method of a tuple. The given array of values was put into a function that accepted the array and the starting and ending indexes of the array. This then found the middle point of the array (and truncated it). Because this is a recursive function and the purpose of this function, named actualMaxArrayVal, is to divide the input array, there was an if statement put in to say that if the array has been broken down to its basic elements, then simply return that element. Otherwise, the function was to be called again, twice, with the left, right, and current values. These calls, coupled with the function to find the actual maximum sum, were then maxed using the built-in python function max, so only the maximum sum of the entire input (and its array indexes) was returned.

To find the maximum, the function findSum was called. This function separated the given array into its left and right components and then added the values of these components in a sum. The left section ran from left to right—and the right from right to left—to check for negative inputs or inputs that would decrease the sum. By running in this direction, it would check that the value being put in would not decrease the current sum, and if it did, it would not replace the sum any further and would call the next index. The position of the newly added value was also returned after it was checked that it would increase the sum. The new sum and the right and left positions are what the max in actualMaxArrayVal returns.

The complexity of this solution is O(nlogn). The O(logn) comes from the division and recursion of the problem, as there is a logarithmic way to break up the array and it happens recursively at O(1). The O(n) factor comes from the n times the value must be checked and returned. This accounts for the n-sized length of whatever the input array is.

To test this code, various sized or arrays were called, as well as arrays with only a single positive value to arrays where the entire array itself was the maximal subarray. This test all yielded the correct results and can be seen in the code file.

## Question 2:

To find the two maximal non-overlapping segments of the input array, the simplest way to implement was to replace the values within the input array itself after the indexes had been found. As the indexes are called into a new array, they are also called in the input array and the values are replaced with -1000. This value was chosen because within the test cases used, the values are never less than -1000 when added together. The -1000 simply ensures they will never be part of the newest maximal sum.

The complexity of this question is also O(nlogn) as the only addition to the previous code was the added line of inputArray[i] = -1000. This addition does not add any higher factor of time to the code than is already used.

## Question 3:

To find the two largest segments that are overlapping, the sum value could be returned to a separate global variable after it is returned. As the array is input into the code again, the new sum is checked against this previous sum. If the sums are equal, then it is the same subarray. Once it is know that it is the same sub array, the code can then shift its indexes so as to not include on of the values originally used and to include one new value. If this new sum also happens to be the same as the previous maximum sum, the new sum is then replaced at a global level and the new array with that sum can then be returned.

The complexity of this is O(nlogn) also as there is an additional loop within findSum, but it is a O(1) complexity loop.