Andrew McCann

CS350

Assignment #2

1. a) θ(n2) brute force algorithm for finding maximal subarray sum.

**Input**: A one dimensional array, ‘array’, of length ‘n’.

**Output**: the sum of the contiguous subarray of numbers with the largest sum.

func maximal\_sum(array)

max\_subarray = 0

for i = 0 ... n-1

current\_subarray = 0

for j = i ... n-1

current\_subarray += array[j]

if current\_subarray > max\_subarray

max\_subarray = current\_subarray

return max\_subarray

Complexity Class:

b) θ(nlog n) algorithm for finding maximal subarray sum.

Notes: I use iteration over the array instead of a for loop. The syntax is more easily readable. Just know each value is the value represented within the array, not the index.

**Input**: A one dimensional array, ‘array’, of length ‘n’.

**Output**: the sum of the contiguous subarray of numbers with the largest sum.

LOW\_VAL = lowest value we are representing in array

func merge\_sum(array)

if n == 1

return array[0], array

mid = n//2

left\_array = array[:mid] # not inclusive

right\_array = array[mid:]

l\_max, l\_result\_array = merge\_sum(left\_array)

r\_max, r\_result\_array = merge\_sum(right\_array)

merged\_array = l\_result\_array + r\_result\_array #Concat

left\_sum = 0

left\_max = LOW\_VAL # Lowest value possible

for value in reversed(l\_result\_array) # note reversed

left\_sum += value

if left\_sum > left\_max

left\_max = left\_sum

right\_sum = 0

right\_max = LOW\_VAL # Lowest value possible

for value in r\_result\_array # not reversed

right\_sum += value

if right\_sum > right\_max

right\_max = right\_sum

spanning\_total = right\_max + left\_max

left\_right\_max = max(l\_max, r\_max)

real\_max = max(left\_right\_max, spanning\_total)

return real\_max, merged\_array

Complexity Class:

Virtually identical to MergeSort. We split the array in two, yielding b of 2. We have to solve for every part we divide into, yielding a of 2. There are three cases we have to cover once the array has been split. Sum is within the left half, right half, or spans the middle and it in both. Left and right is easy to check as it is returned, but we have an runtime solution to checking that spanning case. That section is going to run a number of times, but it should never enter the n2­­­ complexity class.

c) Correctness: The start is pretty straightforward. Style of MergeSort. Split the array over and over until we get down to single units. Now we come to the three cases:

The left side encompasses the entire subarray,

The right side encompasses the entire subarray, or

The subarray spans across the middle of both.

To cover the first two is trivial, we just compare them. Comparing against a spanning value is more challenging since we have to start from the inside of the two pieces returned from the previous calls. To accomplish this, I loop through the left half from the end. It could be implemented a number of ways, iterating from the length to zero, or in my case (Python) calling reversed on it and iterating through it like normal. This could feasibly increase runtime [in python], but it’s simply an implementation representation to iterate through from its end, to the 0th. Once we’ve locked in the maximal sum on the left, we move to the right side, iterate through is saving from 0th position outward.

Once we have the maximum values coming from the middle of our soon-to-be-combined array we can add those two, get the max of the three values, and then return that value, with the combined array.

d) This should be able to expand out to a second dimension conceptually. It sounds like it would be a pretty insane order of growth though. It would be necessary to check the maximal subarray (1st dimension), followed by calculating the maximal subarray in the 2nd dimension of that initial subarray. It seems like this would escalate quickly into another 1st dimension traversal, causing the cascading effect that would cause an absolutely insane growth.

It I believe a similar solution would be to use dynamic programming to store an extra set of the current maximum values through some of that fancy new dynamic programming. I still don’t see a way to get it down to a ‘good’ runtime since all these matrix operations are pretty expensive.