Richard Moot
CS325
Group 19
July 11, 2016

Project 1

Theoretical Run-time Analysis

Algorithm 1: Enumeration

This algorithm is simply a brute force way of eventually finding the sub array with the largest sum by computing all possible sub-arrays.

Pseudocode:

```
Enumeration_algo (Array[1…N])
    For each pair (I, j) with 1 ≤ I < j ≤ n
        Compute a[i]+a[i+1]+…+a[j-1]+a[j]
        Keep max sum found so far
    Return max sum found
```

This algorithm shows to be $O(n^3)$ due to its depth of looping through all values and calculating the sum.

Algorithm 2: Better Enumeration

This algorithm is a minor improvement on regular enumeration but takes advantage of avoiding calculating the sum over and over again and simply using the same from the last time it was calculated and adding the next number.

Pseudocode:

```
Better_Enumeration_algo(Array[1…N])
        For i = 1, …, n
                Sum = 0
                For j = i, …, n
                        Sum = sum + a[j]
                        Keep max sum found so far
        Return max sum found
```

This algorithm is $O(n^2)$ since it doesn't require an additional run to get the sum.

Algorithm 3: Divide and Conquer

This algorithm is recursive and works on the idea to split the array in half to find if the largest sum sub array is located in either the first half, the second half, or a combination of parts of both. MaxSuffix in the pseudocode below can just be understood as another recursive way to get the max sum sub array from part of the first half and then part of the second half. Whichever of the three should contain the largest sum sub array.

Pseudocode:

```
Divide_&_Conquer(Array, Start, End)
        If start == end
                Return Array[start]
        Middle = start + end / 2
        Return Max of (Divide_&_Conquer(Array, start, Middle),
                        Divide_&_Conquer(Array, middle +1, end),
                        MaxSuffix(Array, start, middle, end))
```

This algorithm is O(n log n) since it can split the work into two, but still requires n time since the suffix must still be calculated.

Algorithm 4: Linear-time

This algorithm works on the notion that either the max sub array contains the element you are currently scanning or it does not. You are progressively returning the maximum between your current value and your current sum + your current value.

Pseudocode:

```
Linear_Algo(Array[1…n])
    Max_ending_here  = max_so_far = A[1]
    For elements i in A
        Max_ending_here = max of A[i] or max_ending_here + A[i]
        Max_so_far = max of max_so_for or max_ending_here
    Return max_so_far
```

This algorithm run in O(n) time since it only requires a single scan through the provided data to calculate the max sum subarray.

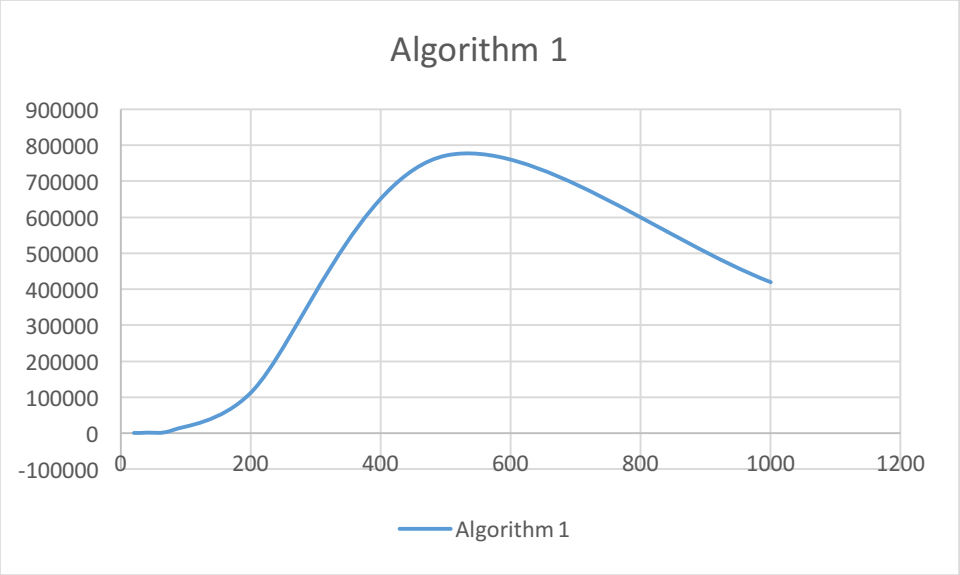
Testing

Using the provided test values, I had tested my implementations through a lot of trial and error. I then proceeded to generate my own larger and larger data sets and compares all values between algorithms to ensure that I would have consistent results throughout.

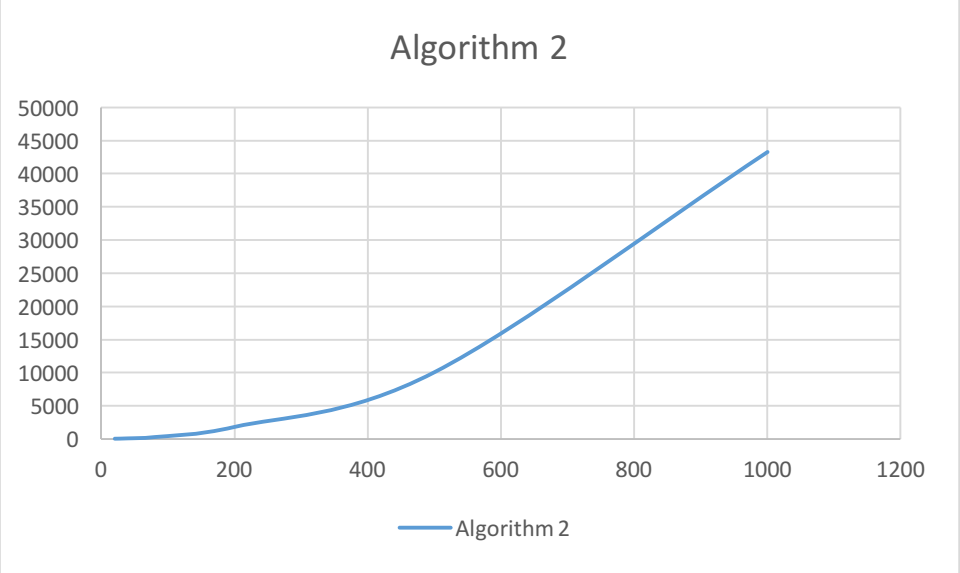
Experimental Analysis

All times shown are in milliseconds. The below table shows the average run times of the various algorithms that were run 10 times for each value of N to get the average.

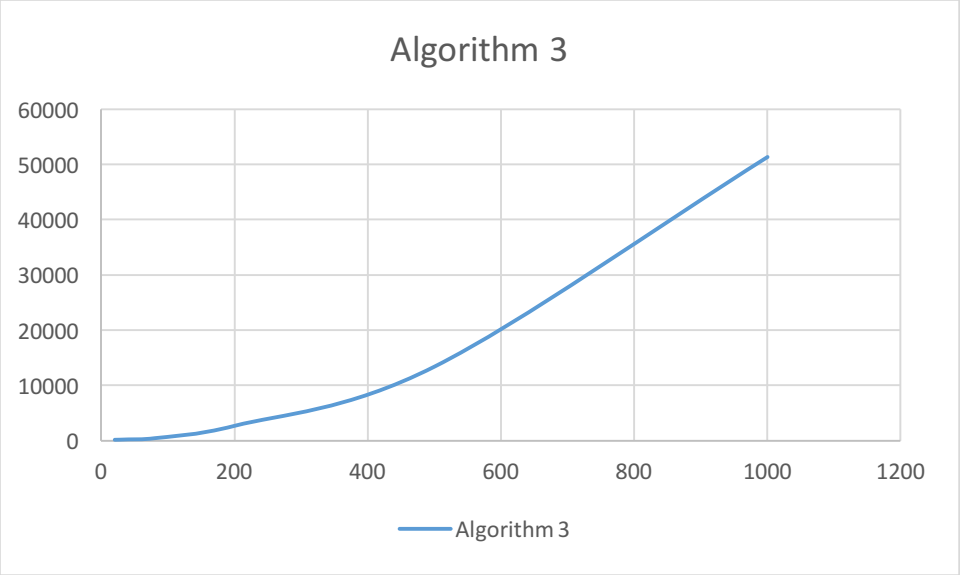
n	20	40	80	200	500	1000
Algorithm 1	**192**	**1123**	**8219**	**112278**	**770916**	**419102**
Algorithm 2	**26**	**79**	**275**	**1808**	**10065**	**43302**
Algorithm 3	**216**	**286**	**506**	**2741**	**13434**	**51307**
Algorithm 4	**66**	**21**	**39**	**165**	**243**	**997**



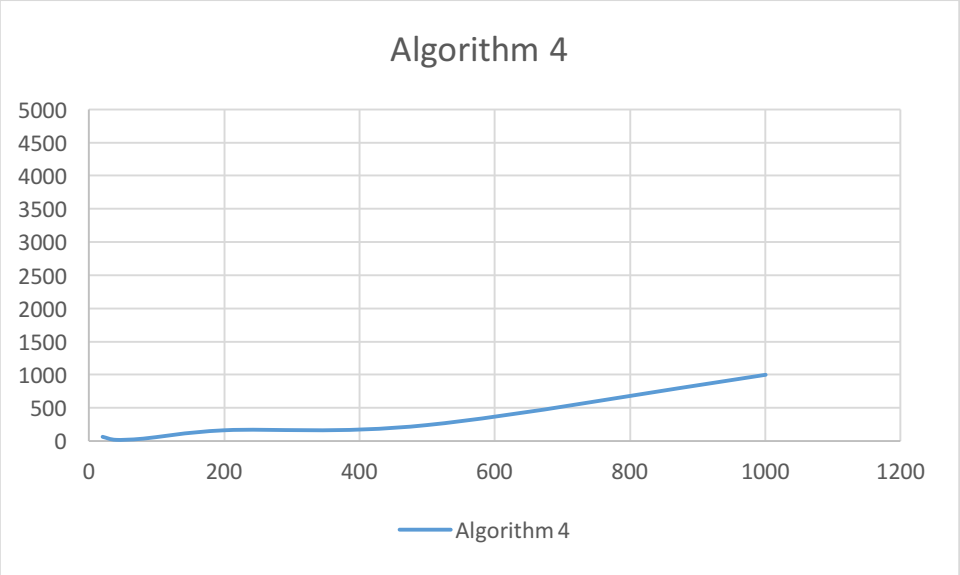
Algorithm 1 appears to display an exponential curve but then starts to decrease as N grows larger. There could be an enumeration optimization taking place here that is allowing the system to more rapidly calculate the max sum subarray.



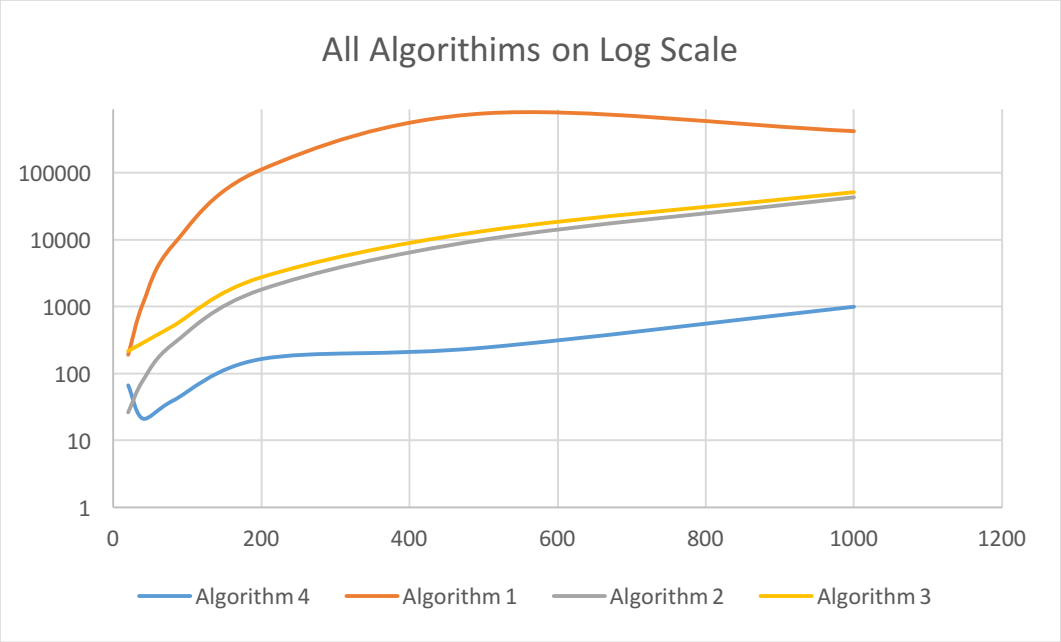
Algorithm 2 has a shallow quadratic curve that could likely continue to curve for higher values of n.



Algorithm 3 is almost exactly like Algorithm 2 and for some reason is a bit slower. This might be because the recursive functions were not structure to be able to be tail-optimized, so the execution might still take just as long when returning out of stack frames.



Algorithm 4 was clearly the fastest with an obvious linear curve. Even for very large values of N, this algorithm is unhindered in finding the max sum sub array.



Looking at all 4 algorithms on a log scale, you can see they have similar curves and all generally have the same progression but at different scales. Algorithm 1 being extremely slow shows it growing quite rapidly as N grows. You can also see that Algorithm 2 and 3 have a near same progression, with Algorith 4 growing the least.