Project 1 Report
Maximum Sum Subarray
CS325

<u>Group 11</u>

Kyle Livermore

William George

Karen Thrasher

Theoretical Runtime Analysis

Algorithm 1: Enumeration

Pseudocode:

```
MaxSubarrayEnumeration(array[1...n])
    for i, where 1<=i
        for j, where i < j <= n
            compute all array[i]+array[i+1]+...+array[j-1]+array[j]
            keep highest maximum sum
    return max sum
```

Asymptotic Analysis:

There can be $O(n^2)$ pairs and the computation of the sum of each pair will take $O(n)$ time. So the time for this algorithm will be worst case $O(n^2) * O(n) = O(n^3)$ time and have an average time of $\Theta(n^3)$.

Algorithm 2: Better Enumeration

Pseudocode:

```
MaxSumEnumeration2(array[1...n])
    for i from array beginning →  end
        currSum = array[i]
        currentArrayStart = i
        if currSum > maxSum
            maxSum = currSum
            maxSumStart = maxSumFinish = i
        for j from i → end - 1
            currSum += add array[j+1]
            if currSum > maxSum
                maxSum = currSum
                maxSumStart = currSumStart
                maxSumFinish = j+i+1
    return MaxArray from MaxSumStart → MaxSumFinish, maxSum
```

Asymptotic Analysis:

Because this algorithm only iterates over i O(n) times and iterates j O(n) times, then uses constant time to calculate the sum, the complete algorithm runs in O(n) * O(n) * O(1) time or worst case $O(n^2)$ time. It also works at average case $\Theta(n^2)$ time.

Algorithm 3: Divide and Conquer

Pseudocode:

```
MaxSubArray(array[1...n])
        if array size = 1
                return
        else
                mid = array length/2
                leftMaxArray = MaxSubArray(array, begin, mid)
                rightMaxArray = MaxSubArray(array, mid + 1, end)
                midMaxArray = MaxMidArray(array)
                if leftMaxArray >= rightMaxArray AND leftMaxArray >= midMaxArray
                        return leftMaxArray
                else if rightMaxArray >= leftMaxArray AND rightMaxArray >= midMaxArray
                        return rightMaxArray
                else
                        return midMaxArray

MaxMidArray(array[1..n)
        for i = mid → beginning
                sum = sum + array[i]
                if sum > leftSum
                        leftSum = sum
                        maxLow = i
        sum = 0
        for j = mid + 1→ end
                sum = sum + array[j]
                if sum > rightSum
                        rightSum = sum
                        maxHigh = j
        return maxLow, maxHigh, leftSum + rightSum
```

Asymptotic Analysis:

The divide and conquer algorithm splits the array into smaller parts halving each time. This build a recursive tree of depth log n. To do the non-recursive work of iteration and addition it takes O(n) time. The complete formula for the recurrence is T(n) = 2T(n/2) + n as each part is broken into 2 subproblems of size n/2 each. Using the Master Method you can see that $n^{\log(2)\,2} = n^1 = \Theta(n)$ which is Case Two. Meaning the algorithm runs at $\Theta(n^{\log(2)\,2} \lg n)$ or $\Theta(n \log n)$ time.

Algorithm 4: Linear-time

Pseudocode:

```
MaxSubArray(array)
    maxSum = maxStart = maxEnd = start = end = maxCurrent = 0
    for i from beginning → array end
        maxCurrent = maxCurrent + array[i]
        if maxCurrent < 0
            maxCurrent = 0
            start = end = i + 1
        else
            end = i
            if maxCurrent > maxSum
                maxSum = maxCurrent
                maxStart = start
                maxEnd = end
    return array[maxStart → maxEnd], maxSum
```

Asymptotic Analysis:

The final algorithm only needs to iterate through each value once making a maximum of two decisions each time, first checking if maxCurrent is greater than 0 then if it is checking if the maxCurrent is greater than the maxSum. If it is then it adjusts the max sub array. Because this only iterates through each value once and the computation of the maximum subarray of 1 → n and max suffix from 1 → n from the max subarray 1 → n-1 and max suffix 1 → n-1 can be done in constant time this algorithm takes O(n) * O(1) = O(n) time.

Testing

Algorithm 1: Enumeration

An array of random numbers was generated including both positive and negative numbers. Using a seed number in the generation of the random array allowed the results to be duplicated. Algorithm 1 was run against a set of test numbers and the results were checked by hand. The results proved correct and then the same array was used for the other algorithms to confirm that they were generating the same results. Next, the algorithm was run against a case in which the answer was provided. Algorithm 1 produced the expected result.

Algorithm 2: Better Enumeration

An array of random numbers was generated including both positive and negative numbers. Using a seed number in the generation of the random array allowed the results to be duplicated. Algorithm 2 was run against a set of test numbers and the results were checked by hand. The results proved correct and then the same array was used for the other algorithms to confirm that they were generating the same results. Next, the algorithm was run against a case in which the answer was provided. Algorithm 2 produced the expected result.

Algorithm 3: Divide and Conquer

An array of random numbers was generated including both positive and negative numbers. Using a seed number in the generation of the random array allowed the results to be duplicated. Algorithm 3 was run against a set of test numbers and the results were checked by hand. The results proved correct and then the same array was used for the other algorithms to confirm that they were generating the same results. Next, the algorithm was run against a case in which the answer was provided. Algorithm 3 produced the expected result.

Algorithm 4: Linear-time
An array of random numbers was generated including both positive and negative numbers. Using a seed number in the generation of the random array allowed the results to be duplicated. Algorithm 4 was run against a set of test numbers and the results were checked by hand. The results proved correct and then the same array was used for the other algorithms to confirm that they were generating the same results. Next, the algorithm was run against a case in which the answer was provided. Algorithm 4 produced the expected result.

Experimental Analysis

1. *Calculate the average running time for each n*

Algorithm 1: Enumeration

Algorithm 1 Runtime for Input of Size N												
n	10	20	40	80	160	200	350	500	650	800	1000	1500
Runtime (μs)	50	185	807	3971	21976	38584	174048	472831	995526	1811328	3477105	11554133

Algorithm 2: Better Enumeration

Algorithm 2 Runtime for Input of Size N												
n	20	40	80	160	200	350	500	650	800	1000	2000	3000
Runtime (μs)	71	242	877	3321	5158	16731	35829	61501	94252	149358	606130	1358884

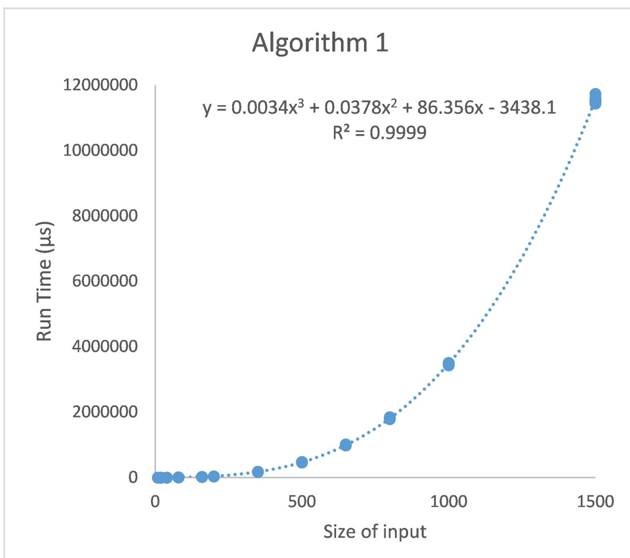
Algorithm 3: Divide and Conquer

Algorithm 3 Runtime for Input of Size N												
n (2.5(n-1))	10	25	63	156	391	977	2442	6104	15259	38147	95368	238419
Runtime (μs)	55	142	371	954	2572	6805	17935	46477	121415	314206	822168	2143147

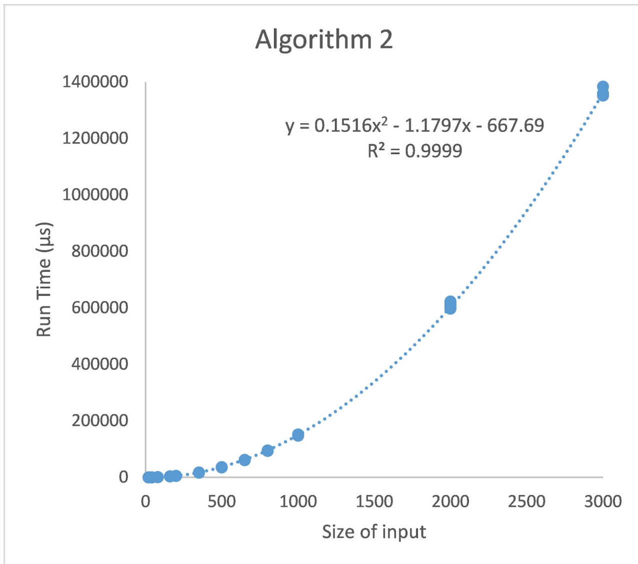
Algorithm 4: Linear-time

Algorithm 4 Runtime for Input of Size N												
n	*10*	*50*	*100*	*500*	*1000*	*5000*	*10000*	*50000*	*100000*	*500000*	*1000000*	*5000000*
Runtime (µs)	*7*	*17*	*30*	*148*	*297*	*1480*	*2962*	*14873*	*29815*	*149257*	*300853*	*1502051*

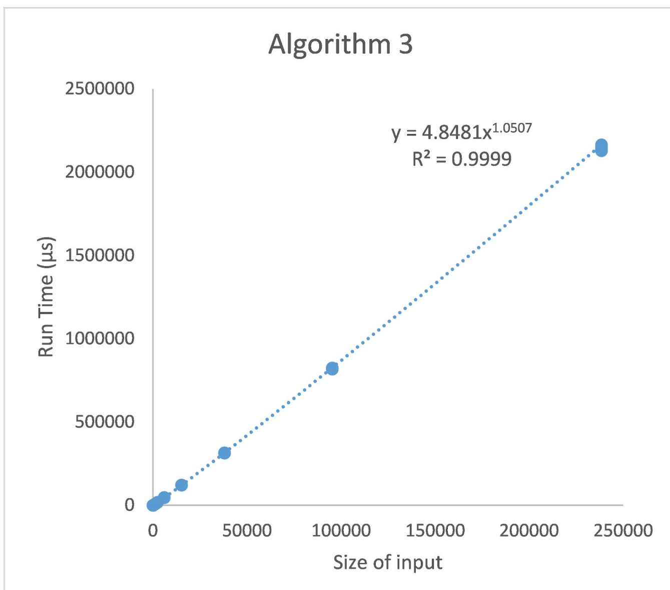
2. *Plot the running times as a function of input size n*
   Algorithm 1: Enumeration



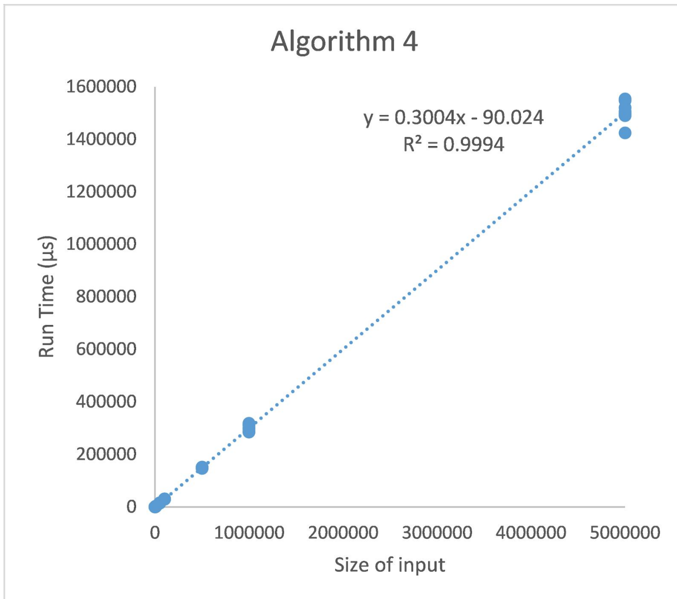
Algorithm 2: Better Enumeration



Algorithm 3: Divide and Conquer



Algorithm 4: Linear-time



3. *Find a function that models the relationship between input n and time. This function will produce a curve that "fits" the data plotted in part 2. To determine the equation of the function for the curve use regression techniques.*

   Algorithm 1: Enumeration
   $y= 0.0034x^3 + 0.0378x^2 + 86.356x - 3438.1$
   Data is exponential.

   Algorithm 2: Better Enumeration
   $y = 0.1516x^2 - 1.1797x - 667.69$
   Data is quadratic.

   Algorithm 3: Divide and Conquer
   $y=4.8481x^{1.0507}$
   Data is logarithmic.

   Algorithm 4: Linear-time
   $y = 0.3005x - 90.024$
   Data is linear.

4. *Discuss any discrepancies between the experimental and theoretical running times.*

   Algorithm 1: Enumeration
   Our initial asymptotic analysis and experimental data analysis both came up with a running time of $\Theta(n^3)$.

Algorithm 2: Better Enumeration
Our initial asymptotic analysis and experimental data analysis both came up with a running time of $\Theta(n^2)$.

Algorithm 3: Divide and Conquer
Our initial asymptotic analysis and experimental data analysis both came up with a running time of $\Theta(n \log n)$. The experimental data suggests $\Theta(n \log n)$ because the runtime of the algorithm is <= $O(n^{1.5})$ which is $n\sqrt{n}$.

Algorithm 4: Linear-time
Our initial asymptotic analysis and experimental data analysis both came up with a running time of $\Theta(n)$.

5. *Use the regression model to determine the largest input for the algorithm that can be solved in 1 minute, 2 minutes, and 5 minutes.*

Algorithm 1: Enumeration
One minute:
$y = 0.0034x^3 + 0.0378x^2 + 86.356x - 3438.1$
60,000,000 microseconds = $0.0034x^3 + 0.0378x^2 + 86.356x - 3438.1$
$x = 2{,}597$

Two minutes:
$y = 0.0034x^3 + 0.0378x^2 + 86.356x - 3438.1$
$1.2 * 10^8$ microseconds $= 0.0034x^3 + 0.0378x^2 + 86.356x - 3438.1$
$x = 3{,}274$

Five minutes:
$y = 0.0034x^3 + 0.0378x^2 + 86.356x - 3438.1$
$3 * 10^8$ microseconds $= 0.0034x^3 + 0.0378x^2 + 86.356x - 3438.1$
$x = 4{,}447$

Algorithm 2: Better Enumeration
One minute:
$y = 0.1516x^2 - 1.1797x - 667.69$
60,000,000 microseconds $= 0.1516x^2 - 1.1797x - 667.69$
$x = 19{,}898$

Two minutes:
$y = 0.1516x^2 - 1.1797x - 667.69$
$1.2 * 10^8$ microseconds $= 0.1516x^2 - 1.1797x - 667.69$
$x = 28{,}138$

Five minutes:
$y = 0.1516x^2 - 1.1797x - 667.69$
$3 * 10^8$ microseconds $= 0.1516x^2 - 1.1797x - 667.69$
$x = 44{,}488$

Algorithm 3: Divide and Conquer
One minute:
$y = 4.8481x^{1.0507}$
60,000,000 microseconds $= 4.8481x^{1.0507}$
$x = \sim 5.5$ million

Two minutes:
$y = 4.8481x^{1.0507}$
$1.2 * 10^8$ microseconds $= 4.8481x^{1.0507}$
$x = \sim 10.5$ million

Five minutes:
$y = 4.8481x^{1.0507}$
$3 * 10^8$ microseconds $= 4.8481x^{1.0507}$
$x = \sim 25.2$ million

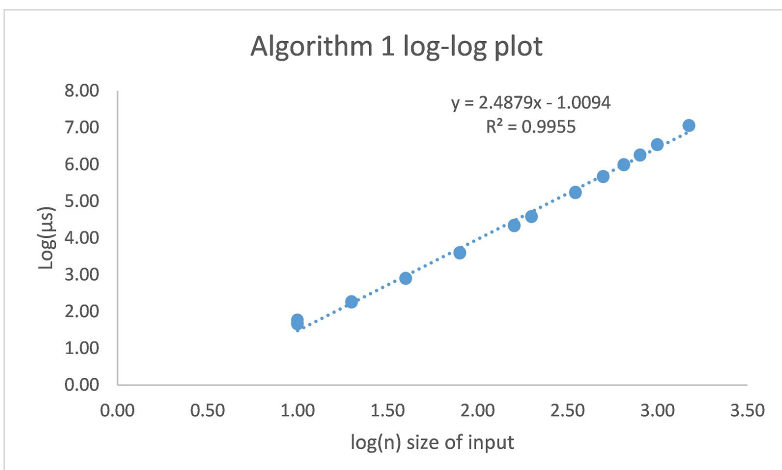
Algorithm 4: Linear-time
One minute:
y = 0.3005x - 90.024
60,000,000 microseconds = 0.3005x - 90.024
x = 1.99668 * 10^8

Two minutes:
y = 0.3005x - 90.024
1.2 * 10^8 microseconds = 0.3005x - 90.024
x = 3.99335 * 10^8

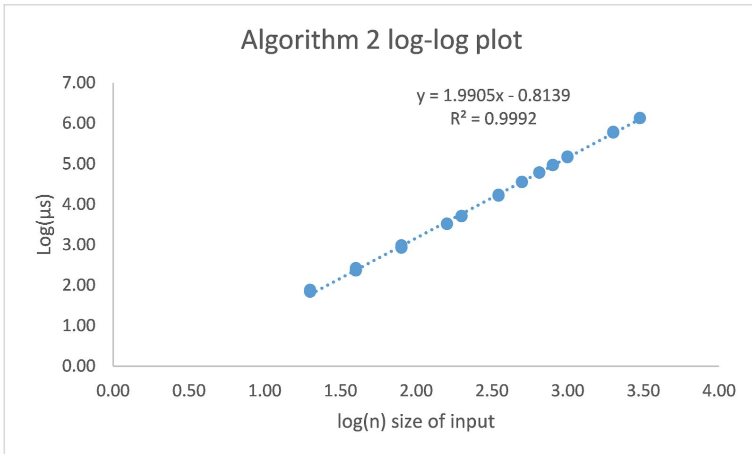
Five minutes:
y =0.3005x - 90.024
3 * 10^8 microseconds = 0.3005x - 90.024
x = 9.98336 * 10^8

6. *Create a log-log plot of the running times*

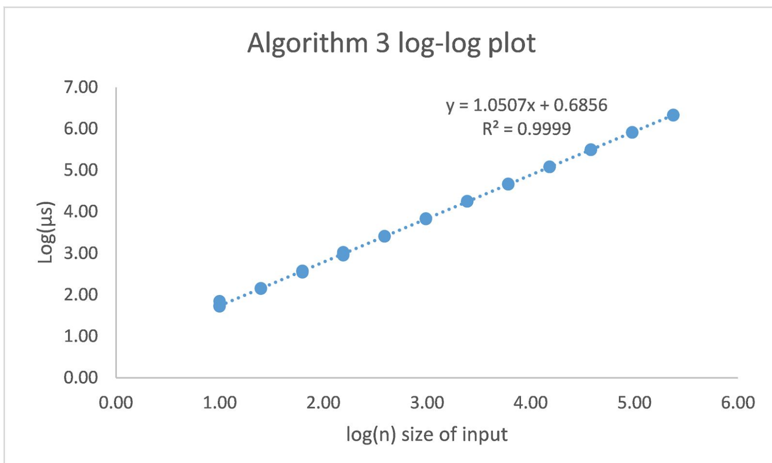
Algorithm 1: Enumeration



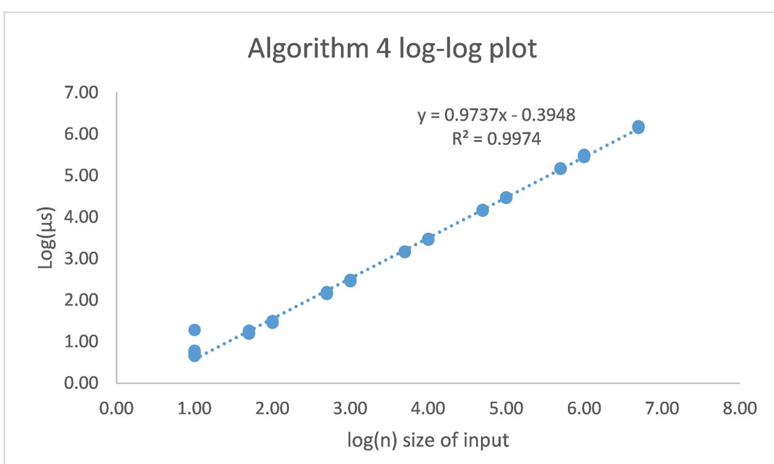
Algorithm 2: Better Enumeration



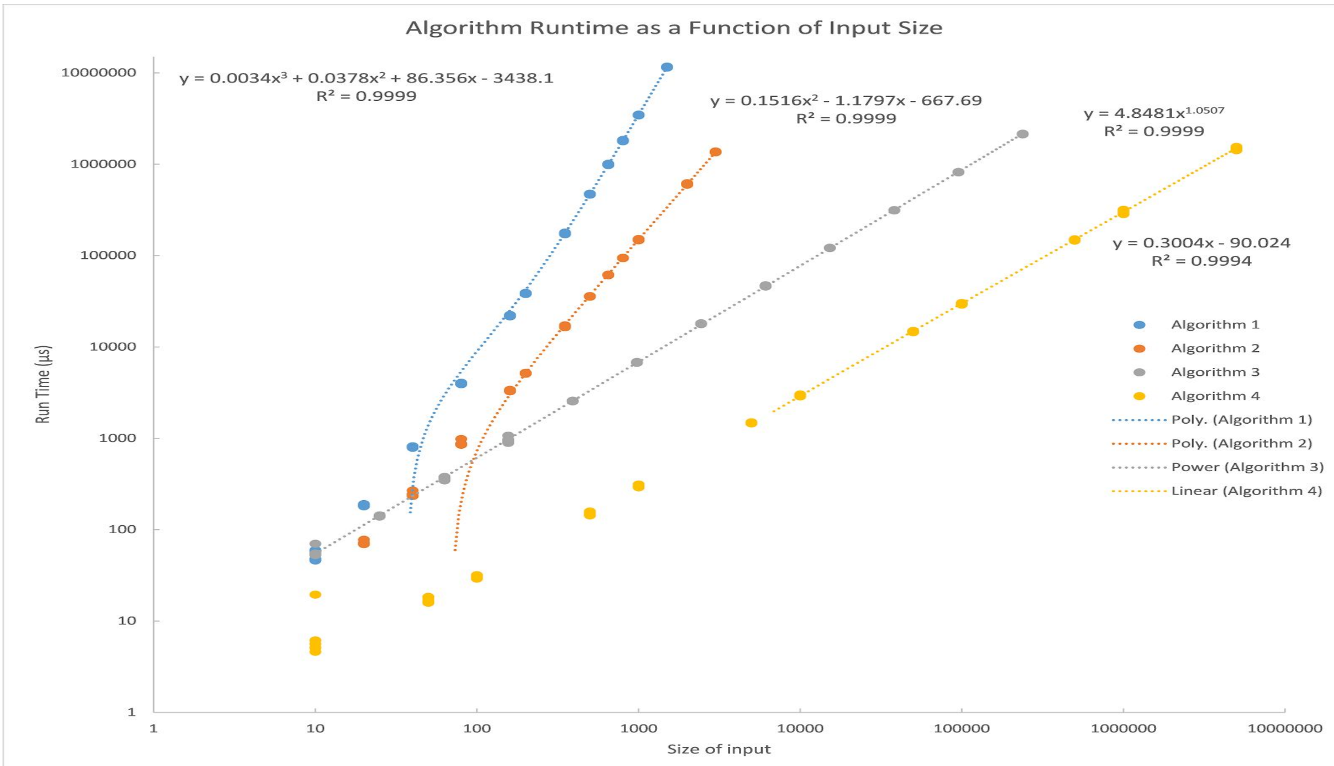
Algorithm 3: Divide and Conquer



Algorithm 4: Linear-time



7. *All four algorithms together on a single graph and/or log log plot depending on scale.*



References

Poly Time Algorithms Lecture (https://www.youtube.com/watch?v=QKgwslVGLGA)