Robert Ottolia
Myles Chatman
Andrew Brown
CS 325 Algorithms
Group 26 Winter 2016

Project 1: Maximum Sum Subarray

1. Theoretical Run-Time Analysis

1.1 **Enumeration Algorithm**

Pseudocode:

```
maxSum ←0
for i = 0 to size of array do
        for j = i to size of array do
                newSum← sum of array from index i to index j
                if newSum > maxSum then
                        maxSum←newSum
                end if
        end for
end for
return maxSum
```

Analysis:

There is no recursion in this algorithm. There is n*n computation done for the nested for-loops. Additionally, there is a summing up of n items within the innermost for-loops. Thus, the running time is $\boldsymbol{\theta}(n^3)$. This is a tight bound because we must sum up all elements in the array in order to accurately gauge the greatest sum.

1.2 **Better Enumeration Algorithm**

Pseudocode:

```
maxSum ←0
for i = 0 to size of the array do
        oldSum ←0
        for j = i to size of array do
                if j = 0 then
                        newSum← content of the array at index i
                else
                        newSum← oldSum + content of the array at index j
                end if
                if newSum > maxSum then
                        maxSum← newSum
                end if
                oldSum← newSum
        end for
end for
return maxSum
```

Analysis:
This algorithm removes the implicit third loop adding up the array elements from index *i* to index *j* in the first algorithm. Thus, this algorithm has a better complexity of $\boldsymbol{\theta}(n^2)$.

1.3 **Divide and Conquer Algorithm**

Pseudocode:

```
function maxSuffix(array):
        reverseArray ← array is reversed
        max ← first element of reversed array
        sum ←  0
        for i = 0 to end of reverseArray do
                sum← sum + i
                if sum > max then
                        max← sum
                endif
        end for
return max

function maxPrefix(array):
        max← first element of array
        sum← 0
        for i = 0 to end of array do
                sum← sum + i
                if sum > max then
                        max← sum
                end if
        end for
return max

function divideAndConquer(array):
        if size of array <= 1 then
                return first element of array
        else
                firstHalf ←first half of array
                secondHalf← second half of array
                first← divideAndConquer(firstHalf)
                last← divideAndConquer(secondHalf)
                middle← maxSuffix(firstHalf) + maxPrefix(secondHalf)
                return max(first, last, middle)
        end if
```

Analysis:
Assuming the size of the input array is > 1, the division of the array into two halves takes $\boldsymbol{\theta}(1)$ time. From there, the algorithm has to solve two subarrays of $n/2$. This takes $2T(n/2)$ time. We then must look at the case when the maximum subarray could be contained in the suffix of the first half and a prefix of the second half. The two helper algorithms in the code determine the max array sum in these two pieces. It is clear that these two algorithms have to loop through n elements, adding a complexity of $\boldsymbol{\theta}$ (n), and then the calling function has to combine the results which is a $\boldsymbol{\theta}(1)$ operation.

Therefore, the overall recurrence is:

$T(n) = 2T(n/2) + \boldsymbol{\theta}(n) + \boldsymbol{\theta}(n) + \boldsymbol{\theta}(1)$

Combining and dropping constants/low order terms we get the recurrence:
$T(n) = \boldsymbol{\theta}(1)$, if n = 1, and
$T(n) = 2T(n/2) + \boldsymbol{\theta}(n)$ if n > 1

By using the master theorem, the solution to the recurrence is $\boldsymbol{\theta}(n \log n)$.

1.4 **Linear-Time Algorithm**

Pseudocode:

bestSum← *–MAX* (lowest possible value that can be stored in type int)
bestStart← *bestEnd* = -1
localStart← *localSum* = 0
for *i* = 0 to size of array - 1 **do:**
    localSum← *localSum* + value at index *i*
    if *localSum* > *bestSum* **then**
        bestSum← *localSum*
        bestStart← *localStart*
        bestEnd← *i*
    end if
    if *localSum* <= *0* **then**
        localSum← *0*
        localStart← *i* + 1

Analysis:

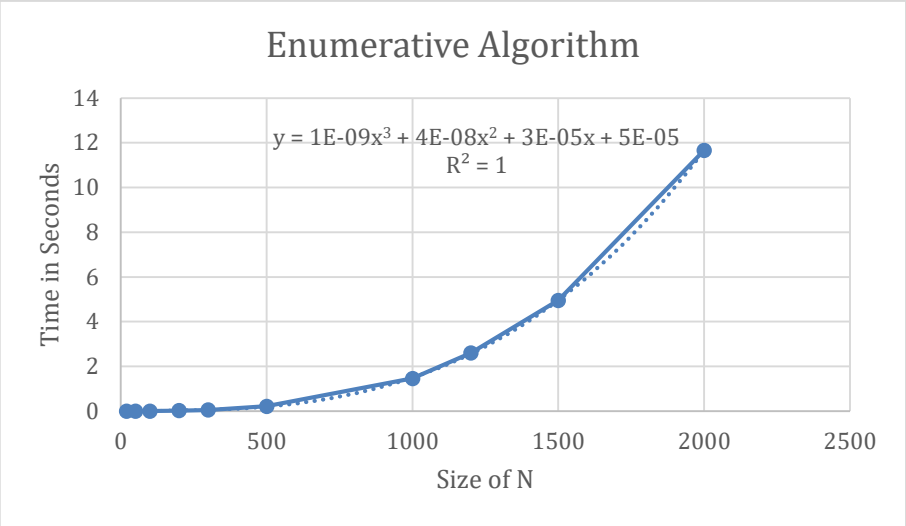
This algorithm iterates through the array, keeping a tally of the current subarray. If the current subarray sum is greater than zero, it will contribute to future subset sums, so it is kept. Otherwise, if the current subset sum is less than or equal to zero, it will not contribute to any future subarray sums so it is discarded and the algorithm starts over with a new sum. From there, all that must be done is update the maximum sum that has been encountered.

The algorithm has one for-loop, looping over all of the elements of the array. The addition of variables are all of complexity $\boldsymbol{\theta}(1)$, so the overall complexity of this algorithm is $\boldsymbol{\theta}(n)$.

2. Experimental Analysis

2.1 Enumeration Algorithm

Enumerative	
Input(N)	Time
20	0.00008
50	0.000587
100	0.003105
200	0.018828
300	0.052911
500	0.215364
1000	1.461163
1200	2.596658
1500	4.956266
2000	11.66336



2.1.1 Avg. Running Time

2.1.2 Plot of Running Times

2.1.3 Function Model

Given two x-values on our loglog plot, $x_0 = 10^2$ and $x_1 = 10^3$, and their corresponding y-values, $F_0 = 10^{0.1}$ and $F_1 = 10^{-2.8}$:

$$m = \log (F_1/F_0) / \log (x_1/x_0)$$
$$= \log (1.2589 / 0.001584) / \log (1000 / 100)$$
$$\approx 2.900$$

The graph is exponential and grows quickly as the input grows. Thus, the function that models this relationship could be stated as:

$$T(n) = cn^{2.900}$$
$$\sim T(n) = (1E\text{-}09)n^{2.900}$$

2.1.4 Discrepancies Between Experimental & Theoretical

The data matched our theoretical prediction of $\boldsymbol{\theta}(n^3)$ very closely. The small discrepancy could be explained by external factors such as machine load, processors, etc.

2.1.5 Determining Largest Input for Algorithm

1 Minute = 60 Seconds:

(Using equation from the graph along with Wolfram Alpha)

$$60 = 1 \times 10^{-9}\, x^3 + 4 \times 10^{-8}\, x^2 + 3 \times 10^{-5}\, x + 5 \times 10^{-5}$$

$x \approx 3899$

2 Minutes = 120 Seconds:

$$120 = 1 \times 10^{-9}\, x^3 + 4 \times 10^{-8}\, x^2 + 3 \times 10^{-5}\, x + 5 \times 10^{-5}$$

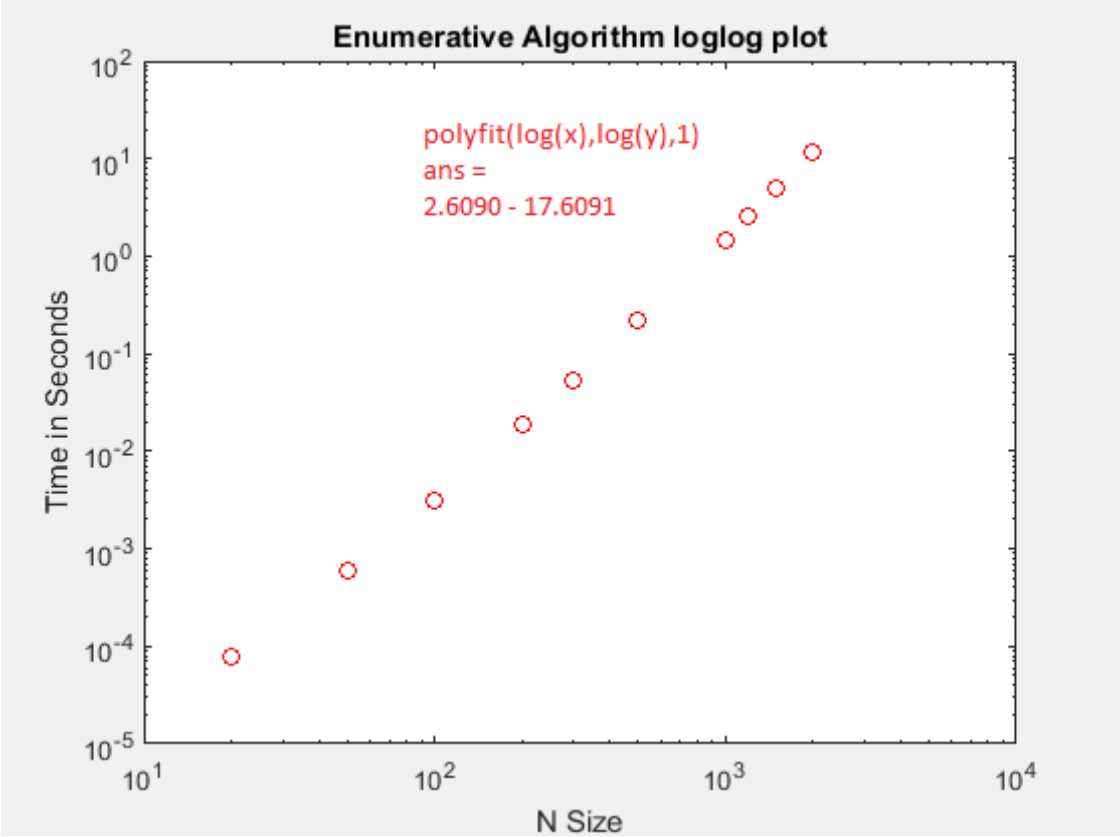
$x \approx 4917.1$

5 Minutes = 300 Seconds:

$$300 = 1 \times 10^{-9}\, x^3 + 4 \times 10^{-8}\, x^2 + 3 \times 10^{-5}\, x + 5 \times 10^{-5}$$

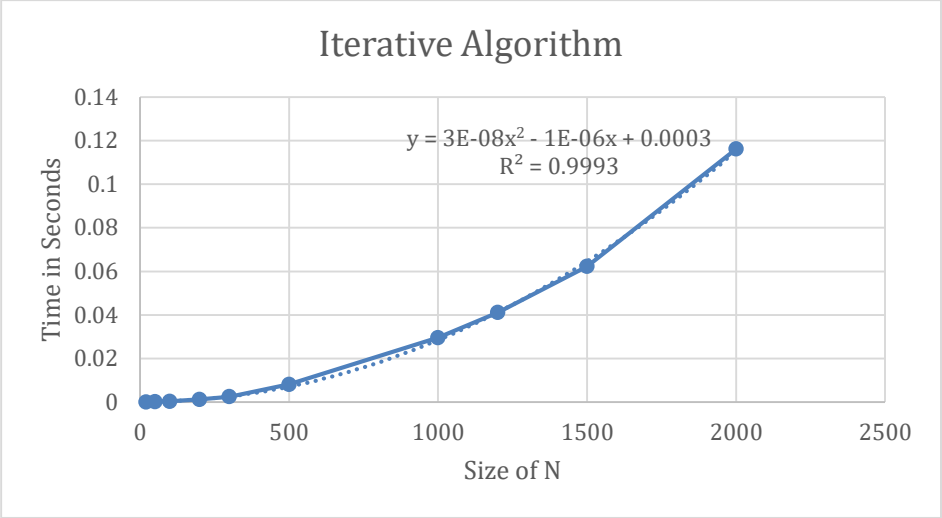
$x \approx 6679.5$

2.1.6 Loglog Plot of Running Times



2.2 Better Enumeration Algorithm

Iterative	
N	Time
20	0.0000256
50	0.0001441
100	0.0003382
200	0.0012358
300	0.0024545
500	0.0081496
1000	0.0295787
1200	0.0410674
1500	0.0622348
2000	0.1161804



2.2.1 Avg. Running Time

2.2.2 Plot of Running Times

2.2.3 Function Model

Given two x-values on our loglog plot for this algorithm, and their corresponding values, we arrive at the *m* value:

$$m = \log (F_1/F_0) / \log (x_1/x_0)$$
$$= \log (10^{-1.6}/10^{-3.5}) / \log (10^3/10^2)$$
$$\approx 1.9$$

The graph is also exponential, but does not grow as quickly as the first algorithm's graph. The function that models this relationship could be stated as:

$$T(n) = cn^{1.9}$$
$$\tilde{T}(n) = (3E\text{-}08)n^{1.9}$$

2.2.4 Discrepancies Between Experimental & Theoretical

The derived function matches our theoretical analysis of $\boldsymbol{\theta}(n^2)$ extremely well. Once again, the small variances could be explained by external factors. Overall, it was clear by the regression that while still exponential in nature, this algorithm grew at a slower rate than the first.

2.2.5 Determining Largest Input for Algorithm

60 Seconds:

$$60 = 3 \times 10^{-8}\, x^2 - 1 \times 10^{-6}\, x + 0.0003$$

$$x \approx 44737$$

120 Seconds:

$$120 = 3 \times 10^{-8}\, x^2 - 1 \times 10^{-6}\, x + 0.0003$$

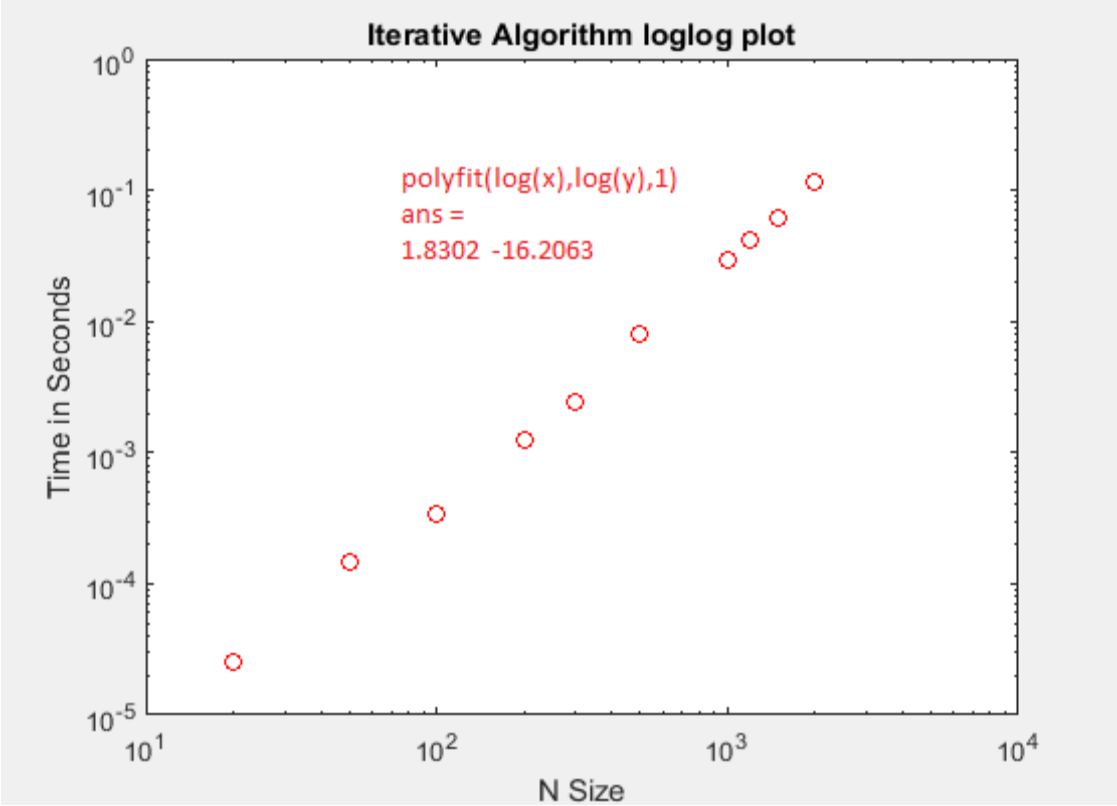
$$x \approx 63262$$

300 Seconds:

$$300 = 3 \times 10^{-8}\, x^2 - 1 \times 10^{-6}\, x + 0.0003$$

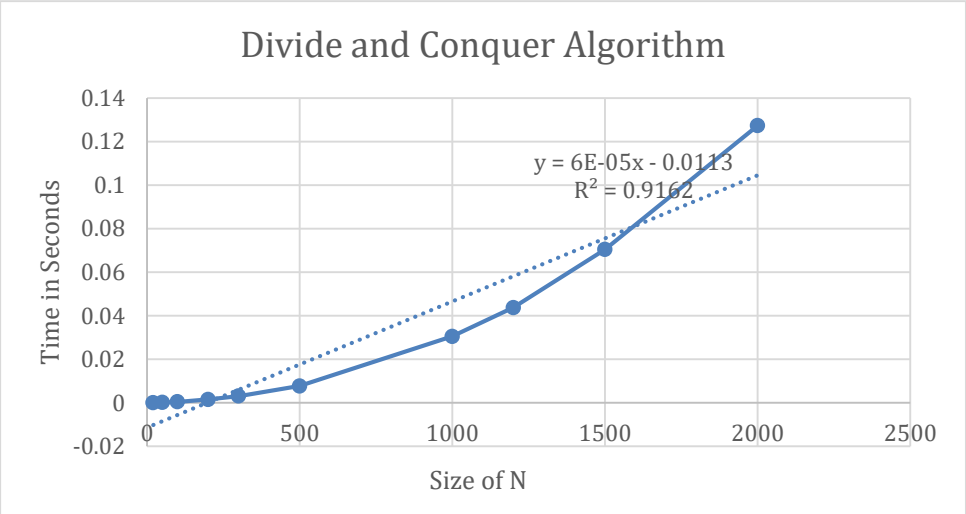
$$x \approx 100017$$

2.1.6 Loglog Plot of Running Times



2.3 Divide And Conquer Algorithm

Divide & Conquer	
N	Time
20	0.00006
50	0.000169
100	0.000487
200	0.001543
300	0.003079
500	0.007769
1000	0.030524
1200	0.043727
1500	0.070405
2000	0.127336



2.3.1 Avg. Running Time

2.3.2 Plot of Running Times

2.3.3 Function Model

Using the equation from our regression line best fit:

$$y = 6E\text{-}05x - 0.0113$$

We can pick two points (1000, 0.0487), (2000, 0.1087) and then use it to find out *m*:

$$m = (0.1087/0.0487) / (2000/1000)$$

$$m \approx 1.1$$

The slope then appears to be very close to one. If we use data from our loglog plot instead, we get a slope closer to 1.8. Overall, the regression line on the graph appears to be linear in nature, but is growing faster than a pure linear solution. As such, this algorithm is much more efficient than either of the previous two.

We can then rewrite our model function as:

$$T(n) = 6E\text{-}05n = cn$$

2.3.4 Discrepancies Between Experimental & Theoretical

The experimental did not match our prediction of $\boldsymbol{\theta}(n \log n)$ perfectly, but we see that log n is often almost a constant when n is low. We observe this pattern in our graph before the size of n increases.

2.3.5 Determining Largest Input for Algorithm

60 Seconds:

$$60 = 6E\text{-}05x - 0.0113$$
$$x \approx 1E6$$

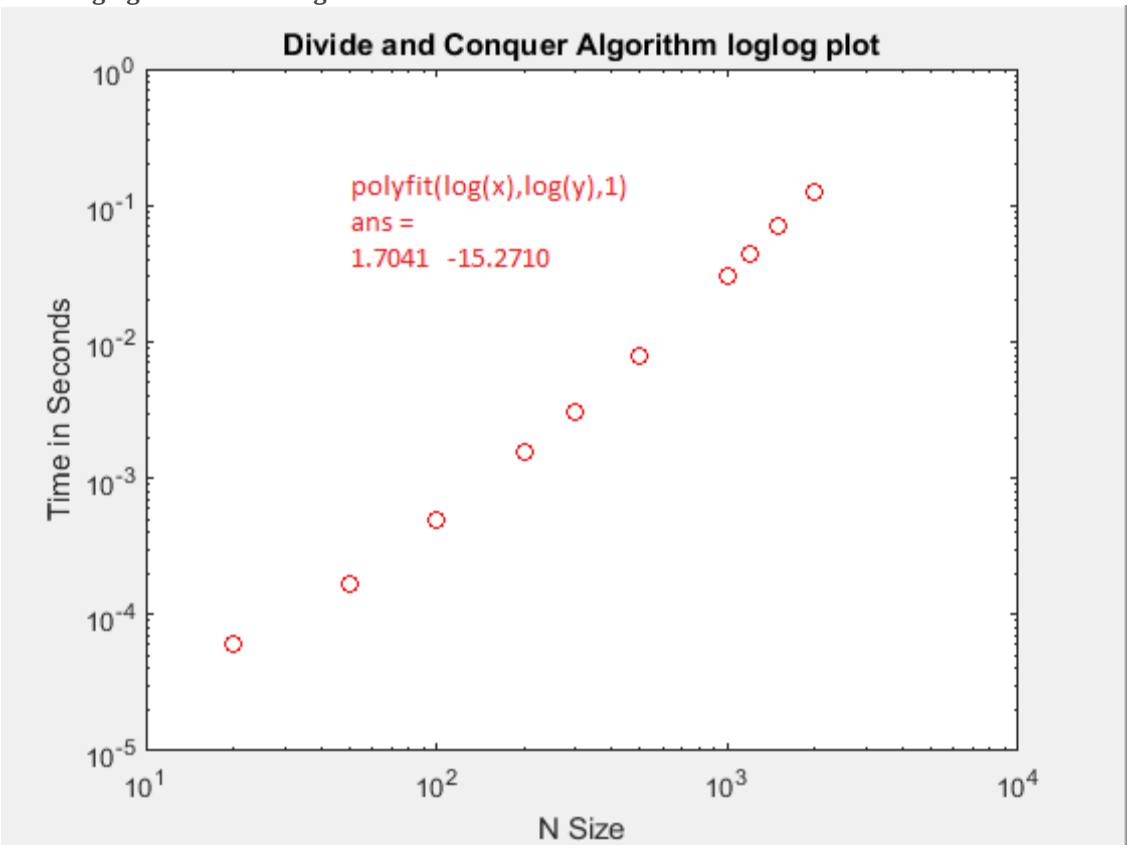
120 Seconds:

$$120 = 6E\text{-}05x - 0.0113$$
$$x \approx 2E6$$

300 Seconds:

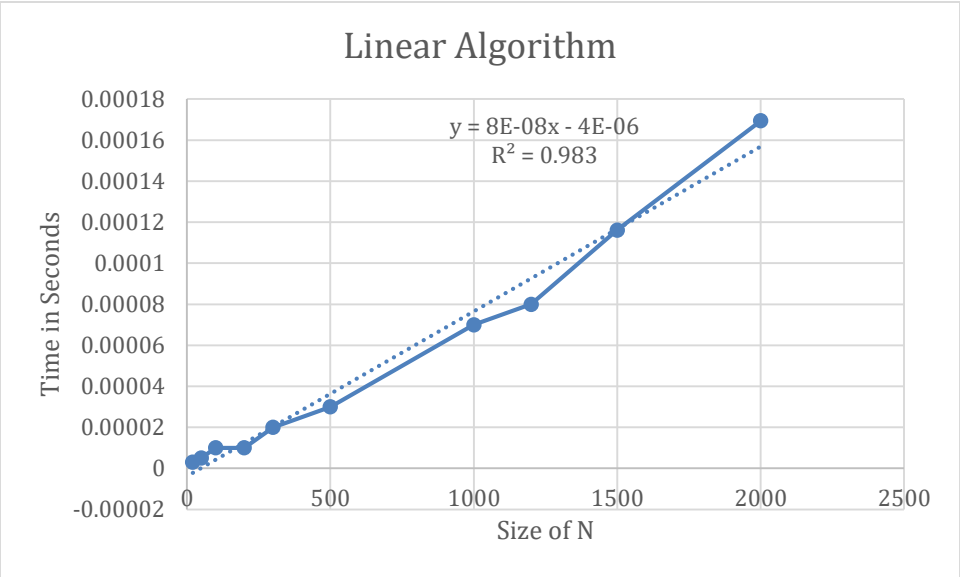
$$300 = 6E\text{-}05x - 0.0113$$
$$x \approx 5E6$$

2.3.6 Loglog Plot of Running Times



2.4 Linear Algorithm

Linear	
N	Time
20	0.000003
50	0.000005
100	0.00001
200	0.00001
300	0.00002
500	0.00003
1000	0.00007
1200	0.00008
1500	0.000116
2000	0.000169



2.4.1 Avg. Running Time

2.4.2 Plot of Running Times

2.4.3 Function Model

Picking points from the loglog plot we obtain the slope *m:*

$$m = 0.87$$

$$m \approx 0.9$$

This is a reasonable slope considering for a linear algorithm we would expect to have a slope of 1. We must also note than the slope is lower than in our previous algorithm, the divide and conquer. This means that our experimental data shows that the running time grows slower than even the divide and conquer algorithm, which is what we expected.

$$T(n) = (8E\text{-}08)n = cn$$

2.4.4 Discrepancies Between Experimental & Theoretical

Our experimental data matched up closely with our theoretical analysis of $\boldsymbol{\theta}(n)$, and as seen by the graph, the regression line matches the analysis very well. As the input grows, the running times does as well in a linear fashion.

2.4.5 Determining Largest Input for Algorithm

60 Seconds:

$$60 = 8E\text{-}08x - 0.000004$$
$$x \approx 7.5E8$$

120 Seconds:

$$120 = 8E\text{-}08x - 0.000004$$
$$x \approx 1.5E9$$

300 Seconds:

$$300 = 8E\text{-}08x - 0.000004$$
$$x \approx 3.75E9$$

2.3.6 Loglog Plot of Running Times