Group 11

Charles Bennett

Conrad Lewin

Vlad Predovic

# CS 325 Project 1: Maximum Sum Subarray

Theoretical Runtime Analysis

**Algorithm 1 - Enumeration**

**Pseudocode:**

Function enumerate\_sum(array){

For each number i in the array

For each number j in the array

Set the current sum = 0

For each number k from i to j in the array

Add number k to the current sum

If the current sum is greater than the max sum found

Max sum is now equal to current sum.

Save the current i and j values as the array start/end

}

The algorithm operates with three nested loops. The first loop iterates through the entire array. The second loop iterates through the entire array. The final loop iterates from the current value of the first loop until the current value of the second loop. Therefore the third loop will at worst also iterate from when the starting point i is the beginning of the array until the ending point j is the last value in the array.

It is safe to assume that the run-time complexity of of this algorithm

T(n) n(n(n + c) where c is a constant. So T(n) = O(n3)

**Algorithm 2 Better Enumeration**

**Pseudocode:**

Function betterEnumeration(array){

Set max sum to 0

for i = 0 to array.length

Set start index and end index to i

Set current sum to 0

for j = i to in array.length

current sum += array[j]

if current sum > max sum

Set max sum to current sum

Set end index to j

return (array[start index...end index], max sum)

}

This algorithm uses nested for loops to iterate over the given array in order to find the maximum subarray. The outer for loop begins at index 0, setting the current max sum to 0 and assumes that the max subarray is an an array of size 1, namely that array whose sole element index is represented by the current value of *i*. This loop then uses the inner for loop to iterate over the remaining contents of the array, adding each value to the sum of the numbers that came before it and storing that value in the current sum variable. If that sum proves greater than the max sum, initially set to 0, then the max sum is updated to reflect the current sum and the end index is set to the index containing the value that caused the current sum to posses a value greater than the previous max sum. Essentially, this algorithm tests for the maximum subarray by iterating over each index in the array and calculating the sum of all elements that come after that that initial index. The outer loop must perform 2 constant time operations alongside running the inner for loop *n* times, while the inner loop must perform 4 constant time operations, including a comparison, *n* + *n* - 1 + *n* - 2 + … 2 + 1 times. Thus, the complexity of this algorithm can be stated as such:

We can prove this assertion using induction:

This inequality holds for all , thus, .

**Algorithm 3 - Divide and Conquer**

**Pseudocode:**

Function maxCrossSum(array,start,middle,end){

Set left sum to

Set current sum to 0

Set start index and end index to Nil

for i = middle down to in start

Current sum += array[i]

If current sum > left sum

Set left sum to current sum

Set start index to i

Set right sum to

Set current sum to 0

for j = middle + 1 to end

Current sum += array[j]

If current sum > right sum

Set right sum to current sum

Set end index to j

return (start index, end index, left sum + right sum)

}

Function maxSubarray(array,start,end){

If start == end

return (start,end,array[start])

Set middle to (start + end) / 2

Set (start index, end index, left sum) to maxSubarray(array,start,middle)

Set (start index, end index, right sum) to maxSubarray(array,middle + 1, end)

Set (start index, end index, cross sum) to maxCrossSum(array,start,middle,end)

If left sum >= right sum and left sum >= cross sum

return (start index, end index, left sum)

elif right sum >= left sum and right sum >= cross sum

return (start index, end index, right sum)

else

return (start index, end index, cross sum)

}

The divide and conquer approach to the maximum subarray problem involves two functions, one linear in nature and the other recursive. The maxCrossSum function iteratively calculates the max sum of both the left and right sections of an array demarcated by a supplied middle index. Since this algorithm is meant to find the max sum that crosses the given middle index, the left sum is derived by stepping through the array from right to left, assuming that the middle index marks the end of the maximum subarray, in search of the subarray’s start index. The right sum is calculated in the opposite fashion, assuming the element right after the middle is the start index and searching for the correct end index. Once both the maximum left and right subarrays are found, they are added together, thus forming the cross sum, and this new sum is returned along with the start and end index.

In order to furnish the crossSum function with input, we use the maxSubarray function to recursively divide the input array in half until an array of a just a single element is formed. The maxSubarray procedure calls the crossSum function on each of these smaller arrays which results in calculating the maximum subarray for all inputs of size 1 to *n*/2. When the recursion returns to the array of size *n*, it now knows the max subarray for both the left and right half of the array, and so needs only to call the crossSum function one last time to determine whether or not this cross sum is greater than either the left or the right sums. Once this is done, a final comparison is made to determine which of the three sums is greater and the resulting sum is returned along with the relevant start and end index.

The recurrence relation for this algorithm must then take into consideration the crossSum function, which performs various constant time operations *n* times using two separate for loops each performing *n*/2 iterations, and the array divisions performed by two different recursive calls to the maxSubarray procedure. Thus, the recurrence is:

Using the master theorem, we can solve this recurrence relation and derive the run time complexity of the algorithm:

Thus, the maxSubarray function is tightly bound by .

**Algorithm 4 - Linear Time**

**Pseudocode:**

Function maxSubarrayLinear(arr){

Set size to array length

Set maxSum and endHereSum to

Set low and high to 0

For i to size

Set endHereHigh = i

If endHereSum > 0  
 Set endHereSum = endHereSum + arr[i]

Else

Set endHereLow = i

Set endHereSum = arr[i]

If endHereSum > maxSum

Set maxSum = endHereSum

Set low = endHereLow

Set high = endHereHigh

Return (low, high, maxSum)

}

The linear function to find the max subarray starts at the left side of the array and progresses towards the right, storing the value of the max subarray sum. The subarray for any iteration is either empty (sum = zero) or has one more element than the max subarray at the previous position.

The variables low and high are used to show the bounds of a maximum subarray. The maxSum stores the sum of the values in the maximum subarray up to that point. The endHereLow and endHereHigh variables show a maximum subarray ending at index i. As i increments, endHereHigh is set to the new value for i. The endHereSum variable stores the sum of the values in a maximum subarray ending at i.

The for loop traverses the size of the array with an incrementing index i. The first if else statement in the for loop determines if the endHereSum has already been created and a subarray started, and if it has, it adds the current endHereSum to the value at arr[i]. If the subarray hasn’t been started, the endHereLow is set to i and the endHereSum is set to arr[i], since arr[i] would be the only value in the new subarray.

The second if statement in the for loop checks to see if the endHereSum is greater than (not equal to) the current maxSum, and if it is, the maxSum is set to the endHereSum. (Note that there can be multiple maximum subarrays with a maxSum.) The low variable is set to endHereLow and the high variable is set to endHereHigh. The loop iterates through the entire array then returns the low and high values of the subarray, as well as the maxSum.

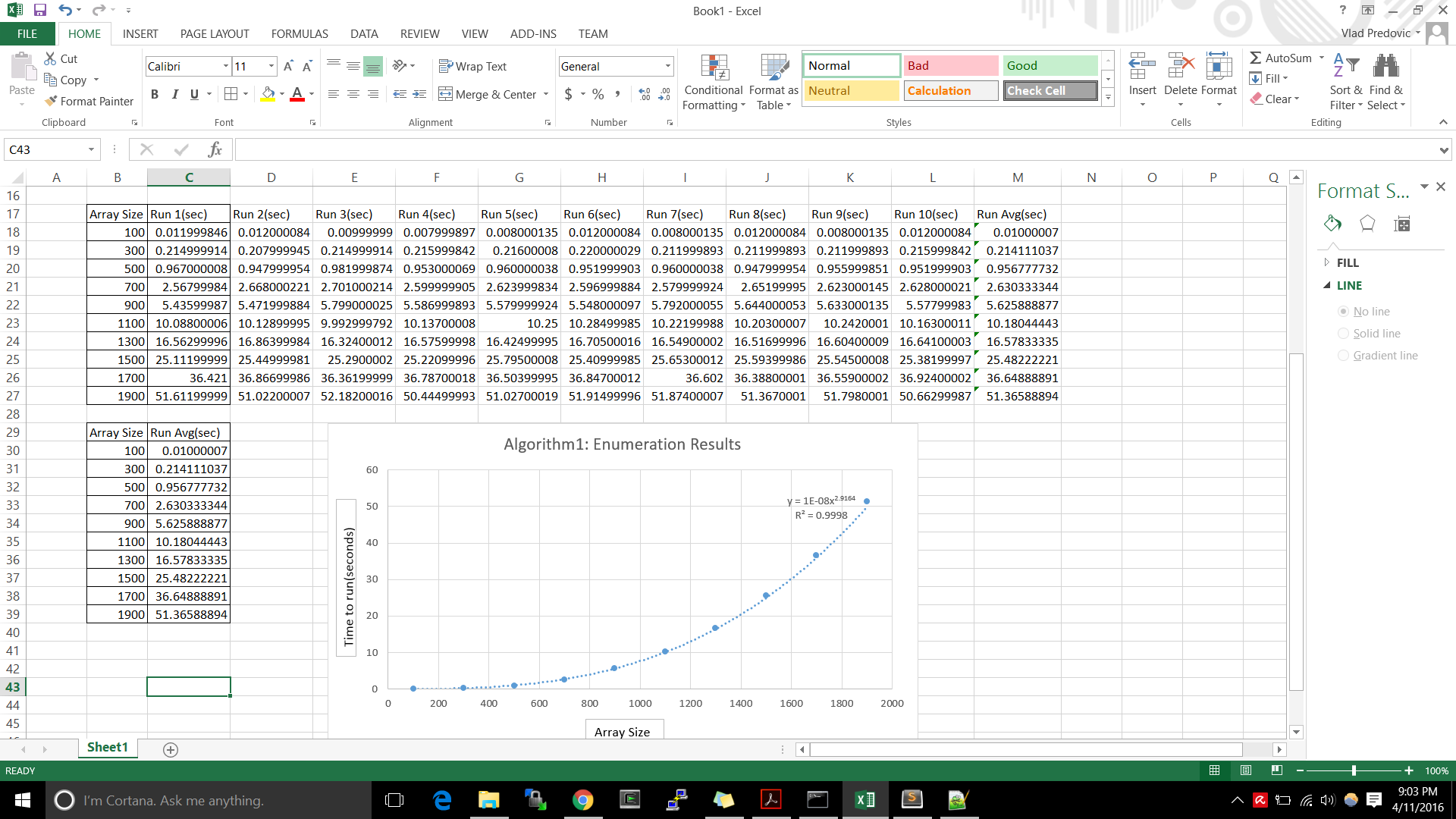
The expected runtime is O(n), because the for loop must iterate through the entire length of the array a single time.

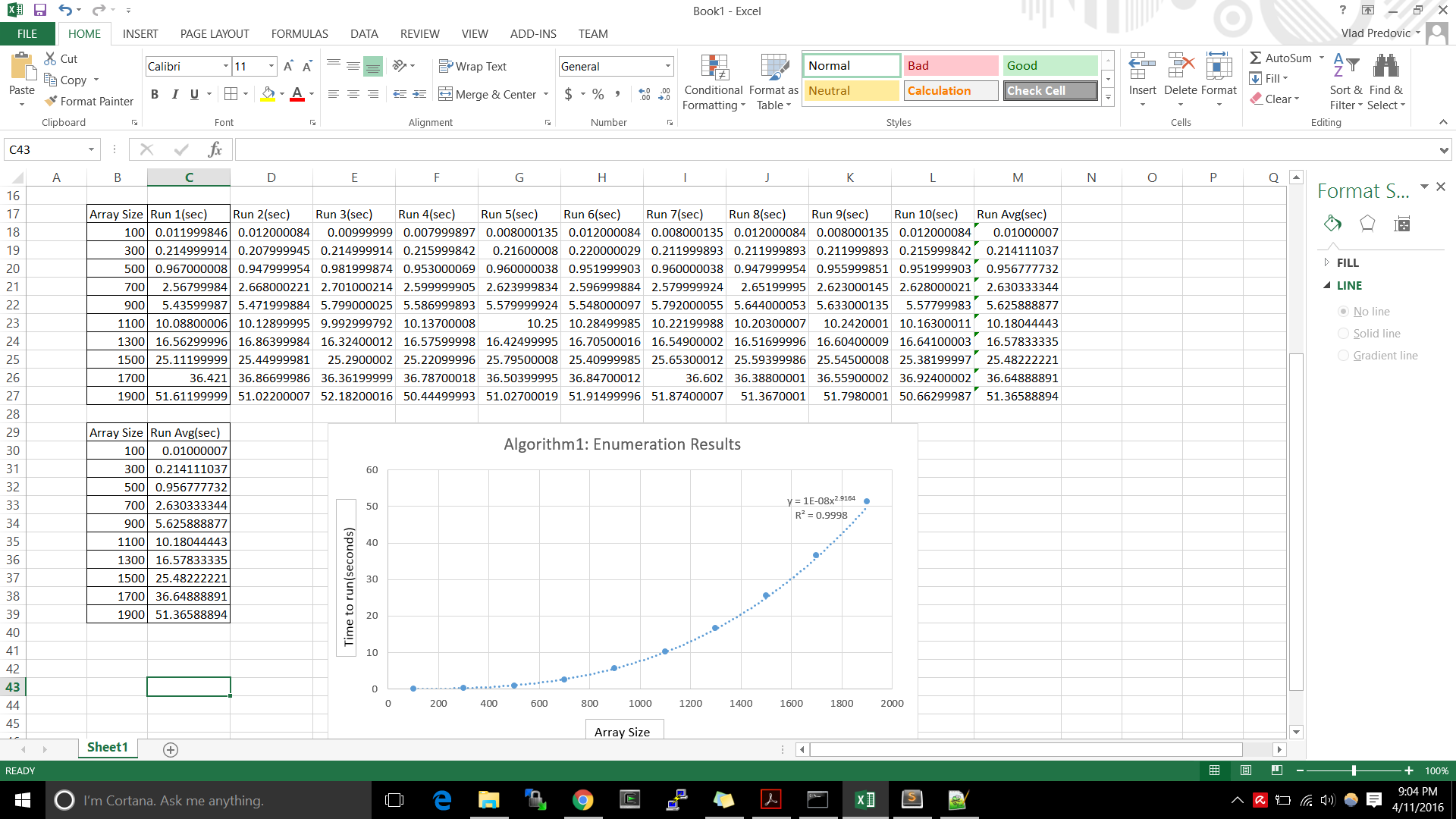
Testing Description:

The initial test of each algorithm was conducted using an instructor provided file containing several arrays of both positive and negative whole numbers. The content of this file was used as input for each algorithm and the resulting output was then written to a file and compared with a set of solutions provided to us by the instructor. After the initial tests had concluded and the output of each algorithm proved correct, we used a test script written by Conrad Lewin to create several arrays of varying length and random numbers to further test the robustness of the algorithms. The results were confirmed as correct by manually adding up the longest subarray in each of the randomly generated arrays and checking that the results matched.

Experimental Analysis

**Algorithm 1**

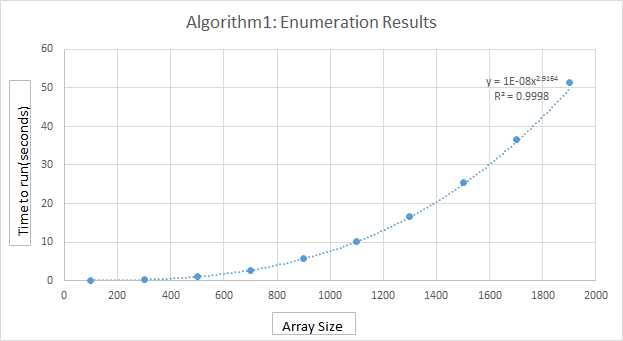
1. The figure below is a view of all test case results and averages.



2. Results for running the enumeration algorithm can be found graphed below. Using Microsoft Excel, the power

trendline seemed the best fit with a R² value of 0.9998.

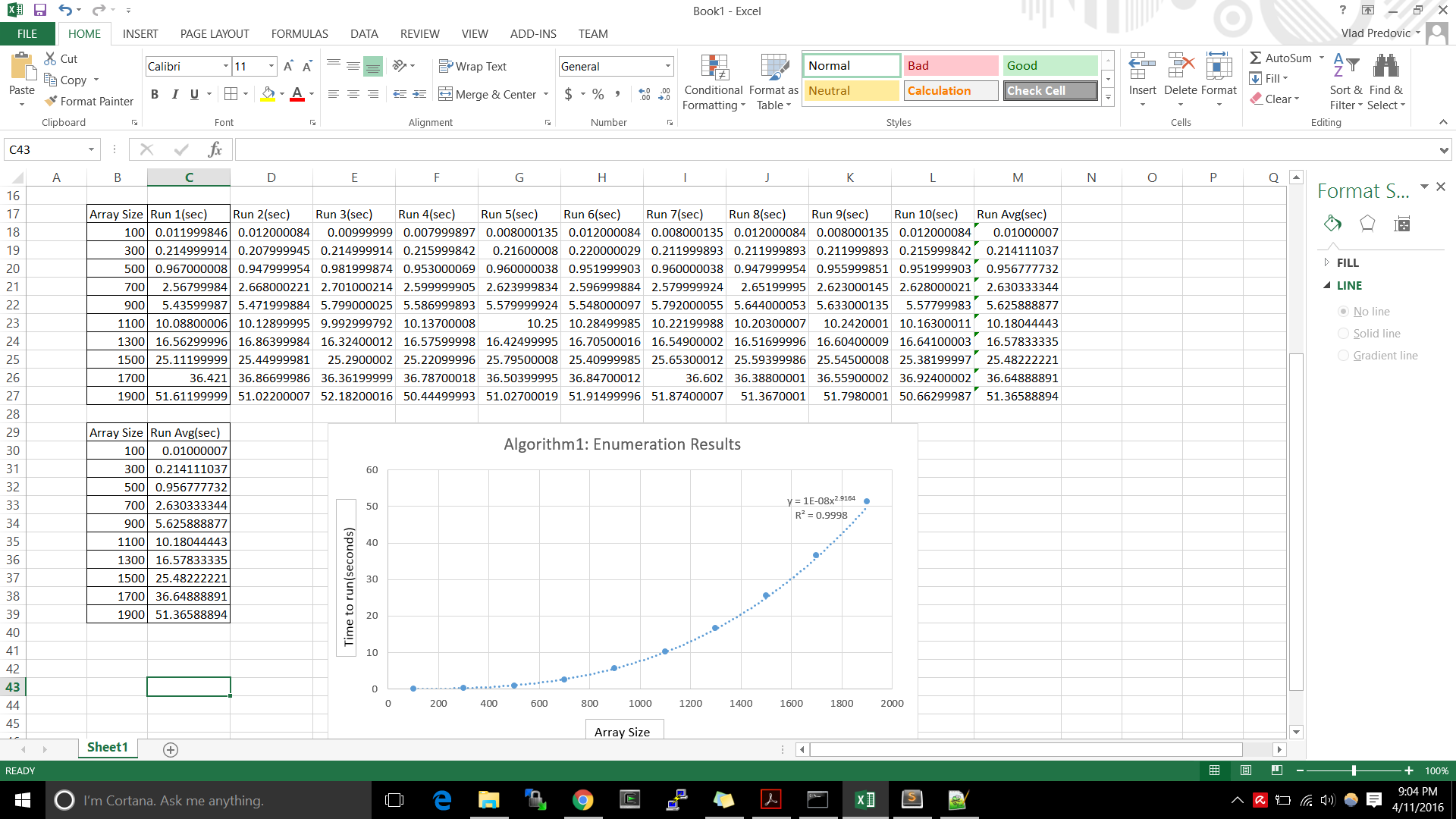
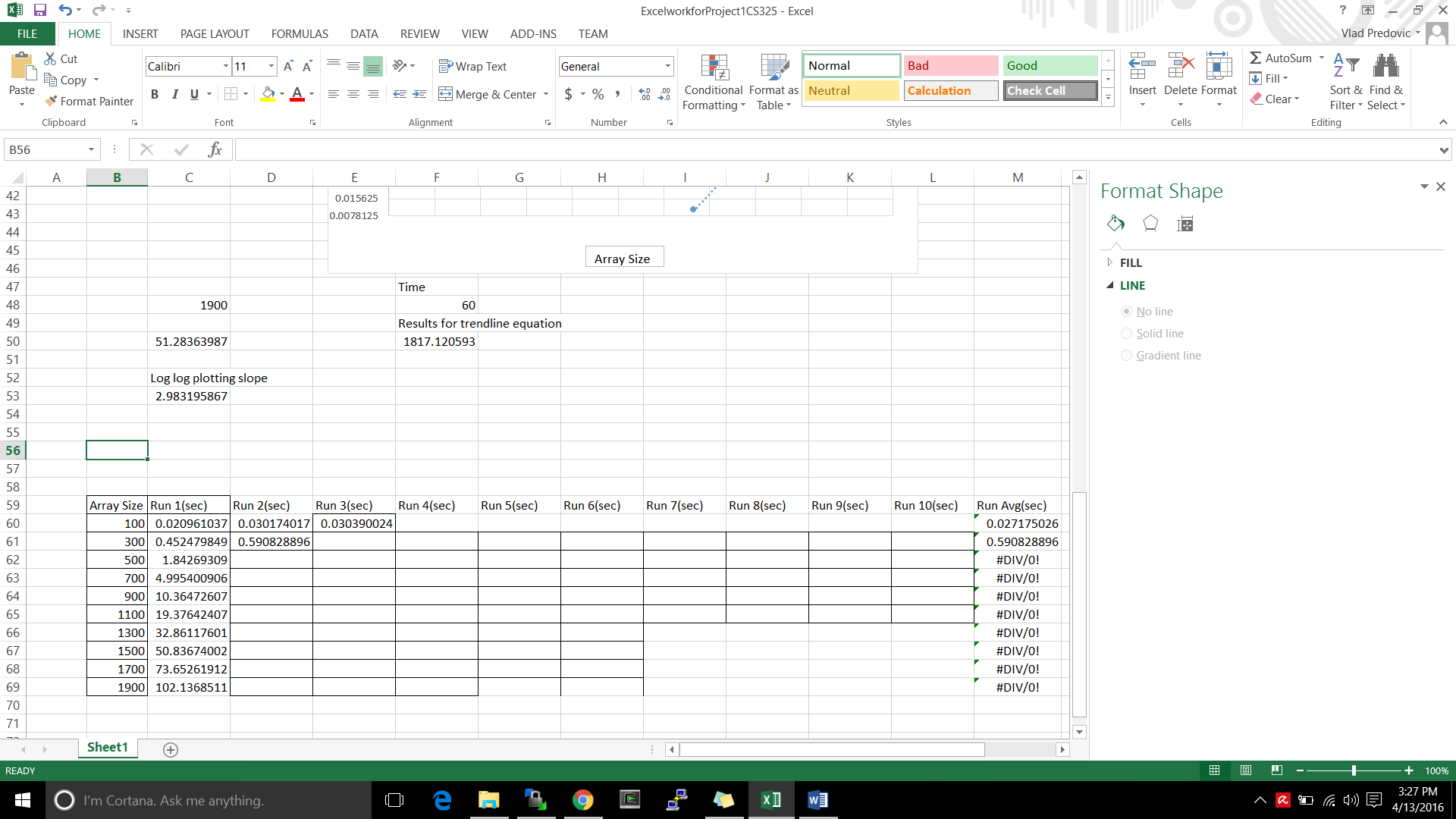
The resulting equation was y = c\*x2.9164 where c is a constant.



3. A regression fit performed using wolfram alpha, a Ti-84 calculator and MS excel provided the following model.

y = 7.319037×10^-9 x^3+2.6741543×10^-7 x^2+0.000075836365 x-0.0270937092

4. The equation given by performing regression analysis in excel is very close to the theoretical results. Slight differences can be a result of limitations by the equipment used as well as how the laptop itself decided to compute results (multithreading, process use etc.) Since algorithm 1 was tested on Vlad’s computer, a comparison between his laptop and flip was done to see if the use of a different CPU would throw off the results regarding time complexity analysis. Below you can see that the results were similar in terms of trends. On the left is the original run on Vlad’s computer and on the right are the results from using OSU’s servers. The OSU results are close to double that of Vlad’s computer but they increase at the same rate, confirming that the data is consistent.



5. Solving for the amount of entries (x) given time (y):

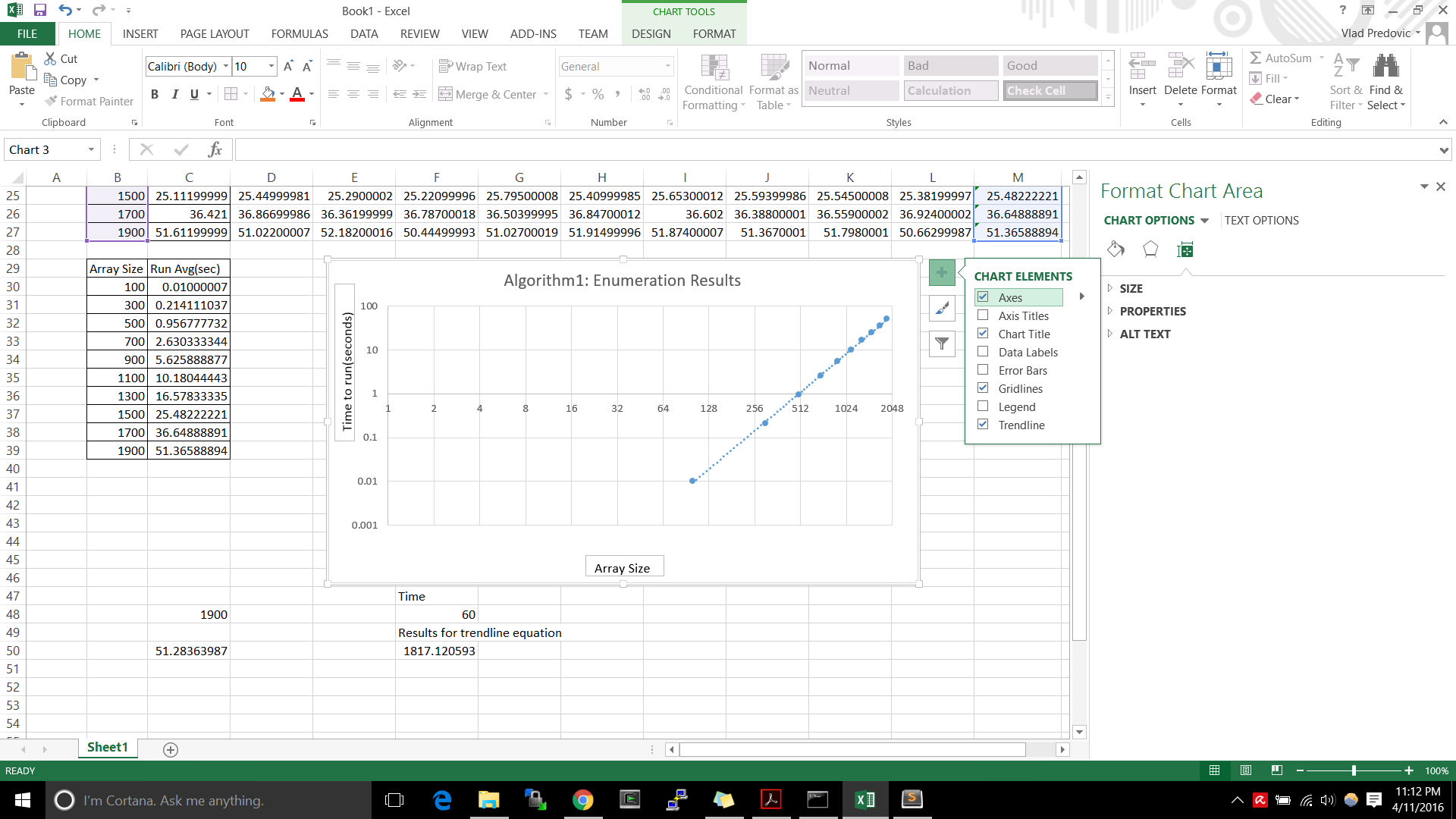
Y = 10 seconds, 30 seconds, and 60 seconds

x = 1095.52x = 2002.84x = 1586.63

Ten seconds gives a maximum of **1095 entries**.

Thirty seconds gives a maximum of **1586 entries.**

One minute gives a maximum of  **2002 entries.**



6. Plotting the data from excel on a loglog plot yields a linear trend as shown in the graph to the right.

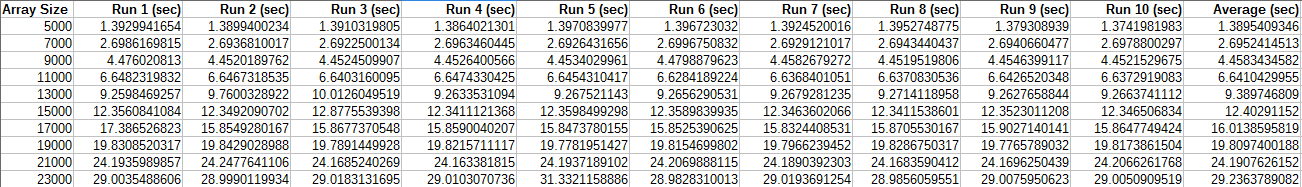
Therefore the equation is given by log(y) = m log(x) + b

m = slope of line = == 2.9836

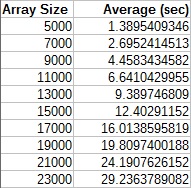
Given some room for error, it is safe to assume that the function F(n) = O(n2.9836) which is very close to the O(n3) found through the theoretical analysis.

**Algorithm 2**

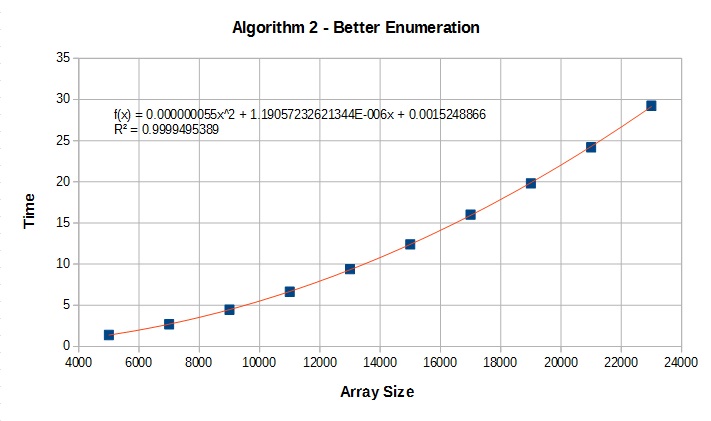
1. Individual and average running times:



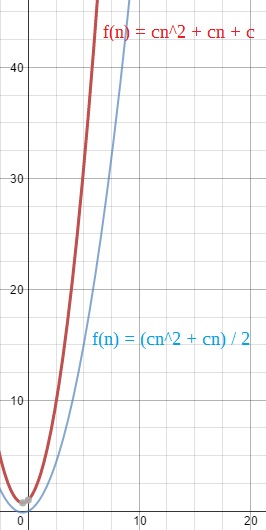
2. Achieving an R2 = 0.9999495389, a quadratic trendline proved the best fit for this data set. The equation of the trendline is calculated as . See the complete plot in part 3 below.



3. Regression model of average runtimes using LibreOffice Calc:



4. There are no major discrepancies between the theoretical and experimental analyses insofar as both examinations are bound by a quadratic runtime complexity and any difference between the two is dependent on constant factors. The graph shown below illustrates this fact by showing that both functions achieve an end behavior equivalent to a quadratic function.

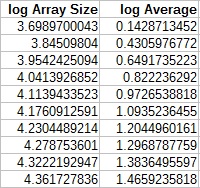
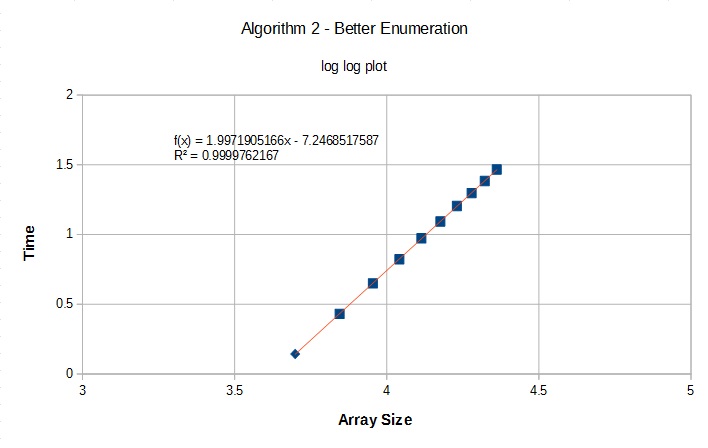


5. Using the formula derived from the regression model in parts 2 and 3

We can solve the largest input size the algorithm can solve in 10, 30 and 60 seconds by substituting *y* with the appropriate time value and then solving for *x*. Doing so returns the following results:

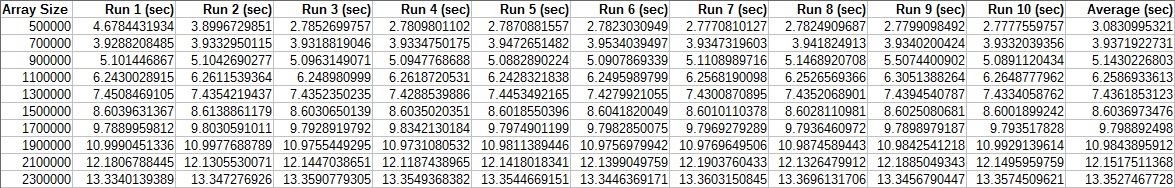
|  |  |
| --- | --- |
| **Time (seconds)** | **Largest Solvable Input Size** |
| 10 | 13472 |
| 30 | 23343 |
| 60 | 33017 |

6. Log-log plot of algorithm 2 using LibreOffice Calc:

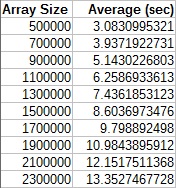
The linear function that best fits this data is . The slope of this linear equation is approximately 1.997 which suggests the runtime complexity for this algorithm is , illustrating a mere 0.1 difference in power values between the theoretical and experimental analysis.

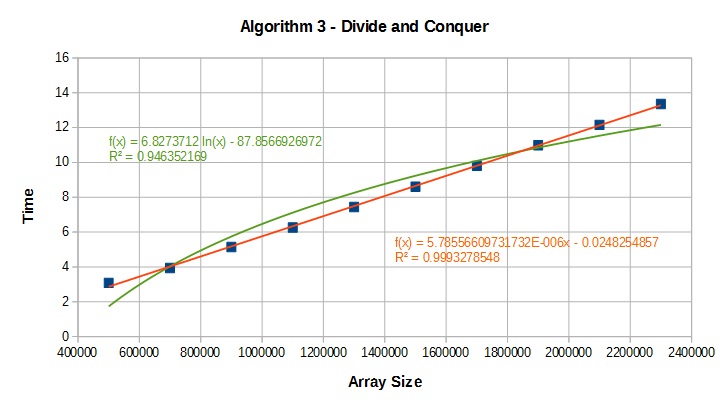
**Algorithm 3**

1. Individual and average running times:

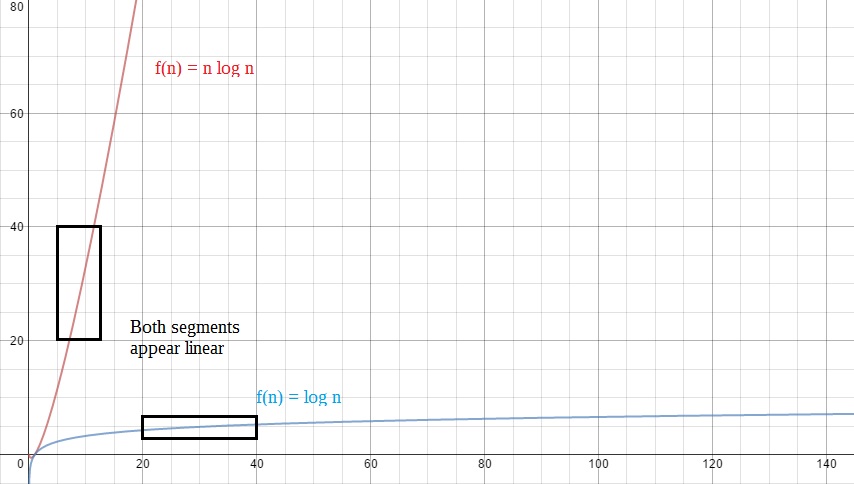


2. While a logarithmic trendline achieves an R2 = 0.946352169, a linear trendline reaches an R2 = 0.9993278548 and appears to be the best fit for this data set. The equation of this trendline is calculated as . See the complete plot in part 3 below.



3. Regression model of average runtimes using LibreOffice Calc:

4. The theoretical analysis asserts that this algorithm should be asymptotically bounded by a modified logarithmic function of the order . The experimental analysis, however, seems to suggest that the algorithm is bounded by a linear function. Given the rather limited domain of the data set, it is likely that the linear facade presented by this analysis is due to the fact that only a small portion of the logarithmic function is being examined. Looking at the graph below, we can see that if the domain of the given data set is small enough, then the true nature of the graph cannot adequately be understood.



We can prove the error in the experimental analysis further using induction to manifest a contradiction. Thus, if we assume that , then our hypothesis is that . The inductive step is then:

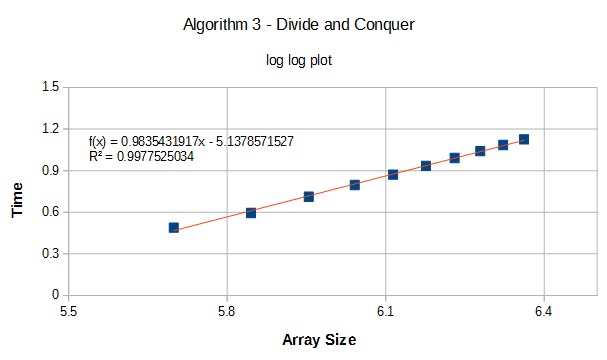
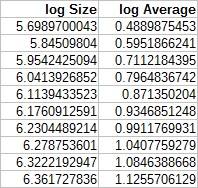
This final inequality does not hold for all and all and so, by definition,

5. The formula derived from the regression model in parts 2 and 3 that most closely models the domain of the given data set is

We can solve the largest input size the algorithm can solve in 10, 30 and 60 seconds by substituting *y* with the appropriate time value and then solving for *x*. Doing so returns the following results:

|  |  |
| --- | --- |
| **Time (seconds)** | **Largest Solvable Input Size** |
| 10 | 1732730 |
| 30 | 5189610 |
| 60 | 10374900 |

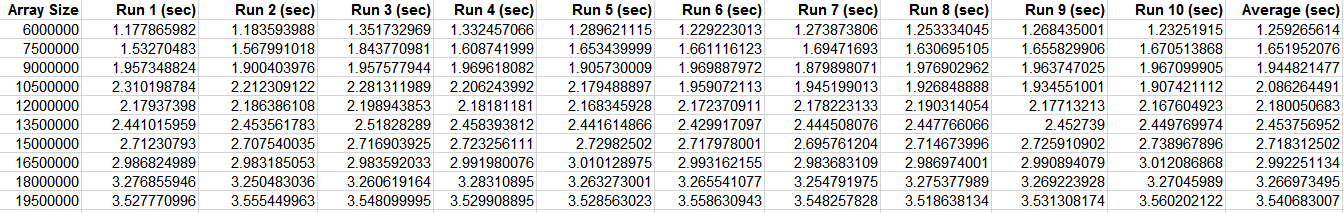
6. Log-log plot of algorithm 3 using LibreOffice Calc:



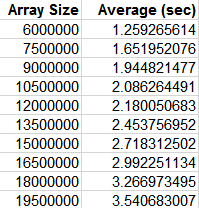
The linear function that best fits this data is . The slope of this linear equation is approximately 0.98, however, due to rounding that occurs during calculation, it is safe to say that the slope is approximately 1. This suggests the runtime complexity for this algorithm is bound by , showing that theoretical and experimental analysis are roughly equivalent.

**Algorithm 4**

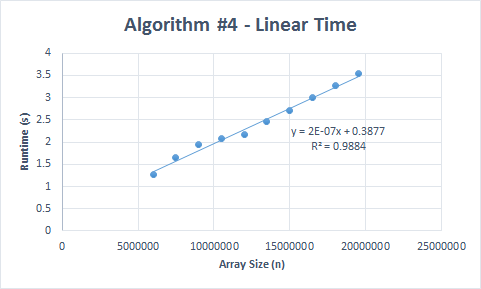
1. Individual and average running times:



2. Achieving an R2 = 0.9884, a linear trendline proved the best fit for this data set. The equation of the trendline is calculated as . See the complete plot in part 3 below.



3. Regression model of average runtimes using Microsoft Excel:



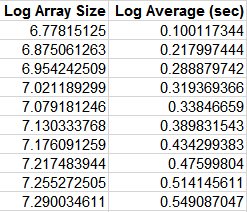
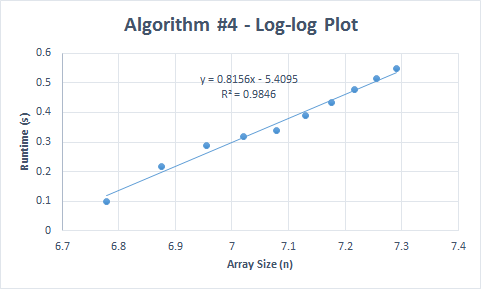
4. There were no major discrepancies between the experimental and theoretical running times. Runtime increases in direct proportion to the array size, reinforcing that the linear approach has an O(n) runtime.

5. The linear function from the plot is:

We can substitute the values 10 seconds, 30 seconds, and 60 seconds in the equation for the value of y and then solve for x in order to find the largest input that can be solved in the associated amount of time.

|  |  |
| --- | --- |
| **Time (seconds)** | **Largest Solvable Input Size** |
| 10 | 48061500 |
| 30 | 148061500 |
| 60 | 298061500 |

6. Taking the log of the array sizes and the run time averages gives us a table we can use to make the log-log plot:



In the log-log plot, the runtime increases at a linear rate as array size increases. This is expected, and fits the O(n) runtime observed in the experimental and theoretical data. The equation for the linear progression is . The slope (m) of the line is 0.8156, which rounds up to 1. The runtime is which is which is essentially O(n).

**Log log plot of all four algorithms**

