Group 11 - Jesse Thoren, Shawn Hillyer, Jason Goldfine-Middleton
October 16, 2016
CS 325 Project 1: Maximum Sum Subarray

Theoretical Run-time Analysis:

Algorithm 1: Enumeration

A1 - Pseudocode:

```
mss_enumerative(inputArray)
    Initialize variables: Maximum sum found, Lower/Upper indices of Max subarray found
    For i in length(inputArray)                      //Current lower index being tested
        For j between i and length(inputArray)       //Upper index being tested
            Sum elements in inputArray from index i to j inclusive
            If this sum is greater than the max sum found so far
                Update max sum with current sum
                Update lower index with i
                Update upper index with j
    Return Max sum found, Lower/Upper indices of the max sum subarray.
```

A1 - Asymptotic running time:

The total number of iterations processed by the outside (i) loop is n, where n is length(inputArray).
The total number of iterations processed by the inner (j) loop is n-i.

This yields a total number of iterations = n + (n-1) + (n-2) + … + (1), where each term corresponds to an increment in i from the outside loop.
This can be simplified by using an arithmetic series, to get a total number of iterations = n(n+1)/2.

However, each iteration is of variable length, because it is necessary to loop from index i to j in order to calculate the value of the sum. The n iterations that correspond to i = 0 account for n iterations to compute the sum (for i=0, j=n), n-1 iterations (for i=0, j=n-1), and so on… so there is actually a number of arithmetic sequences imbedded inside the arithmetic sequence already computed. This can be visualized on a table:

Number of iterations to compute sum for given values of lower and upper bounds, i and j:

	i=0	i=1	i=2	...	i=n-2	i=n-1	i=n
j=0	1	2	3	...	n-2	n-1	n
j=1	n/a	1	2	...	n-3	n-2	n-1
j=2	n/a	n/a	1	...	n-4	n-3	n-2
...	...	...	...	...	...	...	...
j=n-2	n/a	n/a	n/a	...	1	2	3
j=n-1	n/a	n/a	n/a	...	n/a	1	2
j=n	n/a	n/a	n/a	...	n/a	n/a	1

The total number of iterations for a given arrayLength n can then be computed by using a double sum:

$$\sum_{i=1}^{n}\left(\sum_{j=1}^{i} j\right) = \frac{1}{6}\, n\,(n+1)\,(n+2)$$

Which is expanded to $\frac{n^3}{6} + \frac{n^2}{2} + \frac{n}{3}$

Hence, Algorithm 1 runs in $\Theta(n^3)$.

Algorithm 2: Better Enumeration

A2 - Pseudocode:

```
mss_better_enum(inputArray)
    Initialize variables: Max sum found, Lower and Upper indices of Max subarray found
    For i in length(inputArray)                    //Current lower index being tested
        Initialize a running sum                   //Stores sum of all elements starting at i
        For j between i and length(inputArray)          //Current upper index being tested
            Add element at inputArray[j] to running sum
            If the running sum is greater than the max sum found so far
                Update max sum
                Update lower index with i
                Update upper index with j
    Return max sum found, Lower/Upper indices of the max sum subarray.
```

A2 - Asymptotic Running Time:

The better enumeration algo is improved because it doesn't have to loop through values each iteration in order to compute the sum. It accomplishes this by keeping a running sum for a given i value as it loops through the j values greater than it, which yields the same sum as the final loop did in A1, yet only requires 1 calculation rather than O(n) calculations. Because of this, only the initial calculations from A1 apply, as follows:

The total number of iterations processed by the outside (i) loop is n, where n is length(inputArray).
The total number of iterations processed by the inner (j) loop is n-i.

This yields a total number of iterations = n + (n-1) + (n-2) + … + (1), where each term corresponds to an increment in i from the outside loop.
This can be simplified by using an arithmetic series, to get a total number of iterations = n(n+1)/2.

Since there are a constant number of operations for each of the n(n+1)/2 iterations, the asymptotic running time of the Better Enumeration algorithm is $\Theta(n^2)$.

Algorithm 3: Divide and Conquer

A3 - Pseudocode:

```
mss_divconq(inputArray):
    resultsArray = mss_divconq_helper(inputArray)
    return max sum found, Lower/Upper indices of the max sum subarray.
```

//The helper function returns a lot of values back to the calling function. These are in the form:
[max prefix value - Sum of the max prefix, max prefix end index - Terminal index of the max prefix, max suffix value - Sum of the max suffix, max suffix start index - Initial index of the max suffix,
max sum value - Sum of the MSS, max left index- Initial index of the MSS,
max right index - Terminal index of the MSS, total sum value - sum of all elements in inputArray]

```
mss_divconq_helper(a):
    If length of a is 1
        return [a[0],0,a[0],0,a[0],0,0,a[0]] //element value or element index where needed
    Split a into 2 equal arrays, left and right, by computing a middle value and slicing the array.
    Get results from mss_divconq_helper(left)
    Get results from mss_divconq_helper(right)
    Assign the max prefix value to the maximum value of either the max prefix value of left, or the max prefix value of right+the total sum value of left.
    Assign the max prefix end index to the max prefix end index of left, or the max prefix end index of right plus the amount of elements in left, as the result from the previous option requires.
    Assign the max suffix value to the maximum value of either the max suffix value of right, or the max suffix value of left+the total sum of right.
    Assign the max suffix start index to the max suffix start index of right plus the amount of elements in left, or the max suffix start index of left, as the result from the previous option requires.
    Assign the total sum value to the total sum value of left + the total sum value of right.
    Assign the max sum value to the greatest of max sum value of left, max sum value of right, or max suffix value of left + max prefix value of right.
    Assign the max left index to the max left index of left, max left index of right + amount of elements in left, or the max suffix start index of left, as the result from previous requires.
    Assign the max right index to the max right index of left, max right index of right + amount of elements in left, or the max prefix value of right + amount of elements in left, as the result from the operation before last requires.
```

A3 - Asymptotic Running Time:

The divide and conquer algorithm has recurrence T(n) = 2T(n/2) + O(1). We determined this because there are 2 function calls to arrays half the size of the original input array, and a number of constant time operations: comparisons, assignments, and determining the maximum of at most 3 objects.

Solving the recurrence using the master method:
*Let a ≥ 1 and b > 1 be constants, let f(n) be a function and T(n) be defined on the non-negative integers by the recurrence T(n) = a*T(n/b) + f(n) where n/b = floor(n/b) or ceil(n/b). Then T(n) can be bounded asymptotically as :*
Case 1: If f(n) = $O(n^{logb(a)-})$ for some >0, then T(n) = $\Theta(n^{logb(a)})$
Note that a = 2, b = 2, f(n) = c, $n^{log2(2)} = n^1$
It follows that c = $n^{1-\epsilon}$ for some ϵ >0 (specifically ϵ = 1)
Hence, T(n) = Θ(n) by the master method.

Algorithm 4: Linear Time

A4 - Pseudocode:

```
mss_linear(inputArray)
    Initialize max lower index, max upper index to the index of the first positive in inputArray
    Initialize max sum to the inputArray value at that index.

    Initialize curr_sum to 0 and current lower index to max lower index.
    Loop through elements in inputArray, starting at max lower index.
        Add the array value at the current index to the current sum
        If the current sum is greater than max sum
            Set max sum to current sum
            Set max lower index to current lower index
            Set max upper index to the current index in the loop.
        If the current sum is less than 0
            Reset current sum to 0
            Set the max lower index to the current loop index + 1.
```

A4 - Asymptotic Running Time:

A4 loops through each of the array values of the input array at most once. For each of these iterations, at most a constant amount of comparisons and assignments occur, so the asymptotic running time is indeed Θ(n).

Testing:

Algorithms were tested using a fair amount of automation. Using p1correct.py, we automated the process of reading in arrays from a text file and printing the results out. We also configured the file to allow for a custom file input and output name. We started out by using the test cases provided in the Project 1 Module on Canvas. These tests were actually helpful for us in identifying a bug in algorithm 4.

Verification of the tests is done by manually inspecting the input arrays and comparing the output to the expected result. Creating additional test cases is easy because we can just create plaintext files with arrays in a text file without modifying the code base.

To run a custom test of correctness, simply run:

```
python3 p1correct.py [in_filename out_filename]
```

at the command prompt, where the input file is formatted as plaintext arrays separated by newlines. For example:

```
# Start of file
[1,2,3,4,5]
[-1,2,5,-30]

# EOF
```

Experimental Analysis:

Our experimental analysis is executed using the `p1analysis.py` script. The file allows for easily configuring the the n-values passed into the random array generator and then the algorithm itself. Timing of the algorithm includes no print statements or file in-out as the results are written to *-results.csv files in between calls to the timing function.

Average running times

Average Running Times are presented below for each algorithm.These are the first 10 values only for each (To conserve space), but we ran the data on a total of 50 n-values for every algorithm.

Algorithm 1

A1 Enumerative	
n	time(sec)
10	4.77E-05
20	0.000230798
30	0.000628297
40	0.001313026
50	0.002349441
60	0.003875227
70	0.005756618
80	0.008317
90	0.011900549
100	0.015432082

Algorithm 2

A2 Better Enumerative	
n	time(sec)
10	1.20E-05
60	0.000259038

110	0.000814497
160	0.001665552
210	0.002863832
260	0.004348409
310	0.006357484
360	0.008623296
410	0.011313725
460	0.01439789

Algorithm 3

A3 DnC	
n	time(sec)
10	3.13E-05
1010	0.003482363
2010	0.007026087
3010	0.01046889
4010	0.013899159
5010	0.017362706
6010	0.021179277
7010	0.024381593
8010	0.027711034
9010	0.031269578

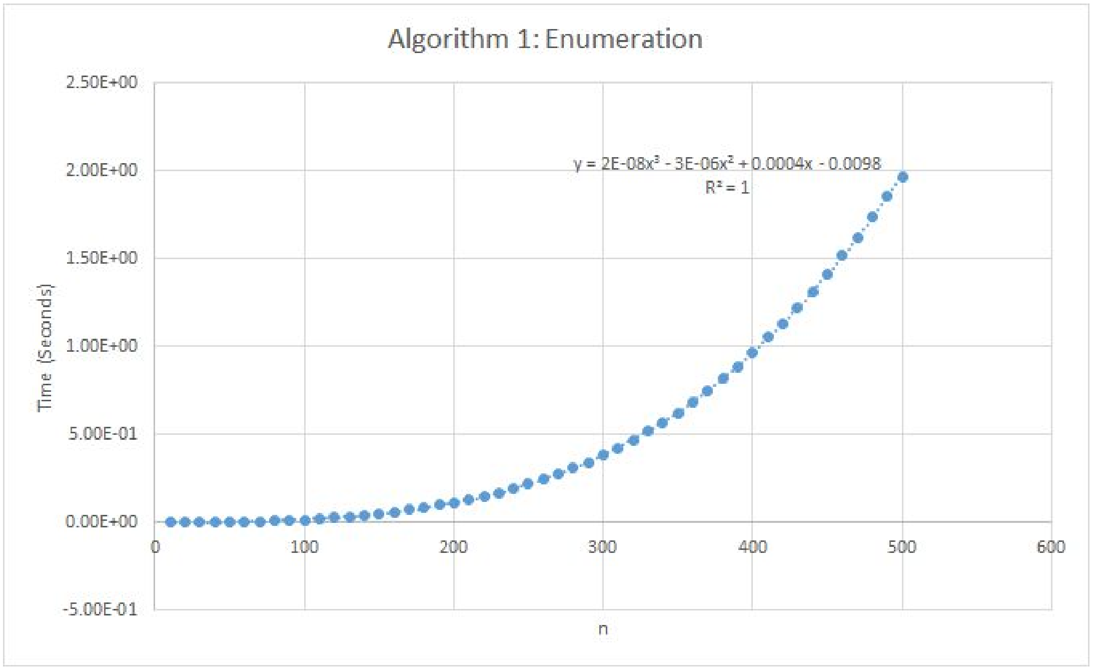
Algorithm 4

A4 Linear	
n	time(sec)
10	3.17E-06
2510	0.000476587
5010	0.000991118

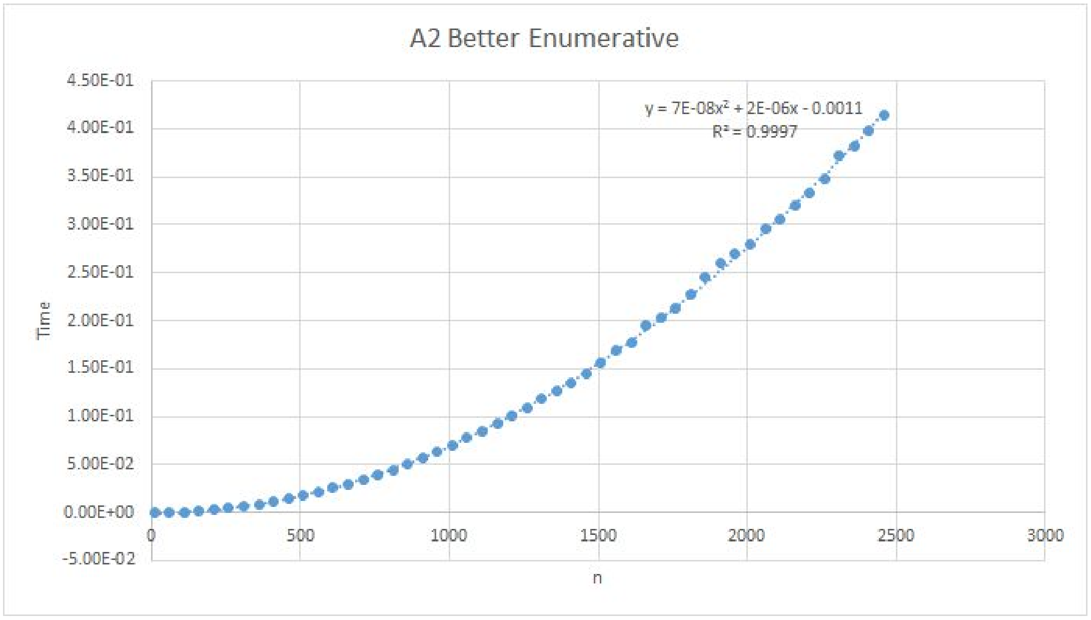
7510	0.001471188
10010	0.001964212
12510	0.002442324
15010	0.002997145
17510	0.003407985
20010	0.003906856
22510	0.004415244

Algorithm Plots

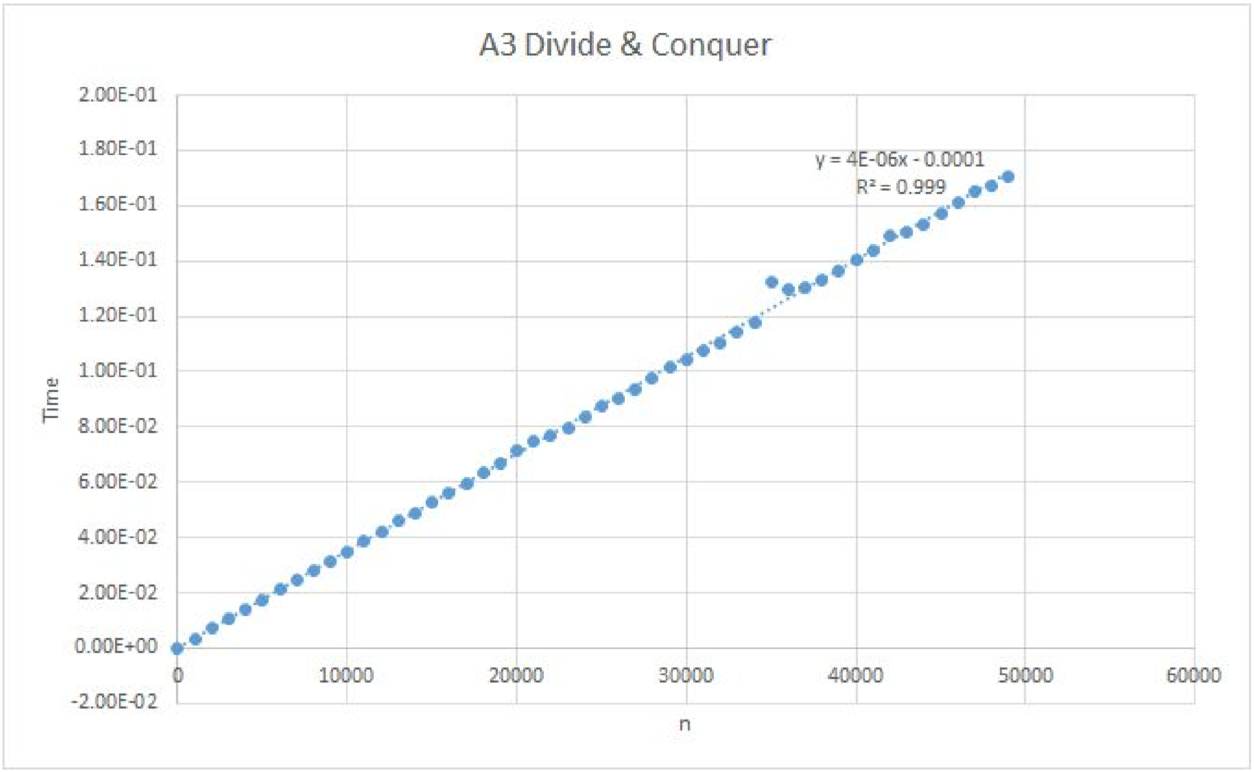
Algorithm 1



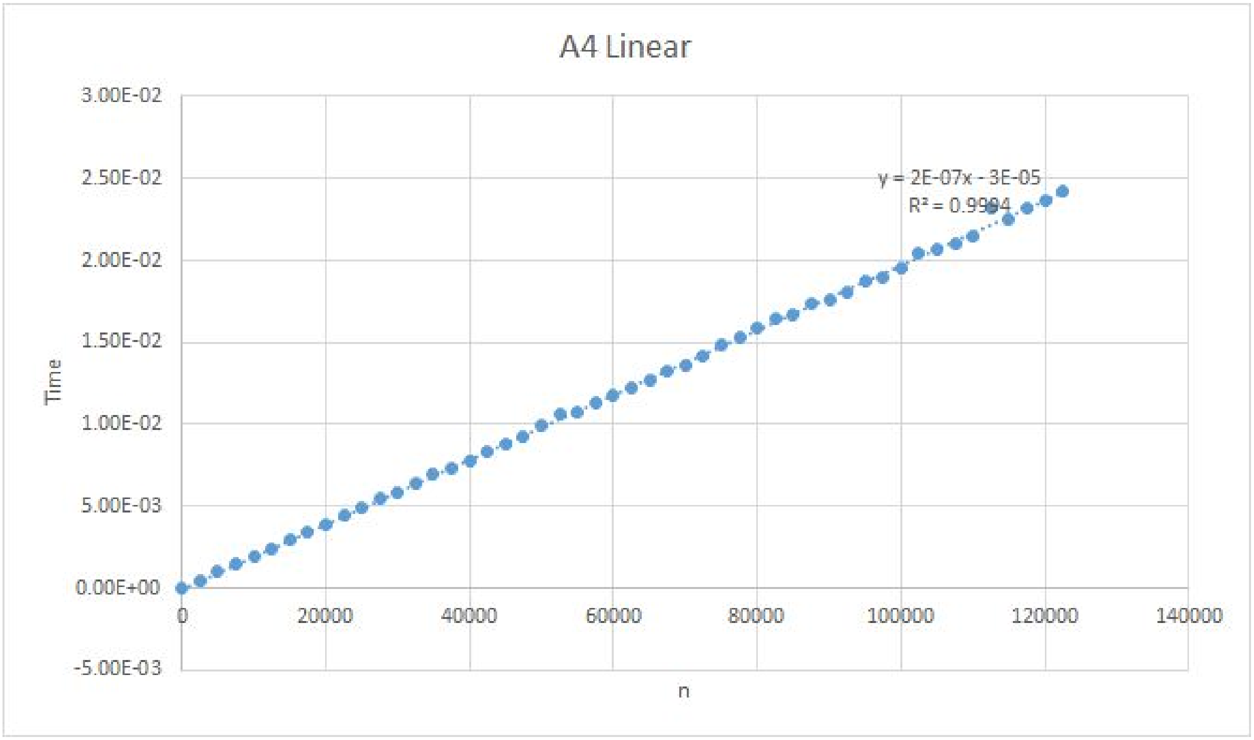
Algorithm 2



Algorithm 3



Algorithm 4



Best Fit Functions

For each function, ‘n’ is represented by x.

Algorithm 1 :Enumeration

The function that fits is a 3rd-degree polynomial function. This is cubic growth.

$y = 2E\text{-}08x^3 - 3E\text{-}06x^2 + 0.0004x - 0.0098$

$R^2 = 1$

Algorithm 2

The function that fits is a 2nd-degree polynomial function. This is quadratic growth.

$y = 7E\text{-}08x^2 + 2E\text{-}06x - 0.0011$

$R^2 = 0.9997$

Algorithm 3

The Divide and Conquer algorithm has for its best-fit a first-degree polynomial function. This is linear growth.

$y = 4E\text{-}06x - 0.0001$

$R^2 = 0.999$

Algorithm 4

The linear function also has, unsurprisingly, a best-fit function of a first-degree polynomial function. This is linear growth.

$y = 2E\text{-}07x - 3E\text{-}05$

$R^2 = 0.9994$

Comparison to Theoretical Analysis

All of our theoretical analysis matched the results from our experimental results. As indicated in the prior section, the R-square values for the four functions are all very close to 1.

We used a total of 50 different sizes of n-element arrays in our analysis. Each of these arrays was run through the algorithm 20 times and calculated as an average. We have high confidence that both our analysis and experimental results are accurate representations of the algorithms described and neither are defective in their design.

Regression Predictions

We used Wolfram Alpha to crunch the numbers and solve for n and set equation = to 10, 30, and 1 minute each. We then confirmed it using the script found in p1predict.py by inspection of results. When n gets very large for the two linear growth functions, precision is dropped a bit.

For algorithm 1, the maximum number values you can run, n, in t=10, 30, and 60 seconds are n=838, n=1191, and n=1489, based on the function from prior section.

Algorithm 2 can execute within the times for n= 11944, n=20691, and n=29265 times respectively.

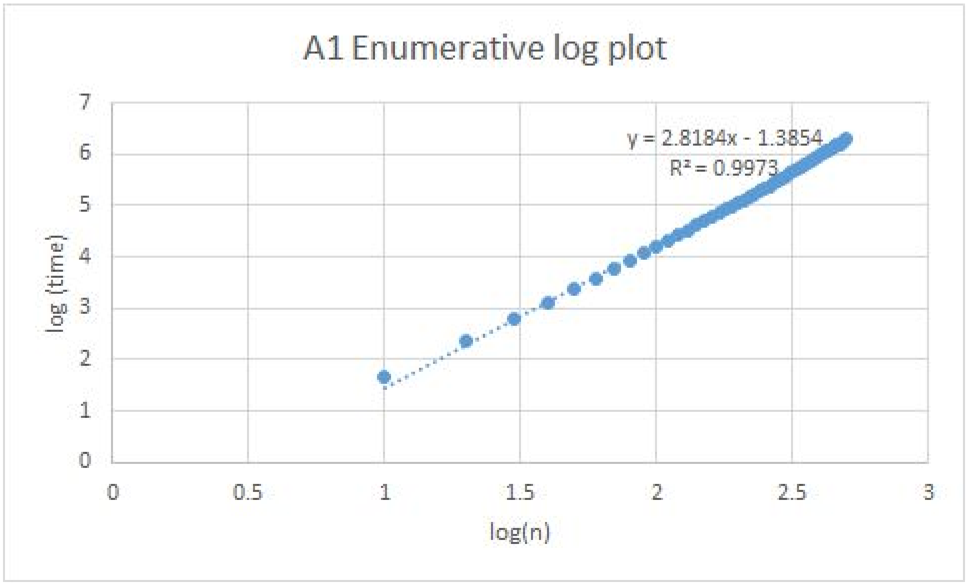
Algorithm 3 can execute n=~2,500,025, n=~7,500,025, and n=~15,000,025 times approximately.

Algorithm 4 can execute n=50,000,150, n=150,000,150 and n=300,000150 times approximately.

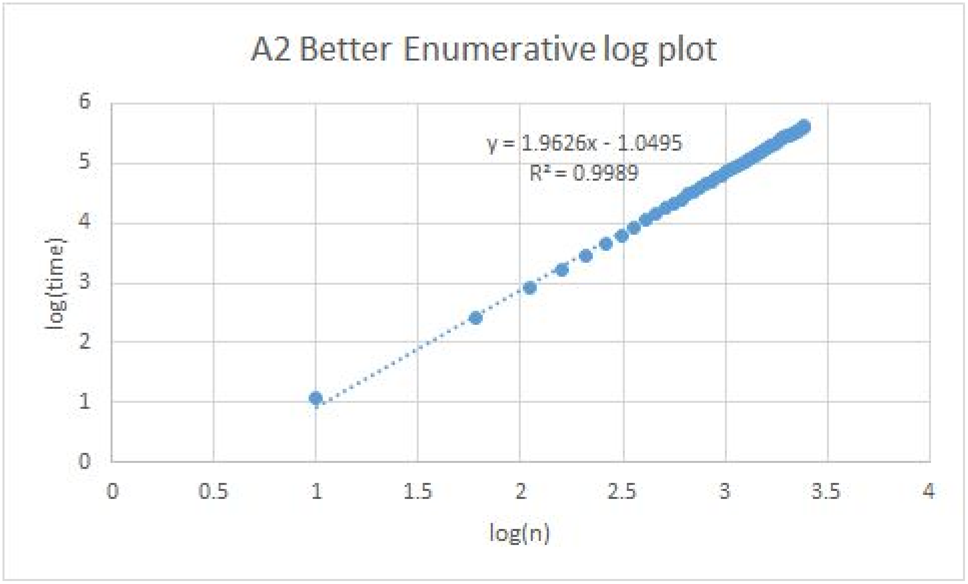
Log-Log Plot of Running Times

The log-log plot of running times are with a time scale of microseconds, rather than seconds, as taking the log of numbers this small ended up with negative values.

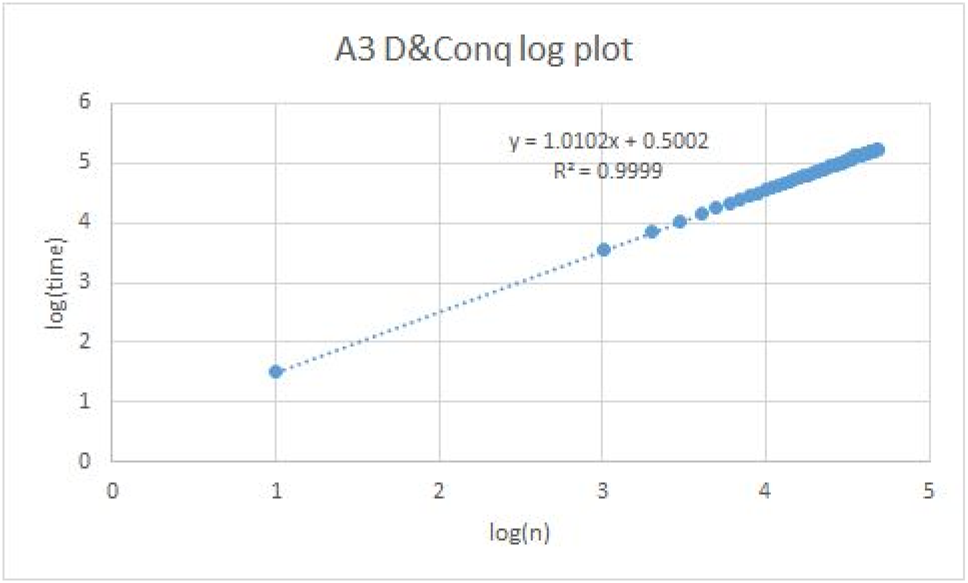
Algorithm 1



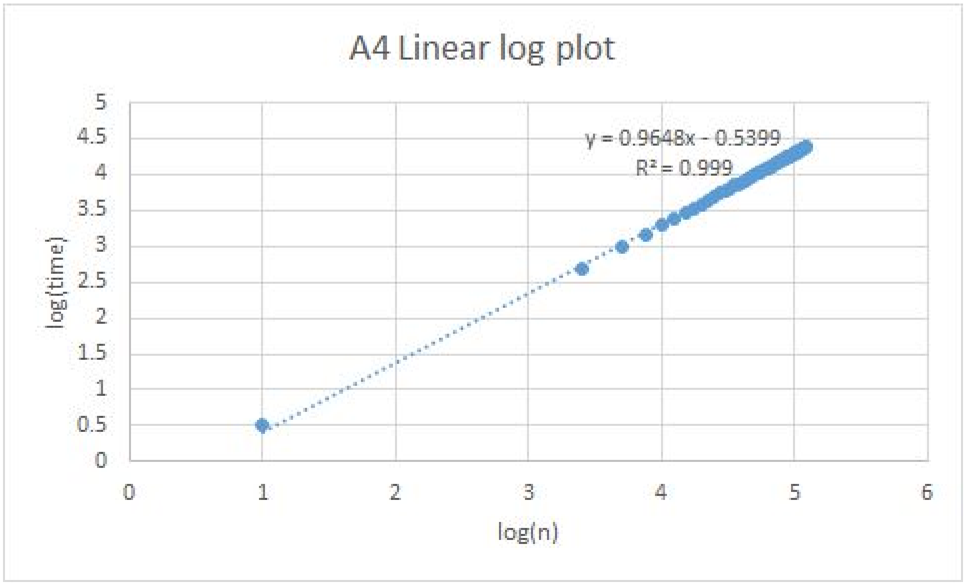
Algorithm 2



Algorithm 3

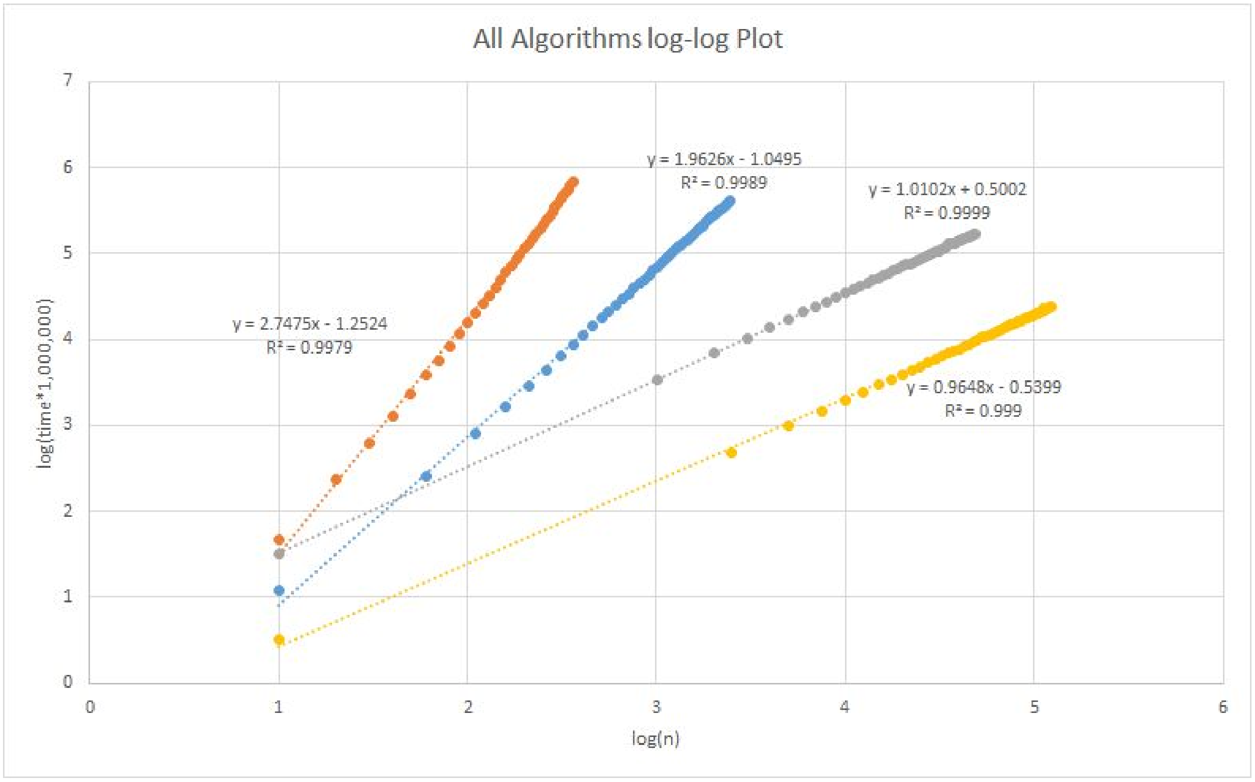


Algorithm 4



Combined Plot/Graph

All algorithms were plotted on a log-log plot together. Note that, again, time was multiplied by 1,000,000 so the time-scale is microseconds. As expected, the coefficient for the 3rd degree, 2nd degree, and 1st degree functions each are close to 3, 2, and 1 respectively.


All Algorithms log-log Plot
y = 2.7475x - 1.2524
R² = 0.9979
y = 1.9626x - 1.0495
R² = 0.9989
y = 1.0102x + 0.5002
R² = 0.9999
y = 0.9648x - 0.5399
R² = 0.999
log(time*1,000,000)
log(n)