Group 7 - Project 1

Team Members
Justin Kruse, Nicholas Koller, ~~Phillip Scott Cooper~~ (dropping class, did not contribute)

Introduction

For this assignment we were asked to design, implement, and analyze four different algorithms to solve the maximum subarray problem.

$$\underset{i \le j}{max} \sum_{k=i}^{j} A[K]$$

The four algorithms are: (1) Enumeration, (2) Better enumeration, (3) Divide and Conquer, and (4) Linear-time.

For programming, we decided to use Python 3 to solve code and run these four algorithms. The structure of our program is as follows:

project1.py
Main program, structured as two parts:

1. Calling all four algorithms to run data from MSS_Problems.txt.
2. Calling all four algorithms to run experimental analysis on arrays of various N sizes.

p1algorithms.py
File containing the functions for all four algorithms.

Testing
Correctness:
TestProblems.txt and TestResults.txt were used initially to ensure that the algorithms yielded identical results. Once we established that confidence, we ran our algorithms on MSS_Problems.txt.

Experimental Runs:
The Experimental Runs of each algorithm were accomplished accordingly:

1. An array consisting of 10 increasing values of N was used in creating 10 arrays consisting of random positive and negative integers. Since the algorithms required different magnitudes of N to get comparable data, a multiplier for N was used. The base values of N were:
   N = [25, 35, 50, 75, 100, 125, 250, 500, 750, 1000]
   The multipliers for N for each algorithm were:
   Algorithm 1: 1 * N
   Algorithm 2: 10 * N
   Algorithm 3: 1000 * N
   Algorithm 4: 10000 * N
2. Timers were used to time how long each algorithm ran on each value of N. The results were outputted to the console for visual feedback, as well as a file called MSS_ExperimentalResults.txt for ease in copying data for analysis.
3. Matlab was used in plotting and determining the regression equation for each algorithm.
4. Results of plots and regression functions were analyzed against theoretical results.
5. Microsoft Excel was used to show all 4 plots on a Log Log graph.

Group 7 - Project 1

Algorithm 1: Enumeration

Overview
This algorithm takes the brute force approach by checking every possible pair of (i,j) as a range of sums in the array and comparing it to the last largest sum that had been calculated. This requires looping through the array a total of 3 times. It runs twice to find all possible combination of (i,j) where $1 <= i < j <= n$ and then a third time to compute the total sum of the pair.

Theoretical Runtime Analysis
($O(n^2)$ pairs) * ($O(n)$ time to compute each sum) = $O(n^3)$ time

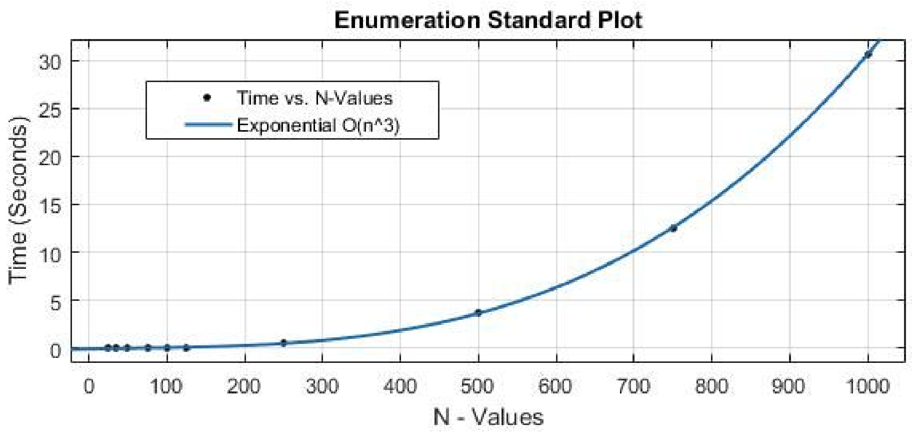
Pseudocode:
```
MAXSUBARRAY_Enum(array, low, high):
    for i in range(low, high+1, 1):
        for j in range(i, high+1):
            sa_sum = 0
            for ea in array[i:j+1]:
                sa_sum += int(ea)
            if sa_sum > max_sum:
                max_sum = sa_sum
                max_i = i
                max_j = j
    return max_i, max_j, max_sum
```

Experimental Analysis
1. *Average Runtime for n*:

N Values	Time (Seconds)
25	0.000501
35	0.002007
50	0.005517
75	0.01454
100	0.034091
125	0.061663
250	0.496319
500	3.6999
750	12.544902
1000	30.661715

Group 7 - Project 1

2. *Plot:*



3. *Function:*

   Type: Polynomial Order 3:

   $f(x) = p1*x^3 + p2*x^2 + p3*x + p4$

   Coefficients (with 95% confidence bounds):

   p1 = 3.478e-08 (3.305e-08, 3.65e-08)
   p2 = -5.532e-06 (-8.188e-06, -2.876e-06)
   p3 = 0.001469 (0.0003977, 0.00254)
   p4 = -0.06447 (-0.145, 0.01602)

   Asymptotic Run Time:O(n^3) is what our theoretical time estimates predicted and our plotting supports that analysis.

4. *Discrepancies:*

   We did not see any discrepancies between our theoretical prediction and actual data, as both are $O(n^3)$

5. *Regression model to determine largest input that can be solved:*

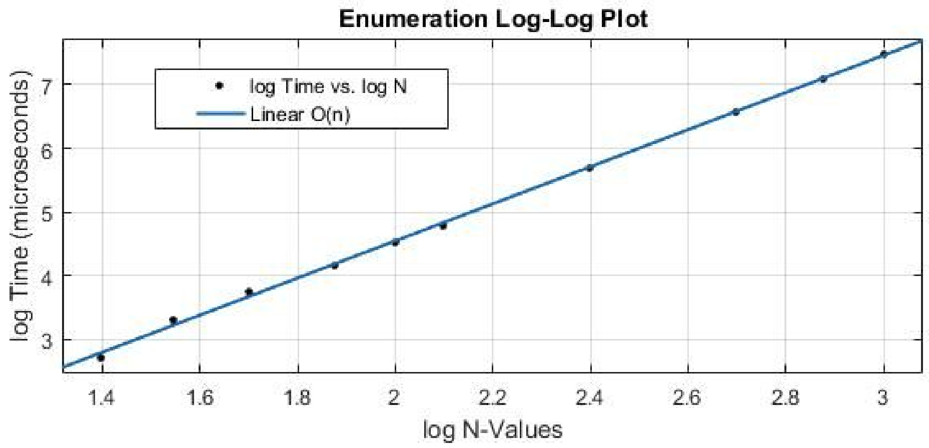
   Wolfram Alpha Equation:

   $Time=(0.00000003478*x^3)+(-0.000005532*x^2)+(0.001469*x)+(-0.06447)$

   a. 10 seconds: x = 695.901
   b. 30 seconds: x = 993.063
   c. 60 seconds: x = 1242.94

Group 7 - Project 1

6. *Log Log Plot*



Type: Polynomial Order 1

f(x) = p1*x + p2

Coefficients (with 95% confidence bounds):

```
p1 =       2.917  (2.842, 2.992)
p2 =      -1.288  (-1.454, -1.122)
```

Algorithm 2: Better Enumeration

Overview

This algorithm is a slightly different approach on our previous brute force attempt. It improves upon it by checking every possible pair of (i,j) as a range of sums in the array against future combinations of contiguous elements and storing the largest sum. This requires looping through the array only 2 times as opposed to 3 in the previous version of enumeration. This improvement comes from the logic that we start with the largest range of array and its possible sum and continue to check the sums of smaller arrays until we find one that has the greatest sum.

Theoretical Runtime Analysis:

(O(n) i-iterations)*(O(n) j-iterations)*(O(1) time to update sum) = O(n^2)

Pseudocode:

```
def MAXSUBARRAY_BetterEnum(array, low, high):
    max_sum, max_i, max_j
    for i in range(low, high+1, 1):
        sa_sum = 0
        for j in range(i, high+1):
            sa_sum += int(array[j])
            if sa_sum > max_sum:
                Set max_sum, max_i, max_j to current sum max, i, and j values
    return max_i, max_j, max_sum
```

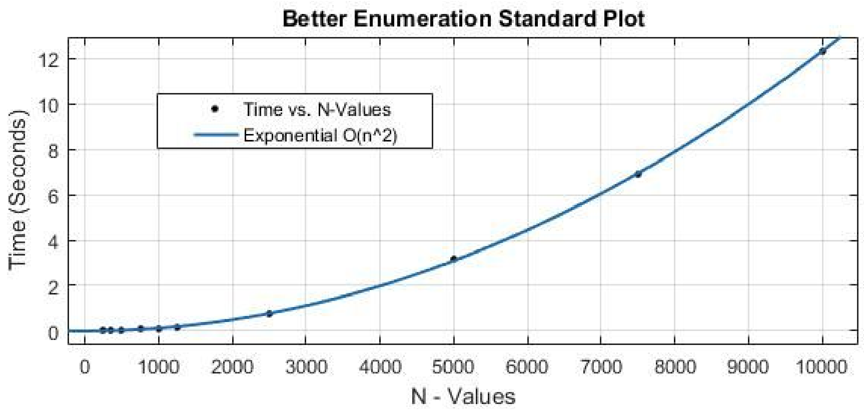
Experimental Analysis

1. *Average Runtime for n:*

N Values	Time (Seconds)
250	0.010025
350	0.016042
500	0.028577
750	0.06668
1000	0.12231
1250	0.193991
2500	0.764028
5000	3.136839
7500	6.905722
10000	12.339972

Group 7 - Project 1

2. *Plot:*



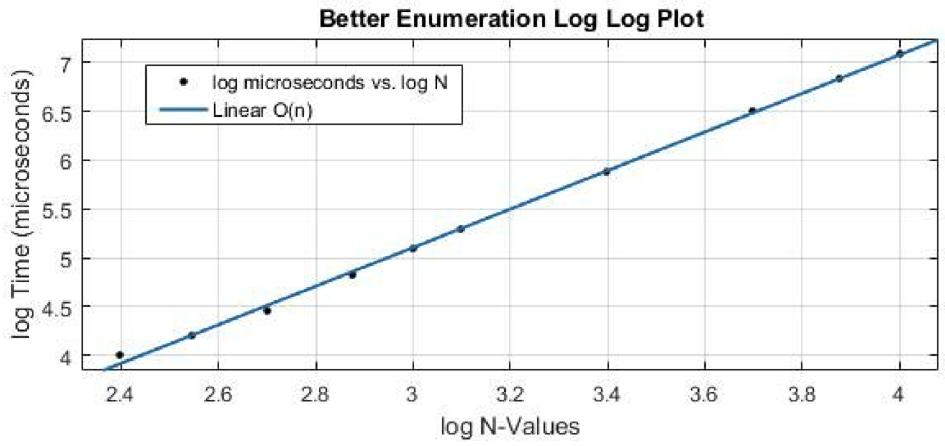
3. *Function:*
   Type: Polynomial Order 2
   $f(x) = p1*x^2 + p2*x + p3$
   Coefficients (with 95% confidence bounds):
   p1 = 1.227e-07 (1.204e-07, 1.249e-07)
   p2 = 6.409e-06 (-1.543e-05, 2.825e-05)
   p3 = -0.004089 (-0.03394, 0.02576)
   Asymptotic Run Time: $O(n^2)$ is what our theoretical time estimates predicted and our plotting supports that analysis.

4. *Discrepancies*
   We did not see any discrepancies between our theoretical prediction and actual data since both fit $O(n^2)$ line.

5. *Regression model to determine largest input that can be solved:*
   Wolfram Alpha Equation:
   $Time=(0.0000001227*x^2)+(0.000006409*x)+(-0.004089)$
   a. 10 seconds: x = 9003.48
   b. 30 seconds: x = 15611.4
   c. 60 seconds: x = 22087.9

Group 7 - Project 1

6. *Log Log Plot:*



Type: Polynomial Order 1

f(x) = p1*x + p2

Coefficients (with 95% confidence bounds):

p1 = 1.972 (1.919, 2.025)

p2 = -0.8105 (-0.9806, -0.6404)

Algorithm 3: Divide and Conquer

Overview:
This algorithm divides the problem into logical sub problems either the greatest sum sub array is contained in the left half, right half or crosses int to both. To solve these subproblems we used developed a helper function to determine the crossing array and sum whenever it crosses across mid for the left or right arrays. As the recursions unravel the largest sum of the selected array will be found utilizing the crossing array function and base cases to finish calculating left, right and across.

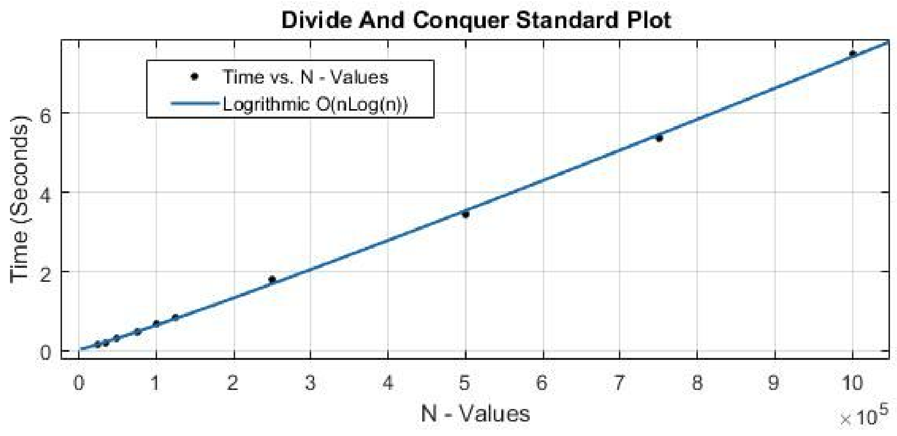
Theoretical Runtime Analysis:
(O(n) time for non-recursive work) * (O(log n) depth) = O(n log n)

Pseudocode:
```
1) Base Case: If high value == low value return array[low value]
2) Divide the given array in two halves
3) Return the maximum of following three
        a) Maximum subarray sum in left half (Make a recursive call)
        b) Maximum subarray sum in right half (Make a recursive call)
        c) Maximum subarray sum such that the subarray crosses the midpoint
```

Experimental Analysis:
1. Average Runtime for n:

N Values	Time (Seconds)
25000	0.144865
35000	0.213569
50000	0.306814
75000	0.487336
100000	0.670283
125000	0.83121
250000	1.79427
500000	3.434632
750000	5.386322
1000000	7.510968

2. Plot:



3. Function:

Type: N Log(N) model:

f(x) = b * x * log(x) + c

Coefficients (with 95% confidence bounds):

b = 5.356e-07 (5.247e-07, 5.464e-07)

c = 0.0295 (-0.03478, 0.09379)

Asymptotic Run Time: Logarithmic - O(n log(n)) is what our theoretical time estimates predicted and our plotting supports that analysis.

4. Discrepancies:

We did not see any discrepancies between our theoretical prediction and actual data since both fit O(n log(n)) line.

5. Regression model to determine largest input that can be solved:

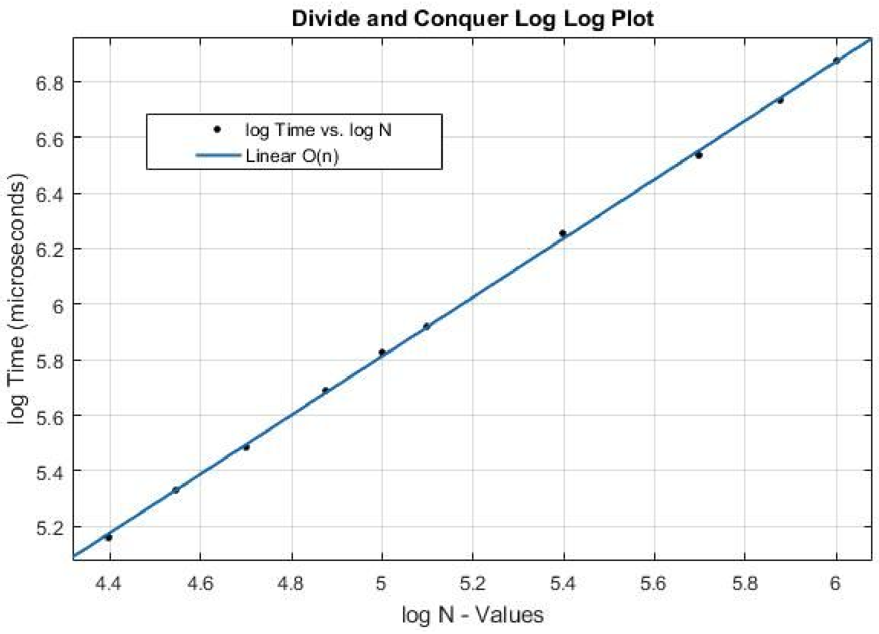
Wolfram Alpha Equation:

Time = 0.0000005356*x*log(x) + (0.0295)

a. 10 seconds: x = 1320840
b. 30 seconds: x = 3699920
c. 60 seconds: x = 7097740

Group 7 - Project 1

6. Log Log Plot:



Linear model Poly1:

f(x) = p1*x + p2

Coefficients (with 95% confidence bounds):

```
p1 =       1.06  (1.043, 1.077)
p2 =     0.5129  (0.4237, 0.6021)
```

Group 7 - Project 1

Algorithm 4: Linear Time

Overview

Our linear time algorithm is an implementation of Kadane's algorithm which looks for all positive contiguous segments and compares it to the running max sum and if it is greater we set the max sum to this new segments sum. Since this is running in linear when once the sum array drops negative it resets to zero and iterates the start position and tries to find a max sum larger than the current stored. One general bug with our version of this algorithm is that it does not function for an all negative array (it will return zero), but that case falls outside the scope of this assignment.

Theoretical Runtime Analysis

Computing MaxSubarray(a[1, ... ,n]) and Curr_Max(a[1, ... ,n] from MaxSubarray(a[1, ... , n-1]) and Curr_Max(a[1, ... , n-1]) takes O(1) time --> O(n) things to compute = O(n) time

Pseudocode:

```
def MAXSUBARRAY_Linear(array, low, high):
    max_sum = -99999999
    max_i = 0
    max_j = 0
    temp = 0
    curr_max = 0

    for i in range(low, high+1, 1):
        curr_max += int(array[i])
        if max_sum < curr_max:
            max_sum = curr_max
            max_i = temp
            max_j = i
        if curr_max < 0:
            curr_max = 0
            temp = i+1
    return max_i, max_j, max_sum
```

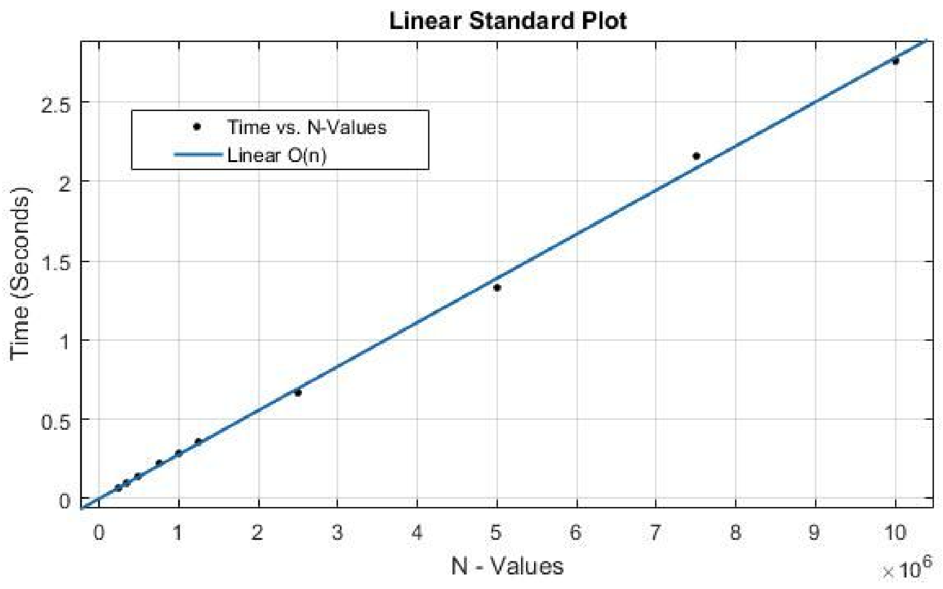
Group 7 - Project 1

<u>Experimental Analysis</u>

1. Average Runtime for n:

N Values	Time (Seconds)
250000	0.071189
350000	0.100307
500000	0.143382
750000	0.218079
1000000	0.283756
1250000	0.353439
2500000	0.667273
5000000	1.326527
7500000	2.164281
10000000	2.755826

2. Plot



3. Function

Type: Linear model Polynomial Degree 1:

f(x) = p1*x + p2

Coefficients (with 95% confidence bounds):

p1 = 2.781e-07 (2.695e-07, 2.867e-07)

p2 = -0.0007634 (-0.03838, 0.03686)

Asymptotic Run Time: Linear - O(n) is what our theoretical time estimates predicted and our plotting supports that analysis.

Group 7 - Project 1

4. *Discrepancies*

We did not see any discrepancies between our theoretical prediction and actual data since they both fit the O(n) line.

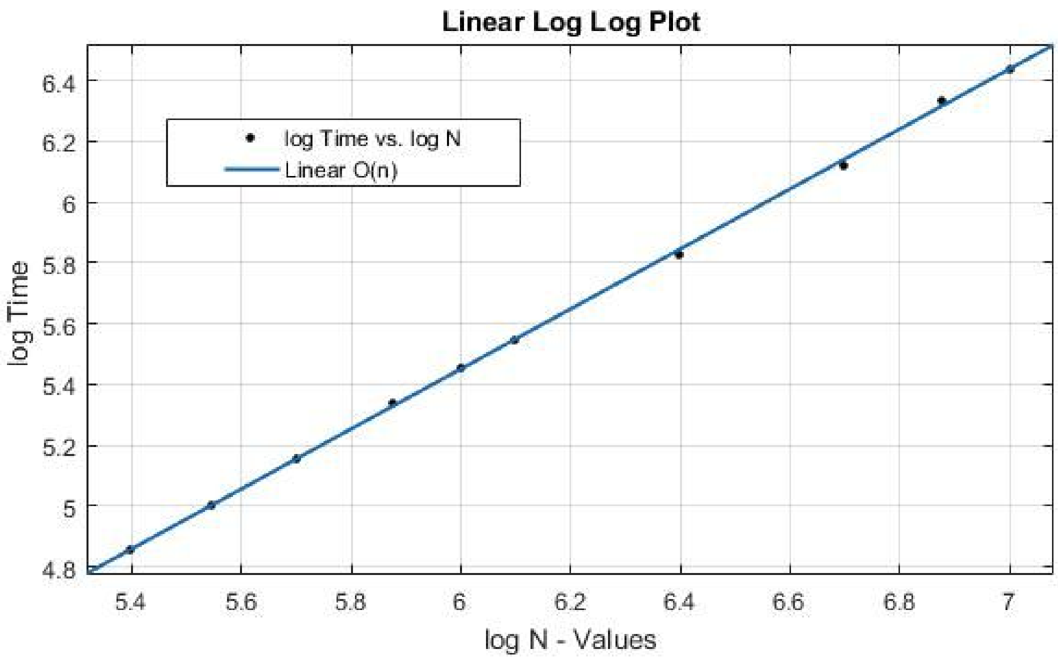
5. *Regression model to determine largest input that can be solved:*

   Wolfram Alpha Equation:

   Time = (0.0000002781*x) + (-0.0007634)

   a. 10 seconds: x = 35961000
   b. 30 seconds: x = 107878000
   c. 60 seconds: x = 215752000

6. Log Log Plot



Type: Linear model Polynomial Degree 1:

f(x) = p1*x + p2

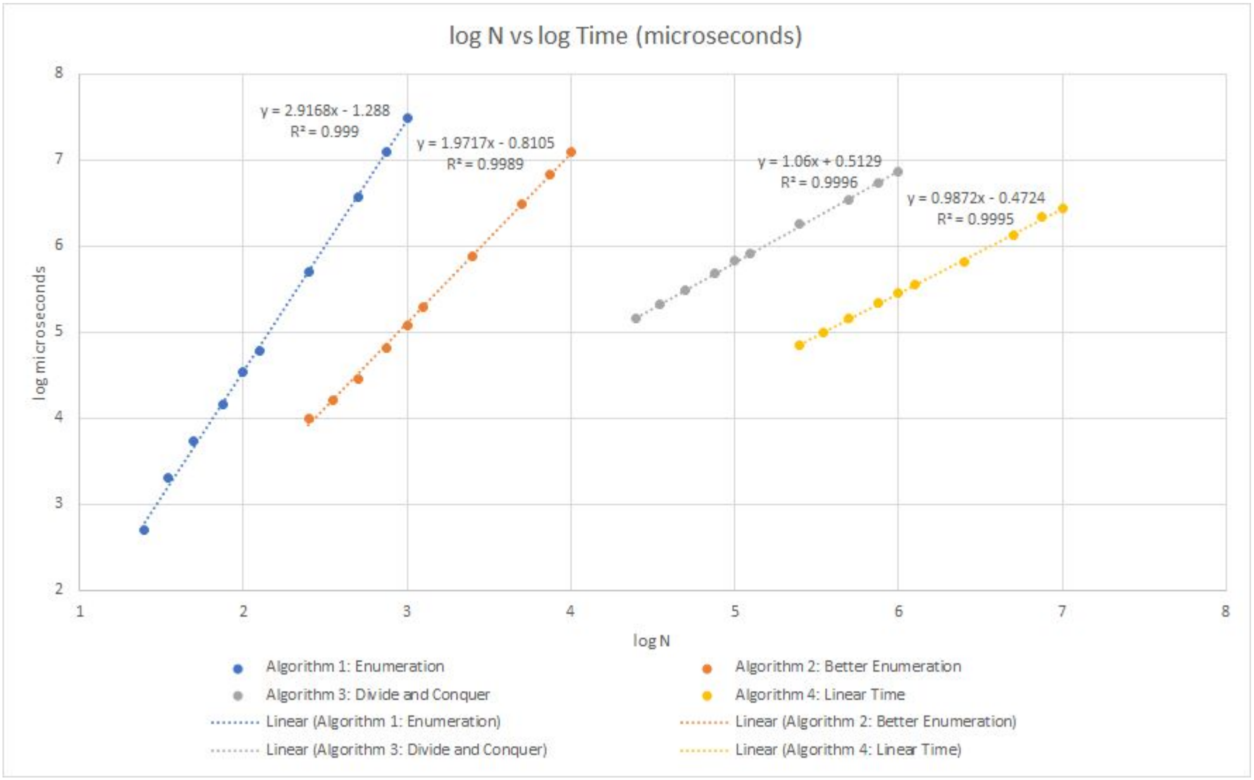
Coefficients (with 95% confidence bounds):

p1 = 0.9872 (0.9698, 1.005)

p2 = -0.4724 (-0.5798, -0.3651)

Group 7 - Project 1

All Algorithms Log Log Plot


log N vs log Time (microseconds)
y = 2.9168x - 1.288
R² = 0.999
y = 1.9717x - 0.8105
R² = 0.9989
y = 1.06x + 0.5129
R² = 0.9996
y = 0.9872x - 0.4724
R² = 0.9995
log microseconds
log N
Algorithm 1: Enumeration
Algorithm 2: Better Enumeration
Algorithm 3: Divide and Conquer
Algorithm 4: Linear Time
Linear (Algorithm 1: Enumeration)
Linear (Algorithm 2: Better Enumeration)
Linear (Algorithm 3: Divide and Conquer)
Linear (Algorithm 4: Linear Time)