Group 46
10/16/2016

Project 1

Colleen Minor and Thomas Prefontaine

I. Theoretical run-time analysis

Give pseudo-code for each of the four algorithms and an analysis of the asymptotic running-times of the algorithms.

Enumeration Pseudocode:

```
Algorithm enumeration(A[1…n])
for i in A[1…n]:
    for j in A[1…n]:
        for k in A[1…n]:
            sum = sum + a[k]
            if sum > max:
                 firstIndice = i
                 secondIndice = j
                  max = sum
return firstIndice, secondIndice, max
```

Enumeration Theoretical Analysis:

You have one outer loop, one loop inside of the first loop, and one loop inside of the middle loop, this is two inner loops, one of the loop is not quite n but n-I, however you still have n*n*n-I, this makes the asymptotic runtime O(n^3).

Better Enumeration Pseudocode:

```
betterEnumeration(A[1…n])
X = 0
    For I = 0 thru n I++
        Sum = 0
        For J = I thru n; j++
```

```
            If sum > X
                X = sum
    Return current
```

Better Enumeration Theoretical Analysis:

You have two loops, one inside of another which makes n*n or n^2. Technically it's not quite n*n since the inner loop doesn't loop through the entire array for every outer loop, it starts at I so you have n*n-I, but this is still O(n^2) Asymptotically.

Linear Pseudocode:

```
algorithm linear(A[1…n])
    for each n in A[1…n]:
        if sum < 0:
            sum = A[n]
            firstIndice = n
        else:
            sum += A[n]
        if maxSum < sum:
            maxSum= sum;
            secondIndice = n
    return firstIndice, secondIndice, max
```

Linear Theoretical Analysis:

This is one loop going through the entire array (n). Asymptotic running time of this linear time function is merely O(n).

Divide and Conquer pseudocode:

Source: Introduction to Algorithms, 3rd Edition, by CLRS, page 71:

```
maxCrossingSubaray(A, low, mid, high)
leftSum = -infinity
sum = 0
for I = mid downto low
    sum = sum + A[i]
    if sum > leftSum
    leftSum = sum
    maxLeft = i
rightSum = -infinity
sum = 0
for j = mid+1 upto high
    sum = sum + A[j]
    if sum > rightSum
    rightSum = sum
    maxRight = j
return maxLeft, maxRight, leftSum +
rightSum
```

Source: Introduction to Algorithms, 3rd Edition, by CLRS, page 72:

```
findMaximumSubarray(A, low, high)
if high == low
    return (low, high, A[low])
else mid = [(low + high)]/2
    leftLow, leftHigh, leftSum =
findMaximumSubarray(A, low, mid)
    rightLow, rightHigh, rightSum =
findMaximumSubarray(A, mid+1, high)
    crossLow, crossHigh, crossSum =
findMaximuSubarray(A, low, might, high)

if leftSum >= rightSum and leftSum >=
crossSum
```

```
    return leftLow, leftHigh, leftSum
elseif rightSum >= leftSum and rightSum
>= crossSum
    return rightLow, rightHigh, rightSum
else return
    crossLow, crossHigh, crossSum
```

Divide and Conquer Theoretical Analysis:

For this you have n/2 operations in the recursive call and this is height of lgn, so you have lgn, but in a recursive function to find if there is a cross subarray, rather than on one side or the other, you have two loops. The loops are separated, so it just adds n, but this is still multiplied by the lgn of the recursion, so the asymptotic analysis is O(nlgn)

II. Testing

At first we tried to test the algorithms by creating a program that would time, run, and average each of the tests. This worked well for three of the functions, but we found that for some reason it did not work for divide-and-conquer. Luckily however, for reasons we could not figure out, that same timing mechanism worked fine (on all four algorithms, including divide-and-conquer), in our main program (mss.py).

We decided to scrap the tester program and instead make a program to generate a random.txt file full of random-number arrays of whatever amount and sizes specified. Once we had done that we simply ran our main program on the random files,

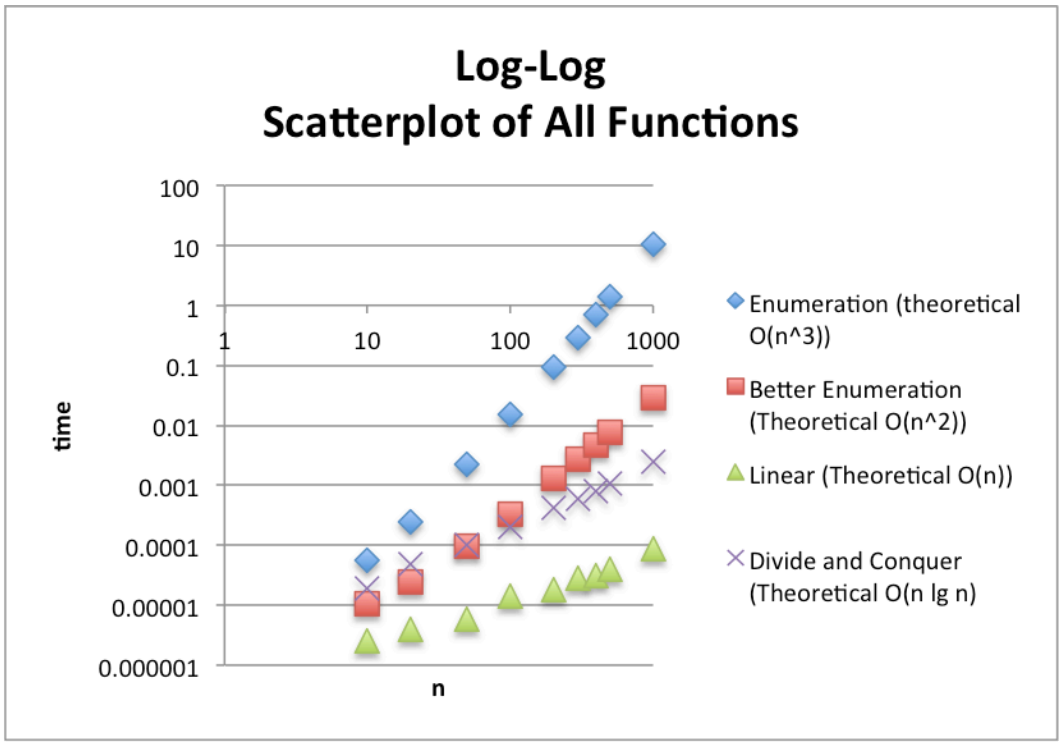
We used random.txt file: containing 10 n=10 arrays + 10 n=20 arrays +10 n=30 arrays + 10 n=,40 arrays, + 10 n=50 arrays + 10 n=100 arrays + 10 n=200 + 10 n=300 arrays + 10 n=400 arrays + 10 n=500 arrays. This was generated with arrayGenerator.py.

III. Experimental Analysis: Overview

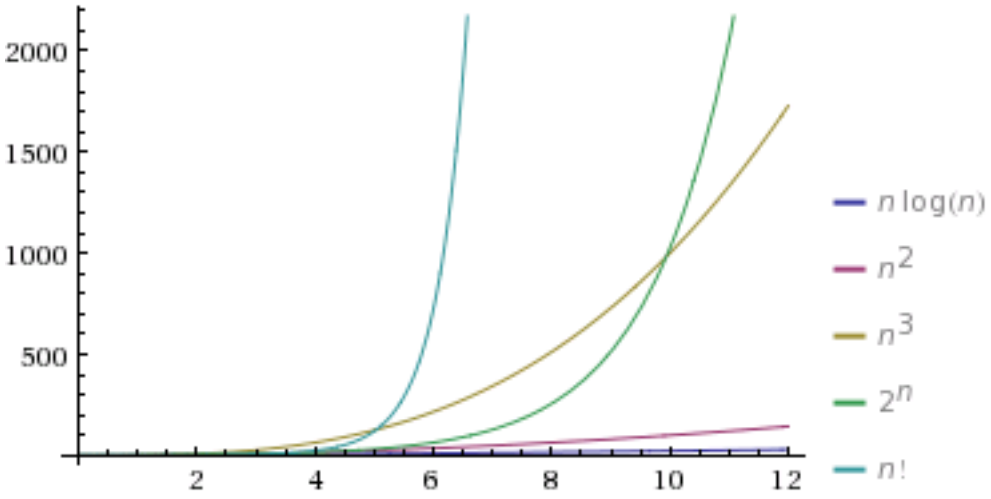
Here is our raw data:

n	Enum	BetterEnum	Linear	DivideAndConq
10	5.6982E-05	1.04904E-05	2.50334E-06	1.91E-05
20	0.000246048	2.39611E-05	4.05E-06	4.85182E-05
50	0.00226903	9.59635E-05	5.96E-06	0.000102997
100	0.015091658	0.000325084	1.39474E-05	0.000204563
200	0.094312429	0.001276969	1.78813E-05	0.000417948
300	0.292328835	0.002739906	2.79E-05	0.000592947
400	0.707027912	0.004562855	3.10E-05	0.000782967
500	1.369219542	0.007598996	4.00543E-05	0.001076937
1000	10.55915308	0.028506041	8.80E-05	2.46E-03

Before we begin let's look at this log-log graph of all four functions…



and compare it with this image from http://bigocheatsheet.com/:

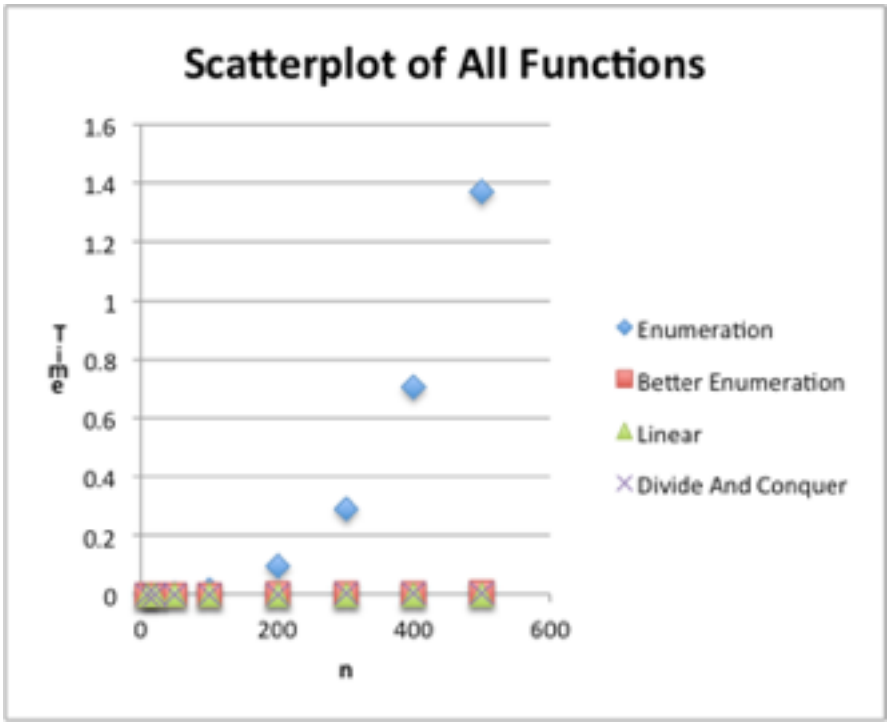


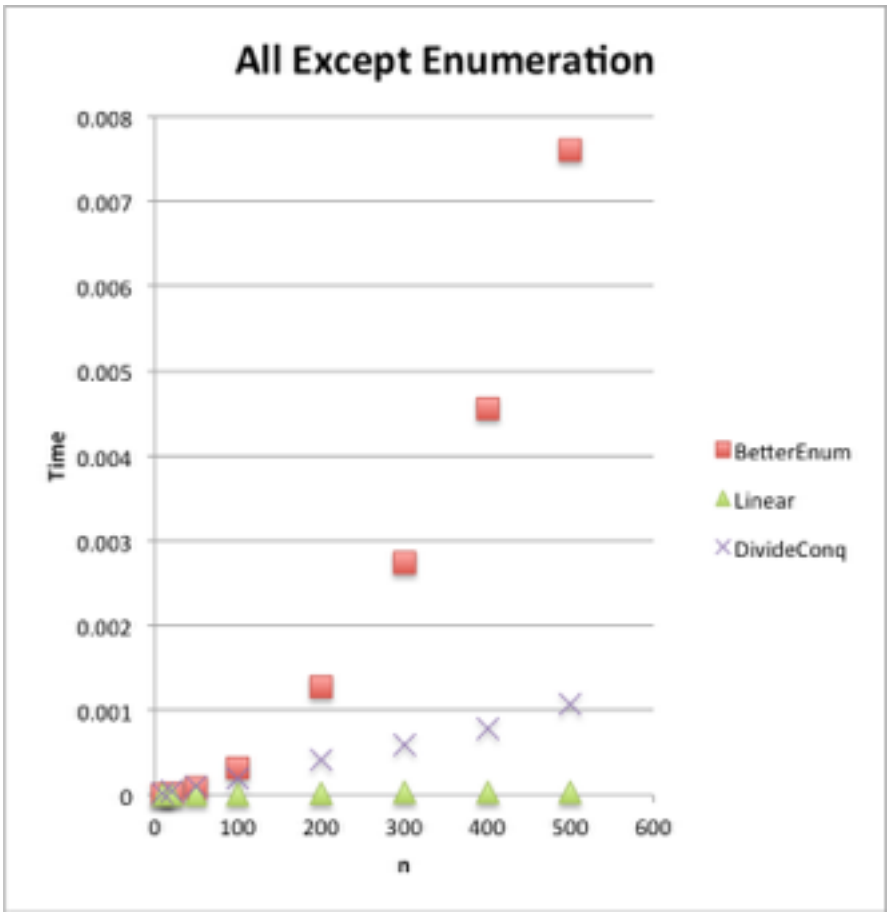
This graph features 3 of the big-Os that should theoretically correlate to our four functions: n lg n, n^3, and n^2.
Let's compare the lowest 3 lines in the big-O cheatsheet image(ignoring the green and light blue lines) to the top three lines in our log-log graph (ignoring the green triangles for Linear).

One thing we see n^3 graph (yellow above, blue triangles in our project graphs) should quickly do much, much worse than n^2 (magenta above, red squares in our project graphs) as n increases, and n lg n should do better than both in the longrun. This is true for our data. Whether or not the curves fit as they should, the relationships do look similar, and at least we can say that we do not have a situation where a graph that we theorized would grow slower than another ended up going faster (or vise versa).

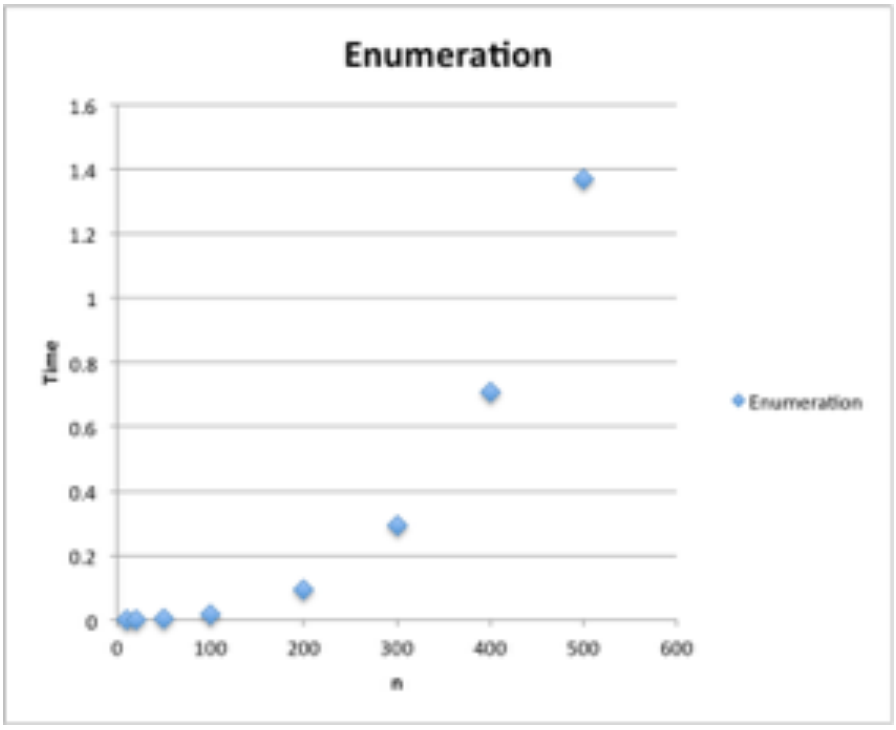
We couldn't really find any discrepancies between the running times and the experimental analysis. For the most part, the lines plotted by our data fit the curves of the theoretical run time analysis. The only curve that seemed to be a little off was the linear curve. Some of the data points were a little above the line, or below the line, but for the most part they still followed the linear 'curve' as it were.

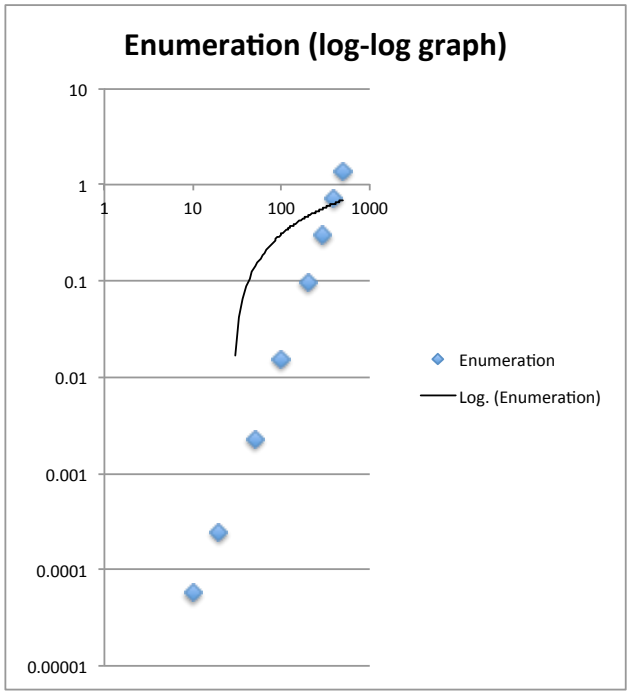
Here are some other (not log-log) graphs all of our data to give you an idea:


Scatterplot of All Functions
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
Time
0
200
400
600
n
Enumeration
Better Enumeration
Linear
Divide And Conquer


All Except Enumeration
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
Time
0
100
200
300
400
500
600
n
BetterEnum
Linear
DivideConq

V. Analysis of individual functions





Here is what we got with the data with MatLab:

Enumeration F(x) = a*x^b

Coefficients (with 95% confidence bounds):

a = 1.485e-08 (1.396e-08, 1.573e-08)

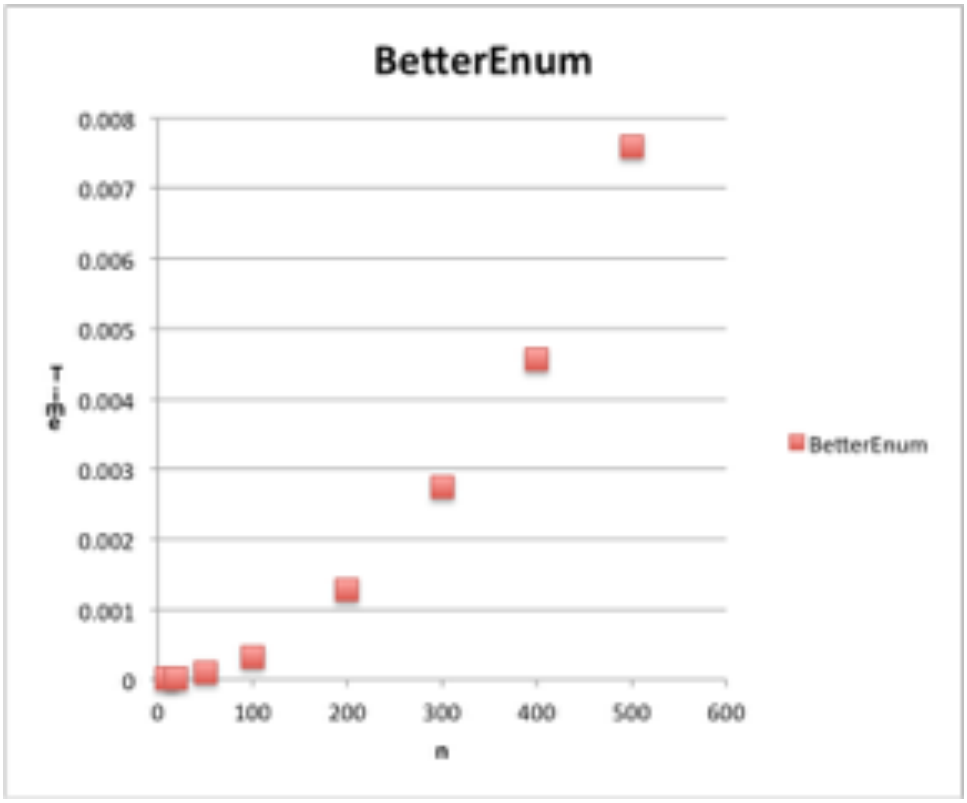
b = 2.951 (2.942, 2.959)

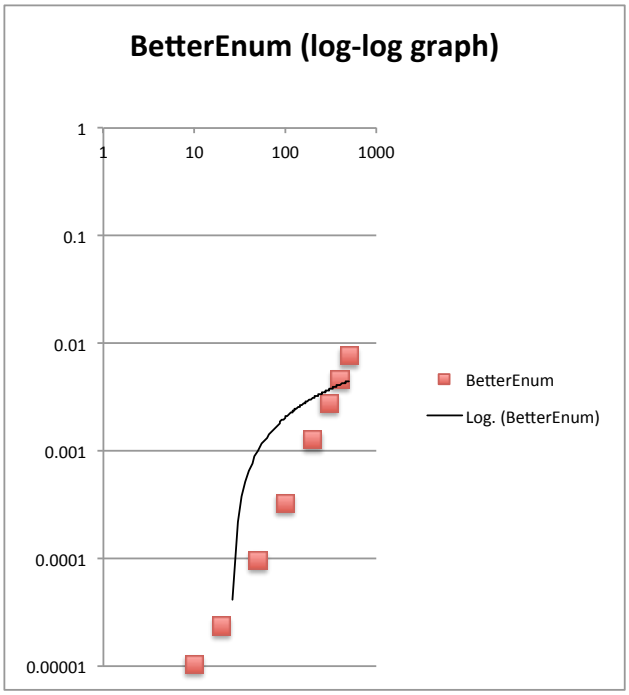
Here are the sizes n that we computed for the algorithm various times:

$n \leq 105\,884.$ for 10 seconds

$n \leq 153\,642.$ for 30 seconds

$n \leq 194\,321.$ for 1 minute





Here is what we got with the data with MatLab:

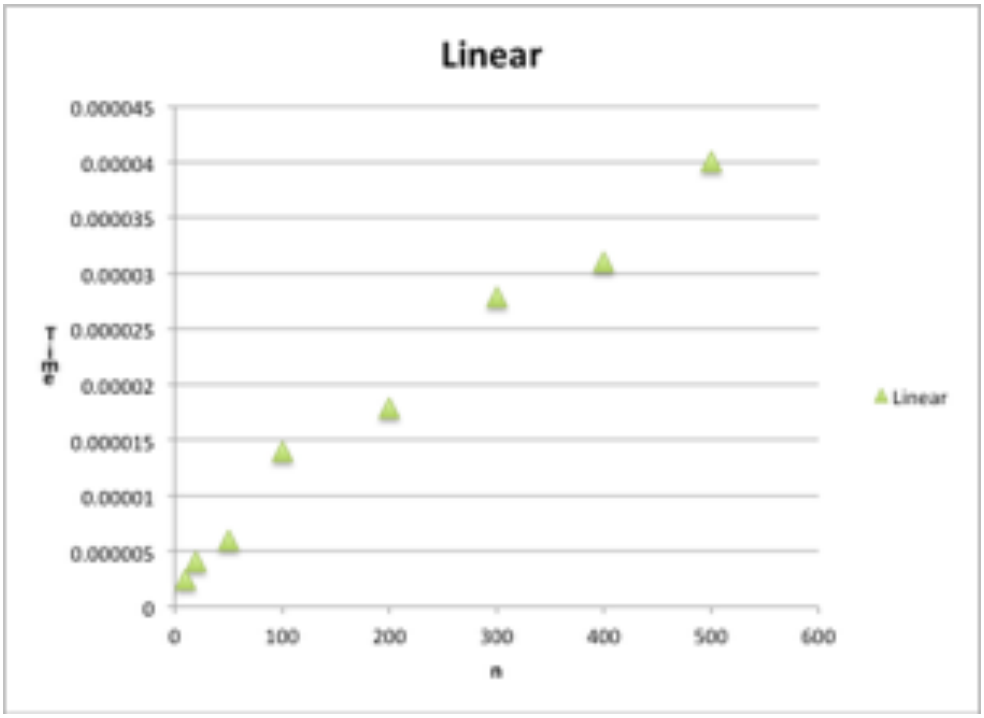
Better Enumeration F(x) = a*x^b

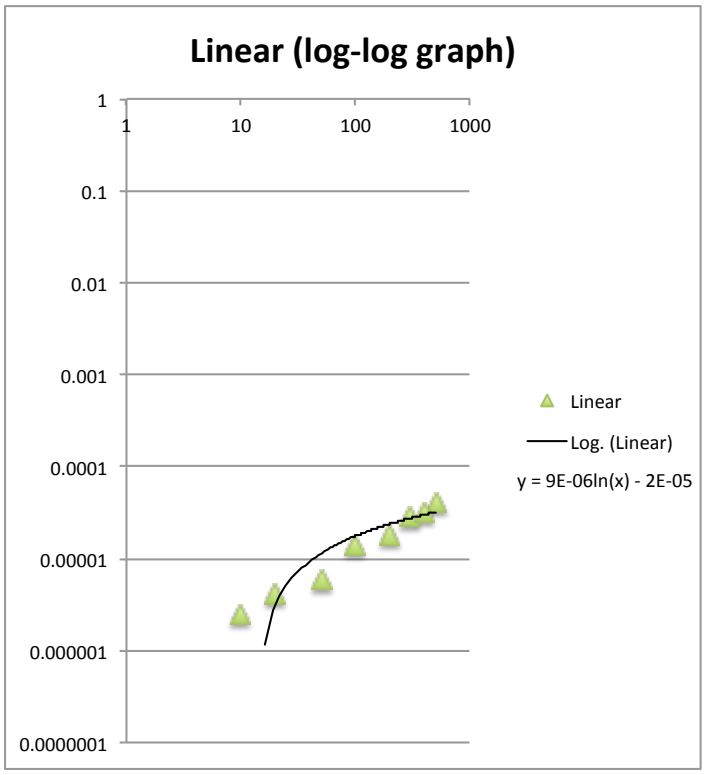
Coefficients (with 95% confidence bounds):
a = 4.209e-08 (3.153e-08, 5.265e-08)
b = 1.944 (1.907, 1.98)

Here are the sizes n that we computed for the algorithm various times:

$n \leq 2.48296 \times 10^7$ for 10 seconds

$n \leq 4.36921 \times 10^7$ for 30 seconds
$n \leq 6.24099 \times 10^7$ for 1 minute





Here is what we got with the data with MatLab:

Linear time F(x) = p1*x + p2
Coefficients (with 95% confidence bounds):

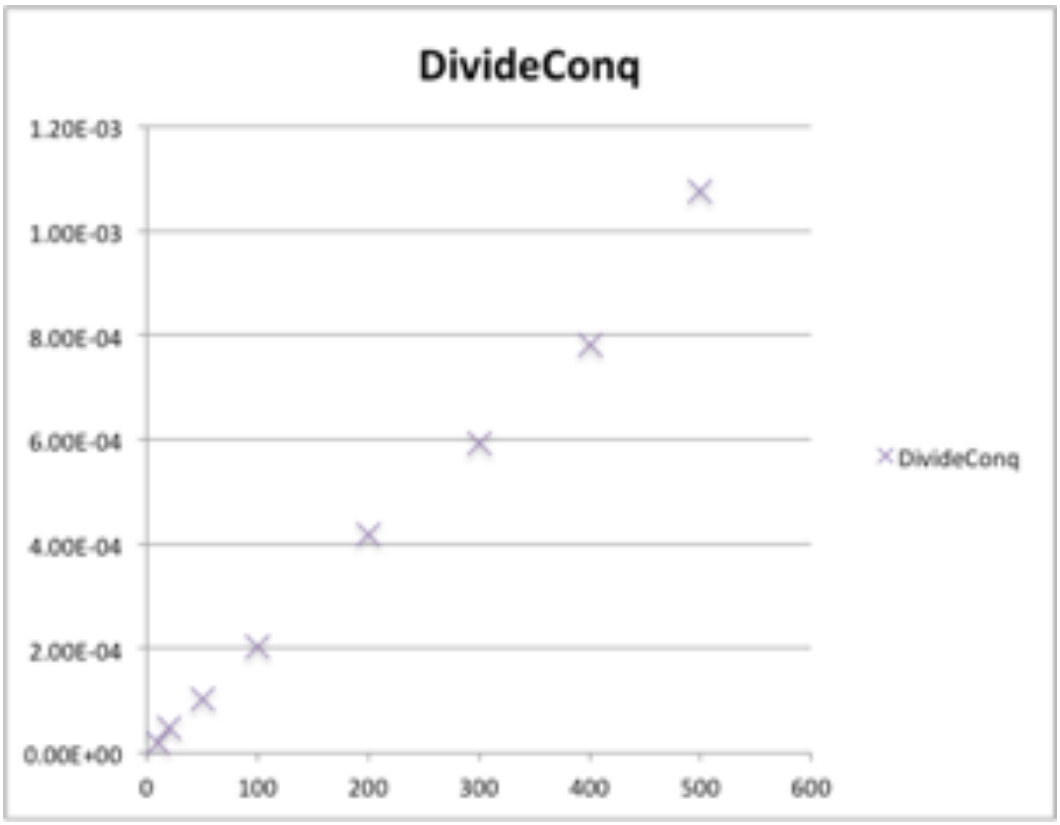
p1 = 8.341e-08 (7.618e-08, 9.065e-08)
p2 = 1.784e-06 (-1.221e-06, 4.79e-06)

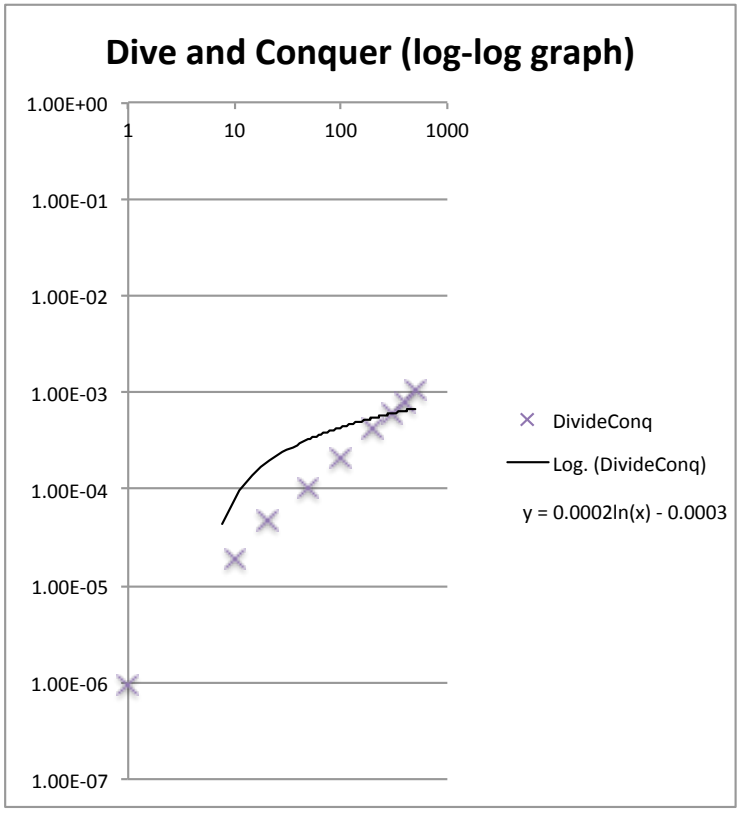
Here are the sizes n that we computed for the algorithm various times:

$n = 119\,889\,701\,474\,622$ for ten seconds.

$n = 3.59669 \times 10^{14}$ for 30 seconds

n = 7.19338×10^14 for 1 minute





Here is what we got with the data with MatLab:

Divide and Conquer:
f(x) = a*x*log(x)
Coefficients (with 95% confidence bounds):
a = 3.533e-07 (3.429e-07, 3.636e-07)

Here are the sizes n that we computed for the algorithm various times:

$n = 7.18637 \times 10^{11}$ for ten seconds

$n = 2.07529 \times 10^{12}$ for 30 seconds

$n = 4.05483 \times 10^{12}$ for 1 minute