CS 325 Project 1 Write-Up
Group # 25, Winter 2016
Brett Irvin, Aleksandr Balab, Dane Schoonover

Theoretical Runtime Analysis

Algorithm 1 Pseudocode, Enumeration.

```
count=array.length
for i=0 up to count
        for j=i up to count
        sum = 0
                for k=i up to and including j
                        sum += array[k]
                        if sum > max sum
                        max sum=current sum
return max sum
```

Asymptotic Analysis.
There are three loops total for this algorithm. The outer loop takes *n* amount of work for an array of size *n*. The first inner loop takes *n-i* work. The third loop takes *n-i-j*.

$T(n) = n(n-i)(n-j-i)$
$=(n^2-ni)(n-j-i)$
$=(n^3-n^{2j}-n^{2i}-n^{2i}+nij+ni^2$

Running Time: $\mathbf{T(n) = \Theta(n^3)}$

Algorithm 2 Pseudocode, Better Enumeration.

```
count=array.length
for i=0 up to count
        sum = 0
        for j=i up to count
        sum += array[j]
        if sum > best
                maxSum= sum
                subarray.left = i
                subarray.right = j
    subarray.maxSum = maxSum
return subarray
```

Asymptotic Analysis.
The outer loop takes *n* work for an array of size *n*. The inner loop takes *n-i* work.

$T(n) = n(n-i)= n^2-ni$

Running Time: **T(n) = Θ(n^2)**

Algorithm 3 Pseudocode, Divide & Conquer.

```
for i = 0 to ArraySize do
        maxSuffix= maxSuffix + Array[i + +]
        if maxSuffix <=  0 then
                maxSuffix= 0
        end if
        if maxSuffix>= maxSubArray then
                maxSubArray=maxSuffix
        end if
end for
```

Asymptotic Analysis.

This algorithm splits the problem into smaller pieces, which has logarithmic time complexity. Since this is done *n* times, T(n) = **O(nlogn)**.

Algorithm 4 Pseudocode, Linear Time.

```
max_sum = infinity
sum = -infinity
        endMax
        endMin
for i in range(count)
        endMax = i
        if sum > 0:
                sum += arr[i]
        else:
                endMin = i
                sum = arr[i]
        if sum > max_sum
                max_sum = sum
                leftIndex = endMin
                rightIndex = endMax
subarray.maxSum = max_sum
return subarray
```

Asymptotic Analysis.

This is Kadane's Algorithm. It scans through the array, computing at each position the maximum subarray ending at that position. The subarray is either empty (the sum is zero) or consists of one more element than the maximum subarray that ends at the previous position. Since there is only one loop, T(n) = **Θ(n).**

Testing

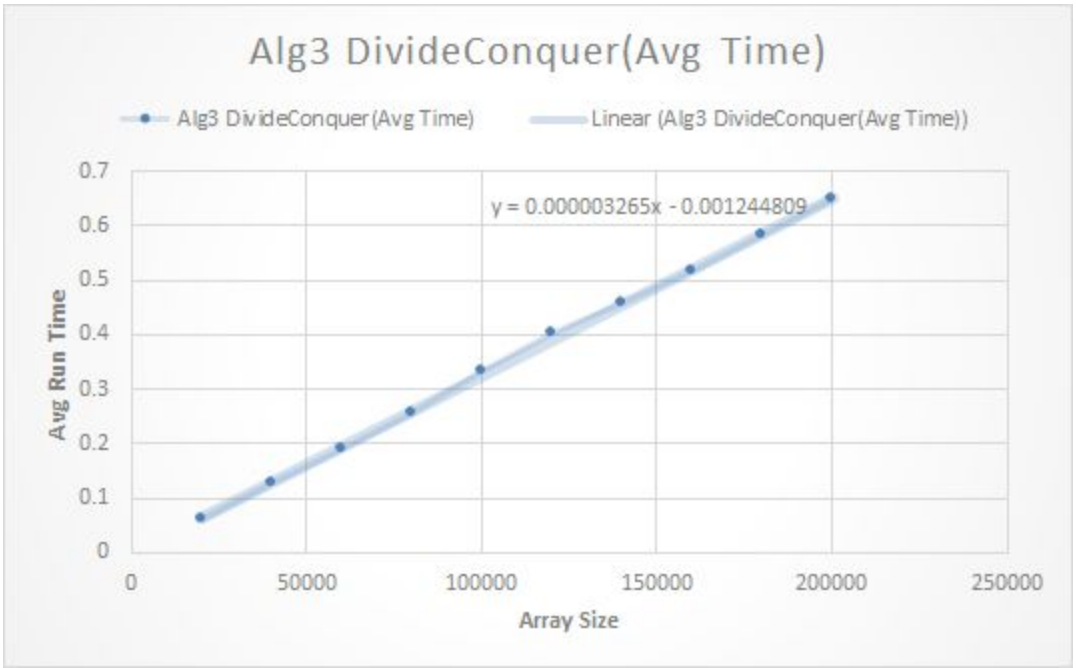
For testing, these algorithms were implemented using Python. To account for the various time complexities, input sizes were adjusted to obtain meaningful data for each algorithm. Each algorithm was evaluated based on its speed in calculating a random array, and the speeds were recorded using Python's system clock function.

Experimental Analysis.
Regression was used to find the closest matching function for each algorithm.

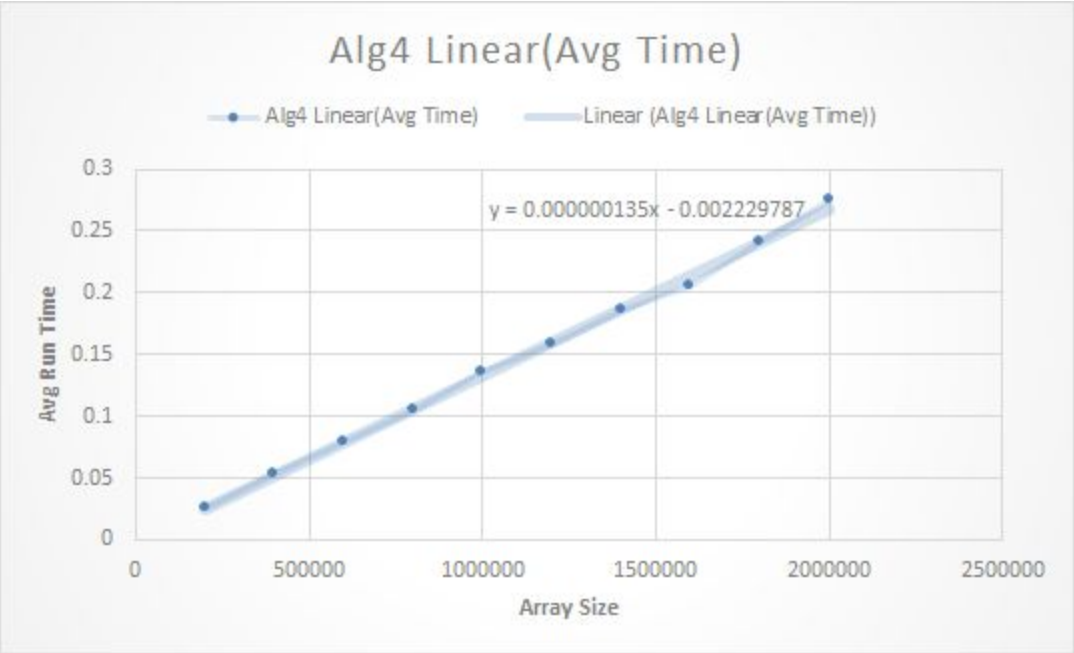
Algorithm 1: Expected runtime is $O(n^3)$. This is very close to the $n^{2.85}$ result we calculated from our data. The closest matching function curve was exponential.

Algorithm 2: The expected runtime is $O(n^2)$. This was close to the $n^{1.99}$ result we calculated from our data. The closest matching function curve was exponential.

Algorithm 3: The expected runtime was O(nlogn). The matching function curve was linear, but this is because the nlogn complexity is small enough that it's almost constant.



Algorithm 4: The expected runtime was $\Theta(n)$. The linear function curve matched as expected.



Discussion of Discrepancies.
There were slight discrepancies between the theoretical and running times, which was expected. In our case, the discrepancies were negligible, and could be attributed to a variety of factors, including the language of implementation and the computer we were testing on. Also, the expected runtimes were based on a worst-case scenario, which is not always what occurs in real-world analysis.

Calculating Largest Inputs (Regression).
Largest *n* possible in 1 minute:

- Algorithm 1: 2715
- Algorithm 2:42600
- Algorithm 3:190000
- Algorithm 4:4400000

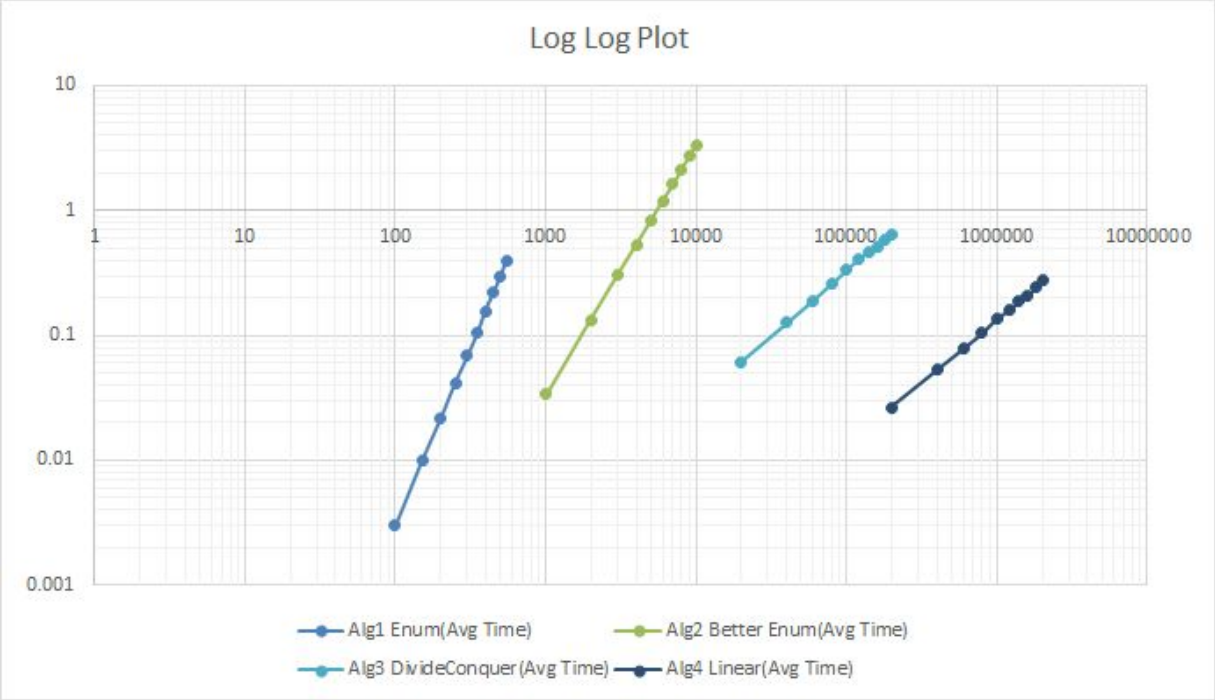
Largest *n* possible in 2 minutes:

- Algorithm 1: 3457
- Algorithm 2: 60100
- Algorithm 3: 620000
- Algorithm 4: 14800000

Largest *n* possible in 5 minutes:

- Algorithm 1: 4774
- Algorithm 2: 95200
- Algorithm 3: 1540000
- Algorithm 4: 37600000

Log Log Plot



The log log plot indicates that algorithm 1, 2, 3, and 4 are increasingly more efficient as evidenced by the decrease in slopes, respectively.

Slopes:

- Algorithm1: 2.86
- Algorithm2: 1.99
- Algorithm3: 1.03
- Algorithm4: 1.01