Solomon Huynh
Jonathan Lagrew
Yunsik Choi

CS325 Group 3
Project 1

<u>Theoretical Run-time Analysis</u>

Algorithm 1: Enumeration O(n^3)

T(n) = n * n * n

Note: Pseudocode was provided in Project 1 Video

```
MaxSubarray(a[1,...,n])
    for each pair (i,j) with 1 ≤ i < j ≤ n
        compute a[i]+a[i+1]+· · · +a[j-1]+a[j]
        keep max sum found so far
    return max sum found
```

Algorithm 2: Better Enumeration O(n^2)

T(n) = n * n

Note: Pseudocode was provided in Project 1 Video

```
MaxSubarray(a[1,...,n])
    for i = 1, ..., n
        sum = 0
        for j = i, ..., n
            sum = sum + a[j]
            keep max sum found so far
    return max sum found
```

Algorithm 3: Divide and Conquer O(n lg n)

T(n) = { c if n = 1
{ 2T(n/2) + cn if n > 1

T(n) = 2T(n/2) + cn
Using the master theorem,
a = 2, b = 2, f(n) = cn
$n^{loga(b)} = n^{log2(2)} = n$
Therefore, case #2 will be applied.

```
MaxSubarray(a[1,...,n], int n)
	//base case if n = 1
	if n = 1
		return a[0]
	else
		mid = n / 2
		leftMSS = MaxSubarray(a, mid)
		rightMSS = MaxSubarray(a+mid, n-mid)
	sum = 0
	for i = mid to n
		sum += a[i]
		rightSum = max(rightMSS.sum, sum)
	sum = 0
	for i = 0 to mid
		sum += a[i]
		leftSum = max(leftMSS.sum, sum)
	MSS = max(leftMSS.sum, rightMSS.sum)
	return max(MSS, rightSum + leftSum)
```

Algorithm 4: Linear-time O(n)
T(n) = n

Note: Pseudocode was provided in maxsumsubLinear.pdf given by professor

```
MAX-SUBARRAY-LINEAR(A)
  n = A.length
  max-sum = −∞
  ending-here-sum = −∞
  for j = 1 to n
      ending-here-high = j
      if ending-here-sum > 0
          ending-here-sum = ending-here-sum + A[j]
      else ending-here-low = j
          ending-here-sum = A[j]
      if ending-here-sum > max-sum
          max-sum = ending-here-sum
          low = ending-here-low
          high = ending-here-high
  return (low, high, max-sum)
```

<u>Proof of Correctness</u>

Claim 1:

Given an array *A* containing *n* integers a_0, a_1,...a_{n-1} for n > 0, the divide and conquer algorithm 3 will correctly generate the sum of the maximum subarray, s=max($\sum_{k=i}^{j} a_k$) for integers *i, j* < *n*

The max subarray will be contained entirely in the first half denoted as leftMSS.sum
The max subarray will be contained entirely in the second half denoted as rightMSS.sum
The max subarray will be made of a suffix of the first half of the subarray and a prefix of the second half denoted as crossMSS.sum

Proof:

By induction using top-down method:

Base Case:
As a base case, let n = 1. Then sum will be the value of n denoted as variable pass, a[0].
This is found when high equals low. This returns in Θ(1) time.

Inductive hypothesis:
leftMSS = algorithm3(A[0:$\frac{n}{2}$ -1])
rightMSS = algorithm3(A[$\frac{n}{2}$:n])
crossMSS = findMaxCrossingSubarray(A[0:n])

We can consider three possible cases:

Case 1: Max Subarray contained entirely in first half

Case2: Max Subarray contained entirely in second half

Case3: Max Subarray made of a suffix of the first half of the subarray and a prefix of the second half

Claim 2: The algorithm terminates

Proof: Since n > 0 then n must be at least a value of 1 and the algorithm returns. This proves the base case.

For the inductive hypothesis, assume that the algorithm returns for an array of length n ≤ q for some positive integer q > 1. Consider n = q + 1. The array will be split into two branches of positive lengths, which means each branch will have lengths less than or equal to q. In conclusion, the algorithm will return for each branch and the algorithm returns follows.

Claim 3: Algorithm 3 computes the sum of the maximum subarray in O(n log n) time.

Proof: Let n be the size of the array of integers, a. For n > 1, the recurrence for the recursive step of the algorithm can be found to be:

$$\begin{aligned} T(n) &= \Theta(1) + 2T(\tfrac{n}{2}) + \Theta(n) + \Theta(1) \\ &= 2T(\tfrac{n}{2}) + \Theta(n) \end{aligned}$$

The base case will be Θ(1). We know recursive calls take $2T(\frac{n}{2})$, the max() calculations take Θ(n), and the last return will take Θ(1).

$$T(n) = \begin{cases} \theta(1) & if = 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & if\ n > 1 \end{cases}$$

Suppose T(n) ≤ cn log n + n = O(n log n). Then

$$\begin{aligned} T(n) &\le 2(c \cdot \tfrac{n}{2} \log \tfrac{n}{2}) + n \\ &\le cn \log \tfrac{n}{2} + n \\ &= cn \log n - cn \log 2 + n \\ &\le cn \log n \\ &= O(n \log n) \end{aligned}$$

Testing Procedure

The project included a menu option to test both the test problems provided by the professor and the run times based on n inputs where n is the number of inputs the user enters. The programming team compared the results of the test problems with the answers and it produced the correct results in all four algorithms. They also used their own inputs to test for the results. Note that the test results outputs the set of inputs, the set of maximum sub subarrays, and the total sum to test for correct responses.

Experimental Analysis

Average Running Times

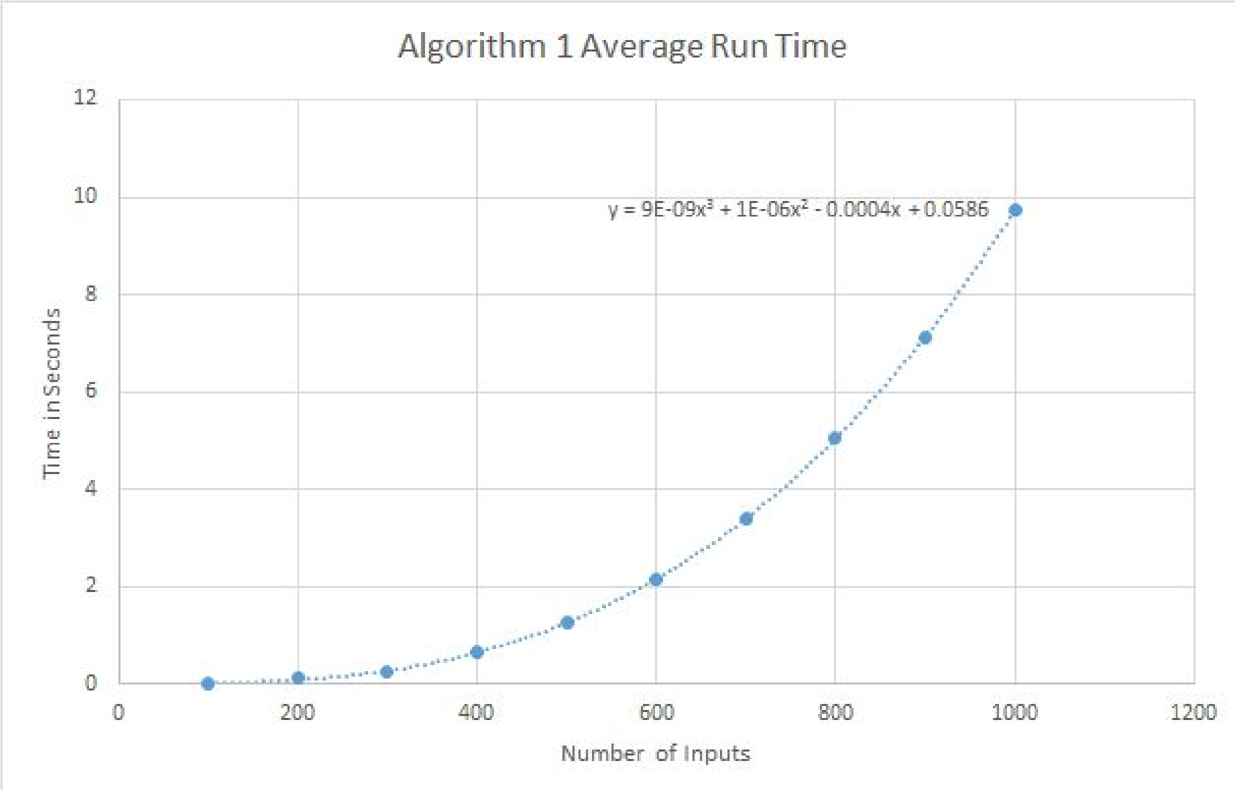
Algorithm 1	
# of inputs	Average Run Time
100	0.027
200	0.113
300	0.268
400	0.65
500	1.247
600	2.165
700	3.397
800	5.062
900	7.118
1000	9.734

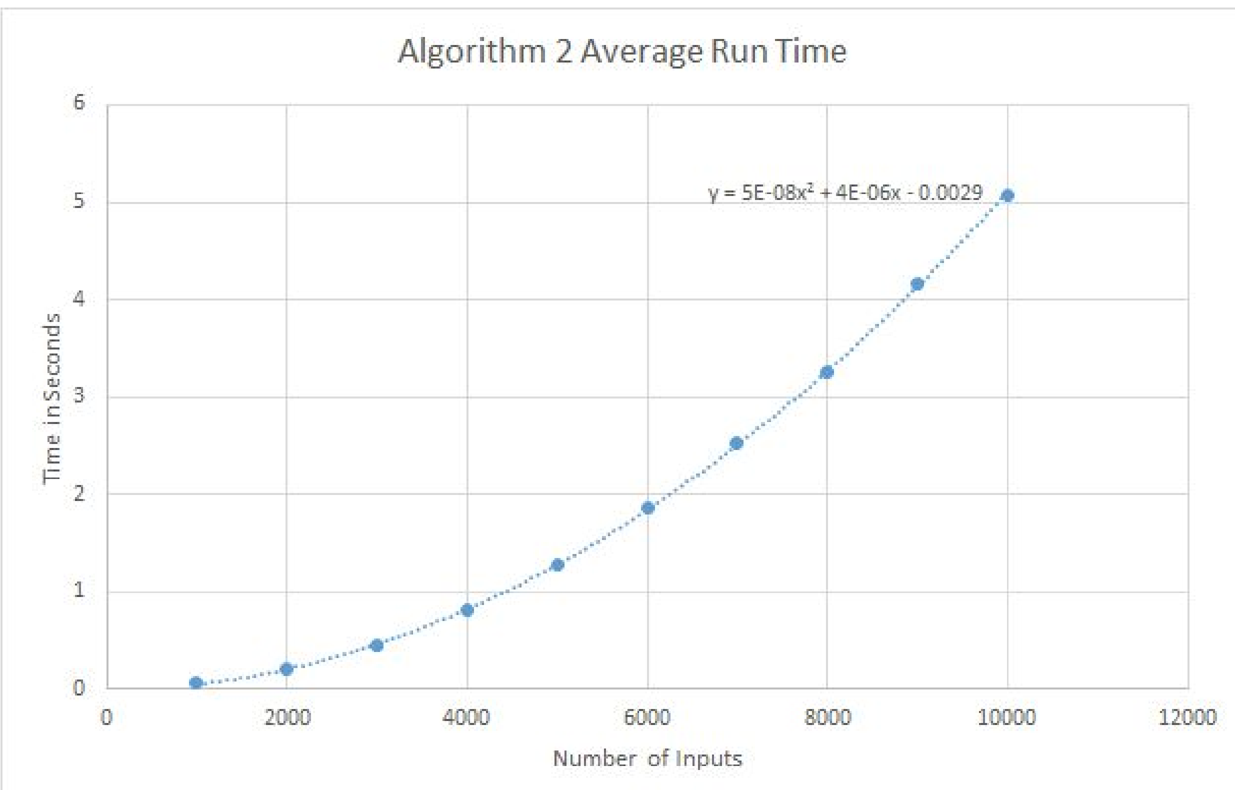
Algorithm 2	
# of inputs	Average Run Time
1000	0.057
2000	0.215
3000	0.457
4000	0.814
5000	1.268
6000	1.853
7000	2.52
8000	3.262
9000	4.165
10000	5.078

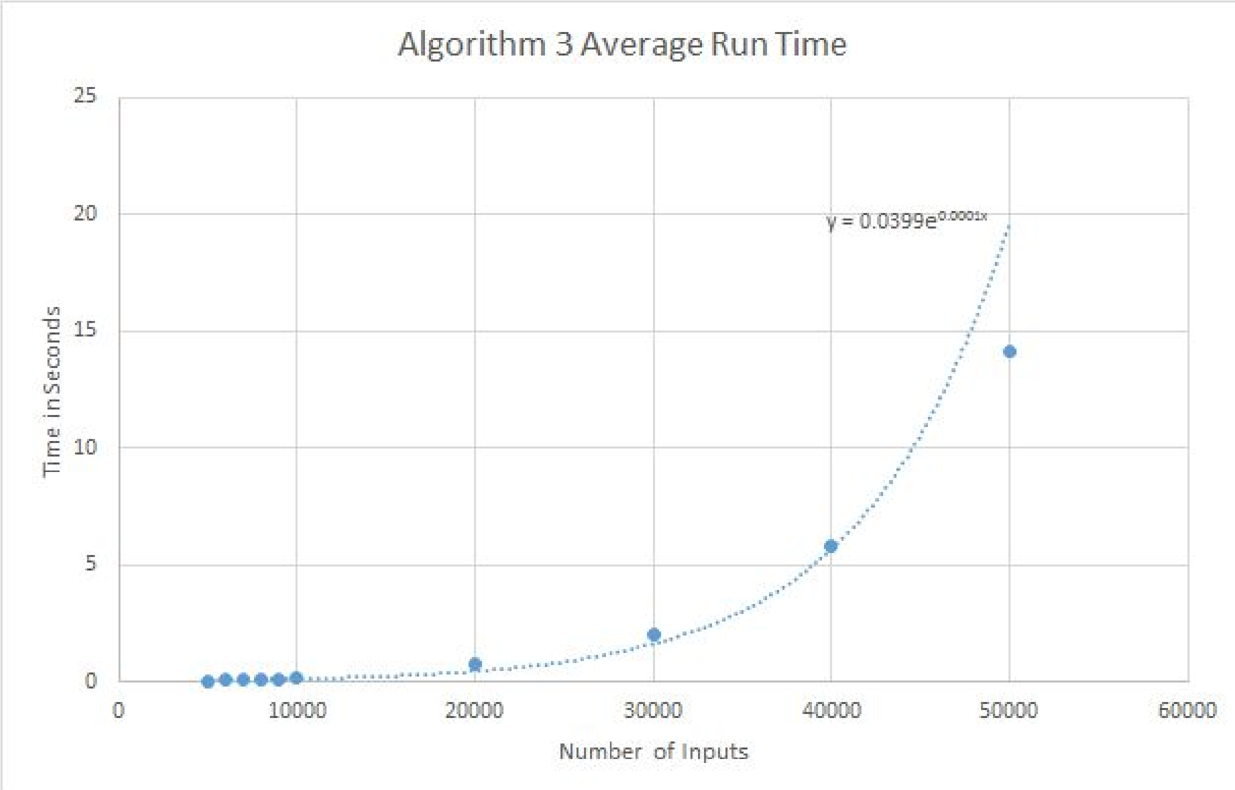
Algorithm 3	
# of inputs	Average Run Time
5000	0.047
6000	0.067
7000	0.087
8000	0.113
9000	0.139
10000	0.166
20000	0.772
30000	2.07
40000	5.836
50000	14.116

Algorithm 4	
# of inputs	Average Run Time
100000	0.012
200000	0.023
300000	0.036
400000	0.046
500000	0.059
600000	0.083
700000	0.093
800000	0.12
900000	0.123
1000000	0.142

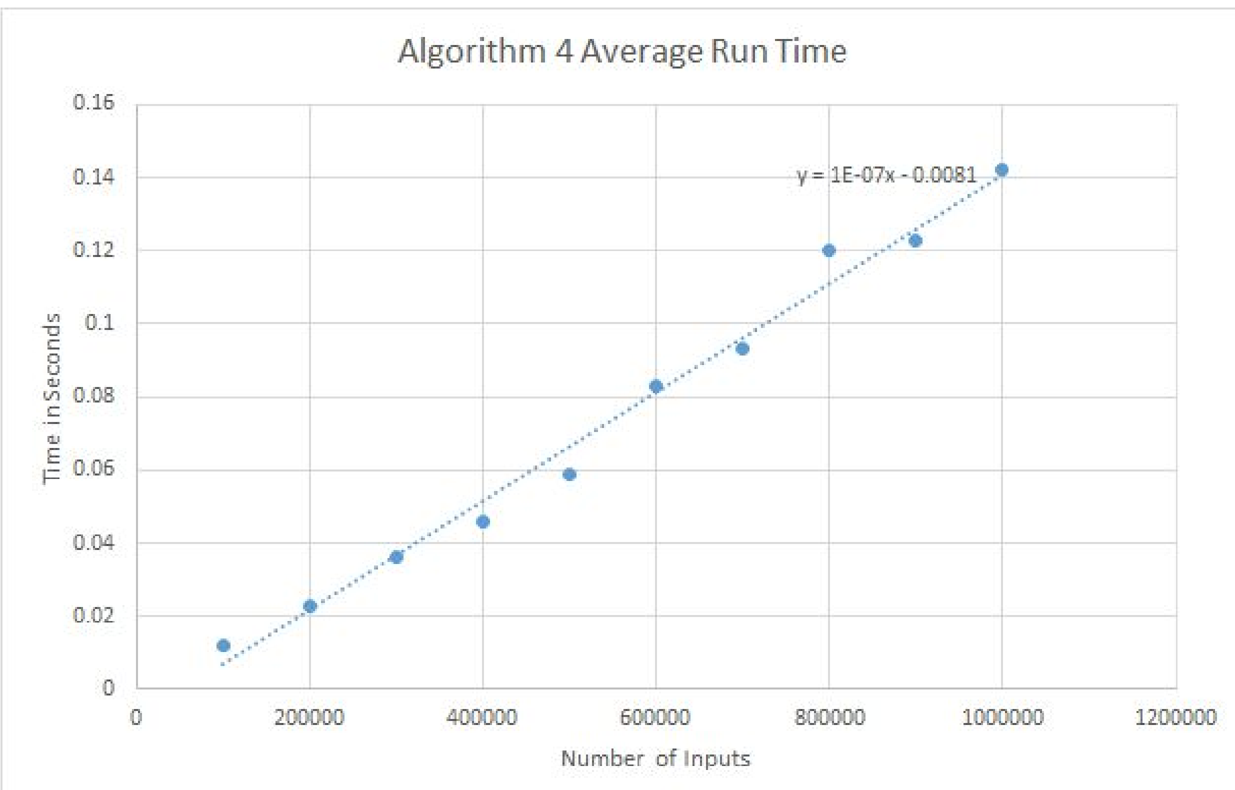
Running Time Graphs


Algorithm 1 Average Run Time
y = 9E-09x³ + 1E-06x² - 0.0004x + 0.0586
Time in Seconds
Number of Inputs
0
2
4
6
8
10
12
200
400
600
800
1000
1200


Algorithm 2 Average Run Time
y = 5E-08x² + 4E-06x - 0.0029
Time in Seconds
Number of Inputs
0
1
2
3
4
5
6
2000
4000
6000
8000
10000
12000



Algorithm 3 Average Run Time
y = 0.0399e^0.0001x
Time in Seconds
Number of Inputs
0
5
10
15
20
25
0
10000
20000
30000
40000
50000
60000



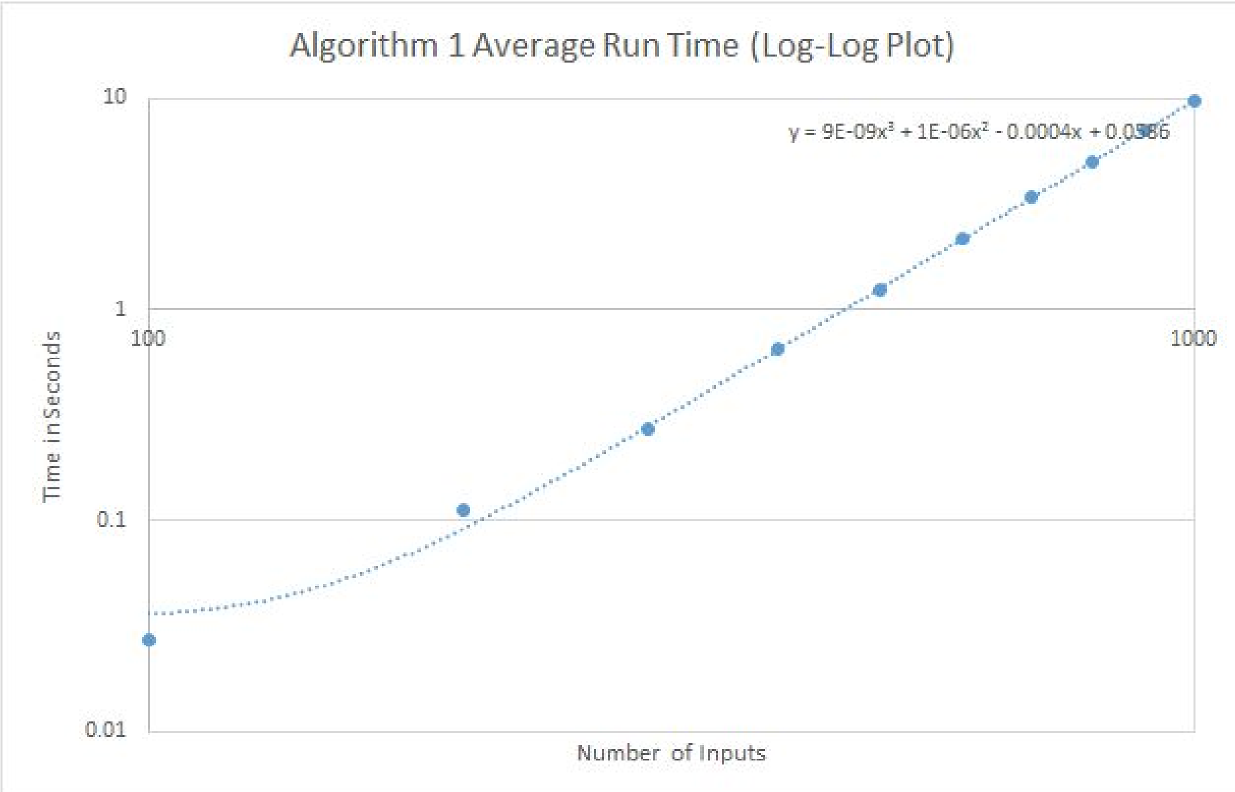
Algorithm 4 Average Run Time
y = 1E-07x - 0.0081
Time in Seconds
Number of Inputs
0
0.02
0.04
0.06
0.08
0.1
0.12
0.14
0.16
0
200000
400000
600000
800000
1000000
1200000

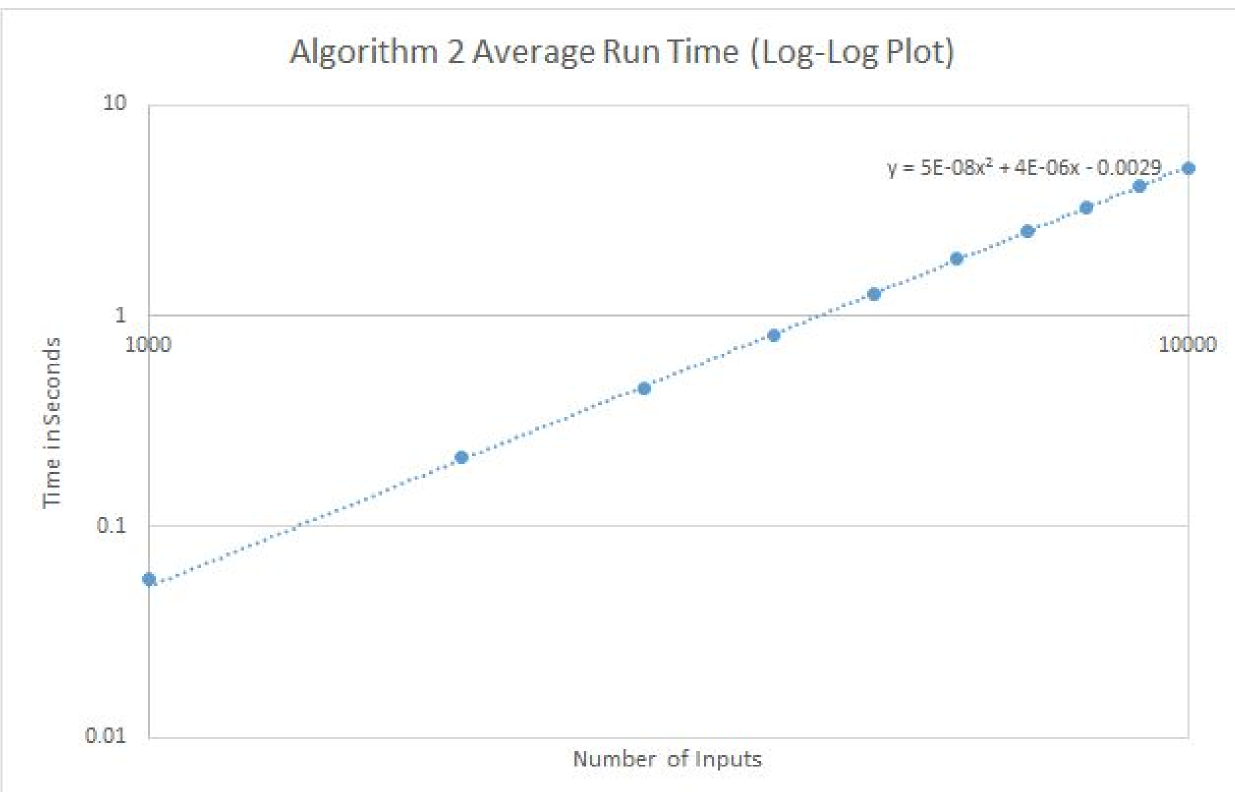
Graph Functions
Algorithm 1: $y = 9E\text{-}09x^3 + 1E\text{-}06x^2 - 0.0004x + 0.0586$
Algorithm 2: $y = 5E\text{-}08x^2 + 4E\text{-}06x - 0.0029$
Algorithm 3: $y = 0.0399e^{(0.0001x)}$
Algorithm 4: $y = 1E\text{-}07x - 0.0081$

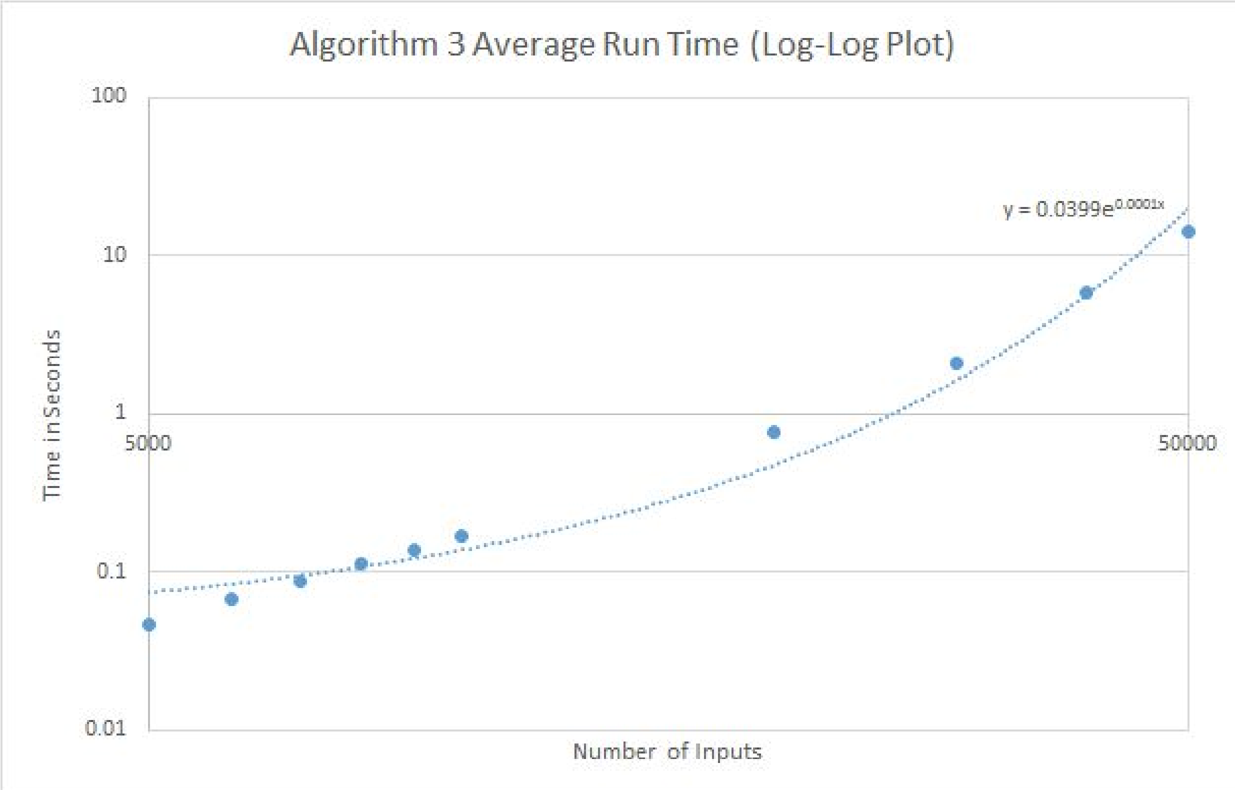
Number of Inputs to Take 10 Minutes of Runtime
Algorithm 1: n = 4,022
Algorithm 2: n = 109,505
Algorithm 3: n = 15,038
Algorithm 4: n = 6,000,081,000

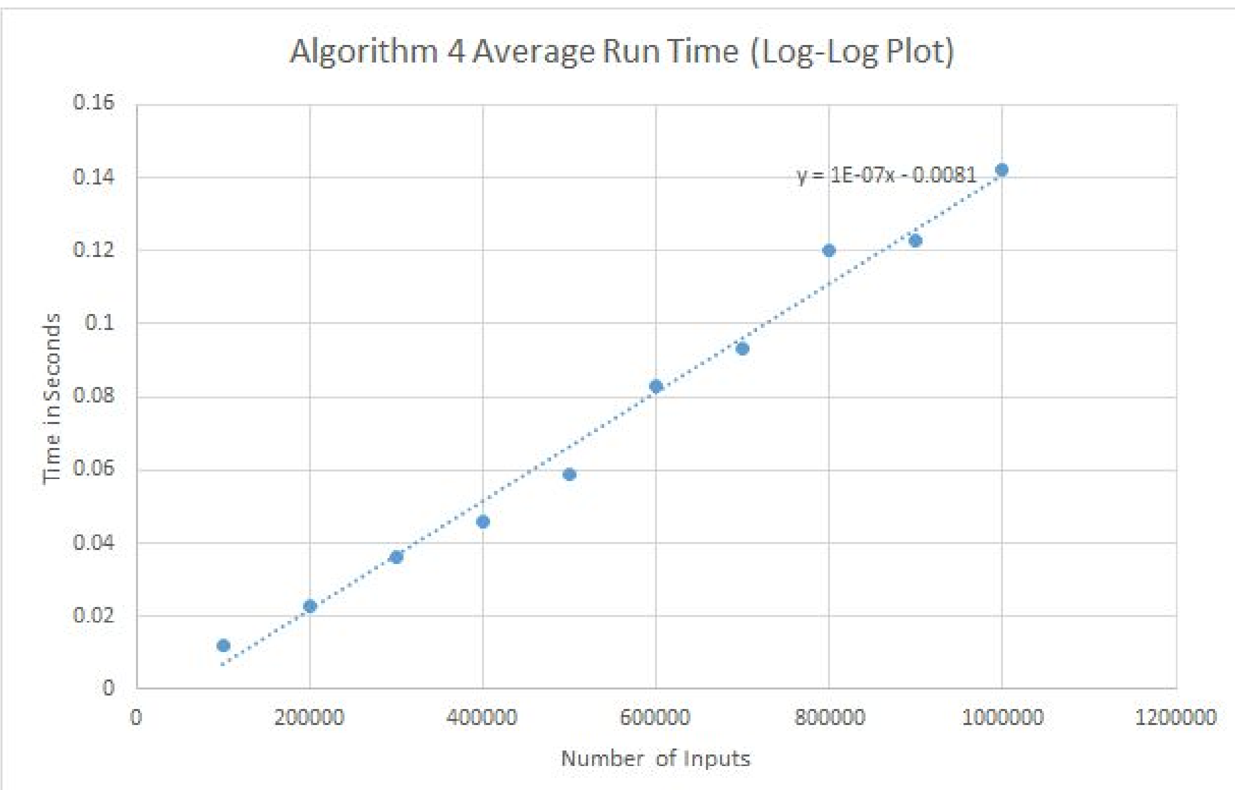
Any Discrepancies between experimental and theoretical running times?
There isn't much difference between experimental and theoretical running times as long as input is far great reaching upto millions.
When the number of inputs reach millions, experimental running times grows exponentially compared to theoretical running times.

Log-Log Plots

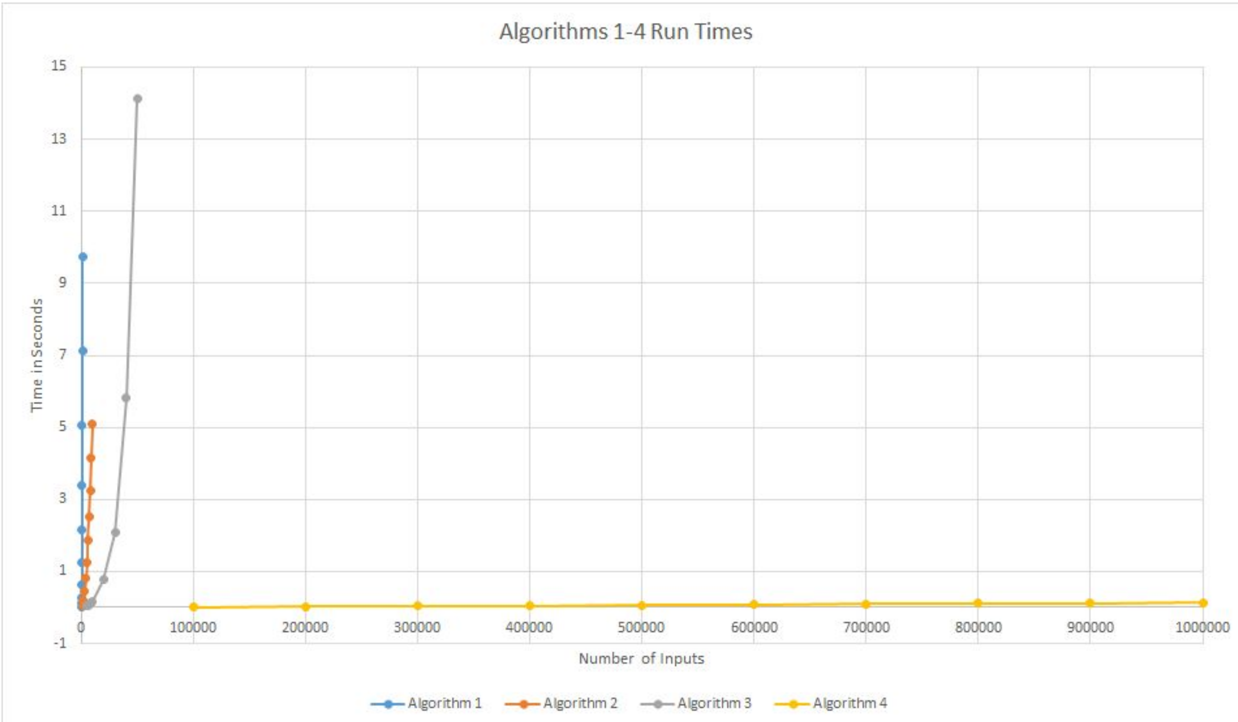





Algorithm 3 Average Run Time (Log-Log Plot)
y = 0.0399e^0.0001x
100
10
1
0.1
0.01
5000
50000
Time in Seconds
Number of Inputs


Algorithm 4 Average Run Time (Log-Log Plot)
y = 1E-07x - 0.0081
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
200000
400000
600000
800000
1000000
1200000
Time in Seconds
Number of Inputs

Graph Containing All Algorithms



Resources:

1) http://www.geeksforgeeks.org/largest-sum-contiguous-subarray/
2) http://codeforces.com/blog/entry/13713
3) http://www.wolframalpha.com/
4) http://www.geeksforgeeks.org/divide-and-conquer-maximum-sum-subarray/