Travis Robinson
Scott Williams
Ben Henning
CS325
Group 6
Project 1

Theoretical Run-Time Analysis:

Algorithm 1 Pseudo-code:

```
//compare every possible array to every other possible array
maxSubArray(array, arraySize)
        for(index < arraySize)
                for(j<arraySize)
                        for(i < j)
                                sum = sum+array[j]
                if (sum>currentMaxSum)
                        currentMaxSum = sum
                        lowIndex = index
                        highIndex = j
```

Algorithm 1 Asymptotic analysis:
Using Substitution:
Guess: $T(n) = O(n^3)$
Induction Goal: $T(n) <= dg(n)$
Induction Hypothesis: $T(k)<= dT(k)$ for all $k < n$
Proof: $T(n) = T(n-1) + n <= d(n-1)^3+n$
$=d(n^3-3n^2+3n-1)+x<=dn^3$
if: $d3n^2+d3n-d>=0$
$==d3n^2+d3n>=d$
$==3n^2+3n>=1$
For $3n^2+3n>=1$, any d will work

Algorithm 2 Pseudo-code:

```
//compare all possible subarrays, knowing that adding array[j+1] to array[1..j] is constant time
//calculate sum while progressing through the possible sub arrays
maxSubArray(array,arraySize)
        for (index < arraySize)
                for(i <arraySize)
                        sum = sum+array[i]
                        if (sum > currentMaxSum)
                                currentMaxSum = sum
                                lowIndex = index
                                highHindex = i
```

Algorithm 2 Asymptotic analysis:
Using Substitution:
Guess: $T(n) = O(n^2)$
Induction Goal: $T(n) <= dg(n)$

Induction Hypothesis: $T(k) <= dT(k)$ for all $k < n$
Proof: $T(n) = T(n-1) + n <= d(n-1)^2+n$
$=d(n^2-2n+1)+n<=dn^2$
$=dn^2-d2n+d$
if: $dn^2-d2n+d>=0$
$==dn^2-d2n>=-d$
$==n^2-2n>=-1$
True for $n >= 2$, any d will work

Algorithm 3 Pseudo-code:

```
maxSubArraySum(arr, l, h)
//Base case
if l = h
        return arr[l]
//Divide array into two and make recursive calls
let m = (l+h)/2
let leftMaxSubArraySum = maxSubArraySum(arr, l, m)
let rightMaxSubArraySum = maxSubArraySum(arr, m+1, h)

//Find maximum subarray which crosses the midpoint
//Max sum left side of midpoint
let sum = 0
let leftSum = lowest possible int value
for i = m l
        let sum = sum + arr[i]
        if sum > leftSum
                let leftSum = sum
//Max sum right side of midpoint
let sum = 0
let rightSum = lowest possible int value
for i = (m+1) h
        let sum = sum + arr[i]
        if sum > rightSum
                let rightSum = sum
//Combine the two to get the max sum that crosses the midpoint
let crossingSum = leftSum + rightSum

return maximum of leftMaxSubArraySum, rightMaxSubArraySum, and crossingSum
```

Preconditions:
    Arr has at least 1 element, l <= h

Postcondition:
    The maximum subarray sum between indices l and h is returned

Algorithm 3 Asymptotic analysis:
$T(n) = 2T(n/2) + n$

Using master method:
$n^{log_b a} = n^{log_2 2} = n^1$
$f(n) = n$
$n = \Theta(n^1 * lg^k n)\ for\ k = 0$
Case 2:
$T(n) = \Theta(n^1 * lg^{0+1} n) = \Theta(n * lgn)$

Algorithm 4 Pseudo-code:

```
highIndex = array.length;
lowIndex = 0;
sum = array[0];
ending_here_sum = array[0];
maxSum = -Infinity

for(index =1; index < array.length)
{
	highIndex = index;
	if (ending_here_sum > 0)
	{
		ending_here_sum = ending_here_sum+array[index];
	}
	else
	{
		lowIndex = index;
		ending_here_sum = array[index]
	}
	if (ending_here_sum > maxSum)
	{
		maxSum = ending_here_sum
		lowIndexToReturn = lowIndex
		highIndexToReturn = highIndex
	}
}
return maxSum, lowIndexToReturn, highIndexToReturn
```

Algorithm 4 Asymptotic analysis:
Using Substitution:
Guess: T(n) = O(n)
Induction Goal: T(n) <= dn
Induction Hypothesis: T(n-1) <= d(n-1)
Proof: T(n) = T(n-1) <= d(n-1) = dn - d <= dn
True when d > 0, n > 1

Proof of Correctness for Algorithm 3:

Number of elements in array: n = h - l + 1

Base Case: n = 1

Array contains a single element. The only possible subarray is the single element, so the value of the element is the max sum subarray.

Inductive Hypothesis:

Assume that maxSubArraySum correctly finds the max sum subarray for an array containing n = 1, 2, ..., k elements.

Inductive Step:

Show that maxSubArraySum correctly finds the max sum subarray for an array containing n = k + 1 elements.

- First recursive call n1 = ((l + h) / 2) - l <= k -> max subarray sum of subarray arr[l .. ((l+h)/2)] is returned
- Second recursive call n2 = h - ((l+h)/2) <= k -> max subarray sum of subarray arr[((l+h)/2) .. h] is returned
- maxSubArraySum finds the max subarray sum of the subarray in arr[l ... h] which crosses element arr[(l+h)/2]
- maxSubArray sum returns the highest value of the above 3 max subarray sums
- This guarantees that maxSubArraySum returns the value of the max subarray sum, because we know through our hypothesis that we can find the max subarray sum for all arrays less than or equal to size k, which all of theses three subarrays are, since they're all subarrays of array k+1

Termination:

Every recursive reduces n to n/2. Eventually, this will cause n to equal 1, at which point the function returns without making additional recursive calls.

Proof of Correctness for Portion of Code for Finding Crossing Sum:

Loop Invariant:

At the start of each iteration of the first loop, the maximum subarray sum in subarray arr[i+1 .. m] has been computed and stored as variable leftSum. If the sum of all elements from arr[i .. m] is greater than what is currently stored in leftSum, leftSum will be replaced by that value. Therefore, at the start of the following iteration, the maximum subarray sum in subarray arr[i+1 .. m] will be stored in variable leftSum.

Initialization:

Prior to the first iteration of the loop, we have i = m, so the subarray arr[i + 1 ... m] is empty. The empty subarray contains the maximum (in this case, only) sub array sum within array arr[i+1
.. m]. As this value is greater than the current value of leftSum, leftSum is replaced with the sum of the current subarray.

Maintenance:

To see that each iteration maintains the loop invariant, suppose that the sum of all elements in subarray arr[i ... m] is greater than the value of leftSum. In this case, leftSum will be given the value of the sum of all elements in the subarray arr[i .. m]. Therefore, when the value of i is incremented in the loop update, leftSum will still hold the value of the maximum subarray sum within subarray arr[i+1 ... m]. If instead the sum of all elements in subarray arr[i .. m] is less than the value currently stored in leftSum, leftSum will not be changed. Therefore, at the start of the next iteration, leftSum will still continue to hold the value of the maximum subarray sum within subarray arr[i+1 ... m].

Termination:
At termination, i = l - 1. By the loop invariant, the subarray arr[i + 1 .. m], which is arr[l .. m], has had its maximum subarray sum stored in the variable leftSum.

The loop for computing the max subarray sum within arr[m+1 ... h] is essentially the same. After the end of that loop, the max subarray sum within the subarray arr[l .. m] is added to the max subarray sum within the subarray arr[m+1 .. h], giving us the maximum subarray sum within the subarray within arr[l .. h] which crosses the midpoint, m.

Testing:

We initially tested the algorithms with randomly generated arrays containing values between -250 and 250. These initial arrays were relatively small (10 to 20 elements) so that the results could be verified by hand. These tests were conducted to ensure that the algorithms produced the correct subarrays as well as the correct sum.

The algorithms were then tested using the provided arrays contained in MSS_TestProblems.txt and compared with the results in MSS_TestResults.txt; these were all validated as producing the correct sum.

After testing the algorithms with small random arrays and the provided test arrays, we performed tests with larger randomly generated arrays to confirm that with larger values of n, all algorithms would still produce the same results. These were not verified by us, as it would be tedious going through an array of 50+ elements, and since if all algorithms produced the same result, it was unlikely that they would all be producing the same incorrect result. These larger test results were perhaps not necessary due to induction (if it produces correct results for small arrays, it should for larger arrays as well), but were still done for thoroughness.

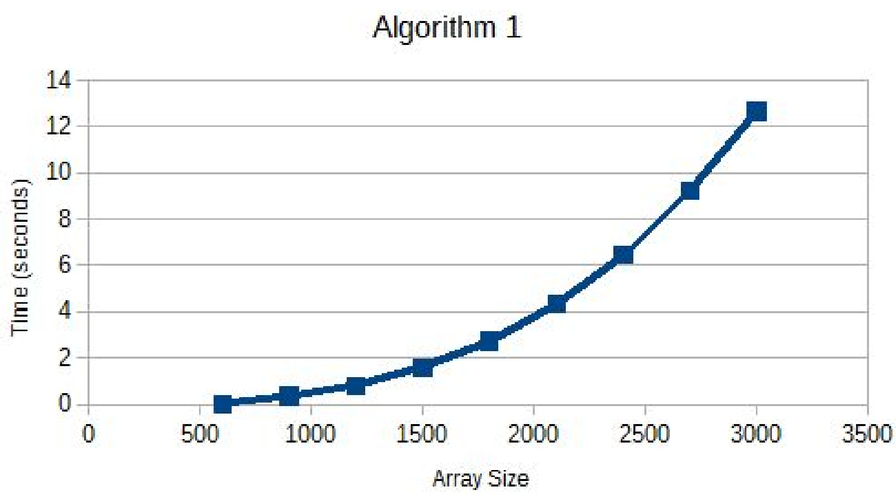
Experimental Analysis:

Algorithm 1:

Regression Equation:

$$5.10849233071458E\text{-}10x^3 - 2.60762385762399E\text{-}07x^2 + .0004980459x - .288357149$$

N	A1 Mean					Algorithm 1 Running Times					
300	0.014	0.01	0.02	0.01	0.01	0.02	0.01	0.01	0.02	0.01	0.02
600	0.014	0.01	0.02	0.01	0.01	0.02	0.01	0.02	0.01	0.01	0.02
900	0.347	0.34	0.35	0.34	0.35	0.35	0.35	0.34	0.35	0.35	0.35
1200	0.814	0.81	0.81	0.81	0.82	0.81	0.82	0.81	0.82	0.81	0.82
1500	1.586	1.59	1.59	1.59	1.58	1.59	1.59	1.58	1.59	1.58	1.58
1800	2.735	2.74	2.73	2.73	2.74	2.74	2.73	2.73	2.74	2.74	2.73
2100	4.338	4.34	4.34	4.34	4.34	4.34	4.34	4.34	4.34	4.33	4.33
2400	6.471	6.47	6.47	6.47	6.47	6.47	6.47	6.47	6.47	6.48	6.47
2700	9.222	9.22	9.22	9.22	9.22	9.22	9.22	9.22	9.22	9.22	9.24
3000	12.644	12.65	12.65	12.65	12.65	12.64	12.64	12.64	12.64	12.64	12.64

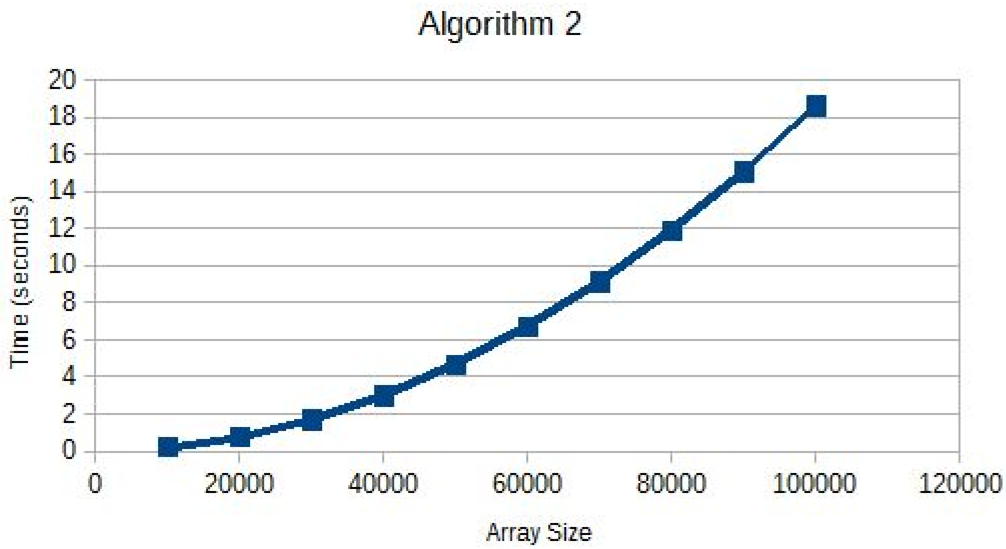


Algorithm 2:

Regression Equation:

$$1.86284090909091E\text{-}09x^2 - 2.55530303030337E\text{-}07x + .0045166667$$

A2 N	A2 Mean					Algorithm 2 Running Times					
10000	0.189	0.19	0.19	0.19	0.19	0.19	0.19	0.19	0.19	0.19	0.18
20000	0.741	0.74	0.74	0.74	0.74	0.74	0.74	0.74	0.74	0.74	0.75
30000	1.675	1.68	1.68	1.67	1.68	1.67	1.67	1.68	1.67	1.67	1.68
40000	2.976	2.97	2.98	2.98	2.98	2.98	2.98	2.97	2.97	2.97	2.98
50000	4.65	4.66	4.65	4.65	4.65	4.65	4.65	4.65	4.65	4.64	4.65
60000	6.698	6.73	6.69	6.7	6.69	6.69	6.7	6.69	6.7	6.69	6.7
70000	9.115	9.11	9.12	9.12	9.11	9.11	9.12	9.12	9.11	9.11	9.12
80000	11.899	11.9	11.9	11.89	11.9	11.9	11.9	11.9	11.9	11.9	11.9
90000	15.071	15.09	15.07	15.07	15.07	15.06	15.07	15.07	15.07	15.07	15.07
100000	18.61	18.62	18.62	18.62	18.61	18.61	18.6	18.6	18.6	18.61	18.61

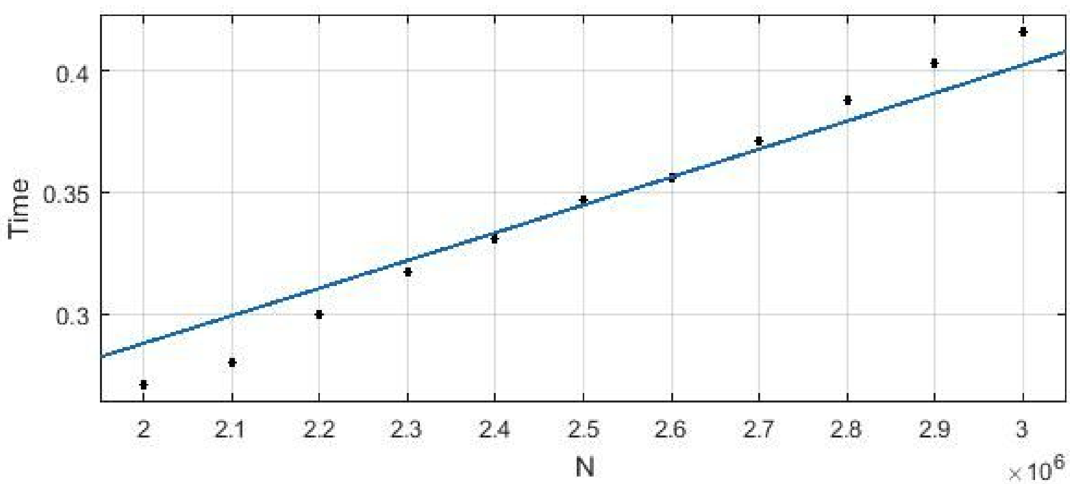


Algorithm 3:

Regression Equation:

7.21E-09*x*log(x)+.07708

A3 N	A3 Mean	Algorithm 3 Running Times									
2000000	0.271	0.28	0.27	0.27	0.27	0.27	0.27	0.27	0.27	0.27	0.27
2100000	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28
2200000	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3
2300000	0.317	0.33	0.32	0.32	0.32	0.32	0.31	0.31	0.31	0.31	0.32
2400000	0.331	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.33	0.34
2500000	0.347	0.34	0.35	0.35	0.35	0.35	0.35	0.35	0.35	0.34	0.34
2600000	0.356	0.36	0.36	0.36	0.35	0.35	0.35	0.36	0.36	0.35	0.36
2700000	0.371	0.37	0.37	0.37	0.37	0.37	0.38	0.37	0.37	0.37	0.37
2800000	0.388	0.38	0.39	0.39	0.39	0.39	0.39	0.38	0.39	0.39	0.39
2900000	0.403	0.4	0.4	0.41	0.4	0.4	0.4	0.4	0.41	0.4	0.41
3000000	0.416	0.42	0.42	0.42	0.42	0.42	0.41	0.41	0.42	0.41	0.41

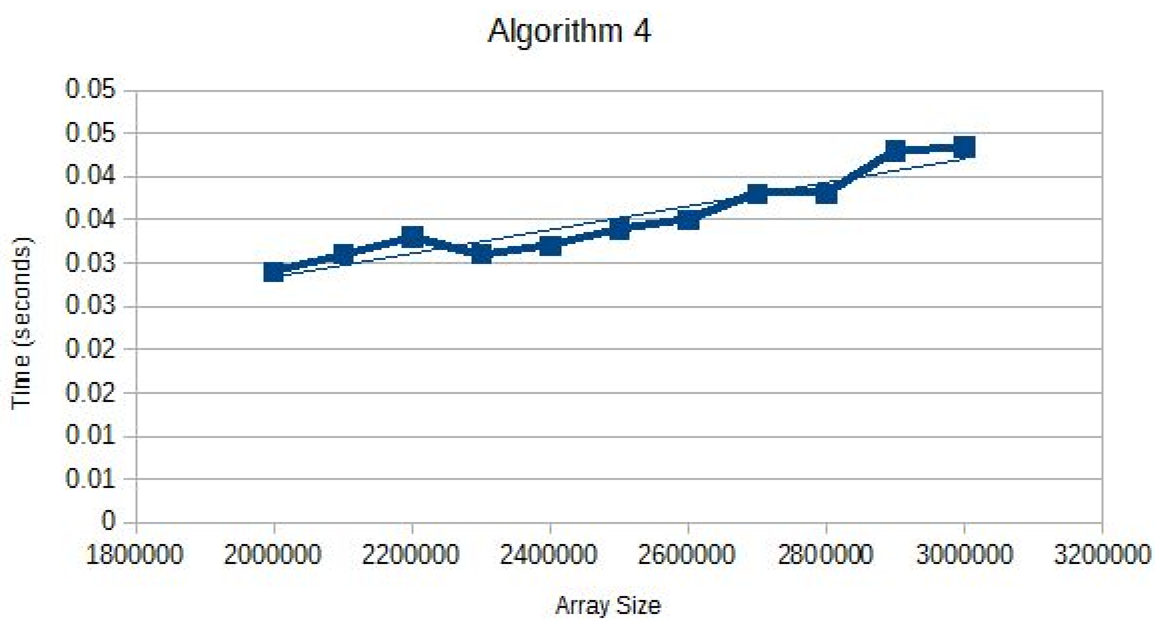


Algorithm 4:

Regression Equation:

1.38181818181818E-08x + .0006727273

A4 N	A4 Mean					Algorithm 4 Running Times					
2000000	0.029	0.04	0.02	0.03	0.03	0.03	0.03	0.04	0.02	0.03	0.02
2100000	0.031	0.04	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
2200000	0.033	0.05	0.04	0.03	0.02	0.03	0.03	0.05	0.03	0.03	0.02
2300000	0.031	0.05	0.03	0.03	0.03	0.02	0.03	0.04	0.02	0.04	0.02
2400000	0.032	0.05	0.04	0.04	0.03	0.03	0.03	0.02	0.02	0.03	0.03
2500000	0.034	0.05	0.03	0.04	0.04	0.03	0.03	0.04	0.02	0.03	0.03
2600000	0.035	0.05	0.03	0.03	0.03	0.04	0.03	0.04	0.03	0.04	0.03
2700000	0.038	0.05	0.04	0.04	0.04	0.04	0.03	0.04	0.03	0.04	0.03
2800000	0.038	0.05	0.03	0.04	0.04	0.03	0.04	0.04	0.04	0.04	0.03
2900000	0.043	0.06	0.04	0.03	0.04	0.04	0.04	0.05	0.04	0.04	0.05
3000000	0.0434	0.07	0.04	0.04	0.04	0.04	0.004	0.06	0.04	0.05	0.05



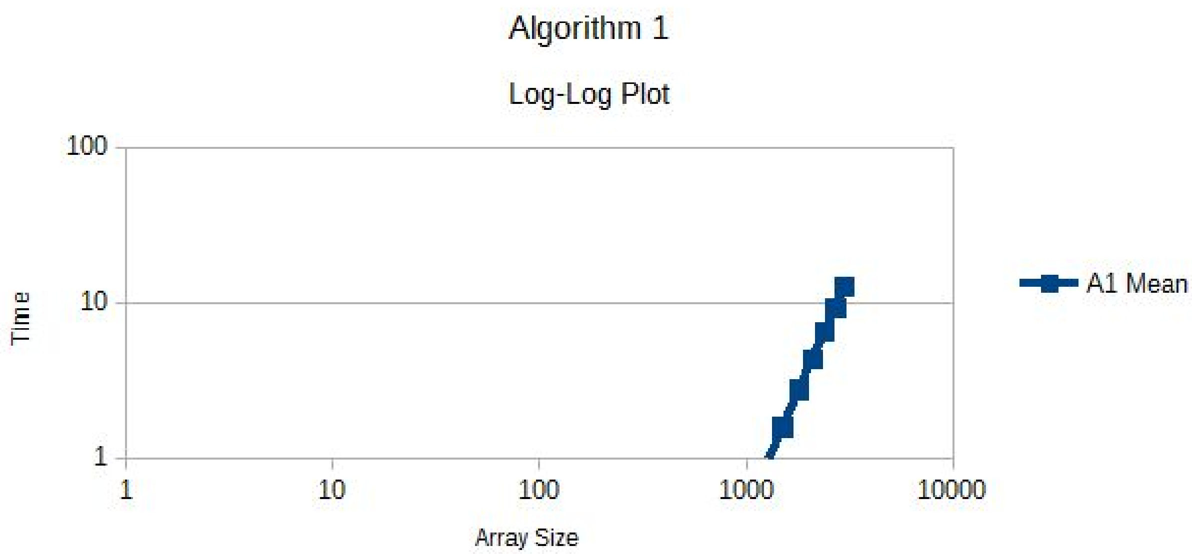
Discrepancies

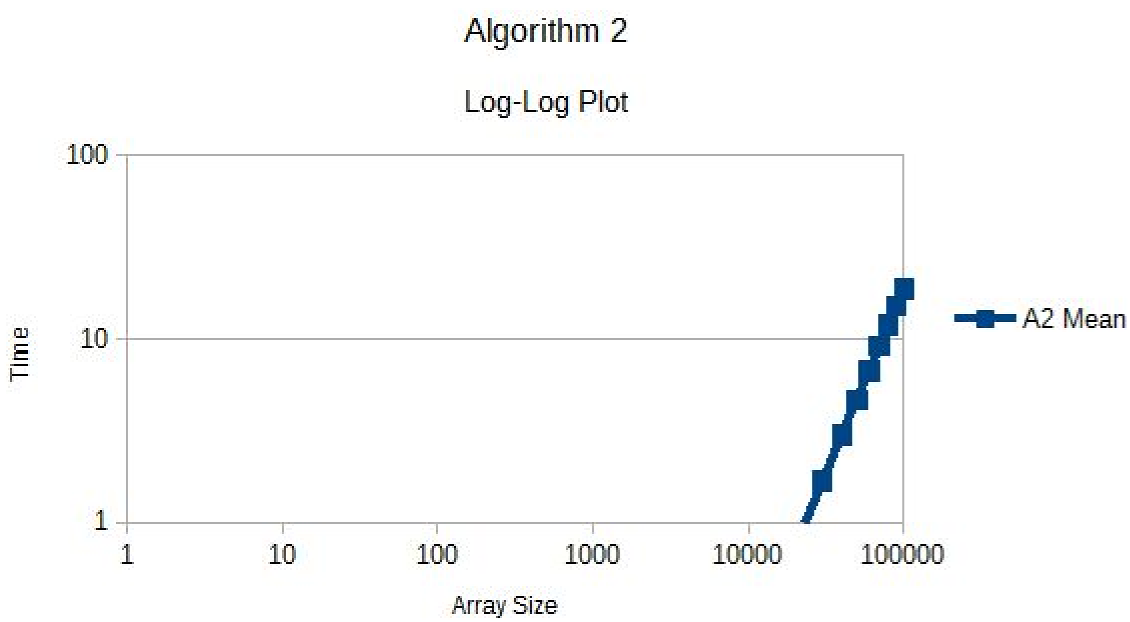
There were no discrepancies noticed, except perhaps that the linear time function (algorithm 4) would sometimes produce an average that was lower than the average for lower values of n. This is likely due to other strains placed on the processor at the time.

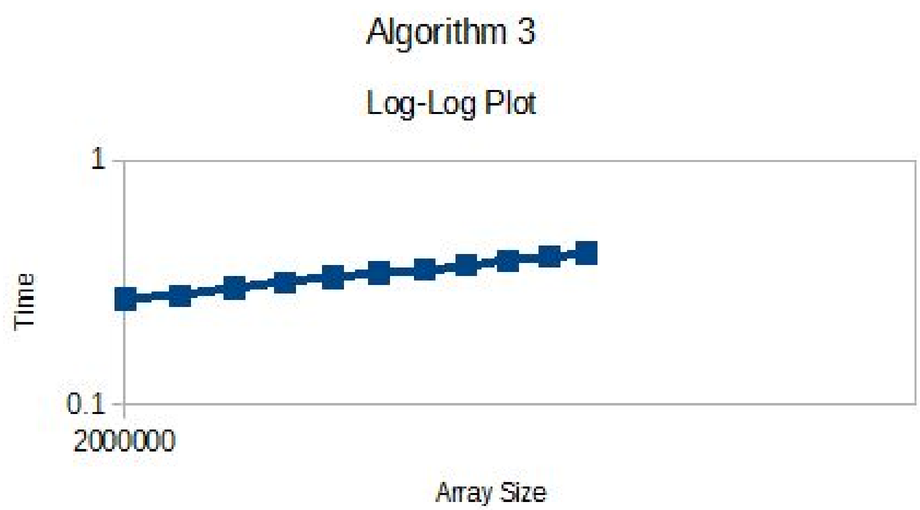
Largest Solvable Input in 10 Minutes:

Algorithm 1: Using the Cubic Formula:10354.774728147
Algorithm 2: Using the Quadratic Formula: 567457.8682526217
Algorithm 3: Using Narrowing/Elimination Methods: 2657731662
Algorithm 4: Using methods for Linear Equations: 43421003947.3665

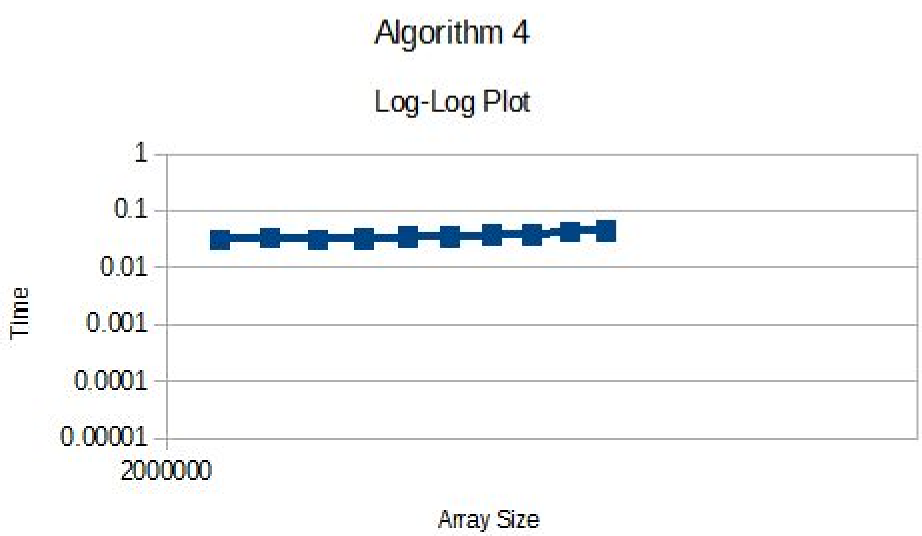
Log-Log Plot:


Algorithm 1
Log-Log Plot
100
10
1
Time
1
10
100
1000
10000
Array Size
A1 Mean


Algorithm 2
Log-Log Plot
100
10
1
Time
1
10
100
1000
10000
100000
Array Size
A2 Mean



Algorithm 3
Log-Log Plot
1
0.1
2000000
Time
Array Size



Algorithm 4
Log-Log Plot
1
0.1
0.01
0.001
0.0001
0.00001
2000000
Time
Array Size

Composite Graph: