Spring 2017

Project 1 – MSS

Group14

Team: Philip, Elizabeth, Aleita
4-23-2017

Problem Recap

Given an array of small integers A[1,…,*n*], compute $\max_{i \le j} \sum_{k=i}^{j} A[k]$, and determine the subarray
For example, MAXSUBARRAY([31, -41, 59, 26, -53, 58, 97, -93, -23, 84])=187 with subarray = [**59, 26,-53, 58, 97].**

Theoretical Run-Time Analysis

Algorithm #1 – Enumeration

Per the instructions, our first algorithm loops over each pair of indices i,j and computes the sum $\sum_{k=i}^{j} A[k]$. Keeping the best sum found so far.

```
vector <int> algorithm1(vector <int> v)
{
    int sum = 0;
    int maxSum = 0;
    vector <int> temp;
    vector<int> maxArr;

    for(int i = 0; i < v.size(); i++ )
    {
        for(int k=i; k < v.size(); k++) {
            sum = 0;
            temp.clear();

            for(int j=i; j < v.size(); j++) {
                sum +=v[j];
                temp.push_back(v[j]);
                if(sum > maxSum) {
                    maxSum = sum;
                    maxArr = temp;
                }
            }
        }
    }
    return maxArr;
}
```

Algorithm #1 has three “for” loops executing at O(n) time. Therefore, the asymptotic run-time is O(n^3).

Algorithm #2 – Better Enumeration

The enumeration can be computed more efficiently from $\sum_{k=i}^{j-1} A[k]$ in O(1) time:

```
vector <int> algorithm2(vector <int> array)
{
    vector <int> temp;
    vector <int> result;
    int sum = 0;
    int maxSum = 0;

    for(int i = 0; i < array.size(); i++) {
        sum = 0;
        temp.clear();

        for(int j = i; j < array.size(); j++) {
            sum = sum + array[j];
            temp.push_back(array[j]);

            if(sum > maxSum) {
                maxSum = sum;
                result = temp;
            }
        }
    }
    return result;
}
```

Algorithm #2 improves by only looping twice through the array, each loop executing at O(n) times. The asymptotic run-time is O(n^2).

Algorithm #3 – Divide & Conquer

Algorithm #3 works by dividing the left and right side up into two halves, then respectively dividing each half into more halves until it gets to the smallest element. Then returning in elements. Please see our implementation on the following page.

We determined the asymptotic run time to be Θ(nlogn) with following analysis:

For base case $\mathrm{n} = 0$ or $\mathrm{n} = 1$, algorithm #3 runs constant time $\Theta(1)$. For $\mathrm{n} > 1$, our algorithm has two recursive calls, for two subproblems of half size and 1 loop through n.

Base Case: $\mathrm{T(n)} = \Theta(1)$

For $\mathrm{n} > 2$: $\mathrm{T(n)} = 2\mathrm{T(n/2)} + \Theta(\mathrm{n})$

Using Master Method

$\mathrm{a} = 2, \mathrm{b} = 2, \mathrm{f(n)} = \Theta(\mathrm{n})$

$\mathrm{n^\wedge logb(a)} = \mathrm{n^\wedge log2(2)} = \mathrm{n^\wedge 1} = \Theta(\mathrm{n})$

Case 2 applies

$T(n) = \Theta(n^{\log 2(2)} * \log 2(n))$

$T(n) = \Theta(n * log_2 n)$

In summary the theoretical run-time for our divide and conquer algorithm is $\Theta(n\log n)$.

Algorithm #3 Code:

```
 ******************************************************************/
vector <int> algorithm3(vector<int> arrIn, int iLow, int iHigh)
{
    vector<int> resultL;
    vector<int> resultR;
    vector<int> resultX;
    vector<int> result;


    int sumL = 0;
    int sumR = 0;
    int sumX = 0;
    int iMid = 0;

    if(iHigh == iLow){
        result.push_back(iLow);
        result.push_back(iHigh);
        result.push_back(arrIn[iLow]);
        return result;
    } else {
        iMid = (iLow+iHigh)/2;
        resultL = algorithm3(arrIn, iLow, iMid);
        sumL = resultL[2];
        resultR = algorithm3(arrIn, iMid+1, iHigh);
        sumR = resultR[2];
        resultX=maxXsubArr(arrIn, iLow, iMid, iHigh);
        sumX = resultX[2];
    }
    if((sumL>=sumR)&&(sumL>=sumX))
        return resultL;
    else if((sumR>=sumL)&&(sumR >=sumX))
        return resultR;
    else
        return resultX;
}
```

Algorithm #4 – Linear Time

Algorithm # 4 determines the maximum subarray of the form A[i . . $j+1$] in constant time based on knowing a maximum subarray ending at index j. We then use those index positions to create a vector of the subarray values to be returned to the main function.

```
vector <int> algorithm4(vector <int> array)
{
    int n = array.size(); //# of elements in array/vector
    int max_sum = array[0];
    int current_sum = 0;
    int right = 0;
    int left = 0;
    int temp_left = 0;
    vector<int> maxArr;

    //calculate max sum & subarray positions
    for(int i = 0; i < n; i++) {
        current_sum = max((current_sum + array[i]), array[i]);
        if(current_sum > max_sum) {
            max_sum = current_sum;
            right = i;
            left = temp_left;
        }
        if( current_sum == array[i]) {
            temp_left = i;
        }
    }

    //create vector based on positions calculated above
    for(int j = left; j <= right; j++) {
        maxArr.push_back(array[j]);
    }
    return maxArr;
}
```

The main portion of algorithm #4, which determines the maximum subarray sum and elements make up that subarray (identified by starting and stopping position), iterates through the given array only once. The second is a utility loop to convert the solution into a vector that be easily returned. Given those complexities, T(n) = 2n , the asymptotic run-time is O(n).

Testing

To test our algorithms "correctness" we created the main program in the separately attached file. We used the provided test set to compare our input & output results and we are confident that there no problems with our algorithms. We also created our own test sets and manually checked.

Based on our testing we are confident in the results provided in MSS_Results.txt.

Experimental Analysis

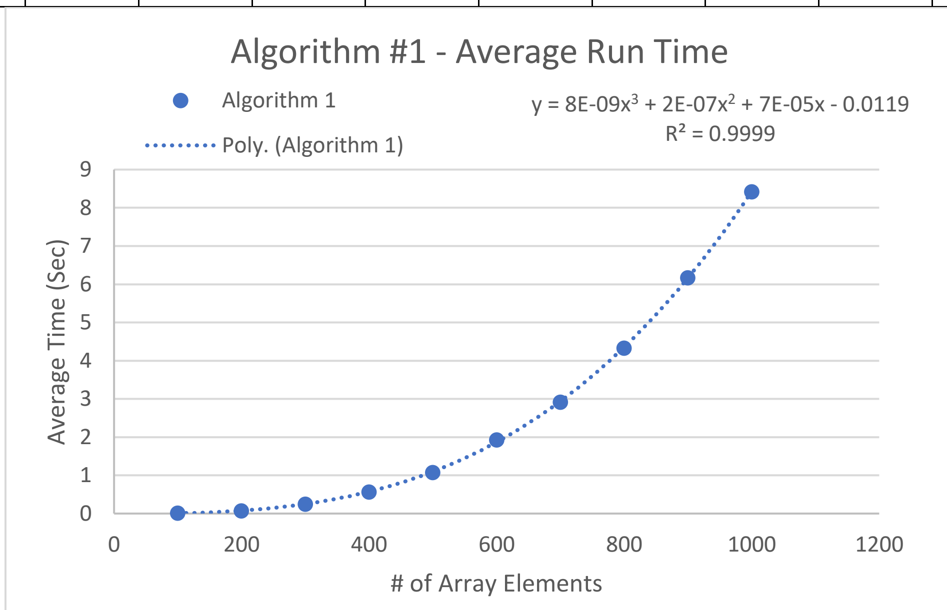
In this section you will find average run time charts and plots, regression modeling, log-log plots, and any deviation analysis, for each of the four algorithms. We then use our models to determine the maximum size of each array that can be processed in 5, 10, and 60 seconds.

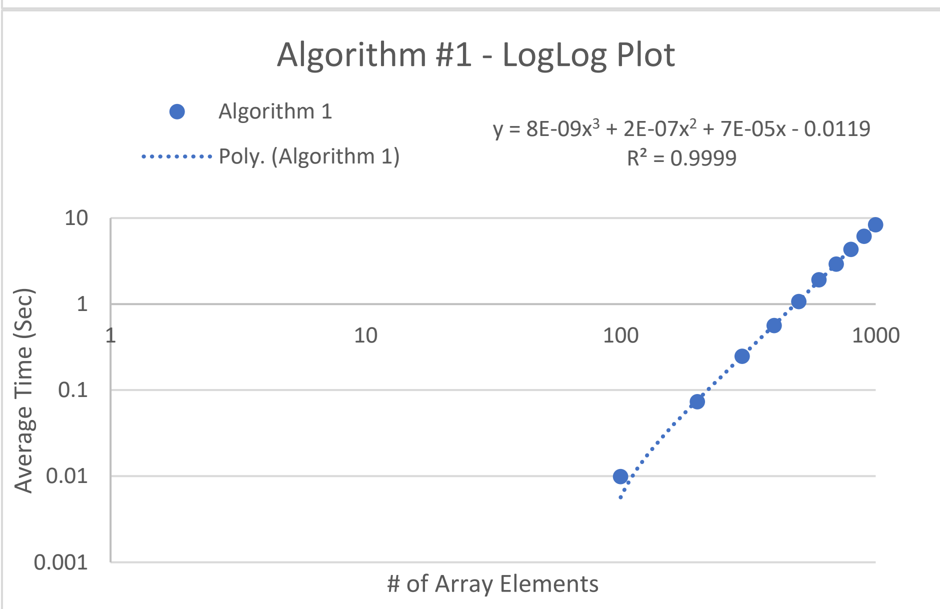
Each average run time is based on 10 experimental times for each (n) number of elements.

Algorithm #1 – Enumeration

Average Run Times

N:	100	200	300	400	500	600	700	800	900	1000
Sec:	0.009	0.074	0.247	0.563	1.075	1.931	2.919	4.330	6.167	8.423





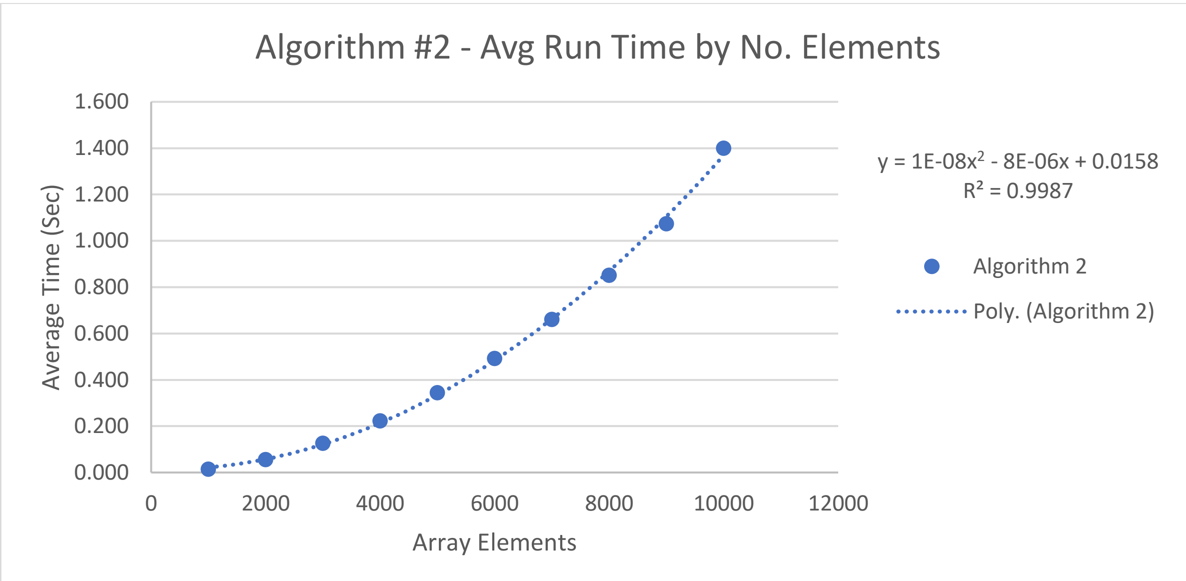
Our algorithm #1 experiment results matched our theorized run time expectations of a polynomial trend n^3. Based on the regression model function: $y = 8E\text{-}09x^3 + 2E\text{-}07x^2 + 7E\text{-}05x - 0.0119$ we estimate the maximum array sizes processed to be:

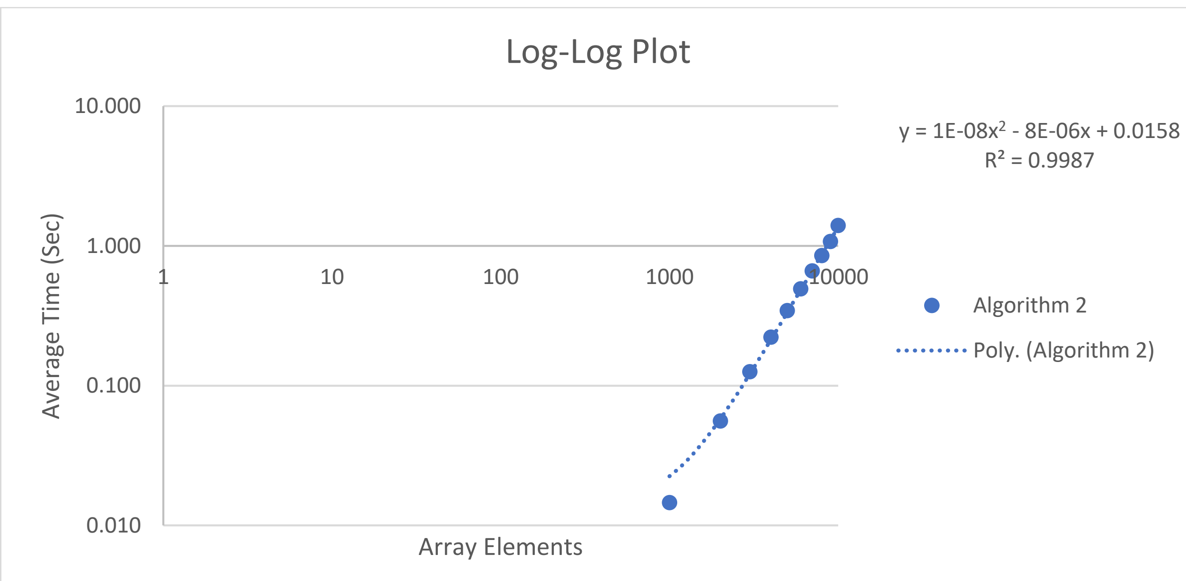
Time:	5 seconds	10 seconds	60 seconds
Max N:	844.038	1,066.69	1,947.78

Algorithm #2 – Better Enumeration

Average Run Times:

N:	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Sec:	0.015	0.056	0.126	0.223	.344	0.492	0.661	0.852	1.074	1.400





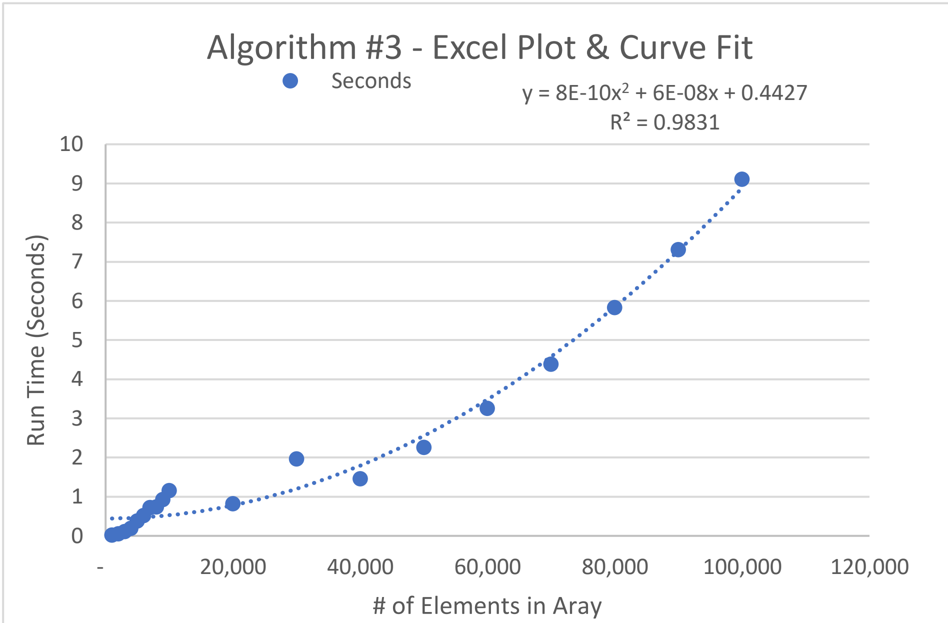
Algorithm #2 experimental run times demonstrate no discrepancies. Based on the regression model function: $y = 1E\text{-}08x^2 - 8E\text{-}06x + 0.0158$ we estimate the maximum array sizes processed to be:

Time:	5 seconds	10 seconds	60 seconds
Max N:	22,728.9	32,000.3	77,050.5

Algorithm #3 – Divide & Conquer

Average Run Times:

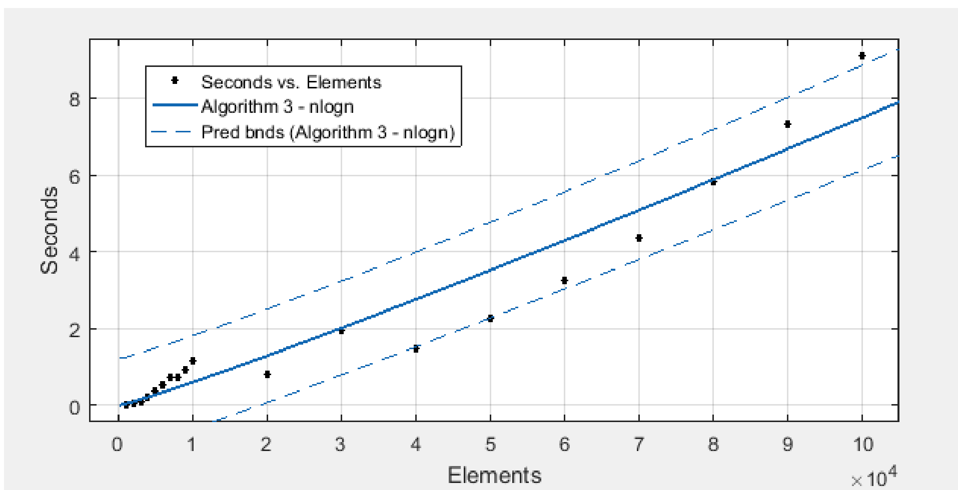
N	Seconds
1,000	0.0141
2,000	0.0495
3,000	0.1111
4,000	0.1925
5,000	0.3748
6,000	0.5201
7,000	0.7239
8,000	0.7403
9,000	0.9286
10,000	1.1519
20,000	0.817
30,000	1.959
40,000	1.458
50,000	2.254
60,000	3.253
70,000	4.38
80,000	5.829
90,000	7.31
100,000	9.106



MATLAB Custom Fit :

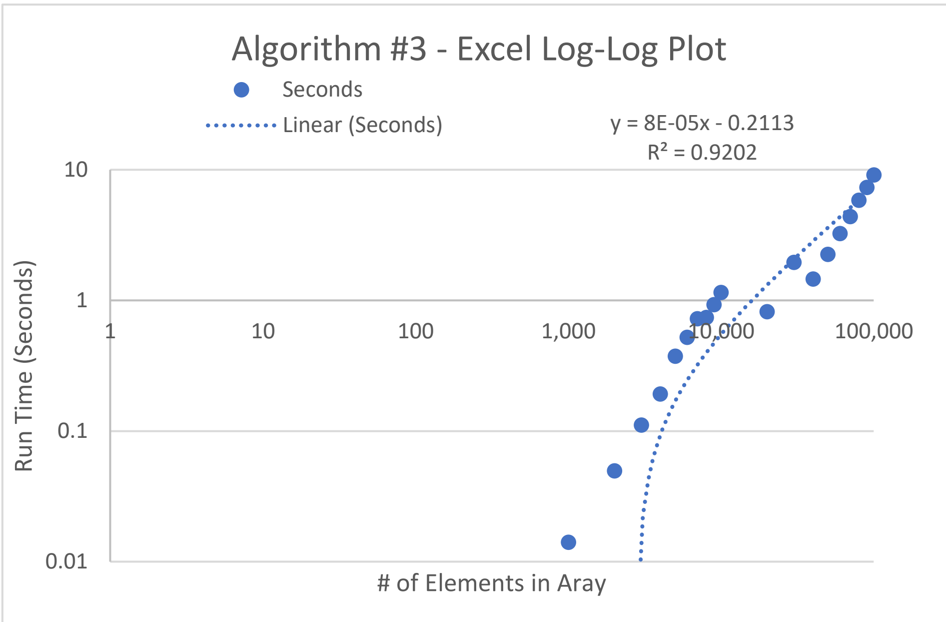
General model:
f(x) = a*x*log(x)
Coefficients (with 95% confidence bounds):
a = 6.51e-06 (5.845e-06, 7.174e-06)

Goodness of fit:
SSE: 8.79
R-square: 0.9304
Adjusted R-square: 0.9304
RMSE: 0.6988



Using MatLab to custom curve fit found that while the experimental run times of algorithm #3 have the best R value,"fit", to a polynomial function, they are also within statistical range of the anticipated n*log(n). Our experiment results fall within 90% predictive bounds of the theorized run-time.

Still, we did not expect our run-times to be a perfect fit due to variations in our code implementation and systems. For example, we removed the cross over function from the main algorithm and had it separate from the main recursive call.

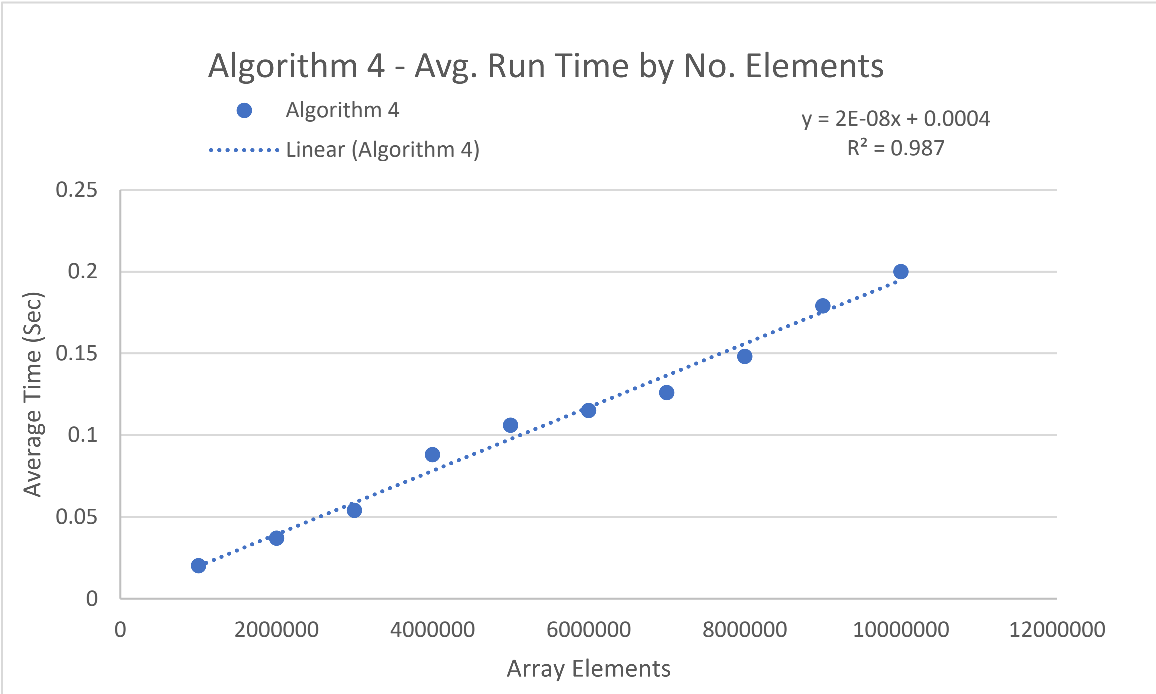


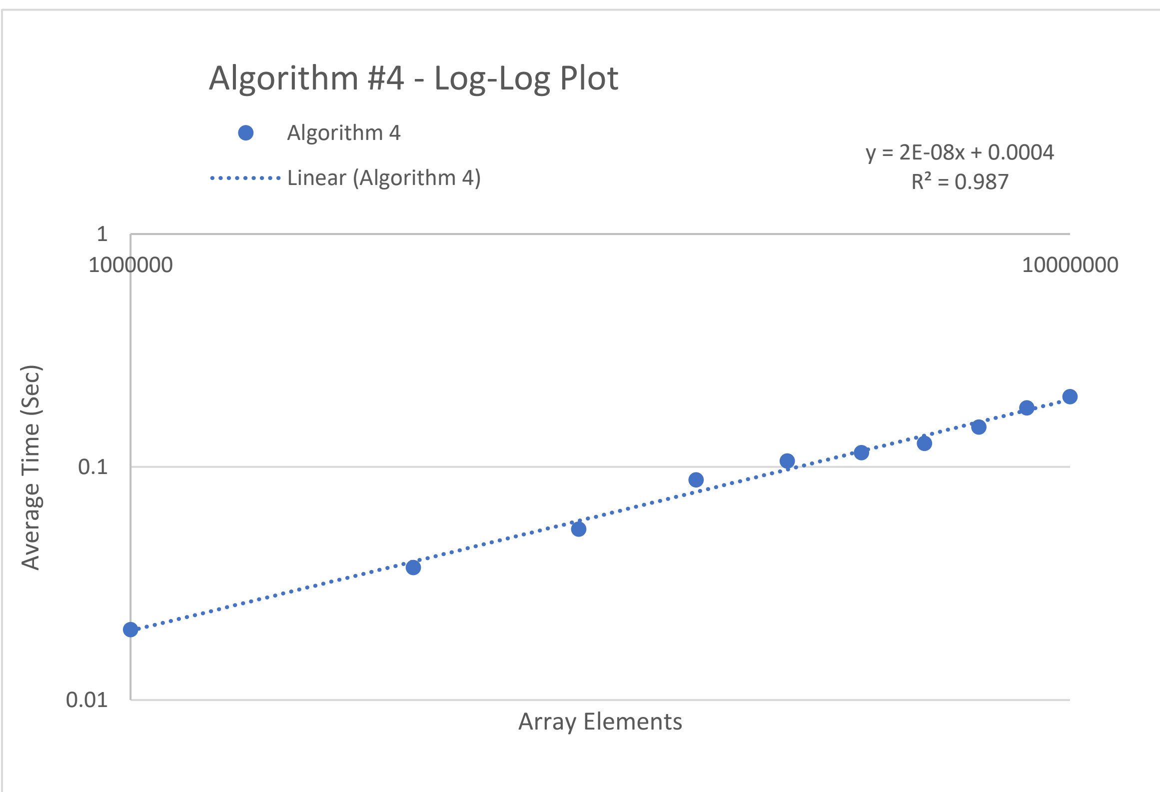
Based on the regression model function: y = 8E-10x^2 + 6E-08x + 0.4427 we estimate the maximum array sizes processed to be:

Time:	5 seconds	10 seconds	60 seconds
Max N:	75,438.5	109,263	272,812

Algorithm #4 – Average Run Times

There were no discrepancies with our experiment results.





Based on the regression model function: y = 2E-08x + 0.0004 we estimate the maximum array sizes processed to be:

Time:	5 seconds	10 seconds	60 seconds
Max N:	249,980,000	~499,980,000	2,999,980,000

Algorithm Comparison Graph