Meredith Kiessling
Evan Lyon
Fahmy Mohammad

PROJECT 1: MAXIMUM SUM SUBARRAY (GROUP 32)

THEORETICAL RUN-TIME ANALYSIS

Algorithm 1 (Enumeration):

Pseudo-code:

For each pair, (i, j) in array of size n, compute the sum of each possible sub-array([i]…[j]), storing the maximum sub-array sum encountered. Compute (a[0..1], a[0…2], etc)

```
for(i=0;  i<n; i++)
    for(j =i; j<n; j++)
        -reset current sum to zero
        for(k=i; k<=j;k++)
        -compute sum of a[i]… a[j]
          -keep maximum sum
-return maximum sum
```

Theoretical Analysis:

Each for-loop iterates n times. There are three for-loops, therefore the theoretical run-time = $O(n^3)$.

Algorithm 2 (Better Enumeration):

Pseudo-code:

Starting at i = 0, iterate over the array. Compute sum for each sub-array a[0… n], a[1….n], a[2…n], …, a[n-1…n]. Keep the maximum sub-array encountered.

```
for(int i = 0; i<n; ++i)
        -set running sum = a[i]
        for(j=i; j < n; ++j)
        -compute running sum (a[i] += a[j])
        -keep max sum
-return maximum sum
```

Theoretical Analysis:

Each for-loop iterates n times. There are two for-loops, therefore the theoretical run-time = $O(n^2)$.

Algorithm 3 (Divide and Conquer):

Pseudo-code:

```
maxSubArray(A, low, high)
        if high == low
                return {low, high, A[low]
        mid = (low + high) / 2
        {left.low, left.high, left.sum} = maxSubArray(A, low, mid)
        {right.low, right.high, right.sum} = maxSubArray(A, mid + 1, high)
        {cross.low, cross.high, cross.sum} = maxCrossSubArray(A, low, mid, high)
        if left.sum >= right.sum and left.sum >= cross.sum
                return {left.low, left.high, left.sum}
        else if right.sum >= left.sum and right.sum >= cross.sum
                return {right.low, right.high, right.sum}
        else
                return {cross.low, cross.high, cross.sum}

maxCrossSubArray(A, low, mid, high)
        leftsum = 0
        sum = 0;
        for i = mid down to low
                sum = sum + A[i]
                if sum > leftsum
                        leftsum = sum
                        cross.left = i
        rightsum = 0
        sum = 0;
        for i = mid + 1 up to low
                sum = sum + A[i]
                if sum > rightsum
                        rightsum = sum
                        cross.right = i
        return {cross.left, cross.right, leftsum + rightsum}
```

Theoretical Analysis:

This is a recursive algorithm that splits the problem into two n/2 subproblems and spends T(n/2) solving each of them. The two recursive cases take 2T(n/2) to solve for the left and right subarrays. The case where the subarray crosses the midpoint is solved by the separate linear

function maxCrossSubArray. This function has two loops that each take $\Theta(1)$ time so the total time is $\Theta(n)$.
Recurrence:
$T(n) = \Theta(1)$ if $n = 1$
$T(n) = 2T(n/2) + \Theta(n)$ if $n > 1$
using master method, $a = 2$, $b = 2$, $n^{\log_b a} = n^{\log_2 2} = n^1$
compare n with $f(n) = \Theta(n)$
$f(n) = \Theta(n^{\log_b a})$, then: $T(n) = \Theta(n^{\log_b a}\lg n) = \Theta(n^1 \lg n) = \Theta(n \lg n)$

Algorithm 4 (Dynamic Programming):

Pseudo-code:

This algorithm iterates over the array 1 time. With each iteration, it keeps track of the lower and upper indices defining the maximum sub-array. When the sum calculated is greater than the current maximum sum, store the sum and the indices defining the max sub-array. If the current sum is zero, simply add the current value to the sum and store the upper index (e.g. expand the maximum sub-array). But if the current sum is negative, we know that this is not the maximum sub-array, so set maximum sum equal to the current value and store the current index as the lower bounds for the max sub-array.

```
-set max Sum to negative infinity
-set sumHere to negative infinity
for (i=0; i < n; i++)
        -set lower bounds of potential max-subarray equal to I (startHere = i)
        -if sumHere > 0
        sumHere = sumHere + a[i]     //increase running sum
        -else
        sumHere = a[i]       //start new sub-array, set sum = a[i]
        startHereIdx = i      //keep track of start of new sub-array
        -if sumHere >maxSum
        maxSum = sumHere
        low = startHere
        high = endHere
return max
```

Theoretical Analysis:

The for-loop iterates one time over each item in the array (e.g. O(n) operations are performed) and each operation within the for-loop occur in constant time $\Theta(1)$. Therefore, $T(n) = O(n) + \Theta(1) \Rightarrow T(n) = O(n)$.

TESTING

The algorithms were tested for correctness using the test cases provided in the MSS_TestProblems.txt file. Additional test cases were also implemented to verify edge conditions. Test cases were created such that the maximum sub-arrays and minimum sub-arrays were located in different sections (i.e. first quarter, middle, and end) of the overall array.

EXPERIMENTAL ANALYSIS

Algorithm 1 (Enumeration):

The results of the experimental analysis for Algorithm 1 (Enumeration) are summarized below in Figures 1 and 2. Analysis of the data yielded a regression equation: y = 2.753E-6x^3 - 2.005E-4x^2 + 0.268x-50.972 (R^2 = 1). The results of the experimental analysis are consistent with the theoretical analysis, indicating Algorithm 1 is exponential in nature and bounded by O(n^3).

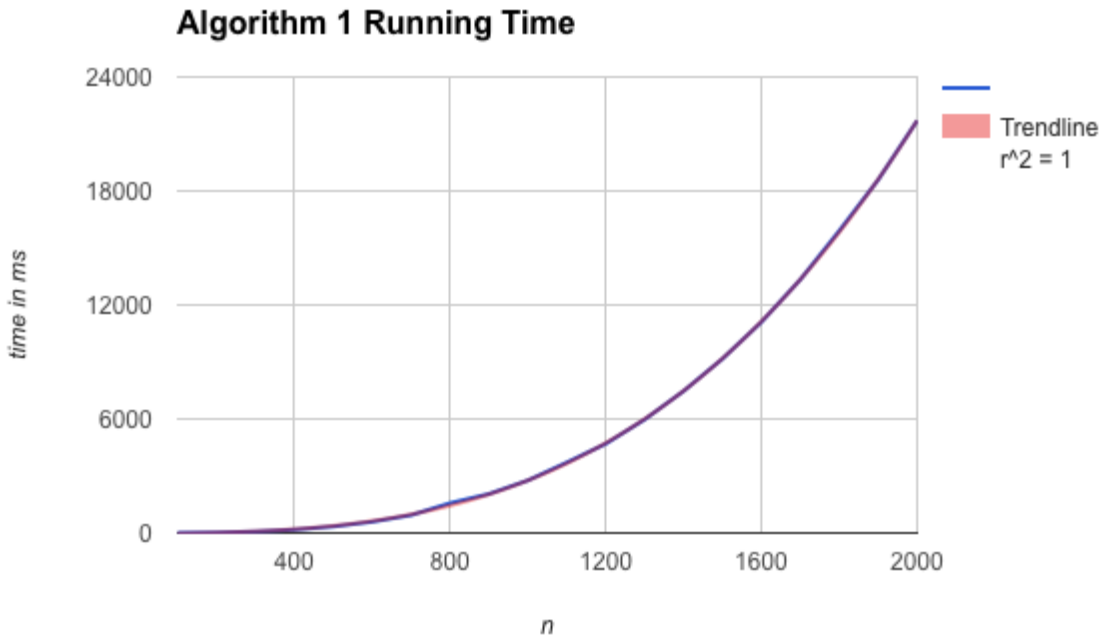

Figure 1: Algorithm 1 - (Enumeration) Experimental Results

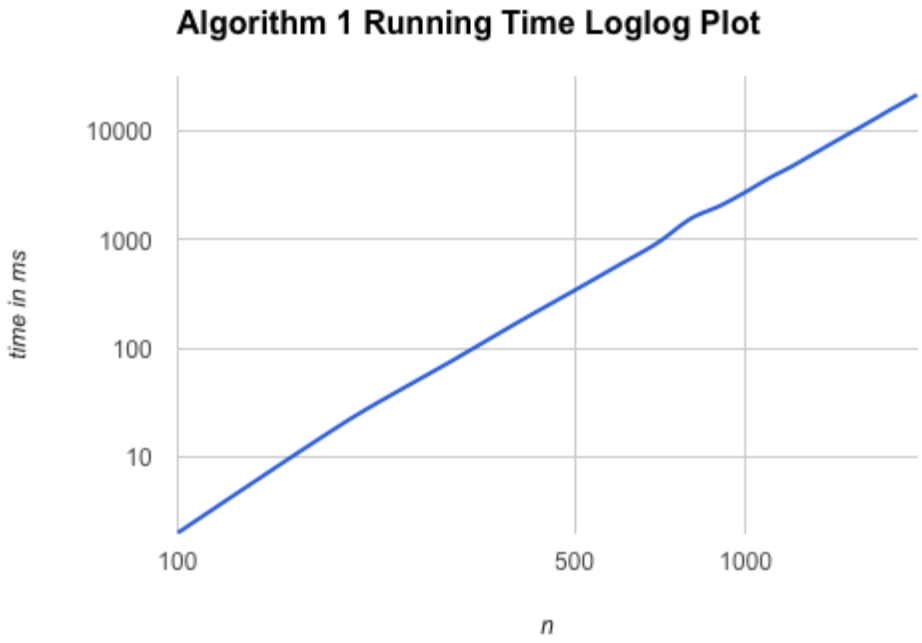

Figure 2: Algorithm 1 - (Enumeration) Loglog Plot

Algorithm 2 (Better Enumeration):

The results of the experimental analysis for Algorithm 2 (Better Enumeration) are summarized below in Figures 3 and 4. Analysis of the data yielded a regression equation:y = 1.101E-5x^2 - 3.618E-3x - 2.742 (R^2 = 0.999). The results of the experimental analysis are consistent with the theoretical analysis, indicating Algorithm 2 is quadratic in nature and bounded by O(n^2).

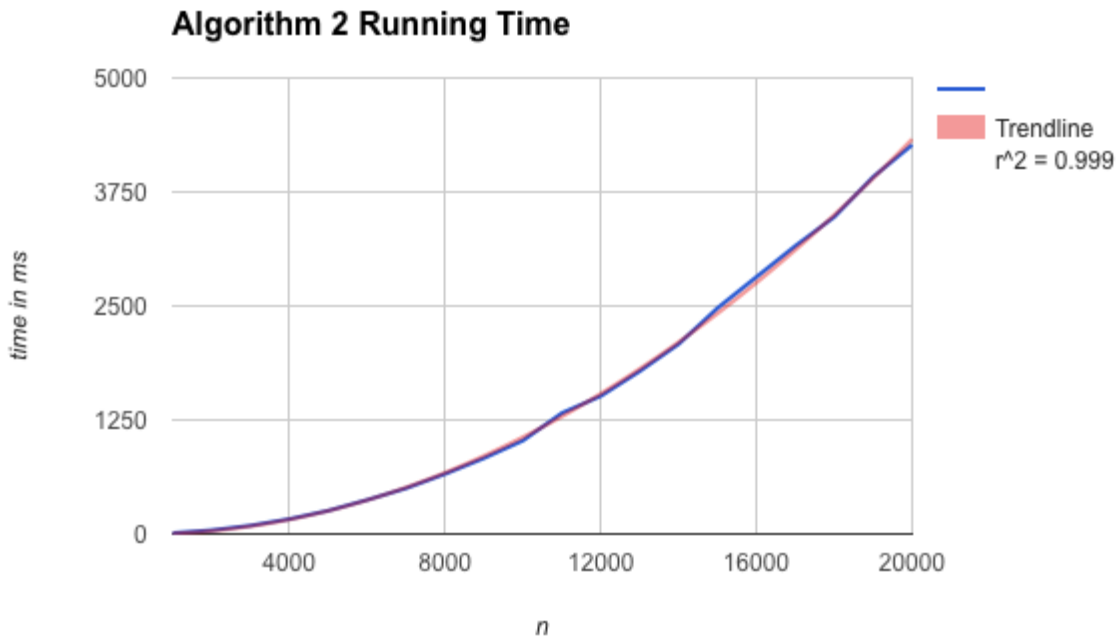

Figure 3: Algorithm 2 - (Better Enumeration) Experimental Results

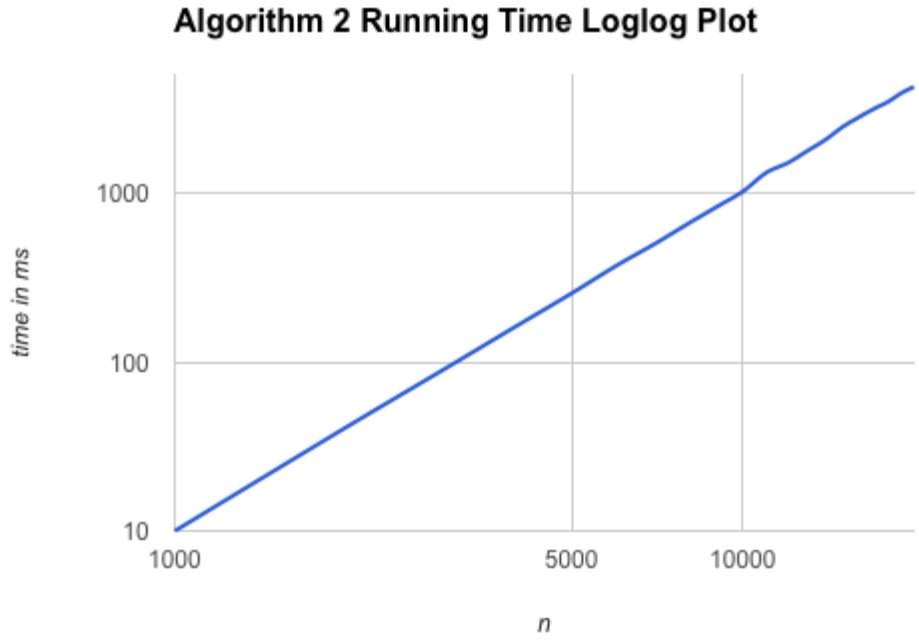

Figure 4: Algorithm 2 - (Better Enumeration) Loglog Graph

Algorithm 3 (Divide and Conquer):

The results of the experimental analysis for Algorithm 3 (Divide and Conquer) are summarized below in Figures 5 and 6. Analysis of the data yielded a regression equation: y=3.115E-6*e^(1.3361x) (R^2 = 0.9929). As shown in Figure 5, the graph is logarithmic. The results of the experimental analysis are consistent with the theoretical analysis, indicating Algorithm 3 is logarithmic in nature and bounded by O(n*log*n).

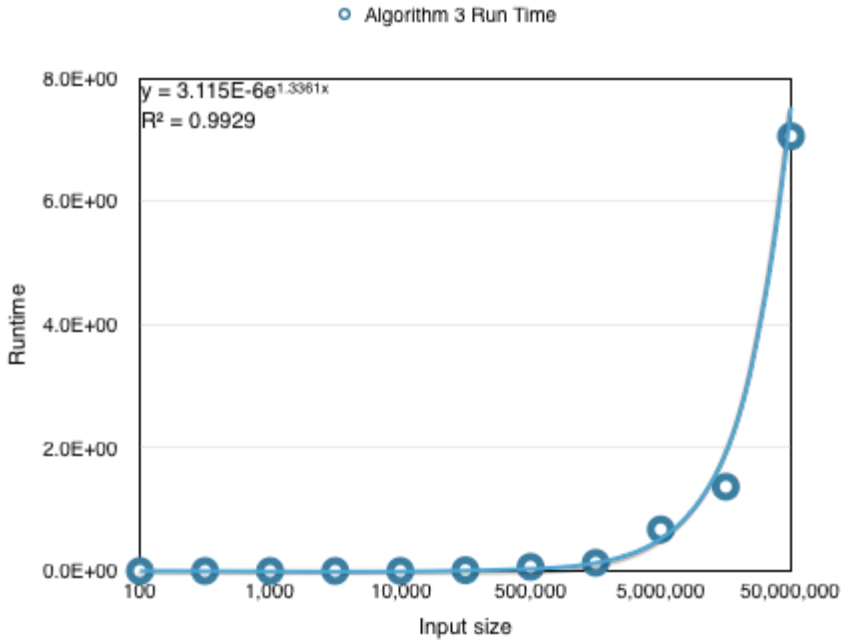

Figure 5: Algorithm 3 - (Divide and Conquer) Experimental Results

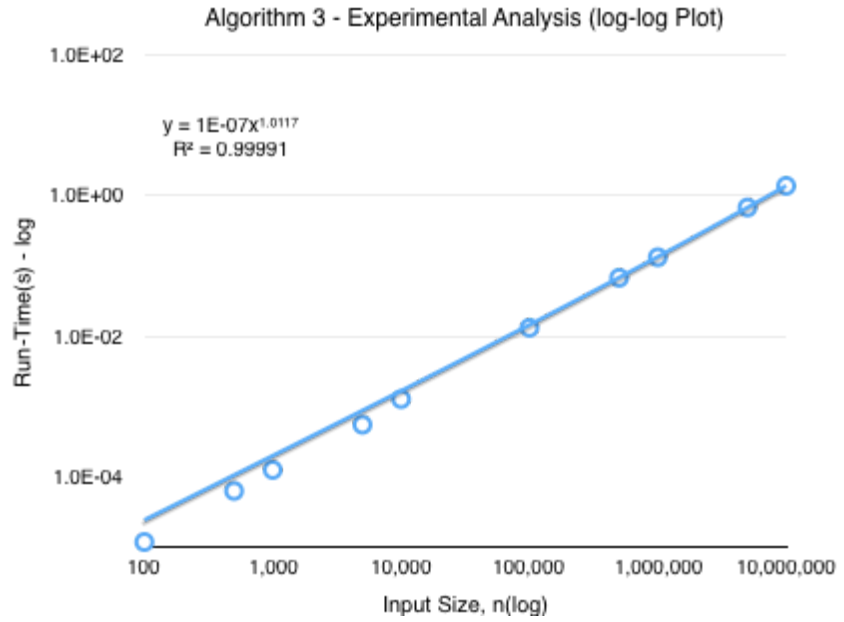

Figure 6: Log-Log Plot of Algorithm 3

Algorithm 4 (Dynamic Programming):

The results of the experimental analysis on the dynamic programming algorithm to determine the maximum sub-array are shown below (see Figure 7). The regression analysis yielded a linear relationship (y = 4E-09x-0.0025) with an r-squared value of 0.99717, indicating a strong level of confidence.

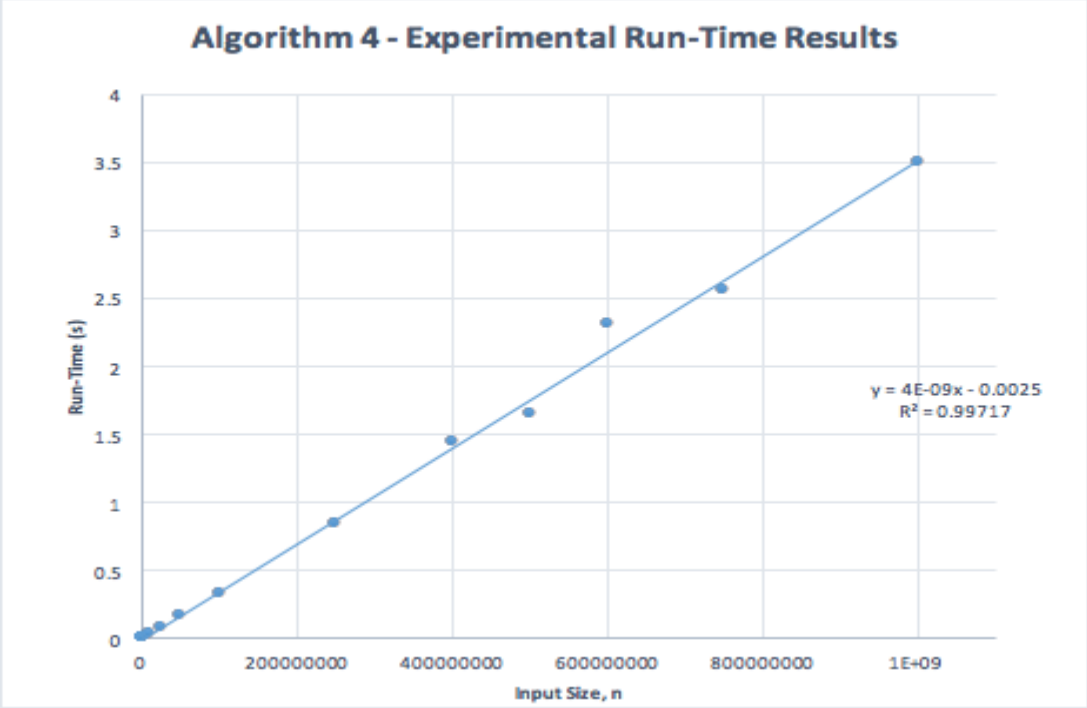

Figure 7: Algorithm 4 - (Dynamic Programming) Experimental Results

Additionally, the analysis of log-log plot of the experimental data from algorithm 4 (shown below in Figure 8), yielded a regression of y = 9E-09x^(0.9488) with an r-squared value of 0.99833, indicating a high level of confidence. The log-log regression is in the form of y= a*x^k (power function), which correlates to a linear function: y = k*x + log(a), with k equal to the slope and log(a) equal to the y-intercept. Converting the log-log regression into a linear function yields: y = 0.9488x - 8.05, which is consistent with both the theoretical run-time analysis, and experimental linear regression analysis.

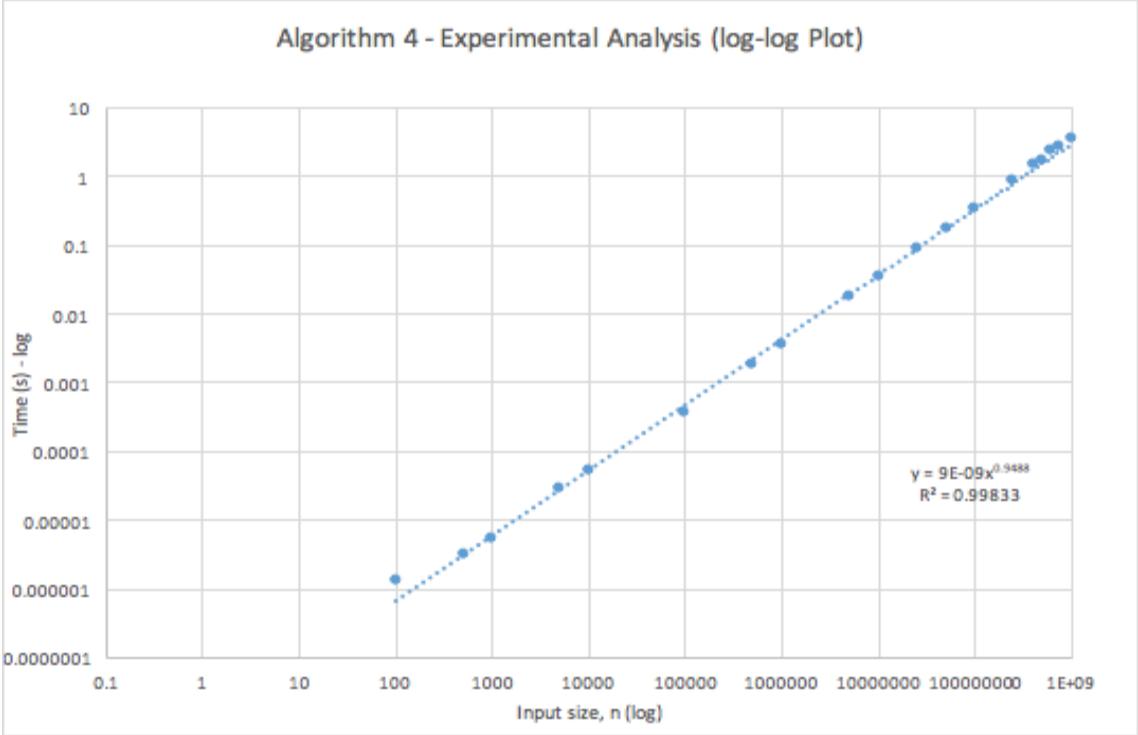

Figure 8: Log-Log Plot of Algorithm 4

The results of the experimental analysis (both linear and log-log regressions) for algorithm 4 are expected, and are consistent with the theoretical theoretical run-time analysis, indicating that algorithm 4 - dynamic programming - is linear with a run-time of O(n).

SUMMARY

The results of the experimental analysis were expected and were in agreement with the theoretical analysis. The Theoretical and Experimental run-time analyses are summarized in Table 1. In addition, the experimental results for all four algorithms are shown below in Figure 9.

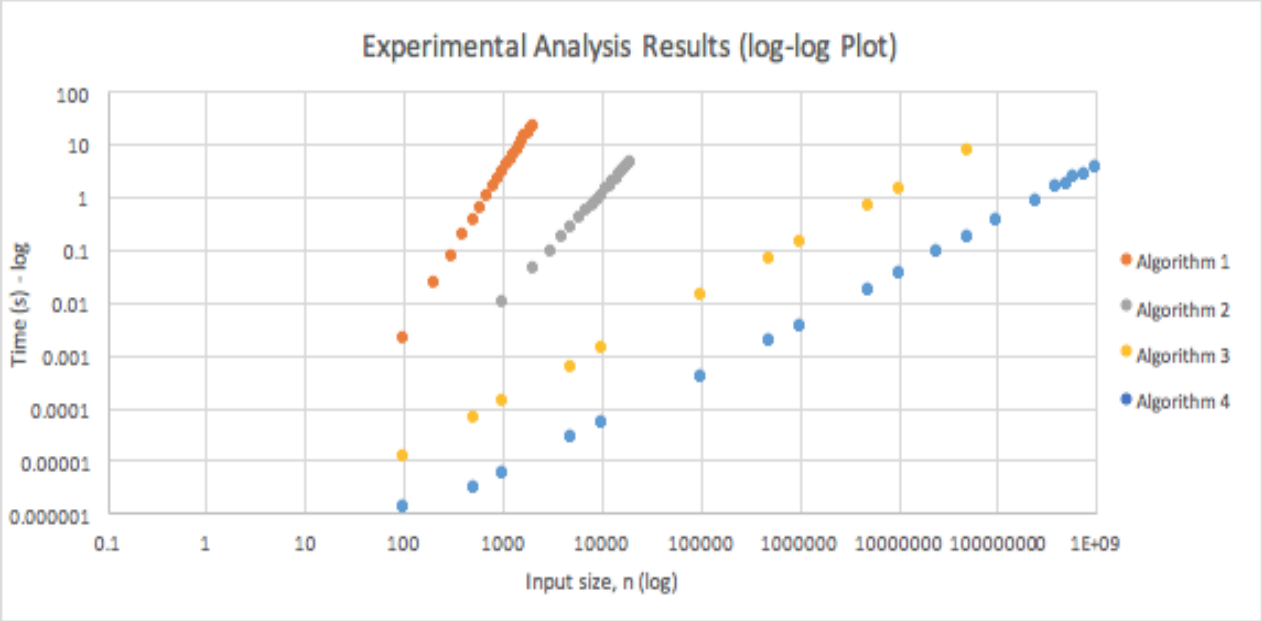

Figure 9: Summary of Experimental Results

Table 1: Summary of Theoretical and Experimental Run-Times

Algorithm	**Theoretical Run-Time**	**Experimental Regression Analysis Equations**	**R^2**
1 - Enumeration	O(n^3)	y = 2.753E-6x^3 - 2.005E-4x^2 + 0.268x-50.972	1
2 - Better Enumeration	O(n^2)	y = 1.101E-5x^2 - 3.618E-3x - 2.742	.999
3 - Divide and Conquer	Θ(n*lg*n)	Y =3.115E-6*e^(1.3361x)	0.9929
4 - Dynamic Programming	O(n)	y = 4E-09x-0.0025	0.99717

As expected, the Algorithm 1 (Enumeration -or- Brute Force) performed the slowest of the four algorithms, while Algorithm 4 (Dynamic Programming) performed the fastest. Algorithm 2 (Better Enumeration) performed slightly more efficiently than Algorithm 1, and Algorithm 3 (Divide and Conquer) performed slightly faster than Algorithm 2.

Table 2 below shows the largest amount of input each algorithm can solve in a specified amount of time (time = 10, 30, 60 seconds) per the experimental regression equations listed in Table 1. Unsurprising, the input size that can be solved with each algorithm is consistent with the results of the experimental analysis and the theoretical analysis (with Algorithm 1 solving the smallest amount of input to Algorithm 4 solving the most in a given time).

Table 2: Largest Input Solved in Time t (per Regression Analysis Eq. in Table 1)

Algorithm	**t =10s**	**t= 30s**	**t=1min (60s)**
1 - Enumeration	1543	2228	2807
2 - Better Enumeration	30320	52390	74021
3 - Divide and Conquer	8.07E07	2.38E08	4.74E08
4 - Dynamic Programming	2.5E09	7.5E09	1.5E10