CS325 Project 1

Aaron Fifer

Spencer Moran

Seth Weston

Group 32

I. Theoretical Run Time Analysis

a. BruteForceMethod

```
void MaxSumBruteForce(int * a, int size, int arrayCount)
{
   Create variables to track the return value, sum, and indicies.
   loop through size-2;
         set sum to 0;
         loop through size -1;
               increment sum by the value in the array
               update the max and indicies if the sum exceeds the current max
    return the final sum.
}
```

The brute force method tests every possible sub array against the current best sum, returning the best sum at completion. Thus, the method makes $n + (n-1) + (n-2) + \cdots + 1$ operations n times. $T(n) = O(n^3)$.

b. ImprovedBruteForceMethod

```
  void MaxSumImprovedBruteForce(int * a, int size, int arrayCount)
{
     Create variables to track the return value, and indices.
     Loop through and parse the array.
           Track the sum and initialize the iterators
           create a temp variable to memorize previous solutions. Set to MaxStart.
           Loop backwards until hitting the temp variable.
                 Increment the sum by the array values.
                 If the sum is greater than the return value, mark the indices.

     Return the max sum when finished.

}
```

The improved brute force method makes use of memoization to enable calculation of previously solved pairs in constant time. This removes a number of redundant calls, improving run time to $T(n) = O(n^2)$.

c. Divide and Conquer

This algorithm required a helper function to calculate the max crossing.

Create struct maxSumTuple, to track left and right indices, and the max sum.

```
maxCrossing (int values[], int left, int right, int middle) {
Create variables to track the leftSum, the sum, and declare an instance of the struct to hold the final values.
Loop through the values in the left using the passed in variables indicating the left, right and center indices.
    Increment sum with values from the array.
    If the sum exceeds the leftSum, update the sum and indices.
Repeat the process for the right side.
Return the tuple containing sum of the left and right values and their indices.
```

```
divideConquer(int values[], int left, int right) {
Establish the base case by checking if the left and right indices are the same.
If the same, initialize all values for the tuple and return the tuple.
Get the middle of the array by obtaining the average of the left and right indices.
Recurse over the left, right, and max Crossing.
   maxSumTuple leftSum = divideConquer(values, left, middle);
   maxSumTuple rightSum = divideConquer(values, middle + 1, right);
   maxSumTuple crossing = maxCrossing(values, left, right, middle);
Return the maximum value between the leftSum, rightSum, and the crossing.
```

Implementation of this algorithm required creation of the maxSumTuple struct, which tracked the maximum sum and indices for the maximum subarray. The algorithm evaluates the left and right sides, and then compares these sides against a cross of the sides and selects the largest sum for the final value. The algorithm must run in at least n time since every element must be evaluated, and the calculations require an additional lgn time. $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$. By Master Method. $T(n) = \Theta(nlgn)$.

d. Linear Time DP

```
Create variables to track the max indices, current index, max sum, and current sum.
Loop through the array.
    Increment the current sum by each value in the array.
    If the current sum exceeds the max sum,
            set the current sum as the new sum.
            update the starting and terminating max index.
    Else if the current sum dips below zero,
            If the value of the current iterator position is greater than the maxSum,
                    Then the current iterator is the least negative value so far.
            Regardless, reset the current Sum, as the end of the max array may
            have been reached. And continue until the end of the array.
Return the max sum and indices. Via the struct created in problem c.
```

The linear time algorithm works by parsing through the array and adding each element to the sum of the last element. It then checks this sum against the maximum sum and if it exceeds it, marks the index and moves on. If the current sum dips below zero, then the sum is reset, and the algorithm begins working on the next positive sub array. In the case where all values are negative, the max will be the least negative value. This continues until the entire array has been processed. Each of these operations takes place in constant time, requiring n operations to parse the entire array. Thus, the algorithm run in linear time. $T(n) = \Theta(n)$

II. Testing

To test our algorithms for correctness, we ran them against the MSS_Problems.txt file that was provided in the coursework, and compared our outputs to the expected result file which was also provided. Once we were satisfied that our algorithms passed the test cases, we began generating arrays of random values and checking the runtimes of those values for increasing values of n. The runtime for each test was measured in system time divided by clock cycles, and measured down to microseconds for small values of n. Section III contains the results of these tests.

III. Experimental Analysis

This section lists the data collected from the timing analysis of each algorithm. All run times are listed in seconds. The line best fit equation is presented in each of the graphs of the data using regression functions in excel.

A. Brute Force

	Brute Force								
n=100	n=200	n=500	n = 1000	n=2000	n=5000	n=10000	n=20000	n=50000	n=1000000
2.00E-05	5.30E-05	0.000314	0.001489	0.005105	0.032017	0.133363	0.531272	3.3681	13.2832
1.80E-05	6.20E-05	0.000388	0.001518	0.005294	0.03276	0.132925	0.541575	3.32885	13.2687
1.70E-05	6.30E-05	0.000395	0.001452	0.004989	0.032542	0.130558	0.534111	3.31621	13.3264
1.60E-05	6.20E-05	0.000352	0.001343	0.00491	0.034062	0.131184	0.519187	3.29927	13.3021
1.60E-05	6.90E-05	0.000412	0.00137	0.005179	0.035344	0.134863	0.52589	3.33833	13.5319
1.70E-05	6.70E-05	0.00035	0.001399	0.005358	0.033945	0.140684	0.526301	3.34994	13.3201
1.80E-05	6.40E-05	0.000423	0.001486	0.005322	0.032465	0.141604	0.529129	3.30668	13.3379
1.80E-05	6.20E-05	0.00039	0.001374	0.005287	0.032296	0.138743	0.529723	3.30983	13.3526
1.60E-05	6.70E-05	0.000382	0.001407	0.005694	0.034539	0.132262	0.522947	3.36832	13.3437
2.60E-05	6.20E-05	0.000373	0.001487	0.004973	0.035327	0.133398	0.53646	3.33432	13.51
Avg	Avg	Avg	Avg	Avg	Avg	Avg	Avg	Avg	Avg
1.82E-05	**6.31E-05**	**0.000378**	**0.001433**	**0.005211**	**0.03353**	**0.134958**	**0.52966**	**3.331985**	**13.35766**

Table 1: Brute Force Timing Data

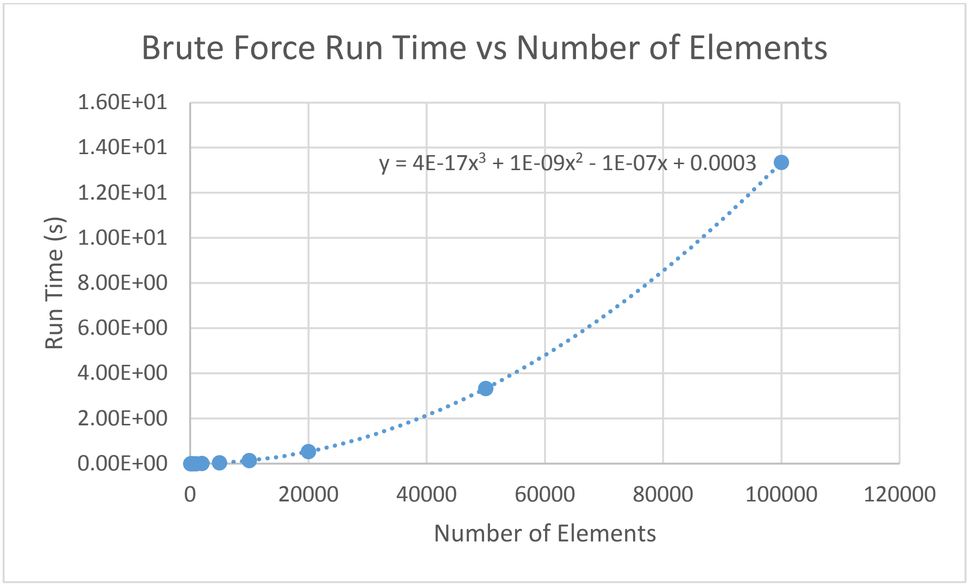

Figure 1: Brute Force Run Time vs Number of Elements

Using regression equation $y = 4 * 10^{-17}x^3 + 1 * 10^{-9}x^2 - 1 * 10^{-7}x + 0.0003$
We find the number of elements that the algorithm can work for in 5, 10, and 60 seconds.

$$(1)\ 5 = 4 * 10^{-17}x^3 + 1 * 10^{-9}x^2 - 1 * 10^{-7}x + 0.0003 \rightarrow x = 70658.7$$
$$\boldsymbol{\#\ elements = 70658}$$
$$(2)\ 10 = 4 * 10^{-17}x^3 + 1 * 10^{-9}x^2 - 1 * 10^{-7}x + 0.0003 \rightarrow x = 99849.2$$
$$\boldsymbol{\#\ elements = 99849}$$
$$(3)\ 60 = 4 * 10^{-17}x^3 + 1 * 10^{-9}x^2 - 1 * 10^{-7}x + 0.0003 \rightarrow x = 243812$$
$$\boldsymbol{\#\ elements = 243812}$$

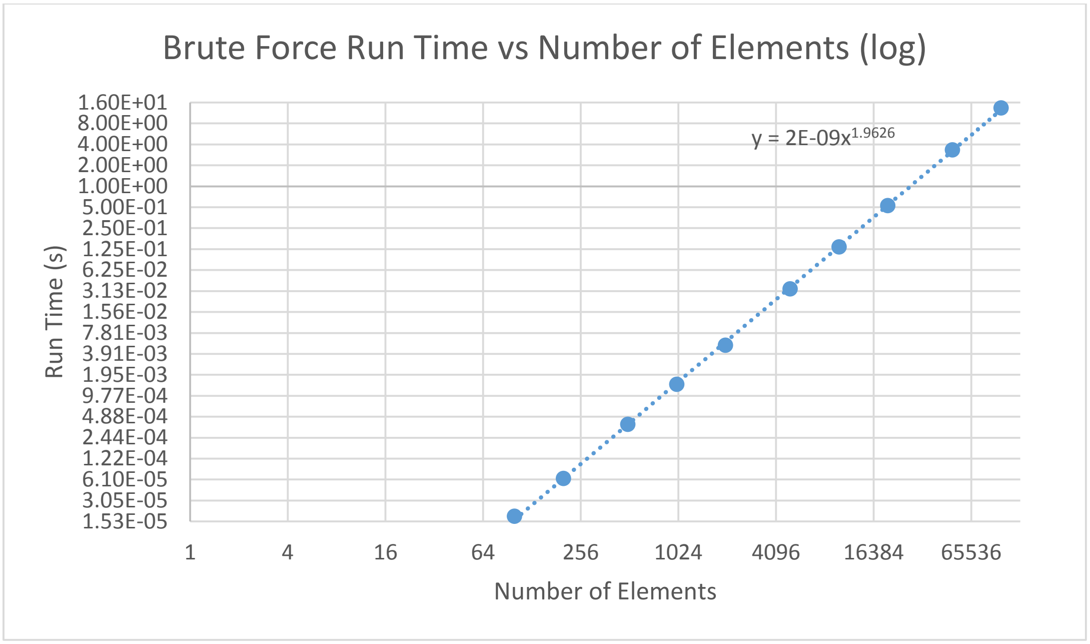

Figure 2: Brute Force Run Time vs Number of Elements log log plot

B. Improved Brute Force

Improved Brute Force									
n=100	n=200	n=500	n = 1000	n=2000	n=5000	n=10000	n=20000	n=50000	n=1000000
8.00E-06	4.00E-05	0.000137	0.000618	0.005182	0.009162	0.080605	0.295142	1.42131	9.95867
1.40E-05	2.70E-05	0.000151	0.001058	0.004064	0.024849	0.076248	0.223009	0.965965	6.85034
8.00E-06	2.20E-05	0.000205	0.000781	0.002704	0.021468	0.044611	0.279406	1.25897	4.72566
8.00E-06	4.20E-05	0.000187	0.000828	0.002633	0.022205	0.066709	0.322016	2.03356	10.2935
1.20E-05	4.10E-05	0.000148	0.000642	0.00163	0.012647	0.118127	0.253288	2.08973	6.83575
1.10E-05	4.30E-05	0.000235	0.000802	0.002878	0.013625	0.075705	0.280478	1.26384	8.04637
1.30E-05	3.30E-05	0.00019	0.000637	0.002046	0.01635	0.038793	0.173169	2.56585	6.93253
1.20E-05	4.20E-05	0.000226	0.001159	0.002563	0.011407	0.101194	0.304477	1.95178	4.94376
6.00E-06	4.50E-05	0.000282	0.000927	0.003061	0.014643	0.072252	0.134293	1.58763	5.38412
7.00E-06	4.20E-05	0.000225	0.000864	0.003305	0.01084	0.066611	0.183326	1.2177	13.6168
Avg	Avg	Avg	Avg	Avg	Avg	Avg	Avg	Avg	Avg
9.90E-06	**3.77E-05**	**0.000199**	**0.000832**	**0.003007**	**0.01572**	**0.074086**	**0.24486**	**1.635634**	**7.75875**

Table 2: Improved Brute Force Timing Data

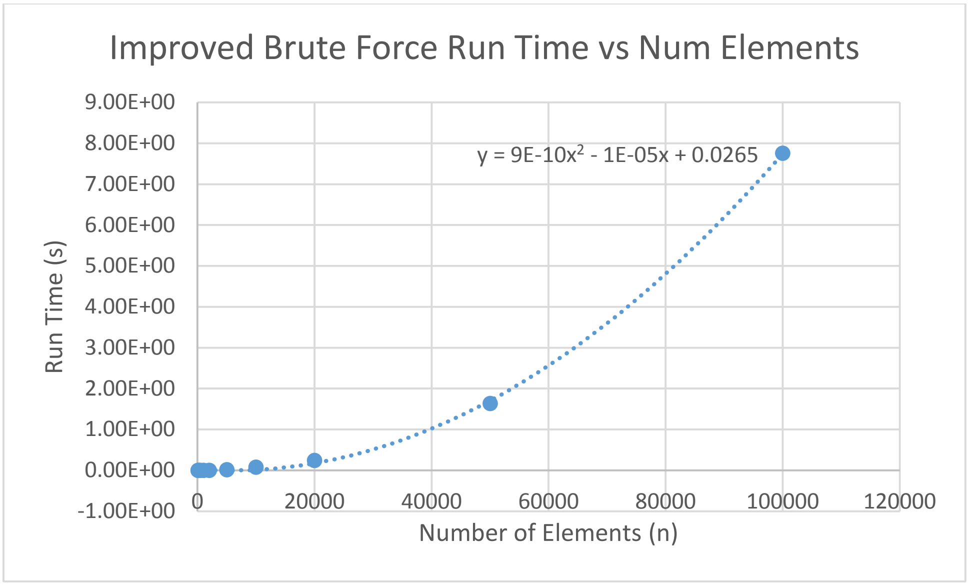

Figure 3: Improved Brute Force Run Time vs Number of Elements

Using regression equation $y = 9 * 10^{-10}x^2 - 1 * 10^{-5}x + 0.0265$

We find the number of elements that the algorithm can work for in 5, 10, and 60 seconds.

$$5 = 9 * 10^{-10}x^2 - 1 * 10^{-5}x + 0.0265 \rightarrow x = 80100.7 \tag{4}$$

$$\boldsymbol{\#\ elements = 80100}$$

$$10 = 9 * 10^{-10}x^2 - 1 * 10^{-5}x + 0.0265 \rightarrow x = 110972 \tag{5}$$

$$\boldsymbol{\#\ elements = 110972}$$

$$60 = 9 * 10^{-10}x^2 - 1 * 10^{-5}x + 0.0265 \rightarrow x = 263757 \tag{6}$$

$$\boldsymbol{\#\ elements = 263757}$$

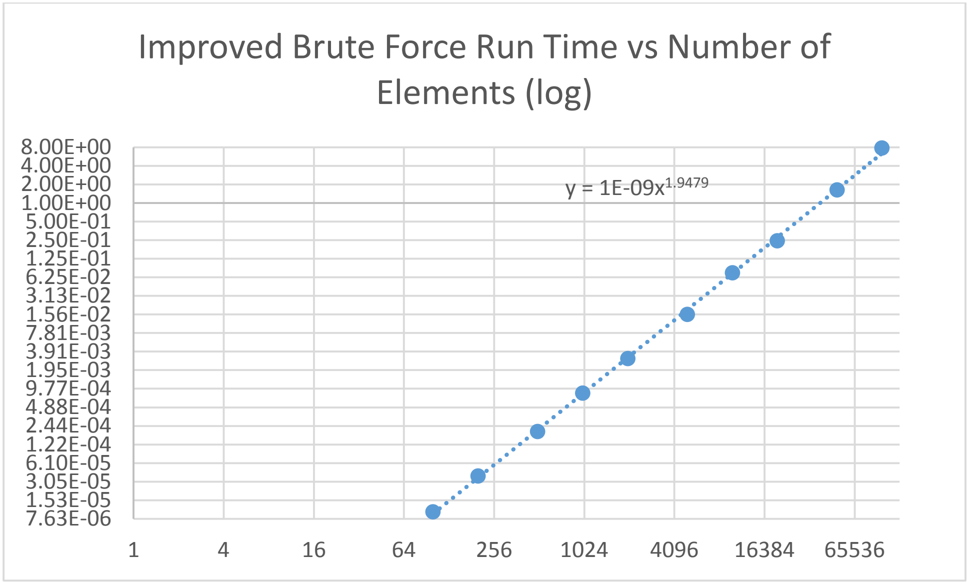

Figure 4: Improved Brute Force Run Time vs Number of Elements log log plot

C. Divide and Conquer

Divide and Conquer (3)									
n=1000	n=2000	n=5000	n=10000	n=20000	n=50000	n=100000	n=200000	n=500000	n=1000000
7.70E-05	0.000174	0.000444	0.000795	0.001736	0.005128	0.008957	0.02083	0.05074	0.108285
8.40E-05	0.000169	0.00041	0.000853	0.001693	0.004969	0.009459	0.021699	0.05016	0.105968
8.40E-05	0.000183	0.000382	0.000821	0.00194	0.0045	0.008918	0.022058	0.050968	0.104339
7.00E-05	0.000188	0.000396	0.000837	0.001907	0.005288	0.009003	0.021638	0.049365	0.101969
7.10E-05	0.000198	0.000405	0.000825	0.001909	0.004996	0.008989	0.020692	0.049536	0.10073
7.00E-05	0.000186	0.000381	0.000848	0.001737	0.00451	0.009206	0.020485	0.049613	0.101952
7.00E-05	0.000171	0.000384	0.000809	0.001632	0.005171	0.009348	0.019843	0.049926	0.10174
7.50E-05	0.000176	0.000383	0.000822	0.001731	0.004296	0.009241	0.018716	0.04927	0.100748
7.00E-05	0.000189	0.000384	0.000842	0.00169	0.005246	0.009179	0.019295	0.049569	0.100396
7.00E-05	0.000187	0.000418	0.000887	0.001627	0.005183	0.008907	0.01899	0.048761	0.103655
Avg	Avg	Avg	Avg	Avg	Avg	Avg	Avg	Avg	Avg
7.41E-05	**0.000182**	**0.000399**	**0.000834**	**0.00176**	**0.004929**	**0.009121**	**0.020425**	**0.049791**	**0.1029782**

Table 3: Divide and Conquer Timing Data

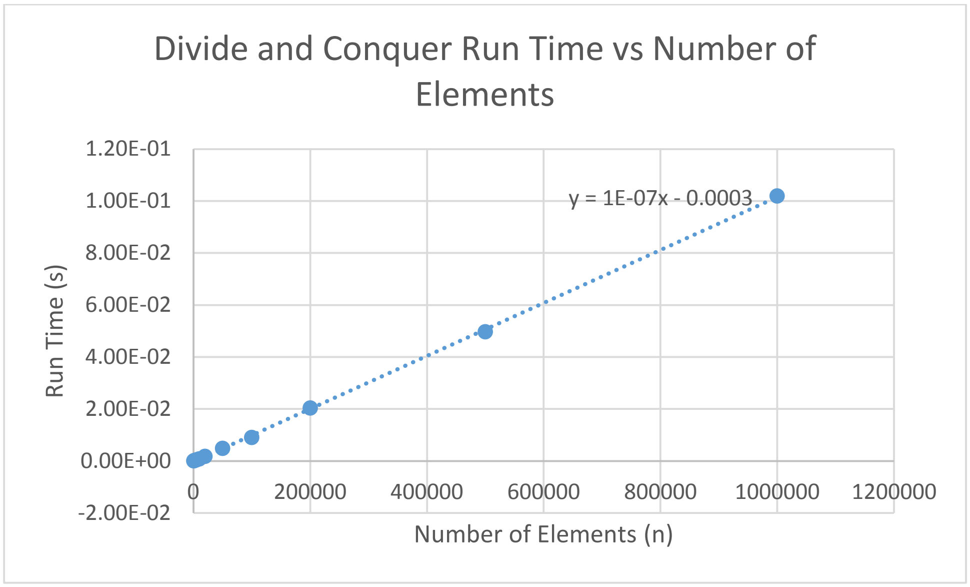

Figure 5: Divide and Conquer Run Time vs Number of Elements

Using regression equation $y = 1 * 10^{-7}x + 0.0003$

We find the number of elements that the algorithm can work for in 5, 10, and 60 seconds.

$$(7)\ 5 = 1 * 10^{-7}x + 0.0003 \rightarrow x = 49997000$$

$$\boldsymbol{\#\ elements = 49997000}$$

$$(8)\ 10 = 1 * 10^{-7}x + 0.0003 \rightarrow x = 99997000$$

$$\boldsymbol{\#\ elements = 99997000}$$

$$(9)\ 60 = 1 * 10^{-7}x + 0.0003 \rightarrow x = 599997000$$

$$\boldsymbol{\#\ elements = 599997000}$$

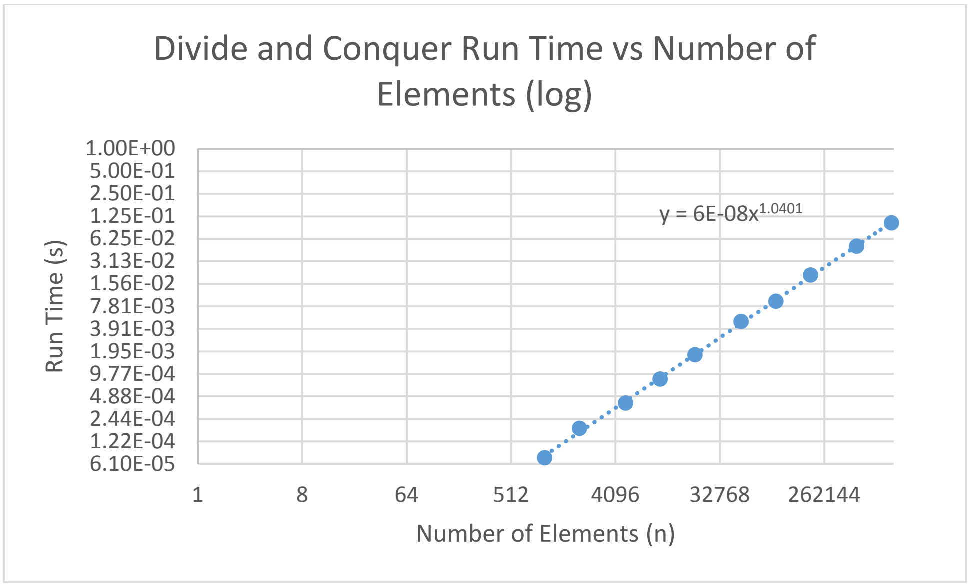

Figure 6: Divide and Conquer Run Time vs Number of Elements log log plot

D. Dynamic Programming

Dynamic Programming (4)									
n=1000	n=2000	n=5000	n=10000	n=20000	n=50000	n=100000	n=200000	n=500000	n=1000000
8.00E-06	1.30E-05	4.60E-05	7.20E-05	0.000123	0.000385	0.000659	0.001225	0.003392	0.006541
1.00E-05	1.60E-05	3.80E-05	7.70E-05	0.000123	0.000308	0.000697	0.001234	0.003182	0.006525
1.40E-05	2.10E-05	3.40E-05	6.20E-05	0.000163	0.000352	0.000613	0.001277	0.003115	0.006407
8.00E-06	1.50E-05	3.20E-05	6.30E-05	0.000164	0.000455	0.000613	0.001261	0.004152	0.006605
9.00E-06	1.30E-05	3.20E-05	6.20E-05	0.000133	0.000381	0.000613	0.00126	0.003939	0.007233
7.00E-06	1.40E-05	3.20E-05	6.20E-05	0.000135	0.000416	0.001192	0.001251	0.004309	0.007457
7.00E-06	1.40E-05	3.10E-05	6.30E-05	0.000122	0.000417	0.00076	0.001272	0.003161	0.006336
7.00E-06	1.20E-05	3.10E-05	6.30E-05	0.000123	0.000426	0.000705	0.001284	0.003342	0.007711
6.00E-06	1.20E-05	3.10E-05	7.40E-05	0.000171	0.0004	0.000756	0.001258	0.003105	0.006685
7.00E-06	1.20E-05	3.10E-05	6.10E-05	0.000122	0.000407	0.000801	0.00141	0.003262	0.00639
Avg	Avg	Avg	Avg	Avg	Avg	Avg	Avg	Avg	Avg
8.30E-06	**1.42E-05**	**3.38E-05**	**6.59E-05**	**1.38E-04**	**3.95E-04**	**7.41E-04**	**1.27E-03**	**3.50E-03**	**6.79E-03**

Table 4: Dynamic Programming Timing Data

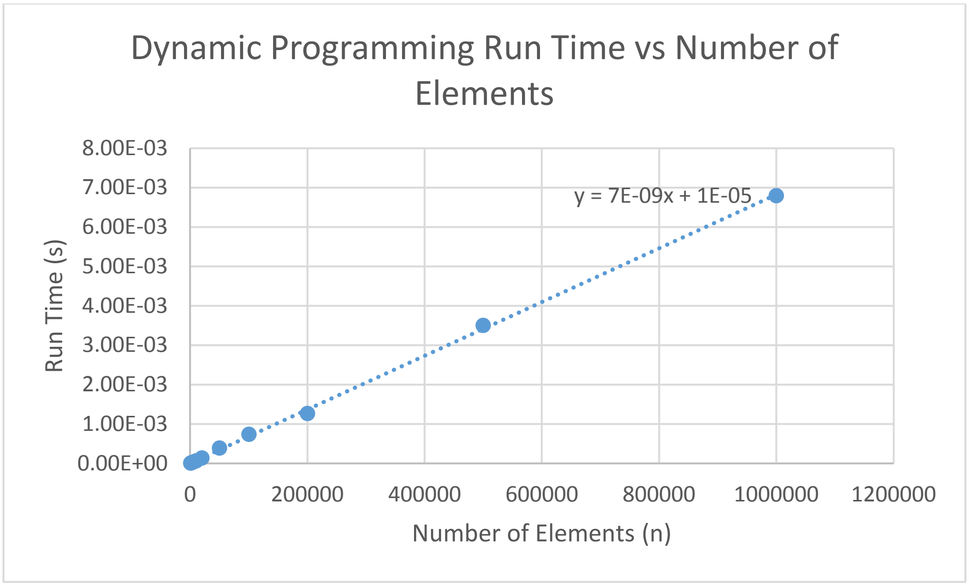

Figure 7: Dynamic Programming Run Time vs Number of Elements

Using regression equation $y = 7 * 10^{-9}x + 1 * 10^{-5}$

We find the number of elements that the algorithm can work for in 5, 10, and 60 seconds.

(10) $5 = 7 * 10^{-9}x + 1 * 10^{-5} \rightarrow x = 714284285.7$

$\boldsymbol{\# \, elements = 714284285}$

(11) $10 = 7 * 10^{-9}x + 1 * 10^{-5} \rightarrow x = 1428570000$

$\boldsymbol{\# \, elements = 1428570000}$

(12) $60 = 7 * 10^{-9}x + 1 * 10^{-5} \rightarrow x = 8571427143$

$\boldsymbol{\# \, elements = 8571427143}$

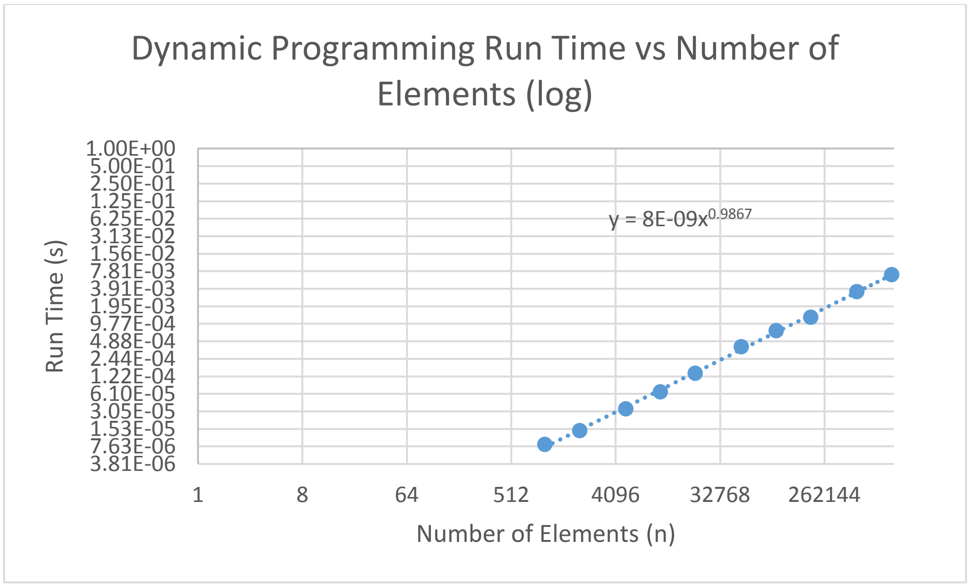

Figure 8: Dynamic Programming Run Time vs Number of Elements log log plot