Group 11
Charles Bennett
Conrad Lewin
Vlad Predovic

CS 325 Project 1: Maximum Sum Subarray

Theoretical Runtime Analysis

Algorithm 1 - Enumeration

Pseudocode:

```
Function enumerate_sum(array){
    For each number i in the array
        For each number j in the array
            Set the current sum = 0
            For each number k from i to j in the array
                Add number k to the current sum
            If the current sum is greater than the max sum found
                Max sum is now equal to current sum.
                Save the current i and j values as the array start/end
}
```

The algorithm operates with three nested loops. The first loop iterates through the entire array. The second loop iterates through the entire array. The final loop iterates from the current value of the first loop until the current value of the second loop. Therefore the third loop will at worst also iterate from when the starting point i is the beginning of the array until the ending point j is the last value in the array.
It is safe to assume that the run-time complexity of of this algorithm
$T(n) \leq n(n(n + c)$ where c is a constant. So $T(n) = O(n^3)$

Algorithm 2 Better Enumeration

Pseudocode:

```
Function betterEnumeration(array){
```

```
        Set max sum to 0
        for i = 0 to array.length
                Set start index and end index to i
                Set current sum to 0
                for j = i to in array.length
                        current sum += array[j]
                        if current sum > max sum
                                Set max sum to current sum
                                Set end index to  j

        return (array[start index...end index], max sum)
}
```

This algorithm uses nested for loops to iterate over the given array in order to find the maximum subarray. The outer for loop begins at index 0, setting the current max sum to 0 and assumes that the max subarray is an an array of size 1, namely that array whose sole element index is represented by the current value of *i*. This loop then uses the inner for loop to iterate over the remaining contents of the array, adding each value to the sum of the numbers that came before it and storing that value in the current sum variable. If that sum proves greater than the max sum, initially set to 0, then the max sum is updated to reflect the current sum and the end index is set to the index containing the value that caused the current sum to posses a value greater than the previous max sum. Essentially, this algorithm tests for the maximum subarray by iterating over each index in the array and calculating the sum of all elements that come after that that initial index. The outer loop must perform 2 constant time operations alongside running the inner for loop *n* times, while the inner loop must perform 4 constant time operations, including a comparison, *n* + *n* - 1 + *n* - 2 + … 2 + 1 times. Thus, the complexity of this algorithm can be stated as such:

$$T(n) \; = \; cn \sum_{i=1}^{n} i \; = \; cn(\tfrac{n+1}{2}) \; = \; \tfrac{cn^2 + cn}{2} \; = \; cn^2$$

We can prove this assertion using induction:

$$T(n) \; \leq \; cn^2$$
$$T(n \; - \; 1) \; \leq \; c(n-1)^2$$
$$T(n) \; \leq cn^2 \; - \; 2cn \; + \; c \; \leq \; cn^2$$

This inequality holds for all $c \; \geq \; 0$, thus, $T(n) \; = \; O(n^2)$.

Algorithm 3 - Divide and Conquer

Pseudocode:

```
Function maxCrossSum(array,start,middle,end){
    Set left sum to −∞
    Set current sum to 0
    Set start index and end index to Nil

    for i = middle down to in start
        Current sum += array[i]
        If current sum > left sum
            Set left sum to current sum
            Set start index to i

    Set right sum to −∞
    Set current sum to 0

    for j = middle + 1 to end
        Current sum += array[j]
        If current sum > right sum
            Set right sum to current sum
            Set end index to j

    return (start index, end index, left sum + right sum)
}

Function maxSubarray(array,start,end){
    If start == end
        return (start,end,array[start])

    Set middle to (start + end) / 2

    Set (start index, end index, left sum) to maxSubarray(array,start,middle)
    Set (start index, end index, right sum) to maxSubarray(array,middle + 1, end)
    Set (start index, end index, cross sum) to maxCrossSum(array,start,middle,end)

    If left sum >= right sum and left sum >= cross sum
        return (start index, end index, left sum)
    elif right sum >= left sum and right sum >= cross sum
        return (start index, end index, right sum)
    else
        return (start index, end index, cross sum)
}
```

The divide and conquer approach to the maximum subarray problem involves two functions, one linear in nature and the other recursive. The maxCrossSum function iteratively calculates the max sum of both the left and right sections of an array demarcated by a supplied middle index. Since this algorithm is meant to find the max sum that crosses the given middle index, the left sum is derived by stepping through the array from right to left, assuming that the middle index marks the end of the maximum subarray, in search of the subarray's start index. The right sum is calculated in the opposite fashion, assuming the element right after the middle is the start index and searching for the correct end index. Once both the maximum left and right subarrays are found, they are added together, thus forming the cross sum, and this new sum is returned along with the start and end index.

In order to furnish the crossSum function with input, we use the maxSubarray function to recursively divide the input array in half until an array of a just a single element is formed. The maxSubarray procedure calls the crossSum function on each of these smaller arrays which results in calculating the maximum subarray for all inputs of size 1 to *n*/2. When the recursion returns to the array of size *n*, it now knows the max subarray for both the left and right half of the array, and so needs only to call the crossSum function one last time to determine whether or not this cross sum is greater than either the left or the right sums. Once this is done, a final comparison is made to determine which of the three sums is greater and the resulting sum is returned along with the relevant start and end index.

The recurrence relation for this algorithm must then take into consideration the crossSum function, which performs various constant time operations *n* times using two separate for loops each performing *n*/2 iterations, and the array divisions performed by two different recursive calls to the maxSubarray procedure. Thus, the recurrence is:

$$T(n) = 2T(\frac{n}{2}) + cn$$

Using the master theorem, we can solve this recurrence relation and derive the run time complexity of the algorithm:

$$a = 2,\ b = 2,\ f(n) = cn$$
$$n^{log_2 2} = n = f(n) = \Theta(n\ log\ n)$$

Thus, the maxSubarray function is tightly bound by $\Theta(n\ log\ n)$.

Algorithm 4 - Linear Time

Pseudocode:

Function maxSubarrayLinear(arr){

```
        Set size to array length
        Set maxSum and endHereSum to −∞
        Set low and high to 0

        For i to size
            Set endHereHigh = i
            If endHereSum > 0
                Set endHereSum = endHereSum + arr[i]
            Else
                Set endHereLow = i
                Set endHereSum = arr[i]
            If endHereSum > maxSum
                Set maxSum = endHereSum
                Set low = endHereLow
                Set high = endHereHigh

        Return (low, high, maxSum)
    }
```

The linear function to find the max subarray starts at the left side of the array and progresses towards the right, storing the value of the max subarray sum. The subarray for any iteration is either empty (sum = zero) or has one more element than the max subarray at the previous position.

The variables low and high are used to show the bounds of a maximum subarray. The maxSum stores the sum of the values in the maximum subarray up to that point. The endHereLow and endHereHigh variables show a maximum subarray ending at index i. As i increments, endHereHigh is set to the new value for i. The endHereSum variable stores the sum of the values in a maximum subarray ending at i.

The for loop traverses the size of the array with an incrementing index i. The first if else statement in the for loop determines if the endHereSum has already been created and a subarray started, and if it has, it adds the current endHereSum to the value at arr[i]. If the subarray hasn't been started, the endHereLow is set to i and the endHereSum is set to arr[i], since arr[i] would be the only value in the new subarray.

The second if statement in the for loop checks to see if the endHereSum is greater than (not equal to) the current maxSum, and if it is, the maxSum is set to the endHereSum. (Note that there can be multiple maximum subarrays with a maxSum.) The low variable is set to endHereLow and the high variable is set to endHereHigh. The loop iterates through the entire array then returns the low and high values of the subarray, as well as the maxSum.

The expected runtime is O(n), because the for loop must iterate through the entire length of the array a single time.

Testing Description:

The initial test of each algorithm was conducted using an instructor provided file containing several arrays of both positive and negative whole numbers. The content of this file was used as input for each algorithm and the resulting output was then written to a file and compared with a set of solutions provided to us by the instructor. After the initial tests had concluded and the output of each algorithm proved correct, we used a test script written by Conrad Lewin to create several arrays of varying length and random numbers to further test the robustness of the algorithms. The results were confirmed as correct by manually adding up the longest subarray in each of the randomly generated arrays and checking that the results matched.

Experimental Analysis

Algorithm 1

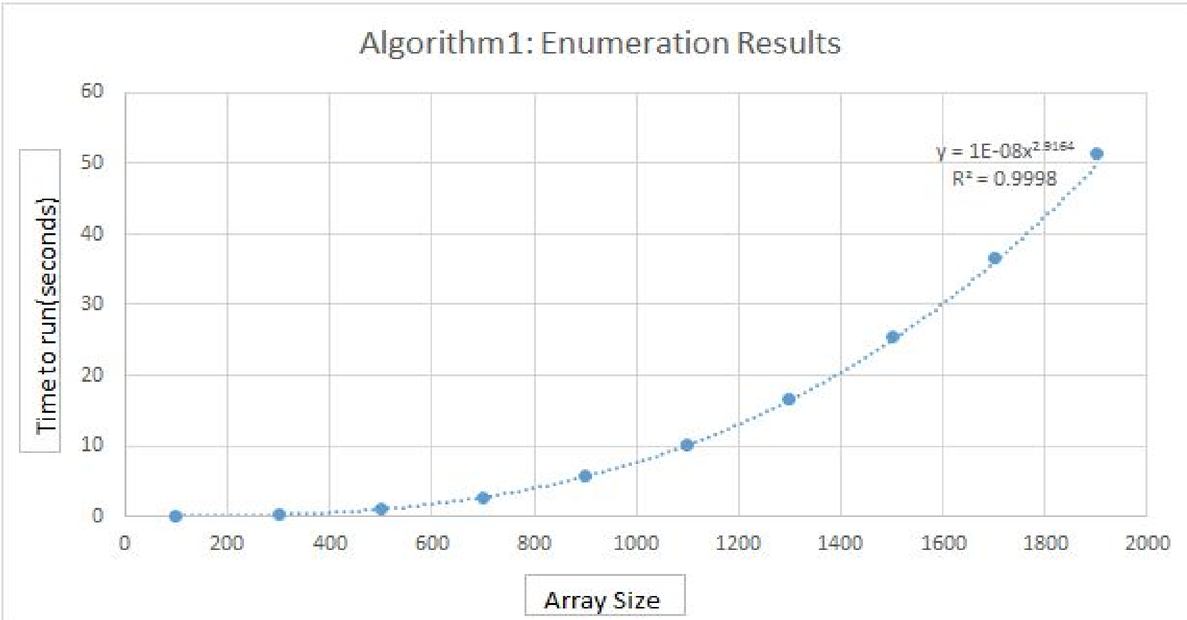
1. The figure below is a view of all test case results and averages.

Array Size	Run 1(sec)	Run 2(sec)	Run 3(sec)	Run 4(sec)	Run 5(sec)	Run 6(sec)	Run 7(sec)	Run 8(sec)	Run 9(sec)	Run 10(sec)	Run Avg(sec)
100	0.011999846	0.012000084	0.00999999	0.007999897	0.008000135	0.012000084	0.008000135	0.012000084	0.008000135	0.012000084	0.01000007
300	0.214999914	0.207999945	0.214999914	0.215999842	0.21600008	0.220000029	0.211999893	0.211999893	0.211999893	0.215999842	0.214111037
500	0.967000008	0.947999954	0.981999874	0.953000069	0.960000038	0.951999903	0.960000038	0.947999954	0.955999851	0.951999903	0.956777732
700	2.56799984	2.668000221	2.701000214	2.599999905	2.623999834	2.596999884	2.579999924	2.65199995	2.623000145	2.628000021	2.630333344
900	5.43599987	5.471999884	5.799000025	5.586999893	5.579999924	5.548000097	5.792000055	5.644000053	5.633000135	5.57799983	5.625888877
1100	10.08800006	10.12899995	9.992999792	10.13700008	10.25	10.28499985	10.22199988	10.20300007	10.2420001	10.16300011	10.18044443
1300	16.56299996	16.86399984	16.32400012	16.57599998	16.42499995	16.70500016	16.54900002	16.51699996	16.60400009	16.64100003	16.57833335
1500	25.11199999	25.44999981	25.2900002	25.22099996	25.79500008	25.40999985	25.65300012	25.59399986	25.54500008	25.38199997	25.48222221
1700	36.421	36.86699986	36.36199999	36.78700018	36.50399995	36.84700012	36.602	36.38800001	36.55900002	36.92400002	36.64888891
1900	51.61199999	51.02200007	52.18200016	50.44499993	51.02700019	51.91499996	51.87400007	51.3670001	51.7980001	50.66299987	51.36588894

2. Results for running the enumeration algorithm can be found graphed below. Using Microsoft Excel, the power trendline seemed the best fit with a R^2 value of 0.9998.

The resulting equation was $y = c*x^{2.9164}$ where c is a constant.

Array Size	Run Avg(sec)
100	0.01000007
300	0.214111037
500	0.956777732
700	2.630333344
900	5.625888877
1100	10.18044443
1300	16.57833335
1500	25.48222221
1700	36.64888891
1900	51.36588894



3. A regression fit performed using wolfram alpha, a Ti-84 calculator and MS excel provided the following model.

$$y = 7.319037 \times 10^{-9}\, x^3 + 2.6741543 \times 10^{-7}\, x^2 + 0.000075836365\, x - 0.0270937092$$

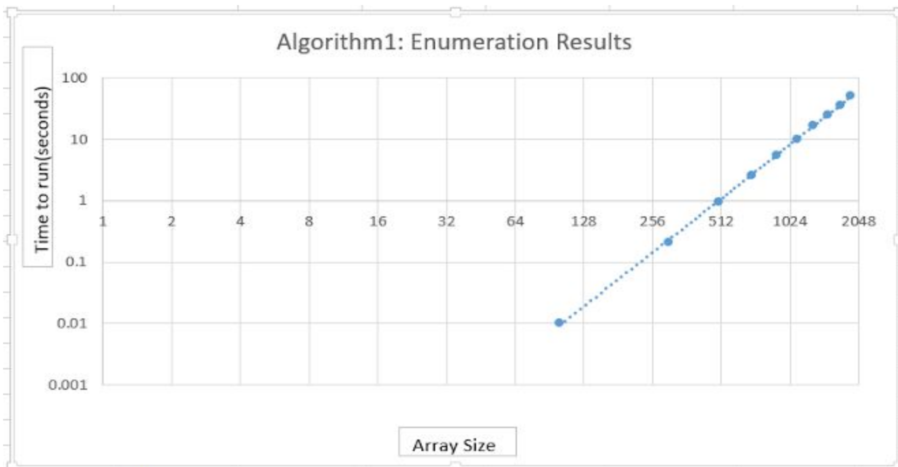
4. The equation given by performing regression analysis in excel is very close to the theoretical results. Slight differences can be a result of limitations by the equipment used as well as how the laptop itself decided to compute results (multithreading, process use etc.) Since algorithm 1 was tested on Vlad’s computer, a comparison between his laptop and flip was done to see if the use of a different CPU would throw off the results regarding time complexity analysis. Below you can see that the results were similar in terms of trends. On the left is the original run on Vlad’s computer and on the right are the results from using OSU’s servers. The OSU results are close to double that of Vlad’s computer but they increase at the same rate, confirming that the data is consistent.

5. Solving for the amount of entries (x) given time (y):
Y = 10 seconds, 30 seconds, and 60 seconds

$$x = 1095.52 \quad x = 1586.63 \quad x = 2002.84$$

Ten seconds gives a maximum of **1095 entries**.
Thirty seconds gives a maximum of **1586 entries.**
One minute gives a maximum of **2002 entries.**

6. Plotting the data from excel on a loglog plot yields a linear trend as shown in the graph to the right.



Therefore the equation is given by log(y) = m log(x) + b

m = slope of line = $\frac{\Delta(log\,y)}{\Delta(log\,x)} = \frac{\Delta(log\,51.366 - log 0.967)}{\Delta(log\,1900 - log\,500)}$ = 2.9836

Given some room for error, it is safe to assume that the function F(n) = O($n^{2.9836}$) which is very close to the O(n^3) found through the theoretical analysis.

Algorithm 2

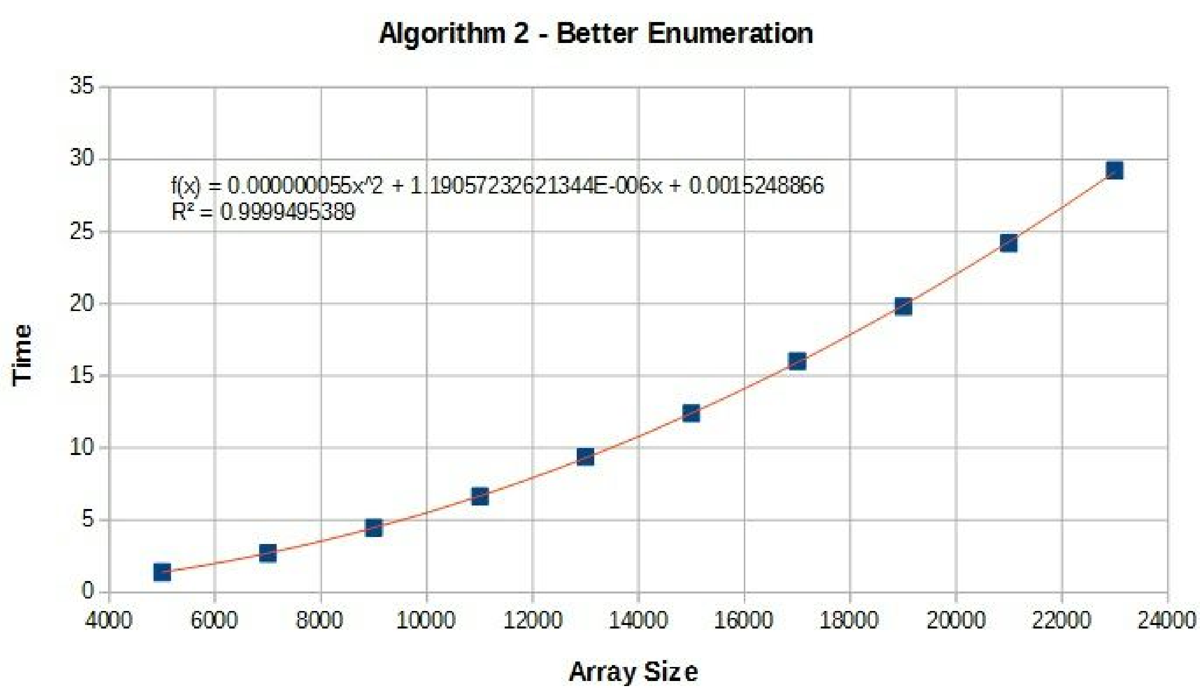
1. Individual and average running times:

Array Size	Run 1 (sec)	Run 2 (sec)	Run 3 (sec)	Run 4 (sec)	Run 5 (sec)	Run 6 (sec)	Run 7 (sec)	Run 8 (sec)	Run 9 (sec)	Run 10 (sec)	Average (sec)
5000	1.3929941654	1.3899400234	1.3910319805	1.3864021301	1.3970839977	1.396723032	1.3924520016	1.3952748775	1.379308939	1.3741981983	1.3895409346
7000	2.6986169815	2.6936810017	2.6922500134	2.6963460445	2.6926431656	2.6996750832	2.6929121017	2.6943440437	2.6940660477	2.6978800297	2.6952414513
9000	4.476020813	4.4520189762	4.4524509907	4.4526400566	4.4534029961	4.4798879623	4.4582679272	4.4519519806	4.4546399117	4.4521529675	4.4583434582
11000	6.6482319832	6.6467318535	6.6403160095	6.6474330425	6.6454310417	6.6284189224	6.6368401051	6.6370830536	6.6426520348	6.6372919083	6.6410429955
13000	9.2598469257	9.7600328922	10.0126049519	9.2633531094	9.267521143	9.2656290531	9.2679281235	9.2714118958	9.2627658844	9.2663741112	9.389746809
15000	12.3560841084	12.3492090702	12.8775539398	12.3411121368	12.3598499298	12.3589839935	12.3463602066	12.3411538601	12.3523011208	12.346506834	12.40291152
17000	17.386526823	15.8549280167	15.8677370548	15.8590040207	15.8473780155	15.8525390625	15.8324408531	15.8705530167	15.9027140141	15.8647749424	16.0138595819
19000	19.8308520317	19.8429028988	19.7891449928	19.8215711117	19.7781951427	19.8154699802	19.7966239452	19.8286750317	19.7765789032	19.8173861504	19.8097400188
21000	24.1935989857	24.2477641106	24.1685240269	24.163381815	24.1937189102	24.2069888115	24.1890392303	24.1683590412	24.1696250439	24.2066261768	24.1907626152
23000	29.0035488606	28.9990119934	29.0183131695	29.0103070736	31.3321158886	28.9828310013	29.0193691254	28.9856059551	29.0075950623	29.0050909519	29.2363789082

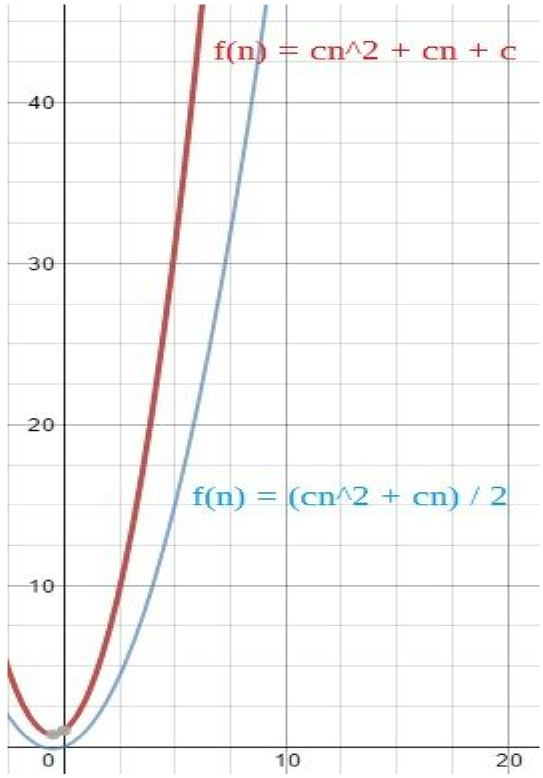
2. Achieving an $R^2 = 0.9999495389$, a quadratic trendline proved the best fit for this data set. The equation of the trendline is calculated as $f(n) = cn^2 + cn + c$. See the complete plot in part 3 below.

Array Size	Average (sec)
5000	1.3895409346
7000	2.6952414513
9000	4.4583434582
11000	6.6410429955
13000	9.389746809
15000	12.40291152
17000	16.0138595819
19000	19.8097400188
21000	24.1907626152
23000	29.2363789082

3. Regression model of average runtimes using LibreOffice Calc:



4. There are no major discrepancies between the theoretical and experimental analyses insofar as both examinations are bound by a quadratic runtime complexity and any difference between the two is dependent on constant factors. The graph shown below illustrates this fact by showing that both functions achieve an end behavior equivalent to a quadratic function.



5. Using the formula derived from the regression model in parts 2 and 3

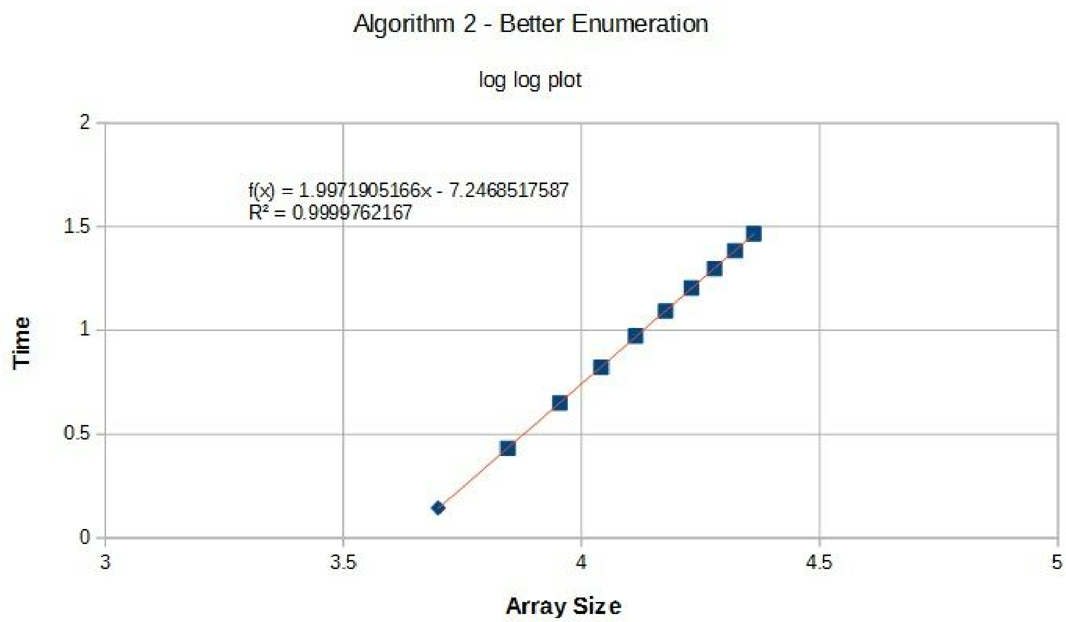
$$y = 0.000000055x^2 + (1.19057232621344 \times 10^{-6})x + 0.0015248866$$

We can solve the largest input size the algorithm can solve in 10, 30 and 60 seconds by substituting *y* with the appropriate time value and then solving for *x*. Doing so returns the following results:

Time (seconds)	**Largest Solvable Input Size**
10	13472
30	23343
60	33017

6. Log-log plot of algorithm 2 using LibreOffice Calc:

log Array Size	log Average
3.6989700043	0.1428713452
3.84509804	0.4305976772
3.9542425094	0.6491735223
4.0413926852	0.822236292
4.1139433523	0.9726538818
4.1760912591	1.0935236455
4.2304489214	1.2044960161
4.278753601	1.2968787759
4.3222192947	1.3836495597
4.361727836	1.4659235818



The linear function that best fits this data is $f(n) = 1.9971905166x - 7.2468517587$. The slope of this linear equation is approximately 1.997 which suggests the runtime complexity for this algorithm is $O(n^{1.997})$, illustrating a mere 0.1 difference in power values between the theoretical and experimental analysis.

Algorithm 3

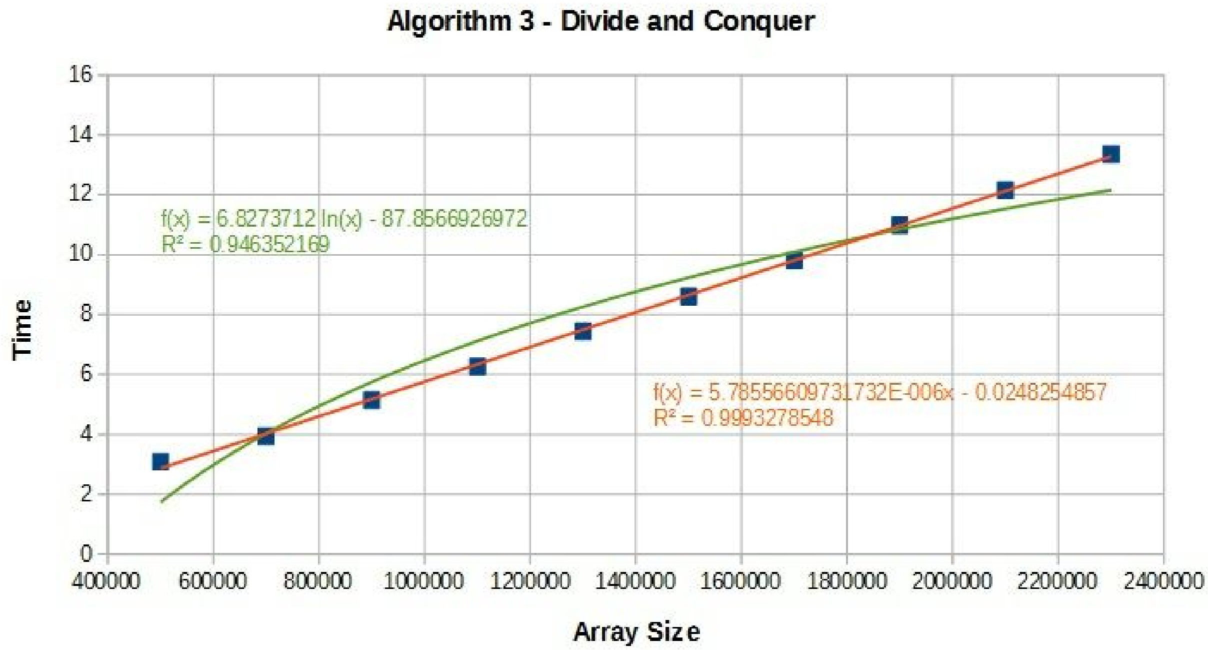
1. Individual and average running times:

Array Size	Run 1 (sec)	Run 2 (sec)	Run 3 (sec)	Run 4 (sec)	Run 5 (sec)	Run 6 (sec)	Run 7 (sec)	Run 8 (sec)	Run 9 (sec)	Run 10 (sec)	Average (sec)
500000	4.6784431934	3.8996729851	2.7852699757	2.7809801102	2.7870881557	2.7823030949	2.7770810127	2.7824909687	2.7799098492	2.7777559757	3.0830995321
700000	3.9288208485	3.9332950115	3.9318819046	3.9334750175	3.9472651482	3.9534039497	3.9347319603	3.941824913	3.9340200424	3.9332039356	3.9371922731
900000	5.101446867	5.1042690277	5.0963149071	5.0947768688	5.0882890224	5.0907869339	5.1108989716	5.1468920708	5.5074400902	5.0891120434	5.1430226803
1100000	6.2430028915	6.2611539364	6.248980999	6.2618720531	6.2428321838	6.2495989799	6.2568190098	6.2526569366	6.3051388264	6.2648777962	6.2586933613
1300000	7.4508469105	7.4354219437	7.4352350235	7.4288539886	7.4453492165	7.4279921055	7.4300870895	7.4352068901	7.4394540787	7.4334058762	7.4361853123
1500000	8.6039631367	8.6138861179	8.6030650139	8.6035020351	8.6018550396	8.6041820049	8.6010110378	8.6028110981	8.6025080681	8.6001899242	8.6036973476
1700000	9.7889959812	9.8030591011	9.7928919792	9.8342130184	9.7974901199	9.7982850075	9.7969279289	9.7936460972	9.7898979187	9.793517828	9.798892498
1900000	10.9990451336	10.9977688789	10.9755449295	10.9731080532	10.9811389446	10.9756979942	10.9769649506	10.9874589443	10.9842541218	10.9929139614	10.9843895912
2100000	12.1806788445	12.1305530071	12.1447038651	12.1187438965	12.1418018341	12.1399049759	12.1903760433	12.1326479912	12.1885049343	12.1495959759	12.1517511368
2300000	13.3340139389	13.347276926	13.3590779305	13.3549368382	13.3544669151	13.3446369171	13.3603150845	13.3696131706	13.3456790447	13.3574509621	13.3527467728

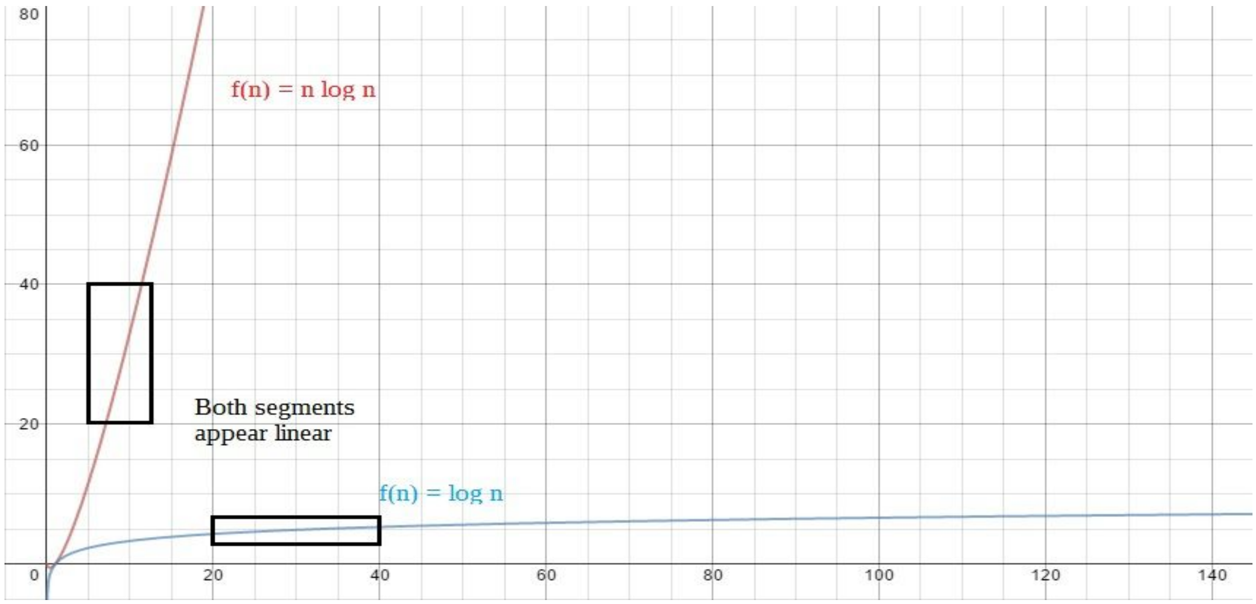
2. While a logarithmic trendline achieves an R^2 = 0.946352169, a linear trendline reaches an R^2 = 0.9993278548 and appears to be the best fit for this data set. The equation of this trendline is calculated as $f(n) = cn - c$. See the complete plot in part 3 below.

Array Size	Average (sec)
500000	3.0830995321
700000	3.9371922731
900000	5.1430226803
1100000	6.2586933613
1300000	7.4361853123
1500000	8.6036973476
1700000	9.798892498
1900000	10.9843895912
2100000	12.1517511368
2300000	13.3527467728

3. Regression model of average runtimes using LibreOffice Calc:



4. The theoretical analysis asserts that this algorithm should be asymptotically bounded by a modified logarithmic function of the order $\Theta(n \log n)$. The experimental analysis, however, seems to suggest that the algorithm is bounded by a linear function. Given the rather limited domain of the data set, it is likely that the linear facade presented by this analysis is due to the fact that only a small portion of the logarithmic function is being examined. Looking at the graph below, we can see that if the domain of the given data set is small enough, then the true nature of the graph cannot adequately be understood.



We can prove the error in the experimental analysis further using induction to manifest a contradiction. Thus, if we assume that $T(n) \leq O(n)$, then our hypothesis is that $T(n-1) \leq 2(\frac{n-1}{2}) + c(n-1) \leq cn$. The inductive step is then:

$$T(n) \leq 2(\tfrac{n-1}{2}) + c(n-1)$$
$$T(n) \leq n - 1 + cn - c$$
$$T(n) \leq cn - c + n - 1 \leq cn$$

This final inequality does not hold for all $c \geq 0$ and all $n \geq 2$ and so, by definition, $T(n) \neq O(n)$.

5. The formula derived from the regression model in parts 2 and 3 that most closely models the domain of the given data set is

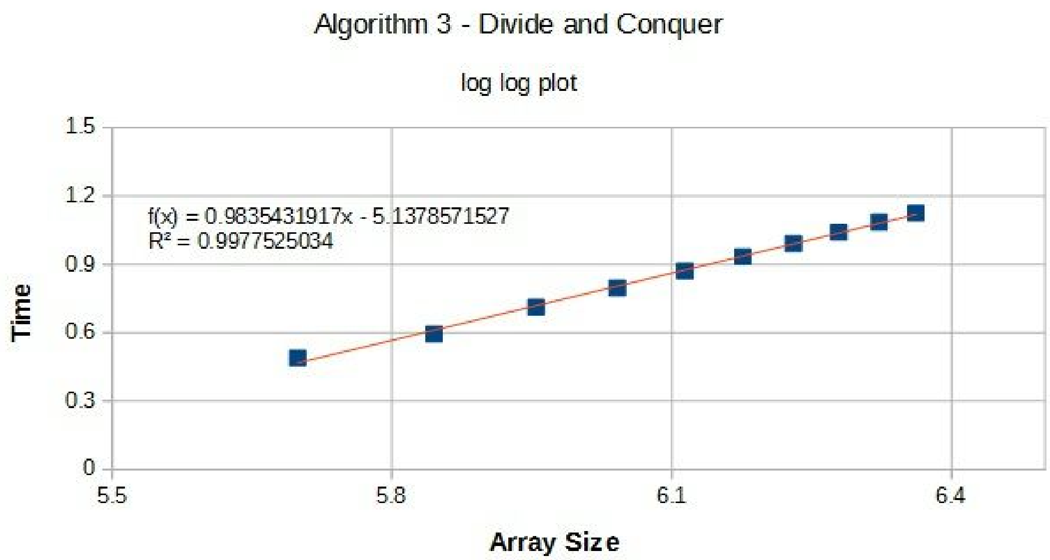
$$y = (5.785566099731732 \times 10^{-6})x - 0.0248254857$$

We can solve the largest input size the algorithm can solve in 10, 30 and 60 seconds by substituting *y* with the appropriate time value and then solving for *x*. Doing so returns the following results:

Time (seconds)	Largest Solvable Input Size
10	1732730
30	5189610
60	10374900

6. Log-log plot of algorithm 3 using LibreOffice Calc:

log Size	log Average
5.6989700043	0.4889875453
5.84509804	0.5951866241
5.9542425094	0.7112184395
6.0413926852	0.7964836742
6.1139433523	0.871350204
6.1760912591	0.9346851248
6.2304489214	0.9911769931
6.278753601	1.0407759279
6.3222192947	1.0846388668
6.361727836	1.1255706129



The linear function that best fits this data is $f(n) = 0.9835431917x - 5.1378571527$. The slope of this linear equation is approximately 0.98, however, due to rounding that occurs during calculation, it is safe to say that the slope is approximately 1. This suggests the runtime complexity for this algorithm is bound by $\Theta(n\ log\ n)^1 = \Theta(n\ log\ n)$, showing that theoretical and experimental analysis are roughly equivalent.

Algorithm 4

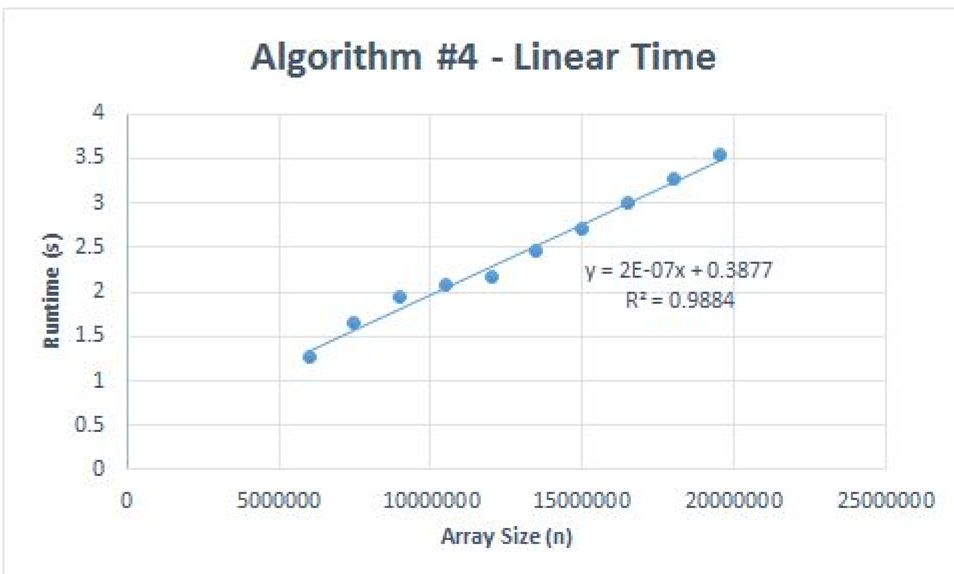
1. Individual and average running times:

Array Size	Run 1 (sec)	Run 2 (sec)	Run 3 (sec)	Run 4 (sec)	Run 5 (sec)	Run 6 (sec)	Run 7 (sec)	Run 8 (sec)	Run 9 (sec)	Run 10 (sec)	Average (sec)
6000000	1.177865982	1.183593988	1.351732969	1.332457066	1.289621115	1.229223013	1.273873806	1.253334045	1.268435001	1.23251915	1.259265614
7500000	1.53270483	1.567991018	1.843770981	1.608741999	1.653439999	1.661116123	1.69471693	1.630695105	1.655829906	1.670513868	1.651952076
9000000	1.957348824	1.900403976	1.957577944	1.969618082	1.905730009	1.969887972	1.879898071	1.976902962	1.963747025	1.967099905	1.944821477
10500000	2.310198784	2.212309122	2.281311989	2.206243992	2.179488897	1.959072113	1.945199013	1.926848888	1.934551001	1.907421112	2.086264491
12000000	2.17937398	2.186386108	2.198943853	2.18181181	2.168345928	2.172370911	2.178223133	2.190314054	2.17713213	2.167604923	2.180050683
13500000	2.441015959	2.453561783	2.51828289	2.458393812	2.441614866	2.429917097	2.444508076	2.447766066	2.452739	2.449769974	2.453756952
15000000	2.71230793	2.707540035	2.716903925	2.723256111	2.72982502	2.717978001	2.695761204	2.714673996	2.725910902	2.738967896	2.718312502
16500000	2.986824989	2.983185053	2.983592033	2.991980076	3.010128975	2.993162155	2.983683109	2.986974001	2.990894079	3.012086868	2.992251134
18000000	3.276855946	3.250483036	3.260619164	3.28310895	3.263273001	3.265541077	3.254791975	3.275377989	3.269223928	3.27045989	3.266973495
19500000	3.527770996	3.555449963	3.548099995	3.529908895	3.528563023	3.558630943	3.548257828	3.518638134	3.531308174	3.560202122	3.540683007

2. Achieving an $R^2 = 0.9884$, a linear trendline proved the best fit for this data set. The equation of the trendline is calculated as $f(n) = cn + c$. See the complete plot in part 3 below.

Array Size	Average (sec)
6000000	1.259265614
7500000	1.651952076
9000000	1.944821477
10500000	2.086264491
12000000	2.180050683
13500000	2.453756952
15000000	2.718312502
16500000	2.992251134
18000000	3.266973495
19500000	3.540683007

3. Regression model of average runtimes using Microsoft Excel:



4. There were no major discrepancies between the experimental and theoretical running times. Runtime increases in direct proportion to the array size, reinforcing that the linear approach has an O(n) runtime.

5. The linear function from the plot is:

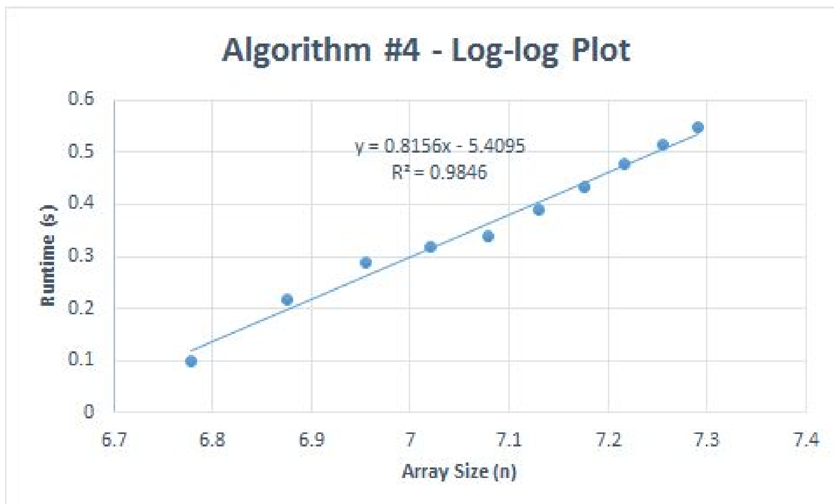
$$y = 2 \times 10^{-7}x + 0.3877$$

We can substitute the values 10 seconds, 30 seconds, and 60 seconds in the equation for the value of y and then solve for x in order to find the largest input that can be solved in the associated amount of time.

Time (seconds)	Largest Solvable Input Size
10	48061500
30	148061500
60	298061500

6. Taking the log of the array sizes and the run time averages gives us a table we can use to make the log-log plot:

Log Array Size	Log Average (sec)
6.77815125	0.100117344
6.875061263	0.217997444
6.954242509	0.288879742
7.021189299	0.319369366
7.079181246	0.33846659
7.130333768	0.389831543
7.176091259	0.434299383
7.217483944	0.47599804
7.255272505	0.514145611
7.290034611	0.549087047



In the log-log plot, the runtime increases at a linear rate as array size increases. This is expected, and fits the O(n) runtime observed in the experimental and theoretical data. The equation for the linear progression is $y = 0.8156x - 5.4095$. The slope (m) of the line is 0.8156, which rounds up to 1. The runtime is $O(x^m)$ which is $O(x^{0.8156})$ which is essentially O(n).

Log log plot of all four algorithms