Aryan Aziz
Bryon Burleigh

CS325 - Project 1

Run Time Analysis

Algorithm 1:

```
void function(int arr[], int size) {
        for(int i = 0; i < size; i++) {
                for(int j = i; j < size; j++) {
                        add up all of the values from i to j in the array
                        If the sum from above is closer to zero or equal to zero
                                set the best sum as this value and hold the i/j coordinates
                }
        }
}
```

In our function, we have 3 different loops being traversed here, one going down the line for each value of i, one going down the line for each value of j, and one going down to add all of the values together. Because of this, our function for algorithm 1 would have a BigO runtime of O(n^3).

Algorithm 2:

```
void function(int arr[], int size) {
        for(int i = 0; i < size; i++) {
                for(int j = i; j < size; j++) {
                        Set the current sum equal to the initial value at arr[i]
                        Add each value of arr[j] to the end of it
                        If the sum from above is closer to zero or equal to zero
                                set the best sum as this value and hold the i/j coordinates
                }
        }
}
```

In algorithm 2, we only have 2 loops being traversed and adding an extra value at the end. One loop going down the value of i and one going down the value of j, giving us a BigO runtime of O(n^2).

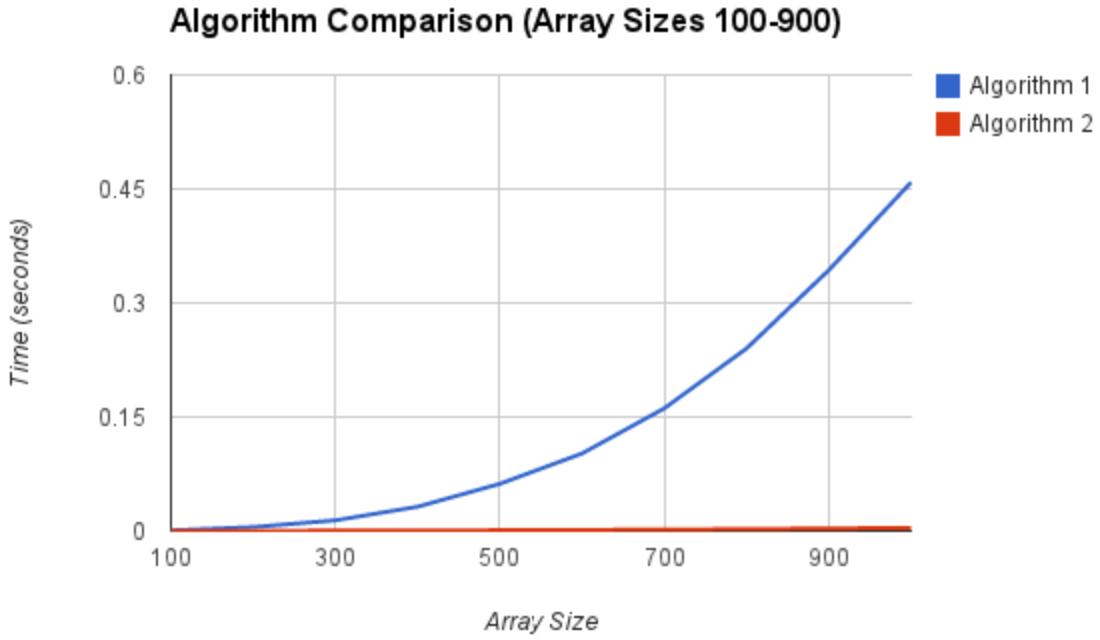
Testing

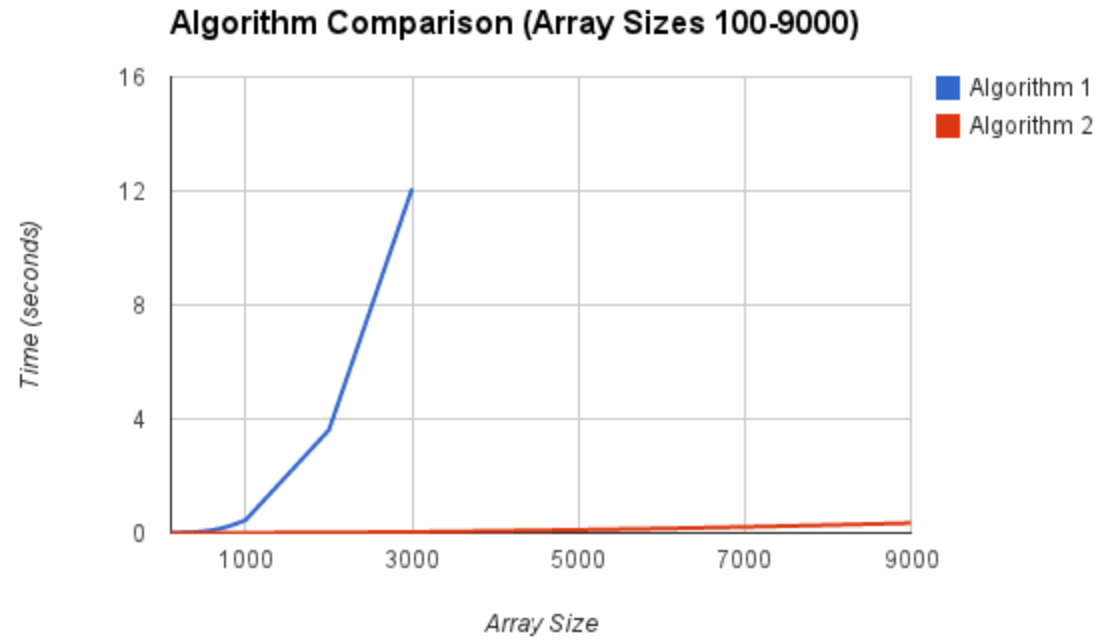
Please see the CtZ_Results.txt file for our results.

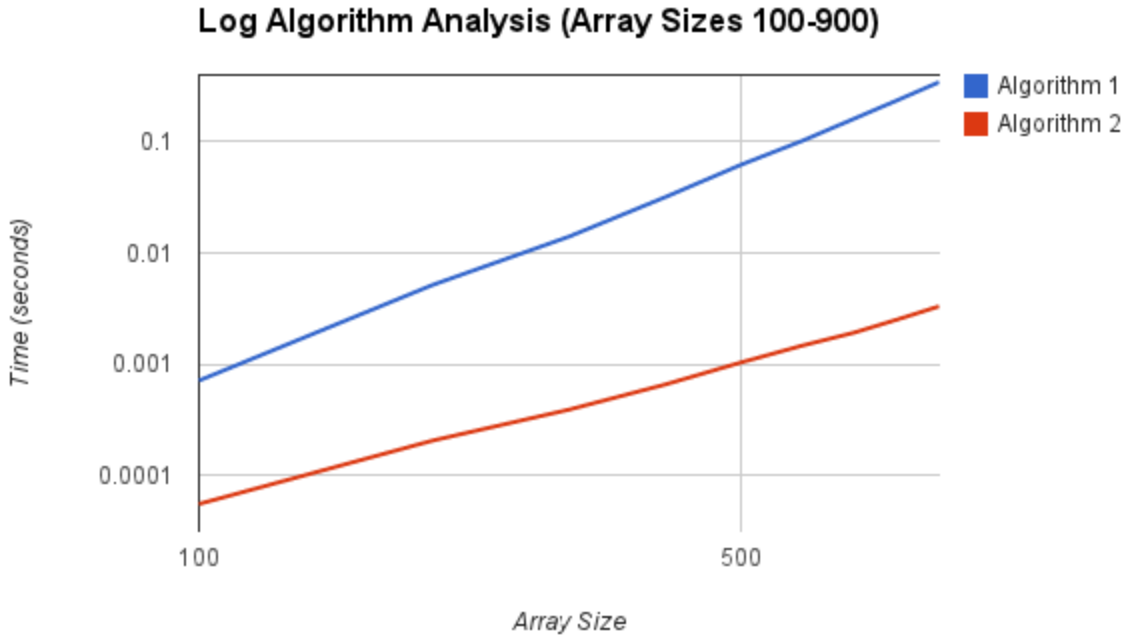
Experimental Analysis

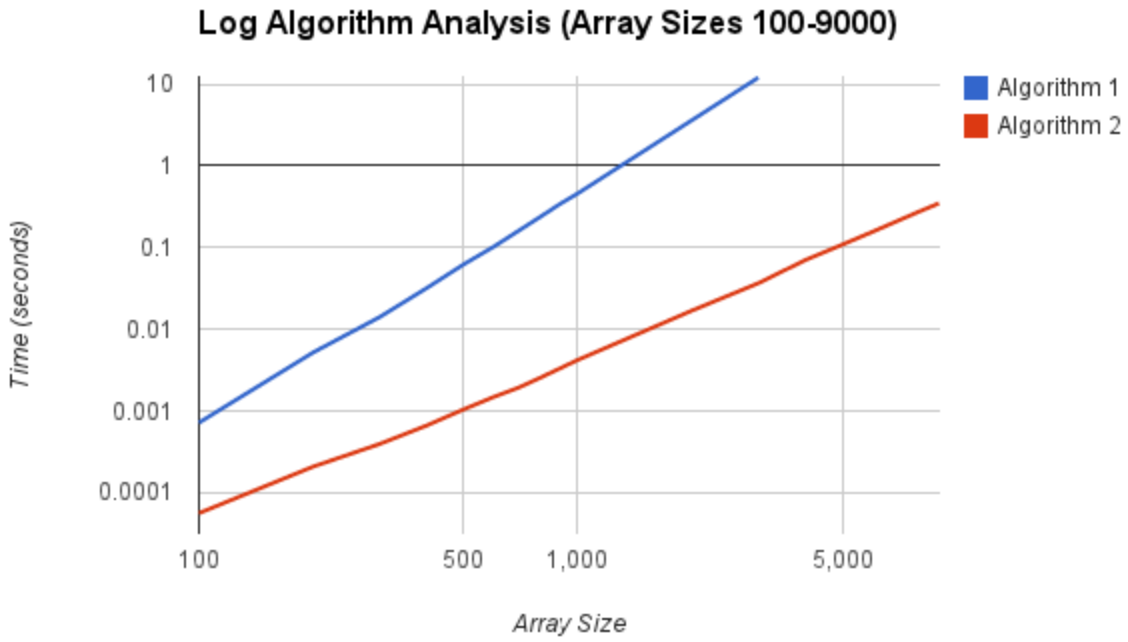
Below are our runtime results. All numbers are in seconds and are an average of 10 different data-set runs with the same array size.

Array Size	**Algorithm 1**	**Algorithm 2**
100	0.0007117	0.0000553
200	0.005148	0.0002053
300	0.0140183	0.0003899
400	0.031973	0.0006585
500	0.0620478	0.0010359
600	0.1019908	0.0014697
700	0.1613566	0.0019225
800	0.2401529	0.002552
900	0.3431394	0.0033096
1000	0.45863	0.0041738
2000	3.613709	0.016775
3000	12.08286	0.036476
4000	--	0.070339
5000	--	0.1083757
6000	--	0.1551861
7000	--	0.2118456
8000	--	0.2770747
9000	--	0.3475754


Algorithm Comparison (Array Sizes 100-900)
0.6
0.45
0.3
0.15
0
Time (seconds)
100
300
500
700
900
Array Size
Algorithm 1
Algorithm 2


Algorithm Comparison (Array Sizes 100-9000)
16
12
8
4
0
Time (seconds)
1000
3000
5000
7000
9000
Array Size
Algorithm 1
Algorithm 2


Log Algorithm Analysis (Array Sizes 100-900)
Algorithm 1
Algorithm 2
0.1
0.01
0.001
0.0001
Time (seconds)
100
500
Array Size


Log Algorithm Analysis (Array Sizes 100-9000)
Algorithm 1
Algorithm 2
10
1
0.1
0.01
0.001
0.0001
Time (seconds)
100
500
1,000
5,000
Array Size

Extrapolation And Interpretation

1. What is the biggest array size you could solve for in an hour? (Please note, for both answers I rounded up to the nearest “nice” number, meaning they are not the values at exactly 3600 seconds)
    a. Algorithm 1: Using the data from array sizes of 100-900 (ignoring the 2 extra values) and using the line of best fit slope, this algorithm could theoretically parse through an array of ~8,500,000 values in roughly 3600 seconds (1 hour).
    b. Algorithm 2: By extrapolating the data, based on the line of best fit slope, this algorithm could theoretically parse through an array of ~65,000,000 values within 3600 seconds (1 hour).

2. Slope (with help from http://mathonweb.com/help_ebook/html/expoapps_3.htm)
    a. Algorithm 1:
    y=mx+b

    Point 1: 0.0007117 = m(100) + b
    Point 2: 0.3431394 = m (900) +b

    Subtract 1st point from 2nd point:
    0.3424277 = 800m
    m= 0.33424277/800 = 0.000428035

    Algorithm 1 slope = 0.000428035
    b = −0.0420918

    b. Algorithm 2:
    y=mx+b

    Point 1: 0.0000553 = m(100) + b
    Point 2: 0.0033096 = m (900) +b

    Subtract 1st point from 2nd point:
    0.0032543 = 800m
    m= 0.0032543/800 = 0.000004068

    Algorithm 2 slope = 0.000004068
    b = −0.0003515