Group 13
Bryon Burleigh
Jennifer Erland
Jilian LaFerte

Theoretical Run-time Analysis

Algorithm 1

<u>Pseudocode</u>

```
for i=first to last
        for j=i to last
                tempSum = sum (array[i] through array[j])
                if tempSum>maxSum
                        maxSum = tempSum
                        store i,j
```

<u>Analysis</u>

n = 1 : 0 operations, 1 comparison = 1 total
n = 2 : 1 operation, 3 comparisons = 4 total
n = 3 : 4 operations, 6 comparisons = 10 total
n = 4 : 10 operations, 10 comparisons = 20
n = 5 : 20 operations, 15 comparisons = 35
and so on. When graphed, this pattern closely lines up with n! Thus the runtime is O(n!)

Algorithm 2

<u>Pseudocode</u>

```
For i=first to last
        tempSum = array[i]
        for j=i+1 to last
                tempSum = tempSum + array[j]
                if tempSum>maxSum
                        maxSum=tempSum
                        store i,j
```

<u>Analysis</u>

This algorithm runs (1+2+…+n) times, performing 1 operations and 1 comparison each time. Thus, the runtime is $2*[(n(n+1)/2] = n(n+1) = O(n^2)$

Algorithm 3

Pseudocode

```
def maxCross(array, low, mid, high):
    leftSum = rightSum =  tempSum = 0
    leftHigh = mid
    for i from mid to low:
        tempSum = tempSum + array[i]
        if tempSum > leftSum:
            leftSum = tempSum
            leftHigh = i

    tempSum = 0
    rightHigh = mid+1
    for j from mid+1, high:
        tempSum = tempSum + array[j]
        if tempSum > rightSum:
            rightSum = tempSum
            rightHigh = j

    return(leftHigh, rightHigh, leftSum+rightSum)


def maxSubArray(array, low, high):
    if high =low:
        return(low, high, array[low])
    else:
        recursive call to find maxsubarray of left half of array (leftSum)
        recursive call to find maxsubarray of right half of array (rightSum)
        recursive call to find maxsubarray spanning middle of array (crossSum)

    compare left/right/cross sums, return highest value
```

Analysis

This algorithm can be expressed recursively by breaking it into 2 problems of n/2 size. The crossover sum can be found in $\Theta(n)$ time. Thus,

$T(n) = 2T(n/2) + \Theta(n)$ for $n>1$

Master Theorem Case 2 yields $\Theta(n\log(n))$

Proof of Correctness

Algorithm 3

For simplicity, assume original array is of size 2^k

Basis Step: n = 1. It is trivial to see that leftMax = array[0], rightMax = array[0], and crossoverMax = array[0]. Thus, the maxSum = max(leftMax, rightMax, crossoverMax) = array[0].

Inductive Step: (top-down induction): Because any subarray will be found in either the left half, the right half, or in a crossover of the two halves, it is apparent that the maximum subarray will be found in any of these three locations.

Case 1: maxSum can be found in the left half.
    The algorithm will be recursively called on the sub-left halves until leftMax is found, and returned.
Case 2: maxSum can be found in the right half.
    Similarly to case 1, the algorithm will recursively calculate rightMax.
Case 3: maxSum can be found in the crossover.
    This will only be found if the leftMax's highest index and the rightMax's lowest index are consecutive indexes. It is trivial to see that the maxSum would be the sum of the leftMax and the rightMax.

Thus, the algorithm is correct.

Testing

Results of MSS_TestSets-1.txt

Test results for Algorithm 1
[1, 4, -9, 8, 1, 3, 3, 1, -1, -4, -6, 2, 8, 19, -10, -11], [8, 1, 3, 3, 1, -1, -4, -6, 2, 8, 19], 34
[2, 9, 8, 6, 5, -11, 9, -11, 7, 5, -1, -8, -3, 7, -2], [2, 9, 8, 6, 5], 30
[10, -11, -1, -9, 33, -45, 23, 24, -1, -7, -8, 19], [23, 24, -1, -7, -8, 19], 50
[31, -41, 59, 26, -53, 58, 97, -93, -23, 84], [59, 26, -53, 58, 97], 187
[3, 2, 1, 1, -8, 1, 1, 2, 3], [3, 2, 1, 1], 7
[12, 99, 99, -99, -27, 0, 0, 0, -3, 10], [12, 99, 99], 210
[-2, 1, -3, 4, -1, 2, 1, -5, 4], [4, -1, 2, 1], 6
[-1, -3, -5], [-1], -1

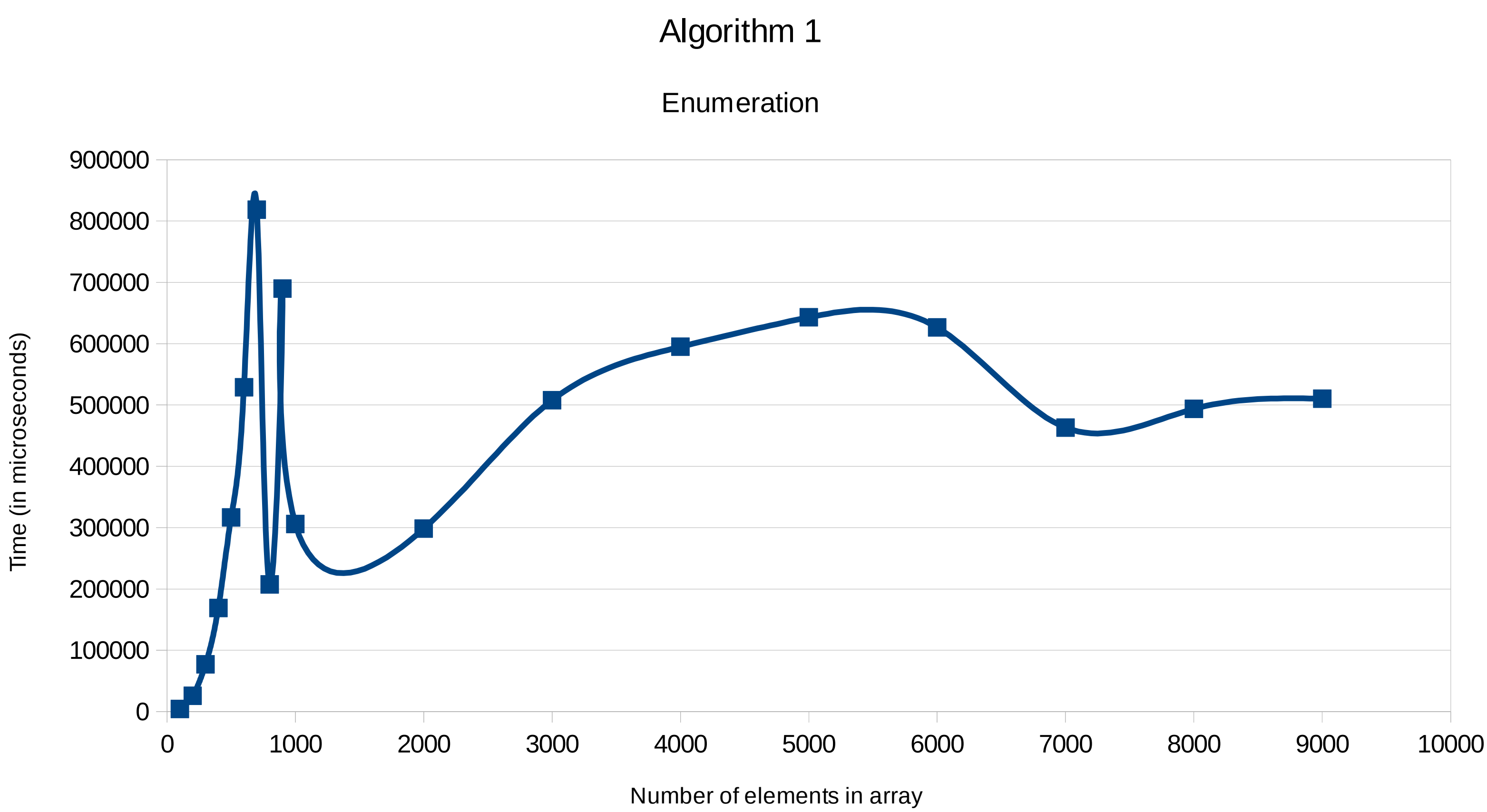
Test results for Algorithm 2
[1, 4, -9, 8, 1, 3, 3, 1, -1, -4, -6, 2, 8, 19, -10, -11], [8, 1, 3, 3, 1, -1, -4, -6, 2, 8, 19], 34
[2, 9, 8, 6, 5, -11, 9, -11, 7, 5, -1, -8, -3, 7, -2], [2, 9, 8, 6, 5], 30
[10, -11, -1, -9, 33, -45, 23, 24, -1, -7, -8, 19], [23, 24, -1, -7, -8, 19], 50
[31, -41, 59, 26, -53, 58, 97, -93, -23, 84], [59, 26, -53, 58, 97], 187
[3, 2, 1, 1, -8, 1, 1, 2, 3], [3, 2, 1, 1], 7
[12, 99, 99, -99, -27, 0, 0, 0, -3, 10], [12, 99, 99], 210
[-2, 1, -3, 4, -1, 2, 1, -5, 4], [4, -1, 2, 1], 6
[-1, -3, -5], [-1], 0

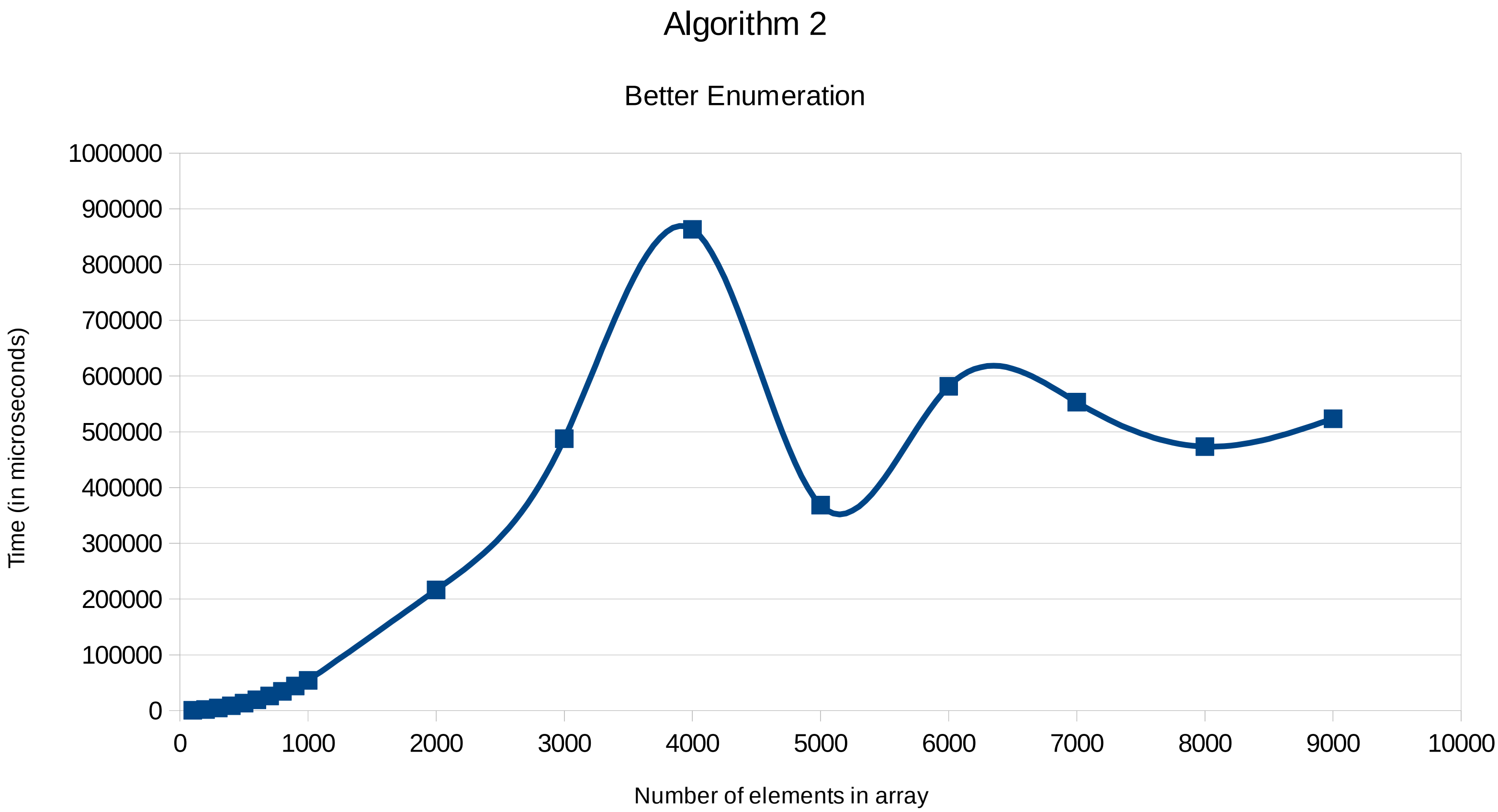
Test results for Algorithm 3
[1, 4, -9, 8, 1, 3, 3, 1, -1, -4, -6, 2, 8, 19, -10, -11], [8, 1, 3, 3, 1, -1, -4, -6, 2, 8, 19], 34
[2, 9, 8, 6, 5, -11, 9, -11, 7, 5, -1, -8, -3, 7, -2], [2, 9, 8, 6, 5], 30
[10, -11, -1, -9, 33, -45, 23, 24, -1, -7, -8, 19], [23, 24, -1, -7, -8, 19], 50
[31, -41, 59, 26, -53, 58, 97, -93, -23, 84], [59, 26, -53, 58, 97], 187
[3, 2, 1, 1, -8, 1, 1, 2, 3], [3, 2, 1, 1], 7
[12, 99, 99, -99, -27, 0, 0, 0, -3, 10], [12, 99, 99], 210
[-2, 1, -3, 4, -1, 2, 1, -5, 4], [4, -1, 2, 1], 6
[-1, -3, -5], [-1], 0

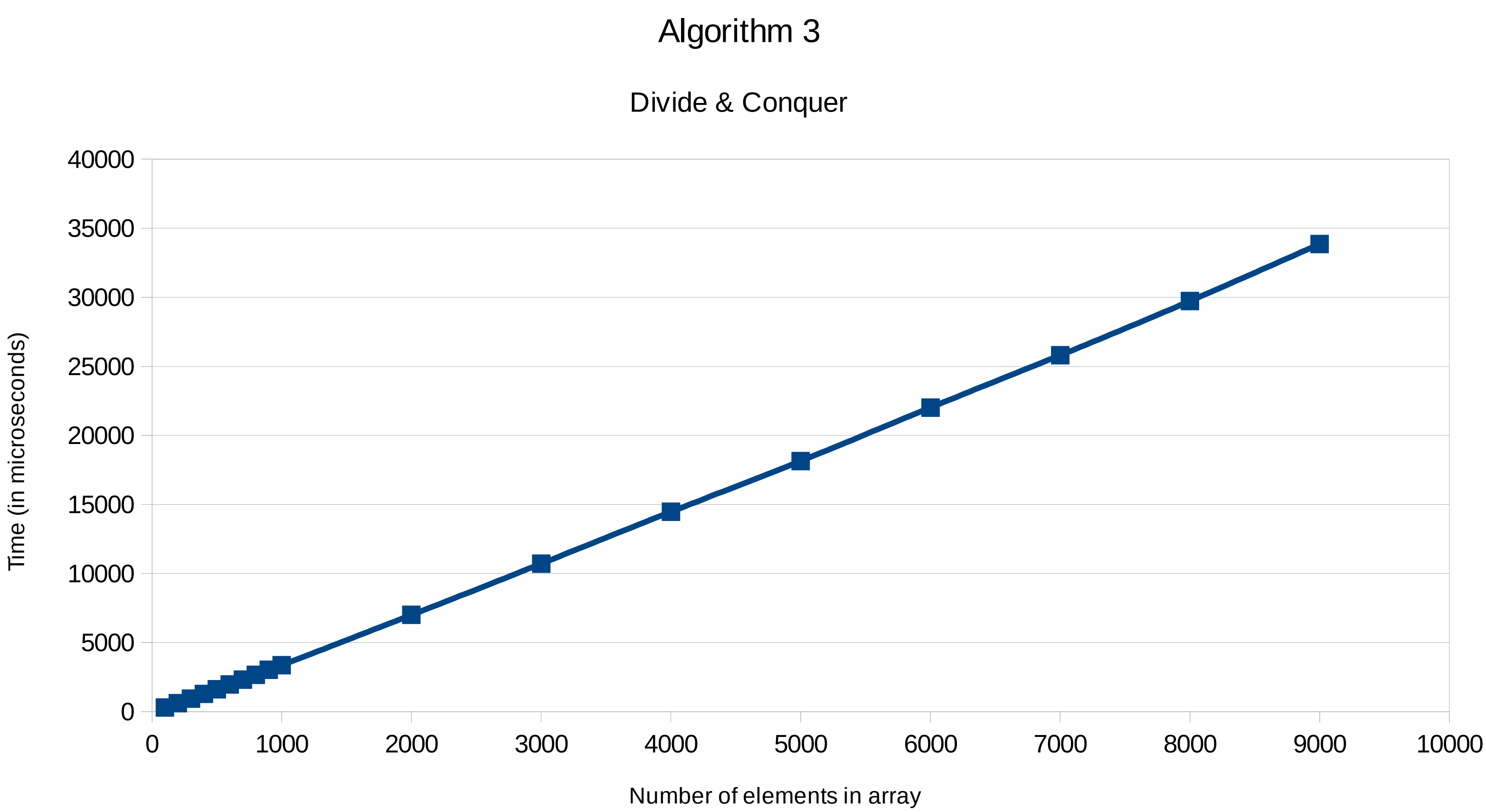
We did not understand how to implement Algorithm 4, so it is absent from our results.

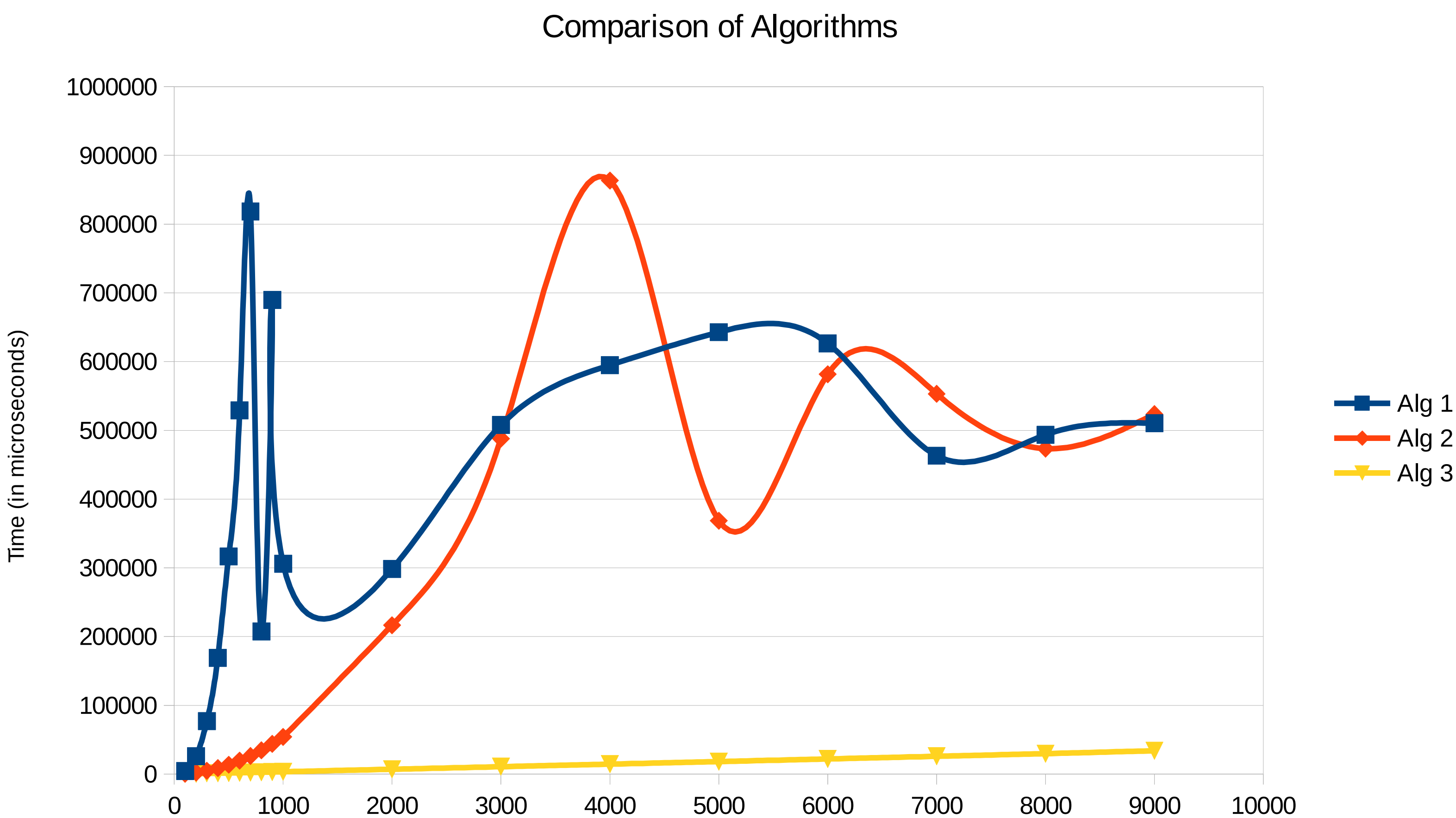
Experimental Analysis

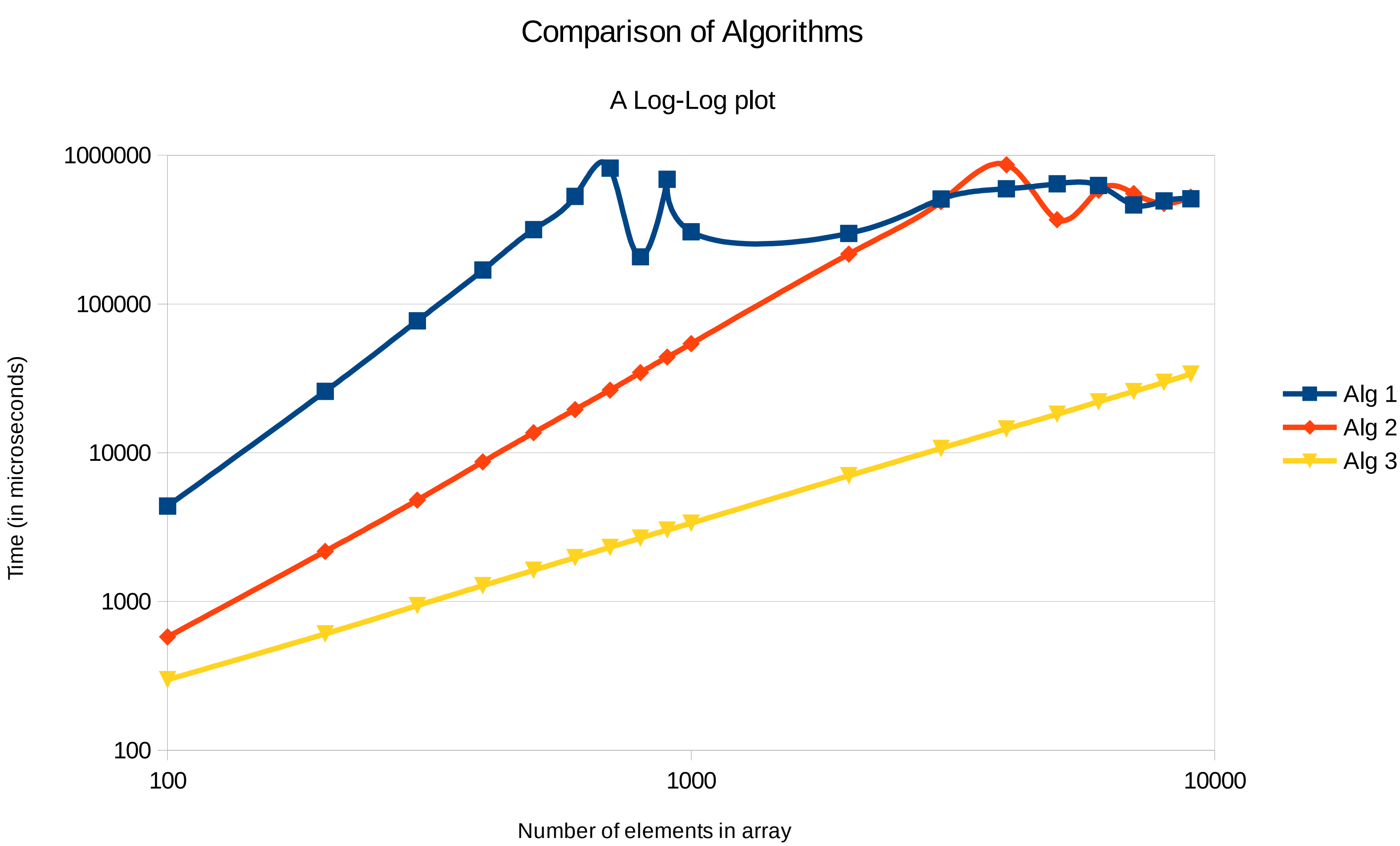
Average running times for algorithms




Algorithm 2
Better Enumeration
Time (in microseconds)
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Number of elements in array


Algorithm 3
Divide & Conquer
Time (in microseconds)
40000
35000
30000
25000
20000
15000
10000
5000
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Number of elements in array


Comparison of Algorithms
Time (in microseconds)
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Alg 1
Alg 2
Alg 3


Comparison of Algorithms
A Log-Log plot
Time (in microseconds)
1000000
100000
10000
1000
100
100
1000
10000
Number of elements in array
Alg 1
Alg 2
Alg 3

It seems like there is maybe some kind of error in our implementations of algorithms 1 and 2. We would expect a smoother upward curve for both of them, but they had weird dips in their timings. For example, Algorithm1's average run time for n=800 is significantly lower than the time for n=700. Maybe this fluctuation was due to processor load on flip? Algorithm 2 has similar fluctuation in it too.

Extrapolation & Interpretation

Algorithm 1

Our theoretical run time was O(n!). The beginning of the graph supports this, as the timing rapidly climbs as *n* grows.

Algorithm 2

Our theoretical run time was $O(n^2)$. The beginning of the graph supports this, as the timing rapidly climbs as *n* grows, but not as rapidly as n!.

Algorithm 3

Our theoretical run time was $\Theta(n\log(n))$. The graph seems to support this, with no noticeable curve for our relatively small values of *n.*

We were not sure how to calculate max *n* size for running times of 1 hour.