Project 2
CS 325
October 23, 2014
Group Members: Emma Paul, Ian Paul, Abdulhalim Bambang
Group Number: 4

Run-time analysis:

Algorithm 1: Alg1.c

```
for all the elements in array1 (j=0; j < array 1 size ; j++)
        for all the possible iterations of the suffices of array 1
                compute sum
        End for
        for all the elements in array 2 (z=0; z < array 2 size ; z++)
                for all the iterations of the suffices of array 2
                compute sum
                End for
        End for
        compute total of sums from array 1 and array 2
        if absolute sum is less than lowest sum
                save absolute sum as lowest sum
                save start and end positions of both arrays
End for

print full array 1
print full array 2
print suffix of array 1 from start position to end position
print suffix of array 2 from start position to end position
print sum of suffix 1 + suffix 2
```

Analysis of the asymptotic running-time: O(n^4) because for every iteration of suffices of array 1 (which is O(n^2), and every iteration of suffices of array 2 is computed (which is also O(n^2)). Since there is an O(n^2) within an O(n^2), mathematically that gives us (n^2)(n^2) which is O(n^4).

Algorithm 2: Alg2.c

```
for all the elements in array1 (j=0; j < array 1 size ; j++)
        for all the possible iterations of the suffices of array 1
                compute sum
        End for
        put sums of all iterations into their own array called array1Sums
End for
for all the elements in array2 (z=0;  z< array 1 size ; z++)
        for all the possible iterations of the suffices of array 2
                compute sum
        End for
        put sums of all iterations into their own array called array2Sums
End for
```

```
for all the elements in array1sums
	store positions of array1Sums into a new array called array1SumsPos
End for
for all the elements in array2sums
	store positions of array2Sums into a new array called array2SumsPos
End for
for all the elements in array1sums (j=0; j < array1sums size - 1; j++)
	for all elements in array1sums (z= 0; z < array1sums size – 1 – j; z++)
		sort elements using bubble sort
		sort array1SumsPos to reflect updated sorted positions
	End for
End for
for all the elements in array2sums (j=0; j < array2sums size - 1; j++)
	for all elements in array2sums (z= 0; z < array2sums size – 1 – j; z++)
		sort elements using bubble sort
		sort array2SumsPos to reflect updated sorted positions
	End for
End for
for all the elements in array1sums
	for all elements in array2sums
		compute sum of elements
		store absolute lowest sum
		store corresponding sum positions
	End for
End for
print full array 1
print full array 2
print suffix of array 1 starting from position
print suffix of array 2 starting from position
print sum of suffix 1 + suffix 2
```

*Analysis of the asymptotic running-time: O(n log n) because sums of suffices of array 1 and 2 are computed and stored once, which is O(n), and then the sums of array 1 and 2 are sorted and iterated through to find the lowest sum between the two, which is O(log n). Because instead of summing through brute force, the sums have already been stored and sorted, any additional array element does not cause an n^2 time increase, just a log n time increase, so the time complexity combined is O(n log n).

<u>Algorithm 3: Alg3.c</u>

```
for all the elements in an array
	populate array 1 with first half of elements in array
end for
for all the elements in an array
	populate array 2 with second half of elements in array
end for
for all the elements in array1 (j=0; j < array 1 size ; j++)
	for all the possible iterations of the suffices of array 1
		compute sum
```

```
        End for
        for all the elements in array 2 (z=0; z < array 2 size ; z++)
                for all the iterations of the prefixes of array 2
                compute sum
                End for
        End for
        compute total of sums from array 1 and array 2
        if absolute sum is less than lowest sum
                save absolute sum as lowest sum
                save start and end positions of both arrays
End for

print full array 1
print full array 2
print suffix of array 1 from start position to end position
print prefix of array 2 from start position to end position
print sum of suffix 1 + prefix 2
```

Analysis of the asymptotic running-time: O(n^4) because for every iteration of suffices of array 1 (which is O(n^2), and every iteration of prefixes of array 2 is computed (which is also O(n^2)). Since there is an O(n^2) within an O(n^2), mathematically that gives us (n^2)(n^2) which is O(n^4).

Algorithm 4: Alg4.c

```
for all the elements in an array
        populate array 1 with first half of elements in array
end for
for all the elements in an array
        populate array 2 with second half of elements in array
end for
for all the elements in array1 (j=0; j < array 1 size ; j++)
        for all the possible iterations of the suffices of array 1
                compute sum
        End for
        put sums of all iterations into their own array called array1Sums
End for
for all the elements in array2 (z=0;  z< array 1 size ; z++)
        for all the possible iterations of the prefixes of array 2
                compute sum
        End for
        put sums of all iterations into their own array called array2Sums
End for
for all the elements in array1sums
        store positions of array1Sums into a new array called array1SumsPos
End for
for all the elements in array2sums
        store positions of array2Sums into a new array called array2SumsPos
End for
for all the elements in array1sums (j=0; j < array1sums size - 1; j++)
```

```
        for all elements in array1sums (z= 0; z < array1sums size – 1 – j; z++)
                sort elements using bubble sort
                sort array1SumsPos to reflect updated sorted positions
        End for
End for
for all the elements in array2sums (j=0; j < array2sums size - 1; j++)
        for all elements in array2sums (z= 0; z < array2sums size – 1 – j; z++)
                sort elements using bubble sort
                sort array2SumsPos to reflect updated sorted positions
        End for
End for
for all the elements in array1sums
        for all elements in array2sums
                compute sum of elements
                store absolute lowest sum
                store corresponding sum positions
        End for
End for
print full array 1
print full array 2
print suffix of array 1 starting from position
print prefix of array 2 starting from position
print sum of suffix 1 + prefix 2
```

*Analysis of the asymptotic running-time: O(n log n) because sums of suffices of array 1 and sums of prefixes of array 2 are computed and stored once, which is O(n), and then the sums of array 1 and 2 are sorted and iterated through to find the lowest sum between the two, which is O(log n). Because instead of summing through brute force, the sums have already been stored and sorted, any additional array element does not cause an n^2 time increase, just a log n time increase, so the time complexity combined is O(n log n).

Proofs of Correctness:

<u>Proof by contradiction that Algorithm 2 returns the correct solution</u>

Claim: Algorithm 2 returns a sum of suffices of array 1 and array 2 that is closest to zero.
Proof: To prove by contradiction, assume that there exists an input I on which Algorithm 2 does not return a sum of suffices of array1 and array 2 that is closest to zero.
-Algorithm 2 computes all possible sum of suffices iterations of array 1 and array 2
-Algorithm 2 calculates the difference between all the sums found in array 1 and 2, keeping the lowest sum
-Because it stores the lowest sum after exhausting all possible sum of suffices, there does not exist an input I in which Algorithm 2 does not return a sum of suffices of array 1 and array 2 that is closest to zero, therefore a contradiction.

<u>Proof by induction that Algorithm 4 returns the correct solution</u>

Claim: In an array, there exists a subset array where its sum is closest to zero. Ignoring the case that the subset array with sum closest to zero exists entirely in the first half, or second half, there exists a subset array consisting of the suffix of the first half and a prefix of the second half whose sum is closest to zero.

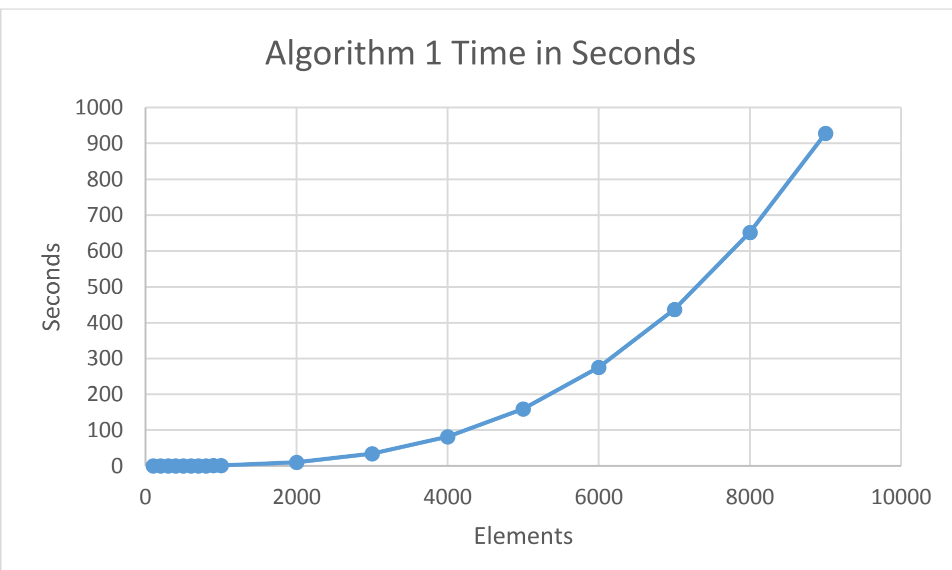
Inductive Hypothesis:
An array is split into 2 arrays, array 1 containing the first half of the array, array 2 containing the second half of the array. Algorithm 4 produces a subset array consisting of the suffix of the first half and a prefix of the second half whose sum is closest to zero.

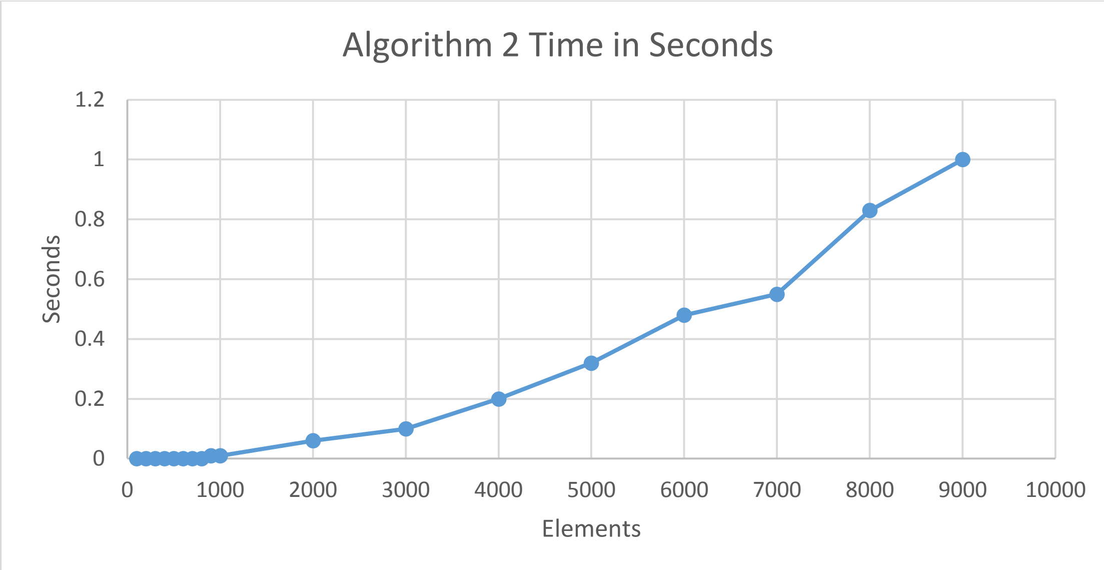
Base case:
Array1sortedSums: [m – 2] < [m - 1] and Array2sortedSums: [0] < [1]
*The line above states that the array sums are sorted from lowest to highest

Proof:
Where any value of Array1sortedSums[m - j] will always be less than Array1sortedSums[m - 1] because they are sorted, and any value of Array2sortedSums[0] will always be less than Array2sortedSums[0 + j].

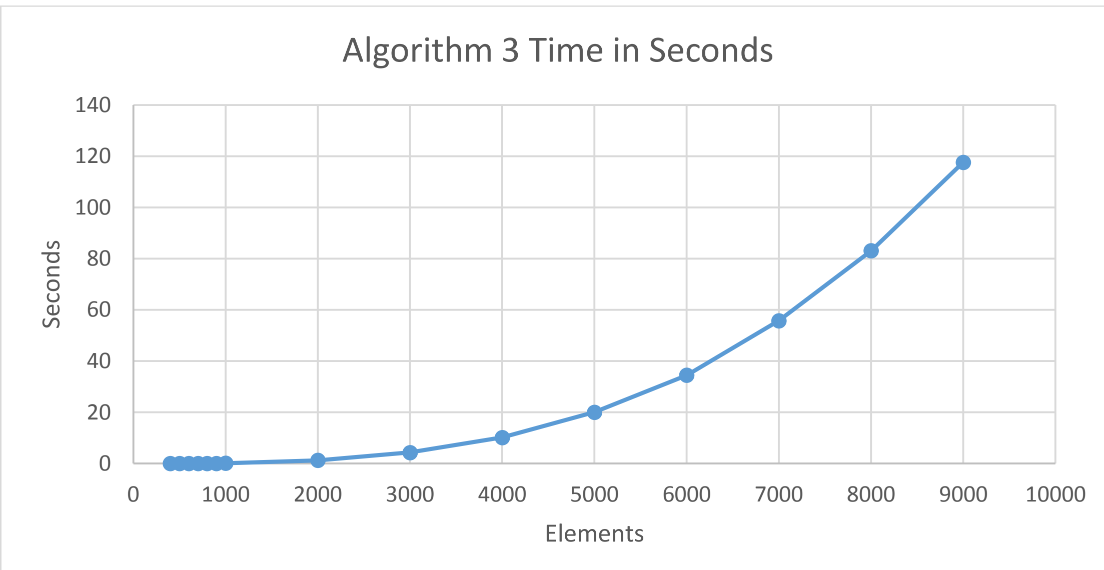
Induction:
Therefore, any combination of either Array1sortedSums[m - j] + Array2sortedSums[0] or Array1sortedSums[m - 1] + Array2sortedSums[0 + j] will always be greater than Array1sortedSums[m - 1] + Array2sortedSums[0]

Experimental analysis:

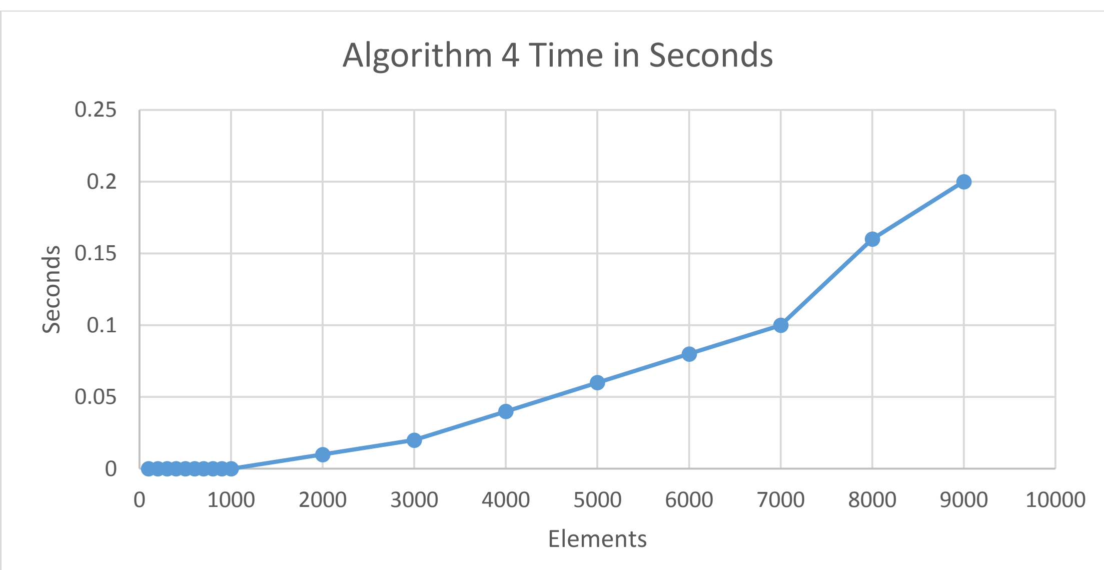




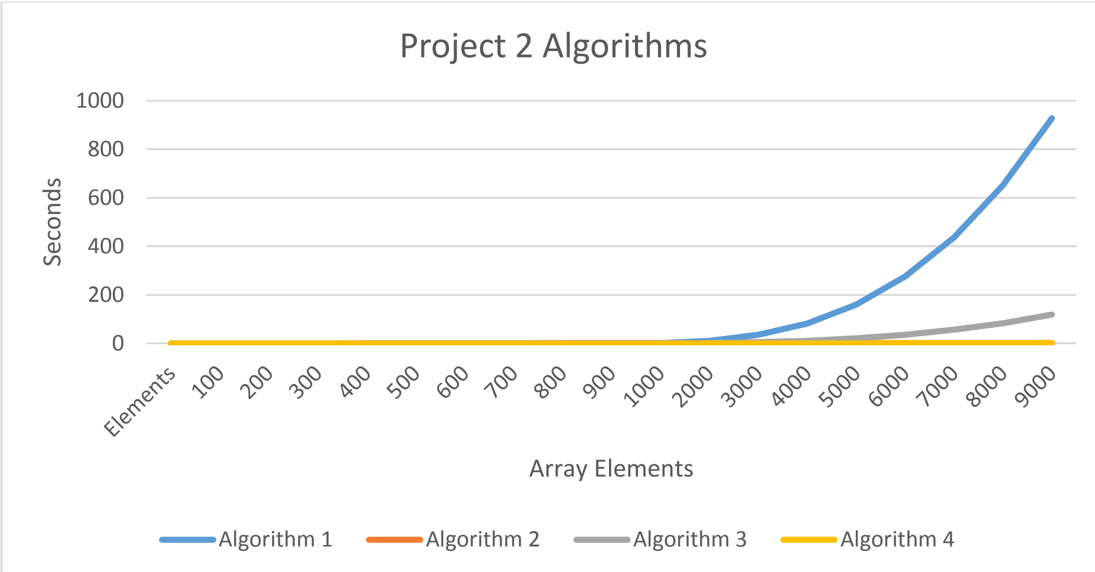
Algorithm 2 Time in Seconds
Seconds
1.2
1
0.8
0.6
0.4
0.2
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Elements

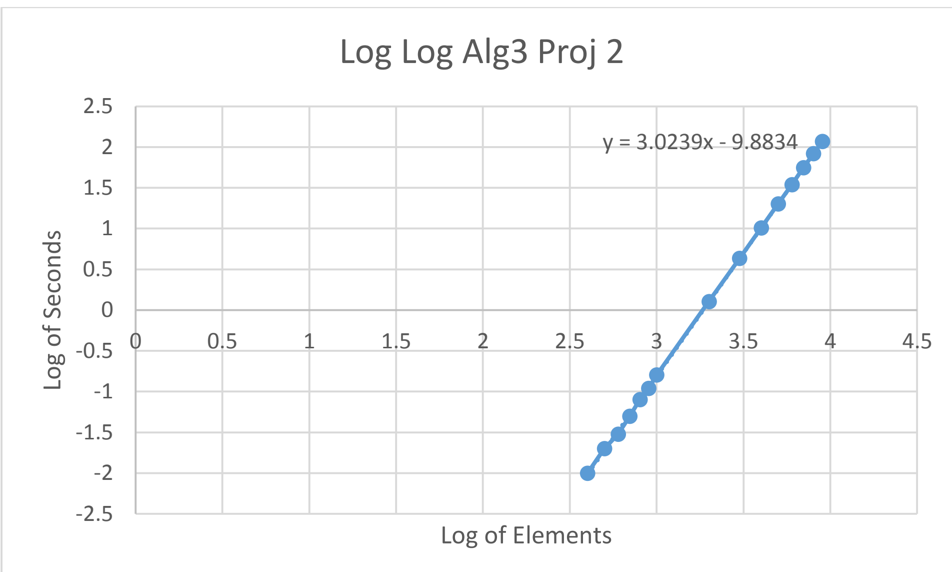


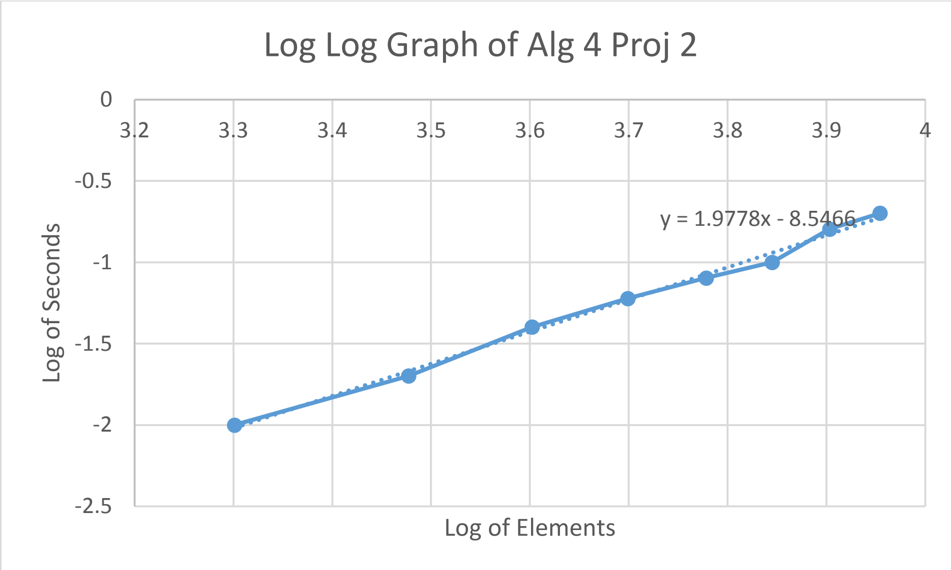
Algorithm 3 Time in Seconds
Seconds
140
120
100
80
60
40
20
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Elements

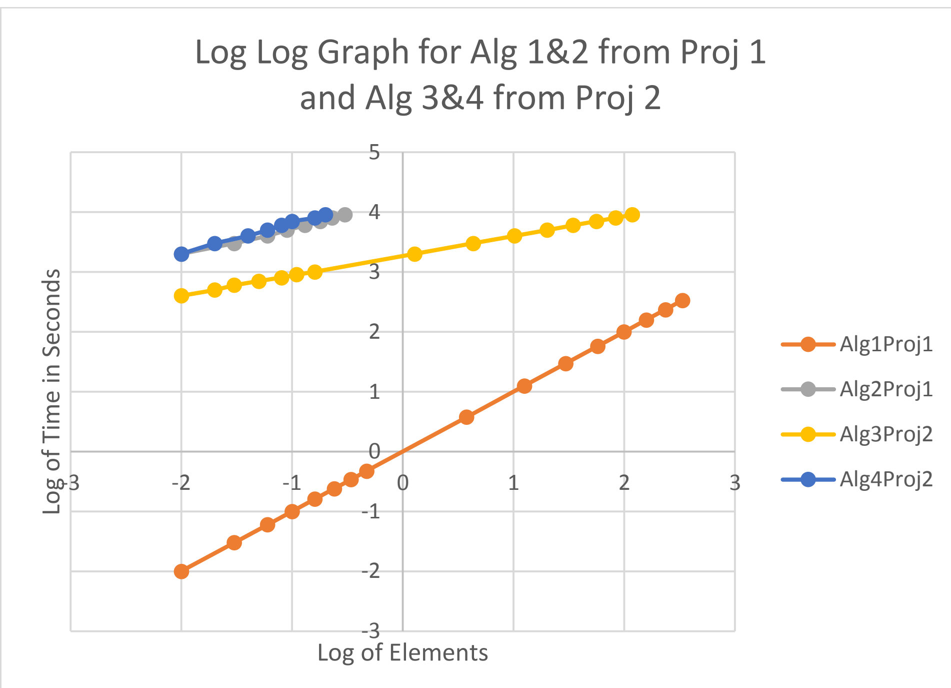


Algorithm 4 Time in Seconds
Seconds
0.25
0.2
0.15
0.1
0.05
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Elements


Project 2 Algorithms
Seconds
1000
800
600
400
200
0
Elements
100
200
300
400
500
600
700
800
900
1000
2000
3000
4000
5000
6000
7000
8000
9000
Array Elements
Algorithm 1
Algorithm 2
Algorithm 3
Algorithm 4


Log Log Alg3 Proj 2
Log of Seconds
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
y = 3.0239x - 9.8834
Log of Elements





Extrapolation and interpretation:

1. Highest number of elements that will run in under an hour
   Algorithm 3: 28,000 elements
   Algorithm 4: 1,310,000 elements

2. Algorithm 3 Slope = 3.0239, experimental running time = O(n^3.0239), which is a faster running time than our theoretical analysis (which makes sense, because our theoretical analysis was for the worst case scenario)

Algorithm 4 Slope = 1.9778, experimental running time = O(n^1.9778), which is a faster running time than our theoretical analysis (which makes sense, because our theoretical analysis was for the worst case scenario)

Project Resources

http://cs.stackexchange.com/questions/10091/what-are-the-characteristics-of-an-on-log-n-time-complexity-algorithm

https://justin.abrah.ms/computer-science/big-o-notation-explained.html

http://www.cs.mcgill.ca/~dprecup/courses/IntroCS/Lectures/comp250-lecture12.pdf

http://imps.mcmaster.ca/

http://stackoverflow.com/questions/1952070/writing-a-proof-for-an-algorithm

http://wcherry.math.unt.edu/math1650/exponential.pdf

http://stackoverflow.com/questions/1674032/static-const-vs-define-in-c

http://stackoverflow.com/questions/1712592/variably-modified-array-at-file-scope

http://www.codebeach.com/2008/09/sorting-algorithms-in-c.html