Project 1 Report

John Fitzpatrick, Nicholas Hartley
Group 27
July 10, 2016

Theoretical Run-time Analysis

Pseudo-code and run-time analysis of algorithms 1-4.

NOTE: results is the variable name for a result structure created in our code with two integer variables named length and maxSum, and an integer array named data.

Algorithm 1 - Enumeration:

```
Enumeration(a[1,...,n], length, results)
        sum ← max ←  0
        For each pair (i, j) with i <= 1 <= j <= length
                Sum ← Compute a[i] + a[i+1] + … + a[j-1] + a[j]
                If max < sum
                        max ← sum
                        low_Index ← i
                        hi_Index ← j
        results.maxSum ← Max
       j ← 1
        For i ← low_Index upto hi_Index
                Results.data[j] ← a[i]
                Increment j
        Results.length ← j
        Return results
```

Analysis: (O(n^2) pairs * (O(n) time to compute each sum) = O(n^3) runtime

Algorithm 2 - Better Enumeration:

```
BetterEnumeration(a[1,...,n], length, results)
        sum ← max ←  0
        For i ← 1 upto length
                sum ← a[i]
                For j ← i upto length
                        sum ← sum + a[j]
                        If max < sum
                                max ← sum
                                low_Index ← i
                                hi_Index ← j
        results.maxSum ← Max
        j ← 1
        For i ← low_Index upto hi_Index
                Results.data[j] ← a[i]
                Increment j
        Results.length ← j
        Return results
```

Analysis: (O(n) i-iterations * (O(n) j-iterations) * O(1) = O(n^2) runtime

Algorithm 3 - Divide and Conquer:

```
DivideConquer(a[1,...,n], length, results)
        low_Index ← hi_Index ← 0
        results.maxSum ← a[1]
        results.maxSum, low_Index, hi_Index ← MaxSubArray(a[1,...,n], 0, length, results)
         j ← 0
        For i ← low_Index upto hi_Index
                Results.data[j] ← a[i]
                Increment j
        Results.length ← j
        Return results

MaxSubArray(a[1,...,n], low_Index, hi_Index, results)
        If low_Index >= hi_Index
                return a[low_Index], lo_Max, hi_Max
        Else
                Mid ← (hi_Index - low_Index) / 2 + low_Index
                lowSum, lo_lo_Idx, lo_hi_Idx ← MaxSubArray(a[1,...,n], low_Index, Mid, results)
                hiSum, hi_lo_Idx, hi_hi_Idx ← MaxSubArray(a[1,...,n], Mid+1, hi_Index, results)
                cxSum, cx_lo_Idx, cx_hi_idx ← MaxSuffix(a[1,...,n], low_Index, Mid, cx_lo_Idx) +
                                                MaxPrefix(a[1,...,n], Mid+1, hi_Index, cx_hi_idx)
                If lowSum >= cxSum and lowSum >= hiSum
                        If lowSum > results.maxSum
                                results.maxSum ← lowSum
                                return lowSum, lo_lo_Idx, lo_hi_Idx
                Else If cxSum >= lowSum and cxSum >= hiSum
                        If csSum > results.maxSum
                                results.maxSum ← cxSum
                                return cxSum, cx_lo_Idx, cx_hi_idx
                Else
                        If hiSum > results.maxSum
                                results.maxSum ← hiSum
                                return hiSum, hi_lo_Idx, hi_hi_Idx

MaxSuffix(a[1,...,n], end_Index, start_Index)
        max ←  a[start_Index], sum ← 0
        While start_Index >= end_Index
                sum ← sum + a[start_Index]
                If sum > max
                        max ← sum
                        suffix_lo_Idx ← Mid
                Decrement start_Index
        Return max, suffix_lo_Idx

MaxPrefix(a[1,...,n], start_Index, end_Index)
        max ←  a[start_Index], sum ← 0
        While start_Index <= end_Index
                sum ← sum + a[start_Index]
                If sum > max
                        max ← sum
                        preffix_hi_Idx ← Mid
                Increment start_Index
        Return max, preffix_hi_Idx
```

Analysis: (O(n) time for non=recursive work) * (O(log n) depth) = O(n log n) run-time

Algorithm 4 - Linear-time:

```
LinearTime(a[1,...,n], length, results)
        low_Index ← hi_Index ← 0
        Create and allocate new array called maxSubArray
        largestSA ← maxSubArray[1] ← a[1]
        For i ← 2 upto length
                sum ← a[i] + maxSubArray[i-1]
                If sum > a[i]
                        maxSubArray[i] ← sum
                Else
                        maxSubArray[i] ← a[i]
                If sum > largestSA
                        largestSA ← sum
                        hi_Index ← i
        low_Index ← hi_Index
        sum ← a[hi_Index]
        Do while sum != largestSA
                Decrement low_Index
                sum ← sum + a[low_Index]
        results.maxSum ← largestSA
         j ← 0
        For i ← low_Index upto hi_Index
                Results.data[j] ← a[i]
                Increment j
        Results.length ← j
        Return results
```

Analysis: O(n) things to compute = O(n) run-time

Testing

For the initial testing of the algorithms, each was run against the supplied problem set and verified manually for correctness. We determined that a satisfactory method for further testing would be for each team member to create their own set of arrays, which would include random and non-random values, and individually check the output of each for correctness. Once each member was done with their tests, we met to share test results and discuss any potential problems that may have occurred during tests. Once we were able to fix a couple of bugs, each member submitted their test results for review. After closely reviewing each random and individual test cases and results, we were able to conclude and verify the correctness of the algorithms. Confident that our results backed up the accuracy of our algorithms, we next moved on to commence our experimental analysis.

Experimental Analysis

Algorithm 1 - Enumeration:

Size	Average (ms)
600	178
700	262
800	371
900	574
1,000	756
2,000	5,541
3,000	18,258
4,000	43,384
5,000	87,163
6,000	146,577

Algorithm 2 - Better Enumeration:

Size	Average (ms)
8,000	164
10,000	224
12,000	357
14,000	437
16,000	572
18,000	704
20,000	847
30,000	1,888
40,000	3,317
50,000	5,454
60,000	7,407
70,000	9,939

Algorithm 3 - Divide and Conquer:

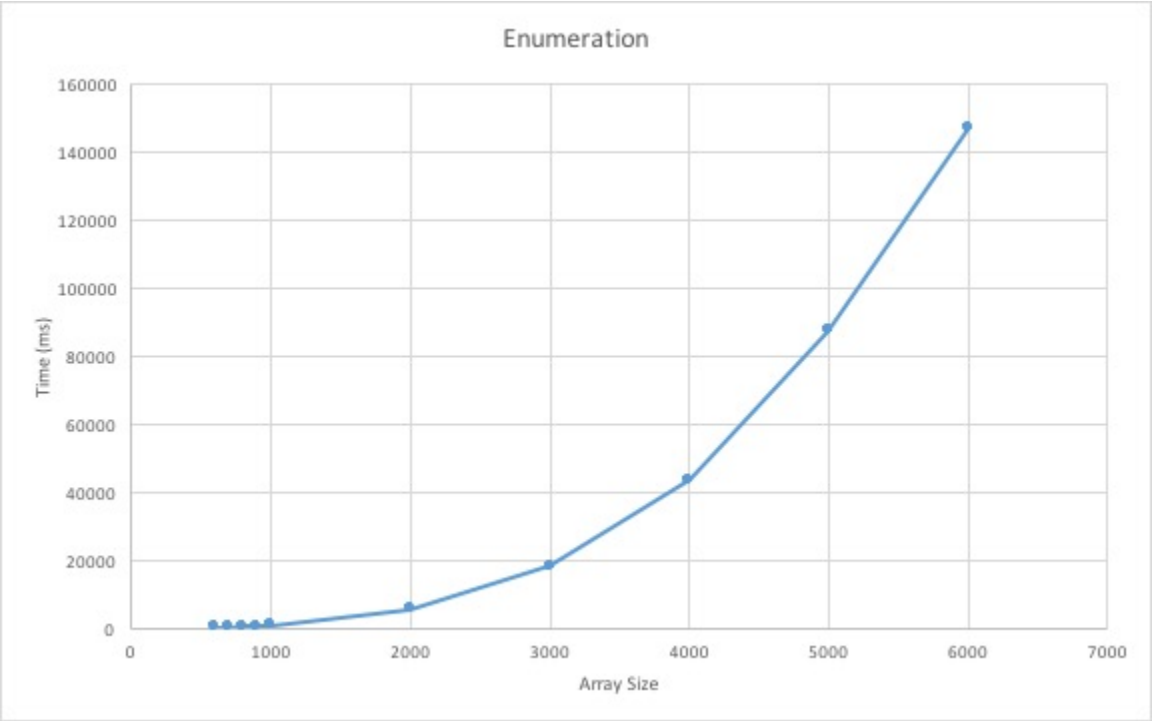
Size	Average (ms)
600,000	86
700,000	110
800,000	120
900,000	125
1,000,000	154
1,100,000	160
1,200,000	175
1,300,000	187
1,400,000	205
1,500,000	229
1,750,000	260
2,000,000	292

Algorithm 4 - Linear Time:

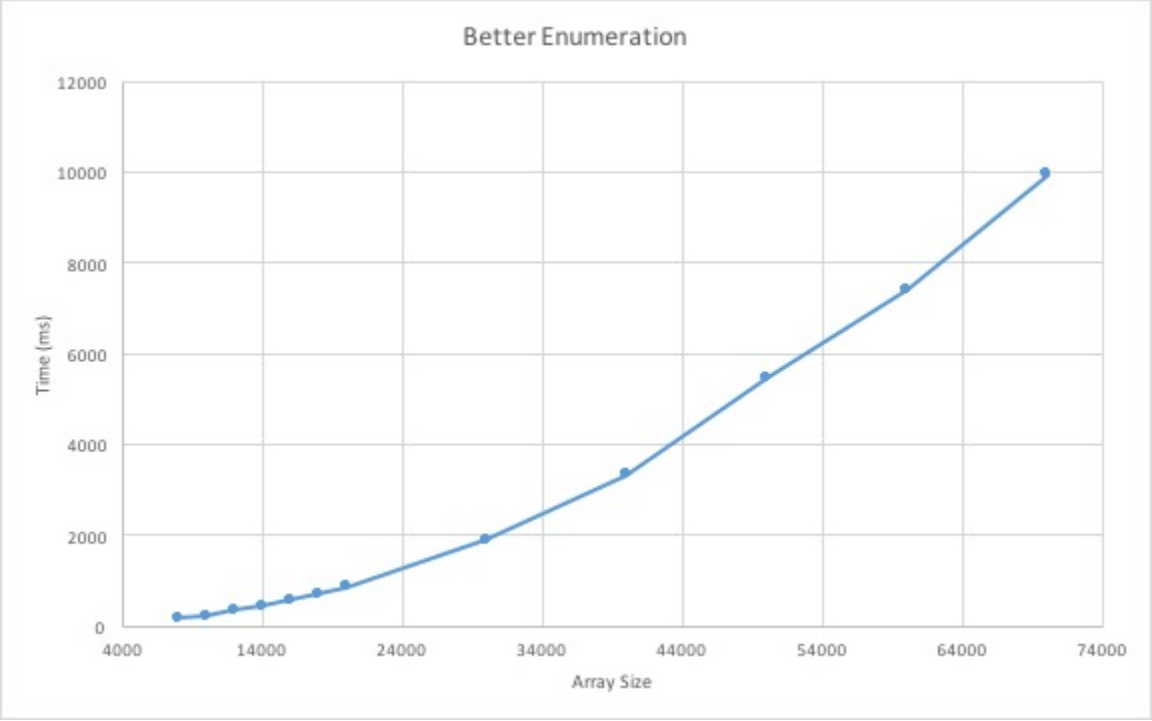
Size	Average (ms)
600,000	10
700,000	11
800,000	10
900,000	11
1,000,000	15
1,100,000	13
1,200,000	16
1,300,000	14
1,400,000	21
1,500,000	21
1,750,000	23
2,000,000	29

Graphs of average running times

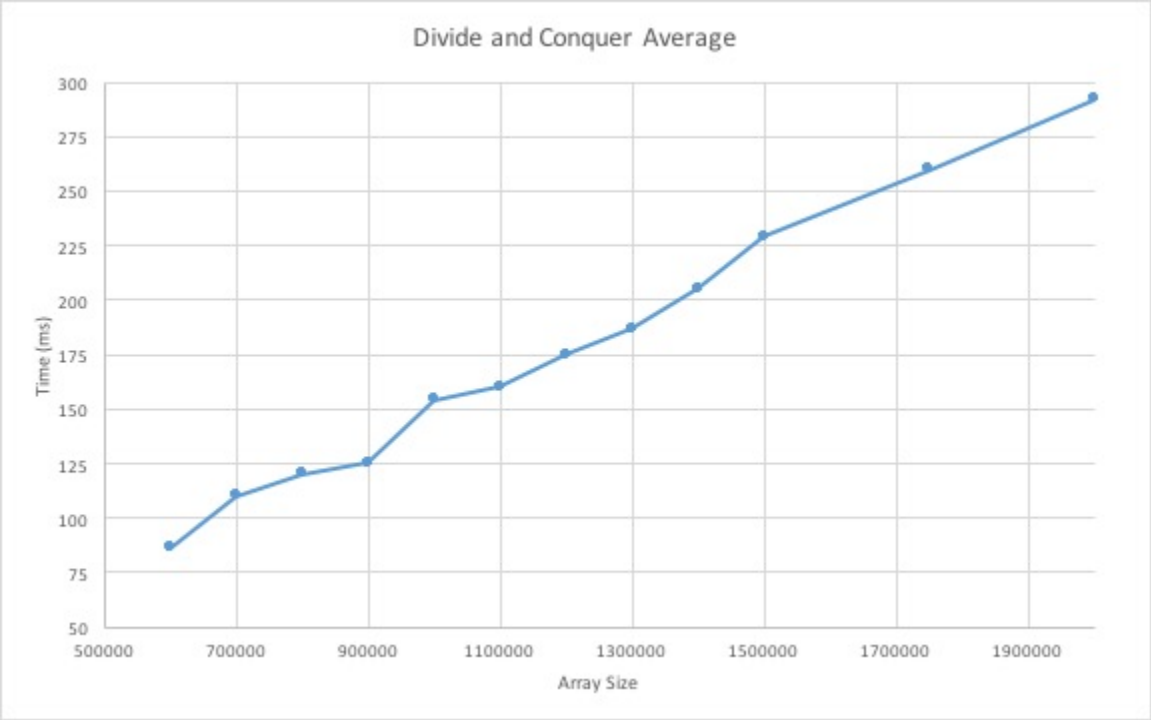
Algorithm 1 - Enumeration:



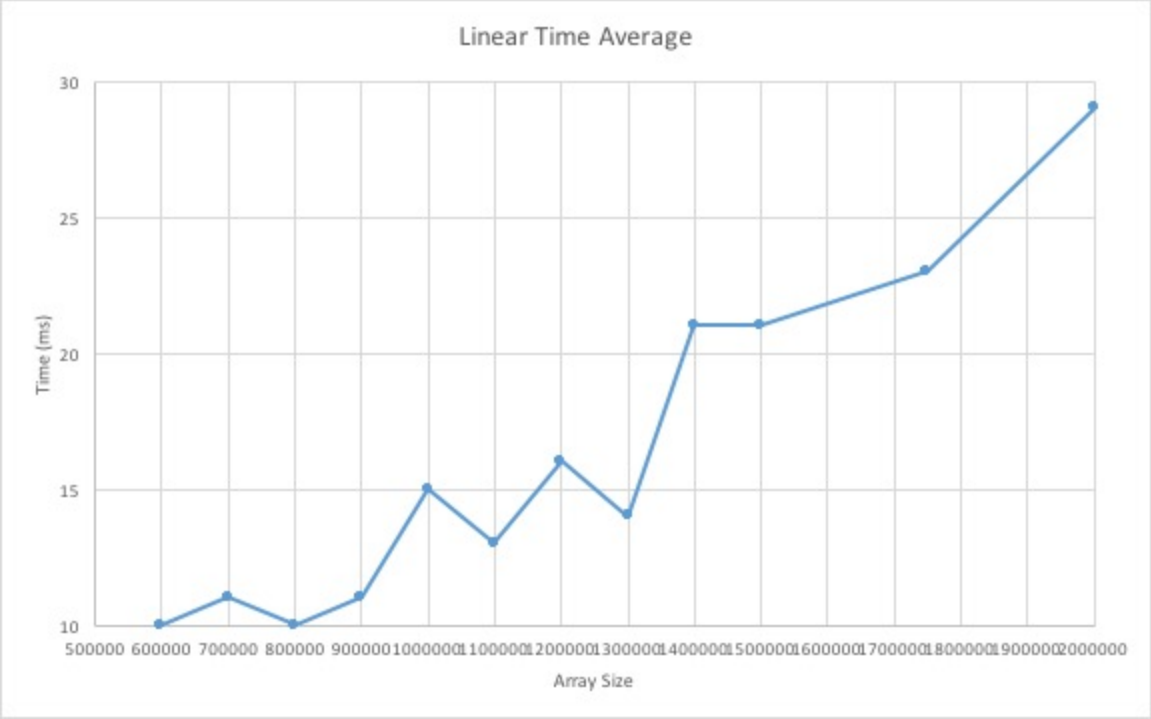
Algorithm 2 - Better Enumeration:



Algorithm 3 - Divide and Conquer:



Algorithm 4 - Linear Time:



Function between input size n and time

Algorithm 1 - Enumeration:
6E-07x^3 + 0.001x^2 - 2.4824x + 1445.3. Based on the shape of the graph curve and our data, we believe the data is quadratic; a polynomial to the power of 3 to be precise.

Algorithm 2 - Better Enumeration:
2E-06x^2 + 0.0118x - 94.43. Based on the shape of the graph curve and our data, we believe the data is quadratic; a polynomial to the power of 2 to be precise

Algorithm 3 - Divide and Conquer:
0.0001x + 0.8273. We know that the runtime of the algorithm is O(n log n). However, based on the shape of the graph line and plugging in different curves, we found that our function's curve fits more appropriately as a linear rather than logarithmic.

Algorithm 4 - Linear Time:
1E-05x+0.0152. Based on the shape of the graph line and our data, we believe that linear fits our equation's graph line the most.

Discrepancies between the experimental and theoretical running times

All of our tests were performed on Oregon State University's FLIP server. A pattern seemed to occur at some degree of randomness; when running times were to be recorded the server had a tendency to "spool up." It was observed that sometimes the first and/or second execution would take longer than the group in the middle. And some other times once the server "got going" the final measurements executed at a small, but noticeable, speed that was faster than the middle ground.

Algorithm 1 - Enumeration:
This algorithm seemed to match our expectation the most. As the size of n increased the run time grew out of control. Testing an array of 5,000 integers averaged approximate 87 seconds. Adding only 1,000 more integers ballooned the execution time to 146 seconds. Since the third degree polynomial dominated the rest of the factors, the correlation between theoretical calculations and real world observations matched very closely.

Algorithm 2 - Better Enumeration:
As with any polynomial it is easy to see the shape it takes once it reaches a certain value of n. The better enumeration algorithm, tough quicker than its relative, matched our theoretical expectations very closely. As with any experiment outside of a tightly controlled environment, some variance was observed but overall the data matched the expected curve very closely.

Algorithm 3 - Divide and conquer:
We expected this graph to conform to O(n log n), but it more closely matched a graph of O(n). This could have been because of a few different reasons. O(n) and O(n log n) only account for the dominant portion of the algorithm. There could be constant factors, variables that influence n as a multiplier, or other weaker polynomials that can still influence n.

Algorithm 4 - Linear Time:
We expected this to run at O(n) time and it seemed to perform that way. Our tests reached arrays of 2.5 million with any significant increase in run time. Though the fit was not as tight as we liked it we determined that it had a linear fit of O(n).

Regression model

Max input that can be solved in time	Enumeration	Good Enumeration	Divide & Conquer	Linear Time
10 seconds	2,464	69,996	6.807 x 10^7	1,688.17
30 seconds	3,512	123.045	2.042 x 10^8	5,069
60 seconds	4,427	175,134	4.084 x 10^8	10,142

Algorithm 1 - Enumeration:

*f(x) = p1*x^3 + p2*x^2 + p3*x + p4*

Coefficients (with 95% confidence bounds):

p1 = 5.772e-07 (4.148e-07, 7.395e-07)

p2 = 0.0009928 (-0.0006183, 0.002604)

p3 = -2.482 (-6.957, 1.992)

p4 = 1445 (-1471, 4361)

Goodness of fit:

SSE: 2.857e+06

R-square: 0.9999

Adjusted R-square: 0.9998

RMSE: 690

Algorithm 2 - Good Enumeration

*f(x) = p1*x^2 + p2*x + p3*

Coefficients (with 95% confidence bounds):

p1 = 1.892e-06 (1.697e-06, 2.087e-06)

p2 = 0.01178 (-0.003057, 0.02661)

p3 = -94.43 (-301.1, 112.2)

Goodness of fit:

SSE: 7.652e+04

R-square: 0.9994

Adjusted R-square: 0.9992

RMSE: 92.21

Algorithm 3 - Divide and Conquer

*f(x) = p1*x + p2*

Coefficients (with 95% confidence bounds):

p1 = 0.0001469 (0.0001388, 0.000155)

p2 = 0.8273 (-9.354, 11.01)

Goodness of fit:

SSE: 263.7

R-square: 0.9939

Adjusted R-square: 0.9933

RMSE: 5.135

Algorithm 4 - Linear Time

*f(x) = p1*x + p2*

where x is normalized by mean 1.188e+06 and std 4.254e+05

Coefficients (with 95% confidence bounds):

p1 = 5.914 (4.99, 6.838)

p2 = 16.17 (15.28, 17.05)

Goodness of fit:

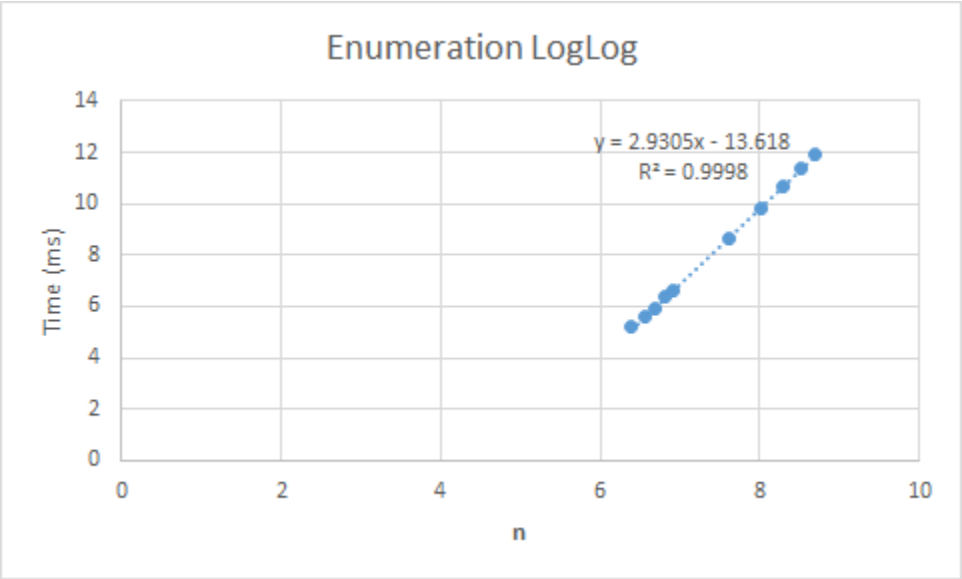
SSE: 18.91

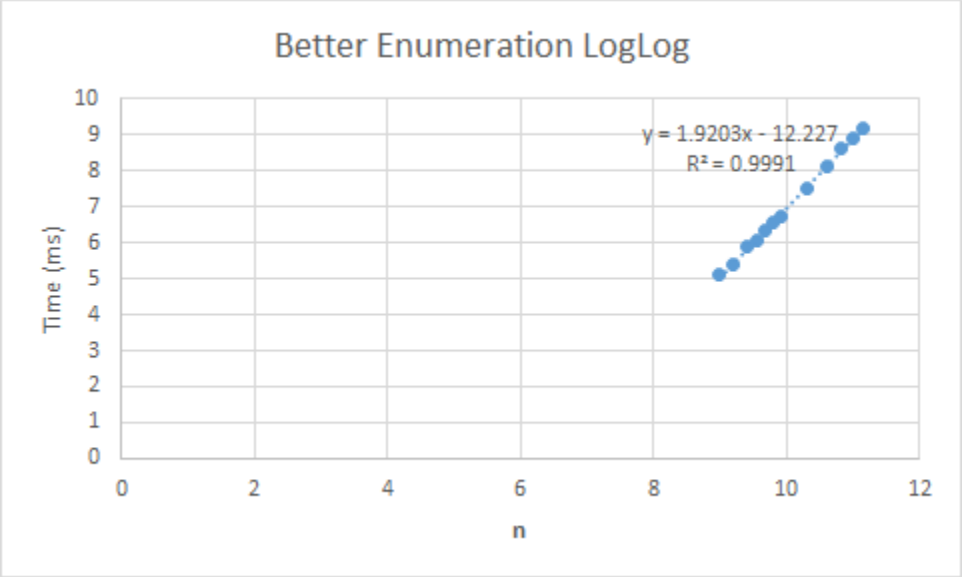
R-square: 0.9532

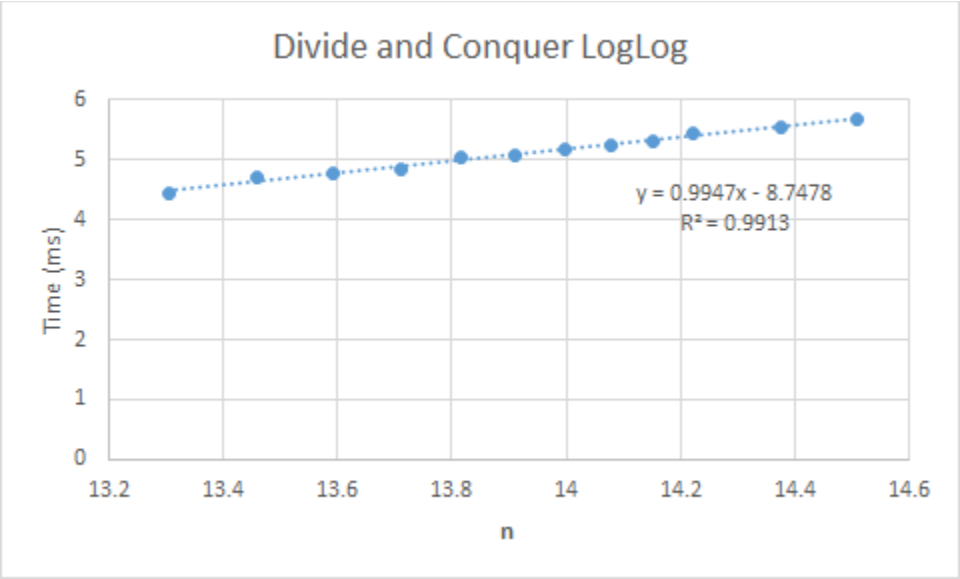
Adjusted R-square: 0.9485

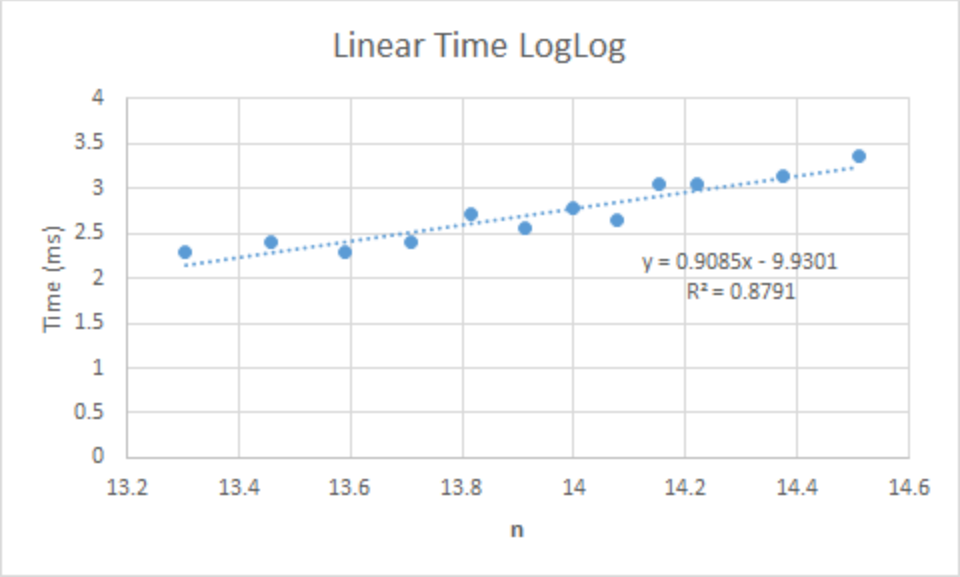
RMSE: 1.375

log-log plot of the running times.






Divide and Conquer LogLog
Time (ms)
n
y = 0.9947x - 8.7478
R² = 0.9913
0
1
2
3
4
5
6
13.2
13.4
13.6
13.8
14
14.2
14.4
14.6


Linear Time LogLog
Time (ms)
n
y = 0.9085x - 9.9301
R² = 0.8791
0
0.5
1
1.5
2
2.5
3
3.5
4
13.2
13.4
13.6
13.8
14
14.2
14.4
14.6

Single graph of all four algorithms