Project 1: Maximum Sum Subarray

Group 46

Group Members

David Brouillette
James Fitzwater
James Stallkamp

Theoretical Run-time Analysis

Enumeration

Pseudo-Code

```
on enumerationAlgo(array)
    set max to 0
    set sumSoFar to 0
    set start to 0
    set end to 0

    repeat with i from 0 to length of array by 1
        repeat with j from i to length of array by 1
            repeat with k from i to j by 1
                set sumSoFar to sumSoFar + array[k]
                if sumSoFar > max
                    set max to sumSoFar
                    set start to i
                    set end to k
                    set sumSoFar to 0
                end if
            end repeat
        end repeat
    end repeat

    return max, array[start, end]
end max_sub1
```

Asymptotic Runtime

O(n^3). There are three(3) nested loops. This causes each loop to run exponentially more times then the loop it is inside of.
This makes the T(n)=(n(n+1)(n+2))/6 = O(n^3)

Better Enumeration

Pseudo-Code

```
on betterEnumerationAlgo(array)
    set max to 0
    set start to 0
    set end to 0

    repeat with i from 0 to length of array by 1
        set sumSoFar to 0
        repeat with j from i to length of array by 1
            set sumSoFar to sumSoFar + array[j]
            if sumSoFar > max
                set max to sumSoFar
                set start to i
                set end to j
            end if
        end repeat
        set sumSoFar to 0
    end repeat

    return max, array[start, end]
end max_sub2
```

Asymptotic Runtime

O(n^2). There are two(2) nested loops, each running from a random number to the end of the array.
This makes T(n)=n(n+1)/2 = O(n^2)

Divide & Conquer

Pseudo-Code

```
on getMaxCrossingSubarray(array, low, mid, high)
    set leftSum to -infinity
    set sum to 0
    set maxLeft to NaN

    repeat with i from mid to low by -1
        sum = sum + (item i of array)
        if sum > leftSum then
            leftSum = sum
            maxLeft = i
        end if
    end repeat

    set rightSum to -infinity
    set sum to 0
    set maxRight to none

    repeat with j from mid + 1 to high
        sum = sum + (item j of array)
        if sum > rightSum then
            rightSum = sum
            maxRight = j
        end if
    end repeat

    return {maxLeft, maxRight, leftSum + rightSum}
end getMaxCrossingSubarray
```

```
on divideConquerAlgo(array, low, high)
    # Base Case of 1 element array
    if high is equal to low then
        return {low, high, item low of array}
    else
        set mid to int(round ((low + high) / 2) rounding down)
        # return list for left half
        set {leftLow, leftHigh, leftSum} to
          getMaxSubarray(array, low, mid)
        # return list for right half
        set {rightLow, rightHigh, rightSum} to
          getMaxSubarray(array, mid + 1, high)
        # return list for crossing
        set {crossLow, crossHigh, crossSum} to
          getMaxCrossingSubarray(array, low, mid, high)

        # subarray on left has greatest sum
        if leftSum ≥ rightSum and leftSum ≥ crossSum then
            return {leftLow, leftHigh, leftSum}
        # subarray on right has greatest sum
        else if rightSum ≥ leftSum and rightSum ≥ crossSum
then
            return {rightLow, rightHigh, rightSum}
        # subarray on across middle has greatest sum
        else
            return {crossLow, crossHigh, crossSum}
        end if
    end if
end getMaxSubarray
```

Asymptotic Runtime

O(n•lg(n)). This algorithm's tree has a depth of lg(n) due to the input being divided by half until a single element input is reached. Each level or step has a sum up to n as the input is iterated over. Since the previous addition results are saved for use by the following n-th element, an addition of 1 is used at each iteration. This yields a constant time work.

This makes the running time across lg(n) • n iterations yield O(n•lg(n))

Linear Time

Pseudo-Code

```
on linearAlgo(array):
    set maxHere to 0
    set maxSoFar to 0
    set sum to 0
    set start to 0
    set end to 0
    set startFinal to 0
    set endFinal to 0

    repeat with x from 0 to array length
        if array[x] > maxHere + array[x]
            set maxHere to array[x]
            set start to x
        else
            set maxHere to maxHere + array[x]
            set end to x
        end if

        if maxSoFar < maxHere
            set maxSoFar to maxHere
            set startFinal to start
            set endFinal to end
        end if
    end repeat

    return maxSoFar, array[startFinal, endFinal]
end max_sub4
```

Asymptotic Runtime

Since this algorithm only loops through the array one(1) time and there are no recursive calls, this makes T(n)=n. This is a linear relationship.

Testing

Process

In order to test the validity of our algorithms we ran unit tests on each of the algorithms. To test we were given some test sets with corresponding solutions, these along with some additional random sets were used for the unit testing. Each algorithm was applied to each test set and the output verified to be correct. All of these results are piped to an output file and checked to be the same as the known correct values. After testing all of our results were verified to be correct.

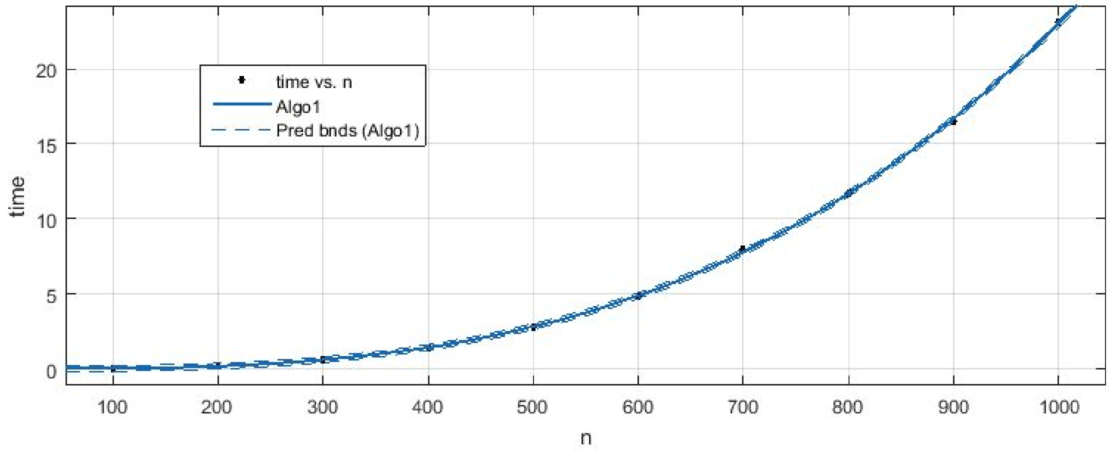
Experimental Analysis

Enumeration

1.) Average Running Time Of Each n

Size of n	Average Run Time (seconds) over 10 attempts for each n on Flip Server using Python 3
100	0.021193265914916992
200	0.16019892692565918
300	0.5201356410980225
400	1.2899384498596191
500	2.617353916168213
600	4.655832529067993
700	7.492222547531128
800	11.304193258285522
900	16.228641510009766
1000	22.396023988723755

2.) Plot of Average Running Times



3.) Functional Relationship Model: Input Size and Time

The following was obtained by analysis with Matlab:

General model Power1:

f(x) = a*x^b

Coefficients (with 95% confidence bounds):

a = 1.783e-08 (1.288e-08, 2.279e-08)

b = 3.037 (2.996, 3.078)

4.) Discrepancies Between Running times

A theoretical n•lg(n) algorithm tends to look linear for a narrow range in practice as the lg(n) part has a relatively insignificant effect on curve of the data plot. For example, lg(1000) = 3 while lg(10000) = 4. For such a range of n, lg(n) has an increase of only 1. For this reason it is practically constant.

5.) Regression Model For Largest Input (n) To Be Solved In:

10 Seconds

n = (10-(3.037))/1.783e-08 = 390521592

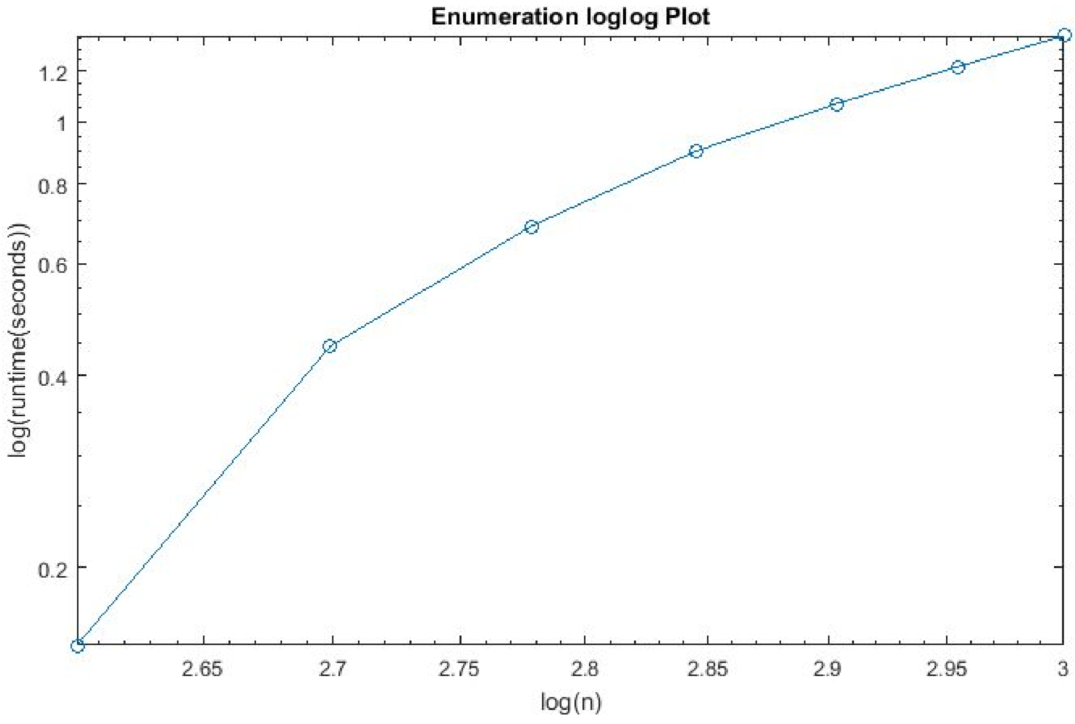
30 Seconds

n = (30-(3.037))/1.783e-08 = 1512226584

60 Seconds

n = (60-(3.037))/1.783e-08 = 3194784071

6.) Log-Log Plot

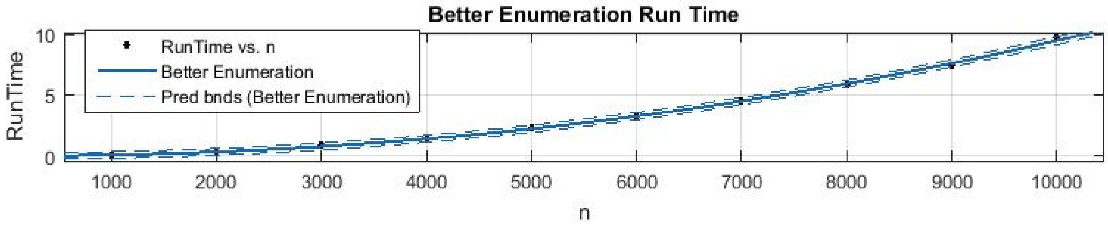


Better Enumeration

1.) Average Running Time Of Each n

Size of n	Average Run Time (seconds) over 10 attempts for each n on Flip Server using Python 3
1000	0.0912926197052002
2000	0.3655087947845459
3000	0.8242120742797852
4000	1.4552950859069824
5000	2.2845561504364014
6000	3.3122377395629883
7000	4.47594428062439
8000	5.88450026512146
9000	7.3693766593933105
10000	9.660614252090454

2.) Plot of Average Running Times



3.) Functional Relationship Model: Input Size and Time

The following was obtained by analysis with Matlab:

General model Power1:

f(x) = a*x^b

Coefficients (with 95% confidence bounds):

a = 4.32e-08 (6.448e-09, 7.995e-08)

b = 2.085 (1.991, 2.179)

4.) Discrepancies Between Running times

A theoretical n•lg(n) algorithm tends to look linear for a narrow range in practice as the lg(n) part has a relatively insignificant effect on curve of the data plot. For example, lg(1000) = 3 while lg(10000) = 4. For such a range of n, lg(n) has an increase of only 1. For this reason it is practically constant.

5.) Regression Model For Largest Input (n) To Be Solved In:

10 Seconds

n = (10-(2.085))/4.32e-08 = 183217592

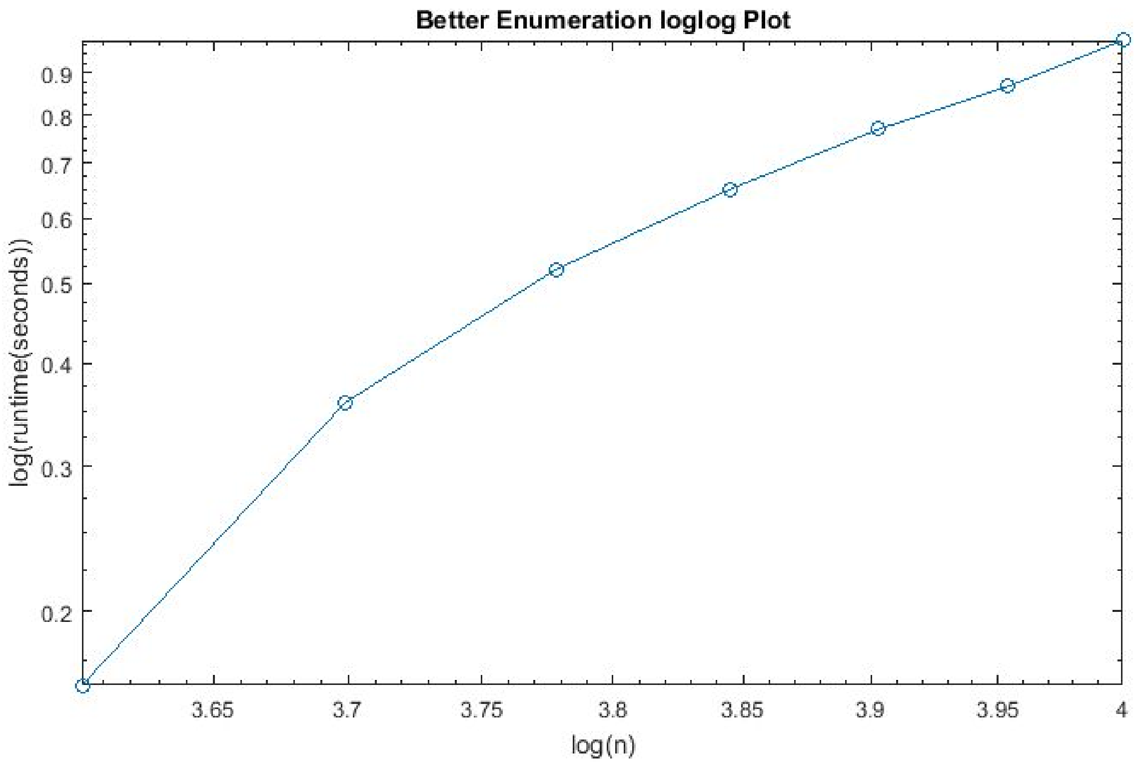
30 Seconds

n = (30-(2.085))/4.32e-08 = 646180555

60 Seconds

n = (60-(2.085))/4.32e-08 = 1340625000

6.) Log-Log Plot

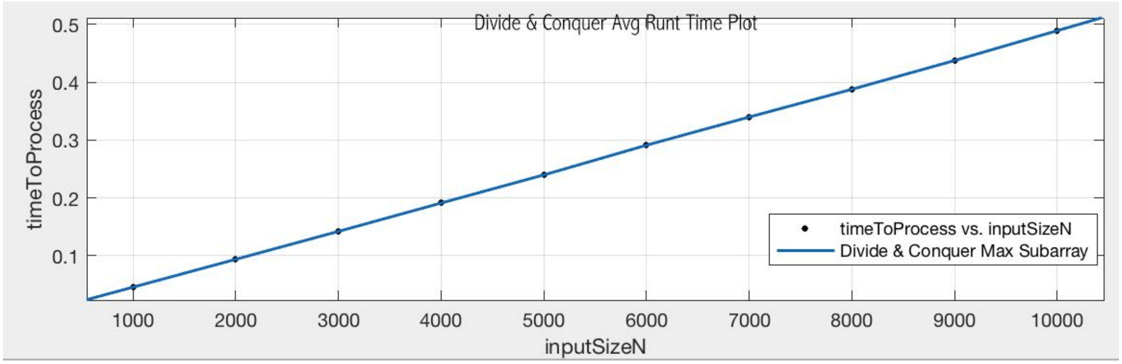


Divide & Conquer

1.) Average Running Time Of Each n

Size of n	Average Run Time (seconds) over 10 attempts for each n on Flip Server using Python 3
1000	0.045655012
2000	0.093781519
3000	0.142104983
4000	0.191171241
5000	0.239782643
6000	0.291201305
7000	0.33992455
8000	0.387874103
9000	0.437656474
10000	0.489305091

2.) Plot of Average Running Times



3.) Functional Relationship Model: Input Size and Time

The following was obtained by analysis with Matlab:

Linear model Poly1:

f(n) = p1*n + p2

Coefficients (with 95% confidence bounds):

p1 = 4.925e-05 (4.896e-05, 4.955e-05)

p2 = -0.005038 (-0.00686, -0.003216)

4.) Discrepancies Between Running times

A theoretical n•lg(n) algorithm tends to look linear for a narrow range in practice as the lg(n) part has a relatively insignificant effect on curve of the data plot. For example, lg(1000) = 3 while lg(10000) = 4. For such a range of n, lg(n) has an increase of only 1. For this reason it is practically constant.

5.) Regression Model For Largest Input (n) To Be Solved In:

10 Seconds

n = (10-(-0.005038))/4.925e-05 = 203147

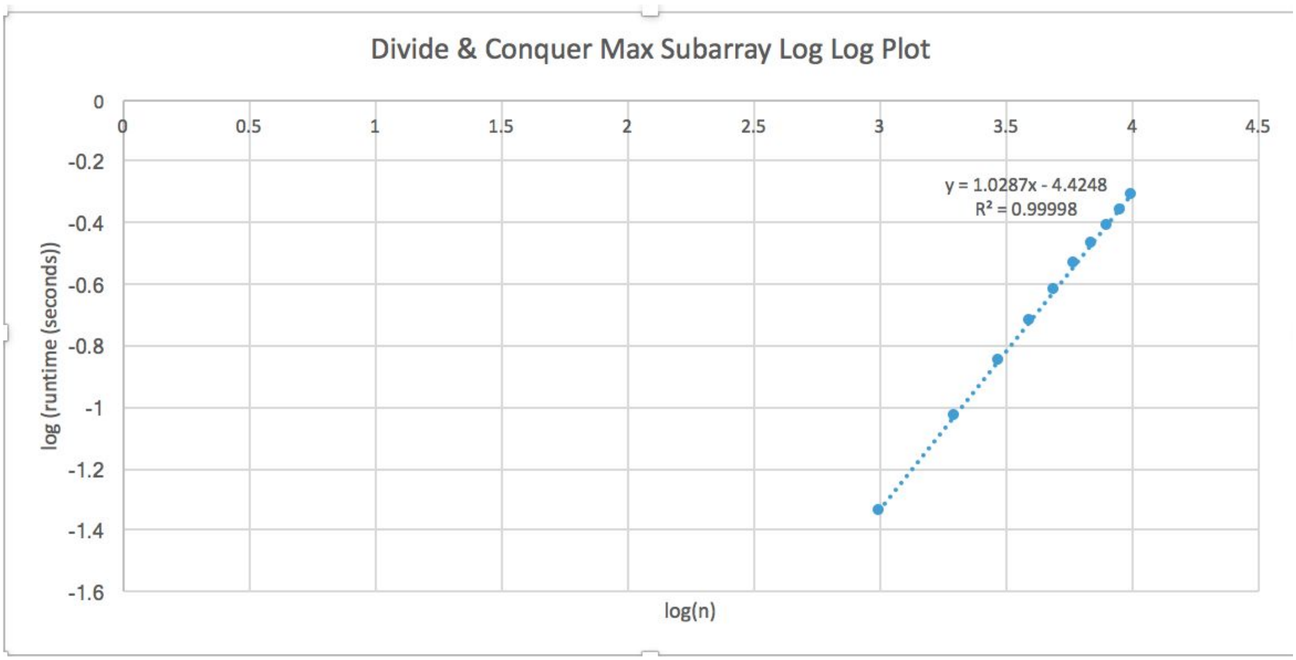
30 Seconds

n = (30-(-0.005038))/4.925e-05 = 609239

60 Seconds

n = (60-(-0.005038))/4.925e-05 = 1218376

6.) Log-Log Plot

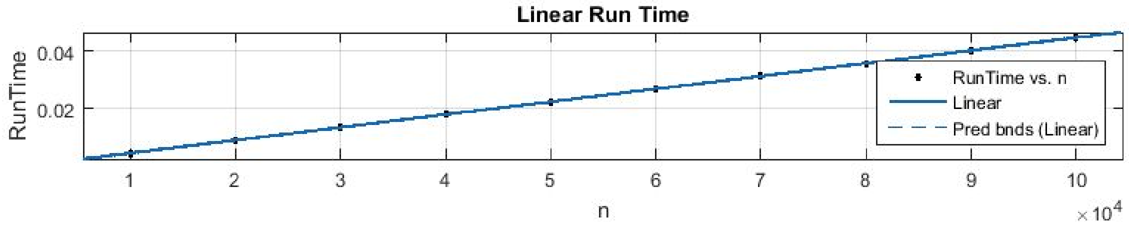


Linear Time

1.) Average Running Time Of Each n

Size of n	Average Run Time (seconds) over 10 attempts for each n on Flip Server using Python 3
10000	0.004388093948364258
20000	0.008914470672607422
30000	0.013427257537841797
40000	0.017940998077392578
50000	0.02219557762145996
60000	0.026788949966430664
70000	0.03158879280090332
80000	0.03575849533081055
90000	0.04043126106262207
100000	0.04465150833129883

2.) Plot of Average Running Times



3.) Functional Relationship Model: Input Size and Time

The following was obtained by analysis with Matlab:

Linear model Poly1:

f(x) = p1*x + p2

Coefficients (with 95% confidence bounds):

p1 = 4.486e-07 (4.452e-07, 4.52e-07)

p2 = -6.417e-05 (-0.0002739, 0.0001456)

4.) Discrepancies Between Running times

A theoretical n•lg(n) algorithm tends to look linear for a narrow range in practice as the lg(n) part has a relatively insignificant effect on curve of the data plot. For example, lg(1000) = 3 while lg(10000) = 4. For such a range of n, lg(n) has an increase of only 1. For this reason it is practically constant.

5.) Regression Model For Largest Input (n) To Be Solved In:

10 Seconds

n = (10-(-6.417e-05))/4.486e-07 = 22291716

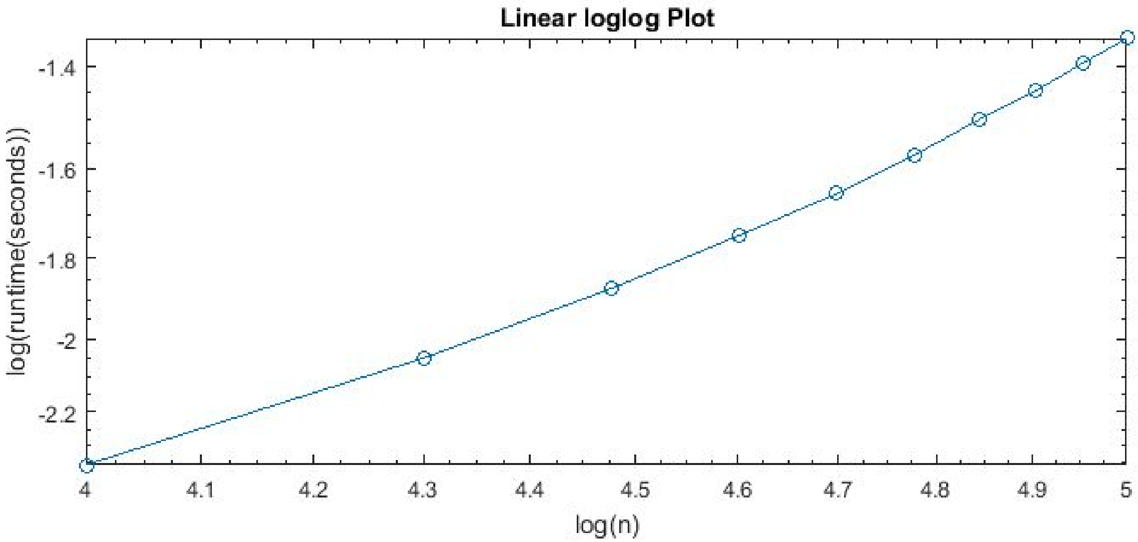
30 Seconds

n = (30-(-6.417e-05))/4.486e-07 = 66874864

60 Seconds

n = (60-(-6.417e-05))/4.486e-07 = 133749585

6.) Log-Log Plot



Graph Presenting loglog Plot of All Four(4) Algorithms