Project One: Maximum Sub-Array Summation

Group 12

Group Members

- Kyle Guthrie
- Michael C. Stramel
- Alex Miranda

Enumeration

Pseudo-code

The "Enumeration" maximum sub-array algorithm is described by the following pseudo-code:

```
ENUMERATION-MAX-SUBARRAY(A[1,...,N]) {
    if N == 0 {
        return 0, A
    } else {
        max_sum = -Infinity
    }

    for i from 1 to N {
        for j from i to N {
            current_sum = 0
            for k from i to j {
                current_sum = current_sum + A[k]
                if current_sum > max_sum {
                    max_sum = current_sum
                    start_index = i
                    end_index = j
                }
            }
        }
    }
    return max_sum, A[start_index, ..., end_index]
}
```

Theoretical Run-time Analysis

The outer i loop runs from 1 to N, the first inner j loop runs from i to N, and the second inner loop runs from i to j. We can compute the number of iterations as:

- $\sum_{i=1}^{N} \sum_{j=i}^{N} \sum_{k=i}^{j} \Theta(1)$
- $\sum_{i=1}^{N} \sum_{j=i}^{N} (j - i + 1)\Theta(1)$
- $\sum_{i=1}^{N} (\sum_{j=i}^{N} (1 - i) + \sum_{j=i}^{N} j)\Theta(1)$
- $\sum_{i=1}^{N} ((i - 1)(i - N - 1) - \frac{1}{2}(i + N)(i - N - 1))\Theta(1)$
- $\sum_{i=1}^{N} ((i^2 - iN - 2i + N + 1) - \frac{1}{2}(i^2 - i - N^2 - N))\Theta(1)$
- $\sum_{i=1}^{N} \frac{1}{2}i^2 - iN - \frac{3}{2}i + \frac{1}{2}N^2 + \frac{3}{2}N + 1)\Theta(1)$
- $\sum_{i=1}^{N} \frac{1}{2}(i^2 - 2iN - 3i) + \sum_{i=1}^{N} \frac{1}{2}(N^2 + 3N + 2)\Theta(1)$

So we can find the sums term by term for the terms with i while the sum of terms without i will remain constant therefore:

- $\sum_{i=1}^{N} \frac{1}{2}(N^2 + 3N + 2) = (\frac{1}{2}N^2 + \frac{3}{2}N + 1) \sum_{i=1}^{N} 1$
- $(\frac{1}{2}N^2 + \frac{3}{2}N + 1) \sum_{i=1}^{N} 1 = (\frac{1}{2}N^2 + \frac{3}{2}N + 1) * N$
- $(\frac{1}{2}N^2 + \frac{3}{2}N + 1) * N = \frac{1}{2}N^3 + \frac{3}{2}N^2 + N$
- $\sum_{i=1}^{N} \frac{1}{2}i^2 = \frac{1}{2}(\frac{1}{6}N(N + 1)(2N + 1))$
- $\frac{1}{2}(\frac{1}{6}N(N + 1)(2N + 1)) = \frac{1}{6}N^3 + \frac{1}{4}N^2 + \frac{1}{12}N$
- $\sum_{i=1}^{N} \frac{1}{2}(-2iN) = -\frac{1}{2}N(N + 1) * N = -\frac{1}{2}(N^3 + N^2)$
- $\sum_{i=1}^{N} \frac{1}{2}(-3i) = \frac{1}{2}(-\frac{3}{2}N(N + 1)) = -\frac{3}{4}(N^2 + N)$

After collecting like terms:

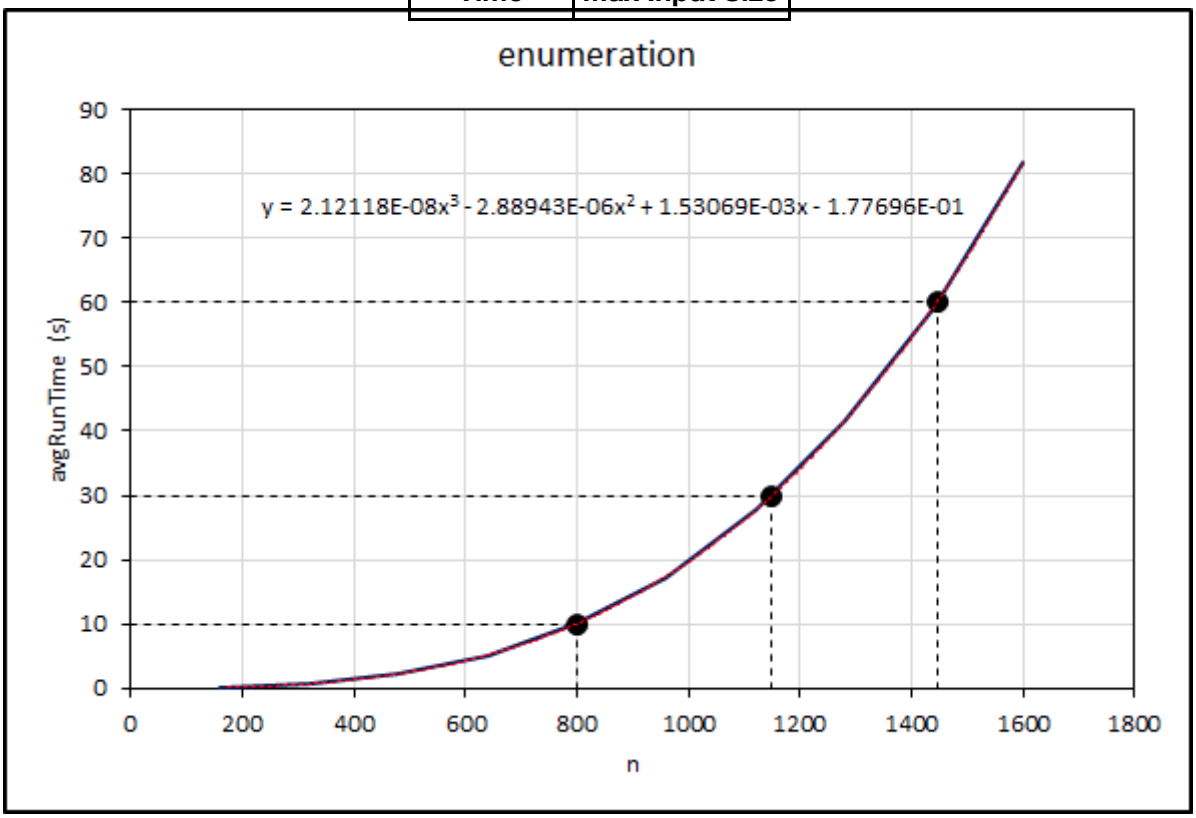
- $\frac{1}{6}N^3 + \frac{1}{2}N^2 + \frac{1}{3}N$
- $\sum_{i=1}^{N} \sum_{j=i}^{N} \sum_{k=i}^{j} \Theta(1) = \frac{1}{6}N^3 + \frac{1}{2}N^2 + \frac{1}{3}N \cdot \Theta(1)$
- $\frac{1}{6}N^3 + \frac{1}{2}N^2 + \frac{1}{3}N \cdot \Theta(1) = \Theta(N^3)$ (Because the N^3 will dominate)

Thus the runtime of the whole algorithm is equivalent to $\Theta(N^3)$.

Experimental Analysis

For a series of array sizes N, 10 random arrays were generated and run through the "Enumeration" algorithm. The CPU clock time was recorded for each of the 10 random array inputs, and an average run time was computed. Below is the plot of average run time versus N for the "Enumeration" algorithm.

Time	Max Input Size



The equation of the best fit curve to the runtime data where x stands in for N:

$$y = 2.12118 * 10^{-8} * x^3 + 1.53069 * 10^{-3} * x - 1.77696 * 10^{-1}$$

The highest degree of the best fit curve is three and as shown in the plot above, fits the data points very closely which stands to corroborate the theoretical run-time of $\Theta(N^3)$.

Based on the average run time data curve fit, we would expect the "Enumeration" to be able to process the following number of elements in the given amount of time:

Time	Max Input Size
10 seconds	798
30 seconds	1150
60 seconds	1445

Better Enumeration

Pseudo-Code

The "Better Enumeration" maximum sub-array algorithm is described by the following pseudo-code:

```
BETTER-ENUMERATION-MAX-SUBARRAY(A[1, ..., N])
    maximum sum = -Infinity
    for i from 1 to N
        current sum = 0
        for j from i to N
            current sum = current sum + A[j]
            if current sum > maximum sum
                maximum sum = current sum
                start index = i
                end index = j

    return maximum sum, A[start index, ..., end index]
```

Theoretical Run-time Analysis

The outer i loop runs from 1 to N, and the inner j loop runs from i to N. Inside the inner loop are constant time operations. We can compute the number of iterations of these constant time operations as:

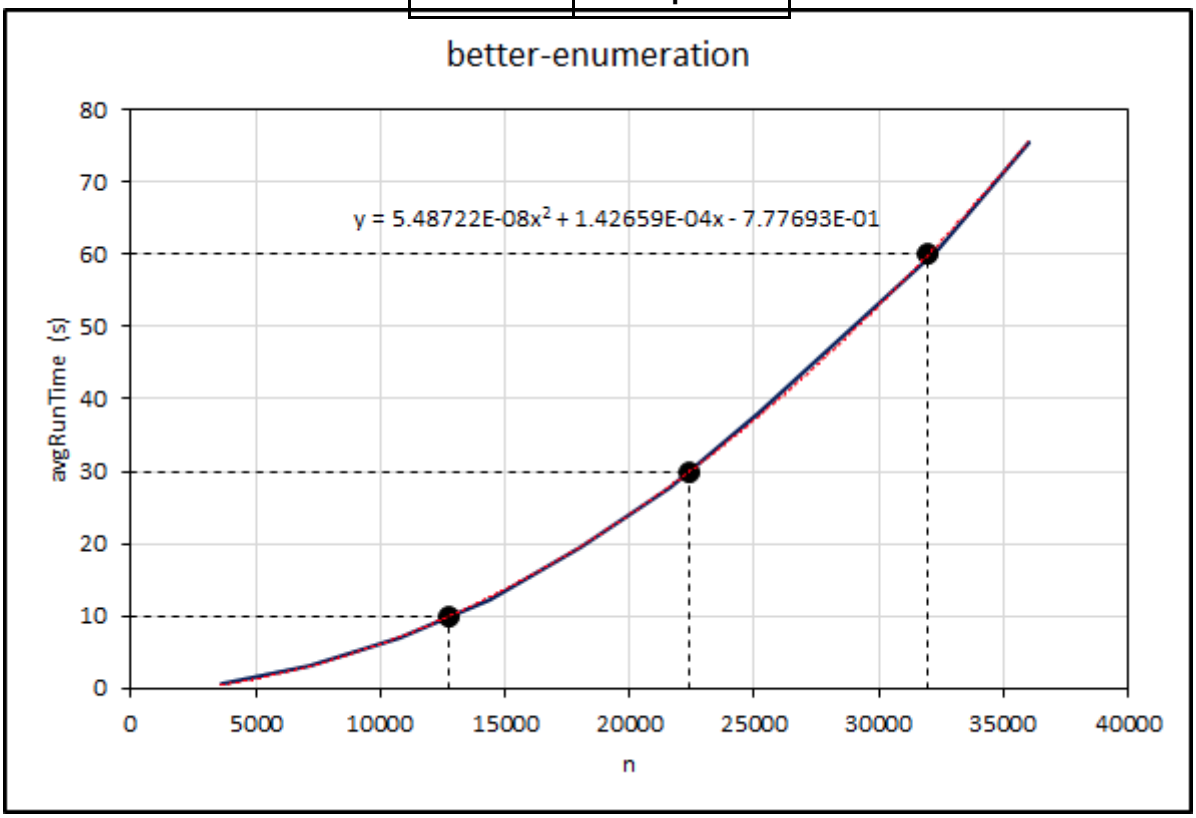
$$\sum_{i=1}^{N}\sum_{j=i}^{N}\Theta(1) = \sum_{i=1}^{N}(N+1-i)\cdot\Theta(1) = N(N+1)\cdot\Theta(1) - \frac{1}{2}N(N+1)\cdot\Theta(1)$$
$$= \frac{1}{2}N(N+1)\cdot\Theta(1) = \Theta(N^2)$$

Thus, the theoritical run-time is $\Theta(N^2)$.

Experimental Analysis

For a series of array sizes N, 10 random arrays were generated and run through the "Better Enumeration" algorithm. The CPU clock time was recorded for each of the 10 random array inputs, and an average run time was computed. Below is the plot of average run time versus N for the "Better Enumeration" algorithm.

Time	Max Input Size



A curve fit was applied to the average run time data, which resulted in the following fit equation as a function of x standing in for N:

$$y = 5.48722 * 10^{-8} * x^2 + 1.42659 * 10^{-4} * x - 7.776 * 10^{-1}$$

The fit curve for the plotted data has the same degree as the theoretical runtime of $\Theta(N^2)$ so the experimental appears to match the theoretical runtime.

Based on the average run time data curve fit, we would expect the "Better Enumeration" to be able to process the following number of elements in the given amount of time:

Time	Max Input Size
10 seconds	12775
30 seconds	22419
60 seconds	32006

Divide and Conquer

Pseudo-code

The "Divide and Conquer" maximum sub-array algorithm is described by the following pseudo-code:

```
DIVIDE_AND_CONQUER(A[1,...,N]){
    if N == 0 {
        return 0, A
    } else if N == 1 {
        return A[0], A
    }

    tmp_max = 0
    mid_max = 0
    mid_start = 0
    mid_end = 0

    left_max = 0
    right_max = 0

    midpoint = N / 2

    mid_start = midpoint
    mid_end = midpoint

    for i from A[N,...,midpoint] {
        tmp_max = tmp_max + A[i]
        if tmp_max > left_max {
            left_max = tmp_max
            mid_start = i
        }
    }
```

```
    tmp_max = 0

    for i from A[midpoint,...,N] {
        tmp_max = tmp_max + A[i]
        if tmp_max > right_max {
            right_max = tmp_max
            mid_end = i + 1
        }
    }

    mid_max = left_max + right_max

    left_max, left_subarray = DIVIDE_AND_CONQUER(A[0,...,midpoint])
    right_max, right_subarray = DIVIDE_AND_CONQUER(A[midpoint,...,N])

    if mid_max >= left_max and mid_max >= right_max {
        return mid_max, A[mid_start,...,mid_end]
    } else if left_max >= right_max and left_max > mid_max {
        return left_max, left_subarray
    } else if right_max > left_max and right_max > mid_max {
        return right_max, right_subarray
    }
}
```

Theoretical Run-time Analysis

In order to determine the run-time of the divide and conquer algorithm we will need to derive its recurrence. We will make a simplifying assumption that the original problem size is a power of two so that all of the subproblem sizes will remain integers. We will denote $T(n)$ as the run-time of "divide and conquer" on a subarray of n elements. The base case is when $n = 0$ or $n = 1$ which will take constant time and as a result:

- $T(1) = \Theta(1)$

The recursive case occurs when $n > 1$. The variable initializing prior to the first for loop will also take constant time. The following for loops are used to find the maximum sub-array that crosses the array's midpoint and will add a runtime of $\Theta(n)$. After the loops there are two recursive calls to the main function where we spend $T(\frac{n}{2})$ solving each of them so the total contribution to the running time is $2T(\frac{n}{2})$. The remaining if-else block would also run in constant time ($\Theta(1)$). So the recurrence in total for $n > 1$ is:

- $T(n) = 2T(\frac{n}{2}) + \Theta(n) + \Theta(1)$
- $T(n) = 2T(\frac{n}{2}) + \Theta(n)$

So the complete recurrence is:

- $T(n) = \Theta(1)$ (when $n = 0$ or $n = 1$)
- $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ (when $n > 1$)

The recurrence can be solved using the master method as shown below:

- $T(n) = 2T(\frac{n}{2}) + \Theta(n)$
- $a = 2, b = 2, f(n) = \Theta(n)$
- $n^{log_b(a)} = n^{log_2(2)} = n^1$
- $\Theta(n^{log_b(a)}) = \Theta(n)$ (Case 2 applies)
- $T(n) = \Theta(n^{log_2(2)} * log_2(n)) = \Theta(n * log_2(n))$ (By the master theorem)

So substituting in the runtime we get:

- $T(n) = \Theta(n * log_2(n)) + \Theta(n)$

The $\Theta(n)$ term drops off leaving us with:

- $T(n) = \Theta(n * log_2(n))$

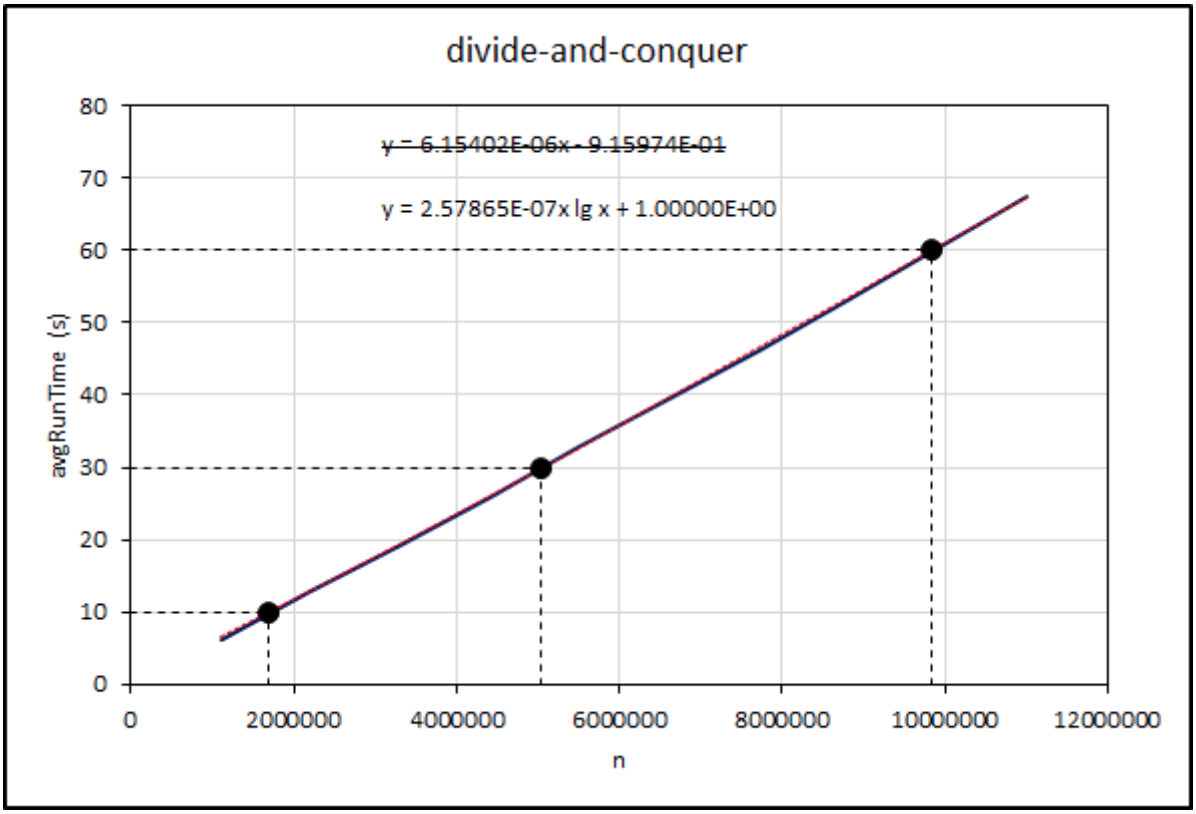
The runtime above is for the divide and conquer algorithm for finding maximum sub arrays.

Experimental Analysis

For a series of array sizes N, 10 random arrays were generated and run through the "Divide and Conquer" algorithm. The CPU clock time was recorded for each of the 10 random array inputs, and an average run time was computed. Below is the plot of average run time versus N for the "Divide and Conquer" algorithm.

Time	Max Input Size

A linear fit and logarithmic fit were applied to the average run time data, which resulted in the following fit equations as a function of N:



The function of the best logarithmic fit curve where x is substituted for N:

$$y = 2.57865 * 10^{-7} * x * log(x) + 1.000$$

The function of the best linear fit curve where x is substituted for N:

$$y = 6.15402 * 10^{-6} * x - 9.15974 * 10^{-1}$$

This plot has both a linear fit and a logarithmic fit to show how similiar they present for the values plotted. The logarithmic curve fits the data points almost exactly therefore it shows that the experimental values are strongly aligned with the theoretically derived runtime of $\Theta(n * log_2(n))$.

Based on the average run time data curve fit, we would expect the "Divide and Conquer" algorithm to be able to process the following number of elements in the given amount of time:

Time	Max Input Size
10 seconds	1687210
30 seconds	5050390
60 seconds	9848770

Linear-time

Pseudo-Code

The "Linear-time" maximum sub-array algorithm is described by the following pseudo-code:

```
LINEAR-TIME-MAX-SUBARRAY(A[1, ..., N])
    maximum sum = -Infinity
    sum ending here = -Infinity
    for i from 1 to N
        ending here high index = j
        if ending here sum > 0
            ending here sum = ending here sum + A[i]
        else
            ending here low index = j
            ending here sum = A[i]

        if ending here sum > maximum sum
            maximum sum = ending here sum
            start index = ending here low index
            end index = ending here high index

    return maximum sum, A[start index, ..., end index]
```

Theoretical Run-time Analysis

The i loop runs from 1 to N. Inside the loop are constant time operations. We can compute the number of iterations of these constant time operations as:

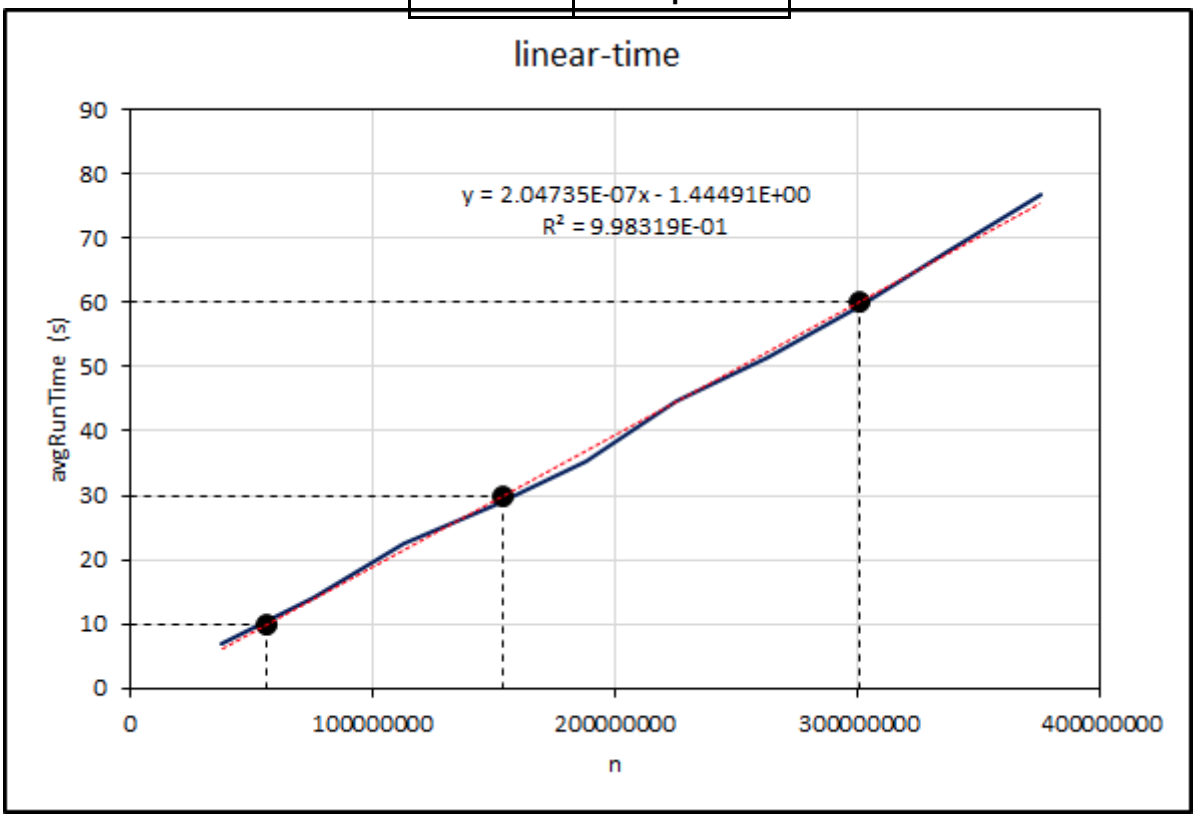
$$\begin{aligned}\sum_{i=1}^{N} \Theta(1) &= N \cdot \Theta(1) \\ &= \Theta(N)\end{aligned}$$

Thus, the theoritical run-time is $\Theta(N)$.

Experimental Analysis

For a series of array sizes N, 10 random arrays were generated and run through the "Linear-time" algorithm. The CPU clock time was recorded for each of the 10 random array inputs, and an average run time was computed. Below is the plot of average run time versus N for the "Linear-time" algorithm.

Time	Max Input Size



A curve fit was applied to the average run time data, which resulted in the following fit equation as a function of x standing in for N:

$$y = 2.04735 * 10^{-7} * x - 1.4449$$

The best fit curve fits the plotted data extremely well, showing that the runtimes reflect a linear trend. The observed linear trend in the data matches with the theoretically derived runtime of $\Theta(n)$.

Based on the average run time data curve fit, we would expect the "Linear-time" algorithm to be able to process the following number of elements in the given amount of time:

Time	Max Input Size
10 seconds	55901100
30 seconds	153588000
60 seconds	300119000

Testing

Several sets of test data were used to ensure the accuracy of each of the algorithms. These test data were comprised of arrays of values with known maximum sub-array solutions. These test sets were run through each algorithm, and the output was compared to the known solution.

Along with the provided set of test cases, several additional test cases were generated in order to test the algorithm accuracy under very specific conditions. Some examples of these test cases include:

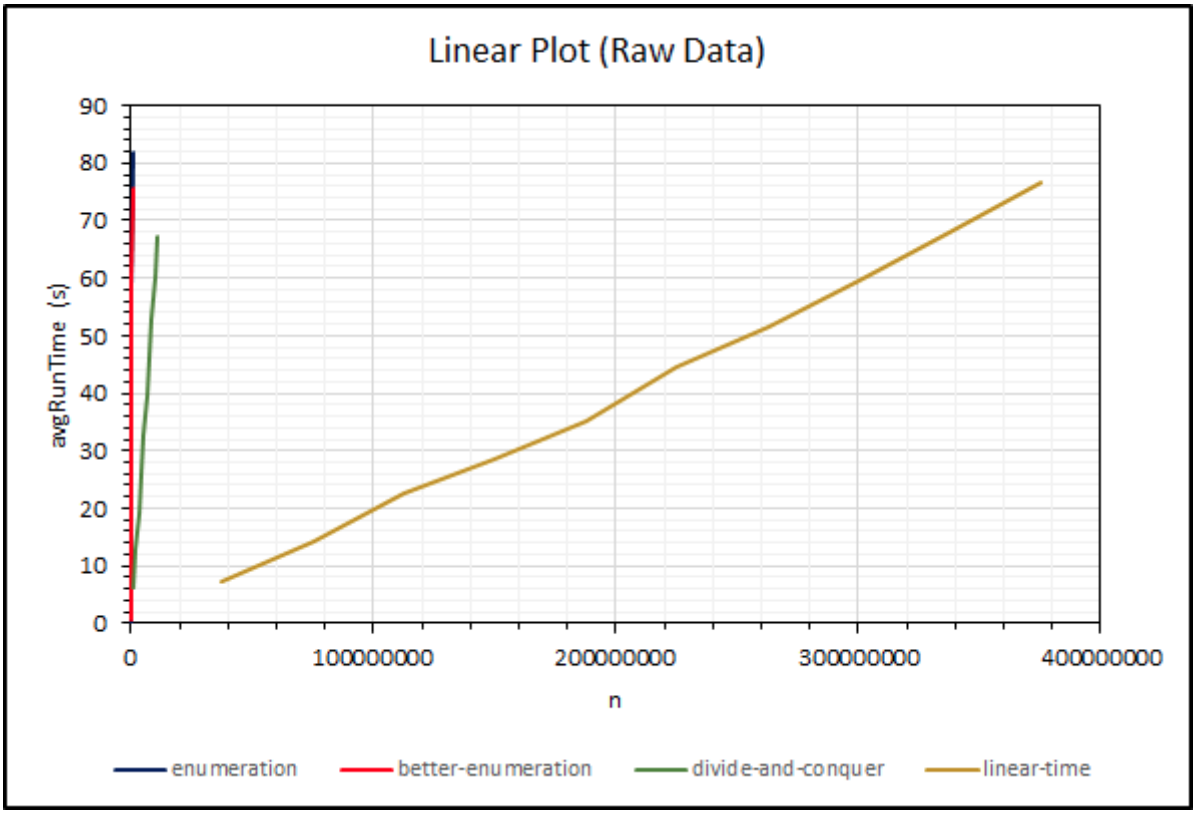
- The trivial case of a single array element
- Arrays with a single positive value as the first or last element (to test the handling of the boundaries)
- Arrays with a single positive value in the middle of the arrary
- Arrays where the running sum reaches 0 at some point (i.e. multiple maximum sum sub-arrays possible)

All algorithms correctly solved all of the test data sets.

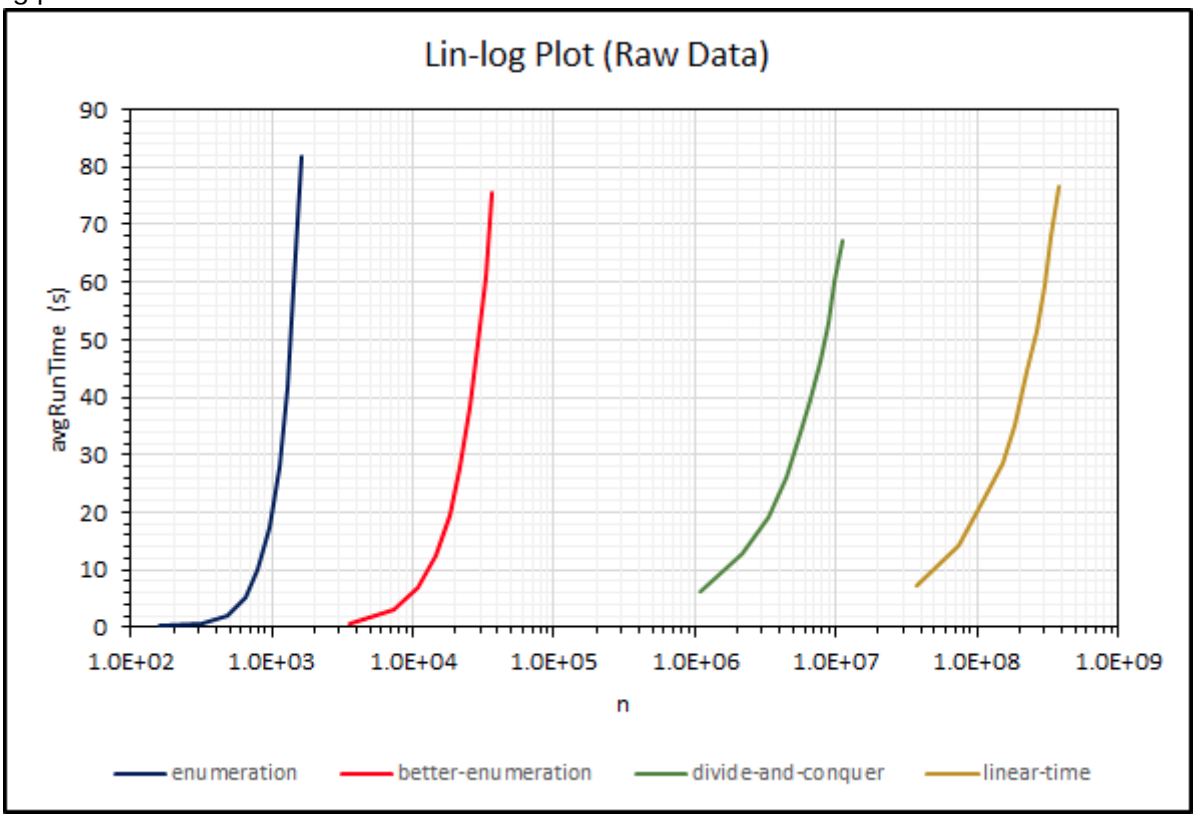
Algorithm Comparison Plots

Three plots were generated with various combinations of linear and log scales to show the performance of the various algorithms together.

Linear plot:



Linear Log plot:



Log Log plot: