Assignment: Project 1
Class: CS325-400
Group: 6
Date: 2016.10.06
Due Date: 2016.10.16
Group Members: Kelsey Helms, Jay Steingold, Johannes Pikel

Pseudo Code and Theoretical run-time Analysis

Pseudo Code: Enumeration, Algorithm 1

```
enumeration(array[1...n])
        max_sum and new_max_sum ⬅ -inf
        low_index and high_index ⬅ 0

        if length of array >= 1
                for outer_index from 0 to n
                        for  inner_index from outer_index + 1 to n
                                new_max_sum is the sum (array[outer_index] to
array[inner_index]

                                if new_max_sum > max_sum
                                        new_max_sum assigned to max_sum
                                        low_index is the current outer_index
                                        high_index is the current inner_index

                return array[low_index...high_index], max_sum
        else
                return no_max_sum
```

Recurrence: Enumeration T(n) = n * n * n + $\Theta(1)$
Asymptotic runtime: Enumeration $T(n) = O(n^3)$, easy to see that n*n*n results in the asymptotic bound and the $\Theta(1)$ can be disregarded for large n.

Pseudo Code: Iteration, Algorithm 2

```
iteration(array[1...n])
        max_sum and new_max_sum ⬅ -inf
        low_index and high_index ⬅ 0

        if array length > 1
                for outer_index from 0 to n
                        new_max_sum ⬅ 0

                                for inner_index from outer_index to n
                                        new_max_sum = new_max_sum + array[inner_index]
                                if new_max_sum > max_sum
                                        max_sum ⬅ new_max_sum
```

```
                              low_index ← outer_index
                              high_index ← inner_index
                return array[low_index...high_index], max_sum
        else if array length is 1
                max_sum ← array[0]
                return array, max_sum
        else
                return no_max_sum
```

Recurrence: Iteration

$T(n) = (O(n)\ outerindex\ iterations) * (O(n)\ innerindex\ iterations) * O(1)$

Asymptotic runtime: Iteration $T(n) = O(n^2)$, due to the two nested for loops that both iterate through the array n times and O(1) can be disregarded for large n.

Pseudo Code: Divide and Conquer, Algorithm 3

Cite: Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford. Introduction to Algorithms (Page 72). The MIT Press. Kindle Edition.

Using a helper function to find the max crossing subarray

```
max_crossing_array(array, low, mid, high)
        Left_sum = right_sum = -inf
        sum = 0
        for i = mid down to  low
                sum = sum + array[i]
                if sum > left_sum
                max_left = i
        sum = 0
        for j = mid + 1 to high
                sum = sum + A[j]
                if sum > right_sum
                        right_sum = sum
                        max_right = j
        return max_left, max_right, (left_sum + right_sum)

find_max_subarray(array, low, high)
        if high == low
                return low, high, array[low]
        else
```

```
        mid = floor((low + high) / 2)
        left_low, left_high, left_sum = find_max_subarray(array, low, mid)
        right_low, right_high, right_sum = find_max_subarray(array, mid + 1, high)
        cross_low, cross_high, cross_sum = max_crossing_array(array, low, mid, high)
        if left_sum >= right_sum and left_sum >= cross_sum
            return left_low, left_high, left_sum
        else if right_sum >= left_sym and right_sum >= cross_sum
            return right_low, right_high, right_sum
        else
            return cross_low, cross_high, cross_sum
```

With implementation in order to effectively write the results to file a function was used to call the initial call to find_max_subarray()

```
recursive(array)
    low, high, sum = fiind_max_subarray(array, 0, len(array) -1)
    return array[low ... high], sum
```

Recurrence: Divide and Conquer $T(n) = 2 * T(n/2) + \Theta(n) + \Theta(1) + 1$ (for the recursive())

Asymptotic runtime: Divide and Conquer
Using the master method then with a = 2, b = 2, and f(n) = n
$Compare\ n^{log_2 2} = n^1 = n\ with\ f(n) = n$
Then by case 2: $if\ f(n) = \Theta(n^{log_b n})$ then: $T(n) = \Theta(n^{log_2 2} log_2 n = \Theta(n\ log_2 n)$

Hence the asymptotic runtime is $\Theta(n\ log_2 n)$

Pseudo Code: Recursion Inversion (Linear time), Algorithm 4

Cite: Based on Kadan's algorithm:
https://en.wikipedia.org/wiki/Maximum_subarray_problemh

```
recursion_inversion(array)
    If array length == 0
        return array, "No max sum"
        maxMax = currentMax = array[0]
    Idx = 0
    Low_idx = high_idx = 0

    For index  in array length
        If array[index] > (currentMax + array[index])
```

```
                currentMax = array[index]
                Idx = index
        else
                currentMax += array[index]
        If currentMax > maxMax:
                maxMax = currentMax
                Low_idx = idx
                High_idx = index
    Return array[low_idx to high_idx + 1], maxMax
```

Recurrence: Linear time

$T(n) = n + O(1) + 1$ for constant work

Asymptotic runtime: Linear Time

As n increases, the constant matters less. Since there is nothing altering n, it remains the same.

$T(n) = O(n$ elements to iterate over$) = O(n)$

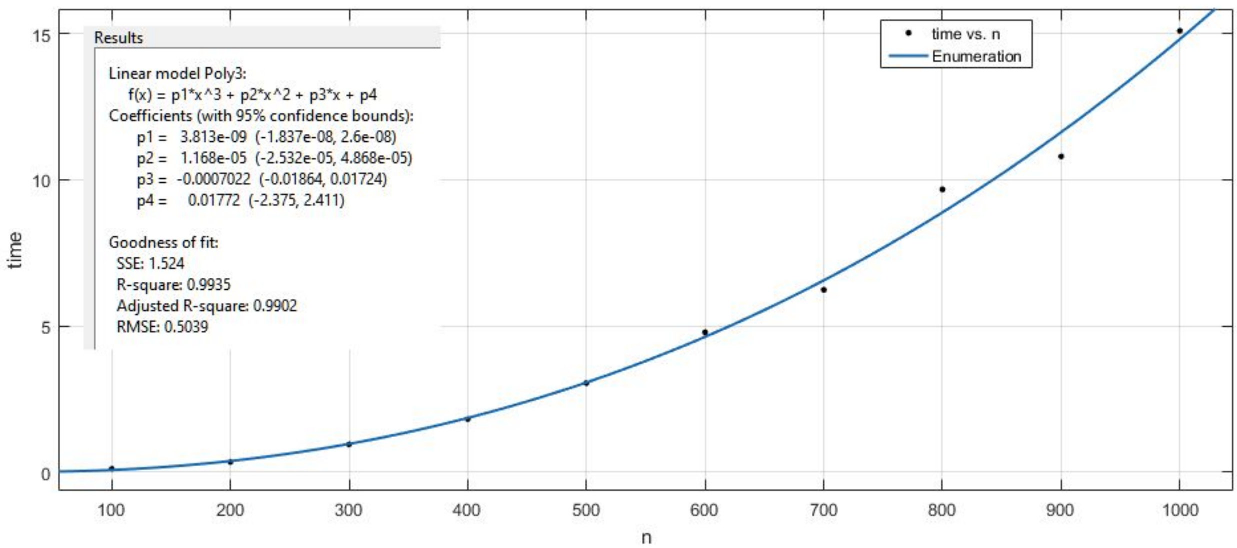
Testing

- When testing each of the algorithms for correctness, first tested each algorithm with MSS_TestProblems.txt to verify that it worked properly for that set of given arrays.
- Created boundary cases using arrays of size n = 1, to ensure we had base cases correct when there was nothing to compare against.
- Tested arrays that contained only negative numbers.
- Tested arrays that contained all zeros with only 1 other integer, either positive or negative.
- With each test case we inspected the MSS_Results.txt that was created to ensure the algorithms worked as intended.

Assignment: Project 1
Class: CS325-400
Group: 6
Date: 2016.10.06
Due Date: 2016.10.16
Group Members: Kelsey Helms, Jay Steingold, Johannes Pikel

Experimental Analysis

#1 Enumeration:

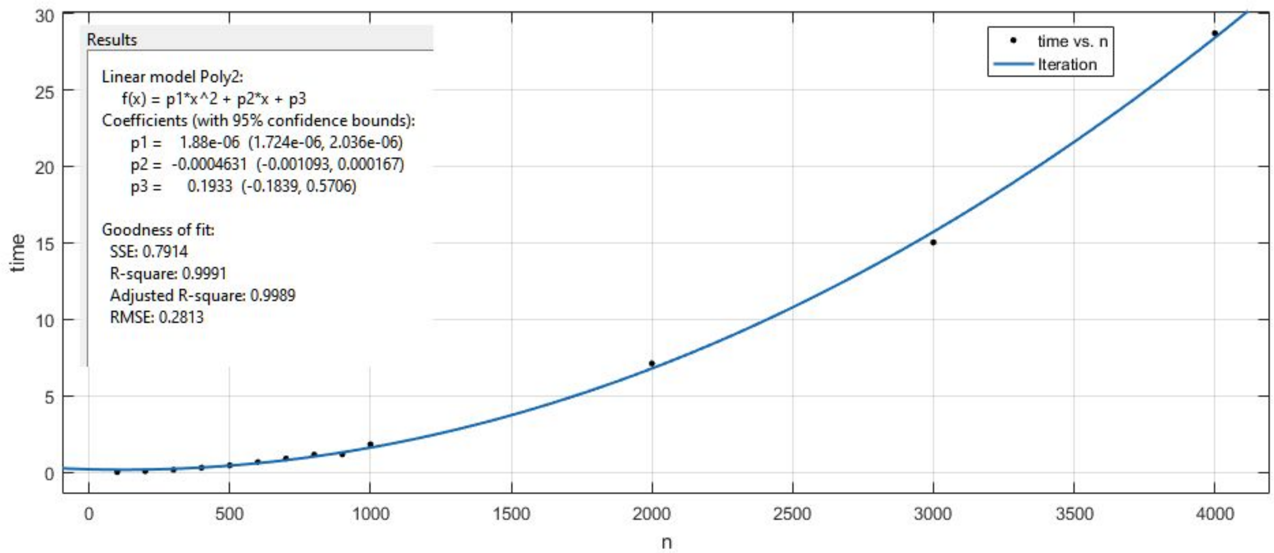


Equation #1 for regression curve: $a * n^3 + b * n^2 + c * n + d,\ where\ a, b, c, d\ are\ constants$

Enumeration	
n	**Average Time**
100	**0.105960677**
200	**0.341997607**
300	**0.943891832**
400	**1.808750487**
500	**3.040338509**
600	**4.780747184**
700	**6.232254964**
800	**9.669896875**

900	**10.79555459**
1000	**15.09172075**

#2 Iteration:

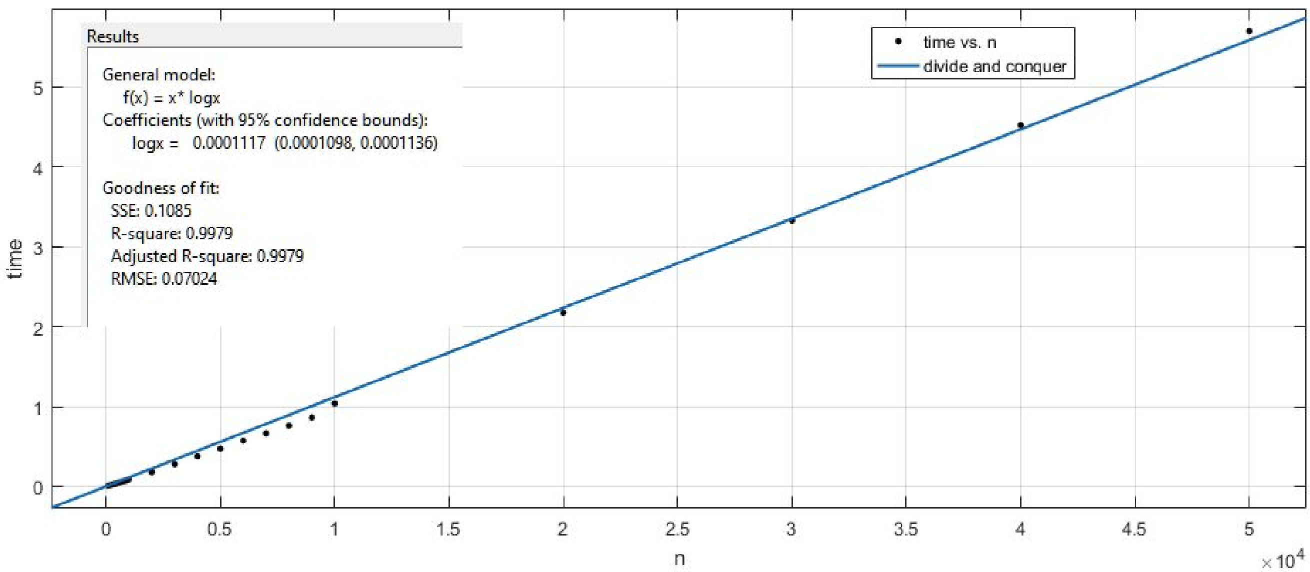


Equation #2 for regression curve: $a * n^2 + b * n + c,\ where\ a,\ b,\ c\ are\ constants$

Iteration			
n	**Average Time**	**n**	**Average Time**
100	0.020099468	1000	1.817726205
200	0.073781952	2000	7.110037233
300	0.163660474	3000	15.03335721
400	0.294904714	4000	28.71316899
500	0.454518362		
600	0.660570064		
700	0.888529375		

800	1.151724451		
900	1.174364603		

#3 Divide and Conquer:

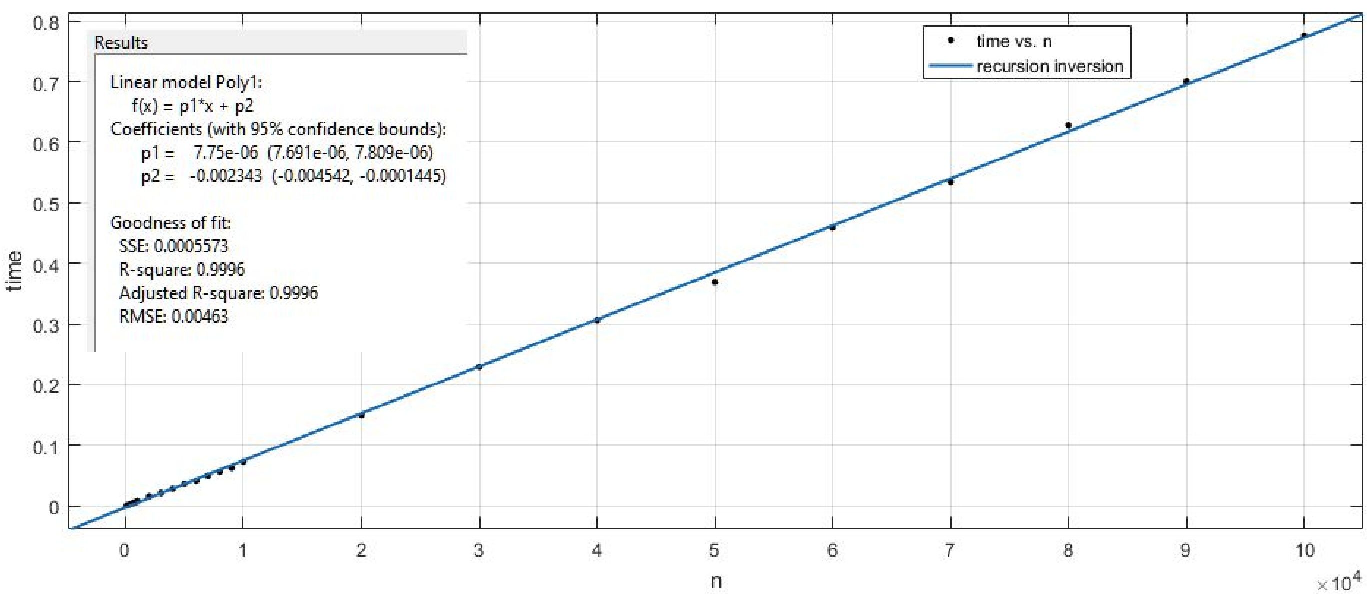


Equation #3 for regression curve: $n \log n$

Divide and Conquer			
n	**Average Time**	**n**	**Average Time**
100	0.00828156	7000	0.663376374
200	0.017074951	8000	0.759921519
300	0.025133213	9000	0.861398635
400	0.032304606	10000	1.037731206
500	0.041581789	20000	2.171978121
600	0.050818943	30000	3.325826343
700	0.05844892	40000	4.519081552
800	0.068020385	50000	5.69710066
900	0.076997825		
1000	0.086130618		

2000	0.178267778		
3000	0.280500386		
4000	0.377630112		
5000	0.471748008		
6000	0.570816677		

#4 Linear (DP)



Equation #4 for regression curve: $a * n + c$, *where a and c are constants*

Recursion Inversion / Dynamic Programming			
n	Average Time	n	Average Time
100	0.000767809	8000	0.056289595
200	0.001502693	9000	0.062582659
300	0.002216309	10000	0.073068458
400	0.003010353	20000	0.149766705
500	0.003685789	30000	0.229626963
600	0.004441979	40000	0.306683045
700	0.005602602	50000	0.369255645
800	0.005850205	60000	0.459359847
900	0.006603973	70000	0.534317694
1000	0.007954147	80000	0.627928438
2000	0.015690907	90000	0.700311114

3000	0.02118845	100000	0.776009099
4000	0.028403731		
5000	0.036918236		
6000	0.041798477		
7000	0.049773685		

Data collection:

On collecting the data sets we noticed that at small values of n, the data points were much less in line with the data points collected at larger values of n. The constant work done by each function in this case is causing a greater deviation. This can been seen in the divide and conquer graph where the small data points lie further from the regression and the R-square value is lower for this particular regression curve than any of the other algorithms.

Generally we had very good data collection and the regression curves fit as expected with their equations, as can been seen by the high R-square of the other graphs.

5. Largest input for Algorithm that can be solved in:

	10 seconds	30 seconds	1 minute
#1 Enumeration	853	1,406	1,926
#2 Iteration	2,416	4,227	6,014
#3 Divide and Conquer	87,926	248,437	478,437
#4 Linear (DP)	1,326,109	3,970,452	7,931,037

To calculate these inputs, the log log trend line formula was used of each algorithm. Solve the formula from the trend line for n and input the time into the formula as log(time), so log(10) = 1, log(30) = 1.477121, log(60) = 1.778151 to find the largest n possible for the given time, rounding down to the nearest integer.

Assignment: Project 1
Class: CS325-400
Group: 6
Date: 2016.10.06
Due Date: 2016.10.16
Group Members: Kelsey Helms, Jay Steingold, Johannes Pikel

Assignment: Project 1
Class: CS325-400
Group: 6
Date: 2016.10.06
Due Date: 2016.10.16
Group Members: Kelsey Helms, Jay Steingold, Johannes Pikel

6. Log log plots of algorithms