CS365 Project 1: Maximum Sub Array Problem

By: William McCumstie, William Kim and Max MacEachern
Group 15

Theoretical Run-Time Analysis

Algorithm 1: Enumeration

```
maxSubarrayEnum(A[], arraySize)
        for i = first to last
                for j = i to last
                        sum = 0
                        for c = i to j
                                sum = sum + A[c]
                        if max < sum
                                max = sum
        return max
```

This algorithm features three nested loops that add to the run-time complexity of the algorithm. Say the input is an array size of n, the first iterates from the first element in the array to the last, adding n operations. The second loop iterates through n – i work, while the third loop adds n – i – j work. In total, that yields n*(n-i)*(n-i-j). Taking out lower order terms that yields a $O(n^3)$ run-time.

Algorithm 2: Better Enumeration

```
maxSubarrayBetterEnum(A[], arraySize)
        for i = first to last
                sum = 0
                for j = i to last
                        sum = sum + A[j]
                        if max < sum
                                max = sum
        return max
```

The second algorithm is similar to the first, but has one less nested loop to iterate through. For an array of size n, the first loop iterates through the entire array, adding n work. The second loop, just as before adds n – i work. Those combined yield a total amount of work equal to n*n-1. Taking out lower order terms, the run-time for this algorithm is $O(n^2)$.

Algorithm 3: Divide and Conquer

```
{max, maxSubArray} = algoritm3(array)
        if sizeOf(array) = 1
                return {firstValueOf(array), array}
        // Split array into two halves
        first = array[start : sizeOf(array)/2]
        second = array[sizeOf(array)/2 + 1 : end]
        // CASE 1 and 2: The max sub array exclusively in the first or second half
```

```
{max1, maxSub1} = algorithm3(first)
{max2, maxSub2} = algorithm3(second)
// CASE 3: The max sub array is the maxSuffix of first and the maxPrefix of second
 // MaxSuffix of first:
sum = 0
maxSuffix = endOf(first)
indexSuffix = sizeOf(first)
for c = sizeOf(first) to 1
        sum = sum + first[c]
        if sum > maxSuffix
                maxSuffix = sum
                indexSuffix = c
// MaxPrefix of second:
sum = 0
maxPrefix = startOf(second)
indexPrefix = 1
for c = 1 to sizeOf(second)
         sum = sum + second[c]
        if sum > maxPrefix
                maxPrefix = sum
                indexPrefix = c
// Merge the maxPrefix and maxSuffix
max3 = maxPrefix + maxSuffix
maxSub3 = array[indexPrefix : indexSuffix]
// Return maximum of the three cases
if max1 > (max2 AND max3)
        return {max1, maxSub1}
else if max2 > max3
        return {max2, maxSub2}
else
        return {max3, maxSub3}
```

The third algorithm utilizes recursion and iteration to find the maximum subarray. Let a function, T(n), describe the algorithm. The base case is a constant time operation and adds O(1) to T(n). The recursive section of the algorithm takes an array of size n and recursively cuts the array in half. This yields logn work. In terms of T(n), the recursive section adds 2T(n/2). The iterative section two loops, one for the left half of the array and the other for the right half of the array, or n/2 each. Therefore, the iterative section adds n work to the algorithm and adds O(n) to f(n). The last section of the code is a constant time operation that returns the maximum subarray, adding O(1) to f(n). Therefore, T(n) = O(1) + 2T(n/2) + O(n) + O(1) = 2T(n/2) + O(n). Using the master method:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \mid (a = 2) \wedge (b = 2) \wedge \left(f(n) = O(n)\right)$$

$$\Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 2}\right) = \Theta(n)$$

As $\Theta\left(n^{\log_b a}\right) = \Theta(n) = f(n)$, the second case of the master method applies.

Therefore, the time complexity is: $T(n) = \Theta\left(n^{\log_b a} \lg n\right) = \Theta(n \lg n)$

Algorithm 4: Linear-time

```
maxSubarrayLinear(A[], arraySize)
        maxSuf = A[0]
        maxSub = A[0]
        for c = 1 to arraySize - 1
                newSuf = maxSuf + A[c]
                if newSuf > A[c]
                        maxSuf = newSuf
                else maxSuf = A[c]
                if maxSuf > maxSub
                        maxSub = maxSuf
        return maxSub
```

The last algorithm features one loop through all elements of the array except the first. If the algorithm takes an array of size n, that adds n-1 work to the algorithm. As all other operations are constant time, the total run-time is O(n).

Testing the Algorithms

The algorithms where initially tested against the arrays contained within MSS_TestProblems.txt. This was easily completed as the main program had already been set to accept a text file input containing the arrays. The outputted results where then compared to those contained within MSS_TestResults.txt and were found to be the same.

Additional testing was also carried out by generating random arrays of variable length of 1 to 1000. The arrays contained randomly generated integers in the range [-100, 100]. The maximum sub array problem was then solved with each of the four algorithms. As there is only one single solution to the maximum sub array problem, the results of each algorithm were tested against themselves to insure that they matched. This was repeated for over 10,000 randomly generated arrays in which all the algorithms passed.

For these reasons, there is a high degree of confidence that the algorithms return the same answer regardless of the array input.

Experimental Analyses

Run Time Data and Graphs

Run time data for each of the algorithms was obtained by using the system clock incorporated into the chrono library in C++11. The chrono library was chosen as it can time the algorithm accurately down to the nanosecond. Different array input sizes were obtained by randomly generating arrays in the same manner as done during testing. Care was taken not to time the creation of the array, only the run time of the algorithm on it. The data contained in Table 1 shows the run time of each of the algorithms with different array input sizes. Due to the vastly different run times of each algorithm, different input sizes had to be selected for each of them.

Table 1: Run Time of Algorithms with Different Array Input Sizes

Algorithm 1		**Algorithm 2**		**Algorithm 3**		**Algorithm 4**	
Input Size	Time (ns)	Input Size	Time (ns)	Input Size	Time (ns)	Input Size	Time (ns)
10	7270	100	19672	100	15395	100	1710
100	635923	1000	5611685	750000	136059632	27000000	118179816
200	3937844	2500	8609545	1250000	172918337	55000000	240472387
400	31027633	5000	34543382	2250000	321372258	83000000	366266596
500	59551775	7500	76831164	3500000	499137634	110000000	482414624
700	160912358	10000	136578378	5000000	723229990	140000000	611974333
900	341670889	15000	315273475	6500000	966300808	170000000	750442943
1100	622742250	20000	558490549	7500000	1111209900	200000000	887984399
1300	1032695058	25000	875765021	9000000	1350976229	220000000	980913770
1500	1569813992	30000	1257339090	10000000	1481818046	250000000	1110663357

Using the data obtained from the run time analyses, the algorithmic complexity could be determined by first plotting the data and then using an appropriate regression model for each algorithm. Figure 1 - Figure 4 shows the run time verses the input size of the arrays for algorithms 1 – 4 respectively. They also contain the best fit for each algorithm.

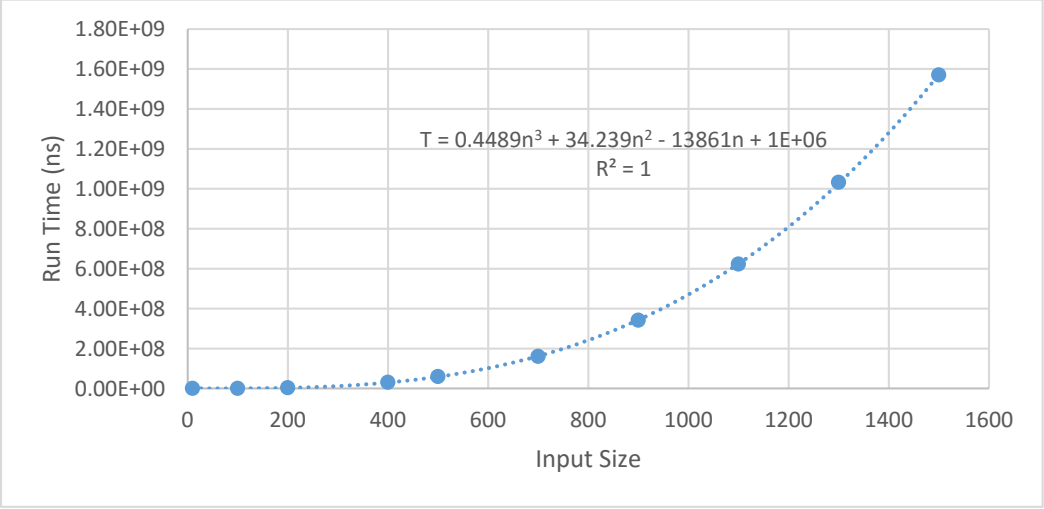

Figure 1: Run Time of Algorithm 1

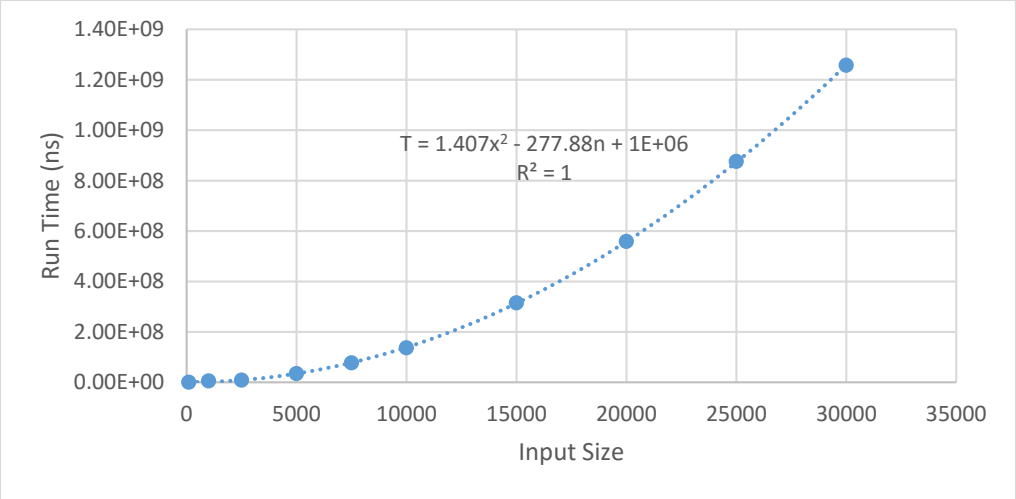

Figure 2: Run Time of Algorithm 2

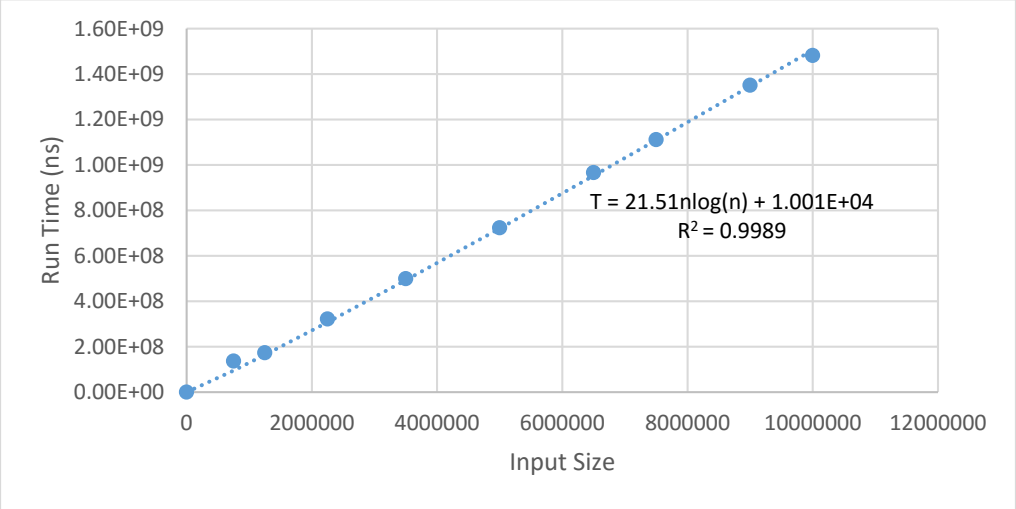

Figure 3: Run Time of Algorithm 3

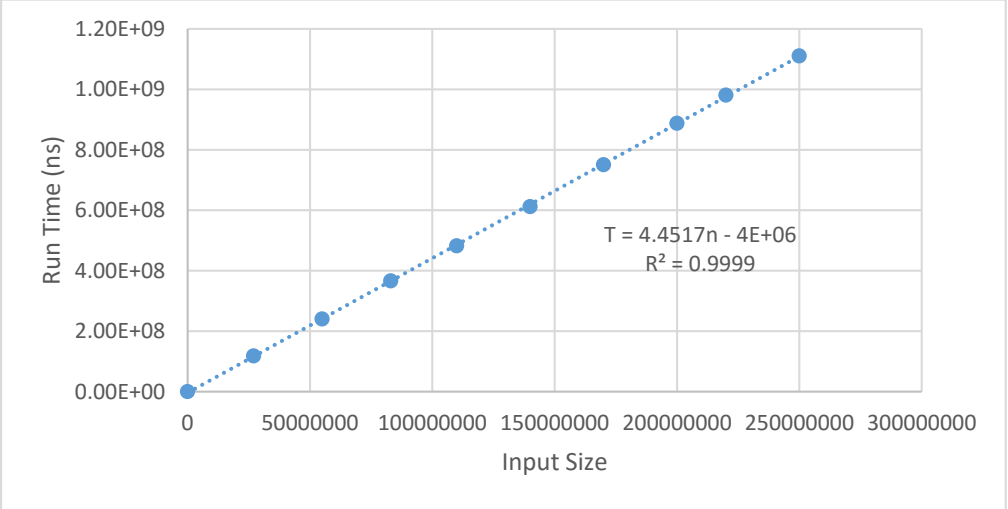

Figure 4: Run Time of Algorithm 3

Experimental Run times

From the curve fitting the experimental run time results are as follows:

Algorithm 1: $T(n) = 0.4489n^3 + 34.239n^2 – 13861n + 1E+06$	$= O(n^3)$	Cubic
Algorithm 2: $T(n) = 1.407n^2 - 277.88n + 1E+06$	$= O(n^2)$	Quadratic
Algorithm 3: $T(n) = 21.51nlog(n) + 1.001E+04$	$= O(nlogn)$	Log-Linear
Algorithm 4: $T(n) = 4.4517n - 4E+06$	$= O(n)$	Linear

From this analyses the algorithm 1 has a cubic execution time, algorithm 2 is quadratic, algorithm 3 is log-linear and algorithm 4 is linear. This matches the theoretical run time analyses in all cases indicating that the regression models are, indeed, correct.

For this reason, there are no discrepancies between the experimental and theoretical run time analyses. It should be noted that algorithm 3 whilst fitting a log-linear regression model also fits a linear model. It is hypothesised that this is because the recursive calls are dominating the execution time making it follow close to a linear trend.

Maximum Problem Size with Time

Table 2 shows the maximum problem size that can be solved in 10 sec, 30 sec and 1 min. This was done by subsisting the time for T(n) in each of the experimental regressions and solving for n. As it can be seen, algorithm 1 (the slowest) is slower by up to a factor of 10^7 then compared to algorithm 4 (the fastest). This shows that a O(n) execution time significantly improves the performance than compared to $O(n^3)$.

Table 2: Maximum Problem size with Time

Time	10 sec	30 sec	1 min
Algorithm 1	2792	4035	5089
Algorithm 2	84399	146177	206601
Algorithm 3	$2.72*10^7$	$7.68*10^7$	$1.48*10^8$
Algorithm 4	$2.25*10^9$	$6.74*10^9$	$1.35*10^{10}$

Log-Log Plots

Figure 5 shows the run time for all four algorithms on a log-log plot.

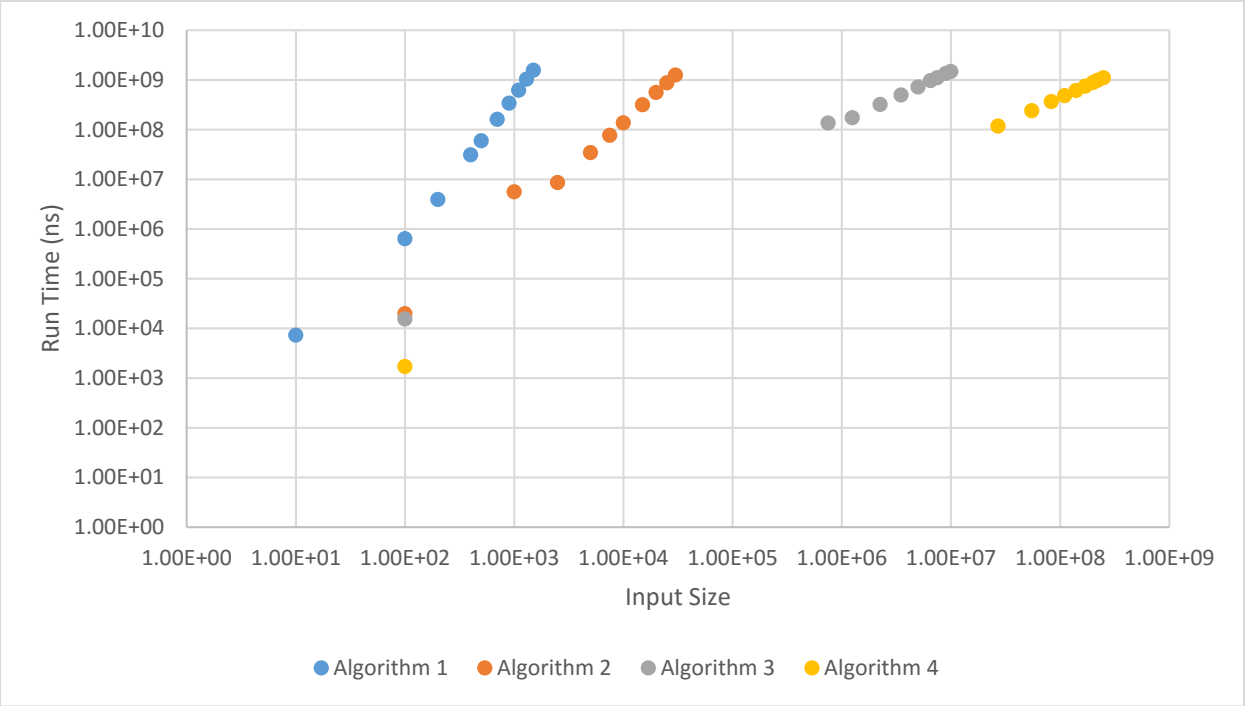

Figure 5: Log-Log Plot of Run Time for all Algorithms

It can clearly be seen that algorithm 4 is the fastest due to its O(n) execution time. The second fastest is algorithm 3 as expected as the O(nlogn) only slows the algorithm down a fraction. When the algorithmic complexity increases to $O(n^2)$ and $O(n^3)$ however the run time starts to slow done significantly. This is why algorithm 1 and 2 are the slowest with execution times of $O(n^3)$ and $O(n^2)$ respectively.

As all the run times look approximately linear on the log-log graph, the algorithmic complexity can be approximated with the linear gradient intercepting at the origin. Figure 6 shows the gradient of the run times on the log-log plot.

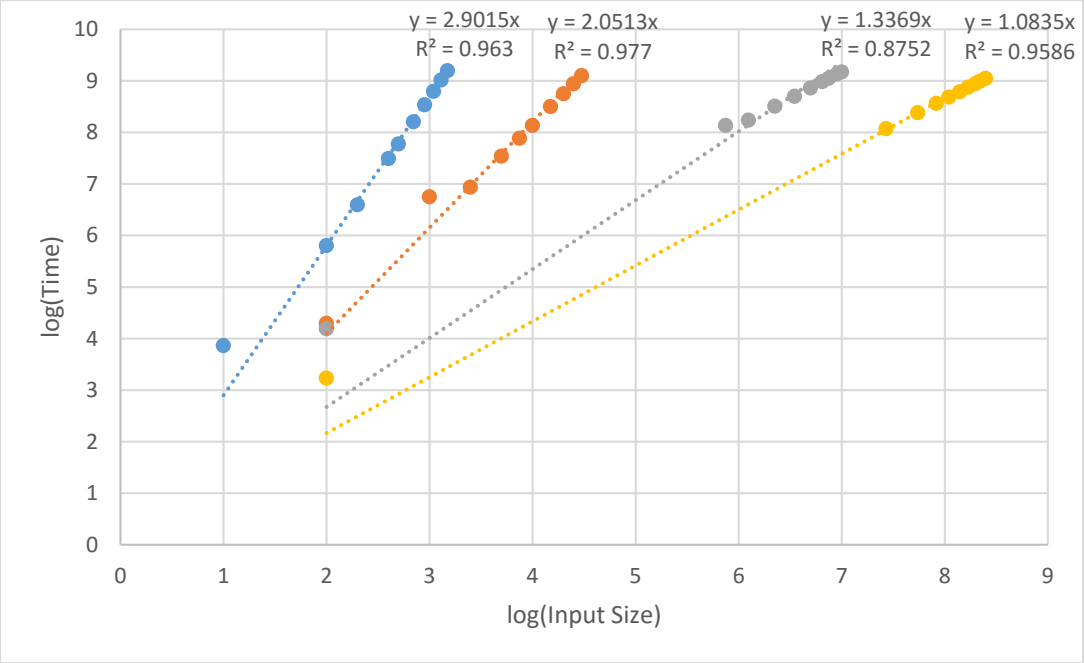

Figure 6: Gradient of Log-Log Plots

Using the gradient of the log-log plot, the algorithmic complexity of each of the algorithms can be approximated as:

- Algorithm 1: $O(n^{2.9})$
- Algorithm 2: $O(n^{2.1})$
- Algorithm 3: $O(n^{1.3})$
- Algorithm 4: $O(n^{1.1})$

The algorithmic complexities are close but do not match exactly with those calculated experimentally nor theoretically. The difference is due to the error induced in the modelling method use. It is particularly bad for algorithm 3 as it is log-linear and therefore deviates from the linear plot (on the log-log plot) due to the log component in the algorithmic complexity.