Project 1
Maximum Sum Subarray

Group Number 1
Rosa Tung, Sam Nelson, Kara Franco

Theoretical Run-time Analysis

Algorithm 1: Enumeration

Pseudocode:

```
1.  enumMSS ( array[ ] )
2.  maxSum ← 0, low ← 0, high ← 0
3.  for when 0 ≤ i ≤ array.length
4.          for 0 ≤ j ≤ array.length
5.                  sum ← 0
6.                  for 0 ≤ k ≤ j
7.                          sum ← sum + array [ k ]
8.                  if sum > maxSum then
9.                                  maxSum ← sum
10.                                 low ← i
11.                                 high ← j
12. return maxSum, array with [low - high]
```

Analysis of Asymptotic Run-time:

In algorithm 1, enumeration, we loop over each pair of indices i, j and compute the sum $\sum_{k=1}^{j} A[k]$. The best sum is stored along with the two indices that make the maximum subarray. As we see by line 3, 4 and 6, *i, j* and *k* are bound by the array length (n). The operations performed on lines 7 - 11 are in constant time *O(1).* By these observations, we have two nested for loops,
$(\ O(n)\ i\ iterations) * (\ O(n)\ j\ iterations) * (\ O(n)\ k\ iterations) * O(1)$ or $O(n^3)$. This is proportional to the equation:
$\sum_{i=1}^{n}\sum_{j=1}^{n}(j-i+1) = \sum_{i=1}^{n}\sum_{j'=1}^{n-i+1} j'$

$$= \sum_{i'=1}^{n}\sum_{j'=1}^{i'}$$

$$= \frac{n(n+1)(n+2)}{6}$$

$$= O(n^3)$$

Algorithm 2: Better Enumeration

Pseudocode:

```
1.  betterEnumMSS ( array[ ] )
2.  maxSum ← 0, low ← 0, high ← 0
3.  for when 0 ≤ i ≤ array.length
4.          sum ← 0
5.          for 0 ≤ j ≤ array.length
6.                  sum ← sum + array [ j ]
7.                  if sum > maxSum then
8.                                  maxSum ← sum
9.                                  low ← i
10.                                 high ← j
11. return maxSum, array with [low - high]
```

Analysis of Asymptotic Run-time:

In algorithm 2, better enumeration, we differ by computing the sums from $\sum_{k=1}^{j-1} A[k]$ in $O(1)$ time (lines 5 - 10). Before, in the enumeration implementation, we were computing each sum in a for loop, at $O(n)$ time. In this implementation, we take out the repeated steps from the outer for loop, and store the best sum along with the two indices that make the maximum subarray. As we see by line 3 and 5, *i* and *j* are bound by the array length (n). The operations performed on lines 6 - 10 are in constant time *O(1).* By these observations, we have one nested for loop, $(\ O(n)\ i\ iterations) * (\ O(n)\ j\ iterations) * O(1)$ or $O(n^2)$.

Algorithm 3:

Pseudocode:

```
1.  MSSDaQ ( array[ ], beg, end )
2.  if ( beg = end ) then
3.          return beg
4.  mid ⟵ int
5.  mid ⟵ ( beg + end ) / 2
6.  SMALL ⟵ int ( = to smallest possible 16 bit int)
7.  calculate max on the left half
8.  leftMax ⟵ int
9.  call algorithm recursively ( array[ ], beg, mid ) set leftMax ⟵ result
10. calculate max on the right half
11. rightMax ⟵ int
12. call algorithm recursively ( array[ ], mid + 1, end ) set rightMax ⟵ result
13. calculate middle sum
14. crossingLeftSum ⟵ int, crossingRightSum ⟵ int, leftSumMax ⟵ int
15. rightSumMax ⟵ int, crossingSum ⟵ int
16. leftSumMax ⟵ SMALL, rightSumMax ⟵SMALL
17. // for left sum
18. for ( int x center to left ) do
19.       crossingLeftSum ⟵ crossingLeftSum + a
20.       if ( crossingLeftSum > leftSumMax ) then
21.                 leftSumMax ⟵ crossingLeftSum
22. // for right sum
23. for ( int y center + 1 to right ) do
24.       crossingRightSum ⟵ crossingRightSum + a
25.       if ( crossingRightSum > rightSumMax ) then
26.                 rightSumMax ⟵ crossingRightSum
27. crossingSum ⟵ ( leftSumMax + rightSumMax )
28. max ⟵ int
29. max ⟵ max of leftMax, rightMax, crossingSum
```

Analysis of Asymptotic Run-time:

$T(1) = 1$

Crossing function takes $\Theta(n)$time ....each for loop takes $\Theta(1)$ time

Dividing takes $\Theta(1)$ time

Conquering takes $T(\frac{n}{2})$ time for each sub problem → $2T(\frac{n}{2})$ time for conquering

Combining takes $\Theta(n)2 + \Theta(1)$ time

Recurrence case becomes:

$T(n) = \Theta(1) + 2T(\frac{n}{2}) + \Theta(n) + \Theta(1) \rightarrow$ absorb $\Theta(1)$'s into $\Theta(n) \rightarrow 2T(\frac{n}{2}) + \Theta(n)$

$T(n) = \Theta(1)$ if $n = 1$

$2T(\frac{n}{2}) + \Theta(n)$ if $n > 1$

Use master method:

$T(n) = \Theta(n \lg n)$

Algorithm 4:

Pseudocode:

Note: When all negative, maximum subarray is empty set 0. Zero's can count as part of a maximum subarray.

```
1.  maxSubLinear ( Array[ ] )
2.  bestMax ← 0, curMax ← 0, rightIndex
3.  for each element of Array [ 0 → n ]
4.      curMax ← curMax + Array [ placeInArray ]
5.      if curMax < 0 then
6.          curMax ← 0
7.      if bestMax < curMax then
8.          bestMax ← curMax
9.      rightIndex ← placeInArray
10. return bestMax
```

Analysis of Asymptotic Run-time:

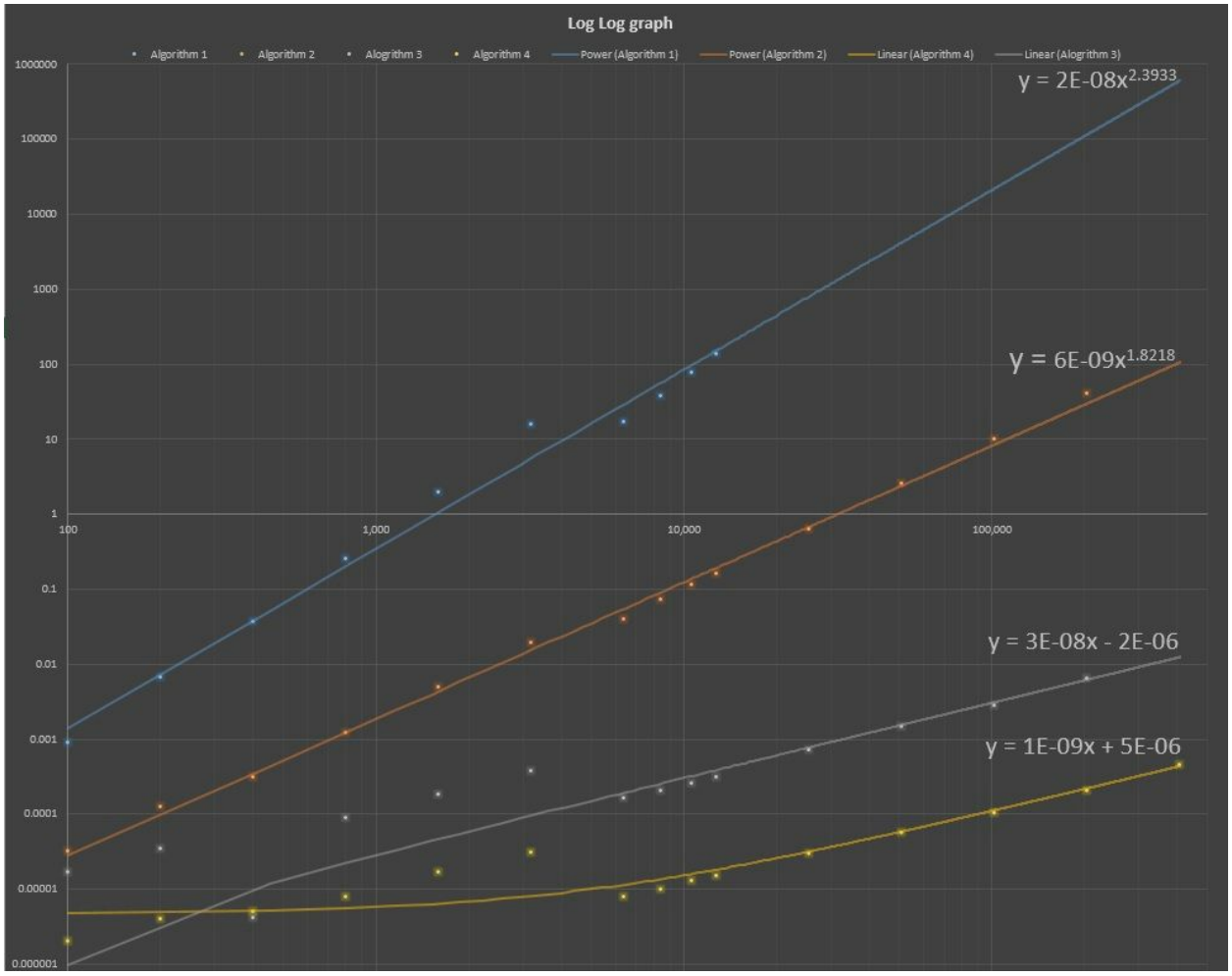
This algorithm starts at the left end and, as it goes towards the right one and a time, compares two sums. The initial max subarray $A[1..j]$ is zero, so if all the elements are negative it would return an empty subset, which shouldn't be the case here. The curMax counter extends itself one by one, adding the next sum to itself. If it is greater than the "bestMax" then the bestMax value will be stored and the index is also stored to locate the end of the current max subArray, and curMax keeps moving on.

The running time of this algorithm is obviously $O(n)$, since we have one loop, and evaluate each element exactly once.

Testing:

After each of our algorithms passed the MSS_TestProblems.txt file, we ran the MSS_Problems.txt and made sure our results matched against each other. We then combined them all in one program and tested them for correctness with each other again, and further checked our results for correctness against the results of three popular maximum sum calculating websites. After that, we continued to test our results when implementing our random number generator and timers, as well as with files that contained newlines.

As for randomly generating our own inputs, we had it in our main as seen in the example, however decided to make it a separate program. This was for efficiency, so one team member could run tests while another could generate files. The third team member was able to write down the results. In this way we were able to complete our testing fairly quickly and even experimented with a file of trillion numbers (we didn't include that in our results that was just for fun).



Experimental Analysis:

	Value of n	Runtime (seconds)	Average Time per n
Algorithm 1:	100	0.000886	0.000009
	200	0.006547	0.000033
	400	0.037114	0.000093
	800	0.251477	0.000314
	1600	1.95739	0.001223
	3200	15.6897	0.004903
	6400	17.0092	0.002658
	8400	38.0159	0.004526
	10600	76.7435	0.00724
	12800	137.07	0.010709

Table of Algorithm 1 Runtime Average

	Value of n	Runtime (seconds)	Average Time per n
Algorithm 2:	100	0.000032	3.20E-07
	200	0.000123	6.15E-07
	400	0.000305	7.63E-07
	800	0.001219	1.52E-07
	1600	0.00484	0.000003
	3200	0.019313	0.000012
	6400	0.0393945	0.000006
	8400	0.071595	0.000009
	10600	0.113543	0.000011
	12800	0.157593	0.000012

Table of Algorithm 2 Runtime Average

	Value of n	Runtime (seconds)	Average Time per n
Algorithm 3:	100	0.000017	1.70E-07
	200	0.000034	1.70E-07
	400	0.0000042	1.05E-08
	800	0.000088	1.10E-07
	1600	0.000182	5.50E-08
	3200	0.000369	1.15E-07
	6400	0.000162	2.53E-08
	8400	0.000202	2.40E-08
	10600	0.000255	2.41E-08
	12800	0.000311	2.43E-08
	25600	0.000723	2.82E-08
	51200	0.00147	2.87E-08
	102400	0.002784	2.72E-08
	204800	0.006464	3.16E-08

Table of Algorithm 3 Runtime Average

	Value of n	Runtime (seconds)	Average Time per n
Algorithm 4:	100	0.000002	2.00E-08
	200	0.000004	2.00E-08
	400	0.000005	1.25E-08
	800	0.000008	1.00E-08
	1600	0.000017	1.06E-08
	3200	0.000031	9.69E-09
	6400	0.000008	1.25E-09
	8400	0.00001	1.19E-09
	10600	0.000013	1.23E-09
	12800	0.000015	1.17E-09
	25600	0.00003	1.17E-09
	51200	0.000057	1.11E-09
	102400	0.000104	1.02E-09
	204800	0.000206	1.01E-09
	409600	0.000451	1.10E-09
	819200	0.001023	1.25E-09
	1638400	0.002483	1.52E-09
	3276800	0.004941	1.51E-09
	6553600	0.00988	1.51E-09
	13107200	0.019919	1.52E-09
	26214400	0.039883	1.52E-09
	52428800	0.079077	1.51E-09
	104857600	0.180543	1.72E-09

Table of Algorithm 4 Runtime Average

Functions that model the relationship between input size n and time

(where y = time and x = input size)

Algorithm 1: $y = 2e-08x^{2.3933}$

Algorithm 2: $y = 6e-09x^{1.8218}$

Algorithm 3: *y = (.00000003)x - 2*10^-6*

Algorithm 4: $y = 1e-09x + 5e-06$

Determining the largest input for 1, 2, and 5 minutes with regression models

Algorithm 1:

$2e-08x^{2.3933}$

Largest Input for 1 Minute: ~ 9117

60 = .00000002($x^{2.3933}$)

Largest Input for 2 Minutes: ~ 12179

120 = .00000002($x^{2.3933}$)

Largest Input for 5 Minutes: ~ 17861

300 = .00000002($x^{2.3933}$)

Algorithm 2:

$6e-09x^{1.8218}$

Largest Input for 1 Minute: ~ 308,373

60 = .000000006($x^{1.8218}$)

Largest Input for 2 Minutes: ~ 451,143

120 = .000000006($x^{1.8218}$)

Largest Input for 5 Minutes: ~ 746,013

300 = .000000006($x^{1.8218}$)

Algorithm 3:

y = (.00000003)x - 2*10^-6

Largest Input for 1 Minute: ~ 2,000,000,067

Largest Input for 2 Minutes: ~ 4,000,000,067

Largest Input for 5 Minutes: ~ 10,000,000,070

Algorithm 4:

$$y = 1e-09x + 5e-06$$

Largest Input for 1 Minute: .000000001x + .000005, x = 59,999,995,000

Largest Input for 2 Minutes: .000000001x + .000005, x = 119,999,995,000

Largest Input for 5 Minutes: .000000001x + .000005, x = 299,999,995,000

Discrepancies Between Experimental and Theoretical Running Times

So in our analysis for Algorithm 1 we came up with a theoretical runtime of $O(n^3)$. The regression curve from our data we got was y=2e - $8^{2.3933}$ making it have a running time of $O(n^{2.3933})$. This algorithm was one of our largest discrepancies. An explanation to why our data behaved unexpectedly is the unreliability of computer clock time. As mentioned in lecture, we cannot always rely on clock time to validate the runtime of an algorithm. Since there appeared to be discrepancies with the server performance we cannot be sure that the data collected from algorithm 1 is truly representative to the it's runtime performance.

For Algorithm 2 we came up with a theoretical runtime of $O(n^2)$. The regression curve from our data we came up with was y = $(6e-09)x^{1.8218}$, making it a runtime of $O(n^{1.8218})$. We were pretty happy with this, and although it wasn't exact the correlation is similar enough for us to get the feel and pattern of the algorithm.

For Algorithm 3 we came up with a theoretical runtime of O(nlgn). In this case, the data points from our results followed a similar pattern to a linear curve, so the regression curve we decided to go with was a linear one, y = (.00000003)x - $2e10^{-6}$. We debated whether to go with a nlgn curve, but decided not to, because that would mean that our decision was not driven by our test results, rather our theory. However, when we took a look at an nlgn curve, we did notice that at the last section of the curve (where our data points would reside), did actually look very linear. Although algorithm 4 and 3 both grow in the same way, the position of the curve for Algorithm 4 on the graph shows that it takes less time in general per n than for Algorithm 3.

For Algorithm 4 we came up with a theoretical runtime of O(n). Looking at the data points, our results followed a similar linear pattern and we were able to come up with $(1e-9)x + (5e-6)$ which matched with a our theoretical runtime. This was the algorithm we were probably the most happy with and pretty much matched with our expectations. It was faster than all the other algorithms by a whole lot, and we were able to get a wider range of data points with it, which maybe explains why it was much easier to extract a regression curve from the data than the other algorithms.