**Algorithm 1**

**Pseudocode and Asymptotic Run Time**

This algorithm works by keeping track of the sum of every subarray. If you loop over every index and keep track of every sum, then you will need three loops. The first two keep track of the array indices and the last one is used for adding. Thus, every element in the array will be looped over three times. Here is the pseudocode:

max = 0

for every number in array:

for every number in array:

placeholder\_sum = 0

for every number in the range of the first two indices:

placeholder\_sum += array[current\_index\_in\_loop]

if placeholder\_sum > max:

max = placeholder\_sum

first\_index = index of outer loop

second\_index = index of inner loop

return (array[first\_index:second\_index], max)

**Testing**

To ensure correctness this algorithm was tested against various test sets including those provided by the instructor as MSS\_Problems.txt and MSS\_TestProblems.txt. Manual verification of the correct answer was matched against the algorithm output to ensure the algorithm was correct. Additional edge cases such as a very large number surrounded by very large negative numbers were also tested (example: [1, 2, 3, -888, 999, -888, 75, 76, 24, -3, -25].

1. Time data

|  |  |
| --- | --- |
| n | time (seconds) |
| 10 | 0.00012 |
| 50 | 0.00257 |
| 100 | 0.01634 |
| 200 | 0.11304 |
| 300 | 0.36752 |
| 400 | 0.99394 |
| 500 | 1.654 |
| 800 | 6.532 |
| 1000 | 12.919 |
| 1100 | 17.152 |

1. Graph with best fit line

**Runtime Analysis**

1. A function that best models the graph is:

f(x) = 2 x 10-5x2 – 0.0077x + 0.4899; The function is exponential.

1. There wasn’t really a discrepancy between the theoretical and experimental runtimes. Even increasing n slightly resulted in a significantly larger runtime; which is to be expected given the slow runtime of the algorithm in general.
2. Using the equation, the largest expected input for the algorithm to solve:

5 seconds: 704

10 seconds: 908

60 seconds: 2777

1. Algorithm 1 Log-Log Plot

**Algorithm 2**

**Pseudocode and Asymptotic Run Time**

This algorithm is much faster (but still quite slow) because it removes the need for the inner loop. It keeps track of the current sum in the second loop. This takes the algorithm from O(n3) to O(n2). Pseudocode for algorithm:

total = 0

max = 0

for every number in array:

total = 0

for every number in array:

total += array[at index of second loop]

if total > max:

max = total

first\_index = index of outer loop

second\_index = index of inner loop

**Testing**

To ensure correctness this algorithm was tested against various test sets including those provided by the instructor as MSS\_Problems.txt and MSS\_TestProblems.txt. Manual verification of the correct answer was matched against the algorithm output to ensure the algorithm was correct. Additional edge cases such as a very large number surrounded by very large negative numbers were also tested (example: [1, 2, 3, -888, 999, -888, 75, 76, 24, -3, -25].

1. Time data

|  |  |
| --- | --- |
| n | time (seconds) |
| 100 | 0.00042 |
| 300 | 0.00345 |
| 500 | 0.00935 |
| 700 | 0.02358 |
| 1000 | 0.03752 |
| 1500 | 0.13256 |
| 2000 | 0.16825 |
| 3000 | 0.38579 |
| 5000 | 1.1052 |

1. Graph with best fit line
2. Function for the best fit:

4 \* 10-8x2 + 3 \* 10-5x; This function is also exponential

1. The theoretical and experimental runtimes of the algorithm were very close. We used FLIP to test so system resources could account for slight discrepancies.
2. Using the equation, the largest expected input for the algorithm to solve:

5 seconds: 11,149

10 seconds: 15,485

60 seconds: 38,356

1. Algorithm 2 Log-Log Plot

**Algorithm 3**

**Pseudocode and Asymptotic Run Time**

This algorithm works on the assumption that the maximum size subset will be contained in the left half, the right half, or cross the middle of the input set. The algorithm recursively calls itself until the set it is working with is size 1 and returns the value.

The code for this algorithm was divided into two functions – one to find the maximum subarray of the left side and right sides of the set and one to find the maximum subarray of the subset at the middle of the set.

The left and right function first finds the middle of the input set. The function then recursively runs on the left then the right side of the input set. The base case for the recursive function is when the input set size is equal to 1 in which case the only element is returned.

The middle function begins at the middle point of the input set given to it and begins searching for the maximum subset radiating out from the center. It searching to the left starting at the middle element and to the right starting at element to the right of the middle element. It starts counting the sum of all of elements to the left and right of the midpoint of the input set. The running total sum is maintained and the sum for each direction (left and right) is set to the sum at its maximum value. If the sum is less than 0 at any point then 0 is used for the total for that side, indicating that no elements from that side of the middle of the set will be used. The function then returns the left and right indices representing the beginning and end of the middle MSS and the total of the left and right sides of the middle MSS.

Pseudocode:

midMSS(start, mid, finish, inputset):

left= inputset[mid]

right = inputset[mid+1]

leftindex = mid

rightindex = mid+1

sum = 0

for (i = mid, i < start-1 i--)

sum = sum + inputset[i]

if sum > left:

left = sum

leftindex = i

sum = 0

for (i = mid+1, i < finish+1, i++)

sum = sum + inputset[i]

if sum > right:

right = sum

rightindex = i

if right < 0:

rightindex = leftindex

if left < 0:

leftindex = rightindex

return leftindex, rightindex, left+right

algorithm3(start, finish, inputset)

if start == finish

return start, finish, intpuset[start]

mid = (finish + start)/2

leftstartindex, leftendindex, leftsum = algorithm3(start, mid, inputset)

rightstartindex, rightendindex, rightsum = algorithm3(mid+1, finish, inputset)

midstartindex, midendindex, midsum = midMSS(start, mid, finish, inputset)

if rightsum >= leftsum and midsum:

return rightstartindex, rightendindex, rightsum

else if leftsum >= midsum:

return leftstartindex, leftendindex, leftsum

else

return midstartindex, midendindex, midsum

**Run-time analysis:**

This algorithm divides the input set in half recursively at each step. Each halved input is then subject to the midMSS function that performs n operations – n/2 for each half of the input set for a total of n operations. This results in a theoretical runtime of O(n) for midMSS() and O(lgn) for algorithm3() resulting in a total runtime of O(nlgn)

**Testing**

To ensure correctness this algorithm was tested against various test sets including those provided by the instructor as MSS\_Problems.txt and MSS\_TestProblems.txt. Manual verification of the correct answer was matched against the algorithm output to ensure the algorithm was correct. Additional edge cases such as a very large number surrounded by very large negative numbers were also tested (example: [1, 2, 3, -888, 999, -888, 75, 76, 24, -3, -25].

1. Time data

|  |  |
| --- | --- |
| n | Average time in seconds |
| 100 | 0.000415897 |
| 1000 | 0.004977274 |
| 5000 | 0.027705288 |
| 10000 | 0.057851863 |
| 25000 | 0.151532793 |
| 50000 | 0.319801092 |
| 100000 | 0.655369711 |
| 250000 | 1.692620134 |
| 500000 | 3.508603716 |
| 1000000 | 7.590828514 |

1. Graph with linear best fit line:
2. The results appear to be linear at first glance, however as n grows larger and larger the results for t appear to curve upward which indicates we are looking at some form of O(n log n) graph. A logarithmic regression curve was found using <https://mycurvefit.com/> :

t = .0000005468513\*n \* ln(n). This is a result of the same order of magnitude as n lg n and is closely aligned with the theoretical results.

1. Potential discrepancies between the experimental and theoretical running times could be caused by performance issues on the running server. The experimental runtimes were obtained on the FLIP server. Since many people may be logged into FLIP performance could be worse for a given value of n if computing resources are being shared with other users. That being said, the experimental and theoretical running times were as close to each other as could be expected under the circumstances.
2. Solving t = .0000005468513\*n \* ln(n) where t = 5, 10, and 60 we obtain an expected input size for the algorithm that can be solved in t seconds:

5 = .0000005468513\*n \* ln(n). n = 680759

10 = .0000005468513\*n \* ln(n). n = 1,299,020

60 = .0000005468513\*n \* ln(n). n = 6,963,550

**Algorithm 4**

**Pseudocode and Asymptotic Run Time**

The algorithm looks at each value starting from the left and going to the right. With each value it checks if we should add it to the maximum array or not. If we encounter a negative number, we check if that negative’s absolute value is larger than our current sum. If it is we store that subarray we have and move past the negative, restarting our search from scratch. If this event is encountered again, the newer subarray will be compared to the sum of the previous and discarding the subarray with lower sum. If we have a negative but it does not return our sum to 0 or a negative value, we then start looking for a temporary sub array to offset that negative value later.

Pseudocode:

*WHILE current index < array length:*

*IF current value being looked at in array is not negative :*

*IF current index is not right after the end of our held subarray:*

*IF we have no temporary subarray set yet:*

*Start of new temporary subarray = current index*

*End of temporary subarray = current index*

*Add the current value being looked at to the sum of*

*our temporary subarray*

*IF sum of temporary subarray > absolute value of our*

*negative value:*

*End of held subarray = end of temporary subarray*

*Add sum of temporary subarray and negative value to sum*

*sum of held subarray*

*Set our negative value and sum of temporary subarray to 0*

*ELSE:*

*End of our held subarray = index*

*Add value being looked at to sum of our held subarray*

*ELSE:*

*Add value being looked at to our negative value*

*IF we have a temporary subarray that is after the end of our held*

*subarray:*

*Add sum of our temporary subarray to our negative value*

*Set the start, end and sum of our temporary subarray to 0*

*IF the absolute value of our negative value >= sum of held subarray:*

*IF sum of our previous subarray <= sum of held subarray:*

*Replace our previous subarray with our held array*

*Set the start and end of our held subarray to the next index*

*Reset our negative value to 0*

*Reset sum of our held subarray to 0*

*Increase index by 1*

Asymptotic Run Time

The algorithm by our pseudocode runs a maximum of 12 statements for each index of the array, depending on the conditional statements. This makes f(x) = 12n as the algorithm loops only once through the array indices. This means the function is bound above and below by g(x) = cn for some integer c making the asymptotic run time of the algorithm 𝚯(n) or a linear run time.

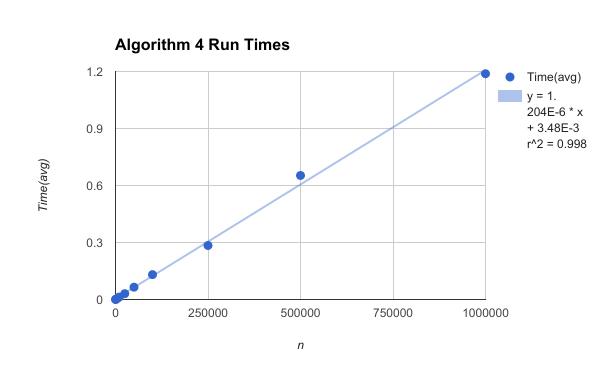
**Testing**

To ensure correctness this algorithm was tested against various test sets including those provided by the instructor as MSS\_Problems.txt and MSS\_TestProblems.txt. Manual verification of the correct answer was matched against the algorithm output to ensure the algorithm was correct. Additional edge cases such as a very large number surrounded by very large negative numbers were also tested (example: [1, 2, 3, -888, 999, -888, 75, 76, 24, -3, -25].

1. Time Data

|  |  |
| --- | --- |
| n | Time(avg) |
| 100 | 0.00012009 |
| 1000 | 0.00120383 |
| 5000 | 0.00604069 |
| 10000 | 0.01276966 |
| 25000 | 0.03061415 |
| 50000 | 0.06486696 |
| 100000 | 0.13099905 |
| 250000 | 0.28430449 |
| 500000 | 0.65270637 |
| 1000000 | 1.18827622 |

2.Graph with Best Fit Line



3.Function Equation for best fit

f(x) = 1.204 \* 10-6x + 3.48 \* 10-3

The data is linear.

4.The run time of the algorithm had some differences at n = 500,000 and n = 250,000. This could partially be due to varying quantities of negative numbers in the test sets created for timing as they were created using python’s pseudo-random number generation functions. The algorithm has more statements for dealing with negative numbers so a greater quantity of negative numbers in the array could affect run time. This is simply variation in the constant multiplied by n in our theoretical run time.

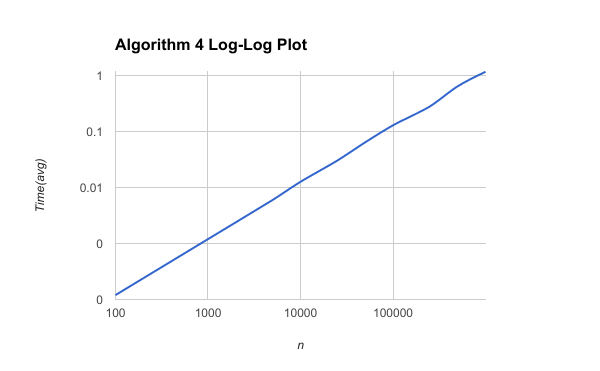
5. By the regression model, the largest input for the algorithm to solve in a given time:

5 seconds: 4,149,933

10 seconds: 8,302,757

60 seconds: 49,830,996

6. Log-log Plot



1. **Combined graph for all algorithms**