Names and Group Number:
Group 16: Richard Mitchell, Kevin Thompson, Tatyana Vlaskin
****

Testing [Proof of Correctness]:
****

We tested the validity of our algorithms by running each implementation with an input of MSS_Problems.txt files provided by the instructor. The file is submitted as part of this assignment. This file contains an array per line that consists of positive and negative integers. Our main.cpp program reads in these arrays and stores the corresponding output to MSS_Results.txt file where the subarrays are listed in addition to the maximum sum found. Each of the algorithm implementations produced the same output, suggesting a correct implementation. Trivial input examples where there is a single positive value, all positive values, or an array with a length of one all return correct results on visual inspection.
****

Theoretical Run-time Analysis and Experimental Run Time Analysis:
****

Algorithm 1: Enumeration

Enumeration –Pseudocode:

For the enumeration algorithm, we need:

1. Determine the ending indices of the subarray to be summed
2. Sum the values from the starting point to the end point
3. if the sum of values between the starting point and ending point is larger than previous sum, assign the new sum to maximum sum.
4. Keep track of starting and ending points of the subarray for later return.
5. Keep track of the sum that will be returned later

```
//moves the starting point of the array
for (int i = 0; i < array.size(); ++i) {
 //moves the ending point of the array
  for (int j = i; j < array.size(); ++j) {
  //set current maximum to 0
    int current_max = 0;
   //sums up numbers between starting and ending point
   for (int k = i; k <= j; ++k) {
     current_max  =  current_max   + array [k];
    //compares new sum to the old sum
    if (curr_maximum > max) {
     //if current maximum is larger than the max stored
     // set current_max to max
      max = curr_max;
     //stores first index of the the maximum subarray
     max_first_idx = i;
     //stores last index of the maximum subarray
     max_last_idx = j;
```

Enumeration –theoretical running time analysis:

The algorithm determines the starting and ending indices of the subarray to be summed. This will generate subarray of length n, two arrays of length n-1, three arrays of length n-2, four subarrays of size n-3 and so on. The total number of arrays: 1,2,3,4,5,6…..n = $\frac{n(n-1)}{2}$ arrays, that gives us T(n) = n^2 , which translates into O(n^2) complexity. However, there is also, the innermost loop that sums the value between two indices to determine if the current array is the maximum sub subarray. This is an n operations that are performed for each of the subarrays, which makes the total time complexity = n* **Θ** (n^2) = **Θ** (n^3). T(n) = **Θ** (n^3)

Enumeration - Experimental Analysis and Plots

For the experimental analysis, the algorithm was run 10 times for each of the following input sizes for example n = 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000. Each array was generated using random number generator. The original size was 100 and we kept incrementing size by 100 until we reached 1000 elements. Numbers in the array were in the range -999 to 1000 inclusive. The time is calculated in seconds using the following formula:

duration = (clock() - begin) / (double) CLOCKS_PER_SEC ;

The code used to generate random arrays of different sizes is provided below:

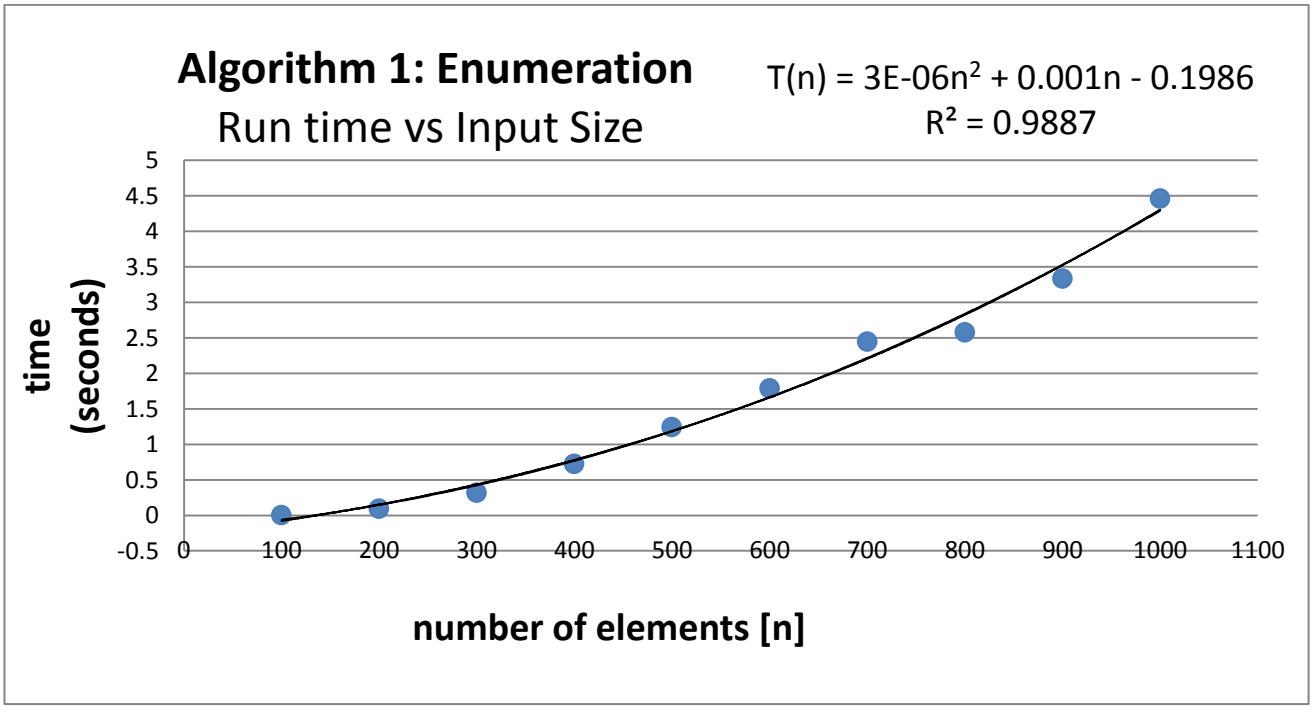
```
for(int size = 100; size < 2000; size += factor){
 //maximum # in the array
  int max = 1000;
 //minimum # in the array
  int min = -999;
 //used in random # generator function
  int range = max - min + 1;
  //arrays are generated
  for(int w = 0; w < size; w++){
    number = rand() % range + min;
    data.push_back(number);
  }
  begin = clock(); //start timer
 //run the algorithm
  int enumeration_sum = enumeration(&data, 0, size-1,ofs);
     //time is in milliseconds
  duration = (clock() - begin) / (double) CLOCKS_PER_SEC ;
           if (size >= 1000)
     factor = 1000;
```

The average running time was calculated for each n and results were plotted [running times as a function of input size n]. See data and graph below:

Algorithm 1: Enumeration--Experimental Running time

n	run 1 [sec]	run 2 [sec]	run 3 [sec]	run 4 [sec]	run 5 [sec]	run 6 [sec]	run 7 [sec]	run 8 [sec]	run 9 [sec]	run 10 [sec]	average time [sec]
100	0.006	0.004	0.01	0.002	0.003	0.004	0.008	0.007	0.005	0.011	0.006
200	0.089	0.109	0.078	0.078	0.087	0.095	0.089	0.111	0.109	0.125	0.097
300	0.375	0.328	0.265	0.296	0.277	0.331	0.289	0.358	0.328	0.375	0.3222
400	0.795	0.75	0.631	0.671	0.614	0.717	0.679	0.843	0.749	0.795	0.7244
500	1.31	1.295	1.085	1.201	1.108	1.226	1.163	1.419	1.31	1.342	1.2459
600	1.857	1.872	1.584	1.782	1.604	1.816	1.608	2.012	1.857	1.887	1.7879
700	2.371	2.496	2.143	2.273	2.262	2.397	2.163	2.339	2.48	3.557	2.4481
800	2.621	2.879	2.364	2.536	2.521	2.417	2.543	2.621	3.135	2.133	2.577
900	3.16	3.273	3.318	3.276	3.247	3.272	3.881	3.291	3.432	3.183	3.3333
1000	4.28	4.565	4.643	4.56	4.479	4.473	4.482	4.275	4.306	4.54	4.4603

Microsoft Excel was used to produce a curve that "fits" the data that we plotted. Regressional technique was used to find a function that models the relationship between input size n and time and to determine the equation of the function. The data is **quadratic** and the best fit equation is $T(n) = 3E\text{-}06n^2 + 0.001n - 0.1986$ with $R^2 = 0.9886$



The best fit equation is T(n) = 3E-06n2 + 0.001n - 0.1986, which suggests an experimental running time of $T(n) = n^2$. This is discrepant with our theoretical time analysis, where we anticipated a running time of $T(n) = \Theta(n^3)$.

So, its lets say n = x and lets use L'Hospital's Rule to find the lim of actual run time divided by theoretical run time as n goes to infinity.

The expected function in the theoretical analysis was $\frac{n*n(n-1)}{2}$, which can be reduced to n^3
If we set n = x and take a limit as x goes to infinity, we'll get:

$$\lim_{x\to\infty} \frac{3\times 10^{-6}\,x^2 + 0.001\,x - 0.1986}{x^3} = 0$$

This indicates that the actual running time function that we got grows at a slower rate than the predicted theoretical function and T(n) = **O(**n^3).
The possible explanation is that the actual running time might be much less than the one predicted by some function and much longer than predicted by other functions. Perhaps there is some input for which the running time is proportional to the given function, but perhaps the input is not found among those expected in practice.

Use the regression model to determine the largest input for the algorithm that can be solved in 10 minutes.
The unit of time in our equation is second. 10 min = 600 sec

$T(n) = 3E\text{-}06n^2 + 0.001n - 0.1986$
$600 = 3E\text{-}06n^2 + 0.001n - 0.1986$
$3E\text{-}06n^2 + 0.001n - 0.1986\text{-}600=0$
$3E\text{-}06n^2 + 0.001n – 600.1986=0$

$$n = \frac{-0.001 \pm \sqrt{0.001^2 - 4 * 3E - 6 * (-600.1986)}}{2 * 3E - 06}$$

$$n = \frac{-0.001 \pm \sqrt{0.0072033832}}{2 * 3E - 06}$$

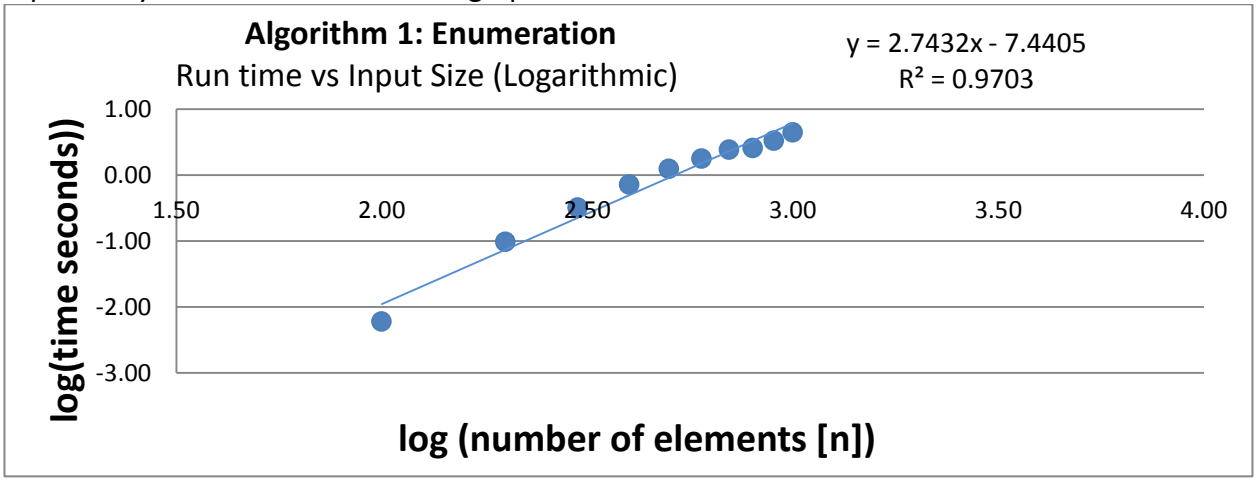
$$n = \frac{-0.001 \pm 0.0848727470982294}{2 * 3E - 06}$$

We are interested only in positive numbers, so

$$n = \frac{0.0838727470982}{2 * 3E - 06}$$

n = 13978 largest input

A log-log plot of the running times was created. The data appears somewhat linear with the following equation: y = 2.7432x - 4.4405. See graph below.



The slope of the log–log plot of the line is 2.7432, which can be rounded up to 3, suggesting a running time of T(n) = n^3. This is congruent with our **theoretical running time analysis** above, where we anticipated a time complexity of T(n) = **Θ** (n^3).

****

Algorithm 2: Better Enumeration –Pseudocode:

For the better enumeration algorithm, we need:

1. Determine the ending indices of the subarray to be summed
2. Sum the values from the starting point to the end point
3. Store the running sum for each subarray
4. If the running sum of values between the starting point and ending point is larger than previous sum, assign the new sum to maximum sum
5. Keep track of starting and ending points of the subarray for later return.
6. Keep track of the sum that will be returned later

```
//moves the starting point
 for (int i = 0; i < array.size(); i++) {
        current = 0;
        //moves the ending point and keeps rolling sum
        for (int j = i; j < array.size(); j++) {
              //computer the  running sum and store it
              current = current + array[j];
              if (current > maximum) {
                    maximum = current;
                    //stores first index of the the maximum subarray
                bestupper=j;
                    //stores last index of the the maximum subarray
                    bestlower=i;
              }}}
```

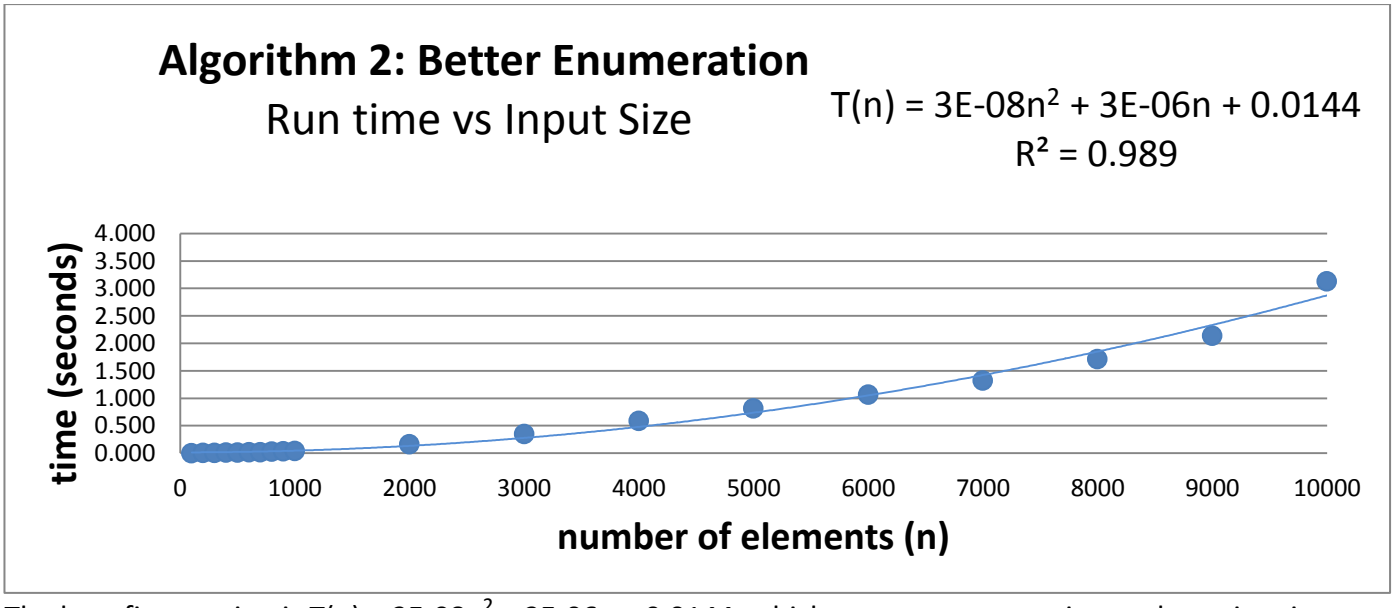
Better - Enumeration –theoretical running time analysis:
This version of the enumeration method removes the innermost addition loop by keeping a running sum for each subarray. The sum is resent when the lower bound is increased and the size of the array is zero. This optimization allows the algorithm to achieve a **Θ** (n^2) complexity. T(n) = **Θ** (n^2)

Better Enumeration - Experimental Analysis and Plots
For the experimental analysis, the algorithm was run 10 times for each of the following input sizes for example n = 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000 and 10000. See Enumeration Analysis section for explanation how arrays as random numbers were generated.
The average running time was calculated for each n and results were plotted [running times as a function of input size n]. See data and graph below:

Algorithm 2: Better Enumeration--Experimental Running time

n	run 1 [sec	run 2 [sec	run 3 [sec]	run 4 [sec]	run 5 [sec]	run 6 [sec]	run 7 [sec]	run 8 [sec]	run 9 [sec]	run 10 [sec]	time [seconds]
100	0	0	0	0	0	0	0	0	0	0	0.000
200	0	0	0.015	0	0.015	0.016	0	0	0	0	0.005
300	0	0	0.016	0	0	0	0	0	0.015	0	0.003
400	0	0.016	0	0	0.016	0.015	0	0.015	0.016	0	0.009
500	0.015	0.015	0.015	0.015	0.015	0.016	0.015	0	0	0.015	0.012
600	0.016	0.031	0.016	0.016	0.016	0.016	0.016	0.016	0	0.016	0.016
700	0.015	0.031	0.015	0.016	0.016	0.015	0.016	0.015	0.016	0.016	0.017
800	0.031	0.031	0.031	0.031	0.031	0.031	0.031	0.031	0.015	0.031	0.029
900	0.032	0.031	0.031	0.031	0.032	0.047	0.032	0.032	0.047	0.031	0.035
1000	0.047	0.047	0.047	0.047	0.047	0.031	0.031	0.031	0.031	0.049	0.040
2000	0.156	0.187	0.172	0.14	0.203	0.156	0.156	0.156	0.141	0.138	0.163
3000	0.358	0.361	0.358	0.312	0.374	0.374	0.358	0.343	0.296	0.317	0.348
4000	0.624	0.62	0.593	0.499	0.609	0.624	0.593	0.577	0.53	0.486	0.585
5000	0.769	0.89	0.78	0.748	0.811	0.905	0.905	0.812	0.718	0.759	0.815
6000	1.03	1.089	1.045	1.045	1.046	1.186	1.107	1.06	0.952	1.038	1.062
7000	1.248	1.326	1.295	1.701	1.263	1.28	1.327	1.295	1.154	1.801	1.321
8000	1.466	1.545	1.467	2.48	1.56	1.435	2.542	1.576	1.326	2.315	1.711
9000	2.73	2.074	1.701	2.465	1.685	1.607	3.18	2.246	1.528	2.645	2.135
10000	3.042	3.885	3.015	3.765	2.324	3.087	3.494	4.025	1.53	3.754	3.130

Microsoft Excel was used to produce a curve that "fits" the data that we plotted. Regressional technique was used to find a function that models the relationship between input size n and time and to determine the equation of the function. The data is quadratic and the best fit equation is T(n) = 3E-08n^2 + 3E-06n + 0.0144, R^2 = 0.989



The best fit equation is T(n) = 3E-08n^2 + 3E-06n + 0.0144, which suggests an experimental running time of T(n) =n^2. This is congruent with our theoretical time analysis, where we anticipated a time complexity of T(n) = **Θ** (n^2).

Use the regression model to determine the largest input for the algorithm that can be solved in 10 minutes.

The unit of time in our equation is second. 10 min = 600 sec

$$T(n) = 3E\text{-}08n^2 + 3E\text{-}06n + 0.0144$$

$$600 = 3E\text{-}08n^2 + 3E\text{-}06n + 0.0144$$

$$3E\text{-}08n^2 + 3E\text{-}06n + 0.0144 - 600 = 0$$

$$3E\text{-}08n^2 + 3E\text{-}06n - 599.9856 = 0$$

$$n = \frac{-3\mathrm{E}-06 \pm \sqrt{3E-06^2 - 4 * 3E-8 * (-599.9856)}}{2 * 3E-08}$$

$$n = \frac{-3\mathrm{E}-06 \pm \sqrt{0.0000719474149067889}}{2 * 3E-08}$$

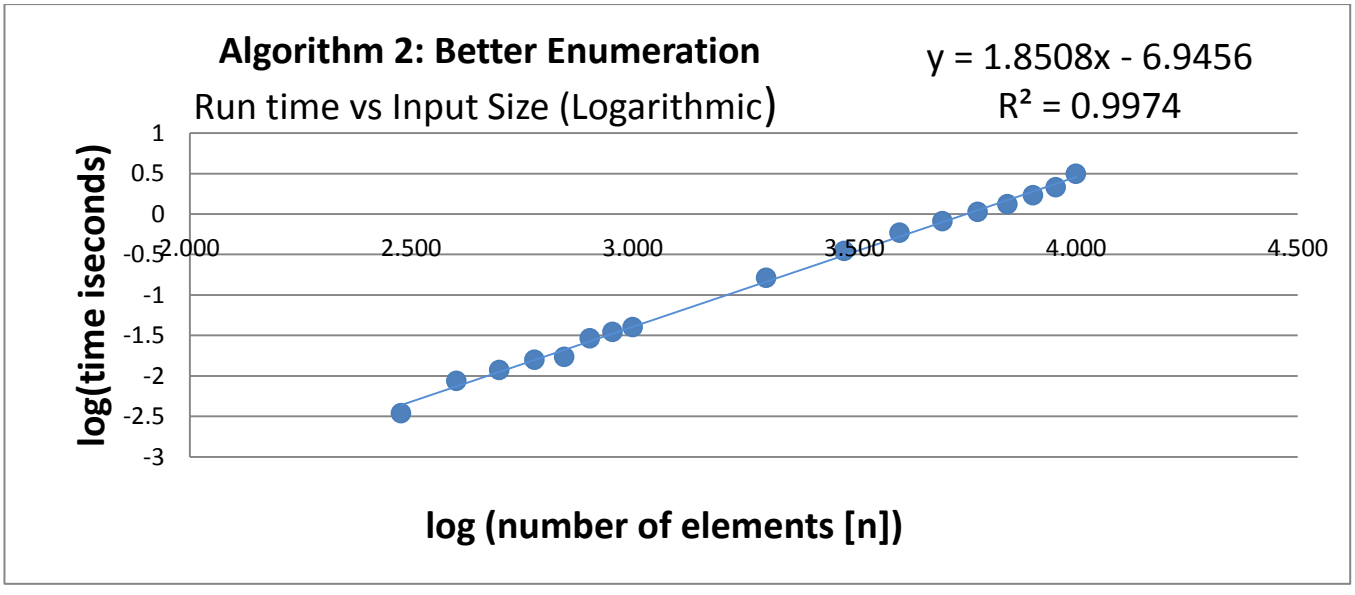
$$n = \frac{-3\mathrm{E}-06 \pm 0.008482182201933}{2 * 3E-08}$$

We are interested only in positive numbers, so

$$n = \frac{0.00711343796810471}{2 * 3E-08}$$

n = 141369 largest input

A log-log plot of the running times was created. The data appears somewhat linear with the following equation: y = 1.8508x - 6.9456. See graph below.



The slope of the log–log plot of the line is 1.8508, which can be rounded up to 2, suggesting a running time of T(n) = n^2. This is congruent with our **theoretical running time analysis** above, where we anticipated a time complexity of T(n) = **Θ** (n^2).

****

Algorithm 3: Divide and Conquer–Pseudocode:

For the divide and conquer algorithm, we need:

- Find midpoint of array
- If array is of size 1, then return sum equal to value of element and start and end position equal to elements location in array
- Else return the max sum, max sum starting position and max sum ending position for
  - Array from position 0 to the midpoint
  - Array from midpoint + 1 to end of array
  - Array from position 0 to end of array progressing from midpoint to the left and right
    - Calculate max sum of left side of array
      - Start at midpoint
      - Set max sum of left equal to midpoint value
      - For each position left of midpoint, move left 1, compare max sum left to max sum left + value of current position. Keep larger of two values and array position.
    - Calculate max sum of right side of array
      - Start at midpoint + 1
      - Set max sum of right equal to midpoint + 1 value
      - For each position to the right, move right 1, compare max sum right to max sum right + value of current position. Keep larger of two values and array position
    - Return sum of max sum left and max sum right and left/right array positions

```
//function maxSumSumArray ( array, size, start, end)
midpoint = start + (end – start) / 2;

if start – end == 0
        return maxSum;
else
        return max (maxSumSubArray( array, size, start, midpoint),
                        maxSumSubArray( array, size, midpoint+1, end),
                        maxSumOverMidPoint(array, size, start, end,
midpoint));

// function maxSumOverMidPoint (array, size, start, end, m)
maxSumLeft = INT_MIN;
maxSumRight = INT_MIN;
cumSumLeft = 0;
cumSumRight = 0;

// left side
for (i = m; i >= start; i--)
        cumSumLeft = cumSumLeft + array[i];
        if (cumSumLeft > maxSumLeft)
                maxSumLeft = cumSumLeft;

//right side
If (i = m + 1; I <= end; i++)
        cumSumRight = cumSumRight + array[i];
        if (cumSumRight > maxSumRight)
                maxSumRight = cumSumRight;

return maxSumLeft + maxSumRight;
```

Divide and conquer ----Theoretical runtime of divide and conquer:

maxSumSubArray halves the array and calls itself recursively for the first half and second half of the array giving time complexity 2 T (n / 2). It also calls maxSumOverMidPoint which runs in Θ (n) time. The runtime for the divide and conquer algorithm can be expressed with the following function:
T(n) = 2 T(n / 2) + Θ (n)
Solving for T(n) using the Master Method:
A = 2, b = 2, f(n) = Θ (n)
By case 2 of master method,
f(n) = Θ (n ^ log 2 2) = Θ (n)
then T(n) = Θ (n ^ log 2 2 lg n) = Θ (n lg n)
Solution T(n) = Θ (n lg n)

Divide and Conquer Proof by induction:

- **Simplifying assumption:** Original problem size is a power of 2, so that all subproblem sizes are integer.
- Let T(n) denote the running time of maxSumSumArray on a subarray of n elements.
- **Base case:** Occurs when high equals low, so that n = 1. The procedure just returns ⇒ T(n) = Θ (1).
- **Recursive case:** Occurs when n > 1.
- Dividing takes Θ (1) time.
- Conquering solves two subproblems, each on a subarray of n / 2 elements. Takes T(n/2) time for each subproblem ⇒ 2T(n/2) time for conquering.
- Combining consists of calling maxSumOverMidPoint, which takes Θ(n) time, and a constant number of constant-time tests ⇒ Θ (n) + Θ (1) time for combining.
- Recurrence for recursive case becomes:
- T(n) = Θ (1) +2T(n/2)+ Θ (n) + Θ (1) = 2T(n/2) + Θ (n) because Θ (1) is absorbed into Θ (n)
- The recurrence for all cases:
  - T(n)=
    - Θ(1) if n =1
    - 2T(n/2) + Θ(n) if n >1
    - Now lets try to use master method to find solution and hope that it will be expected result
    - 2T(n/2) + Θ(n)
    - a=2, b = 2, f(n) = Θ (n)
    - $n^{\log_b a} = n^{(\log a/\log b)} = n^{(\log 2/\log 2)} = n$
    - set x = n [I am using Wolfram alfa website to find limits, so that's why I do this step]
    - $\lim_{x\to\infty} \frac{x}{x} = 1$, so functions increase at the same rate
    - Because rate is the same: f(n) = Θ(n^(log2/log2))
    - From masters theorem: T(n) = Θ ($n^{\log_b a}$lg n) = Θ (n lgn)

Divide and conquer - Experimental Analysis and Plots

For the experimental analysis, the algorithm was run 10 times for each of the input sizes indicated in the table below. See Enumeration Analysis section for explanation how arrays as random numbers were generated.

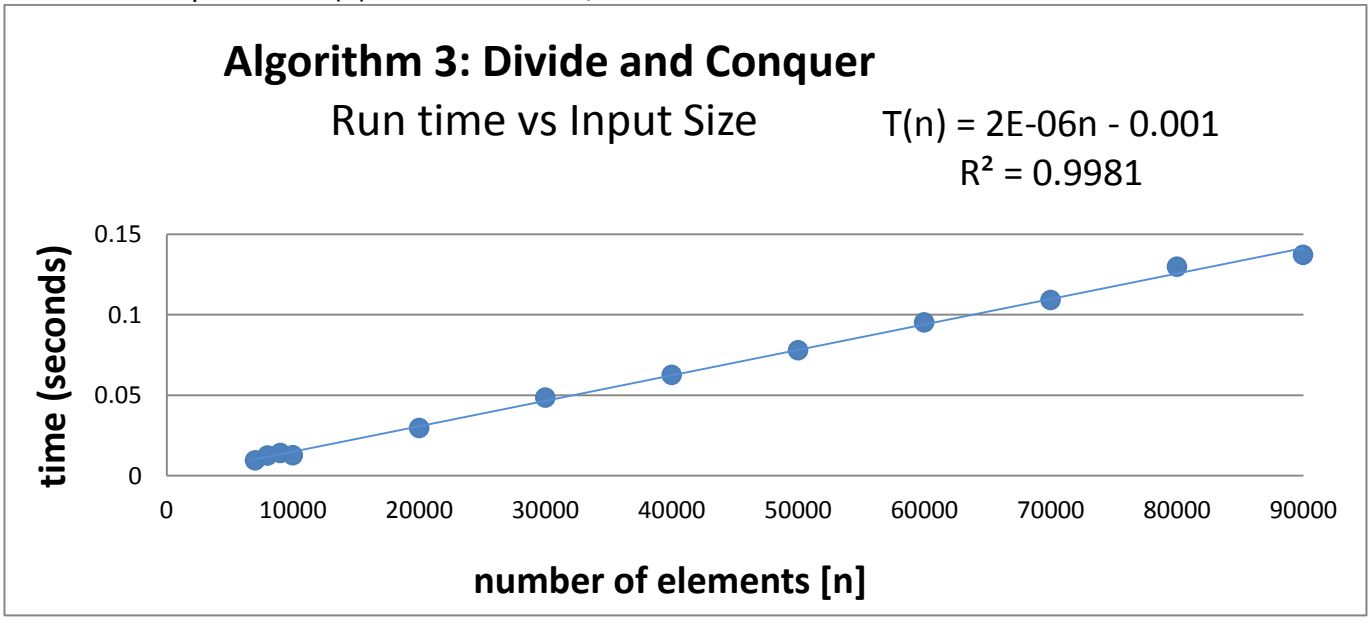
The average running time was calculated for each n and results were plotted [running times as a function of input size n]. The figure below displays the performance time of the algorithm as the size of the array grows.

Algorithm 3: Divide and Conquer--Experimental Running time

n	time 1 [sec]	time 2 [sec]	time 3 [sec]	time 4 [sec]	time 5 [sec]	time 6 [sec]	time 7 [sec]	time 8 [sec]	time 9 [sec]	time 10 [sec]	Average time [sec]
7000	0.016	0.016	0	0	0.016	0.015	0.016	0	0.015	0	0.0094
8000	0.015	0.015	0	0.016	0.015	0.016	0.015	0	0.016	0.016	0.0124
9000	0.016	0.016	0.016	0.015	0.016	0.015	0.016	0.016	0	0.015	0.0141
10000	0.015	0.016	0.016	0	0.016	0.016	0.016	0.016	0.016	0	0.0127
20000	0.047	0.031	0.031	0.016	0.031	0.015	0.031	0.031	0.031	0.031	0.0295
30000	0.047	0.047	0.047	0.046	0.063	0.047	0.047	0.047	0.047	0.047	0.0485
40000	0.063	0.063	0.062	0.047	0.078	0.062	0.062	0.063	0.078	0.047	0.0625
50000	0.078	0.093	0.078	0.062	0.078	0.062	0.078	0.078	0.093	0.078	0.0778
60000	0.109	0.094	0.094	0.078	0.094	0.093	0.078	0.093	0.109	0.109	0.0951
70000	0.109	0.125	0.125	0.078	0.109	0.109	0.093	0.109	0.125	0.109	0.1091
80000	0.156	0.125	0.14	0.11	0.125	0.109	0.125	0.141	0.142	0.124	0.1297
90000	0.141	0.14	0.156	0.124	0.141	0.125	0.124	0.125	0.156	0.14	0.1372

Microsoft Excel was used to produce a curve that "fits" the data that we plotted. Regressional technique was used to find a function that models the relationship between input size n and time and to determine the equation of the function. The line of best fit for the algorithms is **linear**.

The best fit equation is T(n) = 2E-06n - 0.001, R^2 = 0.9981



The best fit equation is T(n) = 2E-06n - 0.001, which suggests an experimental running time of T(n) =n. Its inconsistent with our theoretical time analysis, where we anticipated a time complexity of T(n) = Θ (n log(n)). But lets take the limit of the ratio of the function as number of elements goes to infinity and see what happens.

$$\lim_{x\to\infty} \frac{2\times 10^{-6}\,x - 0.001}{x\log(x)} = 0$$

So running time function grow at the slower rate, thus we can conclude that T(n) = O (n log(n)).
The possible explanation is that the actual running time might be much less or much longer than the one predicted by some function. Perhaps there is some input for which the running time is proportional to the given function, but perhaps the input is not found among those expected in practice.

Use the regression model to determine the largest input for the algorithm that can be solved in 10 minutes.

The unit of time in our equation is second. 10 min = 600 sec

T(n) = 2E-06n - 0.001

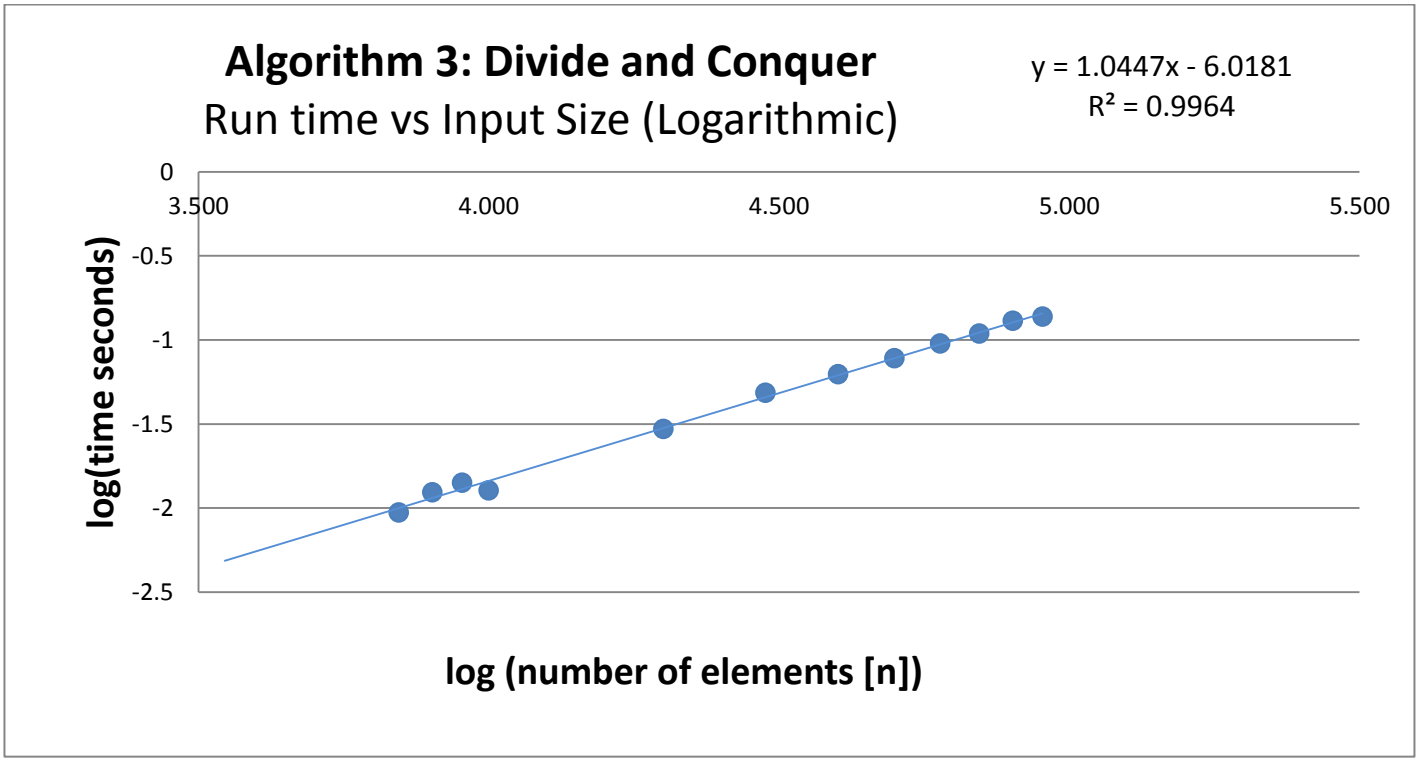
600=2E-06n - 0.001

2E-06n= 600.001

$$n = \frac{600.001}{2E-06}$$

n=300000500 largest input

A log-log plot of the running times was created. The data appears somewhat linear with the following equation: y = 1.0447x - 6.0181. See graph below.



The slope of the log–log plot of the line is 1.0447, which can be rounded up to , suggesting a running time of T(n) = n. Its inconsistent with our theoretical time analysis, where we anticipated a running time of T(n) = Θ (n log(n)). But lets take the limit of the ratio of the function as number of elements goes to infinity and see what happens.

$$\lim_{x\to\infty} \frac{2\times 10^{-6}\,x - 0.001}{x\log(x)} = 0$$

So running time function grow at the slower rate, thus we can conclude that T(n) = O (n log(n)).
The possible explanation is that the actual running time might be much less or much longer than the one predicted by some function. Perhaps there is some input for which the running time is proportional to the given function, but perhaps the input is not found among those expected in practice.

****

Algorithm 4: Linear time –Pseudocode:

For the **Linear time** algorithm, we need:

1. Scan the elements in the array
2. As it scans the elements, add the next element in the are
3. Check of the new sum is larger than the previous sum, assign the new sum to the maximum sum
4. Keep track of starting and ending points of the subarray for later return.
5. Keep track of the sum that will be returned later

```
// scan the element in the array
 for (int i = 0; i < arr.size(); ++i) {
//add the next element in array
   curr_max = curr_max + arr[i];
//check if the current sum is less than 0
   if (curr_max < 0) {
 //if current max is less than 0, set it to 0
    curr_max = 0;
  //increment the index
    first_index = i + 1;
  }
//check if new max is larger than max stored
   if (max < curr_max) {
 //if new max is larger, save it as a max
    max = curr_max;
//stores first index of the maximum subarray
    max_first_index = first_index;
```

Linear time–theoretical running time analysis:

The linear-time has a single loop that steps through the elements of the array. Every iteration performs work of a constant time because only simple comparisons are made. Thus the complexity of this algorithm is O(n). However, according to the reference: https://www.student.cs.uwaterloo.ca/~edlanglo/notes/CS341.pdf, any logarithm should take **Ω(n), so the T(n) = Θ(n)**

Linear time - Experimental Analysis and Plots

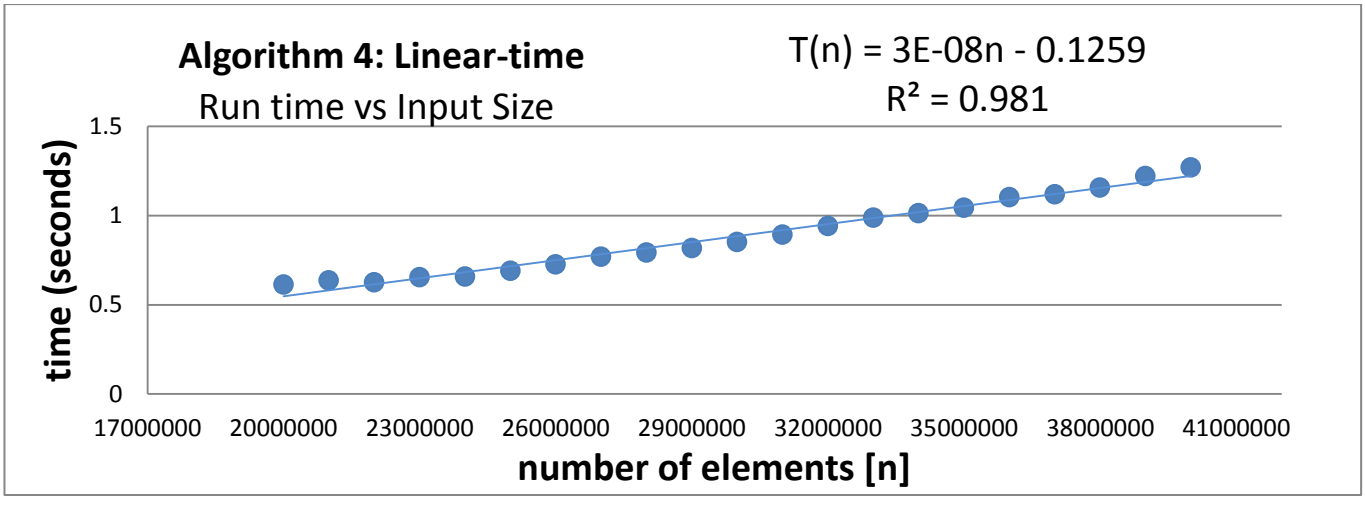
For the experimental analysis, the algorithm was run 10 times for each of the input sizes indicated in the table below. See Enumeration Analysis section for explanation how arrays as random numbers were generated.

The average running time was calculated for each n and results were plotted [running times as a function of input size n]. The figure below displays the performance time of the algorithm as the size of the array grows.

Algorithm 4: Linear-time--Experimental Running time

n	time 1 [sec]	time 2 [sec]	time 3 [sec]	time 4 [sec]	time 5 [sec]	time 6 [sec]	time 7 [sec]	time 8 [sec]	time 9 [sec]	time 10 [sec]	time [sec]
20000000	0.531	0.546	0.531	0.562	0.546	0.546	0.851	0.952	0.53	0.53	0.6125
21000000	0.562	0.593	0.577	0.577	0.577	0.577	0.811	0.936	0.592	0.562	0.6364
22000000	0.577	0.595	0.593	0.593	0.609	0.608	0.609	0.858	0.593	0.624	0.6259
23000000	0.624	0.64	0.64	0.624	0.64	0.624	0.625	0.827	0.64	0.655	0.6539
24000000	0.639	0.655	0.671	0.655	0.671	0.639	0.64	0.671	0.656	0.686	0.6583
25000000	0.671	0.671	0.733	0.702	0.686	0.671	0.671	0.686	0.67	0.749	0.691
26000000	0.717	0.733	0.749	0.718	0.749	0.686	0.687	0.702	0.733	0.78	0.7254
27000000	0.795	0.733	0.749	0.749	0.796	0.733	0.733	0.827	0.764	0.811	0.769
28000000	0.827	0.765	0.842	0.796	0.811	0.749	0.749	0.842	0.748	0.795	0.7924
29000000	0.868	0.811	0.811	0.795	0.842	0.78	0.78	0.827	0.78	0.873	0.8167
30000000	0.877	0.839	0.858	0.858	0.882	0.826	0.796	0.858	0.796	0.92	0.851
31000000	0.921	0.878	0.889	0.89	0.932	0.843	0.842	0.878	0.842	1.014	0.8929
32000000	0.921	0.937	0.921	0.952	0.967	0.858	0.952	0.965	0.936	0.998	0.9407
33000000	0.937	1.046	0.998	0.968	1.061	0.936	0.967	0.965	0.938	1.056	0.9872
34000000	0.983	1.006	1.045	0.982	0.999	0.936	1.029	1.015	0.981	1.149	1.0125
35000000	1.061	0.999	1.107	1.061	1.03	0.967	0.998	1.061	1.026	1.117	1.0427
36000000	1.107	1.108	1.123	1.201	1.092	1.051	1	1.092	1.092	1.167	1.1033
37000000	1.159	1.076	1.201	1.155	1.186	1.066	1.061	1.092	1.045	1.155	1.1196
38000000	1.24	1.093	1.185	1.189	1.202	1.158	1.061	1.154	1.092	1.186	1.156
39000000	1.193	1.17	1.248	1.185	1.295	1.245	1.155	1.185	1.279	1.263	1.2218
40000000	1.246	1.264	1.291	1.272	1.326	1.22	1.185	1.264	1.326	1.295	1.2689

Microsoft Excel was used to produce a curve that "fits" the data that we plotted. Regressional technique was used to find a function that models the relationship between input size n and time and to determine the equation of the function. The line of best fit for the algorithms is **linear**. The equation is T (n)= 3E-08n - 0.1259, $R^2 = 0.981$



The best fit equation is T(n) = 3E-08n - 0.1259, which suggests an experimental running time of T(n) =n . This is consistent with our theoretical time analysis, where we anticipated a time complexity of T(n) = **Θ** (n).

Use the regression model to determine the largest input for the algorithm that can be solved in 10 minutes.

The unit of time in our equation is second. 10 min = 600 sec

$$T(n) = 3E\text{-}08n - 0.1259$$

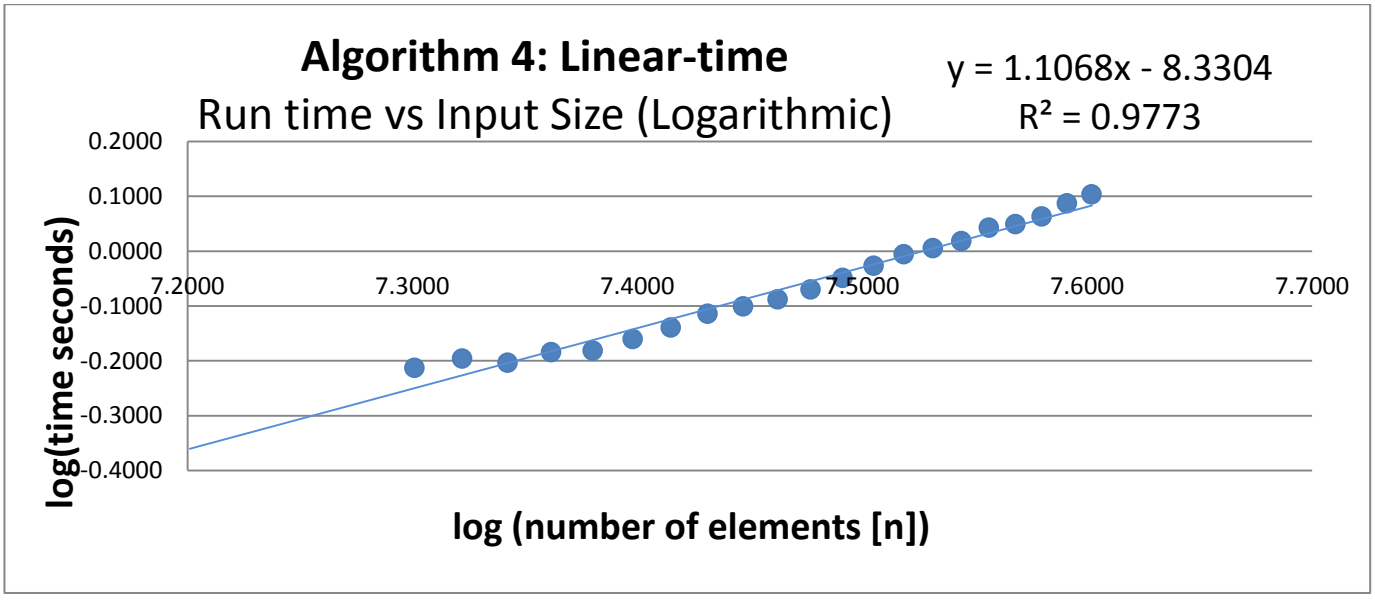
$$600 = 3E\text{-}08n - 0.1259$$

$$3E\text{-}06n = 600.1259$$

$$n = \frac{600.1259}{3E - 06}$$

n=200041966 largest input

A log-log plot of the running times was created. The data appears somewhat linear with the following equation: y = 1.1068x - 8.3304. See graph below.



The slope of the log–log plot of the line is 1.1068, which can be rounded up to , suggesting a running time of T(n) = n. This is congruent with our **theoretical running time analysis** above, where we anticipated a time complexity is T(n) = **Θ** (n).

****

Also present the results of all four Algorithms together on a single graph and/or log log plot depending on the scale

Each algorithm was run at different size ranges [see tables above for exact sizes]. The array sizes varied from 100 to 40000000, so due to scale issues we are not able to present Run time vs Input Size for all algorithms on one graph. The combined log log plot is presented below.