**Analysis of Algorithms**

**Assignment 1**

**Ryan Farber, Zach Singleton, Huarui Wu**

**Project report**

Each student will separately submit a joint group project report to Blackboard. For each of the above algorithms, your report must include:

**Mathematical analysis:** Give pseudocode for each algorithm and an analysis of the asymptotic running times of the algorithms.

**Algorithm 1:**

function maxSubArray(Array)

for each i in ArraySize:

for j = i to ArraySize :

Sum = 0

for k = i to j :

Sum = Sum + Array[k]

if Sum > max :

max = Sum

return max

**Algorithm 2**:

function maxSubArray(Array):

for each i in ArraySize :

Sum = 0

for j = i to ArraySize:

Sum = Sum + Array[j]

if Sum > max :

max = Sum

return max

**Algorithm 3:**

function MaxSubarray(Array):

if ArraySize == 1:

return Array[0]

max\_left\_sum = MaxSubarray\_recursive(Array, left\_branch)

max\_right\_sum = MaxSubarray\_recursive(Array, right\_branch)

for i = mid\_index - 1 to 0:

sum = sum + Array[i]

max\_left\_from\_mid = max( max\_left\_from\_mid , sum)

for i = mid\_index to ArraySize:

sum = sum + Array[i]

max\_right\_from\_mid = max( max\_right\_from\_mid , sum)

max\_left\_right = max(max\_left\_sum, max\_right\_sum)

max\_sum = max(max\_left\_right, max\_left\_from\_mid + max\_right\_from\_mid)

return max\_sum

**Theoretical correctness:** For the third, recursive algorithm, write a formal proof of correctness using induction.

Proof of Algorithm 3 (Divide-and-Conquer) Using Induction:

**Claim A:**

Array *r* contains *z* integers r\_0, r\_1, . . . r\_z-1 for z > 0.

Also, the divide-and-conquer algorithm has accurately calculated the sum of the maximum subarray, p = , for integers I, j < z.

We will show both a top-down and bottom-up inductive proof.

* all.sums : the sum of all the elements in an array
* left.sums : the largest sum starting from the left
* right.sums : the largest sum starting from the right
* overall.sums : the overall max sum

TOP-DOWN PROOF:

*Base case:*

* let z = 1
* Then all.sums = A[0], left.sums = A[0], right.sums = A[0], overall.sums = A[0]

*Inductive hypothesis:*

* + left\_sums = recursiveMaxSubarray(A[ 0 **:**  ])
  + right\_sums = recursiveMaxSubarray(A[ **:** z ])
  + all.sums = left\_all.sums + right\_all.sums
  + left.sums = max(left\_left.sums, left\_all.sums + right\_left.sums)
  + right.sums = max(right\_right.sums, left\_right.sums + right\_all.sums) overall.sums = max(all.sums, left.sums, right.sums, left\_right.sums + right\_left.sums)

*There are three cases that we consider:*

* *1st Case*: This case is completely contained in the first half. This will be returned as left\_overall.sums from the recursive all on left\_sums.
* *2nd Case*: This case is completely contained in the second half. This will be returned from the recursive call on right\_sums.
* *3rd Case:*  This case is made of a suffix of the first half of the maximum sum and the prefix of the second half of the maximum. This can be found using the left\_right.sums + right\_left.sums

BOTTOM-UP PROOF:

*Base case:*

let z = 1

Then recursiveMaxSubarray(z) = *r0,* which is true

Inductive hypothesis:

We assume that the algorithm correctly computes the sum of the maximum subarray, for *z* > 1 and *z* ≤ *q* for some integer *q* > 1.  
.

Consider an array of size *z* = *q* + 1. Then we can consider one of four cases regarding the location of the maximum subarray within the whole array.

* *1st Case*: *p* = *r*. We correctly locate *p* in left.sums.
* *2nd Case*: *p* = for *j* < *q.* We correctly locate *s* in left.sums.
* *3rd Case****:*** *p* = , for *i* > 0. We correctly locate *s* in right.sums.
* *4th Case*: *p* = , for 0 < *i* ≤ *j* < q. We correctly locate *s* in *m*.

In all four cases, we correctly select the maximum sum among all.sums, left.sums, right.sums, and *m* as the sum of the maximum subarray of *r*.

**Claim B**: This is for the termination of the algorithm

**Proof**: Let z be the size of the array of integers, *r*. For *z* > 1, the recurrence for the recursive step of the algorithm can be found to be

T(*z*) = θ(1) + 2T + θ(*z*) + θ(1)

= 2T + θ(*z*),

Where the base case takes θ(*z*), the recursive calls take 2T , the maximum calculations take θ(*z*), and the final return takes θ(1).

Rewritten it shows:

T(*z*) =

Suppose T(*z)* ≤ *cz log z* + *z* = O(*z log z*). Then,

T(z) ≤ 2

≤ *cz*  + *z*

= *cz log z* – *cz log* 2 + *z*

≤ *cz log z*

= O(*z log z),*

And this is the outcome that we expected.

**Testing:** In order to get credit for this project, your code must produce correct responses. Test against the document provided above to ensure proper behavior.

set #1

expected: 239

algo1: 239

algo2: 239

algo3: 239

set #2

expected: 322

algo1: 322

algo2: 322

algo3: 322

set #3

expected: 305

algo1: 305

algo2: 305

algo3: 305

set #4

expected: 271

algo1: 271

algo2: 271

algo3: 271

set #5

expected: 281

algo1: 281

algo2: 281

algo3: 281

set #6

expected: 215

algo1: 215

algo2: 215

algo3: 215

set #7

expected: 207

algo1: 207

algo2: 207

algo3: 207

set #8

expected: 309

algo1: 309

algo2: 309

algo3: 309

set #9

expected: 195

algo1: 195

algo2: 195

algo3: 195

set #10

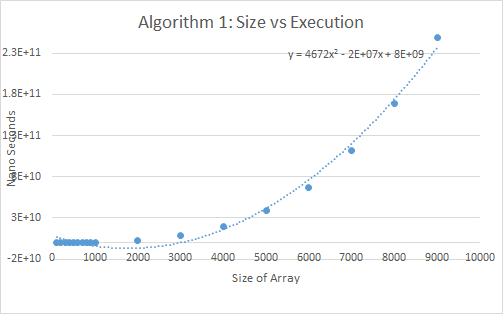
expected: 390

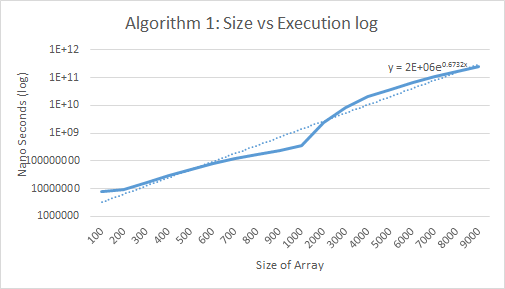
algo1: 390

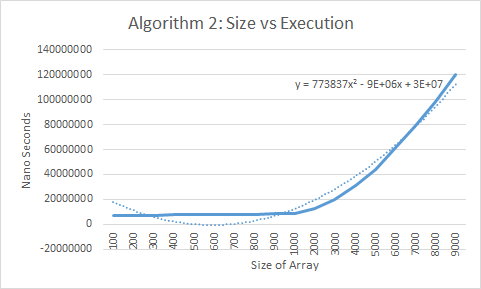
algo2: 390

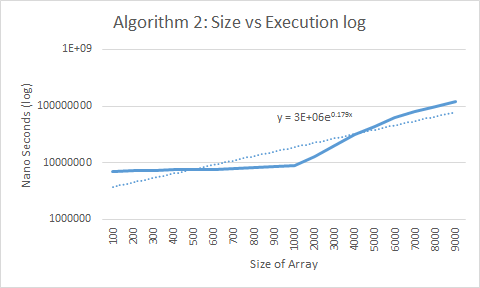
algo3: 390

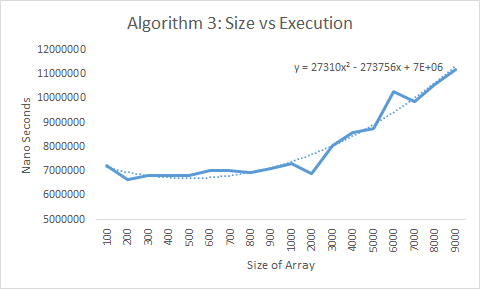
**Experimental analysis:** Perform an experimental analysis and include plots as described above.

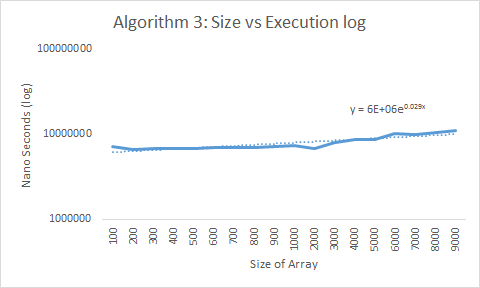












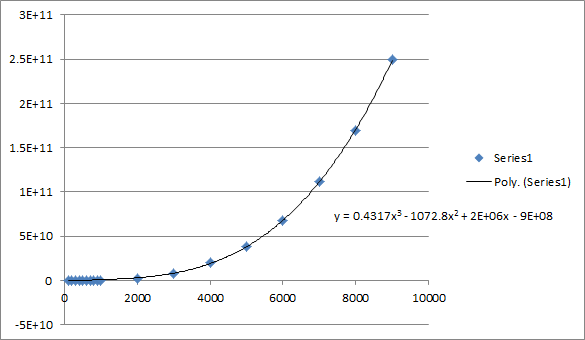
**Extrapolation and interpretation:** Use the data from the experimental analysis to answer the following questions:

1. For each algorithm, what is the size of the biggest instance that you could solve with your algorithm within one hour?

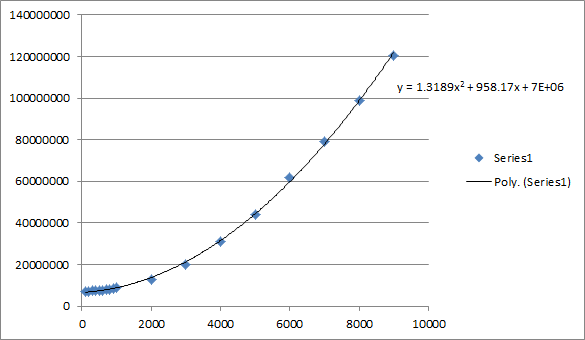
2. Determine the slope of the lines in your log-log plot and from these slopes infer the experimental running time for each algorithm. Discuss any discrepancies between the experimental and theoretical running times.

The three graphs and equations all fit the Mathematical analysis that the algorithms will perform at O(n^3), O(n^2) and O(nlog(n)).

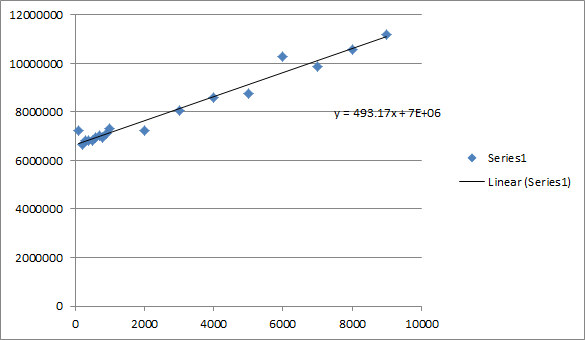
Algorithm 1



Algorithm 2



Algorithm 3



We estimate the n(log n) equation’s running time with a linear equation because it is a recursive divide and conquer problem.

For any recursive algorithm that divides its input of size n into a pieces of size n/b and takes a time f(n) to perform the division and recombination, the running time satisfies T(n)=a⋅T(n/b)+f(n). This leads to a closed form that depends on the values of a and b and the shape of f. If a=b and f(n)=Θ(n), the master theorem states that T(n)=Θ(nlogn). See here [master theorem](http://en.wikipedia.org/wiki/Master_theorem).

**Note that 1 hour = 3.6\*10^12 Nanoseconds**

**Plugging in 3.6\*10^12 Nanoseconds for Y, yields the following value for Algorithm 1, Algorithm 2 and Algorithm 3:**

Algorithm 1 - y = 0.4317x3 - 1072.8x2 + 2E+06x - 9E+08

X = 21064.2

Algorithm 2 - y = 1.3189x2 + 958.17x + 7E+06  
  
X = 1651769.4

Algorithm 3 - y = 493.17x + 7E+06

X = 7299699900.6

We expect the ratios between the slopes to be approximately 1:2:3.

The loglog graphs show that algorithm one has the largest slope. It is 3 times larger than algorithm 2’s slope. Algorithm 3 has the smallest slope. Many many times smaller than algorithm 2. The differences between the slopes are larger than we expected with the mathematical analysis but it’s still consistent with the order of speeds of the algorithms. Much of the difference with the expected slopes could be due to the equation not being very accurately estimated, as well as the small sample of data.

