CS 325 Winter 2015
Group 8
Bryant Hayes
Keenan Bishop
Nathan Ahrens

Project 1: Maximum Sum Subarray

Theoretical Run-time Analysis

Algorithm 1: Enumeration

```
n = array.length
for i from 0 to n
    for j from 0 to n
        for k from i to j
            compute current sum from i to k
            if sum > max sum
            max sum = current sum
return max sum
```

Since this algorithm involves 3 loops with max size n, the running time is O(n^3).

Algorithm 2: Better Enumeration

```
n = array.length
for i from 0 to n
    for k from i to n
        compute current sum from i to k
        if sum > max sum
        max sum = current sum
return max sum
```

This algorithm cuts out the redundancy of computing the sums from k up to j-1 and k up to j separately, and thus cuts out one entire loop. The remaining two loops of max size n gives this algorithm a complexity of O(n^2).

Algorithm 3: Divide and Conquer

```
recursive function maxSubArray(array, low, high)
if low == high
     return low
```

```
middle of array = (low + high)/2

return max of these 3: (maxSubArray(left half), maxSubArray(right
half), max sum of combined)
```

This algorithm uses the divide and conquer method. It splits the problem into smaller pieces, which is a logarithmic complexity action. It does this n times, so the total complexity is O(nlogn).

Algorithm 4: Linear-time

```
for i from 0 to n
    current sum = max(0, current sum + value at i)
    max sum = max(max sum, current sum)
return max sum
```

This algorithm is known as Kadane's Algorithm. At every step, it checks whether the current sum + the value at the current index is bigger than 0. If it is, it continues on, and if not, it resets the current sum to zero, as negative numbers are not desired in our case. Otherwise it updates the max sum if the current sum is larger. This form of the algorithm requires at least 1 positive number (in that case the single number is the greatest subarray). The algorithm consists of only 1 loop through the array, so it is bounded by a linear O(n).

Proof of Correctness

Maximum Subarray Proof for Algorithm 3

The maximum sub array works by dividing an array, A[low …. high] into 2 subarrays of as equal size as possible, where low indicates the first index and high the last. This midpoint, mid is equal to (low+high)/2 so that the left array is A[low .. mid] and the right is A[mid+1 … high].

Any subarray of A[low … high] is given as A[i … j] where it must be:

1. entirely within the subarray: A[low … mid]
2. entirely within the subarray: A[mid+1 … high]
3. contained in both arrays, crossing the midpoint where low ≤ i ≤ mid ≤ j ≤ high

The first 2 conditions can be solved recursively, as they are subsets of the original problem; finding the maximum subarray where i and j correspond to low and high respectively.

This leaves only the last condition to solve: finding the maximum subarray crossing both arrays. We're guaranteed at least one positive integer in our array (as stated in the HW), so our sums for comparison are initialized at 0. Through the algorithm below, the max subarray crossing the midpoint is calculated and this sum returned. In Python:

```python
def maxMiddleSum(array, low, mid, high):
	sum = 0
	leftSum = 0
	for i in range(mid, low-1, -1):
		sum += array[i]
		if sum > leftSum:
			leftSum = sum
	sum = 0
	rightSum = 0
	for i in range(mid+1, high+1, 1):
		sum += array[i]
		if sum > rightSum:
			rightSum = sum

	return (leftSum + rightSum)
```

The original algorithm has a base case, where if the low index is equal to the high index, the array is size one and the single value it contains is returned. Then the mid point is calculated as before, the array passed (initially the whole array A[low...high]) is further divided into 2 closest equal parts by a recursive call to itself, and the maxMiddleSum is called. After the 2 recursive calls to maxSubArray complete, the max subarray sum value is returned, and all 3 are compared (via function max3).

```python
def maxSubArray(array, low, high):
	# BASE CASE
	if low == high:
		return array[low]
```

```
        mid = (low + high) / 2

        # Return the maximum of left sum, right sum, and combined(middle) sum
        return max3(maxSubArray(array, low, mid), maxSubArray(array, mid+1, high),
                        (maxMiddleSum(array, low, mid, high)))
```

The highest sum is then found and returned:

```
def max3(a, b, c):
        return max2(max2(a, b), c)

def max2(a, b):
        if a > b:
                return a
        else:
                return b
```

To prove the recursive correctness of it, we must:

1. Prove the partial correctness of all recursive calls.
2. Prove the calls absolutely terminate.

For part 1, we must prove:

- The preconditions of each step are satisfied
- The post conditions are satisfied by the call

The algorithm performs a number of steps, divided in our code by functions. Each has pre and post conditions:

maxSubArray (the start of our algorithm)

- Precondition:
  - Size $n \geq 1$; the array given is of at least size 1
  - The passed array has at least one positive number
  - The passed variables satisfy $low \leq high$
  - low and high both are indices in the array
- Postcondition:
  - The greatest sum of any subarray $A[i \ldots j]$ is returned

Next, 2 recursive calls are made to maxSubArray (explained above), followed by a call to the function, maxMiddleSum, that finds and sums the best subarray strafing the midpoint.

maxMiddleSum

- Precondition:
  - The passed bounds for the array is size $n>1$
  - The values low, mid, and high are indices in the array
  - These values satisfy that $low \leq mid \leq high$
- Postcondition:
  - The greatest sum of any subarray strafing the midpoint is returned

Finally, the comparison between the left, right, and strafing arrays is made:

max3 (and consequently the utility function, max 2)

- Precondition:
  - The 3 values passed are all integers ≥ 0
- Postcondition:
  - The greatest of the 3 is returned

These are proven inductively, which proposes a theorem, T, and also has 2 requirements:

1. Base Case: Our theorem, T, holds for an array of size n=1
2. Inductive Step: for every n>1, if T holds for n-1, T holds for n

Our algorithm has 3 cases to consider (covered initially), plus a 4th where the array is of size n=1. It's initially started by a call to maxSubArray. Our theorem T is:

$$maxsubarray = max\sum_{i=0}^{n-1} A_i$$

The base case (n=1) is solved in the first step of maxSubArray: if low == high, the function returns the single value of size 1 array, and trivially the max sum of an array of size 1 is the only value in the array. This satisfies the precondition for the maxMiddleSum step; that the size of the passed bounds must be greater than 1, as a value of 1 would return early. Additionally, as the base case returns the value at that single index, the precondition for max3 is partially satisfied

Next, the inductive step: for all n>1, any array of size k satisfies T, when k=n-1, or, n=k+1. The first recursive call (case 1), where the left sum is captured is given as:

mid = (i+j)/2 where i=0 and j=n, thus
mid = n/2 = (k+1)/2;
Thus,
$0 \leq$ low, and low $\leq (k+1)/2 \leq k+1$ for all n>1

This satisfies the precondition for recursive maxSubArray calls and maxMiddleSum that , low≤high, and for maxMiddleSum specifically that low≤mid≤high , as true.

The recursive call is made, and the subarray is further divided, passing the values for the left array:

$$s=\sum_{i=0}^{j} i=\sum_{i=0}^{n/2} A_i=\sum_{i=0}^{(k+1)/2} A_i$$

Case 2 (where the max subarray is located in A[mid+1 … n]) follows the same:

mid = (i+j)/2 where i=0 and j=n, thus

mid = n/2
mid = (k+1)/2 which is true for n>1

The recursive call is then made where the right sum is calculated as:

s=∑i=mid+1ji=∑mid+1nAi=∑i=k+12k+1Ai
Case 3 (where the midpoint is crossed) is evaluated as:

s=∑ijAiwhere $0 < i \le j < n$; $n = k + 1$

Furthermore, the max sum is calculated as the max right-most sum in the left array, and the left-most sum in the right array:

max∑i=lowmidAi+max∑i=mid+1highAi

This recursively combines the results of the left and right sums. We're guaranteed a sum of at least 0, and so:

left sum ≤ max sum ≥ right sum

Thus guaranteeing the sum returned will be the best one or 0. This completes the precondition for max3, in that it returns a value greater or equal to 0

Now the final step can be evaluated; the method max3, which returns the highest sum of the 3 cases. It has the precondition of being given the highest sum of all 3 cases, which have shown to be satisfied. As stated in the problem, at least one value in the array A must be positive, and so 0 can never be returned in the final comparison.

Finally, the termination case: The algorithm terminates for all n≥0. The array size n must be at least one (as given in the problem), and the algorithm returns. The base case is called when the subarray is of size 1, ending the recursive calls and proving the base case.

The inductive hypothesis assumes n>1, and following: n=k+1. The array will be divided into subarrays of closest equal size, where the size is less than or equal to k+1. Thus, the algorithm is guaranteed to return for each subarray.

The recursive calls are guaranteed to be limited (eventually reaching n=1, the base case) as the precondition of each recursive call has been proven to satisfy that n>1. Following that k+1=n, and since the division of the arrays is by 2, $k > (k+1)/2 \ge 1$ is true for all positive n.

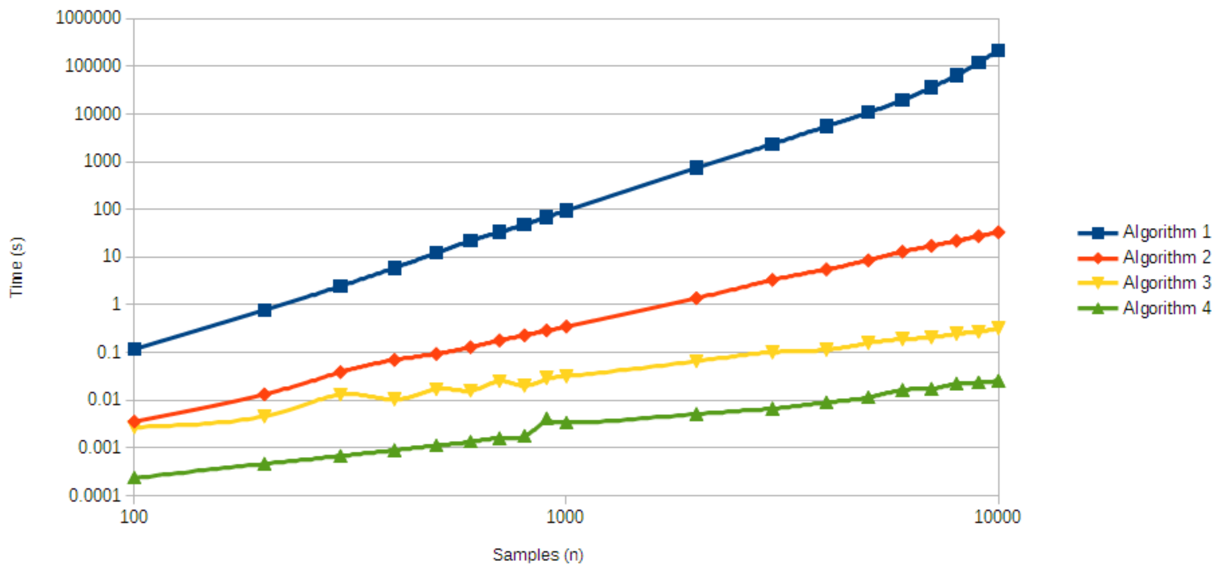
Testing

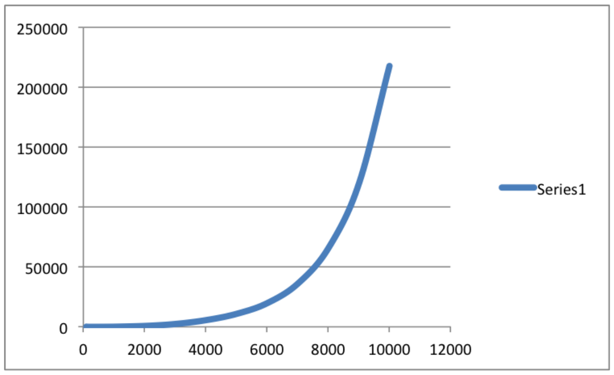
After finishing the algorithms and file I/O capabilities we could begin testing for correctness. Using the MSS_TestSet.txt we were able to check to see if our algorithms returned the correct results (subMaxArray and maxSum). Before is the print out after running algorithm #4 against the MSS_TestSet.txt file. All four algorithms got the same result.

```
Original Array: [1, 4, -9, 8, 1, 3, 3, 1, -1, -4, -6, 2, 8, 19, -10, -11]
Maximum Sum: 34
Max SubArray: [8, 1, 3, 3, 1, -1, -4, -6, 2, 8, 19]
Runtime: 4.05311584473e-06

Original Array: [2, 9, 8, 6, 5, -11, 9, -11, 7, 5, -1, -8, -3, 7, -2]
Maximum Sum: 30
Max SubArray: [2, 9, 8, 6, 5]
Runtime: 2.86102294922e-06

Original Array: [10, -11, -1, -9, 33, -45, 23, 24, -1, -7, -8, 19]
Maximum Sum: 50
Max SubArray: [23, 24, -1, -7, -8, 19]
Runtime: 3.09944152832e-06

Original Array: [31, -41, 59, 26, -53, 58, 97, -93, -23, 84]
Maximum Sum: 187
Max SubArray: [59, 26, -53, 58, 97]
Runtime: 2.14576721191e-06

Original Array: [3, 2, 1, 1, -8, 1, 1, 2, 3]
Maximum Sum: 7
Max SubArray: [3, 2, 1, 1]
Runtime: 2.14576721191e-06

Original Array: [12, 99, 99, -99, -27, 0, 0, 0, -3, 10]
Maximum Sum: 210
Max SubArray: [12, 99, 99]
Runtime: 2.14576721191e-06

Original Array: [-2, 1, -3, 4, -1, 2, 1, -5, 4]
Maximum Sum: 6
Max SubArray: [4, -1, 2, 1]
Runtime: 1.90734863281e-06
```

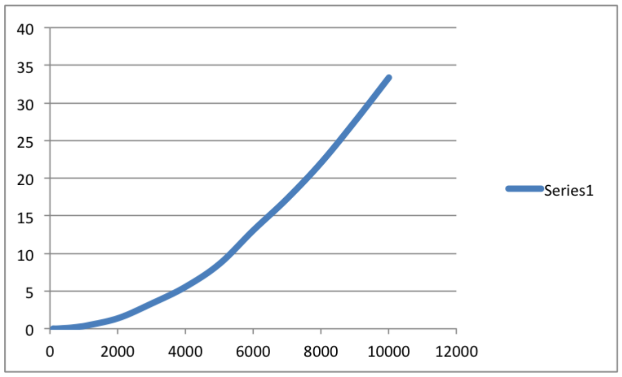
After comparing these results to the given correct answered we were reassured that are algorithms are correct.

Experimental Analysis

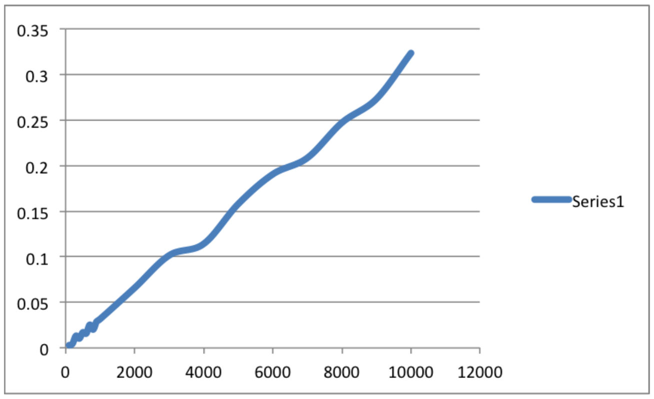




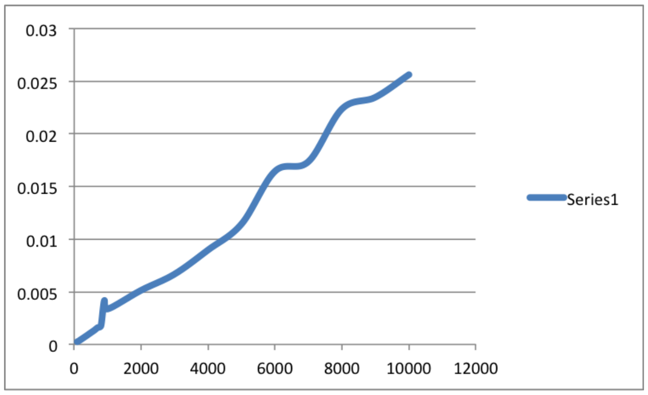
Algorithm #1



Algorithm #2



Algorithm #3



Algorithm #4

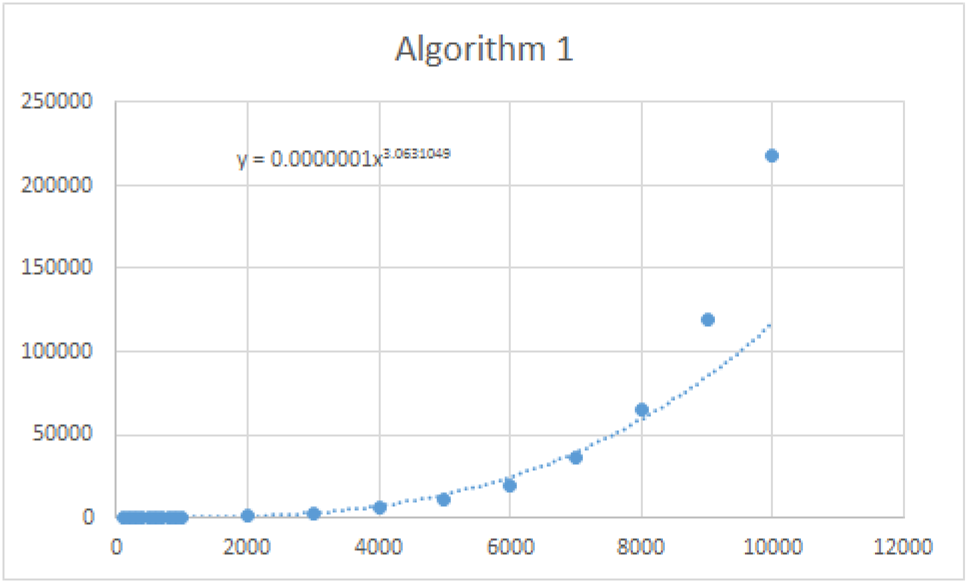
Extrapolation and Interpretation

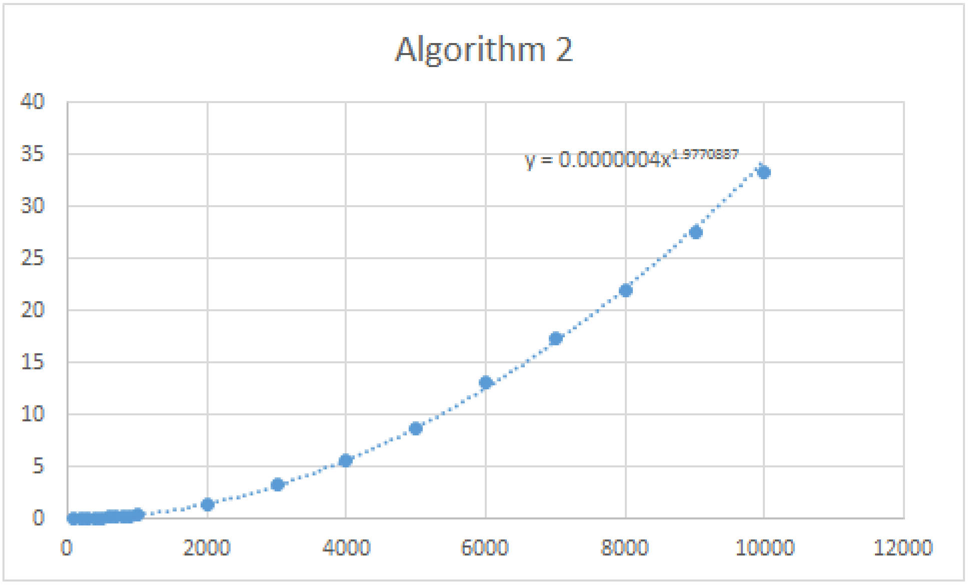
1- For each algorithm use the experimental data to estimate a function that models the relationship between running times and input sizes (n). Discuss any discrepancies between the experimental and theoretical running times.
Excel was used to find the best fit equations for each algorithm. The trend lines can be seen below.

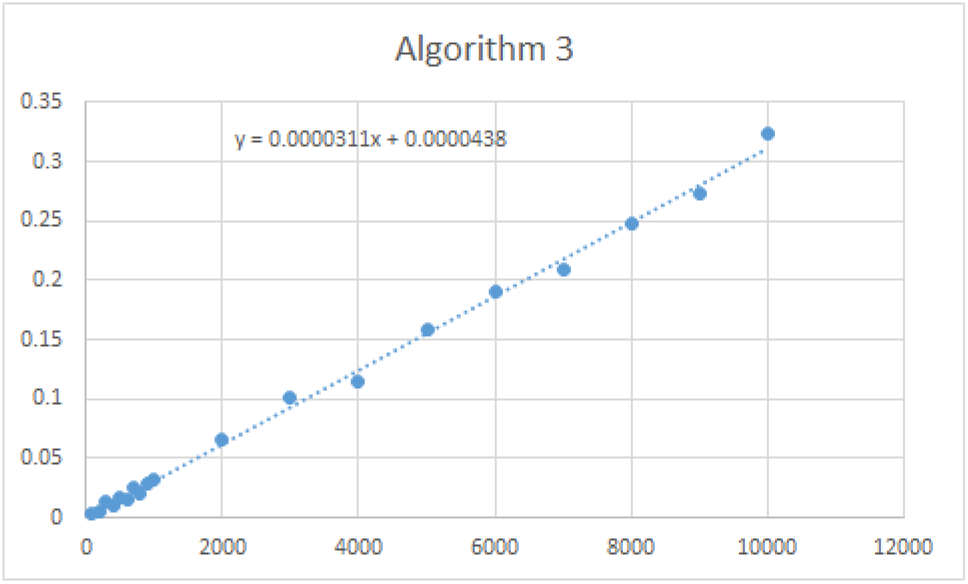
Looking at the first two, the exponent on the equation closely matches our expected running times. For Algorithm 1, we got c*($n^{3.06}$), which matches the running time within an expected error, O(n^3). Algorithm 2 had the form of c*($n^{1.98}$), which matches the estimated running time of O(n^2).

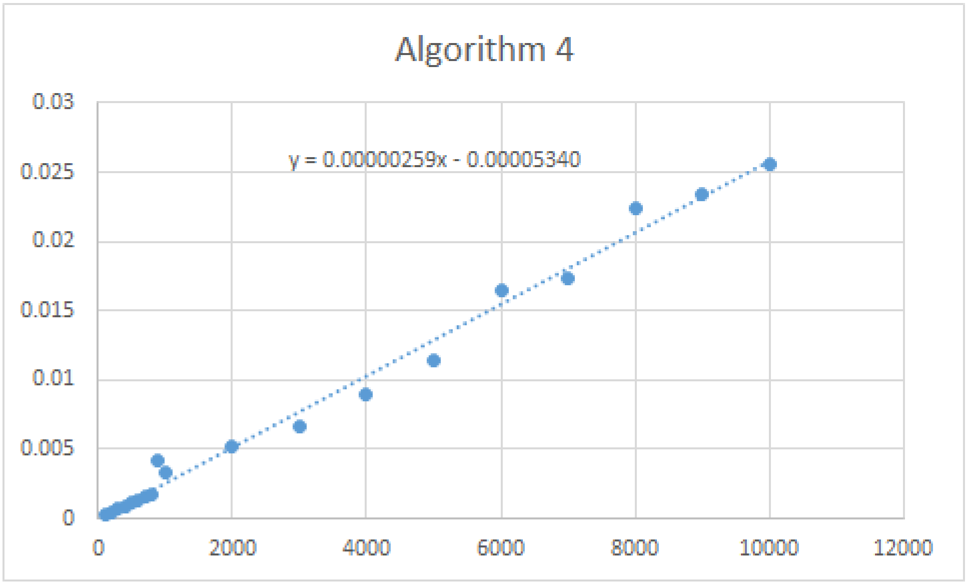
For Algorithm 3, the trend line matched up the best with the linear setting. This was noted by our instructor. The "nlogn" complexity for this range of data and type of function is small enough to almost be constant.

Algorithm 4 followed a linear model pretty well. Any deviations come from the computer or language itself, but overall the trend line is reliable for extrapolation.


Algorithm 1
y = 0.0000001x^3.0631049
250000
200000
150000
100000
50000
0
0
2000
4000
6000
8000
10000
12000


Algorithm 2
y = 0.0000004x^1.9770887
40
35
30
25
20
15
10
5
0
0
2000
4000
6000
8000
10000
12000





2- For each algorithm, what is the size of the biggest instance that you can solve with your algorithm within one hour?

Using our data and trend line equations for y = 3600s.:
Algorithm 1: About n = 3200
Algorithm 2: About n = 108,000
Algorithm 3: About n = 1.16 * 10^8
Algorithm 4: About n = 1.39 * 10^9