**Group Members**: Barbara Hazlett, Wing Lee Zerlina Yeung, Ethan Spiro
**CS 325**:Project 1 - Max Sub Array
**Language**: C++

---

*NOTE TO GRADER: we assumed the max sub array of an all negative array has a min length of 1, and is thus the sum is a negative number (not the empty set which has sum 0). Our code reflects this.

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

~~~Algorithm 1: Enumeration~~~

```
set maxsofar to 0
for i equals 0 to less than size of array
        for j equals i to less than size of array
                set sum to 0
                for k equals i to j
                        set sum to sum plus value at array[k]
                set maxsofar to whichever is the greater value of maxsofar and sum
```

The asymptotic run time is O(n^3) because there are three nested for loops that traverse the array.

~~~Algorithm 2: Better Enumeration~~~

```
set maxsofar to 0
for i from 0 to less than size of array
        set sum to 0
        for j from i to less than size of array
                set sum to sum plus value at array[j]
                set maxsofar to whichever is greater, maxsofar or sum
```

The asymptotic run time is O(n^2) because there are two nested for loops that traverse the array

~~~Algorithm 3: Divide and Conquer~~~

```
If the start is bigger than end, return 0
If start equals to end, return the first element of the array

Find and set mid to middle position of array

Set lmax, lmin and sum to zero
```
~~~

```
from middle to start of array, going down by one
    increment the sum to sum plus the value at array position
    if the sum is bigger than the left max,
        then the left max takes the value of sum
    else
        left min (we mean by that the left max when its all negatives) takes the value of sum
        the sum of left min equals the sum of left min plus the left min
        if the sum of left min is smaller or equal than the left min
            the left min stays the same
        else
            the left min equals to the value at array position

Set right max and sum to zero

from mid + 1 to end of array, going up by one
    if the sum is bigger than the right max,
        then the rightmax takes the value of sum
    else
        right min (we mean by that the right max when its all negatives) takes the value of sum
        the sum of right min equals the sum of right min plus the left min
        if the sum of right min is smaller or equal than the right min
            the right min stays the same
        else
            the right min equals to the value at array position
```

Call alogrithm3 recursively for left array, and right array. Find the max of that. Find the max of previous sum and sum of left max + right max (cross section.) Store this final max in a variable.

If final is zero, it means we have an all negative array, in that case do the same as above, but compare to left min + right min for the cross section.

else return the final max

The asymptotic run time is O(n log n).

---

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

**Claim:** We want to prove that algorithm3(), which takes an array a[1,...,n] where n is arbitrary and stores integers (both positive and negative), will return the sum of the maximum subarray such that:

$$algorithm3(a[n]) \ = max(i \ \leq j) \ \sum_{k=i}^{j} a[k]$$

**Base Case:** When n = 1, algorithm3(a[n]) returns the largest subarray, which is the single element in a[0]..

**Recursive Case:** We have 3 different scenarios that occur when n > 1.
1) The maximum subarray is in the first half of the array.
2) The maximum subarray is in the second half of the array.
3) The maximum subarray crosses the midpoint between the first and second half of the array.

**Inductive hypothesis:** Assume that a[k] is an array of arbitrary size k and algorithm3(a [ k]) returns the sum of the maximum subarray. We must prove that alogrithm3(a[k + 1]) returns the sum of the maximum subarray for our array with size k + 1.

**Inductive step:** Using the inductive hypothesis, we want to prove alogrithm3(a[k +1]) returns the sum of the maximum array.

1) Let us suppose that we have an array[k + 1] where the maximum subarray is contained in the first half of the array. This means that a[1, …, floor((k + 1)/2)] must have our maximum subarray. We only need to consider array ‘a’ of size floor((k+1)/2), and we can see that floor((k+1)/2) <= k. Using our inductive hypothesis, we can see that algorithm3(a[1,..., floor((k+1)/2)] will return the sum of the maximum subarray.

2) Let us suppose that we have an array[k + 1] where the maximum subarray is contained in the second half of the array. This means that a[floor((k + 1)/2) + 1, …, k] must have our maximum subarray. We only need to consider array ‘a’ of size floor((k + 1)/2) + 1, and we can see that floor((k+1)/2) + 1<= k. Using our inductive hypothesis, we can see that algorithm3(a[floor((k+1)/2) + 1, …, k] will return the sum of the maximum subarray.

3) Let us suppose that we have an array[k + 1] where the maximum subarray crosses the midpoint between the first and second half of the array. We take the maximum of the left sum in a[1,..., floor((k+1)/2)] and add that to maximum of the right sum in a[floor((k+1)/2) + 1, …, k]. With the above two statements, we already know and proved by induction that we can find the maximum in the left part or the right part of the array. Thus, by our inductive hypothesis if the max subarray is in the middle, and we can find the max of the left part and the right part, we can add them to get the max subarray for a subarray that spans the left and right part.

**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.*

All tests passed.

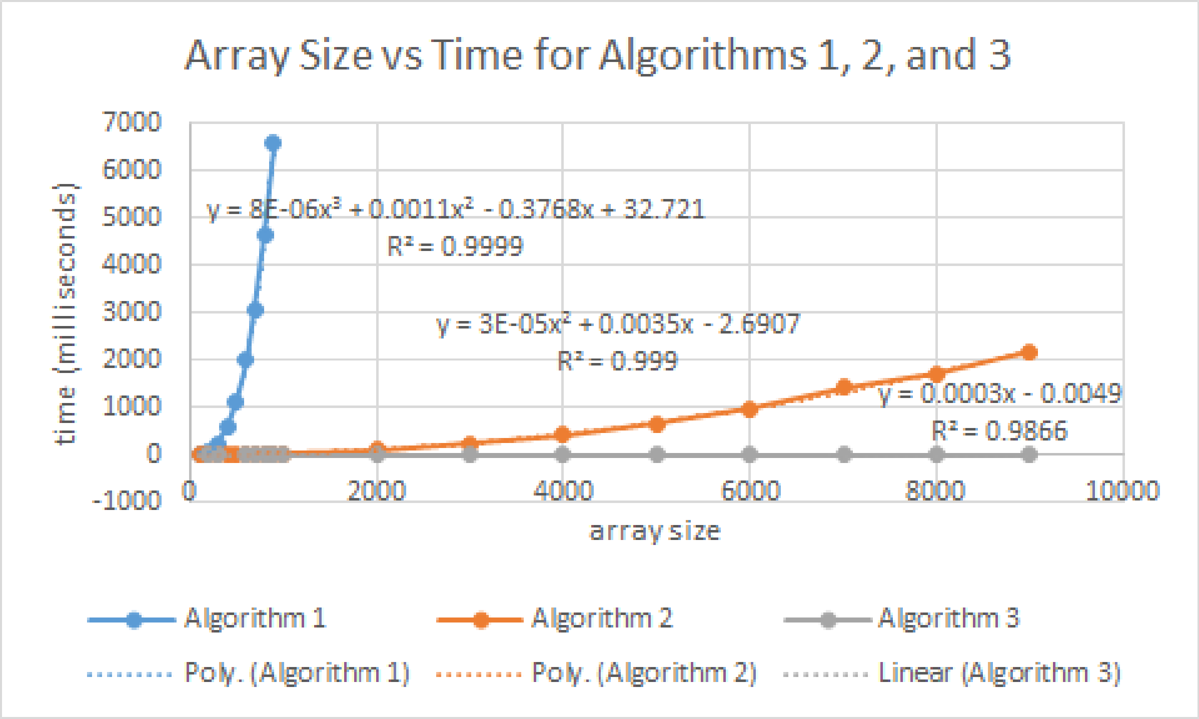
*for testing code see attached file: cs325_project1_testCode.cpp
Compile line on flip: g++ -std=c++0x cs325_project1_testCode.cpp -o <ex. name>

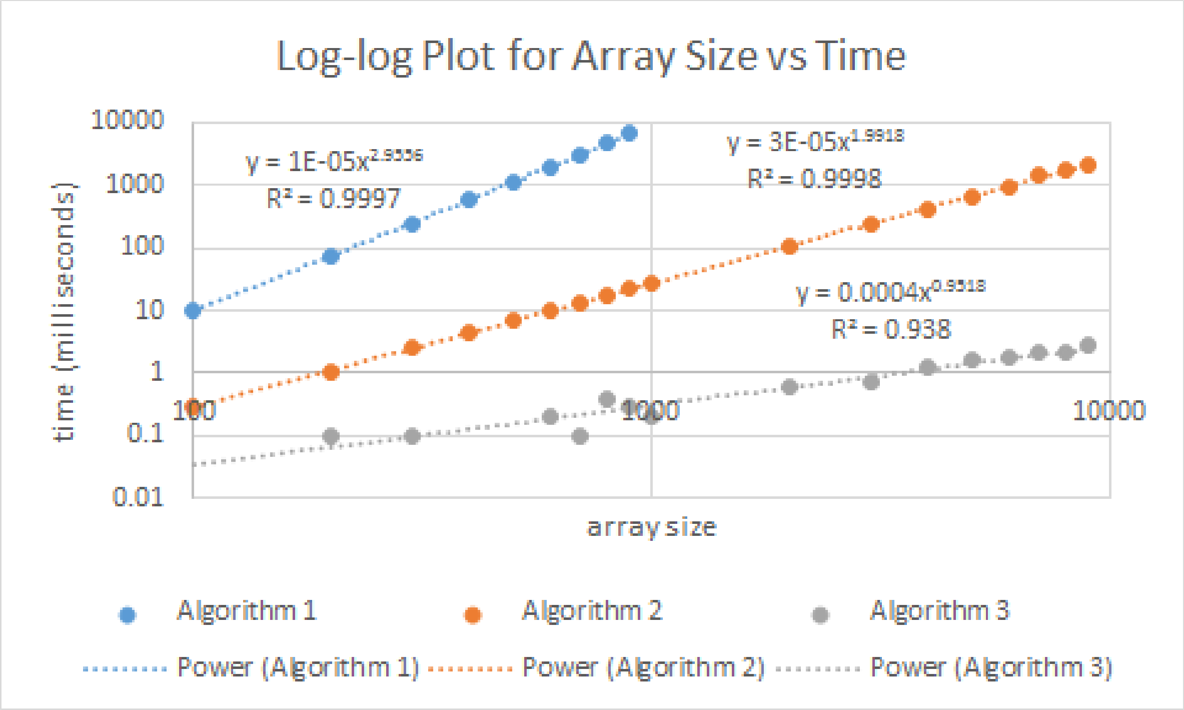
---

**Experimental analysis:** *Perform an experimental analysis and include plots*

The  C++ program was run on a Dell Inspiron laptop with an i7 processor.  Excel was used to plot the data.  We also ran the code on the OSU server but all of the times for algorithm 3 were 0 so we did not use that data for plotting.

*for timing code see attached file: cs325_project1_timingCode.cpp
Compile line on flip: g++ -std=c++0x cs325_project1_timingCode.cpp -o <ex. name>





---

**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.*

1) The size of the biggest instance that can be solved in one hour can be found using the trendline (best fit) equations from the Excel log-log graph.  One hour equals 3,600,000 milliseconds.

~~~Algorithm 1: Enumeration~~~

trendline equation: y = 1E-5x^2.9556
let y = 3600000 = 1E-5x^2.9556
3600000/1E-5 = x^2.9556
3.6E11 = x^2.9556
take log of both sides:  log(3.6E11) = log(x^2.9556)
11.5563 = 2.9556logx
logx = 3.9099
x = 10^3.9099 = 8126
The biggest instance for algorithm 1 is an array of ~8126 values.
~~~

### ~~~Algorithm 2: Better Enumeration~~~

y = 3E-5x^1.9918

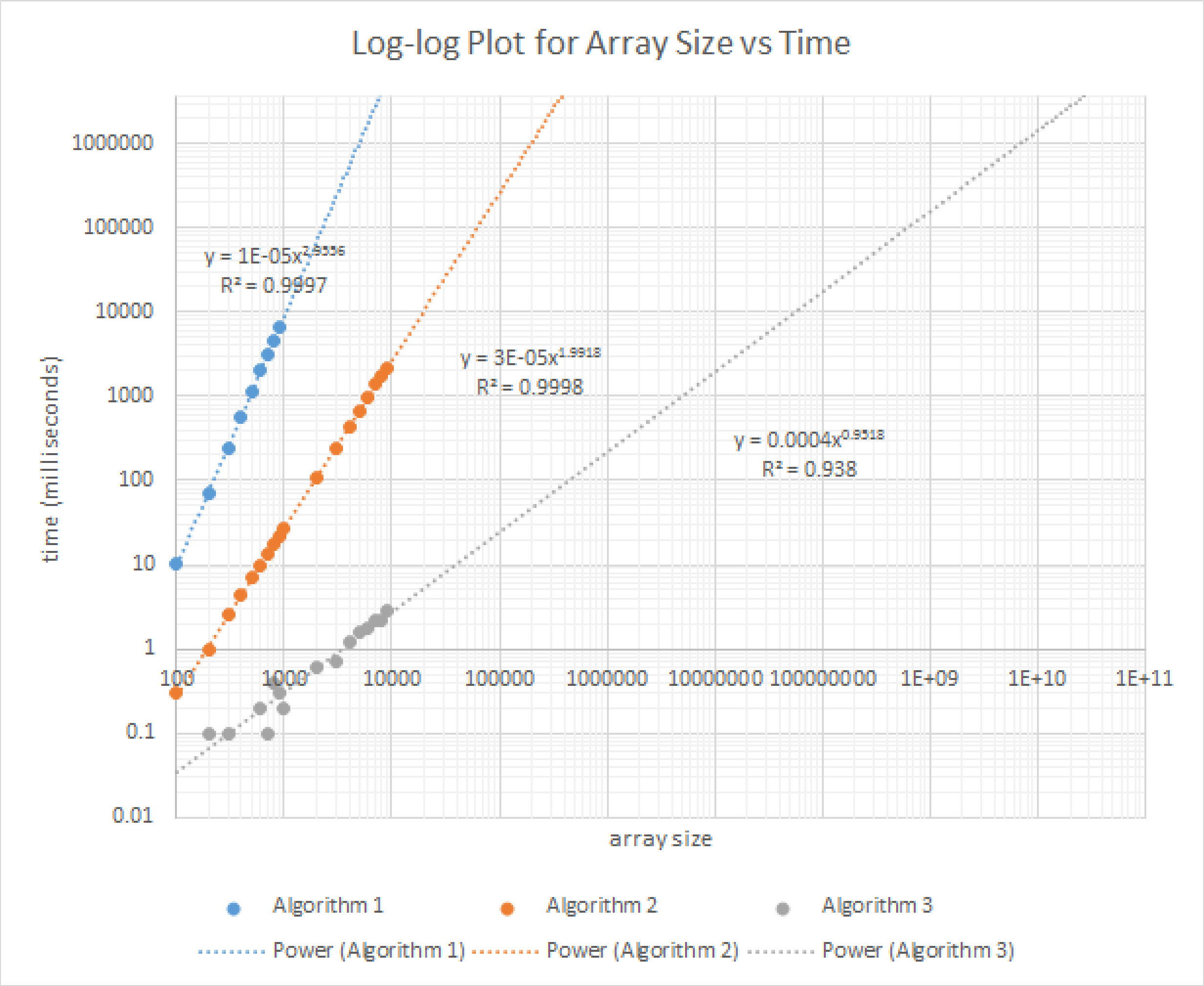
Using the same procedure as for algorithm 1, the biggest instance for algorithm 2 is an array of ~365,087 values.

### ~~~Algorithm 3: Divide and Conquer~~~

y = 0.0004x^0.9518

Using the same procedure as for algorithm 1, the biggest instance for algorithm 3 is an array of ~28,729,906,909 values.

Log-log graph showing the intersection of each line with a time of 3,600,000 milliseconds (one hour).



2) To calculate the slopes use the formula:

m = (log(y2) - log(y1))/(log(x2) - log(x1)) = log(y2/y1)/log(x2/x1)

~~~ Slope for algorithm 1~~~

using the two points (100, 10.4) and (900, 6572.4)
m = log(6572.4/10.4)/log(900/100) = 2.8007/.9542 = 2.9351

The experimental running time of algorithm 1 is therefore O(n^2.9351). The theoretical running time was O(n^3) so this is close. The discrepancy may be due to the fact that we only ran a small number of data points because running arrays larger than 900 would have taken a long time.

~~~ Slope for algorithm 2 ~~~

using the two points (1000, 26.9) and (9000, 2175.4)
m = log(2175.4/26.9)/log(9000/1000) = 1.9078/.9542 = 1.9994

The experimental running time of algorithm 2 is therefore O(n^1.9994). The theoretical running time was O(n^2) so these are essentially the same.

~~~ Slope for algorithm 3 ~~~

using the two points (900, .3) and (9000, 2.8)
m = log(2.8/.3)/log(9000/900) = .9700/1 =.9700

The experimental running time of algorithm 3 is therefore O(n^.97). The theoretical running time was O(nlogn). When we ran the smaller arrays (900 and below) the data was very noisy because the times were quite small (0 to .4 milliseconds). This could account for the discrepancy between theoretical and experimental.

Alternatively, using the trendline feature in Excel the slopes are 2.9556, 1.9918 and .9518 for algorithms 1, 2, and 3 respectively.
~~~