James Hippler (HipplerJ)
Oregon State University
CS 325-400 (Spring 2018)
Analysis of Algorithms

Homework 2

Due Date: Sunday, April 15, 2018

Problem 1: *(3 points)* Suppose you are choosing between the following three algorithms:

- *Algorithm A* solves a problem of size n by dividing them into five subproblems of half the size ($\frac{n}{2}$), recursively solving each subproblem, and then combining the solutions in linear time.
- *Algorithm B* solves problems of size n by recursively solving two subproblems of size $(n-1)$ and then combining the solutions in constant time.
- *Algorithm C* solves problems of size n by dividing them into nine subproblems of size $\frac{n}{3}$, recursively solving each subproblem, and then combining solutions in $\theta(n^2)$ time.

What are the running times of each of these algorithms and which would you select?

ANSWER:

Algorithm A

Solved through Master Theorem

$T(n) = 5T\left(\frac{n}{2}\right) + O(1)$

$a = 5, b = 2, c = 1, d = 0$

$\log_b a = \log_2 5$

$d < \log_2 5$

$T(n) = \theta(n^{\log_2 5})$

Algorithm B

Solved through Muster Theorem

$T(n) = 2T(n-1) + O(1)$

$a = 2, b = 1$

$f(n) = \theta(n^0) = \theta(1)$

$d = 0$

$= 2^{n+1} - 1$

$T(n) = \theta(2^n)$

Algorithm C

Solved through Master Theorem

$T(n) = 9T\left(\frac{n}{3}\right) + O(n^2)$

$a = 9, b = 3, c = 2$

$\log_b a = \log_3 9$

$\log_3 9 = 2$

$T(n) = \theta(n^2 \log n)$

I would select Algorithm C for my program, as Algorithm B has a very slow exponential running time, and Algorithm A has the number of elements raised to the 5th power which dominates the Quadratic in Algorithm C. Algorithm C has the fastest asymptotic running time of the three procedures.

Problem 2: *(6 points)* The ternary search algorithm is a modification of the binary search algorithm that splits the input not into two sets of almost-equal sizes, but into three sets of sizes approximately one-third.

a) Verbally describe and write pseudo-code for the ternary search algorithm.

ANSWER:
VERBAL DESCRIPTION:
The ternary search algorithm works similarly to the binary search algorithm except that it divides arrays into three equal segments. It is an example of a divide and conquer algorithm. The algorithm searches each part for the requested target value. Although it could be assumed that dividing the algorithm into three separate segments would operate at a higher efficiency than binary search, it's surprisingly not. This is because the ternary search algorithm requires a greater number of comparisons to find an element. If the array is 0, we can safely assume that the target element is not present since the array is empty. Next if the array is only 1 element in length, we just compare the target element in the first (and only) element in the array. Next, we begin recursively segmenting the array and performing our search comparisons until we reach the base case.

PSEUDOCODE:

```
ternary_search(array, search_value)
        if len(array) <= 0
                If the array is empty then the value is not present
                return array
        else if len(array) = 1
                if array[0] == value
                return found
        else
                Assign the first 1/3 of the array to front
                front = array[0 : len(array)/3]
                Assign the last 1/3 of the array to back
                back = array[len(front) : len(array)/3]
                Assign the middle 1/3 of the array to middle
                middle = array[len(front)+len(back) : len(array)/3]

                Recursively send the front of the array to be divided and searched
                ternary_search(front, search_value)

                Recursively send the middle of the array to be divided and searched
                ternary_search(middle, search_value)

                Recursively send the back of the array to be divided and searched
                ternary_search(back, search_value)
```

b) Give the recurrence for the ternary search algorithm.

ANSWER:

$\boldsymbol{T(n) = T\left(\frac{n}{3}\right) + 2}$

c) Solve the recurrence to determine the asymptotic running time of the algorithm. How does the running time of the ternary search algorithm compare to that of the binary search algorithm?

ANSWER:

Solved recurrence via Master theorem to determine the asymptotic running time.

$\boldsymbol{a = 1}$

$\boldsymbol{b = 4}$

$\boldsymbol{f(n) = 2}$

$\boldsymbol{\frac{\log_2 3}{\log_2 n} = \theta(\log_3 n)}$

$\boldsymbol{T(n) = \theta(\log_3 n)}$

or

$\boldsymbol{T(n) = \theta(\log n)}$

James Hippler (HipplerJ)
Oregon State University
CS 325-400 (Spring 2018)
Analysis of Algorithms

Problem 3: *(6 points)* Design and analyze a **divide and conquer** algorithm that determines the minimum and maximum value in an unsorted list (array).

a) Verbally describe and write pseudo-code for the min_and_max algorithm.

ANSWER:
VERBAL DESCRIPTION:
The min_and_max algorithm uses the divide, conquer, and merge technique to recursively divide an array of n elements into two segments. It is similar to merge sort in its design. The base case for the recursion is when the array contains only a single element. In this scenario we would assign the value of that element to both the minimum and maximum variables and exit. The next scenario is when the array is of size two, where we would compare the two values and assign them accordingly to the minimum and maximum variables (This is the scenario we are recursively working toward in our array divisions). Finally, if the array is larger than 2 elements, then we continue to separate the array in two and invoke our recursion on the array halves to determine a minimum and maximum value. Pseudocode for this operation is provided below.

PSEUDOCODE:

```
min_and_max(array)
        if len(array) = 1
                minimum = array[0]
                maximum = array [0]

        else if len(array) = 2
                If array[0] > array [1]
                Swap array[0] with array[1]

        else (if array is any size other than 1 or 2)
                Divide the array into two segments
                (left_min, left_max) = min_and_max (array1[0 : len(array)/2])
                (right_min, right_max) = min_and_max ([len(array)/2 : len(array)]
                if left_min <= right_min
                        minimum becomes left_min
                        minimum = left_min
                else
                        minimum becomes right_min
                        minimum = right_min
                if left_max >= right_max
                        maximum becomes left_max
                        maximum = left max
                else
                        maximum becomes right_max
                        maximum = right_max
                return the minimum and maximum values
                return (minimum, maximum)
```

b) Give the recurrence

ANSWER:

The recurrence for the min_and_max recursive algorithms is as follows…

$\boldsymbol{T(n) = 2T\left(\frac{n}{2}\right) + 2}$

c) Solve the recurrence to determine the asymptotic running time of the algorithm. How does the theoretical running time of the recursive min_and_max algorithm compare to that of an iterative algorithm for finding the minimum and maximum values of an array.

ANSWER:

Solved recurrence through Master theorem to determine the asymptotic running time. Steps are included below.

$a = 2$

$b = 2$

$f(n) = 2$

$n^{\log_b a} = n^{\log_2 2} = 1$

$f(n) = O(n^{\log_2 2-\epsilon})$

$Where\ \epsilon = 1$

$T(n) = \theta\left(n^{\log_2 2}\right) = \theta(n^1) =$

$T(n) = \theta(n)$

The recursive and iterative algorithm for finding the minimum and maximum values of an array would both operate with a linear time complexity of $\theta(n)$. The iterative approach could be implemented through a loop that checks each element of the array and continually assigns the highest and lowest values to maximum and minimum, respectively. Like the recursive design, this also would operate at a linear time complexity of $\theta(n)$.

Problem 4: *(5 points)* Consider the following pseudocode for a sorting algorithm.

```
StoogeSort(A[0 ... n - 1])
    if n = 2 and A[0] > A[1]
        swap A[0] and A[1]
    else if n > 2
        m = ceiling(2n/3)
        StoogeSort(A[0 ... m - 1])
        StoogeSort(A[n - m ... n - 1])
        Stoogesort(A[0 ... m - 1])
```

a) Verbally describe how the STOOGESORT algorithm sorts its input.

ANSWER:

Stooge Sort is a surprisingly complex, albeit relatively inefficient sorting algorithm that executes at speeds slower than those associated with both merge or insertion sort (significantly slower). The stooge sort algorithm initially works to recursively divide an array of n elements into two segments containing 2/3rds of the initial array elements (one array has the first 2/3rd elements and the second includes the last 2/3rds). It recursively sends those arrays back until it reaches an array size of two and evaluates the first and last elements of the array segments. The algorithm performs a swapping operation if the first array element [0] is greater than the last array element [1]. It's probably easier to break the steps into individual bullet points.

1. **If the array is one or less than simply return the array because a single element array is already sorted.**
2. **Determine if the array length is two elements in size.**
3. **If it is we compare the first element of the array and the last element of the array.**
4. **If the first element is greater, then perform a swap operation to switch the two values.**
5. **If the array is larger than 2**
6. **The algorithm will index the first 2/3rd elements and recursively call itself to continue breaking the array down and perform swaps as necessary.**
7. **The algorithm will then index the last 2/3rd elements by also recursively calling itself to continue sorting.**
8. **Finally, the algorithm will again send the first 2/3rds elements to break down, perform swaps, and confirm that the array is adequately sorted.**
9. **Once sorted, the algorithm returns the final sorted array.**

b) Would STOOGESORT still sort correctly if we replaced k = ceiling(2n/3) with k = floor(2n/3)? If yes prove if no give a counterexample. (Hint: what happens when n = 4?)

ANSWER:
Stooge Sort would NOT sort correctly if we performed the same 2n/3 division of the array elements but took the floor of the remainder instead of the ceiling. The stooge sort would return incorrect results depending on the array inputs. For instance, if we had an array of 4 elements and attempted to perform an 2n/3 division with floor rounding we would receive the following result

$$\frac{2(4)}{3} = \frac{8}{3} = 2.667 = 2$$

The first segment of the array would contain the first two values and the last would contain the last two values. The sorting algorithm would evaluate both subarrays and sort if necessary. This would result in the array halves being individually sorted but there would never be a comparison for the center numbers or between the two array halves in general. Rounding to the ceiling ensures that the final 2/3rds array comparison will evaluate the sorting from the initial front and back arrays segments and finalizes the sorting. When testing this in my code, I found that I would often also encounter issues where my recursion would not reach the base case and terminate.

Initial Array
```
[9, 10, -9, -10]
```
Front of Array
```
[9, 10, -9, -10]
```
No Swaps made, Array returned unchanged to have the back half evaluated.
```
[9, 10, -9, -10]
```
Swap made. Array returned with the final below value.
```
[9, 10, -10, -9]
```

Thus, proven that Stooge sort does not correctly when rounding division to its floor

c) State a recurrence for the number of comparisons executed by STOOGESORT.

ANSWER:
Stooge Sort recurrence is as follows
3 recursive calls with array elements divided into segments of 2/3rds the original size

$$T(n) = 3T\left(\frac{2n}{3}\right) + \theta(1)$$

d) Solve the recurrence to determine the asymptotic running time.

ANSWER:
Solved recurrence via Master theorem to determine the asymptotic running time.

$a = 3$

$b = \frac{3}{2}$

$\mathrm{c} = \log_{\frac{3}{2}} 3$

$\log_{\frac{3}{2}} 3 = 2.70951129 \ldots$

$= \sim 2.71$

$\theta\left(n^{\log_{\frac{3}{2}} 3}\right)$

$T(n) = \theta(n^{2.71})$

Problem 5: *(10 points)*

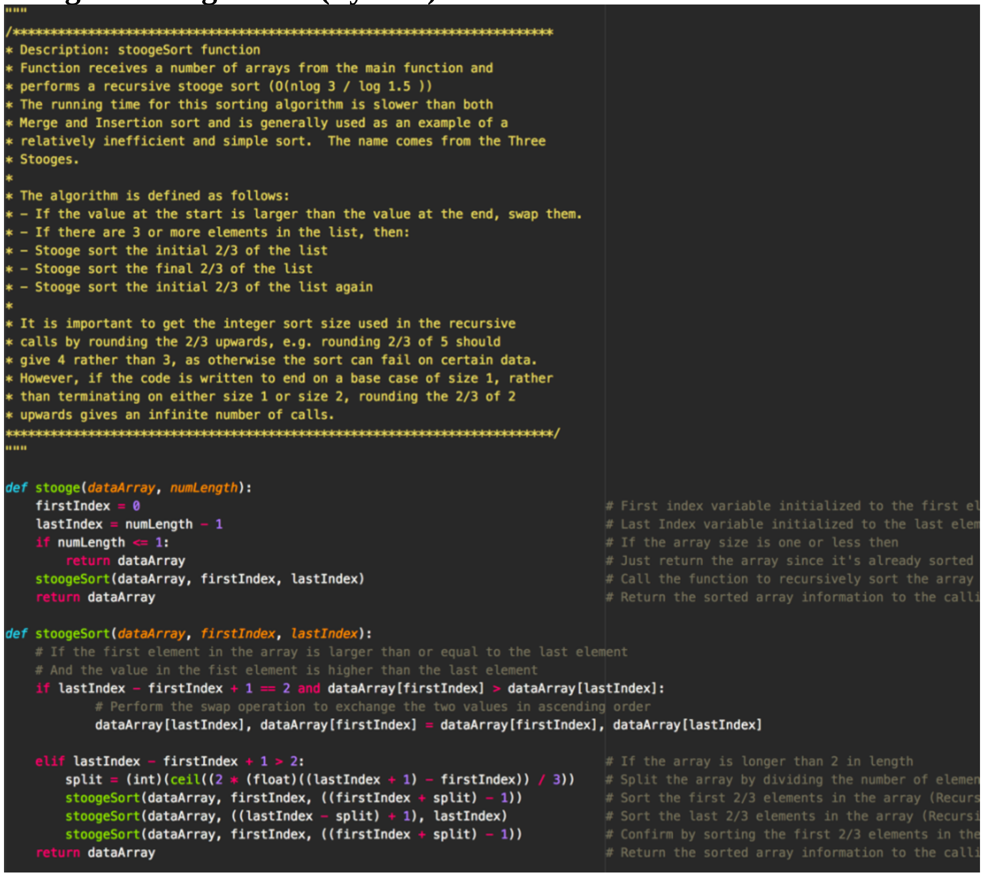
a) Implement STOOGESORT from Problem 4 to sort an array/vector of integers. Implement the algorithm in the same language you used for the sorting algorithms in HW 1. Your program should be able to read inputs from a file called "data.txt" where the first value of each line is the number of integers that need to be sorted, followed by the integers (like in HW 1). The output will be written to a file called "stooge.out".

Submit a copy of all your code files and a README file that explains how to compile and run your code in a ZIP file to TEACH. We will only test execution with an input file named data.txt.

ANSWER:

My implementation for Stooge Sort was submitted to TEACH in a zipped folder under the filename stoogesort.py. the program takes arrays from a file called data.txt, sorts them, and writes the sorted results to a file called stooge.out. As with Homework 1, I again used python as my development language. The algorithm was tested on my desktop and the university FLIP servers and confirmed to work correctly. Below is a screen shot from my code of the function that performs my implementation of stooge sort. Please see my zip file TEACH submission for the full version of the program.

Stooge Sort Algorithm (Python):



```
"""
/*************************************************************************
* Description: stoogeSort function
* Function receives a number of arrays from the main function and
* performs a recursive stooge sort (O(nlog 3 / log 1.5 ))
* The running time for this sorting algorithm is slower than both
* Merge and Insertion sort and is generally used as an example of a
* relatively inefficient and simple sort.  The name comes from the Three
* Stooges.
*
* The algorithm is defined as follows:
* - If the value at the start is larger than the value at the end, swap them.
* - If there are 3 or more elements in the list, then:
* - Stooge sort the initial 2/3 of the list
* - Stooge sort the final 2/3 of the list
* - Stooge sort the initial 2/3 of the list again
*
* It is important to get the integer sort size used in the recursive
* calls by rounding the 2/3 upwards, e.g. rounding 2/3 of 5 should
* give 4 rather than 3, as otherwise the sort can fail on certain data.
* However, if the code is written to end on a base case of size 1, rather
* than terminating on either size 1 or size 2, rounding the 2/3 of 2
* upwards gives an infinite number of calls.
*************************************************************************/
"""

def stooge(dataArray, numLength):
    firstIndex = 0                                                              # First index variable initialized to the first el
    lastIndex = numLength - 1                                                   # Last Index variable initialized to the last elem
    if numLength <= 1:                                                          # If the array size is one or less then
        return dataArray                                                        # Just return the array since it's already sorted
    stoogeSort(dataArray, firstIndex, lastIndex)                                # Call the function to recursively sort the array
    return dataArray                                                            # Return the sorted array information to the calli

def stoogeSort(dataArray, firstIndex, lastIndex):
    # If the first element in the array is larger than or equal to the last element
    # And the value in the fist element is higher than the last element
    if lastIndex - firstIndex + 1 == 2 and dataArray[firstIndex] > dataArray[lastIndex]:
            # Perform the swap operation to exchange the two values in ascending order
            dataArray[lastIndex], dataArray[firstIndex] = dataArray[firstIndex], dataArray[lastIndex]

    elif lastIndex - firstIndex + 1 > 2:                                        # If the array is longer than 2 in length
        split = (int)(ceil((2 * (float)((lastIndex + 1) - firstIndex)) / 3))    # Split the array by dividing the number of elemen
        stoogeSort(dataArray, firstIndex, ((firstIndex + split) - 1))           # Sort the first 2/3 elements in the array (Recurs
        stoogeSort(dataArray, ((lastIndex - split) + 1), lastIndex)             # Sort the last 2/3 elements in the array (Recursi
        stoogeSort(dataArray, firstIndex, ((firstIndex + split) - 1))           # Confirm by sorting the first 2/3 elements in the
    return dataArray                                                            # Return the sorted array information to the calli
```

b) Now that you have proven that your code runs correctly using the data.txt input file, you can modify the code to collect running time data. Instead of reading arrays from a file to sort, you will now generate arrays of size n containing random integer values and then time how long it takes to sort the arrays. We will not be executing the code that generates the running time data so it does not have to be submitted to TEACH or even execute on flip. Include a “text” copy of the modified code in the written HW submitted in Canvas. You will need at least seven values of t (time) greater than 0. If there is variability in the times between runs of the algorithm you may want to take the average time of several runs for each value of n.

ANSWER:
Initially when running the stooge sort running time analysis, I was attempting to use the same array sizes as I did with the insertion and merge sort algorithms which ranged from 1,000 all the way to 100,000 elements. I decided to significantly adjust those values when I noticed that it took almost an hour to process only 5,000 elements. The current version of the program sorts 10 different arrays from sizes 500 to 5,000 elements in increasing increments of 500. I reran the same test on insertion sort and merge sort to get a baseline comparison. I took the results obtained when testing on the FLIP Server and noticed that the performance was slower than those obtained on my computer (I think they must throttle user resources). Below is a copy of my code for testing stooge sort run time comparisons.

Stooge Sort Run Time Code (Python):

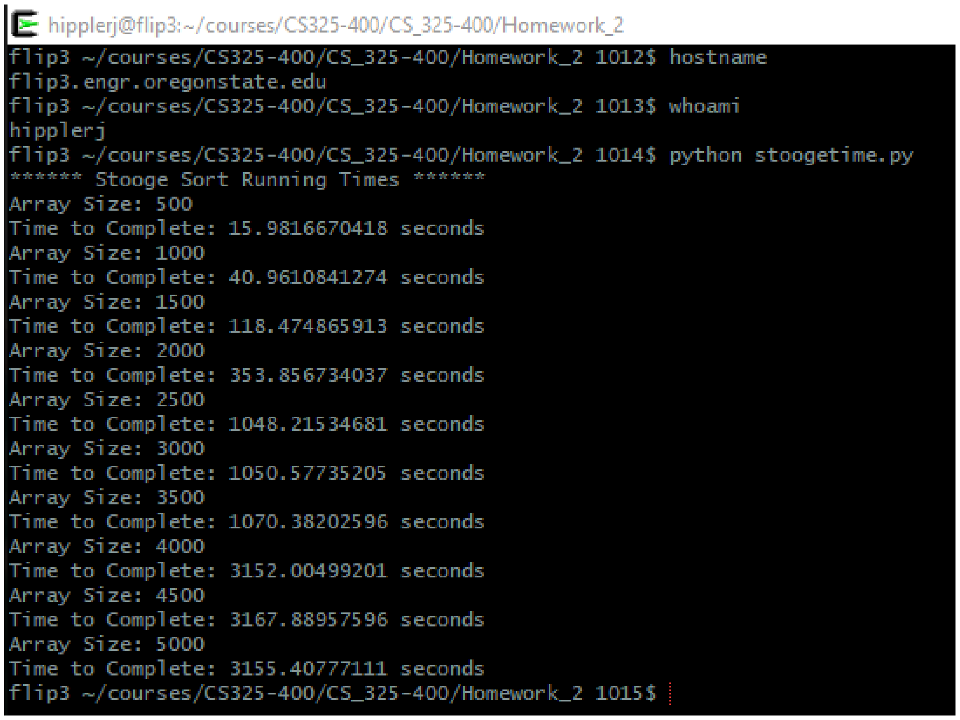
```
#!/usr/bin/python

from math import ceil                                                                    # Math
library needed to perform ceil division rounding later in program
import time                                                                              #
Import the time library to measure time
import random                                                                            #
Import the random number generator library

"""
/*************************************************************************
* Description: stoogeSort function
* Function receives a number of arrays from the main function and
* performs a recursive stooge sort (O(nlog 3 / log 1.5 ))
* The running time for this sorting algorithm is slower than both
* Merge and Insertion sort and is generally used as an example of a
* relatively inefficient and simple sort.  The name comes from the Three
* Stooges.
*
* The algorithm is defined as follows:
* - If the value at the start is larger than the value at the end, swap them.
* - If there are 3 or more elements in the list, then:
* - Stooge sort the initial 2/3 of the list
* - Stooge sort the final 2/3 of the list
* - Stooge sort the initial 2/3 of the list again
*
* It is important to get the integer sort size used in the recursive
* calls by rounding the 2/3 upwards, e.g. rounding 2/3 of 5 should
* give 4 rather than 3, as otherwise the sort can fail on certain data.
* However, if the code is written to end on a base case of size 1, rather
* than terminating on either size 1 or size 2, rounding the 2/3 of 2
* upwards gives an infinite number of calls.
*************************************************************************/
```

```
"""

def stooge(dataArray, numLength):
    firstIndex = 0                                                              #
First index variable initialized to the first element in the array
    lastIndex = numLength - 1                                                   # Last
Index variable initialized to the last element in the array
    if numLength <= 1:                                                          # If
the array size is one or less then
        return dataArray                                                        # Just
return the array since it's already sorted
    stoogeSort(dataArray, firstIndex, lastIndex)                                # Call
the function to recursively sort the array segments

def stoogeSort(dataArray, firstIndex, lastIndex):
    # If the first element in the array is larger than or equal to the last element
    # And the value in the fist element is higher than the last element
    if lastIndex - firstIndex + 1 == 2 and dataArray[firstIndex] >
dataArray[lastIndex]:
            # Perform the swap operation to exchange the two values in ascending order
            dataArray[lastIndex], dataArray[firstIndex] = dataArray[firstIndex],
dataArray[lastIndex]

    elif lastIndex - firstIndex + 1 > 2:                                        # If
the array is longer than 2 in length
        split = (int)(ceil((2 * (float)((lastIndex + 1) - firstIndex)) / 3))    #
Split the array by dividing the number of elements by three (take ceiling)
        stoogeSort(dataArray, firstIndex, ((firstIndex + split) - 1))           # Sort
the first 2/3 elements in the array (Recursive)

        stoogeSort(dataArray, ((lastIndex - split) + 1), lastIndex)             # Sort
the last 2/3 elements in the array (Recursive)
        stoogeSort(dataArray, firstIndex, ((firstIndex + split) - 1))           #
Confirm by sorting the first 2/3 elements in the array a second time (Recursive)
"""
/************************************************************************
* Description: makeArray function
* Function creates an array of random integers based on an increasing
* array size.  Sends the random array to the stooge sort function to
* be sorted.
************************************************************************/
"""

def makeArray(n):
    randomArray = []                                                            #
Establish a new array to store the random integers
    for x in range (n):                                                         # Run
through a loop for the desired number of random elements
        randomArray.append(random.randint (0, 10000))                           #
Append the random elements to the unsorted Array
    stooge(randomArray, n)                                                      # Call
the Stooge sort function to sort the new random array

"""
/************************************************************************
* Description: startClock function
* Function starts a clock at the beginning of the program so that we
* can automatically determine run time.  Hands the time details as a
* variable to main.
************************************************************************/
"""

def startClock():
    startTime = time.time()                                                     #
Collect the start time at the initial execution of the program
    return startTime                                                            #
Return the startTime to main to be used later
```

```
"""
/*******************************************************************************
* Description: finished function
* Function does absolutely nothing but print a message to the console
* letting the user know that stooge sort is complete and the name
* of the output file where the sorted arrays were written. Receives
* nothing and returns nothing
*******************************************************************************/
"""

def finished(startTime):
    print("Array Size: {}".format(n))                                       #
Output the Array size to the screen
    print("Time to Complete: {} seconds".format(time.time() - startTime))       #
Output the time to complete the program.

"""
/*******************************************************************************
* Description: main function
* Program starts here and calls functions as necessary
* Calls the mergesort function that completes most of the program's
* functionality.  Had to dramatically reduce the size of the arrays
* as it was taking more time to sort than the current age of our
* galaxy
*******************************************************************************/
"""

if __name__ == "__main__":
    print("****** Stooge Sort Running Times ******")                              #
Display message to console for what is about to happen
    n = 500                                                                 # Set
number of variables that need to be generated
    for count in range(0, 10):                                              # Run
the sort several times with different array sizes
        startTime = startClock()                                            #
Record the start time when the program begins execution
        makeArray(n)                                                        # Send
the array to the stooge sort function to be sorted
        finished(startTime)                                                 #
Display the sort time and array size to the console
        n += 500                                                            #
Increment the array size by 1000
```

c) Plot the running time data you collected on an individual graph with n on the x-axis and time on the y-axis. You may use Excel, Matlab, R or any other software. Also plot the data from Stooge algorithm together on a combined graph with your results for merge and insertion sort from HW1.

ANSWER:
Below are the run time results for my implementation of stooge sort. I used the University FLIP servers to run the program. As you can see, the sorting times are incredibly slow. Also included below are the individual and combined graphs detailing runtime for stooge sort.

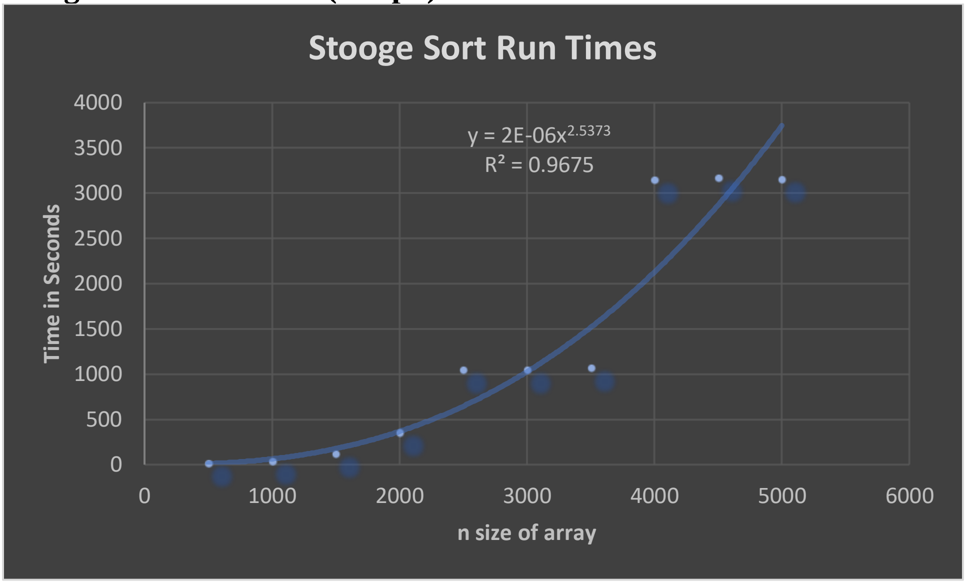

hipplerj@flip3:~/courses/CS325-400/CS_325-400/Homework_2

```
flip3 ~/courses/CS325-400/CS_325-400/Homework_2 1012$ hostname
flip3.engr.oregonstate.edu
flip3 ~/courses/CS325-400/CS_325-400/Homework_2 1013$ whoami
hipplerj
flip3 ~/courses/CS325-400/CS_325-400/Homework_2 1014$ python stoogetime.py
****** Stooge Sort Running Times ******
Array Size: 500
Time to Complete: 15.9816670418 seconds
Array Size: 1000
Time to Complete: 40.9610841274 seconds
Array Size: 1500
Time to Complete: 118.474865913 seconds
Array Size: 2000
Time to Complete: 353.856734037 seconds
Array Size: 2500
Time to Complete: 1048.21534681 seconds
Array Size: 3000
Time to Complete: 1050.57735205 seconds
Array Size: 3500
Time to Complete: 1070.38202596 seconds
Array Size: 4000
Time to Complete: 3152.00499201 seconds
Array Size: 4500
Time to Complete: 3167.88957596 seconds
Array Size: 5000
Time to Complete: 3155.40777111 seconds
flip3 ~/courses/CS325-400/CS_325-400/Homework_2 1015$
```

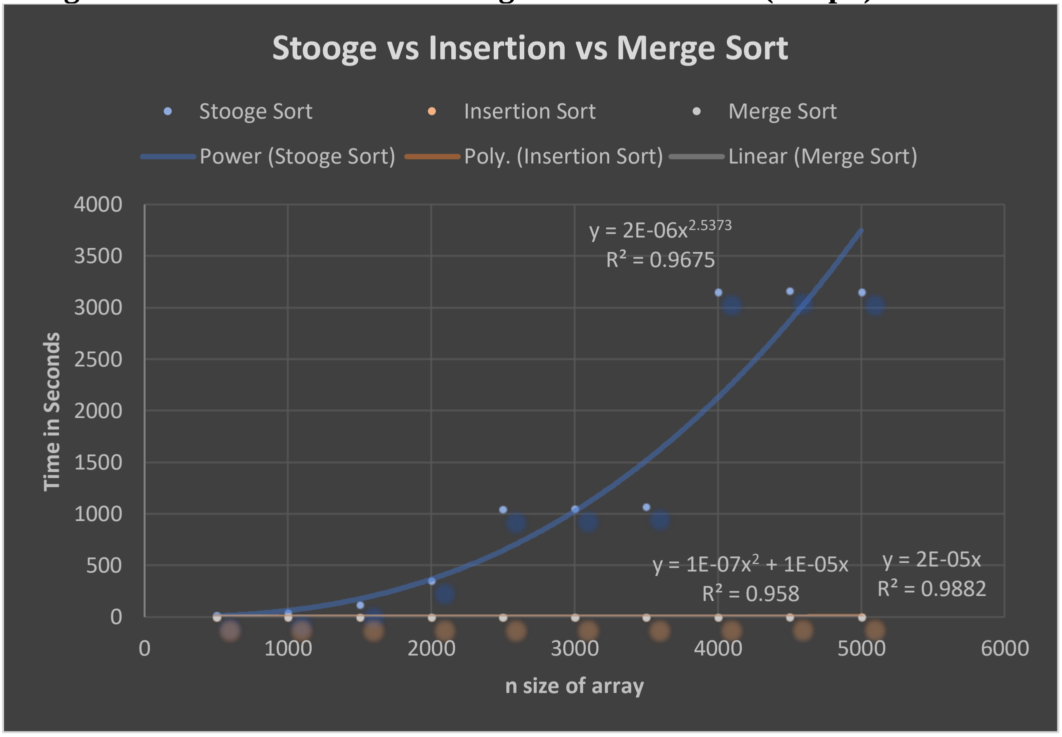
```
flip3 ~/courses/CS325-400/CS_325-400/Homework_2 1014$ python stoogetime.py
****** Stooge Sort Running Times ******
Array Size: 500
Time to Complete: 15.9816670418 seconds
Array Size: 1000
Time to Complete: 40.9610841274 seconds
Array Size: 1500
Time to Complete: 118.474865913 seconds
Array Size: 2000
Time to Complete: 353.856734037 seconds
Array Size: 2500
Time to Complete: 1048.21534681 seconds
Array Size: 3000
Time to Complete: 1050.57735205 seconds
Array Size: 3500
Time to Complete: 1070.38202596 seconds
Array Size: 4000
Time to Complete: 3152.00499201 seconds
Array Size: 4500
Time to Complete: 3167.88957596 seconds
```

```
Array Size: 5000
Time to Complete: 3155.40777111 seconds
```

Stooge Sort Run Times (Graph)



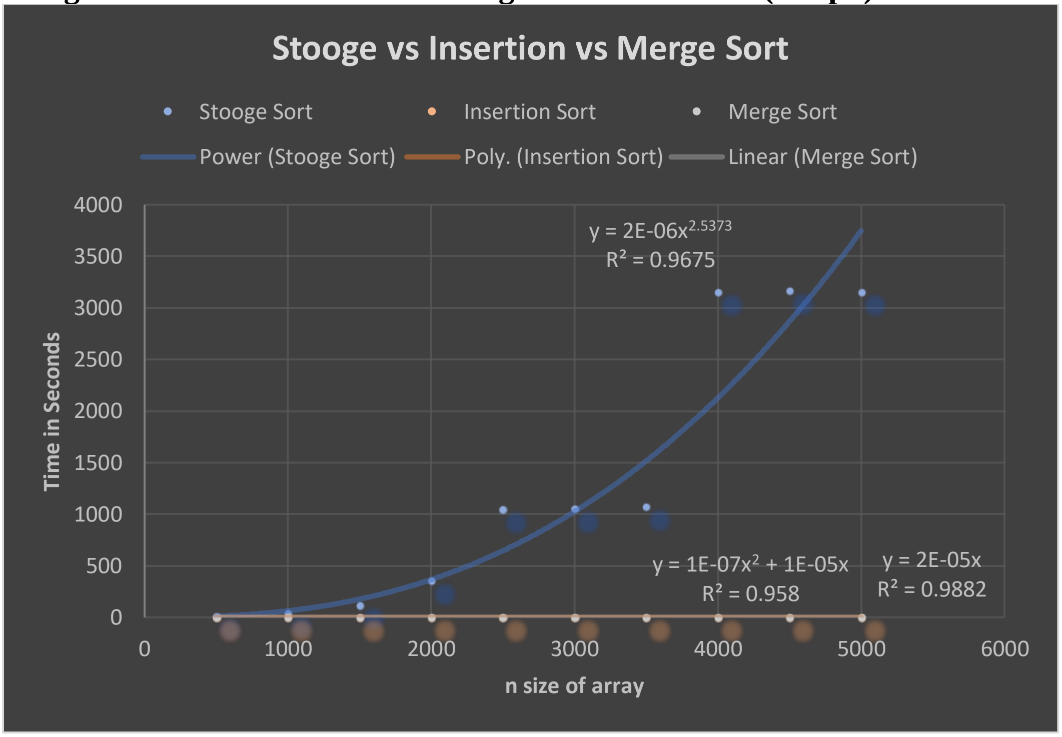
Stooge Sort vs Insertion Sort vs Merge Sort Run Times (Graph)



d) What type of curve best fits the StoogeSort data set? Give the equation of the curve that best "fits" the data and draw that curve on the graphs of created in part c). How does your experimental running time compare to the theoretical running time of the algorithm?

ANSWER:
I found that the power curve best fit the run time data from my Stooge Sort algorithm data set. It gave a regression equation of $y = 2E - 06x^{2.5373}$ ($R^2 = 0.9675$). The power curve has been included in the graph below (as well as the graph in part c since they're the same). The experimental running time of $\theta(n^{2.5373})$ compares pretty closely with the theoretical running time of $\theta(n^{2.71})$ that's generally associated with the algorithm. I'm guessing that the randomly seeded array values could be the cause of the slight delineation. As we can see from the comparative graph, the running times generated from Stooge sort absolutely dwarf the results from Merge and Insertion sort. To process the largest array size of 5,000 randomly seeded elements it took Merge Sort less than a second (0.0998940467834 seconds), Insertion Sort took a little over two seconds (2.13609790802 seconds), and Stooge sort took over 52 minutes (3155.40777111 seconds). Because of this, we can barely even see the graph lines for merge sort and insertion sort compared to the asymptotic growth displayed by Stooge Sort. I cannot even imagine a scenario where Stooge sort would be the best suited sorting algorithm, unless it's needed to demonstrate inefficiency.

Stooge Sort vs Insertion Sort vs Merge Sort Run Times (Graph)


Stooge vs Insertion vs Merge Sort
● Stooge Sort
● Insertion Sort
● Merge Sort
Power (Stooge Sort)
Poly. (Insertion Sort)
Linear (Merge Sort)
y = 2E-06x^2.5373
R² = 0.9675
y = 1E-07x² + 1E-05x
R² = 0.958
y = 2E-05x
R² = 0.9882
Time in Seconds
n size of array