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

Homework 1

Due Date: Sunday, April 08, 2018

1. **(3 pts) For each of the following pairs of functions, either f(n) is O(g(n)), f(n) is Ω(g(n)), or f(n) is Θ (g(n)). Determine which relationship is correct and explain.**
   a. $f(n) = n^{0.25}$ $g(n) = \sqrt{n}$
   b. $f(n) = \log n^2$ $g(n) = \ln n$
   c. $f(n) = n \log n$ $g(n) = n\sqrt{n}$
   d. $f(n) = 2^n$ $g(n) = 3^n$
   e. $f(n) = 2^n$ $g(n) = 2^{n+2}$
   f. $f(n) = 4^n$ $g(n) = n!$

Answers:

Problem	$f(n)$	$g(n)$	Relationship	Explanation
a.	$f(n) = n^{0.25}$	$g(n) = \sqrt{n}$	$f(n)$ is $O(g(n))$ BIG OH	$\lim_{n\to\infty} \frac{n^{0.25}}{\sqrt{n}} = 0$
b.	$f(n) = \log n^2$	$g(n) = \ln n$	$f(n)$ is $\theta(g(n))$ THETA	$\lim_{n\to\infty} \frac{log\ n^2}{\ln n} = \frac{2}{\log(10)} = 0.8686$
c.	$f(n) = n \log n$	$g(n) = n\sqrt{n}$	$f(n)$ is $O(g(n))$ BIG OH	$\lim_{n\to\infty} \frac{n \log n}{n\sqrt{n}} = 0$
d.	$f(n) = 2^n$	$g(n) = 3^n$	$f(n)$ is $O(g(n))$ BIG OH	$\lim_{n\to\infty} \frac{2^n}{3^n} = 0$
e.	$f(n) = 2^n$	$g(n) = 2^{n+2}$	$f(n)$ is $\theta(g(n))$ THETA	$\lim_{n\to\infty} \frac{2^n}{2^{n+2}} = \frac{1}{4} = 0.25$
f.	$f(n) = 4^n$	$g(n) = n!$	$f(n)$ is $O(g(n))$ BIG OH	$\lim_{n\to\infty} \frac{4^n}{n!} = 0$

2. **(3 pts) Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence**

$$T(n) = \begin{cases} 2, & if\ n = 2 \\ 2T\left(\frac{n}{2}\right) + n, & if\ n = 2^k, for\ k > 1 \end{cases}$$

is $T(n) = n \lg n$

Answer:

Base Case:

If $n = 2$,

Then $T(2)$ and $2 \log 2$ both must equal 2 as well

Therefore, $T(2) = 2 \log 2$

Inductive Hypothesis:

If $n = 2^k$ for integers where $k > 0$ then we can assume that $T(n) = n \log n$ is true.

Apply the Axiom of Induction:

If $n = 2^{k+1}$, then $T(2^{k+1})$

$$2T\left(\frac{2^{k+1}}{2}\right) + 2^{k+1}$$
$$= 2T(2^k) + 2^{k+1}$$

Now substitute in $n \log n$ for $T(n)$,

$$= 2(2^k \log 2^k) + 2^{k+1}$$
$$= 2^{k+1} \log 2^{k+1}$$

Thus, proven through mathematical induction as $n = 2^{k+1}$ for both for $T(n)$ and $n \log n$

3. **(4 pts) Let f_1 and f_2 be asymptotically positive non-decreasing functions. Prove or disprove each of the following conjectures. To disprove give counter example.**

   a. **If $f_1(n) = O(g(n))$ and $f_2(n) = O(g(n))$ then $f_1(n) = \theta(f_2(n))$**
   b. **If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$**

Answers:

A. Disproved If $f_1(n) = O(g(n))$ and $f_2(n) = O(g(n))$ then $f_1(n) = \theta(f_2(n))$ through counter example.
$f_1(n) = n$
$f_2(n) = n^2$
$g(n) = n^3$

Therefore,
$f_1(n) \neq \theta(f_2(n))$

Thus, disproved through counter example

B. Prove If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$
If $f_1(n) = O(g_1(n))$
Then constants c_1 and n_1 also exist such that…
Step 1: $0 < f_1(n) \leq c_1 g_1(n)$ for all $n \geq n_1$ and since $f_2(n) = O(g_2(n))$
Therefore constants c_2 and n_2 also exist such that…
Step 2: $0 < f_2(n) \leq c_2 g_2(n)$ for all $n \geq n_2$
Let $g = \max\{g_1(n), g_2(n)\}$
Step 3: $f_1(n) \leq g$ for $n \geq (n_1 + n_2)$
$f_2(n) \leq g$
Combining the two items from step 3
$0 < f_1(n) + f_2(n) \leq 2g(n)$ for $n \geq (n_1 + n_2)$
Therefore,
$f_1(n) + f_2(n) \leq cg(n)$ and
$f_1(n) + f_2(n) = O(\mathrm{g(n)})$ or
$f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$

4. **(10 pts) Merge Sort and Insertion Sort Programs**
   Implement merge sort and insertion sort to sort an array/vector of integers. You may implement the algorithms in the language of your choice, name one program "mergesort" and the other "insertsort". Your programs 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.

   Example values for data.txt:

   4 19 2 5 11
   8 1 2 3 4 5 6 1 2

   The output will be written to files called "merge.out" and "insert.out".
   For the above examples the output would be:

   2 5 11 19
   1 1 2 2 3 4 5 6

   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 test execution with an input file named data.txt.

Answer:

My implementations for Merge Sort and Insertion Sort were submitted to TEACH in a zipped folder under the filenames mergesort.py and insertsort.py, respectively. They take arrays from a file called data.txt, sort them, and write the sorted results to files called merge.out and insert.out. I used python as my development language, it executes slower than C/C++ but abstracts away a lot of the lines of code necessary for implementing these algorithms. The algorithms were tested on my desktop and the university FLIP servers and confirmed to work correctly. Below are screen shots from my code of the functions that perform each version of sort. Please see my zip file TEACH submission for full versions of the code.

Merge Sort Algorithm (Python)

```
/*****************************************************************************
* Description: mergeSort function
* Functions receives a number array from main and performs a merge sort
* to place the numbers in ascending order.  The function takes and array
* and recursively devides until until it reaches array sizes of one.
* It passes data to a merge function that combines the elements in
* ascending order into the original array
*****************************************************************************/
"""

def mergeSort (dataArray, arraySize):
    if arraySize < 1:
        print("Array Size {} is invalid".format(arraySize))
    elif arraySize == 1:
        return dataArray
    else:
        first = dataArray[:(arraySize // 2)]
        last = dataArray[(arraySize // 2):]
        firstLength = len(first)
        lastLength = len(last)
        firstArray = mergeSort(first, firstLength)
        secondArray =mergeSort(last, lastLength)
    return merge (dataArray, firstArray, secondArray)


"""
/*****************************************************************************
* Description: merge function
* Called by the mergeSort function to evaluate and merge the split array
* elements back into a single sorted array.  Receives the original array
* the split right and left arrays and their respective lengths.  Returns
* a sorted integer array to mergerSort
*****************************************************************************/
"""

def merge (dataArray, first, second):
    firstcount = secondcount = originalcount = 0
    while firstcount < len(first) and secondcount < len(second):
        if first[firstcount] < second[secondcount]:
            dataArray[originalcount]=first[firstcount]
            firstcount += 1
        else:
            dataArray[originalcount]=second[secondcount]
            secondcount += 1
        originalcount += 1
    while firstcount < len(first):
        dataArray[originalcount]=first[firstcount]
        firstcount += 1
        originalcount += 1
    while secondcount < len(second):
        dataArray[originalcount]=second[secondcount]
        secondcount += 1
        originalcount += 1
    return dataArray

"""
```

Insertion Sort Algorithm (Python)

```
/*****************************************************************
* Description: insertionSort function
* Functions receives a number array from main and performs an insertion
* sort to place the numbers in ascending order.  Starts with the second
* value in the array and compares to values on the left.  With each
* iteration of the loop the counter is moved forward one place in the
* array
*****************************************************************/
"""

def insertionSort (dataArray, arraySize):
    # Start loop at the second element (1) rather than the first (0)
    for count in range(1, arraySize):
        key = dataArray[count]
        compare = count - 1
        while compare >= 0 and key < dataArray[compare]:
            dataArray[compare + 1] = dataArray[compare]
            compare -= 1
        dataArray[compare + 1] = key
    return dataArray
```

5. **(10 pts) Merge Sort vs Insertion Sort Running time analysis**

   The goal of this problem is to compare the experimental running times of the two sorting algorithms.

   a. **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 from 0 to 10,000 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.**

Answer:

Insertion Sort Timing Code (Screen Shot)

```
import time
import random

"""
/*********************************************************************
* Description: insertionSort function
* Functions receives a number array from main and performs an insertion
* sort to place the numbers in ascending order.  Starts with the second
* value in the array and compares to values on the left.  With each
* iteration of the loop the counter is moved forward one place in the
* array
*********************************************************************/
"""

def insertionSort (dataArray, arraySize):
    # Start loop at the second element (1) rather than the first (0)
    for count in range(1, arraySize):
        key = dataArray[count]
        compare = count - 1
        while compare >= 0 and key < dataArray[compare]:
            dataArray[compare + 1] = dataArray[compare]
            compare -= 1
        dataArray[compare + 1] = key
    return dataArray


"""
/*********************************************************************
* Description: main function
* Program starts here and calls functions as necessary
* Calls the mergesort function that completes most of the program's
* functionality.
*********************************************************************/
"""

if __name__ == "__main__":

    n = 1000
    for count in range(0, 13):
        starttime = time.time()
        if count == 1:
            n = 2000
        if count == 2:
            n = 5000
        if count == 3:
            n = 10000
        unsortedArray = []
        for x in range (n):
            unsortedArray.append(random.randint (0, 10000))
        insertionSort(unsortedArray, n)
        print("Array Size: {}".format(n))
        print("Time to Complete: {} seconds".format(time.time() - starttime))
        n += 10000
```

Text Copy of Insertion Sort Time Code

```
import time                                                                    #
Import the time library to measure time
import random                                                                  #
Import the random number generator library

"""
/*****************************************************************************
* Description: insertionSort function
* Functions receives a number array from main and performs an insertion
* sort to place the numbers in ascending order.  Starts with the second
* value in the array and compares to values on the left.  With each
* iteration of the loop the counter is moved forward one place in the
* array
*****************************************************************************/
"""

def insertionSort (dataArray, arraySize):
    # Start loop at the second element (1) rather than the first (0)
    for count in range(1, arraySize):                                          #
Increment through the array one element at a time using the array's length
        key = dataArray[count]                                                 # Set
the current index to the loop counter (counter starts at 1)
        compare = count - 1                                                    #
Adjusted the comparative number one element left in the array
        while compare >= 0 and key < dataArray[compare]:                       #
While compare number is in range and key is less than the compare number
            dataArray[compare + 1] = dataArray[compare]                        # Swap
the compare number forward (right) in the array
            compare -= 1                                                       # Move
comparitive number to the next element in the array (going left toward 0)
        dataArray[compare + 1] = key                                           # No
swap occurs and the key remains in place
    return dataArray                                                           #
Return the sorted Array to main function


"""
/*****************************************************************************
* Description: main function
* Program starts here and calls functions as necessary
* Calls the mergesort function that completes most of the program's
* functionality.
*****************************************************************************/
"""

if __name__ == "__main__":

    n = 1000                                                                   # Set
number of variables that need to be generated
    for count in range(0, 13):
        starttime = time.time()                                                #
Collect the start time at the initial execution of the program
        if count == 1:
            n = 2000
        if count == 2:
            n = 5000
        if count == 3:
            n = 10000
        unsortedArray = []                                                     #
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
            unsortedArray.append(random.randint (0, 10000))                    #
Append the random elements to the unsorted Array
        insertionSort(unsortedArray, n)                                        # Send
the array to the sort function to be sorted
```

```
        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.
        n += 10000
```

Merge Sort Timing Code (Screen Shot)

```
def mergeSort (dataArray, arraySize):
    if arraySize < 1:
        print("Array Size {} is invalid".format(arraySize))
    elif arraySize == 1:
        return dataArray
    else:
        first = dataArray[:(arraySize // 2)]
        last = dataArray[(arraySize // 2):]
        firstLength = len(first)
        lastLength = len(last)
        firstArray = mergeSort(first, firstLength)
        secondArray =mergeSort(last, lastLength)
    return merge (dataArray, firstArray, secondArray)


"""
/*****************************************************************
* Description: merge function
* Called by the mergeSort function to evaluate and merge the split array
* elements back into a single sorted array.  Receives the original array
* the split right and left arrays and their respective lengths.  Returns
* a sorted integer array to mergerSort
*****************************************************************/
"""

def merge (dataArray, first, second):
    firstcount = secondcount = originalcount = 0
    while firstcount < len(first) and secondcount < len(second):
        if first[firstcount] < second[secondcount]:
            dataArray[originalcount]=first[firstcount]
            firstcount += 1
        else:
            dataArray[originalcount]=second[secondcount]
            secondcount += 1
        originalcount += 1
    while firstcount < len(first):
        dataArray[originalcount]=first[firstcount]
        firstcount += 1
        originalcount += 1
    while secondcount < len(second):
        dataArray[originalcount]=second[secondcount]
        secondcount += 1
        originalcount += 1
    return dataArray

"""
/*****************************************************************
* Description: main function
* Program starts here and calls functions as necessary
* Calls the mergesort function that completes most of the program's
* functionality.
*****************************************************************/
"""


if __name__ == "__main__":

    n = 1000
    for count in range(0, 13):
        starttime = time.time()
        if count == 1:
            n = 2000
        if count == 2:
            n = 5000
        if count == 3:
            n = 10000
        unsortedArray = []
        for x in range (n):
            unsortedArray.append(random.randint (0, 10000))
        mergeSort(unsortedArray, n)
        print("Array Size: {}".format(n))
        print("Time to Complete: {} seconds".format(time.time() - starttime))
        n += 10000
```

Text Copy of Merge Sort Time Code

```
import time                                                                    #
Import the time library to measure time
import random                                                                  #
Import the random number generator library

"""
/*********************************************************************
* Description: mergeSort function
* Functions receives a number array from main and performs a merge sort
* to place the numbers in ascending order.  The function takes and array
* and recursively devides until until it reaches array sizes of one.
* It passes data to a merge function that combines the elements in
* ascending order into the original array
*********************************************************************/
"""


def mergeSort (dataArray, arraySize):
    if arraySize < 1:                                                          # If
Array Size is less than 1, this is invalid
        print("Array Size {} is invalid".format(arraySize))                    #
print an error to the console
    elif arraySize == 1:                                                       # If
the array size is one it's already sorted
        return dataArray                                                       #
Return the array as is without performing any work
    else:                                                                      # If
the array is greater than size one (less than one and already sorted)
        first = dataArray[:(arraySize // 2)]                                   #
Split the array and assign the left (bottom) half to variable (Use Floor Division)
        last = dataArray[(arraySize // 2):]                                    #
Split the array and assign the right (top) half to variable (Use Floor Division)
        firstLength = len(first)                                               # Get
left array length and assign to variable
        lastLength = len(last)                                                 # Get
left array length and assign to variable
        firstArray = mergeSort(first, firstLength)                             #
Recursively call the mergeSort function and send the first half of the array
        secondArray =mergeSort(last, lastLength)                               #
Recursively call the mergeSort function and send the second half of the array
    return merge (dataArray, firstArray, secondArray)


"""
/*********************************************************************
* Description: merge function
* Called by the mergeSort function to evaluate and merge the split array
* elements back into a single sorted array.  Receives the original array
* the split right and left arrays and their respective lengths.  Returns
* a sorted integer array to mergerSort
*********************************************************************/
"""

def merge (dataArray, first, second):
    firstcount = secondcount = originalcount = 0                               #
create three counter variables and initialize to zero
    while firstcount < len(first) and secondcount < len(second):               #
while both the left and right counters are less than their array lengths
        if first[firstcount] < second[secondcount]:                            # If
the left array is less than the right
            dataArray[originalcount]=first[firstcount]                         #
Assign it to the current element in the original array
            firstcount += 1                                                    #
Increment left array counter by one (hate that python can't just do ++)
        else:                                                                  #
otherwise if the right array element is smaller
```

```
            dataArray[originalcount]=second[secondcount]                    #
Assign it to the current element in the original array
            secondcount += 1                                                #
Increment Right array counter by one
        originalcount += 1                                                  #
Increment original array counter by one
    while firstcount < len(first):                                          # If
the first count is still less than the left array length
        dataArray[originalcount]=first[firstcount]                          #
assign the current element from the left array to the current element in the original
array
        firstcount += 1                                                     #
Increment counter by one
        originalcount += 1                                                  #
Increment counter by one
    while secondcount < len(second):                                        #
While the left count is still less than the right array length
        dataArray[originalcount]=second[secondcount]                        #
Assign it's value to the current element in the original array
        secondcount += 1                                                    #
Increment counter by one
        originalcount += 1                                                  #
Increment counter by one
    return dataArray                                                        #
Return the sorted array to the main function for writing.

"""
/*************************************************************************
* Description: main function
* Program starts here and calls functions as necessary
* Calls the mergesort function that completes most of the program's
* functionality.
*************************************************************************/
"""


if __name__ == "__main__":

    n = 1000                                                                # Set
number of variables that need to be generated
    for count in range(0, 13):
        starttime = time.time()                                             #
Collect the start time at the initial execution of the program
        if count == 1:
            n = 2000
        if count == 2:
            n = 5000
        if count == 3:
            n = 10000
        unsortedArray = []                                                  #
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
            unsortedArray.append(random.randint (0, 10000))                 #
Append the random elements to the unsorted Array
        mergeSort(unsortedArray, n)                                         # Send
the array to the sort function to be sorted
        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.
        n += 10000
```

b. Use the system clock to record the running times of each algorithm for n = 1000, 2000, 5000, 10,000, … You may need to modify the values of n if an algorithm runs too fast or too slow to collect the running time data. If you program in C your algorithm will run faster than if you use python. You will need at least seven values of t (time) greater than 0. If there is a variability in the times between runs of the same algorithm you may want to take the average time of several runs for each value of n.

Answer:

Merge Sort Average Run Times

```
hippl@Watson /cygdrive/d/Users/hippl/Documents/GitHub/CS_325-400/Homework_1
$ python mergetime.py
Array Size: 1000
Time to Complete: 0.00241208076477 seconds
Array Size: 2000
Time to Complete: 0.00485610961914 seconds
Array Size: 5000
Time to Complete: 0.0122570991516 seconds
Array Size: 10000
Time to Complete: 0.0255370140076 seconds
Array Size: 20000
Time to Complete: 0.0531551837921 seconds
Array Size: 30000
Time to Complete: 0.0766959190369 seconds
Array Size: 40000
Time to Complete: 0.103230953217 seconds
Array Size: 50000
Time to Complete: 0.131685972214 seconds
Array Size: 60000
Time to Complete: 0.158899068832 seconds
Array Size: 70000
Time to Complete: 0.192122936249 seconds
Array Size: 80000
Time to Complete: 0.215612888336 seconds
Array Size: 90000
Time to Complete: 0.242655992508 seconds
Array Size: 100000
Time to Complete: 0.270738124847 seconds
```

Insertion Sort Average Run Times

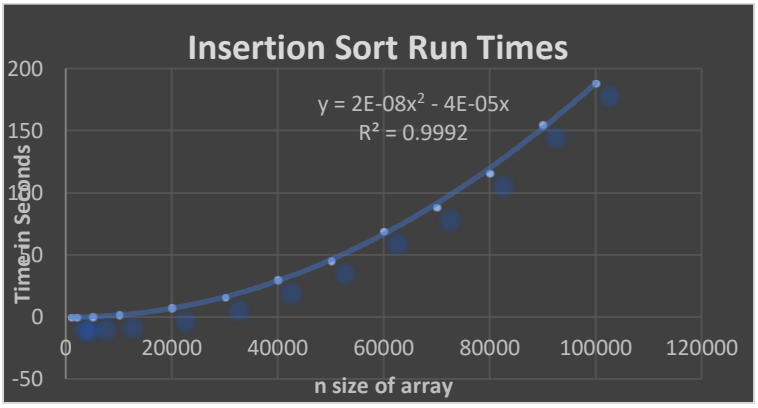
```
hippl@Watson /cygdrive/d/Users/hippl/Documents/GitHub/CS_325-400/Homework_1
$ python inserttime.py
Array Size: 1000
Time to Complete: 0.018031835556 seconds
Array Size: 2000
Time to Complete: 0.0703489780426 seconds
Array Size: 5000
Time to Complete: 0.432154893875 seconds
Array Size: 10000
Time to Complete: 1.77549791336 seconds
Array Size: 20000
Time to Complete: 7.24278497696 seconds
Array Size: 30000
Time to Complete: 16.1745581627 seconds
Array Size: 40000
Time to Complete: 29.8428008556 seconds
Array Size: 50000
Time to Complete: 45.7291140556 seconds
Array Size: 60000
Time to Complete: 69.3028559685 seconds
Array Size: 70000
Time to Complete: 88.496694088 seconds
Array Size: 80000
Time to Complete: 115.828573942 seconds
Array Size: 90000
Time to Complete: 155.123039007 seconds
Array Size: 100000
Time to Complete: 188.390840054 seconds
```

c. For each algorithm 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 both algorithms together on a combined graph. Which graphs represent the data best?

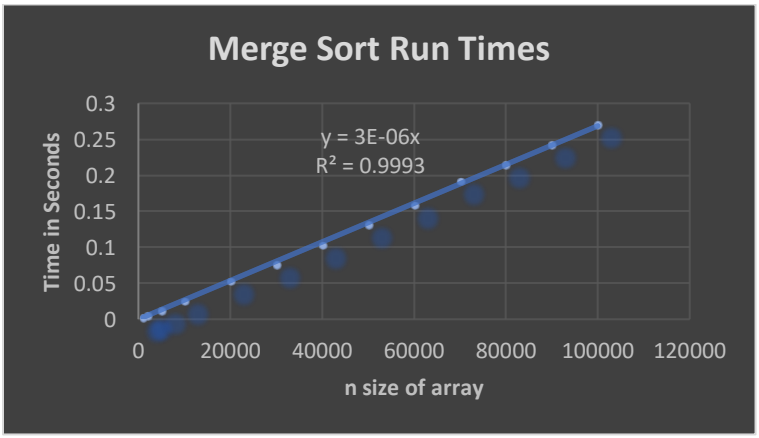
Answer:

Below are the individual runtime graphs generated from the runtime output on varying sized arrays with randomized integer values in both insertion sort and merge sort algorithms (actual results for each array size are presented as screenshots in part b). The charts were created in Excel rather than Matlab. I found that the Scatter Plot graph in Excel best represented the runtime data. I also felt that the graphs were best represented individually rather than in combination. The scales were far too extreme between the merge sort and the insertion sort algorithms. When the two graphs are presented together in comparison, the scaling is relatively poor (even the professor mentioned in the HW 5 Help video that she would not combine the two graphs because of this). That being said, also included below is a graph comparing the runtimes of both the insertion and merge sorting algorithms. The graphs combined does give a good visual of how much faster the insertion sort grows in time compared to merge sort.

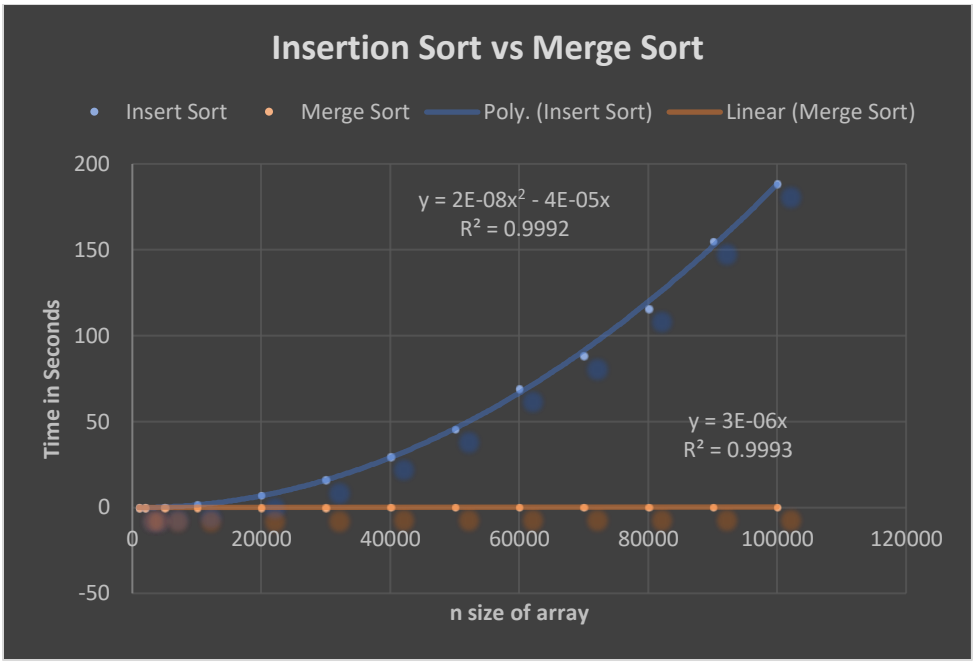
Insertion Sort Run Times (Graph)



Merge Sort Run Times (Graph)



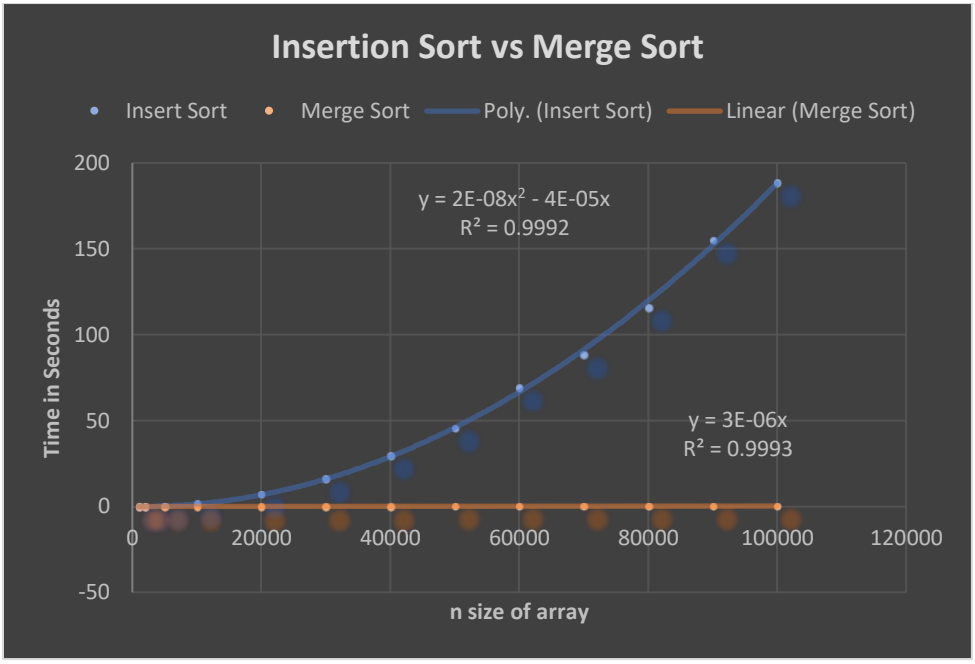
Insertion Sort vs Merge Sort (Graph)



d. What type of curve best fits each data set? Again, you can use Excel, Matlab, any software or a graphing calculator to calculate a regression equation. Give the equation of the curve that best "fits" the data and draw that curve on the graphs of created in part c).

Answer:

When individually graphed the 2nd degree polynomial (exponential) trend line matched insertion sort best with an equation of $y = 2E - 08x^2 - 4E - 05x$ (gives Regression of $R^2 = 0.992$). However, a linear trend line was better suited for merge sort with an equation of $y = 3E - 06x$ (gives Regression of $R^2 = 0.993$). The graph screen shots above include the trend line curves necessary for this question. Just to be safe, I included the combination graph a second time below.



e. How do your experimental running times compare to the theoretical running times of the algorithms? Remember, the experimental running times were "average case" since the input arrays contained random integers.

Answer:

The experimental running times were in line with the theoretical running times expected from both algorithms. The insertion sort was almost exactly on point with the theoretical O(n^2) performance associated with the algorithm and we can certainly see that the runtime increases exponential as the array size gets larger. The running time of merge sort deviated slightly as it looked more linear O(n) rather than the expected O($n \log n$). As mentioned in the lecture videos, this is likely because the effect of the logarithm was negligible with the size of our randomized test arrays. The "Why does my nlogn graph look linear" document states that the variation is so small when compared to the values of n that for all practical purposes it is a constant. In theory the algorithm may be $n \log n$ but for some narrow ranges of n it may appear almost linear.

EXTRA CREDIT: A Tale of Two Algorithms: ***It was the best of times, it was the worst of times…***

> **Generate best case and worst-case inputs for the two algorithms and repeat the analysis in parts b) to e) above. To receive credit, you must discuss how you generated your inputs and your results. Submit your code to TEACH in a zip file with the code from Problem 4.**

Answer:

Code was for each sorting algorithm has been included in my submission to TEACH.

Insertion Sort Best vs Worst Case Scenario

The best-case scenario for Insertion sort would be having the array already sorted (ascending order). This would allow the program to function at linear time $O(n)$ which is even faster than the output of merge sort. The worst-case scenario for Insertion sort would be when the array items are inserted in the exact opposite order of what would be considered sorted (descending order). This would cause the program to perform at a speed of approximately exponential $O(n^2)$ where the algorithm needs to make a comparison with every preceding element for each element that it is attempting to sort. To accomplish the creation of these arrays, I had a function that generated an array of increasing length that inserted elements in ascending order to replicate best case scenarios and then descending order to replicate worst-case scenarios. I then used the system clock to record the time it took insertion sort to sort each back into ascending order. From the results, we can see that the function performs significantly slower than when random integers were assigned to those same array lengths. For example, an array of 100,000 elements already sorted is processed by insertion sort in less than 1 second (0.0450789928436 seconds) whereas the same sized array with integers inserted in the opposite order takes over 12 minutes (737.2076931 seconds) to complete. Code and Results provided below.

```
Jamess-MacBook-Pro:Homework_1 hippler$ python insert_extracredit.py
**** Insertion Sort Best Case *****
Insertion Sort Best Case Array Size: 1000
Time to Complete: 0.00053596496582 seconds
Insertion Sort Best Case Array Size: 2000
Time to Complete: 0.000998973846436 seconds
Insertion Sort Best Case Array Size: 5000
Time to Complete: 0.00273895263672 seconds
Insertion Sort Best Case Array Size: 10000
Time to Complete: 0.00452089309692 seconds
Insertion Sort Best Case Array Size: 20000
Time to Complete: 0.00827598571777 seconds
Insertion Sort Best Case Array Size: 30000
Time to Complete: 0.0127859115601 seconds
Insertion Sort Best Case Array Size: 40000
Time to Complete: 0.0177791118622 seconds
Insertion Sort Best Case Array Size: 50000
Time to Complete: 0.0246050357819 seconds
Insertion Sort Best Case Array Size: 60000
Time to Complete: 0.0249190330505 seconds
Insertion Sort Best Case Array Size: 70000
Time to Complete: 0.0320208072662 seconds
Insertion Sort Best Case Array Size: 80000
Time to Complete: 0.0349249839783 seconds
Insertion Sort Best Case Array Size: 90000
Time to Complete: 0.0390040874481 seconds
Insertion Sort Best Case Array Size: 100000
Time to Complete: 0.0450789928436 seconds


**** Insertion Sort Worst Case *****
Insertion Sort Worst Case Array Size: 1000
Time to Complete: 0.0773389339447 seconds
Insertion Sort Worst Case Array Size: 2000
Time to Complete: 0.286906003952 seconds
Insertion Sort Worst Case Array Size: 5000
Time to Complete: 1.62898492813 seconds
Insertion Sort Worst Case Array Size: 10000
Time to Complete: 6.64928483963 seconds
Insertion Sort Worst Case Array Size: 20000
Time to Complete: 26.4836649895 seconds
Insertion Sort Worst Case Array Size: 30000
Time to Complete: 60.1269919872 seconds
Insertion Sort Worst Case Array Size: 40000
Time to Complete: 104.545208931 seconds
Insertion Sort Worst Case Array Size: 50000
Time to Complete: 162.955533981 seconds
Insertion Sort Worst Case Array Size: 60000
Time to Complete: 258.838598013 seconds
Insertion Sort Worst Case Array Size: 70000
Time to Complete: 341.607811928 seconds
Insertion Sort Worst Case Array Size: 80000
Time to Complete: 457.42297101 seconds
Insertion Sort Worst Case Array Size: 90000
Time to Complete: 615.430315018 seconds
Insertion Sort Worst Case Array Size: 100000
Time to Complete: 737.2076931 seconds
Jamess-MacBook-Pro:Homework_1 hippler$
```

```
"""
/*************************************************************************
* Description: main function
* Program starts here and calls functions as necessary
* Calls the mergesort function that completes most of the program's
* functionality.
*************************************************************************/
"""

if __name__ == "__main__":
    print("**** Insertion Sort Best Case *****")
    n = 1000
    for count in range(0, 13):
        starttime = time.time()
        if count == 1:
            n = 2000
        if count == 2:
            n = 5000
        if count == 3:
            n = 10000
        bestCase = []
        for x in range (n):
            bestCase.append(x + 1)
        insertionSort(bestCase, n)
        print("Insertion Sort Best Case Array Size: {}".format(n))
        print("Time to Complete: {} seconds".format(time.time() - starttime))
        n += 10000

    print("\n\n**** Insertion Sort Worst Case *****")
    n = 1000
    for count in range(0, 13):
        starttime = time.time()
        if count == 1:
            n = 2000
        if count == 2:
            n = 5000
        if count == 3:
            n = 10000
        worstCase = []
        arraySize = n;
        for x in range (n):
            worstCase.append(arraySize)
            arraySize -= 1
        insertionSort(worstCase, n)
        print("Insertion Sort Worst Case Array Size: {}".format(n))
        print("Time to Complete: {} seconds".format(time.time() - starttime))
        n += 10000
```

Merge Sort Best vs Worst Case Scenario

Merge Sort tends to operate at approximately the speed of $O(n \log n)$ as is has to recursively split the array and then remerge the elements linearly regardless of the order of the array elements. This means that merge sort will generally operate less efficiently than even insertion sort when processing arrays that are sorted or nearly sorted. For the best-case scenario, I inserted the integers in ascending order to match what would be considered sorted after merge sort completes. To emulate a worst-case scenario, I followed the recommendation provided in the Geeks for Geeks "Find a permutation that causes worst case of merge sort" (https://www.geeksforgeeks.org/find-a-permutation-that-causes-worst-case-of-merge-sort/) designed to maximize the number of required comparisons. To accomplish this, the integers need to be inserted into the array such that "the left and right sub-array involved in merge operation should store alternate elements of sorted array" resulting in each element being compared at least once. I managed this by inserting odd numbers in the left side of the array and even numbers on the right, both in ascending order. When processing the unsorted, worst-case array we can see that merge sort is slower than the best-case scenario but only marginally. The algorithm is only saving time when comparing ($\log n$) but is still being dominated by the linear (n) need to split and merge all elements of the array. Code and Results provided below.

```
Jamess-MacBook-Pro:Homework_1 hippler$ python merge_extracredit.py
**** Merge Sort Best Case *****
Merge Sort Best Case Array Size: 1000
Time to Complete: 0.00578498840332 seconds
Merge Sort Best Case Array Size: 2000
Time to Complete: 0.0093150138855 seconds
Merge Sort Best Case Array Size: 5000
Time to Complete: 0.0261809825897 seconds
Merge Sort Best Case Array Size: 10000
Time to Complete: 0.0472798347473 seconds
Merge Sort Best Case Array Size: 20000
Time to Complete: 0.0928549766541 seconds
Merge Sort Best Case Array Size: 30000
Time to Complete: 0.133078813553 seconds
Merge Sort Best Case Array Size: 40000
Time to Complete: 0.18545794487 seconds
Merge Sort Best Case Array Size: 50000
Time to Complete: 0.236317157745 seconds
Merge Sort Best Case Array Size: 60000
Time to Complete: 0.296148061752 seconds
Merge Sort Best Case Array Size: 70000
Time to Complete: 0.344013929367 seconds
Merge Sort Best Case Array Size: 80000
Time to Complete: 0.394474029541 seconds
Merge Sort Best Case Array Size: 90000
Time to Complete: 0.439182043076 seconds
Merge Sort Best Case Array Size: 100000
Time to Complete: 0.483067035675 seconds


**** Merge Sort Worst Case *****
Merge Sort Worst Case Array Size: 1000
Time to Complete: 0.00393009185791 seconds
Merge Sort Worst Case Array Size: 2000
Time to Complete: 0.00768899917603 seconds
Merge Sort Worst Case Array Size: 5000
Time to Complete: 0.0220980644226 seconds
Merge Sort Worst Case Array Size: 10000
Time to Complete: 0.0514719486237 seconds
Merge Sort Worst Case Array Size: 20000
Time to Complete: 0.0912239551544 seconds
Merge Sort Worst Case Array Size: 30000
Time to Complete: 0.140603065491 seconds
Merge Sort Worst Case Array Size: 40000
Time to Complete: 0.18675494194 seconds
Merge Sort Worst Case Array Size: 50000
Time to Complete: 0.235261917114 seconds
Merge Sort Worst Case Array Size: 60000
Time to Complete: 0.28733086586 seconds
Merge Sort Worst Case Array Size: 70000
Time to Complete: 0.37122797966 seconds
Merge Sort Worst Case Array Size: 80000
Time to Complete: 0.389809131622 seconds
Merge Sort Worst Case Array Size: 90000
Time to Complete: 0.446621179581 seconds
Merge Sort Worst Case Array Size: 100000
Time to Complete: 0.5421230793 seconds
Jamess-MacBook-Pro:Homework_1 hippler$
```

```
"""
/*********************************************************************
* Description: bestCase function
* Receives an array size from n and creates a sorted integer array
* (ascending) and calls merge sort to sort.
*********************************************************************/
"""

def bestCase (n):
    bestCase = []
    for x in range (n):
        bestCase.append(x + 1)
    mergeSort(bestCase, n)
    print("Merge Sort Best Case Array Size: {}".format(n))

"""
/*********************************************************************
* Description: worstCase function
* Receives an array size from n and creates a worst case scenario
* for merge sort to process (forces a maximum number of comparisons).
* Accomplishes this assigning odd number to the first half of the array
* and even numbers to the second half
*********************************************************************/
"""

def worstCase(n):
    worstleft = []
    worstright = []
    for x in range (n):
        if x % 2 == 0:
            worstleft.append(x + 1)
        else:
            worstright.append(x + 1)
    completeWorst = worstleft + worstright
    mergeSort(completeWorst, n)
    print("Merge Sort Worst Case Array Size: {}".format(n))

"""
/*********************************************************************
* Description: main function
* Program starts here and calls functions as necessary
* Calls the mergesort function that completes most of the program's
* functionality.
*********************************************************************/
"""

if __name__ == "__main__":
    print("**** Merge Sort Best Case *****")
    n = 1000
    for count in range(0, 13):
        starttime = time.time()
        if count == 1:
            n = 2000
        if count == 2:
            n = 5000
        if count == 3:
            n = 10000
        bestCase(n)
        print("Time to Complete: {} seconds".format(time.time() - starttime))
        n += 10000

    print("\n\n**** Merge Sort Worst Case *****")
    n = 1000
    for count in range(0, 13):
        starttime = time.time()
        if count == 1:
            n = 2000
        if count == 2:
            n = 5000
        if count == 3:
            n = 10000
        worstCase(n)
        print("Time to Complete: {} seconds".format(time.time() - starttime))
        n += 10000
```