Name: Brian Allison
Date: 9/30/2017
Course: CS325 Online
Description: Assignment 1

1) (3 pts) Describe a Θ (*n* lg*n*) time algorithm that, given a set S of *n* integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x. Explain why the running time is Θ (*n* lg*n*).

Ans:
We can use MergeSort to sort the array. MergeSort recursively splits the array into upper 0…n/2 and lower n/2+1…n halves. On each call to MergeSort the two halves are merged, comparing the left-most values and inserting them into an array in proper order. MergeSort is Θ (*n* lg*n*).

We can search a sorted array for a specific value using BinarySearch. A BinarySearch starts in the middle of the array at array[n/2], checks if the search term exists in that index, and if not, it checks if the search term is larger or smaller than the value in the current index. If the value is larger, it continues by calling BinarySearch on the left side of the array from the current index. This continues until the search term is found, or there are no more indexes left to search. In order to find whether two elements exist whose sum is x, we can cycle through each index of the array, subtract x from the value in that index, and BinarySearch the array for the result.

BinarySearch is O(lg(n)). In the worst case, we are running BinarySearch n times when we cycle through each item of the array. This results in O(n lgn) time. We then add the time that MergeSort takes to the time that BinarySearch takes: (n lgn) + (n lgn) = 2(n lgn). Dropping the constant and accounting for the fact that MergeSort will always take *n* lg*n* time no matter the input (different from BinarySearch), we get a final result of Θ (*n* lg*n*).

2) (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) = Θ(g(n)). Determine which relationship is correct and explain.

a. f(n) = $n^{0.25}$; g(n) = $n^{0.5}$
Ans: f(n) = O(g(n)
g(n) is growing faster
the limit of f(n)/g(n) as n approaches infinity is 0

b. f(n) = log n^2; g(n) = ln n
Ans: f(n) = Θ(g(n))
log n^2 and ln n only differ by constants

c. f(n) = nlog n; g(n) =$n\sqrt{n}$
Ans: f(n) = O(g(n)
g(n) is growing faster
the limit of f(n)/g(n) as n approaches infinity is 0

d. f(n) = 4^n; g(n) = 3^n
Ans: f(n) = Ω (g(n))
f(n) is growing faster
the limit of f(n)/g(n) as n approaches infinity is infinity

e. $f(n) = 2^n$; $g(n) = 2^{n+1}$

Ans: f(n) = Θ(g(n))
f(n)/g(n) = ½

f. $f(n) = 2^n$; $g(n) = n!$

Ans: f(n) = O(g(n)
g(n) is growing faster than f(n)
the limit of f(n)/g(n) as n approaches infinity is 0

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 a counter example.

a. If $f_1(n) = O(g(n))$ and $f_2(n) = O(g(n))$ then $f_1(n) + f_2(n) = O(g(n))$.

Proof:

Assume that $f_1(n) = O(g(n))$ and $f_2(n) = O(g(n))$.
There are 3 possibilities for $f_1(n)$ and $f_2(n)$: 1) $f_1(n)$ is growing faster than $f_2(n)$. 2) $f_2(n)$ is growing faster than $f_1(n)$. 3) $f_1(n)$ and $f_2(n)$ are growing at the same rate.
For the first possibility, if $f_1(n)$ is growing faster than $f_2(n)$, then the highest degree term of $f_2(n)$ must be lower than the highest degree term of $f_1(n)$. For $f_1(n) + f_2(n)$, the highest degree term has not changed. Therefore, based on the definition of big O notation, $f_1(n) + f_2(n)$ must be equal to O(g(n)).

Then, of course, the second possibility will have the same result. The only difference being that the highest degree term comes from $f_2(n)$ rather than $f_1(n)$. Therefore, $f_1(n) + f_2(n)$ must be equal to O(g(n)).

For the third possibility, if $f_1(n)$ and $f_2(n)$ are growing at the same rate, then they must have terms with the same leading degree. Since we are not concerned with the lower order terms, they can be dropped, and only the highest order terms need to be added together. Based on the rules of mathematics, adding two terms with same degree results in another term with the same degree.
Then, as n increases, there must be a point n_0 at which $f_2(n)$ will never be larger than $f_1(n)$.

b. If $f(n) = O(g_1(n))$ and $f(n) = O(g_2(n))$ then $g_1(n) = \Theta(g_2(n))$

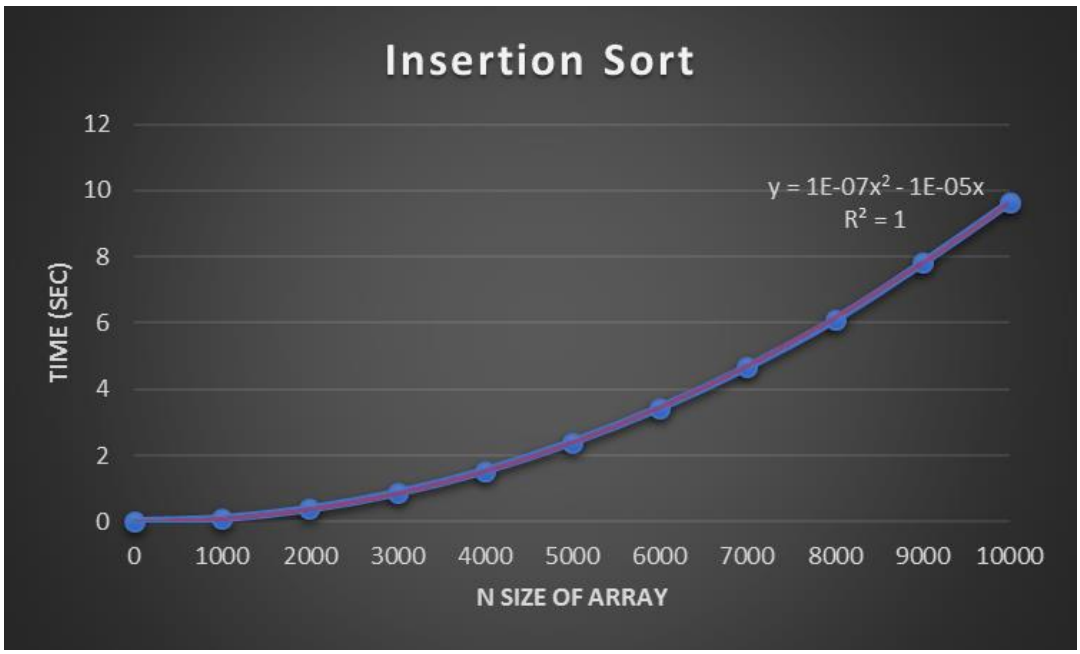
Counter example:
Suppose that $f(n) = n^2$, $g_1(n) = n^3$ and $g_2(n) = n^2$. Then, based on the definition of big O notation, $f(n) = O(g_1(n))$ and $f(n) = O(g_2(n))$. Based on the definition of Θ notation, there must exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g_2(n) \le g_1(n) \le c_2 g_2(n)$ for all $n \ge n_0$. There is no point n_0 at which x^2 will be larger than x^3 for all $n \ge n_0$. Therefore, it is not true that "If $f(n) = O(g_1(n))$ and $f(n) = O(g_2(n))$ then $g_1(n) = \Theta(g_2(n))$".

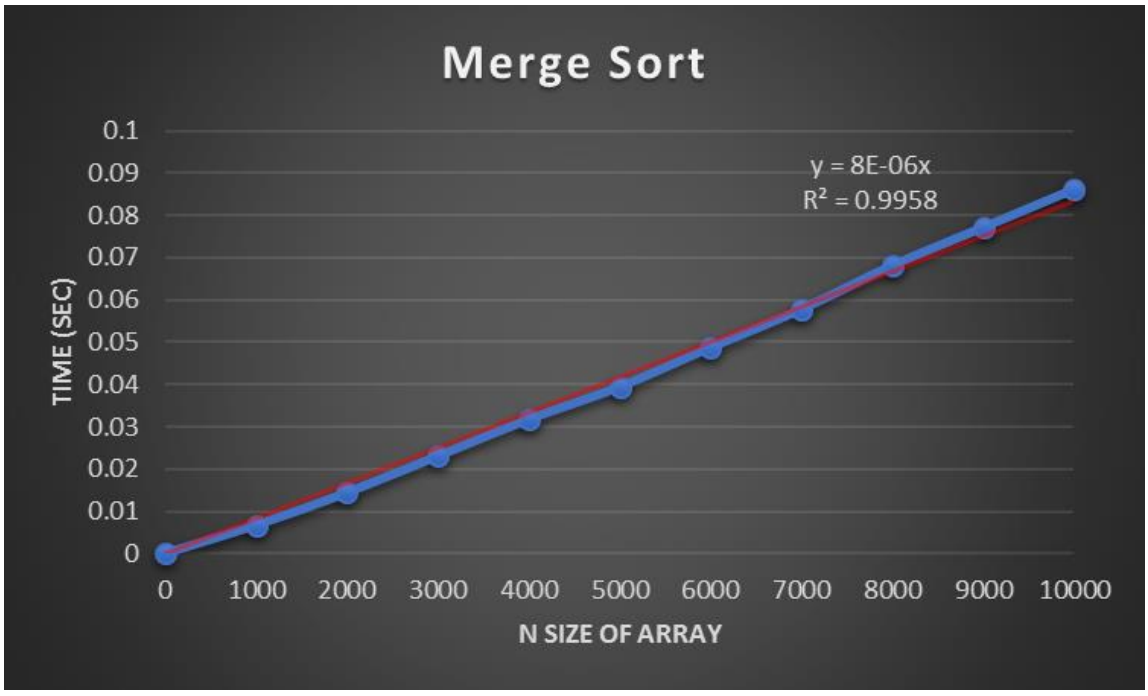
5) (10 pts) **Merge Sort vs Insertion Sort Running time analysis**

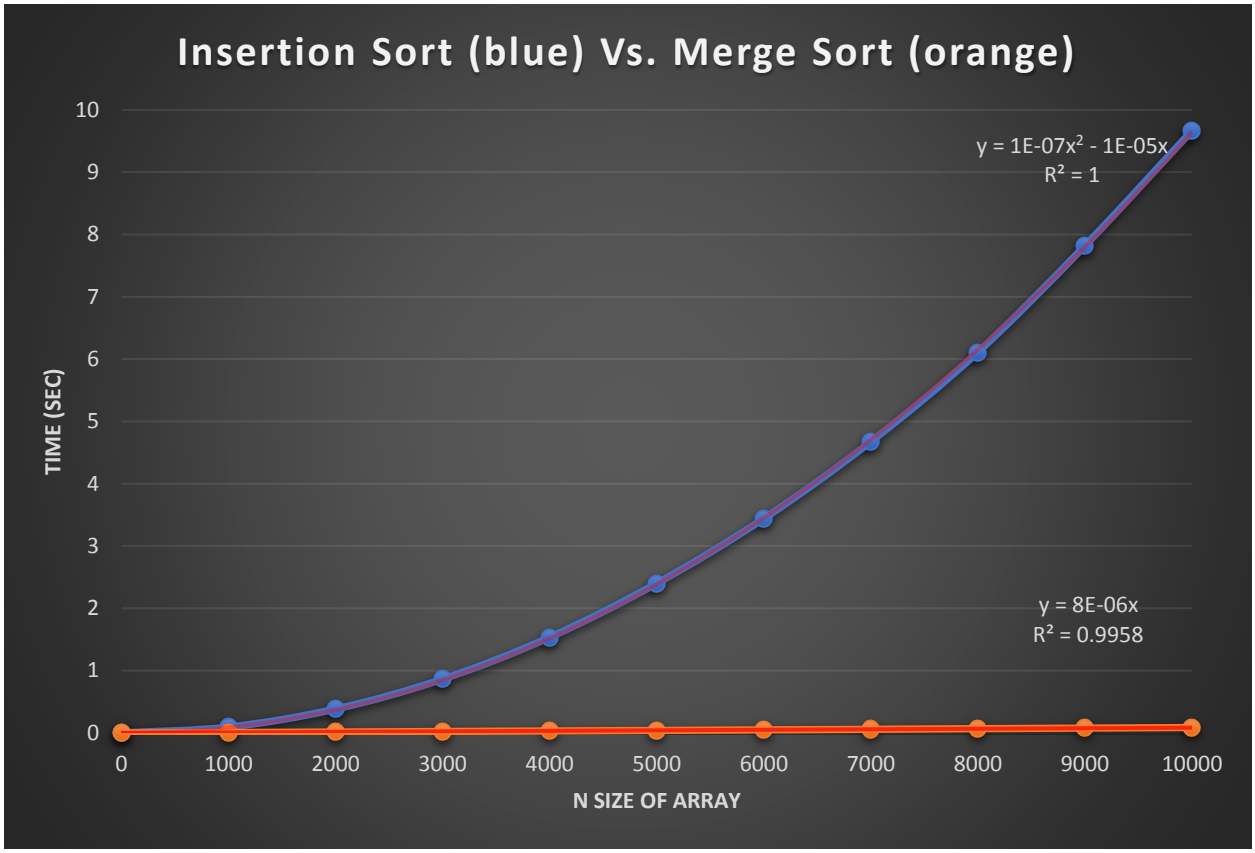
The goal of this problem is to compare the experimental running times of the two sorting algorithms.

a and b) code for insertion and merge sort experiment is included at the bottom of the document.

a.







Which graphs represent the data best?

Ans: They both represent the data well in different ways. The combined graph shows that as N increases, the time required for insertion sort grows by so much, that you cannot properly scale the graph to show that time required for merge sort is increasing. However, if you look at the separate insertion sort and merge sort graphs, you can see that the time required for insertion sort grows at a quadratic rate as n increases and the time required for merge sort grows at a linear rate as time increases.

d) What type of curve best fits each data set?

Ans: A linear curve is the best fit for Merge Sort and quadratic curve is the best fit for Insertion Sort. The actually equations are shown on the graphs.

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.

Ans: The experimental running times are consistent with the theoretical running times. Theoretically merge sort is always Θ (*n* lg*n*). For merge sort, time increased at a linear rate, as would be expected. Looking at the graph, lgn does not seem to have much of an effect on the rate of increase.
Insertion sort is $O(n^2)$. Time required is clearly increasing at a quadratic rate, and this can be seen in the graph and in the model equation that perfectly fits the graph.

Modified code for testing running times:

merge sort

```
import time
import random


# sorts an array of ints in O(nlgn) time
def merge_sort(array):
    if len(array) < 2:
        return array
    else:
        mid_split = len(array)/2
        left = merge_sort(array[:int(mid_split)])
        right = merge_sort(array[int(mid_split):])
        array = merge(left, right)
        return array


# helper function for merge_sort. Creates a single sorted array
# made up of the values of the two arrays that are passed in.
# The two arrays that are passed in must already be sorted.
def merge(left, right):
    s_array = []
    i = 0
    j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            s_array.append(left[i])
            i = i + 1
        else:
            s_array.append(right[j])
            j = j + 1

    # source: below if/else blocks created with help from
https://www.youtube.com/watch?v=3aTfQvs-_hA&t=255s
    if len(left) <= i:
        s_array.extend(right[j:])
    else:
        s_array.extend(left[i:])
    return s_array


# below running time code was written based on canvas lecture for hw1 problem 5
tot = 0
ave = 0
rand_array = []

file_out = open('merge_time.txt', 'w')
for k in range(0, 11000, 1000):
    file_out.write(str(k) + ' , ')
    print(k, end=' , ')
    tot = 0
    ave = 0

    for l in range(3):
        rand_array = []
        for m in range(k):
            random_number = random.randint(0, 1000)
            rand_array.append(random_number)

        start_time = time.time()
```

```
        merge_sort(rand_array)
        end_time = time.time()

        elapsed_time = end_time - start_time
        tot = tot + elapsed_time
        file_out.write(str(elapsed_time) + ' , ')
        print(elapsed_time, end=' , ')

    ave = tot/3
    print("average: " + str(ave))
    file_out.write("average: " + str(ave) + '\n')

file_out.close()
```

insertion sort

```
import random
import time

# sorts an array of ints in O(n^2) time
def insertion_sort(array):
    for i in range(1, len(array)):
        key = array[i]
        j = i - 1
        while key < array[j] and j >= 0:
            array[j + 1] = array[j]
            array[j] = key
            j = j - 1
    return array


# below running time code was written based on canvas lecture for hw1 problem 5
tot = 0
ave = 0
rand_array = []

file_out = open('insert_time.txt', 'w')
for k in range(0, 11000, 1000):
    file_out.write(str(k) + ' , ')
    print(k, end=' , ')
    tot = 0
    ave = 0

    for l in range(3):
        rand_array = []
        for m in range(k):
            random_number = random.randint(0, 1000)
            rand_array.append(random_number)

        start_time = time.time()
        insertion_sort(rand_array)
        end_time = time.time()

        elapsed_time = end_time - start_time
        tot = tot + elapsed_time
        file_out.write(str(elapsed_time) + ' , ')
        print(elapsed_time, end=' , ')

    ave = tot/3
    print("average: " + str(ave))
    file_out.write("average: " + str(ave) + '\n')

file_out.close()
```