Shannon Jeffers
CS 325 F 17
HW1

1. This could be accomplished by using a merge sort so first sort the array (which we will call S), and then using a for loop to visit every array element and plug x-S[i] into a binary search. It would tell us if two numbers added up to x because S[i] would be the first number and if binary search returned true to finding x-S[i] it would indicate we had 2 numbers in the array that added up to x. This would satisfy the run time requirements because merge sort is Θ(n lg n), a for loop is Θ(n) and as we learned in Data Structures the binary search is Θ(lg n). Since the binary search is within the for loop this would indicate these two combined would have a complexity of Θ(n lg n). Adding this all together we would get Θ(n lg n) + Θ(n lg n) which could simply be stated as Θ(n lg n).

2. For all the parts below, to determine the relationship we want to see if the limit of f(n)/g(n) is 0, some constant or infinity. If the limit is 0 f(n) is O(g(n)), if the limit is a constant f(n) is Θ(g(n)) and if the limit is infinity f(n) is Ω(g(n)).

   A. $n^{.25}/n^{.5+}$ can be simplified using the rules of exponents into $1/n^{.25}$. The limit of $1/n^{.25}$ as n approaches infinity is 0, therefor f(n) is O(g(n)).

   B. No matter what the base is, all logs have the same growth rate, so $\log(n^2)$ has the same growth rate as ln(n) which would indicate that f(n) is Θ(g(n))

   C. n log n/n sqrt(n) can be simplified to simply log(n)/sqrt(n). The limit of log(n)/sqrt(n) as n approaches infinity is 0 because nlog in has a slower growth rate than n sqrt n, so f(n) is O(g(n))

   D. When we try to take the limit we get $4^n/3^n$. We can simplify this by pulling the exponent out which gives us $(4/3)^n$. 4/3 is a number greater than 1 and will continue to be greater than 1 when the n is applied. The limit is = infinity so f(n) is Ω(g(n))

   E. When we take f(n)/g(n) with this equation we get $2^n/2^{n+1}$. The bottom of this equation can be pulled apart to give us $2^n/(2 * 2^n)$. After canceling out like terms we get 1/2 , which is a constant so they have a relatively similar growth rate which means f(n) is Θ(g(n))

   F. n! is a much faster growing function than $2^n$ which is apparently when we take the limit, as the limit is 0. Therefore, we can conclude f(n) is O(g(n))

3.

   A. Step 1: By definition $f_1(n) = O(g(n))$ implies that there exists positive constants $c_1$ and $n_0$ such that $0 <= f_1(n) <= c_1 g(n)$ for all $n >= n_0$.

   Step2: By definition $f_2(n) = O(g(n))$ implies that there exists positive constants $c_2$ and $n_1$ such that $0 <= f_2(n) <= c_2 g(n)$ for all $n >= n_1$.

Step3: Show that $f_1(n) + f_2(n) = O(g(n))$ that is there exists positive constants $c_3$ and $n_2$ such that $0 <= f_1(n) + f_2(n) <= c_3g(n)$ for all $n >= n_2$.

If we let $c_3 = c_1 + c_2$ and $n_2 = \max\{n_0, n_1\}$ then it is easy to see that $f_1(n) + f_2(n) = O(g(n))$ for all $n >= n_2$

B. Solve this using a counter example. If $f(n) = n$, $g_1(n) = n^2$ and $g_2(n) = n^3$ then $g_1(n) != \Theta(g_2(n))$ yet $f(n) = O(g_1(n))$ and $f(n) = O(g_2(n))$, because $n <= n^2$ and $n <= n^3$ but $n^2 != n^3$.

4. Question number 4 has been turned in on TEACH.
5. Now let us check out the difference in running times between insertion sort and merge sort.

A. I removed the file parts of the code from number 4 since we no longer needed the output file and were no longer working with an input file. I also made the code where you could either change the value of n, or uncomment out the input statement and get the value of n while the code was running. I simply changed the value of n while doing my testing.

**Insert Sort**

```
import time
from random import randint

#n = int(input("Enter a number for n: "))
n=100000
myList = []

for x in range(n):
    i = randint(1,10000)
    myList.append(i)


#insertSort sorts a list by insertion
def insertSort(myList):
    #goes through every element from 1 (the second element) to the length
    for num in range(1, len(myList)):
            cur = myList[num]
            temp = num - 1
            #while we arent at the front of the list or the element to the left
            #of the element we are looking for isn't bigger.
            while temp > -1 and myList[temp] > cur:
                    #swap the elements.
                    myList[temp + 1] = myList[temp]
                    myList[temp] = cur
                    temp -= 1


#collects the time it takes to run
start = time.time()
insertSort(myList)
end = time.time()

print(end - start)
```

**Merge Sort**

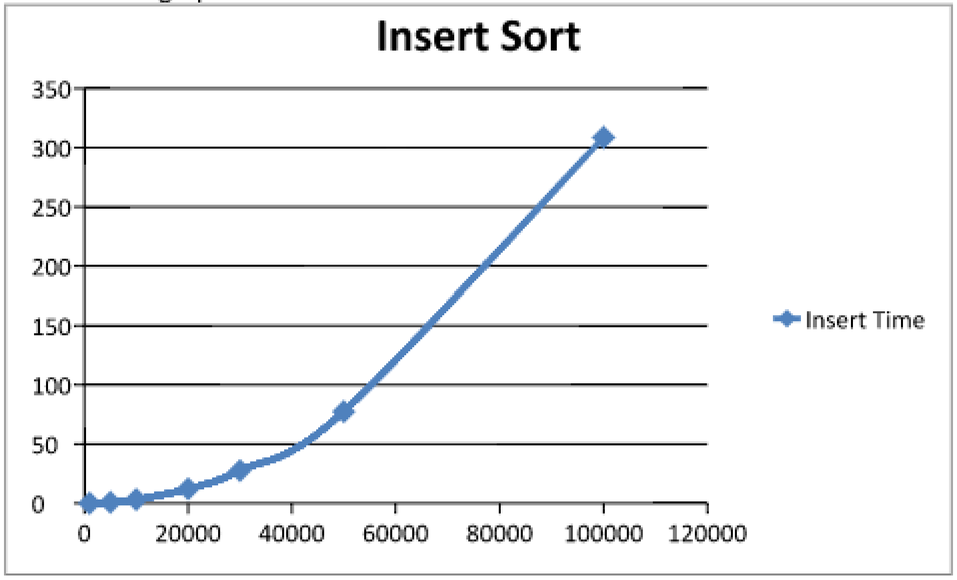
```
import time
from random import randint

#n = int(input("Enter a number for n: "))
n=100000
myList = []

for x in range(n):
    i = randint(1,10000)
    myList.append(i)


#insertSort sorts a list by insertion
#merges two sorted lists into 1 sorted list
def merge(list1, list2):
    sorted = []
    i = 0
    j = 0
    #while both lists have not made it to the end
    while i != len(list1) and j != len(list2):
            # if the element in list 1 is smaller, add it to the sorted list
            if list1[i] < list2[j]:
                    sorted.append(list1[i])
                    i += 1
            #else add the element from list 2 to the sorted list
            else:
                    sorted.append(list2[j])
                    j += 1
    #if some elements remian in list 1 add them all to the sorted list
    if i < len(list1):
            while i < len(list1):
                    sorted.append(list1[i])
                    i += 1
    #if some elements remain in list 2 add them all to the sorted list
    else:
            while j < len(list2):
                    sorted.append(list2[j])
                    j += 1
    return sorted

#mergesort calls itself recursively until there is 1 or fewer items in each list
#it then merges the two lists together.
def mergeSort(myList):
    if len(myList) < 2:
            return myList
    mid = len(myList)/2
    list1 = mergeSort(myList[:mid])
```

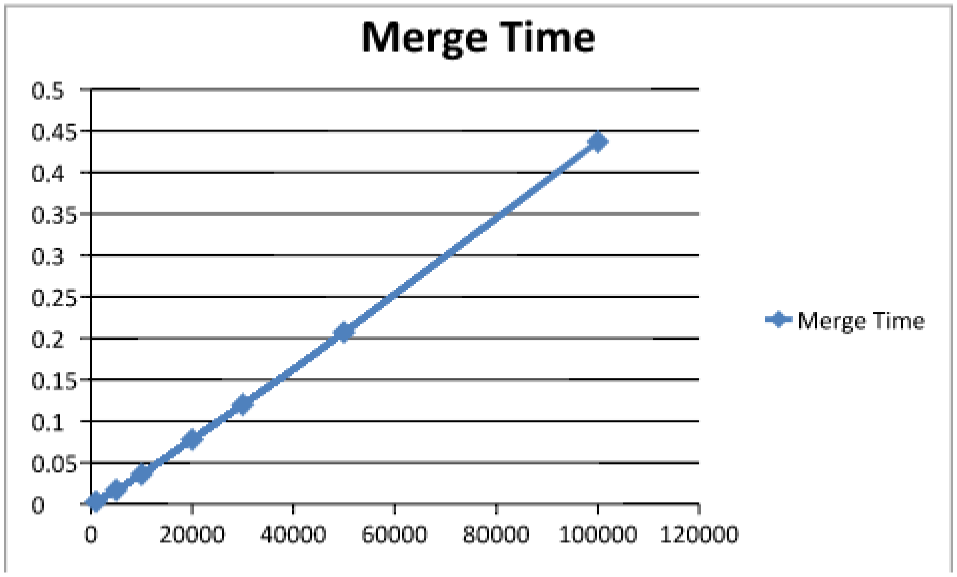
```
        list2 = mergeSort(myList[mid:])
        return merge(list1, list2)


#collects the time it takes to run
start = time.time()
myList = mergeSort(myList)
end = time.time()

print(end - start)
```

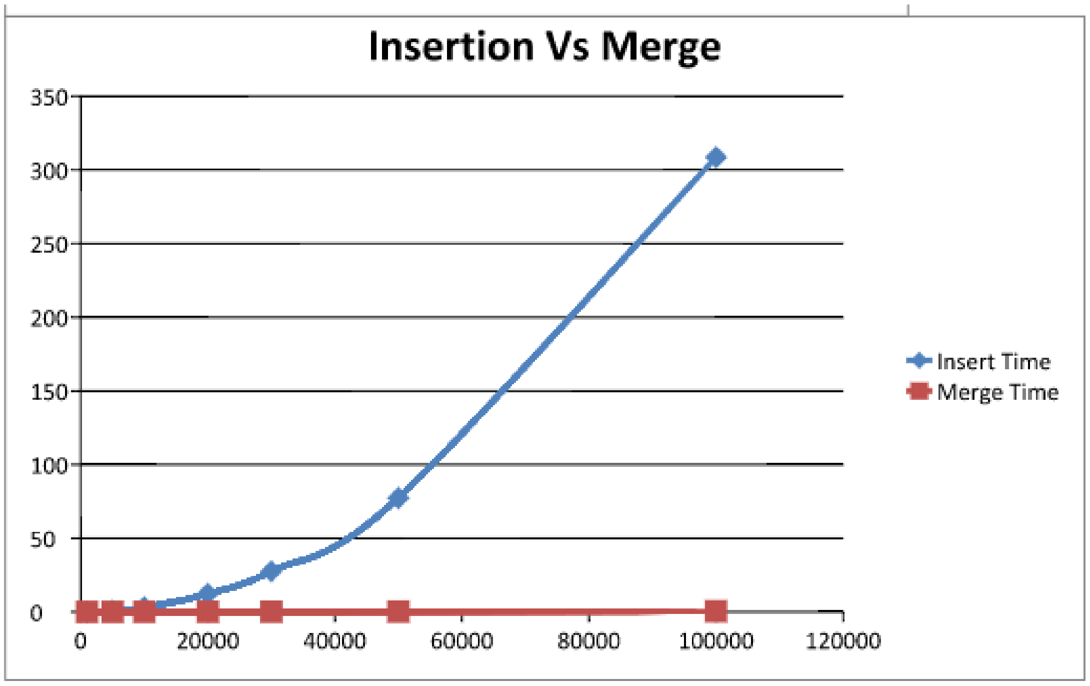
B. Here is a table of the data collected in part B

| n | Insert Time | Merge Time |
|---|---|---|
| 1000 | 0.0329999923706 | 0.00300002098083 |
| 5000 | 0.773000001907 | 0.0169999599457 |
| 10000 | 3.08799982071 | 0.0360000133514 |
| 20000 | 12.2160000801 | 0.0780000686646 |
| 30000 | 27.5920000076 | 0.119999885559 |
| 50000 | 77.3810000042 | 0.207000017166 |
| 100000 | 308.6290000187 | 0.43700003624 |

C. Here are the graphs:





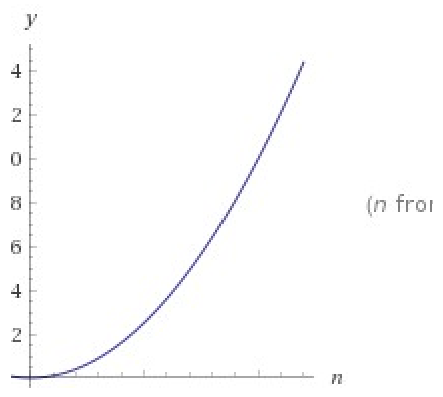


I believe out of all three of these charts that the combined chart gives the best representation of the data. At a glance between the individual charts you wouldn't really notice how much faster merge sort is unless you took the time to look at the values listed on the side of the chart. The combined graph shows the stark difference in run time between the two algorithms. Merge sort is barely on the chart because insertion sort runs so slowly in comparison.

D. For the insertion sort it is easy to see that the curve is polynomial. The next time data point is more than double the one before when the number of n doubles. The curve would probably look more closely like a $n^2$ curve if there were more data points in between.

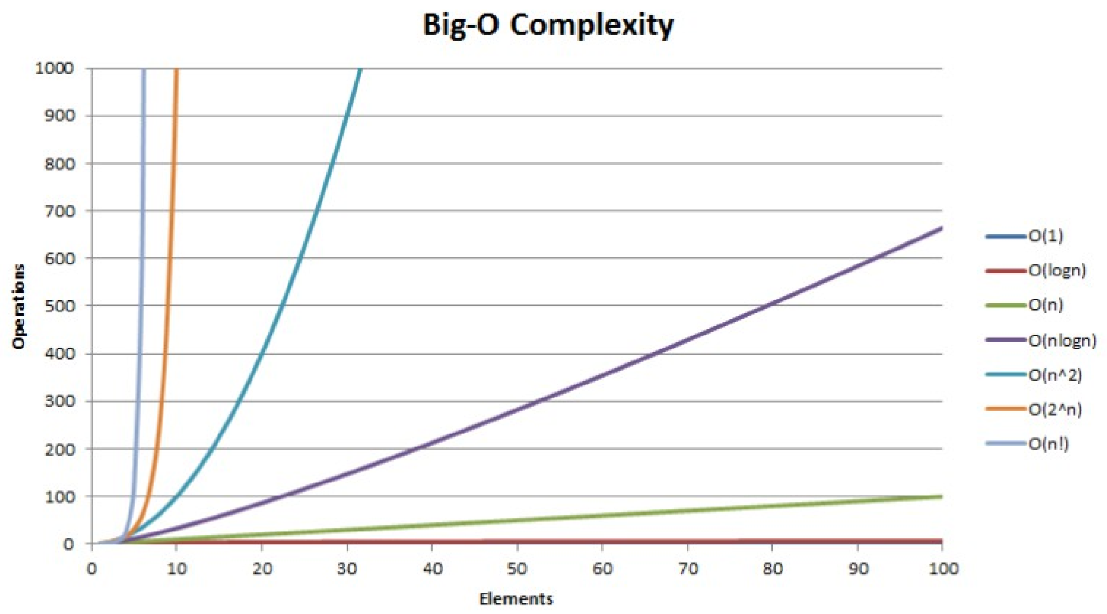
For the merge sort the curve is clearly linier. It is steadily progressing in a straight upward line.
The equation that best fits the insertion sort data would be $y = 3E\text{-}08x^2 + 2E\text{-}06x - 0.0487$
And the equation that best fits the merge sort data would be $y = 4E\text{-}06x - 0.0075$

E. I would say that my experimental running times come fairly close, but not exactly the same as the theoretical running times. Let us start with the insertion sort. The theoretical run time of insertion sort is $O(n^2)$ which would look something roughly like this:



It starts out slow, and then has a very sharp increase in values. My graph is a little "flatter" or straighter than this so it doesn't spike up quite as fast, but it is still very similar. There is a simple explanation for this, however. The theoretical running time is a worst case scenario. My array/list was filled with random integers so it more closely represents an average case scenario. Insertion sort can be anywhere between n and $n^2$ so my data definitely easily represents the average case scenario.

The theoretical running time for Merge Sort is O(n lg n). If we look at this graph of all Big-Oh complexities we can see what the O(n lg n ) graph looks like



This graph looks very linier, but it does have a very slight curve at the start of it. This looks extremely similar to my merge sort graph, so I would say my experimental data for the merge sort gives a very accurate outlook on what the typical merge sort graph would look like. Since the worst case and average case have roughly the same running time, it makes sense that my graph would look so similar.

The combined graph really gives a good view of the difference in growth times, just like it does on the graph above. The insertion sort grows much much faster both theoretically and in my experimental data.

6. EXTRA CREDIT

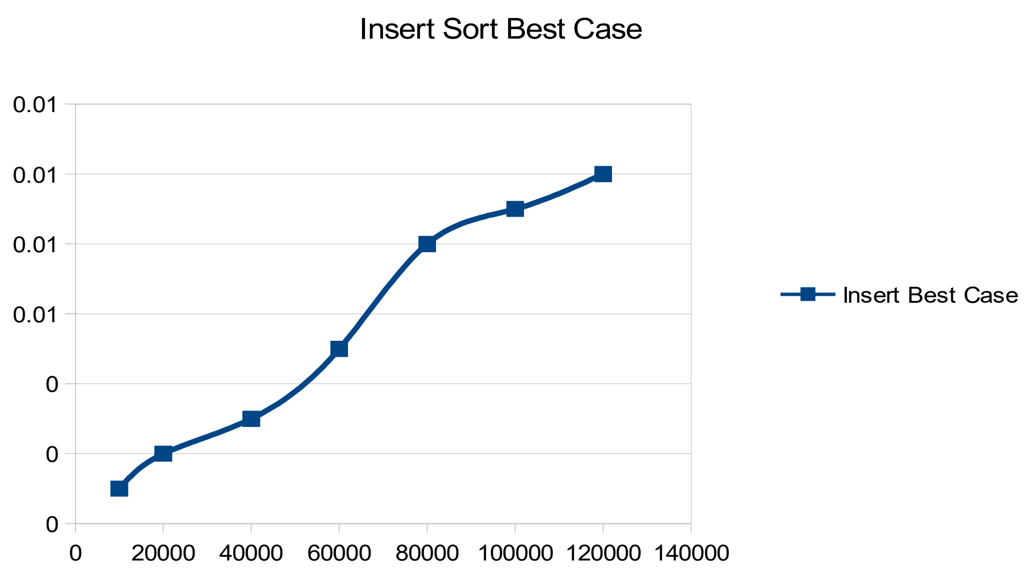
B. I had to massively increase my size of n to get a non 0 run time for the insertion sort, so the amounts of n are rather large for best case scenario.

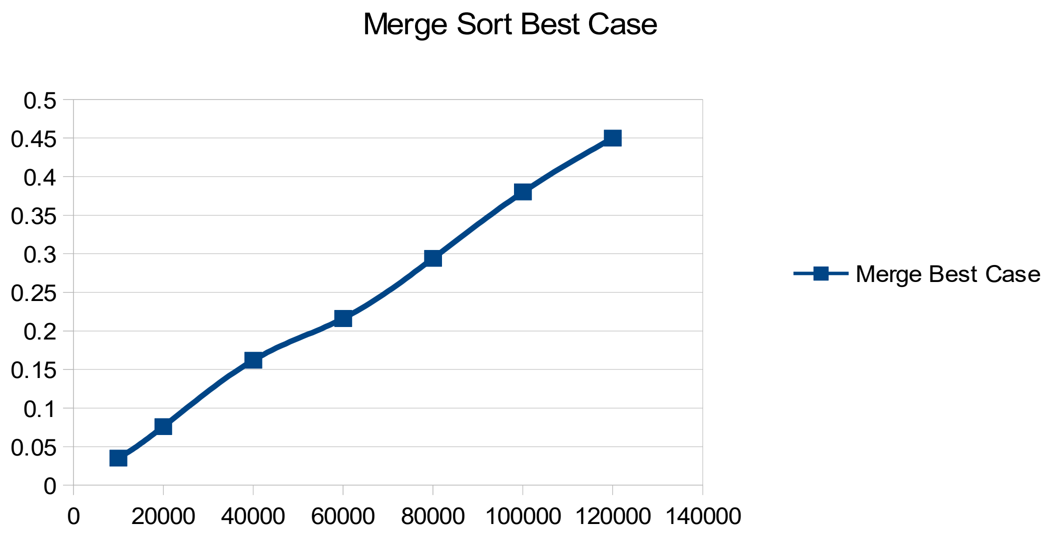
| n | Insert Best Case | Merge Best Case |
|---|---|---|
| 10000 | 0.0010001659 | 0.0349998474 |
| 20000 | 0.001999855 | 0.0759999752 |
| 40000 | 0.003000021 | 0.1620001793 |
| 60000 | 0.0050001144 | 0.2160000801 |
| 80000 | 0.007999897 | 0.2939999104 |
| 100000 | 0.0090000063 | 0.379999876 |
| 120000 | 0.0099999905 | 0.4499980927 |

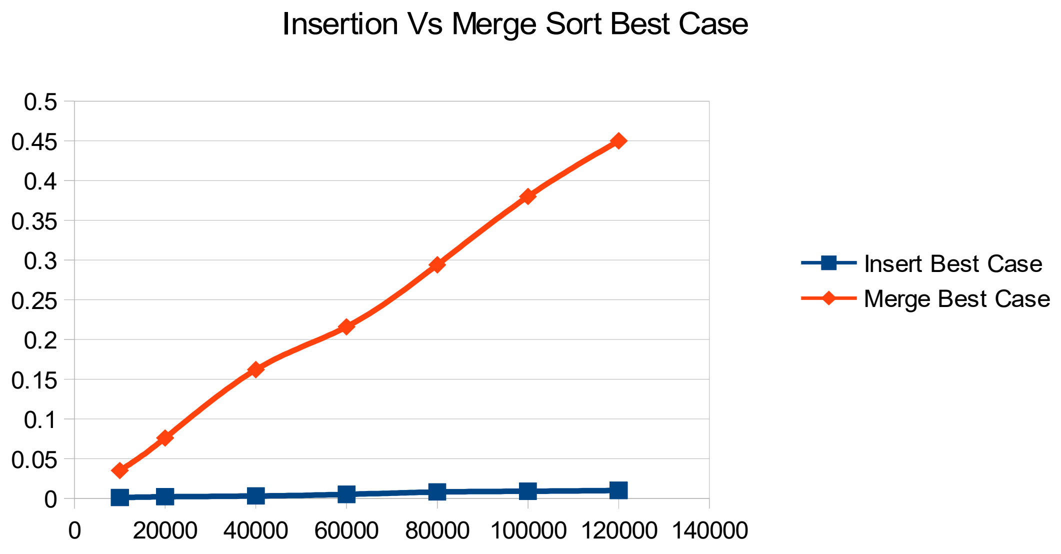
For the worst case scenario I stuck with my original values for n

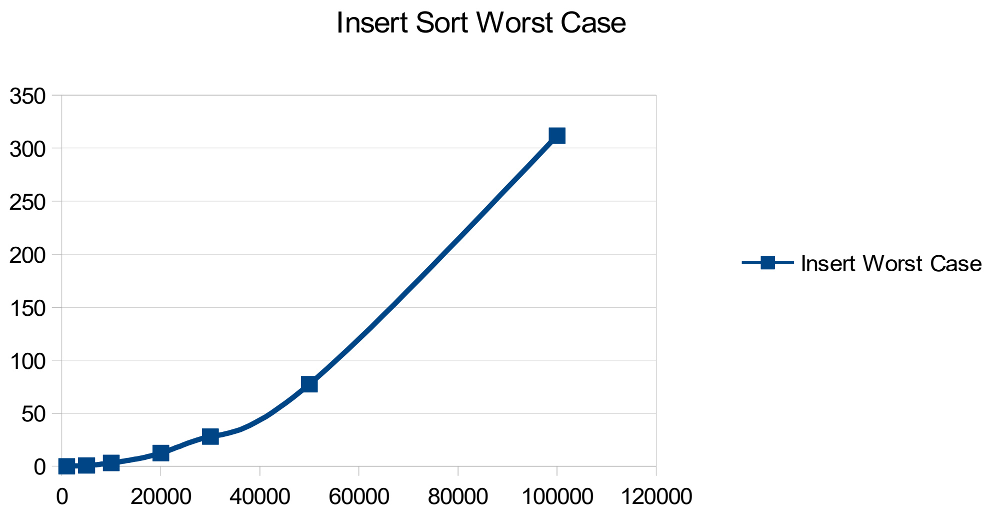
| n | Insert Worst Case | Merge Worse Case |
|---|---|---|
| 1000 | 0.0350000858 | 0.0039999485 |
| 5000 | 0.7829999924 | 0.015999794 |
| 10000 | 3.1260001659 | 0.0329999924 |
| 20000 | 12.4149999619 | 0.0810008965 |
| 30000 | 27.9019999504 | 0.1199998856 |
| 50000 | 77.379999876 | 0.2029998302 |
| 100000 | 311.660000086 | 0.4429998398 |

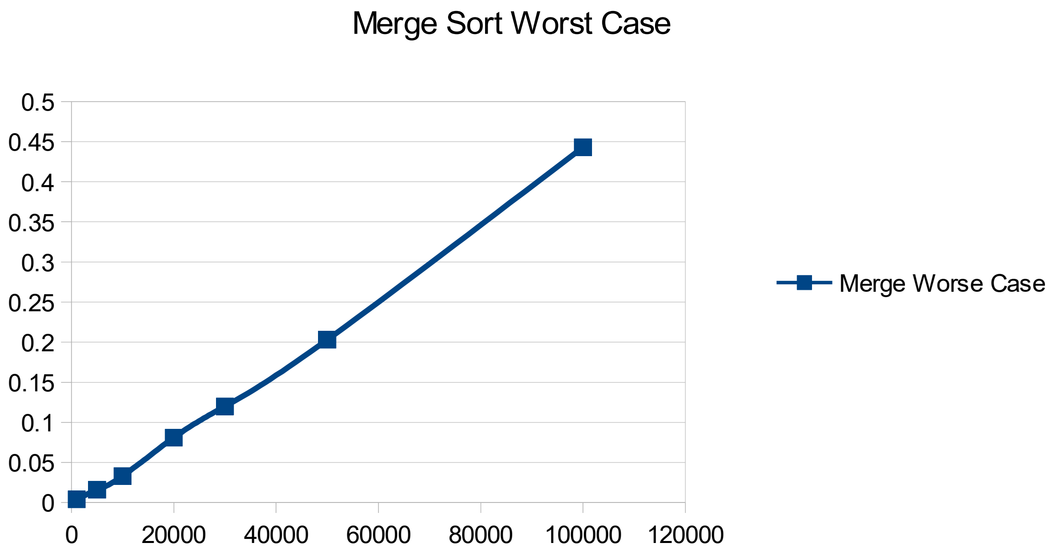
C. Here are the graphs for insert best, inset worst, merge best and merge worse as well as combinations of insert and merge best and inset and merge worst.

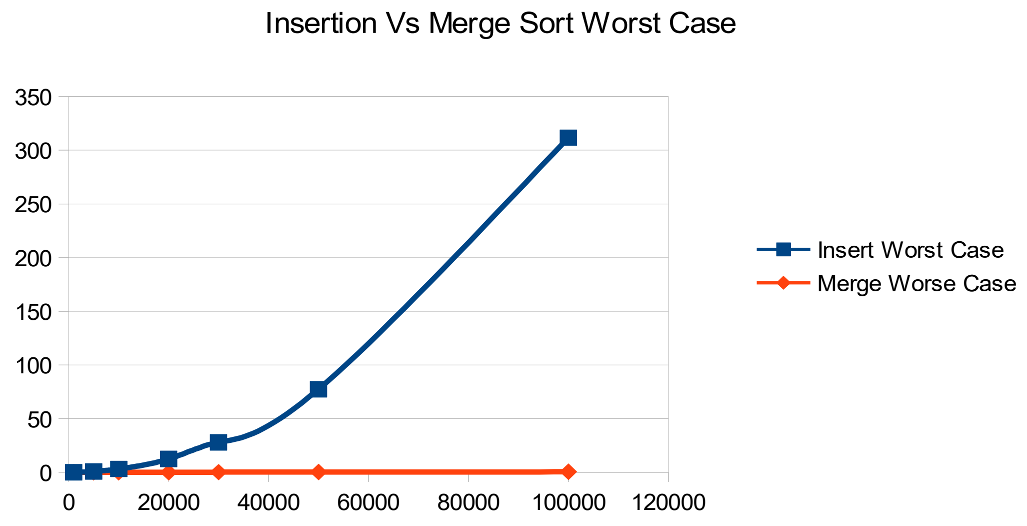



Merge Sort Best Case
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
20000
40000
60000
80000
100000
120000
140000
Merge Best Case


Insertion Vs Merge Sort Best Case
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
20000
40000
60000
80000
100000
120000
140000
Insert Best Case
Merge Best Case


Insert Sort Worst Case
350
300
250
200
150
100
50
0
0
20000
40000
60000
80000
100000
120000
Insert Worst Case


Merge Sort Worst Case
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
20000
40000
60000
80000
100000
120000
Merge Worse Case



Here I believe the best representation of the data is a combination of all of the graphs. The more linier transgression of the best case scenario definitely shows that my way of creating the best case scenario works. When we look at the combined graph of the two we see that the insertion sort is actually faster in best case.  Worse case scenario the combination graph is probably the most telling because it shows even at worse case merge sort doesn't come close to insertion sorts worse case.

D.  The curve that best fits the insertion sort best case would be linier. Yes the plots are a little wonky due to factors like other things running on the computer while the program is running, but the profession is still linier. The equation for this one is

f(x) = 8.6826663003806E-008x + 9.49407245176301E-005

The curve for the best case for merge sort is still linier, with this equation

f(x) = 4.45023966918502E-006x - 0.0087502259

The curve for worst case insertion sort is trending very very closely to n^2 which is exactly what the curve should be at. Here is the equation to show just how close:

f(x) = 3.9538701071594E-008 x^1.9768088335

The curve for worst case merge sort remains ever so linier. Here is the equation:

f(x) = 4.45023966918502E-006x - 0.0087502259

E.  Now let us discuss the results of what is shown above. The results are actually almost spot on with what was expected. The best case for insertion sort was generated by simply creating an already sorted list. This keeps the inner while loop from running which in turn causes a complexity of O(n) instead of O(n^2). Due to not needing to process any of the elements in the list it caused the execution to be extremely fast. Now when we compare it to the best case for merge sort (already sorted in either ascending or declining order) we see that insertion sort best case is much faster. This makes since since the best case complexity of merge sort is still O(n lg n) so it will be about lg n slower than the insertion sort.

Now for the worst case scenarios. The insertion sort worst case scenario was simple. I used the built in reverse function to reverse the already sorted ray since the worse case for insertion sort is an array sorted in descending order. As you can see from the equation of the graph for the worst case, my experimental running times were very very close to perfect. Descending order is the slowest for insertion sort because the algorithm has to go through and move every single element, some more than one space. The worse case for merge sort is a little more complicated. This needed to be an array where essentially the largest and smallest numbers were evenly spaces across sub arrays to create the need to check every single element in all of the sub arrays. I accomplished this recursively by splitting a sorted array into 2 arrays... 1 containing the odd indexes and the other containing the even indexes. When It finally got down to 2 array elements it swapped their positions, leading to a very evenly dispersed array. The run times for this was not much higher than the average case, but this because all cases for merge sort are O(n lg n).