CS 325-400 Fall 17
HW 1
9/23/17

Peter Moldenhauer
moldenhp@oregonstate.edu

1) A Theda(n lg(n)) time algorithm could be described as the following:

   The first step would be to sort the set S using the merge sort algorithm. This weeks lectures discussed how merge sort is a O(n lg(n)) time algorithm. The next step would be then to use the binary search algorithm for each element in set S. Binary search is a O(log(n)) time algorithm. The binary search would check to see if there exists an element in S that equals (x) – (the current element in set S). If such an element does exist in S then the algorithm will return 'true'. It is necessary to conduct merge sort before the binary search because binary search will only work on a sorted list.

   The running time for this algorithm is Theta(n lg(n)) because:

   Merge sort [O(n lg(n))] + Binary search [O(log(n))] = Theta(n lg(n))
   The low order term of log(n) is relatively insignificant and can be disregarded. It should be noted that asymptotically, there is no difference between the two different types of logs with merge sort and binary search. The log base 2 of n is the "same" as log base 10 of n because they only differ by a constant. Log base 2 of n can be written as some constant * log base 10 of n and similarly, log base 10 of n can be written as some constant * log base 2 of n. It should also be noted that the time is Theta(n lg(n)) because it is bound by both O(n lg(n)) and Omega(n lg(n)).

2) I will use the theorem below to solve the following problems:

$$\text{If } lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & then\ f(n)\ is\ O(g(n)) \\ c > 0 & then\ f(n)\ is\ \Theta(g(n)) \\ \infty & then\ f(n)\ is\ \Omega(g(n)) \end{cases}$$

   a. f(n) is O(g(n)). This is because: $\lim_{n\to\infty} n^{0.25}/n^{0.5} = 0$ Since the denominator grows faster, it is dominated by g(n)

   b. f(n) is Theta(g(n)). This is because: $\lim_{n\to\infty} \log n^2/\ln n = 2/\ln(10) = 2/2.3$ Since 2/2.3 is a constant greater than 0, both f(n) and g(n) are growing at the same rate.

   c. f(n) is O(g(n)). This is because: $\lim_{n\to\infty} n\log(n)/\text{nsqrt}(n) = 0$ Since the denominator grows faster, it is dominated by g(n).

d. f(n) is Omega(g(n)). This is because: $\lim_{n\to\infty} 4^n/3^n = \lim_{n\to\infty}(4/3)^n = \infty$ Since the numerator grows faster, it is dominated by 4^n

e. f(n) is Theta(g(n)). This is because: $\lim_{n\to\infty} 2^n/2^{n+1} = ½$ Since ½ is a constant greater than 0, both f(n) and g(n) are growing at the same rate.

f. f(n) is O(g(n)). This is because: $\lim_{n\to\infty} 2^n/n! = 0$ Since the denominator grows faster, it is dominated by n!

3)

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

To prove, we need to assume that the first part { $f_1(n) = O(g(n))$ and $f_2(n) = O(g(n))$ } is true and will show that the second part { $f_1(n) + f_2(n) = O(g(n))$ } is also true. To disprove, we need to give one counter example.

If we assume the first part is true, then by definition, there are positive constants c_1, c_2, n_1, and n_2 such that:
$f_1(n) <= c_1g(n)$, for all $n >= n_1$ and
$f_2(n) <= c_2g(n)$, for all $n >= n_2$

To prove this we need to show that there is some constant c_0 that causes $f_1(n) + f_2(n) <= c_0g(n)$ for $n > 0$ integers.

Proof: We can then let $c_0 = c_1 + c_2$ and let $n_0 = \max(n_1, n_2)$. Then we can say that:
$f_1(n) + f_2(n) <= c_0g(n)$ for all $n >= n_0$

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

To prove, we need to assume that the first part { $f(n) = O(g_1(n))$ and $f(n) = O(g_2(n))$ } is true and will show that the second part { $g_1(n) = \text{Theta}(g_2(n))$ } is also true. To disprove, we need to give one counter example.

If we assume the first part is true, then by definition, there exists positive constants c_1, c_2, n_1, and n_2 such that:
$f(n) <= c_1g_1(n)$, for all $n >= n_1$ and
$f(n) <= c_2g_2(n)$, for all $n >= n_2$

To prove this we need to show that there exists positive constants c_3, c_4, and n_3 such that: $c_3g_2(n) <= g_1(n) <= c_4g_2(n)$ for all $n >= n_3$

However, its rather hard to find a relationship between $g_1(n)$ and $g_2(n)$ to prove this conjecture. They are both just greater than f(n) but there is no relationship between them directly. So instead of proving this conjecture, we

will disprove it. To disprove the conjecture we will use the following counter example:

Let $f(n) = n$, $g_1(n) = n^2$, and $g_2(n) = n^3$

Then it still holds true that $f(n) = O(g_1(n))$ and $f(n) = O(g_2(n))$. However, in this case $g_1(n)$ does NOT equal $Theta(g_2(n))$. This is because it is NOT the case that $n^2 <= n^3 <= n^2$. We can see that $g_1(n)$ and $g_2(n)$ grow at different rates and therefore are not big O and Omega of each other.

4) [Implementation of merge sort and insertion sort submitted separately]

5)

a. "Text" copy of modified mergesort.py and insertsort.py below:

```python
#used for random number generation
import random

# used to get programs execution time
import time
startTime = time.time()

# mergeSort function
def mergesort(theArray):
    # if the array has only one element just return the array
    if len(theArray) == 1:
        return theArray
    # otherwise recursively call mergeSort on both parts of theArray
    else:
        # get the midpoint of theArray
        midpoint = len(theArray)/2
        firstHalf = mergesort(theArray[:midpoint])
        secondHalf = mergesort(theArray[midpoint:])
        return merge(firstHalf, secondHalf)

# merge helper function for mergeSort
def merge(firstHalf, secondHalf):
    # empty array to hold sorted values
    sortedArray = []
    # add to sortedArray until either firstHalf or secondHalf of the array is
empty
    while len(firstHalf) != 0 and len(secondHalf) != 0:
        if firstHalf[0] < secondHalf[0]:
            sortedArray.append(firstHalf[0])
            firstHalf.remove(firstHalf[0])
        else:
            sortedArray.append(secondHalf[0])
            secondHalf.remove(secondHalf[0])
    # add the remaining value to sortedArray
    if len(firstHalf) == 0:
        sortedArray += secondHalf
    else:
        sortedArray += firstHalf
    return sortedArray

# generate an array of size n containing random integer values from 0 to 10000
# note: I will change the value of n each time I execute the program, n = 1000
in this example
n = 100000

# create the array of random integer values
theArray = []
for i in range(n):
    theArray.append(random.randrange(0, 10001, 1))

# call the mergeSort function to sort the array
sortedArray = mergesort(theArray)

# print the programs execution time
print("--- %s seconds ---" % (time.time() - startTime))
```

```
#used for random number generation
import random

# used to get programs execution time
import time
startTime = time.time()

# insertionSort function
def insertSort(theArray):
    # since the 1st element is already sorted, start looping at 2nd element to
the last element
    for i in range(1, len(theArray)):
        # key is the next item to be inserted into the leading sorted section
of the array
        key = theArray[i]
        # lastItem is the last item to compare to
        lastItem = i - 1
        while lastItem >= 0 and theArray[lastItem] > key:
            theArray[lastItem + 1] = theArray[lastItem]
            lastItem = lastItem - 1
        theArray[lastItem + 1] = key
    return theArray

# generate an array of size n containing random integer values from 0 to 10000
# note: I will change the value of n each time I execute the program, n = 1000
in this example
n = 50000

# create the array of random integer values
theArray = []
for i in range(n):
    theArray.append(random.randrange(0, 10001, 1))

# call the insertSort function to sort the array
sortedArray = insertSort(theArray)

# print the programs execution time
print("--- %s seconds ---" % (time.time() - startTime))
```

b. Running times of each algorithm for the given n value:

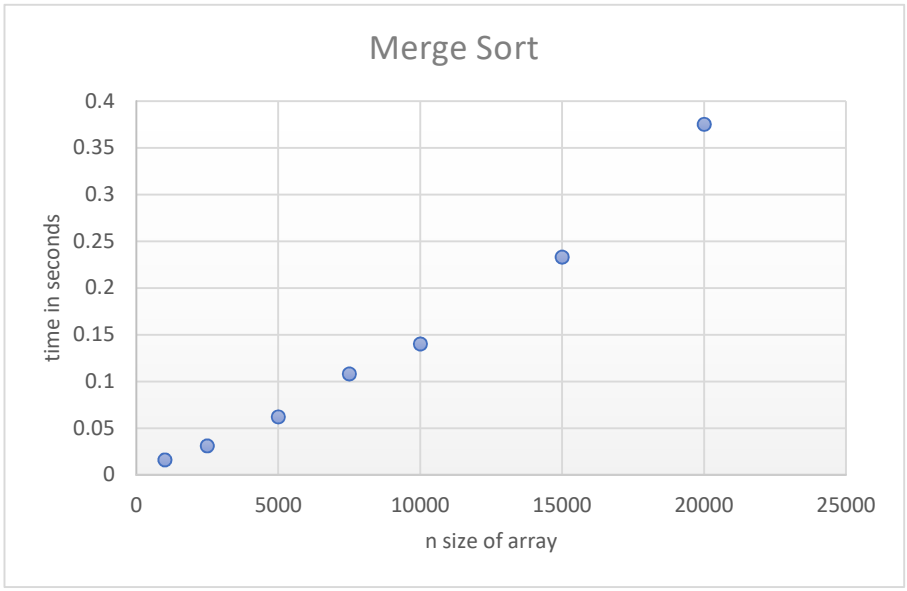
Merge sort:

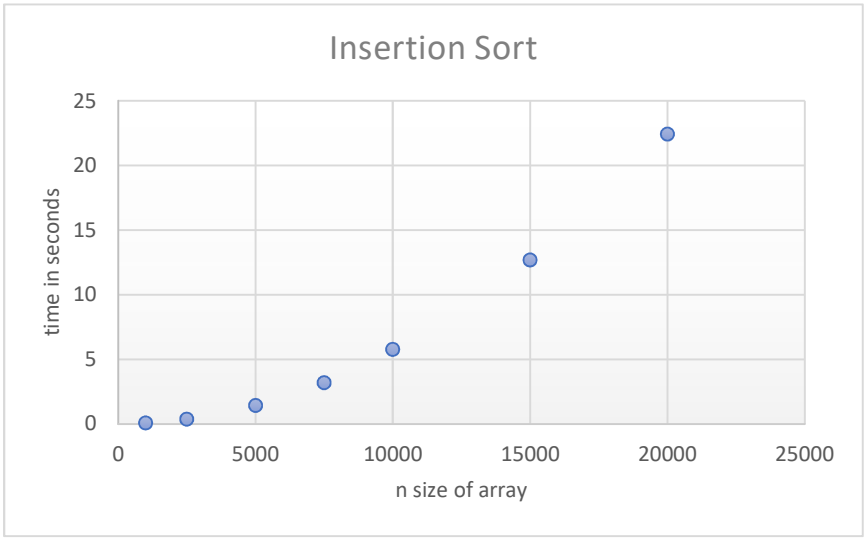
n value	t (time in seconds)
1000	.016
2500	.031
5000	.062
7500	.108
10000	.140
15000	.233
20000	.375

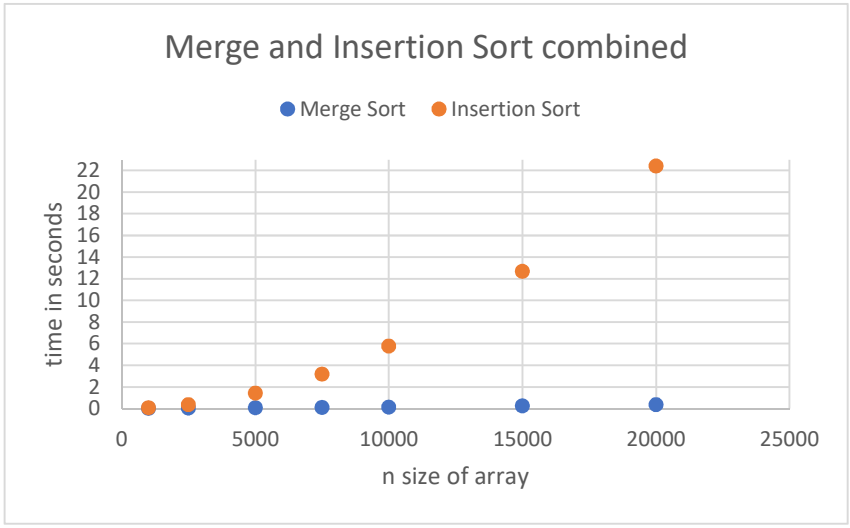
Insertion sort:

n value	t (time in seconds)
1000	.062
2500	.357
5000	1.43
7500	3.19
10000	5.77
15000	12.69
20000	22.42

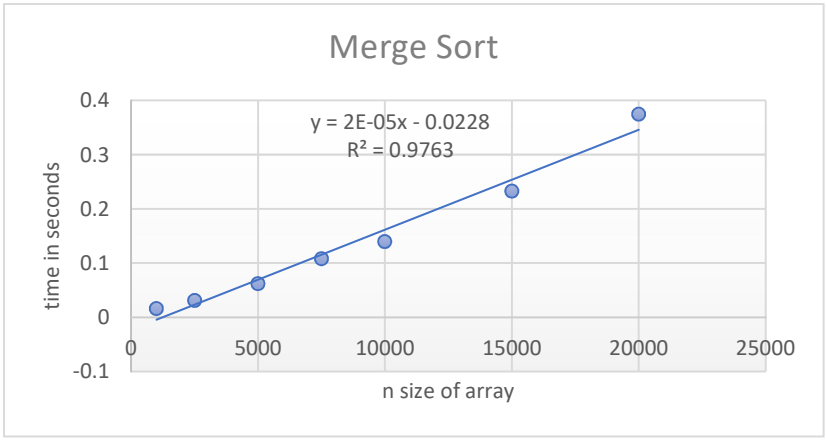
c. Below, I have plotted the running time data that I collected for each of the algorithms. I have plotted the data for merge sort and insertion sort on their own separate graphs as well as one graph that contains the data from both algorithms on the same graph. The individual graphs represent the data best. This is because there is such a dramatic difference in the running times of the two algorithms which makes the combined graph difficult to read and represent the data properly. As long as you are aware of the different increments of the time unites in each graph, the individual graphs for each algorithm represents the data and growth rate more accurately. However, if you are simply looking at how the two graphs differ from each other then the combined graph is best to look at.

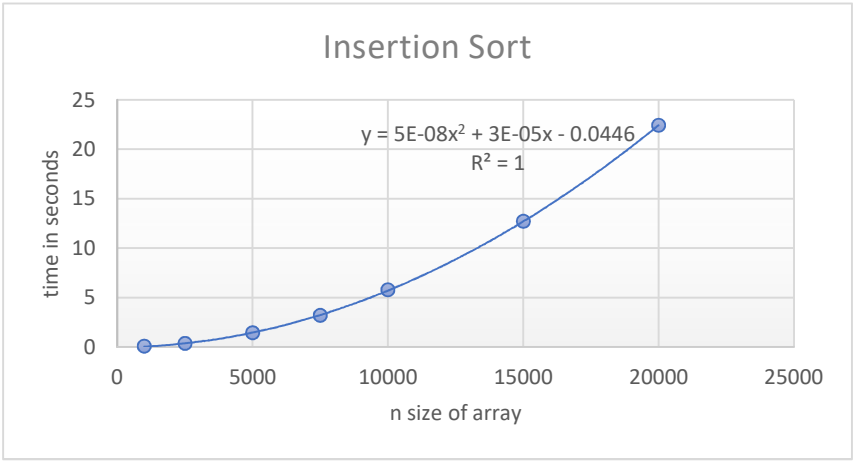


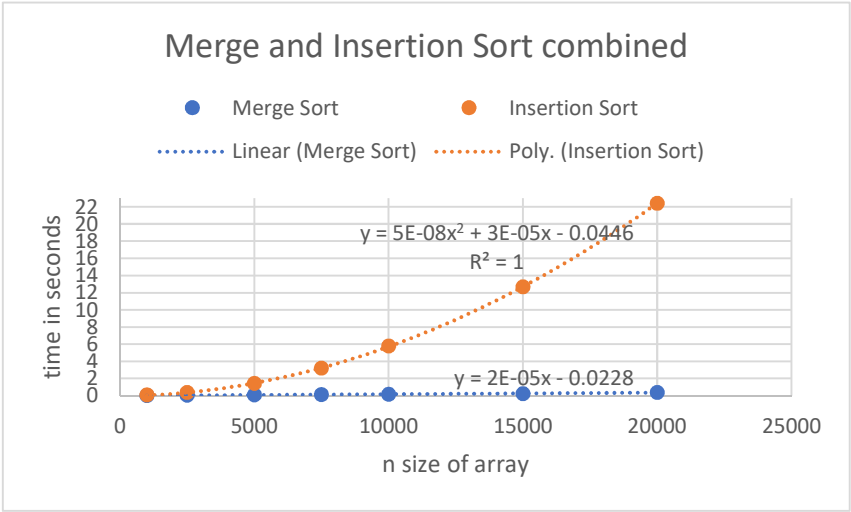




d. Below are the same graphs as above but with the curves added that best fits each data set. The linear curve fits the merge sort data set best with an R^2 of 0.9763. However, even though merge sort is technically a n log n algorithm, the log n is so small when it is graphed so the values look linear. Also, I graphed these in Microsoft Excel and Excel does not offer a n log n curve. With what Excel, offers, the linear curve was best for merge sort. The equation that Excel gives that best fits the merge sort data is: y = 2E-05x - 0.0228 The approx. equation that I would give is f(n) = n. For the insertion sort data set, the polynomial (2^{nd} degree) curve fits best with an R^2 of 1. The equation that Excel gives that best fits the insertion sort data is: $y = 5E\text{-}08x^2 + 3E\text{-}05x - 0.0446$ The approx.. equation that I would give for insertion sort is $f(n) = n^2$


Merge Sort
y = 2E-05x - 0.0228
R² = 0.9763
time in seconds
n size of array
0.4
0.3
0.2
0.1
0
-0.1
0
5000
10000
15000
20000
25000


Insertion Sort
y = 5E-08x² + 3E-05x - 0.0446
R² = 1
time in seconds
n size of array
25
20
15
10
5
0
0
5000
10000
15000
20000
25000


Merge and Insertion Sort combined
Merge Sort
Insertion Sort
Linear (Merge Sort)
Poly. (Insertion Sort)
y = 5E-08x² + 3E-05x - 0.0446
R² = 1
y = 2E-05x - 0.0228
time in seconds
n size of array
22
20
18
16
14
12
10
8
6
4
2
0
0
5000
10000
15000
20000
25000

e. The experimental running times that were generated from my algorithms are fairly similar to the theoretical running times of the algorithms. With merge sort, the theoretical running time of this algorithm is O(n (log n). My data however resulted in more of a linear O(n) running time. This is still fairly close to the theoretical running time for this algorithm. As noted earlier, even though merge sort is technically a O(n log n) algorithm, the log n is so small when it is graphed so the values I plotted look linear. In theory however, the best, average and worst cases for merge sort are all O(n log n). Even though all of my input numbers were random numbers, this made little difference in where my running time compared to the theoretical best, average and worst case running times for merge sort. If I continued to test my algorithm with much larger values of "n", then I am sure my linear O(n) running time will start to appear more like the actual O(n log n) running time for merge sort. With insertion sort, the theoretical running time of this algorithm is $O(n^2)$. This is almost exactly what my data plotted to. Even though the "best case" running time for insertion sort is O(n), to get this running time the data would have to already be in sorted order. Since my input numbers were all random (and not already in sorted order), my running time was closer to the average and worst case theoretical running time of $O(n^2)$. If I had more control over the values that my insertion sort ran on (other than random values), then I am sure that my algorithm will be closer to the best case running time of O(n).

Extra Credit:

Merge sort in general does not have a "worst case" performance. Merge sort has a best, average and worst case of all O(n log n). However, just like any algorithm, for merge sort the worst case scenario would be to do the maximum number of comparisons as possible. Wikipedia states that "n the worst case, the number of comparisons merge sort makes is equal to or slightly smaller than ($n \lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1$), which is between ($n \lg n - n + 1$) and ($n \lg n + n + O(\lg n)$)" [https://en.wikipedia.org/wiki/Merge_sort] So is there a way that we can make merge sort actually result in a "worst case" by modifying the input values? Lets suppose that the array in the final step after sorting is [0, 1, 2, 3, 4, 5, 6, 7]. To result in the worst case, it would make the most sense that the array before this final step to be [0, 2, 4, 6, 1, 3, 5, 7]. This is because the left subarray [0, 2, 4, 6] and the right subarray [1, 3, 5, 7] would result in the maximum number of comparisons. So in theory, the worst case for merge sort would be when during ever merge step, exactly one value remains in the opposing list and no comparisons were skipped.

With similar reasoning as above, it could be argued that merge sort also has a "best case". This would be when the largest element of a given subarray is smaller than the fist element of the opposing subarray. For the following merge steps that then take place, only one element from the opposing list is compared.

With this being said, I modified my code to generate arrays of this nature (for both the worst and best cases) and then inputted the data into my merge sort algorithm. Since I already know that merge sort's performance is O(n log n) for best, average and worst case scenarios, I was expecting for my running times for both cases (worst/best) to still reflect this. This is exactly what my end result was. Giving my merge sort algorithm the best case input and worst case input still resulted in roughly O(n). Note: Excel does not give a n log n curve line. The linear O(n) fit best I think the log n would still be too small to even show up. time. However the best case input was slightly faster than the worse case input. (See the charts below) The equation for merge sort best case was: y = 6E-06x - 0.0173 and the equation for the worst case was: y = 1E-05x - 0.0391

Insertion sort in general has a worst case of $O(n^2)$ and a best case of O(n). To result in the worst case for insertion sort, the algorithm would need to be given input in reverse sorted order. In this case, each element is the smallest or second smallest of the elements before it. This will cause the algorithm to iterate over all of the elements as many times as possible. To make my insertion sort algorithm reach the worst case, I did this very thing. I gave it an array of integers that were sorted in reverse order. The experimental running times that I recorded were very close to the theoretical worst case of $O(n^2)$. This was what I was expecting.

The theoretical best case of insertion sort is O(n). To achieve this, I needed to give my insertion sort algorithm input data that is already sorted. This results in linear time of O(n) because with each iteration, the first element is only compared to the right most element of the sorted subarray. As expected, my experimental running time was very close to the theoretical best case of O(n). (See the charts below) The equation for insertion sort best case was: y = 2E-07x - 0.0001 and the equation for the worst case was: $y = 6E\text{-}08x^2 + 2E\text{-}06x + 0.0027$

Merge Sort – best case

Input (n)	Time in seconds
1000	.0031
2000	.0071
4000	.0136
8000	.0300
16000	.0643
32000	.1567
64000	.4035

Merge Sort – worst case

Input (n)	Time in seconds
1000	.0044
2000	.0096
4000	.0218
8000	.0496
16000	.1179
32000	.2812

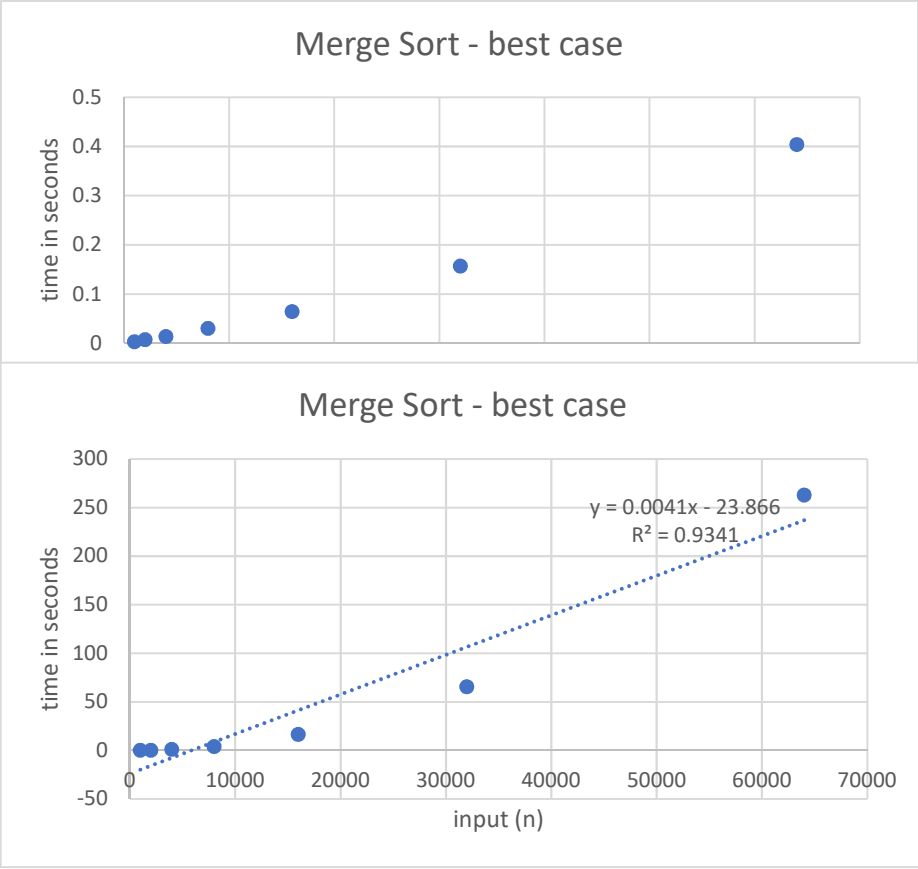
64000	.7736

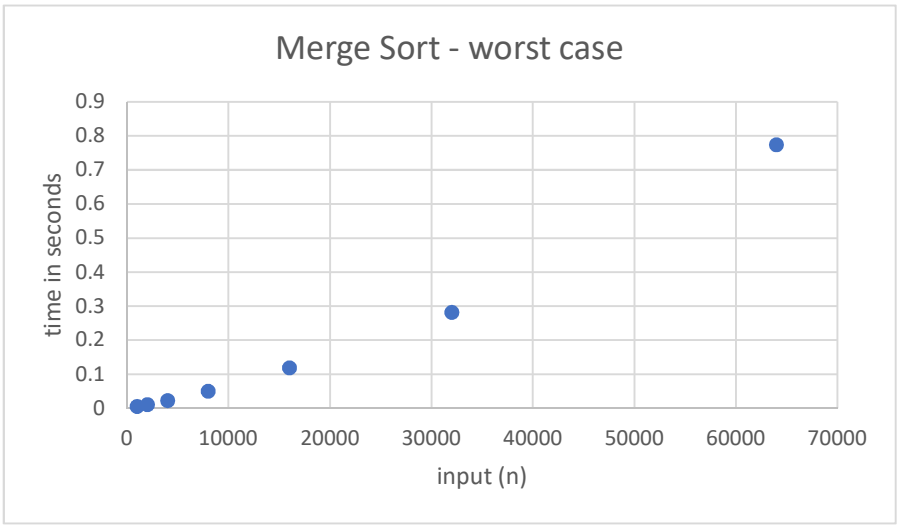
Insertion Sort – best case

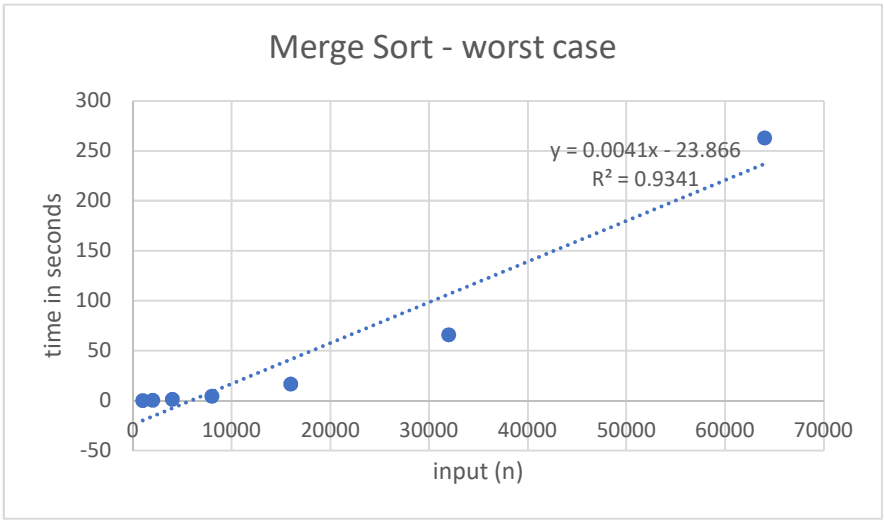
Input (n)	Time in seconds
1000	.0001
2000	.0002
4000	.0005
8000	.0012
16000	.0023
32000	.0053
64000	.0101

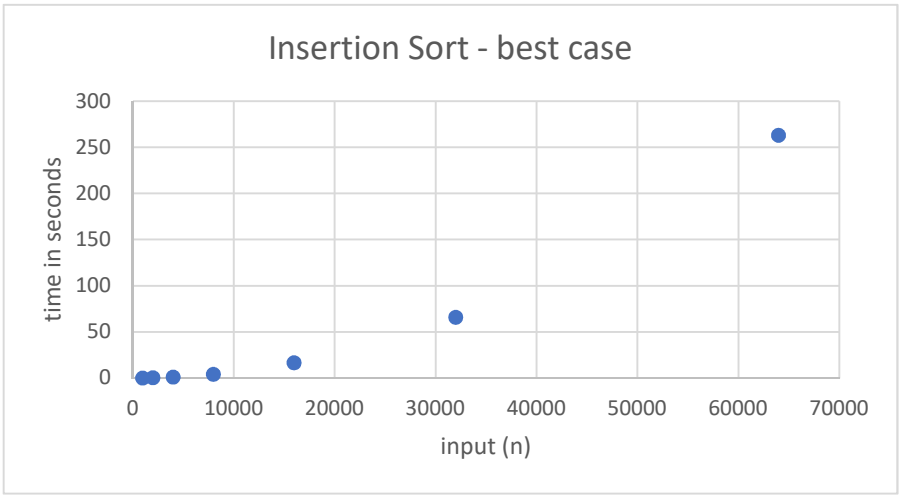
Insertion Sort – worst case

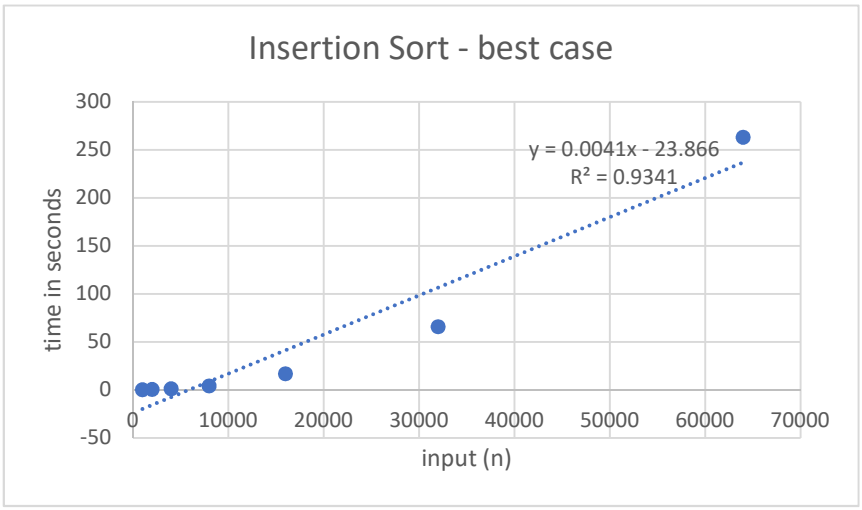
Input (n)	Time in seconds
1000	.0658
2000	.2593
4000	1.0244
8000	4.1372
16000	16.483
32000	65.715
64000	262.80

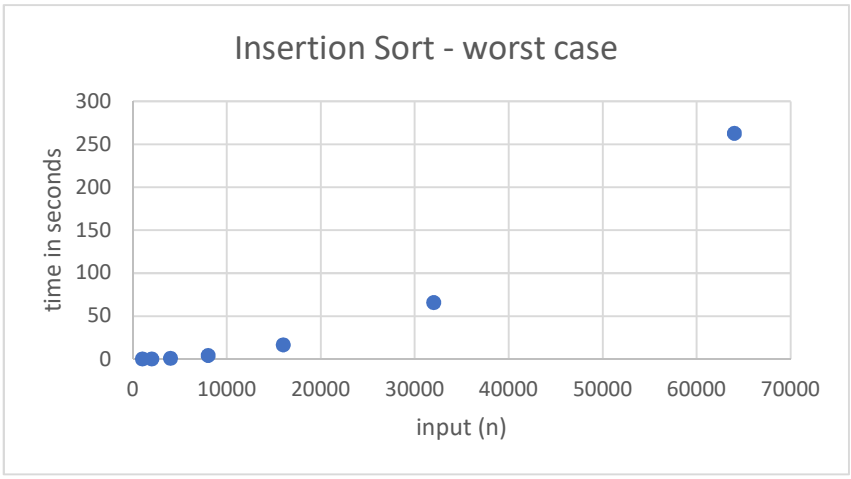


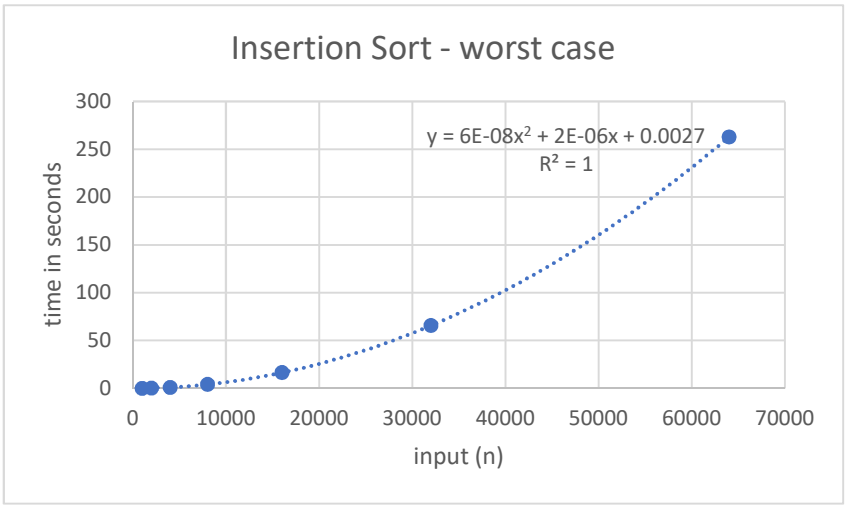

Merge Sort - worst case
time in seconds
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
10000
20000
30000
40000
50000
60000
70000
input (n)


Merge Sort - worst case
time in seconds
300
250
200
150
100
50
0
-50
y = 0.0041x - 23.866
R² = 0.9341
0
10000
20000
30000
40000
50000
60000
70000
input (n)


Insertion Sort - best case
time in seconds
300
250
200
150
100
50
0
0
10000
20000
30000
40000
50000
60000
70000
input (n)


Insertion Sort - best case
y = 0.0041x - 23.866
R² = 0.9341
time in seconds
input (n)
300
250
200
150
100
50
0
-50
0
10000
20000
30000
40000
50000
60000
70000


Insertion Sort - worst case
time in seconds
input (n)
300
250
200
150
100
50
0
0
10000
20000
30000
40000
50000
60000
70000


Insertion Sort - worst case
y = 6E-08x² + 2E-06x + 0.0027
R² = 1
time in seconds
input (n)
300
250
200
150
100
50
0
0
10000
20000
30000
40000
50000
60000
70000