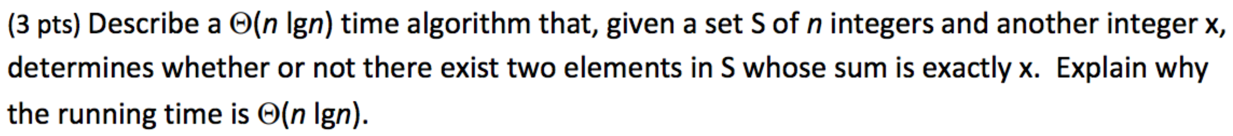
Class: CS-325
Term: Fall 2017
Author: Jon-Eric Cook
Date: September 25, 2017
Homework: #1

1)

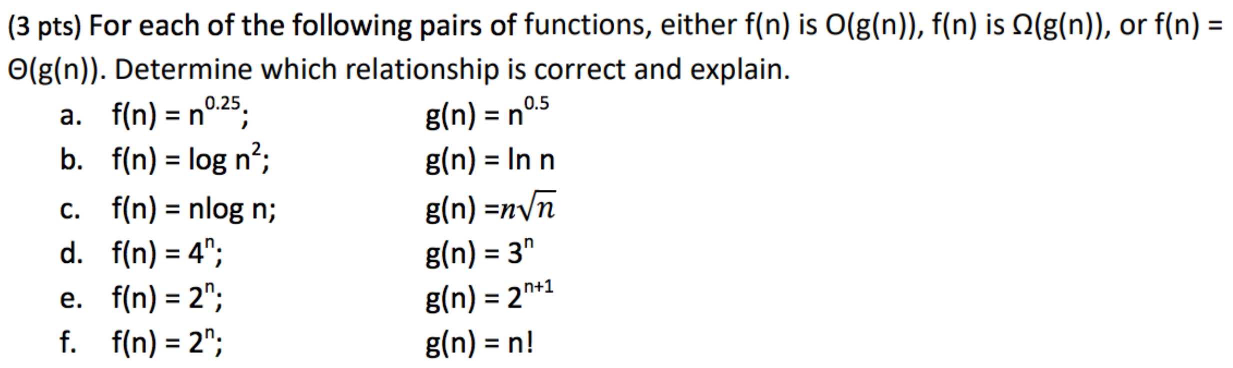


(3 pts) Describe a $\Theta(n \lg n)$ time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x. Explain why the running time is $\Theta(n \lg n)$.

Answer:

Given a set S of n integers and another integer x, a theta(n lgn) algorithm that would determine whether or not there existed two elements in S whose sum would exactly be x would be one that is made up of a Merge Sort and a Binary Search. The Merge Sort would be used to first sort the array S of n integers. Once sorted, each element in the array, S[i], would be selected and then a Binary Search would be done to check if the integer (x - S[i]) existed in the array. The running time is theta(n lgn) because Merge Sort is O(n lgn) and this particular use of the Binary Search is O(n lgn) as well. In total, the running time is O(n lgn) + O(n lgn) but resulting coefficient of 2 is simply ignored. Thusly, the running time is theta(n lgn).

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. $f(n) = n^{0.25}$; $g(n) = n^{0.5}$
b. $f(n) = \log n^2$; $g(n) = \ln n$
c. $f(n) = n\log n$; $g(n) = n\sqrt{n}$
d. $f(n) = 4^n$; $g(n) = 3^n$
e. $f(n) = 2^n$; $g(n) = 2^{n+1}$
f. $f(n) = 2^n$; $g(n) = n!$

Answer:

a. f(n) is O(g(n)) because the limit as n goes to infinity of f(n) / g(n) is 0.
b. f(n) is theta(g(n)) because the limit as n goes to infinity of f(n) / g(n) is 2.
c. f(n) is O(g(n)) because the limit as n goes to infinity of f(n) / g(n) is 0.
d. f(n) is omega(g(n)) because the limit as n goes to infinity of f(n) / g(n) is infinity.
e. f(n) is theta(g(n)) because the limit as n goes to infinity of f(n) / g(n) is 1/2.
f. f(n) is O(g(n)) because the limit as n goes to infinity of f(n) / g(n) 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 $f_1(n)+f_2(n) = O(g(n))$.

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

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

Answer:
a)
Prove:
If f1(n) = O(g(n)) then there exists constants c and n1 such that
0 < f1(n) <= cg(n) for all n >= n1

and since f2(n) = O(g(n)) there exists constants c and n2 such that
0 < f2(n) <= cg(n) for all n >= n2

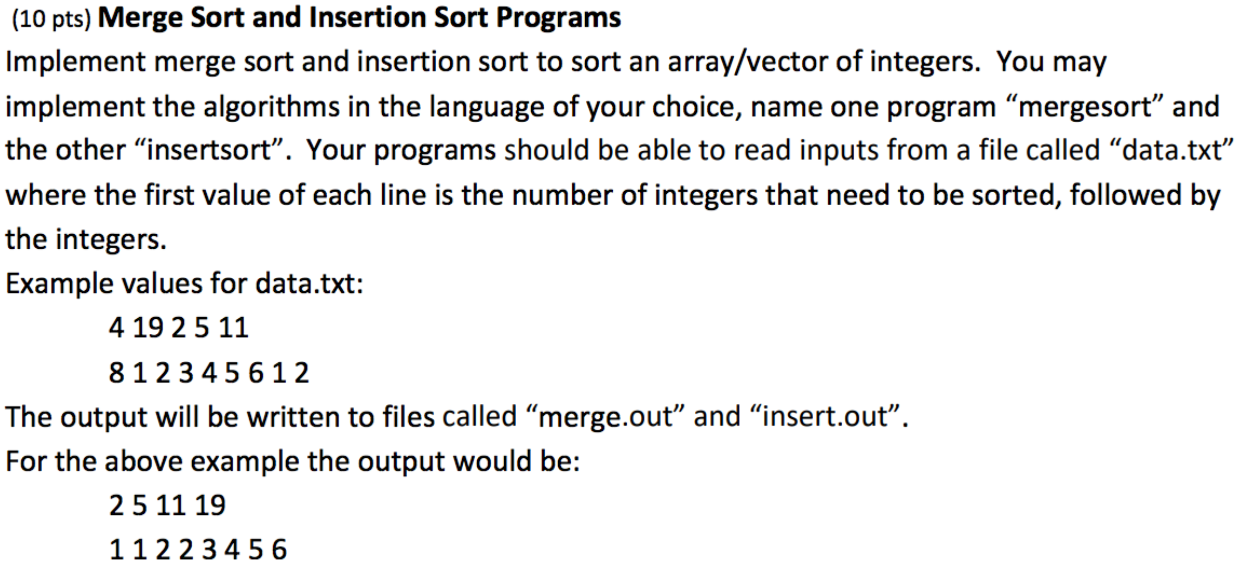
Let g = g(n)
f1(n) <= g
f2(n) <= g
For n >= (n1+n2)

0 < f1 + f2 <= g + g
0 < f1 + f2 <= 2g
Let c = 2
For n >= (n1 + n2)

Therefore
f1(n) + f2(n) <= cg(n)
or
f1(n) + f2(n) = O(g(n))

b)
Disprove:
Let f(n) = n, g1(n) = n^2 and g2(n) = n^3
The limit as n goes to infinity of f(n) / g1(n) is equal to n / n^2. This evaluates to 0 and the above resource leads to the conclusion that f(n) is O(g1(n)). This checks out. The limit as n goes to infinity of f(n) / g2(n) is equal to n / n^3. This evaluates to 0 and the above resource leads to the conclusion that f(n) is O(g2(n)). This checks out. The limit as n goes to infinity of g1(n) / g2(n) is equal to n^2 / n^3. This evaluates to 0 and the above resource leads to the conclusion that g1(n) is O(g2(n)). This disproves the above statement that g1(n) = theta(g2(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:

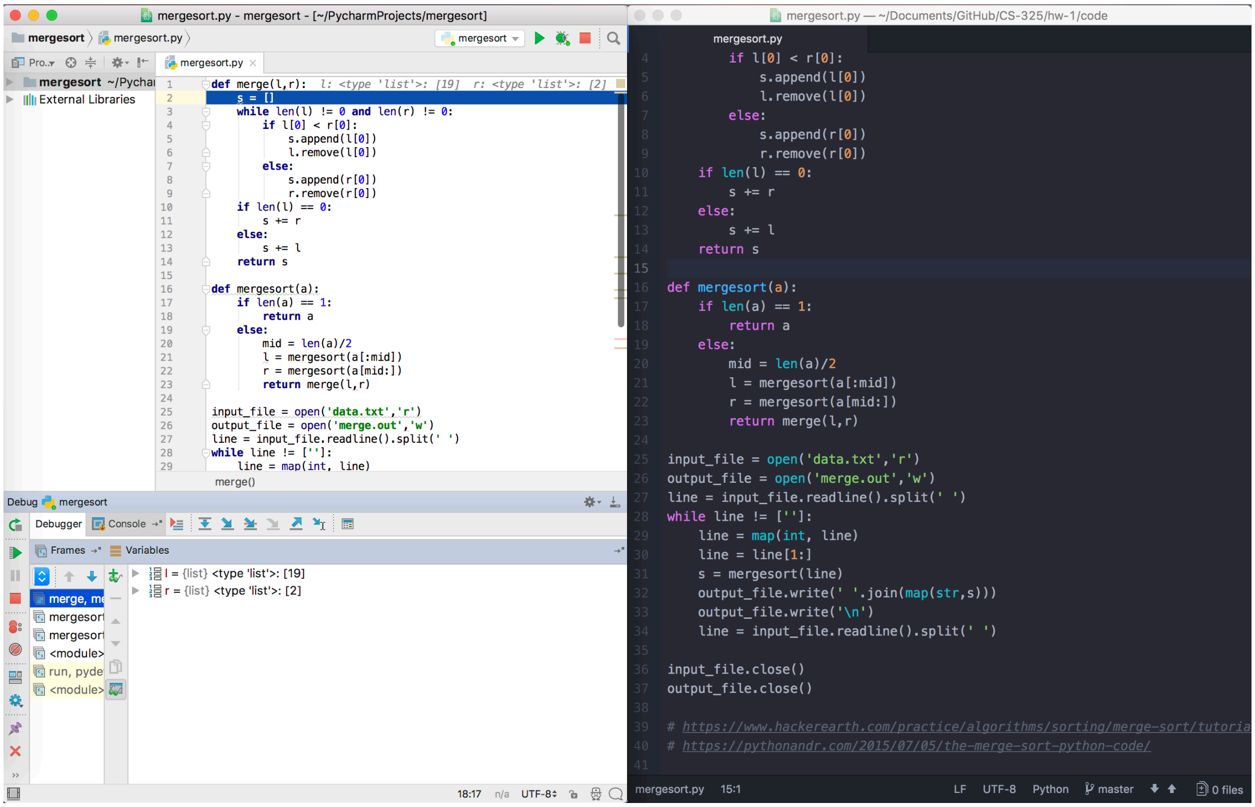
```
4 19 2 5 11
8 1 2 3 4 5 6 1 2
```

The output will be written to files called “merge.out” and “insert.out”.

For the above example the output would be:

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

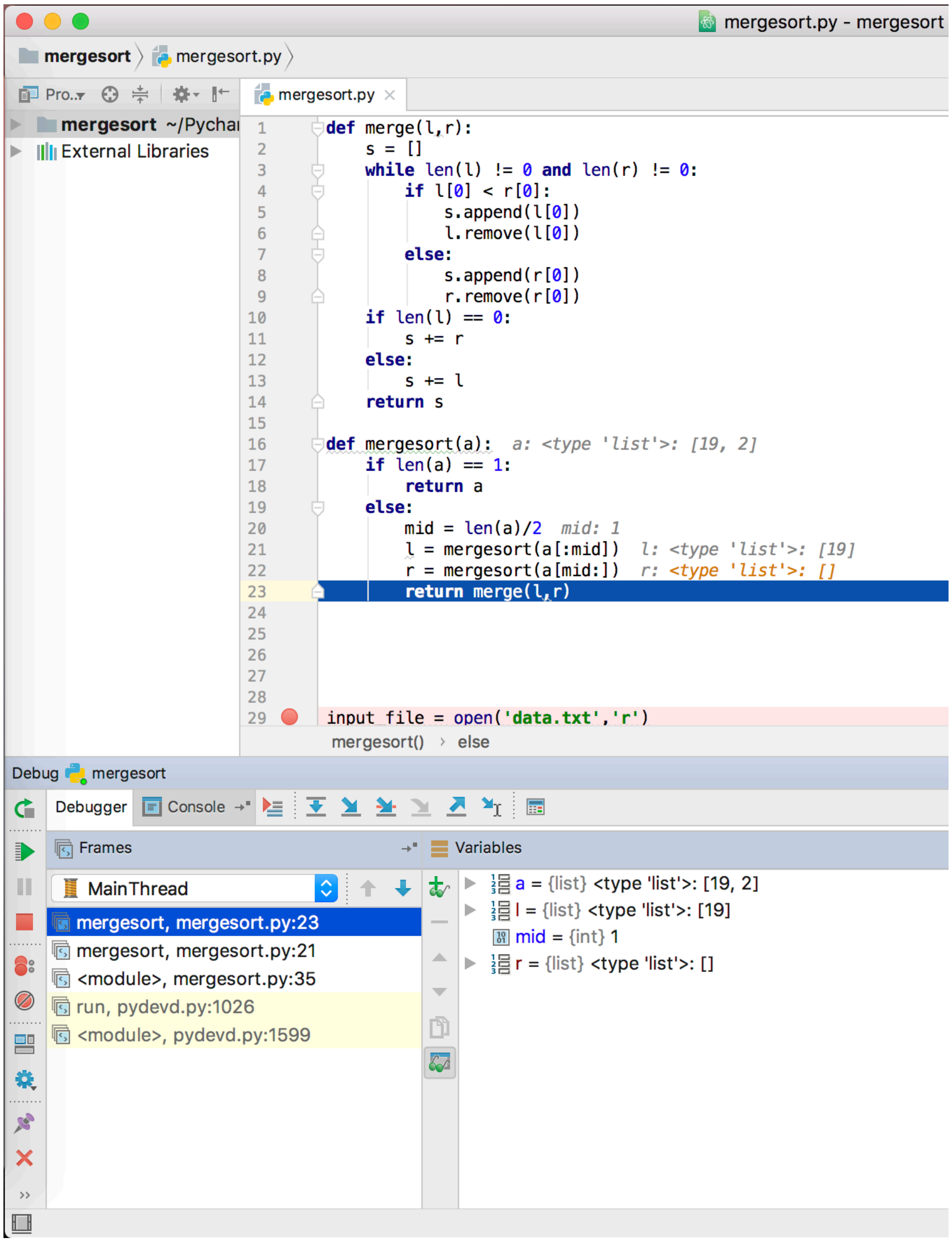
Answer:
See TEACH for code files. See below for my debugging of the code. I did this to try my best to fully understand how the recursion worked and what exactly returned from each call to mergesort and merge.



```python
def merge(l,r):   l: <type 'list'>: [19]   r: <type 'list'>: [2]
    s = []
    while len(l) != 0 and len(r) != 0:
        if l[0] < r[0]:
            s.append(l[0])
            l.remove(l[0])
        else:
            s.append(r[0])
            r.remove(r[0])
    if len(l) == 0:
        s += r
    else:
        s += l
    return s

def mergesort(a):
    if len(a) == 1:
        return a
    else:
        mid = len(a)/2
        l = mergesort(a[:mid])
        r = mergesort(a[mid:])
        return merge(l,r)

input_file = open('data.txt','r')
output_file = open('merge.out','w')
line = input_file.readline().split(' ')
while line != ['']:
    line = map(int, line)
```

```
l = {list} <type 'list'>: [19]
r = {list} <type 'list'>: [2]
```

```python
        if l[0] < r[0]:
            s.append(l[0])
            l.remove(l[0])
        else:
            s.append(r[0])
            r.remove(r[0])
    if len(l) == 0:
        s += r
    else:
        s += l
    return s

def mergesort(a):
    if len(a) == 1:
        return a
    else:
        mid = len(a)/2
        l = mergesort(a[:mid])
        r = mergesort(a[mid:])
        return merge(l,r)

input_file = open('data.txt','r')
output_file = open('merge.out','w')
line = input_file.readline().split(' ')
while line != ['']:
    line = map(int, line)
    line = line[1:]
    s = mergesort(line)
    output_file.write(' '.join(map(str,s)))
    output_file.write('\n')
    line = input_file.readline().split(' ')

input_file.close()
output_file.close()

# https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/tutoria
# https://pythonandr.com/2015/07/05/the-merge-sort-python-code/
```



mergesort.py - mergesort

mergesort 〉 mergesort.py 〉

mergesort.py ×

mergesort ~/Pychar

External Libraries

```
def merge(l,r):
    s = []
    while len(l) != 0 and len(r) != 0:
        if l[0] < r[0]:
            s.append(l[0])
            l.remove(l[0])
        else:
            s.append(r[0])
            r.remove(r[0])
    if len(l) == 0:
        s += r
    else:
        s += l
    return s

def mergesort(a):   a: <type 'list'>: [19, 2]
    if len(a) == 1:
        return a
    else:
        mid = len(a)/2   mid: 1
        l = mergesort(a[:mid])   l: <type 'list'>: [19]
        r = mergesort(a[mid:])   r: <type 'list'>: []
        return merge(l,r)





input_file = open('data.txt','r')
```

mergesort() › else

Debug mergesort

Frames

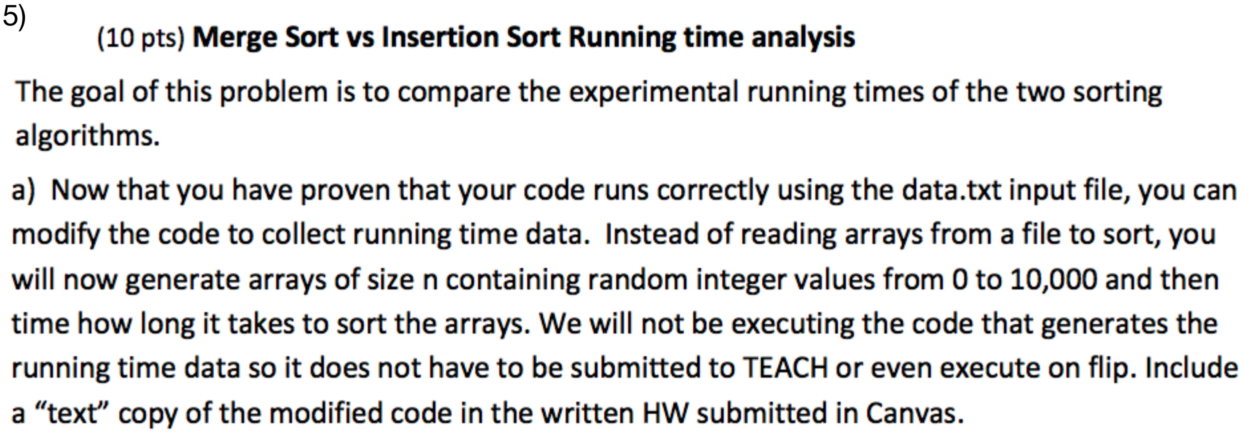
MainThread

mergesort, mergesort.py:23
mergesort, mergesort.py:21
<module>, mergesort.py:35
run, pydevd.py:1026
<module>, pydevd.py:1599

Variables

```
▶ a = {list} <type 'list'>: [19, 2]
▶ l = {list} <type 'list'>: [19]
  mid = {int} 1
▶ r = {list} <type 'list'>: []
```

5)



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

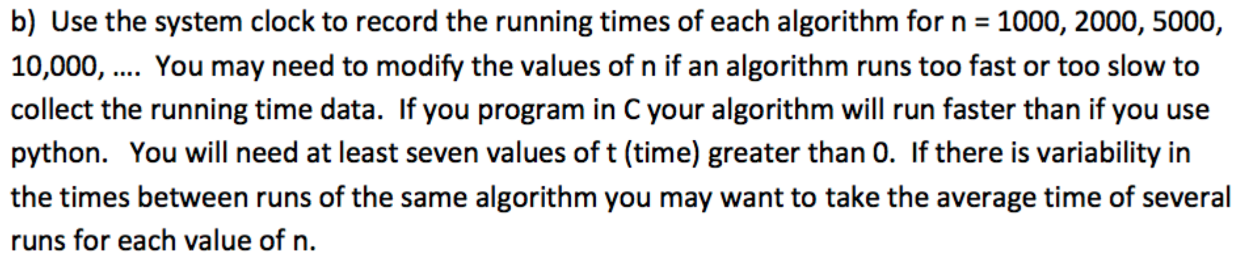
The goal of this problem is to compare the experimental running times of the two sorting algorithms.

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: Below is the contents of mergesort_insertsort.py

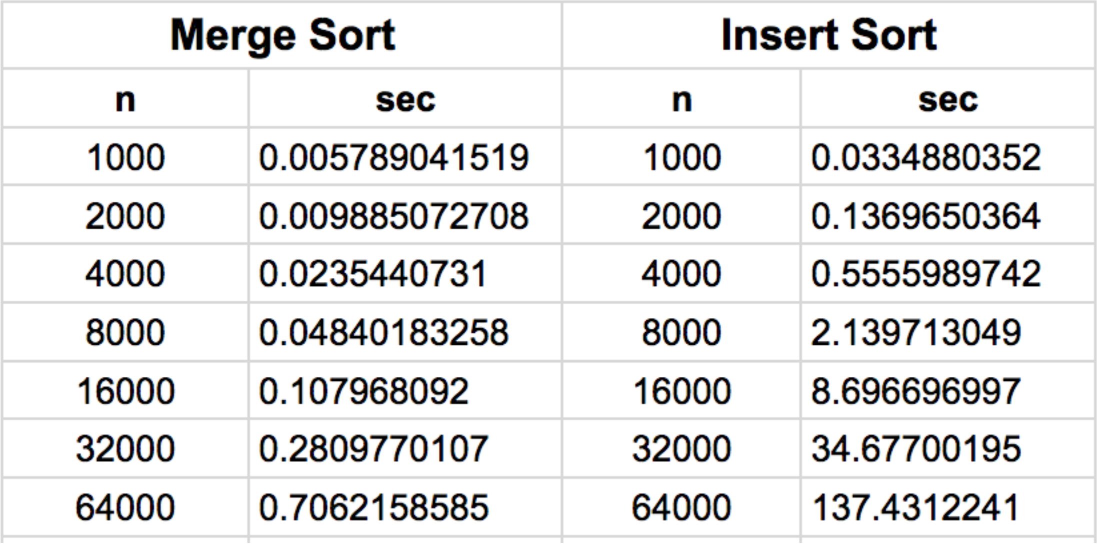
```
'''
Jon-Eric Cook
CS-325
Homework #1
This program compares the experimental running times of both the merger sort
algorithm and insert sort algorithm. This is done timing how long each algorithm
takes to sort a randomly generated array of length n.
'''
import random
import timeit

# wrapper function
def wrapper(func, *args, **kwargs):
    def wrapped():
        return func(*args, **kwargs)
    return wrapped

# merge algorithm
def merge(l,r):
    # empty array to be used to hold sorted values
    s = []
    # append to s array until either l or r array is empty
    while len(l) != 0 and len(r) != 0:
        if l[0] < r[0]:
            s.append(l[0])
            l.remove(l[0])
        else:
            s.append(r[0])
            r.remove(r[0])
    # add the remaining value to the s array
    if len(l) == 0:
        s += r
    else:
        s += l
    return s
```

```
# mergesort algorithm
def mergesort(a):
    # if the array has only one element, return it
    if len(a) == 1:
        return a
    else:
        # call mergesort on half of the current input array
        mid = len(a)/2
        l = mergesort(a[:mid])
        r = mergesort(a[mid:])
        return merge(l,r)

# insert sort algorithm
def insertsort(a):
    # look through each element of the array
    for j in range(1,len(a)):
        key = a[j]
        i = j - 1
        while i >= 0 and a[i] > key:
            a[i + 1] = a[i]
            i = i - 1
        a[i + 1] = key
    return a

# arrays to hold times from sort functions
mergesort_data = []
insertsort_data = []

# number of elements to be in random array
elements = [1000,2000,4000,8000,16000,32000,64000]

# loop 7 times, timing the run time of each sort algorithm for 7 different
# array sizes
for x in range(7):
    # randomly generate arrays to be sorted for mergesort and insertsort
    rand_list_mergesort = [int(1000*random.random()) for i in xrange(elements[x])]
    rand_list_insertsort = rand_list_mergesort

    # wrap mergesort function
    wrapped = wrapper(mergesort, rand_list_mergesort)
    # time mergesort function
    time_taken = timeit.timeit(wrapped, number=1)

    # log number of elements and time taken for mergesort
    n = "n = "
    n += str(elements[x])
    mergesort_data.append(n)
    sec = "sec = "
    sec += str(time_taken)
    mergesort_data.append(sec)

    # wrap insertsort function
    wrapped = wrapper(insertsort, rand_list_insertsort)
    # time insertsort function
    time_taken = timeit.timeit(wrapped, number=1)

    # log number of elements and time taken for insertsort
    n = "n = "
    n += str(elements[x])
    insertsort_data.append(n)
    sec = "sec = "
    sec += str(time_taken)
    insertsort_data.append(sec)

print "Merge Sort"
print mergesort_data

print "Insert Sort"
print insertsort_data


# RESOURCES:
# The below link was used to help me complete this problem.
# http://pythoncentral.io/time-a-python-function/
```



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 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.

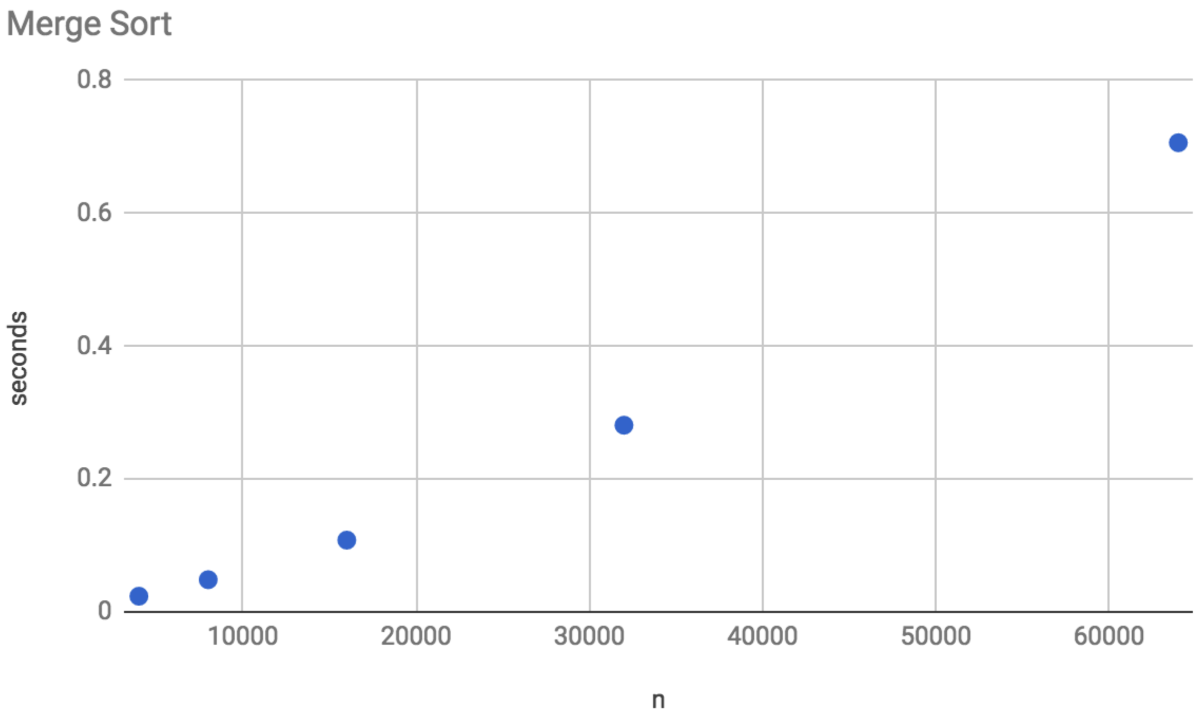
Answer:

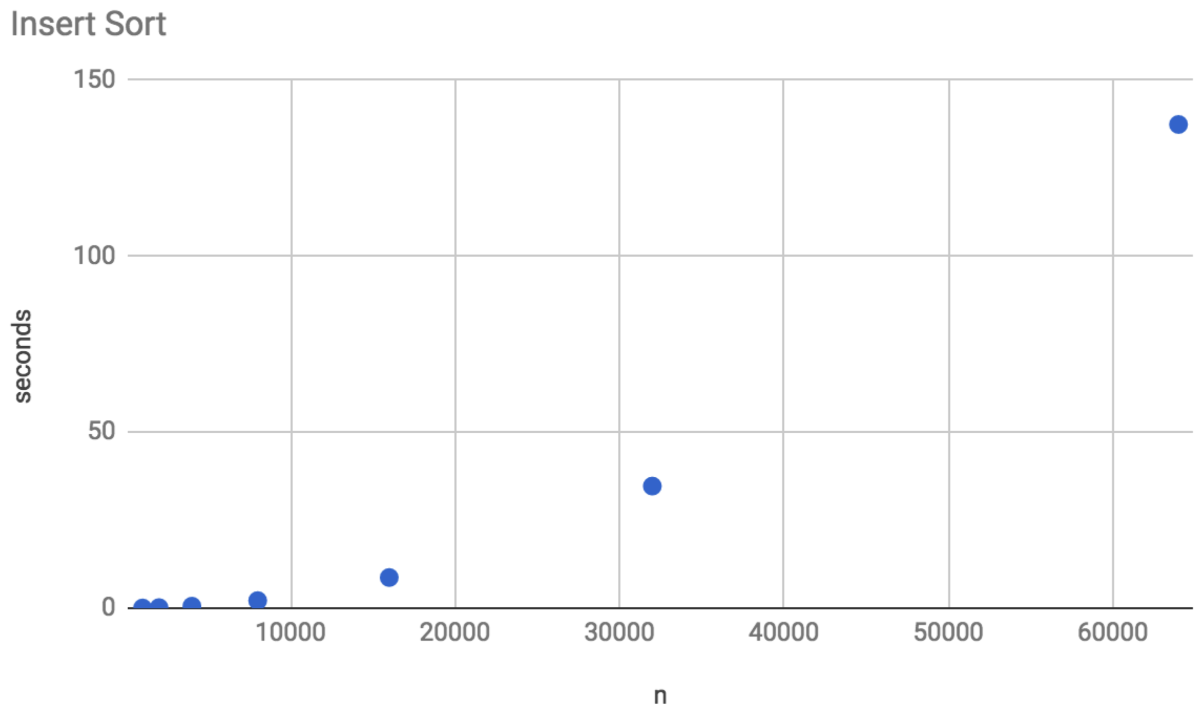


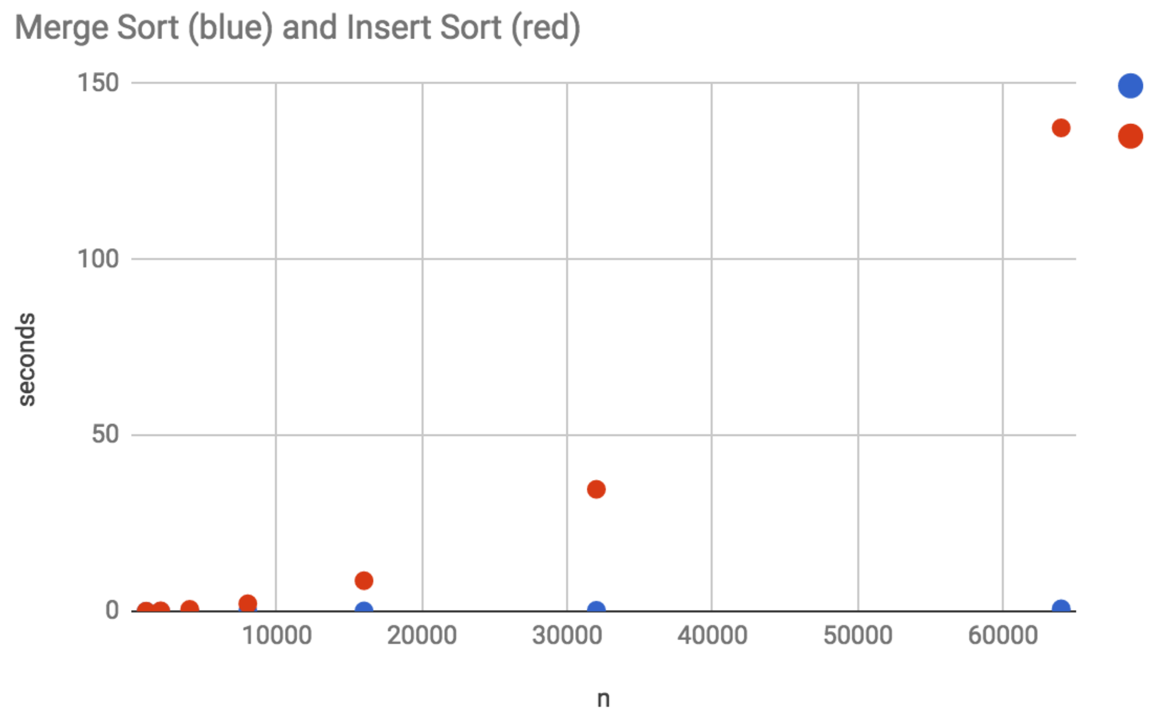
Merge Sort		Insert Sort	
n	sec	n	sec
1000	0.005789041519	1000	0.0334880352
2000	0.009885072708	2000	0.1369650364
4000	0.0235440731	4000	0.5555989742
8000	0.04840183258	8000	2.139713049
16000	0.107968092	16000	8.696696997
32000	0.2809770107	32000	34.67700195
64000	0.7062158585	64000	137.4312241

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: The individual graphs represent the data best.

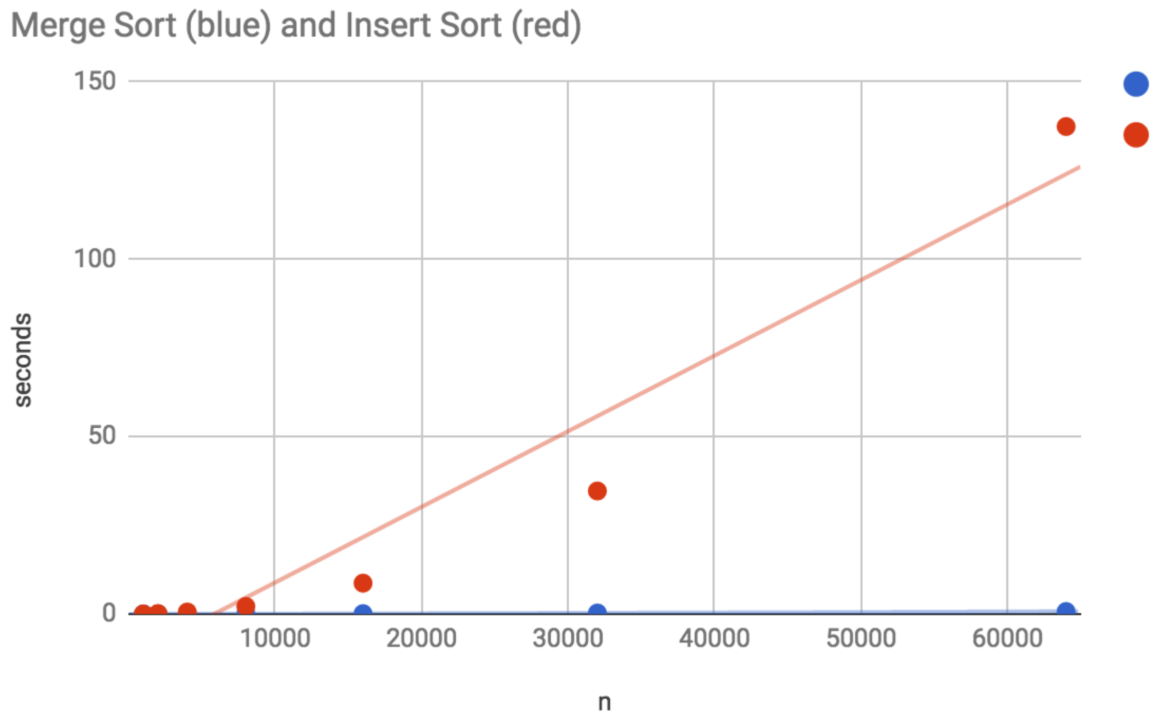


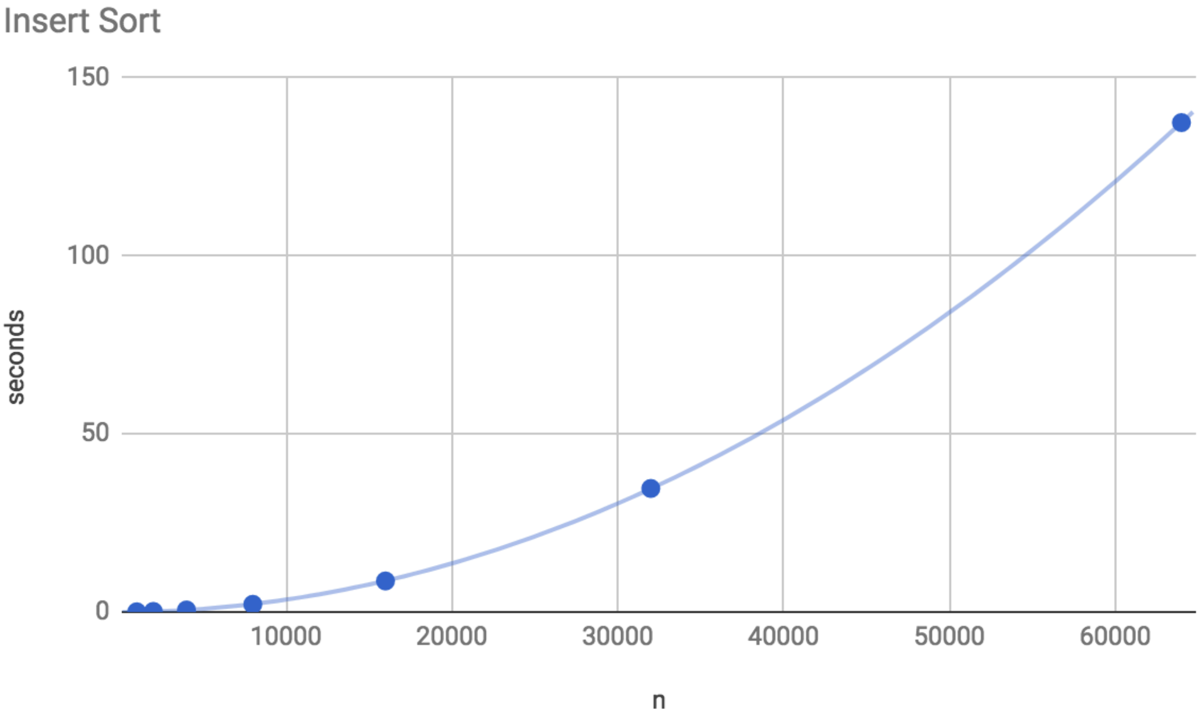


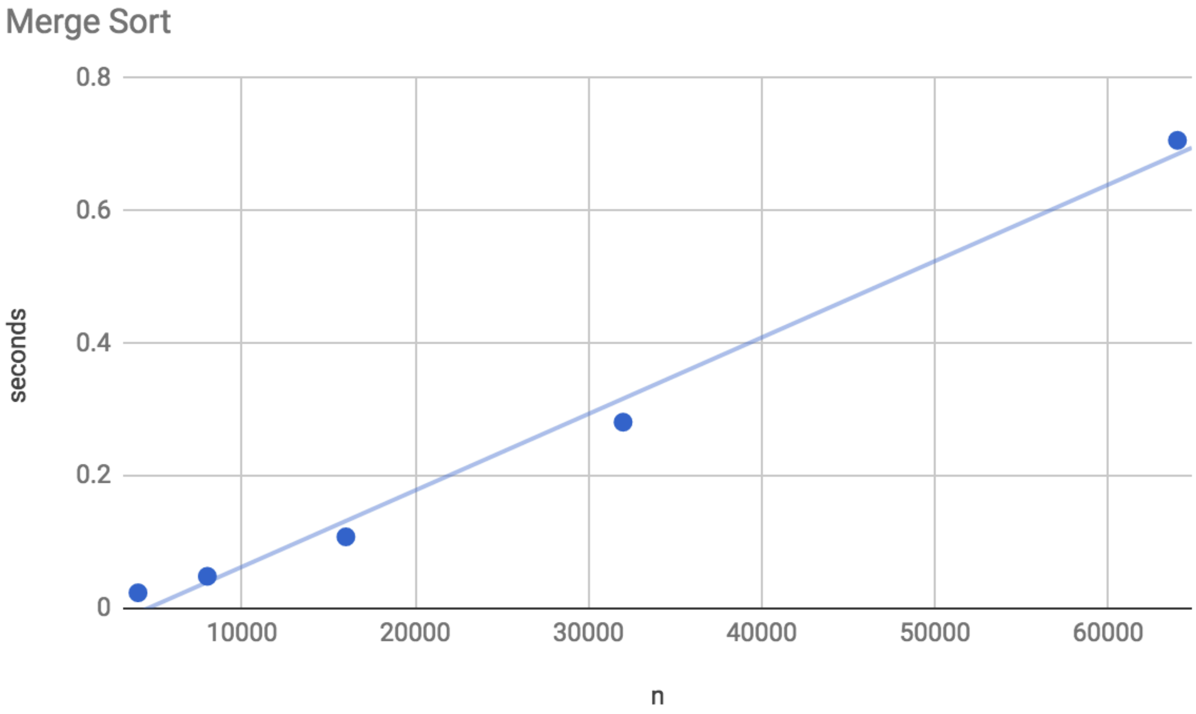


d) What type of curve best fits each data set? Again you can use Excel, Matlab, any software or a graphing calculator to calculate a regression equation. Give the equation of the curve that best "fits" the data and draw that curve on the graphs of created in part c).

Answer: Merge Sort approx equation - f(n) = n Insert Sort approx equation - f(n) = n^2




Insert Sort
150
100
50
0
seconds
10000
20000
30000
40000
50000
60000
n


Merge Sort
0.8
0.6
0.4
0.2
0
seconds
10000
20000
30000
40000
50000
60000
n

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.

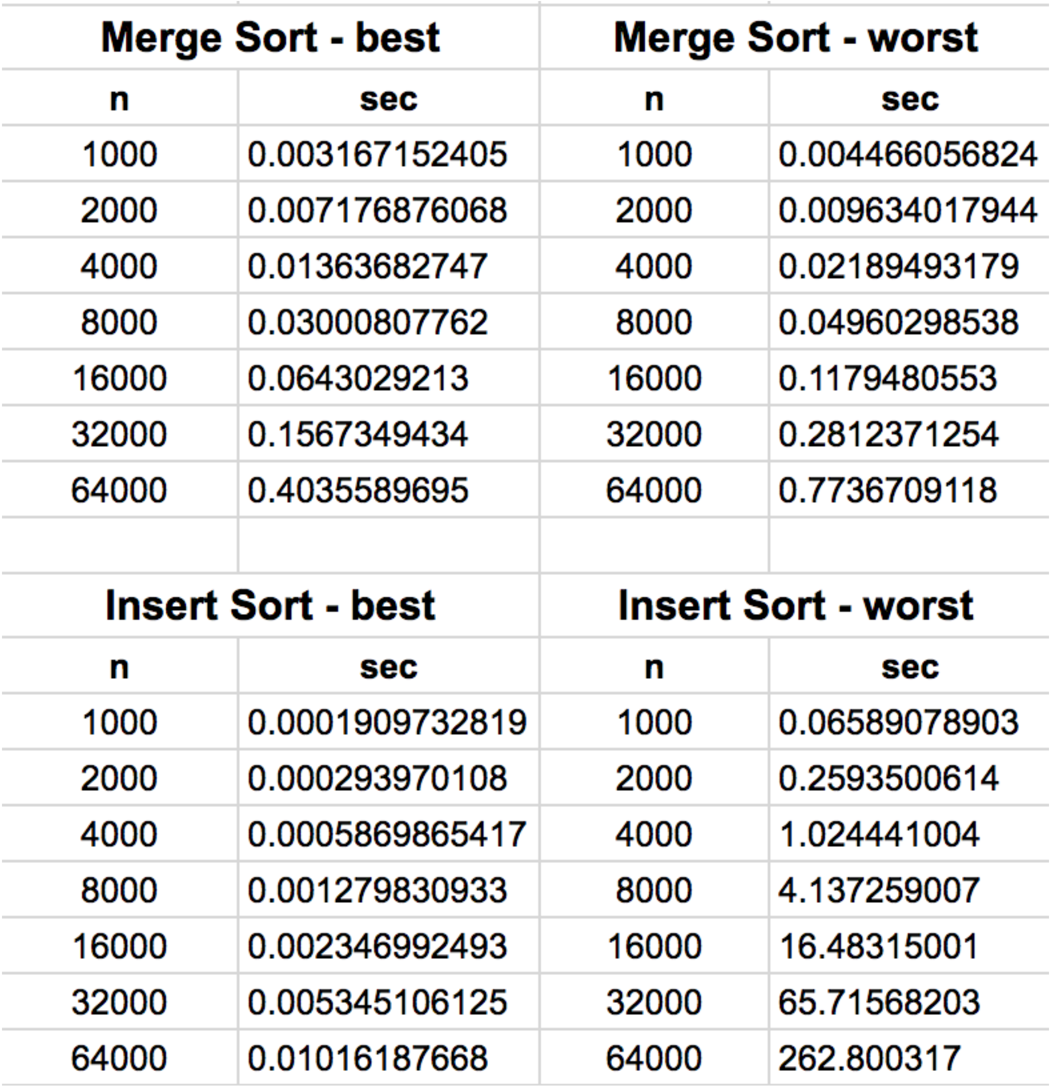
Answer:
For the Insert Sort algorithm the experimental running times appeared to be closer to the theoretical worst case, O(n^2). The curve fit for the above graph points to an equation of the f(n) = n^2 nature.

For the Merge Sort algorithm the experimental running times appeared to be similar the theoretical average case, O(n logn). The curve fit for the above graph points to an equation of the f(n) = n nature. Due to a limit of data points, the above graph may very well be illustrating only the beginning of the equation f(n) = n*log(n).

EXTRA CREDIT

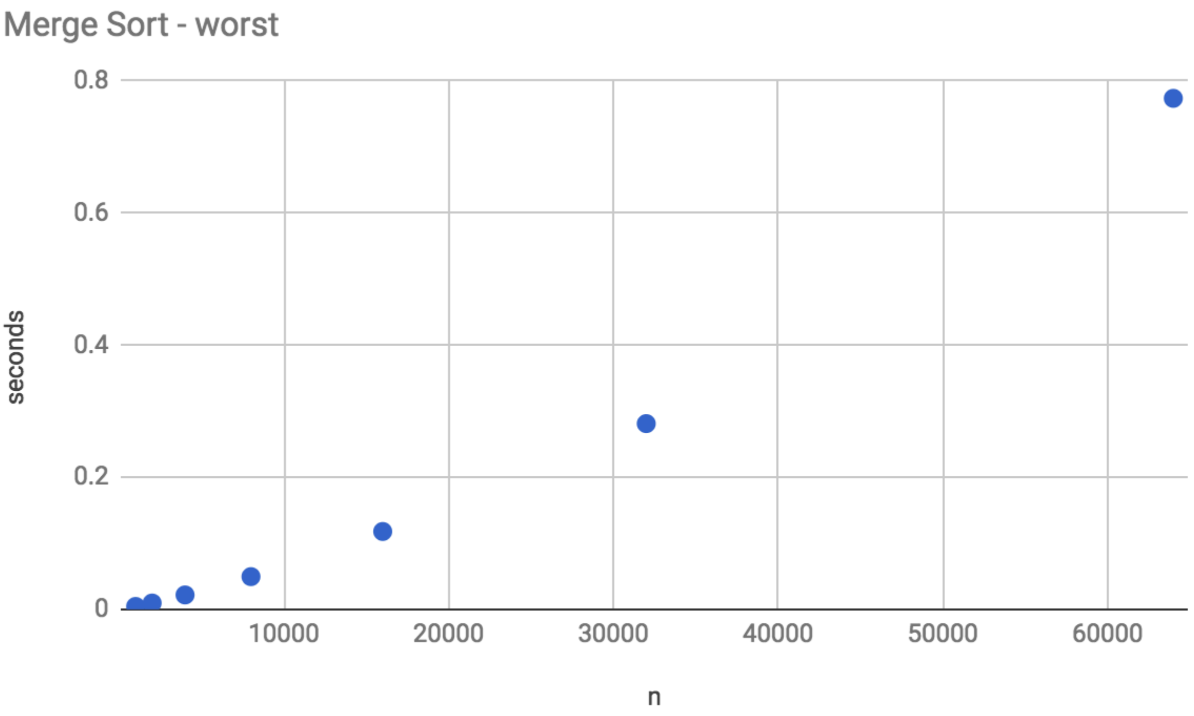
Generate best case and worst case input for both algorithms and repeat the analysis in parts b) to d) above. Discuss your results and submit your code to TEACH.

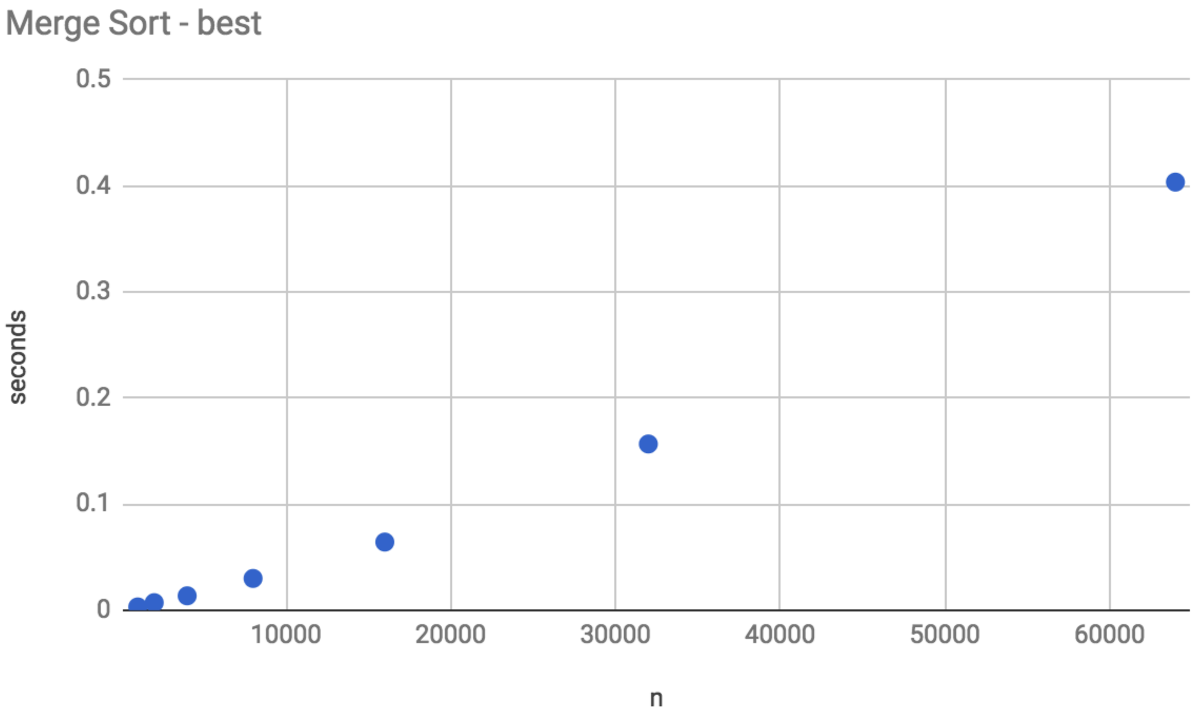
b)

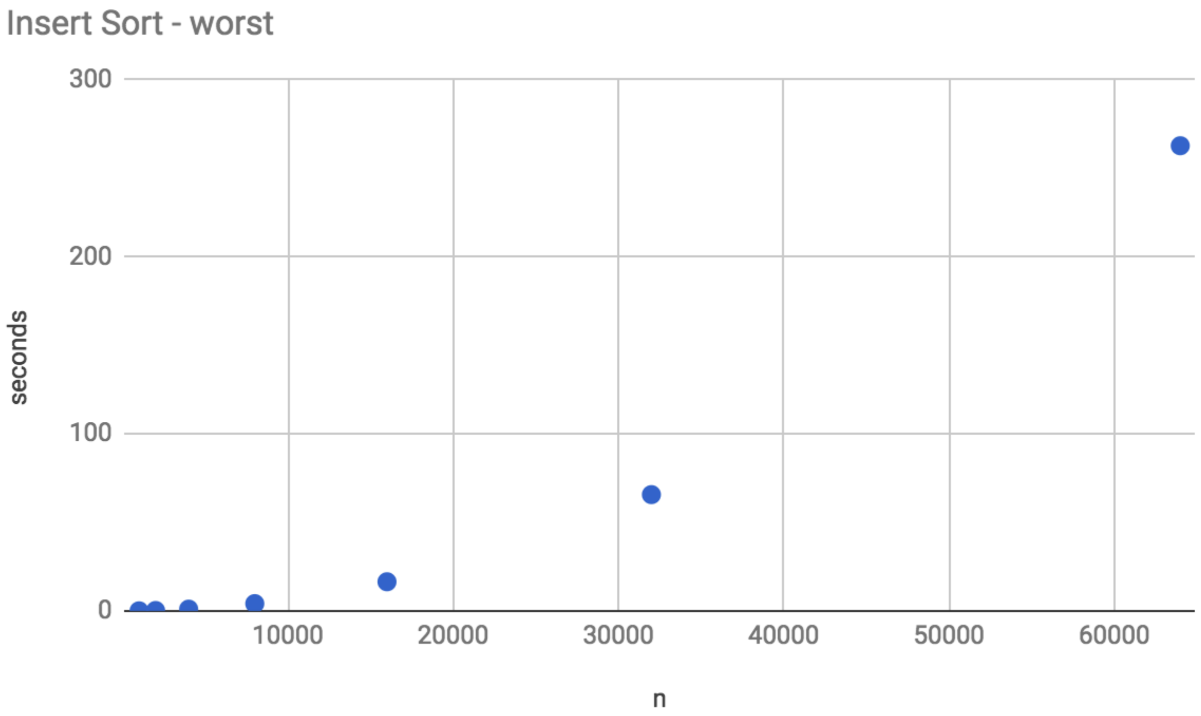


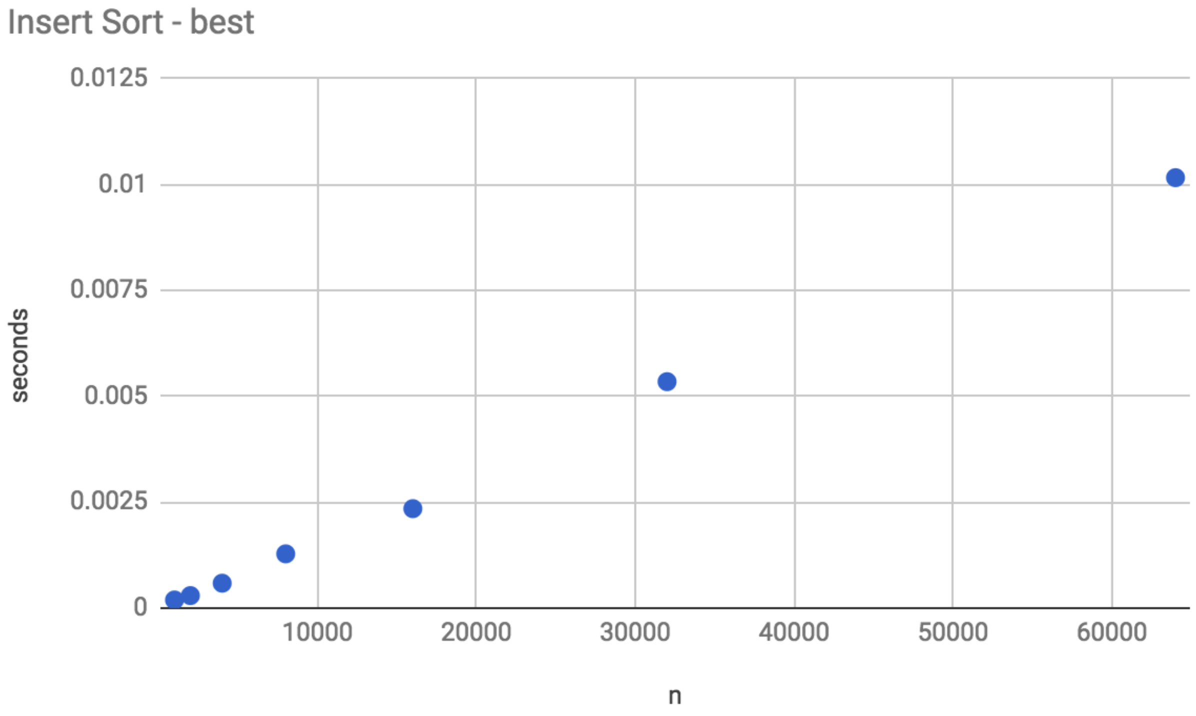
Merge Sort - best		Merge Sort - worst	
n	sec	n	sec
1000	0.003167152405	1000	0.004466056824
2000	0.007176876068	2000	0.009634017944
4000	0.01363682747	4000	0.02189493179
8000	0.03000807762	8000	0.04960298538
16000	0.0643029213	16000	0.1179480553
32000	0.1567349434	32000	0.2812371254
64000	0.4035589695	64000	0.7736709118
Insert Sort - best		**Insert Sort - worst**	
n	sec	n	sec
1000	0.0001909732819	1000	0.06589078903
2000	0.000293970108	2000	0.2593500614
4000	0.0005869865417	4000	1.024441004
8000	0.001279830933	8000	4.137259007
16000	0.002346992493	16000	16.48315001
32000	0.005345106125	32000	65.71568203
64000	0.01016187668	64000	262.800317

c)


Merge Sort - worst
0.8
0.6
0.4
0.2
0
seconds
10000
20000
30000
40000
50000
60000
n

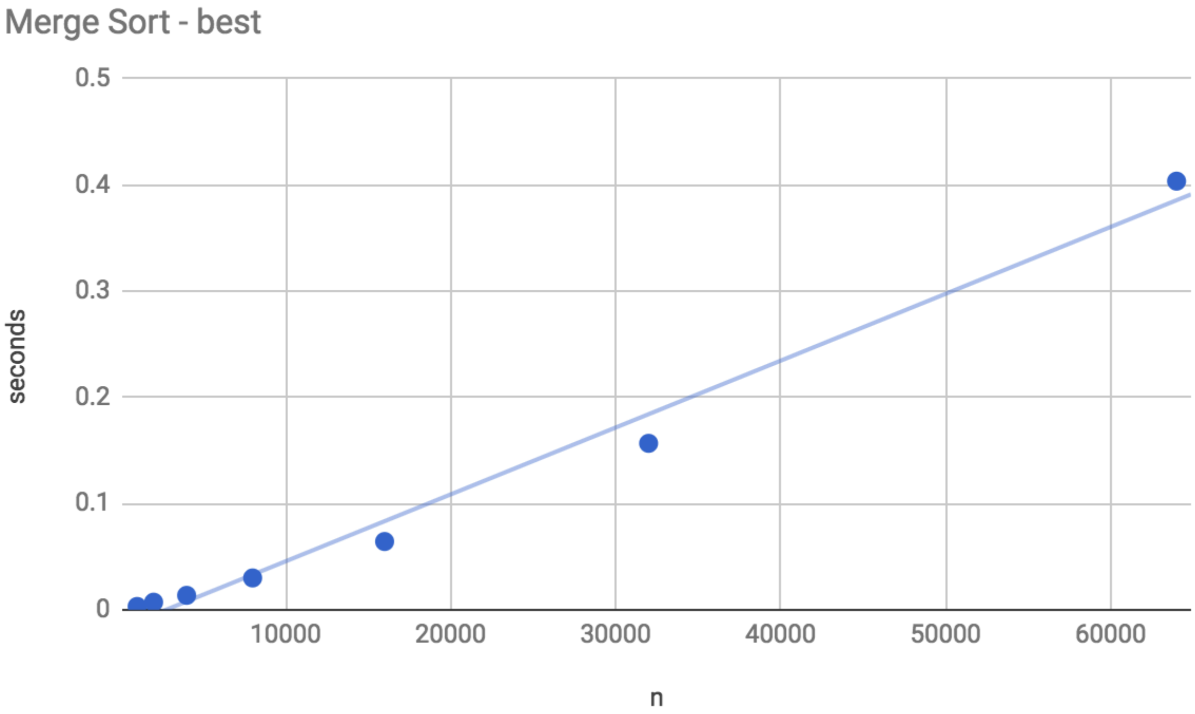

Merge Sort - best
0.5
0.4
0.3
0.2
0.1
0
seconds
10000
20000
30000
40000
50000
60000
n

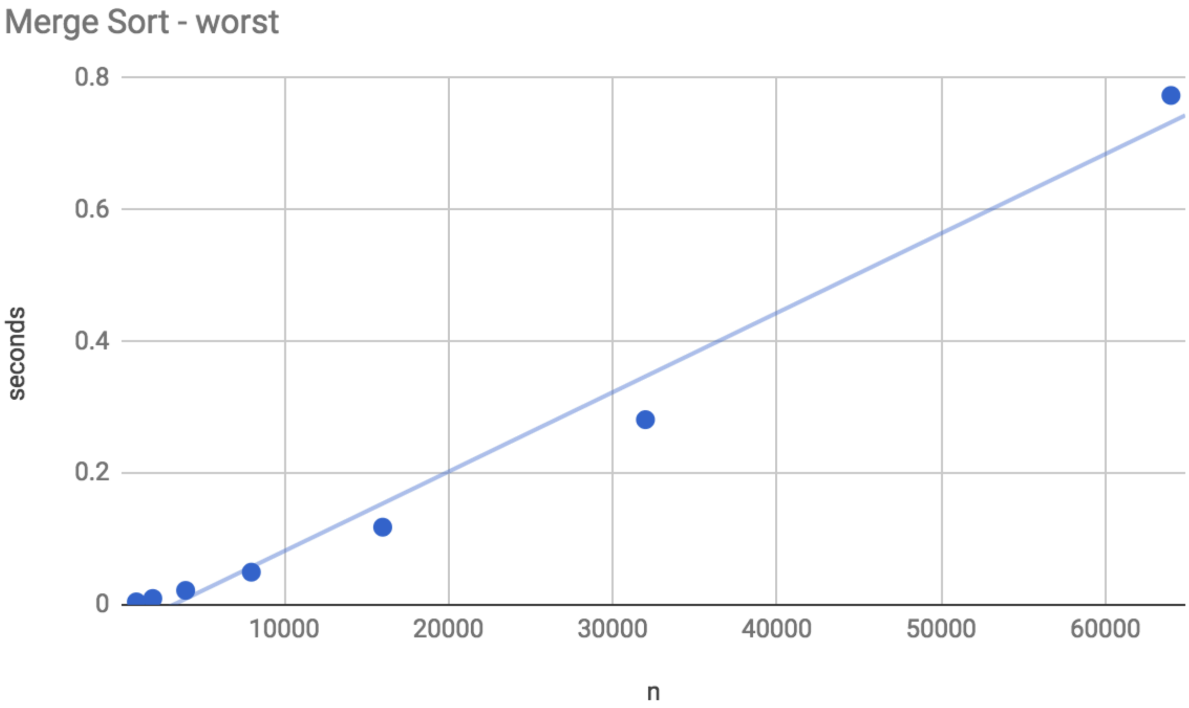

Insert Sort - worst
300
200
100
0
seconds
10000
20000
30000
40000
50000
60000
n

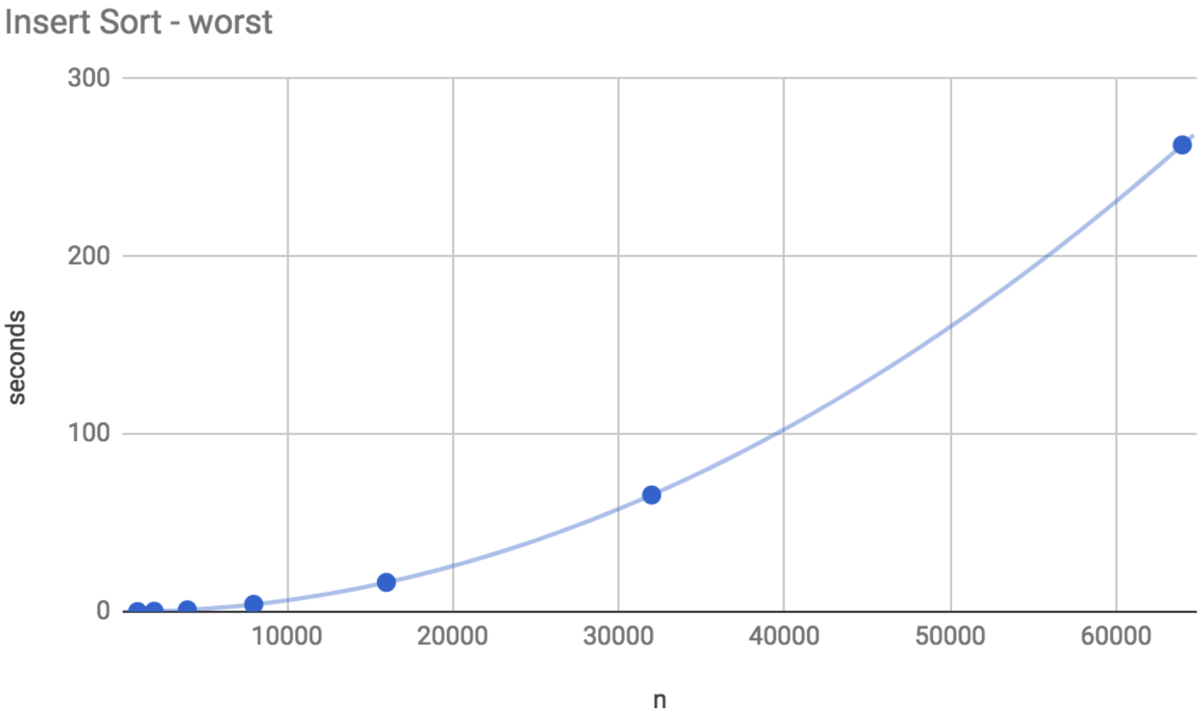

Insert Sort - best
0.0125
0.01
0.0075
0.005
0.0025
0
seconds
10000
20000
30000
40000
50000
60000
n

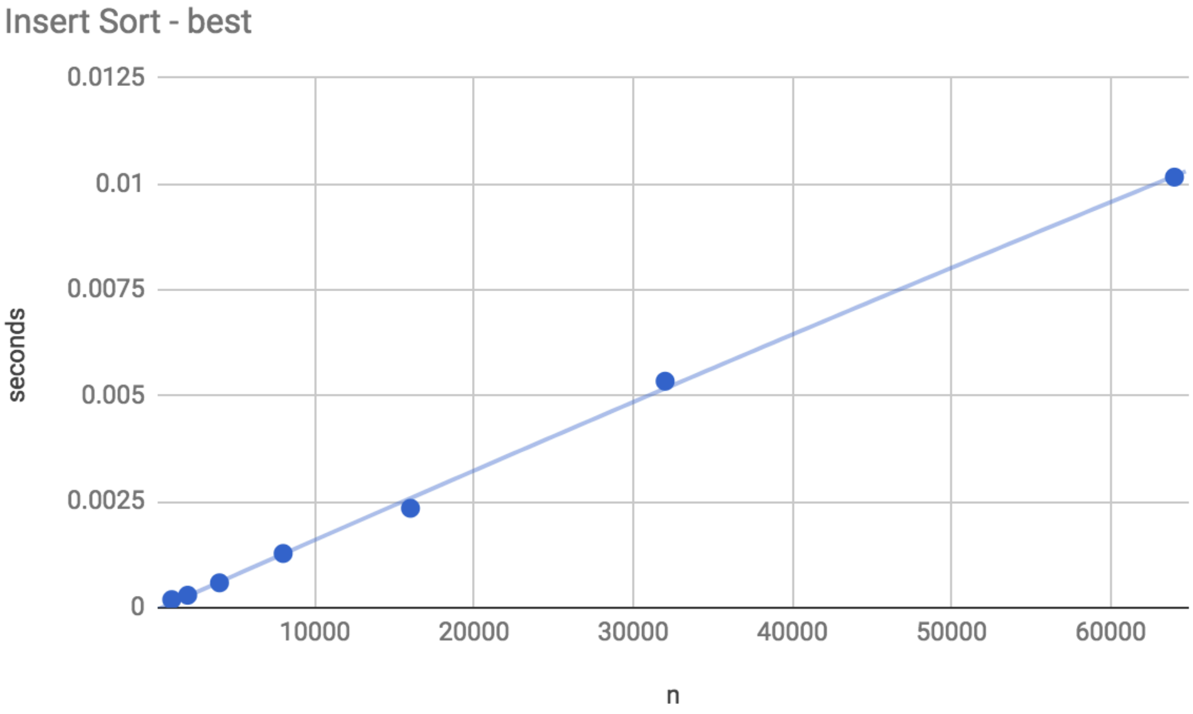
d)

Answer: Merge Sort approx equation - f(n) = n   Insert Sort approx equation - f(n) = n^2






Insert Sort - worst
300
200
100
0
seconds
10000
20000
30000
40000
50000
60000
n


Insert Sort - best
0.0125
0.01
0.0075
0.005
0.0025
0
seconds
10000
20000
30000
40000
50000
60000
n

e)

Answer:

For the Insert Sort algorithm, when using the worst case input, the experimental running times appeared to be very close to the theoretical worst case, O(n^2), as expected. The curve fit for the above graph points to an equation of the f(n) = n^2 nature. When the best case input was use, the experimental running times appeared to be very close to the theoretical best case, O(n), as expected

For the Merge Sort algorithm, when using the worst case input, the experimental running times appeared to be very close to O(n lgn), as expected. There technically isn't a worst case for the Merge Sort algorithm but it was discovered an array setup that would be the hardest to sort. See link at the bottom of mergesort_worst.py The curve fit for the above graph points to an equation of the f(n) = n nature. When the best case input was use, the experimental running times appeared to be very close to O(n lgn), as expected. Again, there technically isn't a best case for the Merge Sort algorithm but the results of this "best case" were faster than the "worst case".