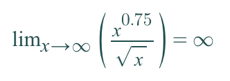
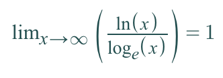
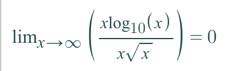
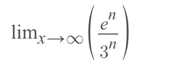
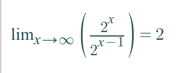
Homework #1

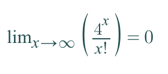
1a.) because 

1b.) because 

1c.) because 

1d.) because = 0

1e.) because 

1f.) because 

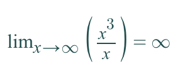
2a.) If f1(n) = Ω(g(n)) and f2(n) = O(g(n)) then f1(n)= Θ (f2(n) )

Prove false with counter-example:

Let , , and

and

BUT,

Because 

2b.) If f1(n) = O(g1(n)) and f2(n) = O(g2(n)) then f1(n)+ f2(n)= O(g1(n) + g2(n) )

Prove true:

If , then there exists constants such that  
 I. for all and

since , there exists constants such that  
 II. for all

Let

III. for all

IV. for all

By adding III & IV,

for all

Therefore,

and

or

4a.)  
#!/usr/bin/python3  
  
*'''  
Name: Kevin Harvell  
Date: 1/9/19  
About: This program has a function called mergeSort that recursively splits  
an array in half until 1 element in each array. Then builds sorted arrays  
by combining the smaller arrays and comparing the values.  
It takes an input file and sorts using the insertionSort function, and creates  
an output file.  
It creates an array of random numbers between 1 and 10000 of size n and  
times how long it takes to sort the array  
  
# The following code is based on pseudocode from Introduction to Algorithms - 3rd Edition p.26  
'''*import random  
import time  
  
def insertionSort(arr):  
 for j in range(1, len(arr)):  
 key = arr[j]  
 # Insert arr[j] into the sorted sequence  
 i = j  
 while i > 0 and arr[i - 1] > key:  
 arr[i] = arr[i - 1]  
 i = i - 1  
 arr[i] = key  
 return arr  
  
  
unsorted = []  
n = input("Enter the number of elements in the array: ")  
for x in range(n):  
 unsorted.append(random.randint(0, 10000))  
print(unsorted)  
t0 = time.time()  
print(insertionSort(unsorted))  
t1 = time.time()  
print(t1 - t0)

#!/usr/bin/python3  
  
*'''  
Name: Kevin Harvell  
Date: 1/9/19  
About: This program has a function called mergeSort that recursively splits  
an array in half until 1 element in each array. Then builds sorted arrays  
by combining the smaller arrays and comparing the values.  
It creates an array of random numbers between 1 and 10000 of size n and  
times how long it takes to sort the array  
  
# The following code is based on code from  
# http://interactivepython.org/courselib/static/pythonds/SortSearch/TheMergeSort.html  
'''*import random  
import time  
  
def mergeSort(arr):  
 # If array is greater than 1, sort. Otherwise, no need to sort  
 if len(arr) > 1:  
 # Split the array into 2 halves  
 mid = len(arr) // 2  
 left = arr[:mid]  
 right = arr[mid:]  
 # Keep splitting arrays into halves until only one element in both halves  
 mergeSort(left)  
 mergeSort(right)  
  
 i = 0  
 j = 0  
 k = 0  
 # If there are elements not yet compared in both halves, compare and  
 # put lower/equal value in next spot in array  
 while i < len(left) and j < len(right):  
 if left[i] < right[j]:  
 arr[k] = left[i]  
 i = i + 1  
 else:  
 arr[k] = right[j]  
 j = j + 1  
 k = k + 1  
 # There are still elements in the left array  
 # Put them in the next spot in array  
 while i < len(left):  
 arr[k] = left[i]  
 i = i + 1  
 k = k + 1  
 # There are still elements in the right array  
 # Put them in the next spot in array  
 while j < len(right):  
 arr[k] = right[j]  
 j = j + 1  
 k = k + 1  
 return arr  
  
unsorted = []  
n = input("Enter the number of elements in the array: ")  
for x in range(n):  
 unsorted.append(random.randint(0, 10000))  
print(unsorted)  
t0 = time.time()  
print(mergeSort(unsorted))  
t1 = time.time()  
print(t1 - t0)

4b.)

|  |  |
| --- | --- |
| Insertion Sort | |
| n | T(n) |
| 625 | 0.04333333 |
| 1250 | 0.49 |
| 2500 | 0.57333333 |
| 5000 | 1.79333333 |
| 7500 | 3.23333333 |
| 10000 | 6.22 |
| 12500 | 7.72666667 |
| 15000 | 13.45 |
| 20000 | 24.6166667 |
| 25000 | 39.97 |

|  |  |
| --- | --- |
| Merge Sort | |
| n | T(n) |
| 20000 | 0.19666667 |
| 40000 | 0.44333333 |
| 60000 | 0.61666667 |
| 80000 | 0.84 |
| 100000 | 0.81333333 |
| 120000 | 0.99666667 |
| 140000 | 1.08 |
| 160000 | 1.14333333 |
| 180000 | 1.67333333 |
| 200000 | 2.09 |

4c.) As expected, a quadratic regression best fit the insertion sort data. The equation for the regression line is featured below on the graph with an R^2 value.

Unexpectedly, a power regression best fit the merge sort data. The equation for the regression line is featured below on the graph with an R^2 value. I attribute this unexpected outcome to flip server irregularities. In a perfect world, my server would not be used by many other students. Even though I took several tests and averaged the results, the data did not fit a logarithmic regression the best as I would expect theoretically.

4d.)

4e.) I am struggling to compare experimental running times to theoretical running times because for my experimental running times, there is a known computer processing speed whereas theoretically, there is no known computer processing speed.

In theory, insertion sort would have a worst-case scenario of O(n^2). If I were to take n=625, where experimentally I got T(n) = 0.04, and put it theoretically into f(n) = n^2, that would yield f(n) = 390625. I am not sure how to compare 0.04 with 390625. 390625 would in theory be multiplied by some constant based on the processing speed of the theoretical computer.

I have similar confusion for merge sort which would have a worst-case scenario of O(nlogn). Similarly, if I were to take n=20000, where experimentally I got T(n)=0.20, and put it theoretically into f(n)=nlgn, that would yield f(n) = 285754. Again, I am not sure how to compare 0.20 with 285754. 285754 would in theory be multiplied by some constant based on the processing speed of the theoretical computer.

All that said, it is clear to me that the experimental running times for insertion sort are clearly quadratic making the theory of it being O(n^2) very apparent.

However, for merge sort, the experimental running times created a trendline that was considered a Power function. This is probably due to inconsistencies in the flip server caused by the ebb and flow of other users using it, even though I ran several test cases and averaged the results. Interestingly, from n = 20000 to n = 160000 the merge sort experimental running times did appear to be close to the theoretical O(nlgn).