Kimberly Tom

CS 325

1/11/19

CS 325 Winter 2019 Homework 1

1)

a. f(n) is Ω(g(n)) because it simplifies to n^.25 and as n goes to infinity, the limit approaches positive infinity.

b. f(n) is Θ(g(n)) because as n goes to infinity, the limit approaches a constant (0.43429).

c. f(n) is O(g(n)) because as n goes to infinity, the denominator gets bigger faster than the numerator, so the limit approaches 0.

d. f(n) is O(g(n)) because as n goes to infinity, the denominator gets bigger faster than the numerator, and the limit approaches 0.

e. f(n) is Θ(g(n)) because as n goes to infinity, the limit approaches a constant (2).

f. f(n) is O(g(n)) because n! gets larger faster than 4^n as n goes to infinity, so the denominator dominates the numerator and the limit approaches zero.

2)

a. If $f_1(n) = \Omega(g(n))$ and $f_2(n) = O(g(n))$ then $f_1(n) = \Theta(f_2(n))$.

Disprove:

If g(n) = n, $f_1(n) = n^3$ and $f_2(n) = 5$

$f_1(n)$ is Ω(g(n)) because as n goes to infinity, the numerator increases faster than the denominator, and the limit approaches positive infinity.

$f_2(n)$ is O(g(n)) because as n goes to infinity, the denominator increases faster than the numerator, and the limit approaches 0.

Since $f_1(n) = n^3$ and $f_2(n) = 5$, as n goes to infinity, the numerator increases faster than the denominator, so the limit approaches positive infinity, which is not a constant value. Therefore, we have disproven the conjecture.

b. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$

prove:

I. If $f_1(n) = O(g_1(n))$, then there exists constants c_1 and n_1 such that $0 < f_1(n) <= c_1g_1(n)$ for all $n >= n_1$

II. if $f_2(n) = O(g_2(n))$, then there exists constants c_2 and n_2 such that $0 < f_2(n) <= c_2g_2(n)$) for all $n >= n_2$

III. let $g = g_1(n) + g_2(n)$. $0 < f_1(n) <= g$ for all $n >= n_1 + n_2$

IV. let $g = g_1(n) + g_2(n)$. $0 < f_2(n) <= g$ for all $n >= n_1 + n_2$

Put III and IV together

$0 < f_1 + f_2 <= g + g$ becomes $0 < f_1 + f_2 <= 2g$ for all $n >= n_1 + n_2$

So, $f_1(n) + f_2(n) <= cg(n)$, and then

$f_1(n) + f_2(n) = O(g(n))$

or $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$

3) See .zip file and the readme file in the zip folder for how to compile in python.

4)

a)

```
# mergeTime.py
# name: Kimberly Tom
# CS325 Homework 1
# mergesort function with help from https://www.geeksforgeeks.org/merge-sort/
# random integer generation with help from https://www.pythoncentral.io/how-
to-generate-a-random-number-in-python/
# run time with help from https://stackoverflow.com/questions/1557571/how-do-
i-get-time-of-a-python-programs-execution
# and https://docs.python.org/2/library/timeit.html

import random
import timeit

arraySizes = {1000, 3000, 5000, 7000, 9000, 11000, 13000, 15000, 17000,
19000}

def genRandNum(n):
    array_nums = []
    for x in range(n):
        array_nums.append(random.randint(0,10001))
    return array_nums


def mergeSort(integer_array):
    if len(integer_array) > 1:
        # get middle index of the array
        middle = len(integer_array)//2
        # create array of the left side
        left_side = integer_array[:middle]
        # create array of the right side
```

```
        right_side = integer_array[middle:]

        mergeSort(left_side)
        mergeSort(right_side)

        a = b = c = 0

        while a < len(left_side) and b < len(right_side):
            if left_side[a] < right_side[b]:
                integer_array[c] = left_side[a]
                a = a + 1
            else:
                integer_array[c] = right_side[b]
                b = b + 1
            c = c + 1

        while a < len(left_side):
            integer_array[c] = left_side[a]
            a = a + 1
            c = c + 1

        while b < len(right_side):
            integer_array[c] = right_side[b]
            b = b + 1
            c = c + 1

for number in arraySizes:
        array_rand = genRandNum(number)
        start_time = timeit.default_timer()
        mergeSort(array_rand)
        print("array size: " + str(number) + ", running time: " + "%s
seconds" % (timeit.default_timer() - start_time) + "\n")
```

```
# insertTime.py
# name: Kimberly Tom
# CS325 Homework 1
# insertSort with help from https://www.geeksforgeeks.org/insertion-sort/
# random integer generation with help from https://www.pythoncentral.io/how-
to-generate-a-random-number-in-python/
# run time with help from https://stackoverflow.com/questions/1557571/how-do-
i-get-time-of-a-python-programs-execution
# and https://docs.python.org/2/library/timeit.html


import random
import timeit

arraySizes = {1000, 3000, 5000, 7000, 9000, 11000, 13000, 15000, 17000,
19000}

def genRandNum(n):
    array_nums = []
```

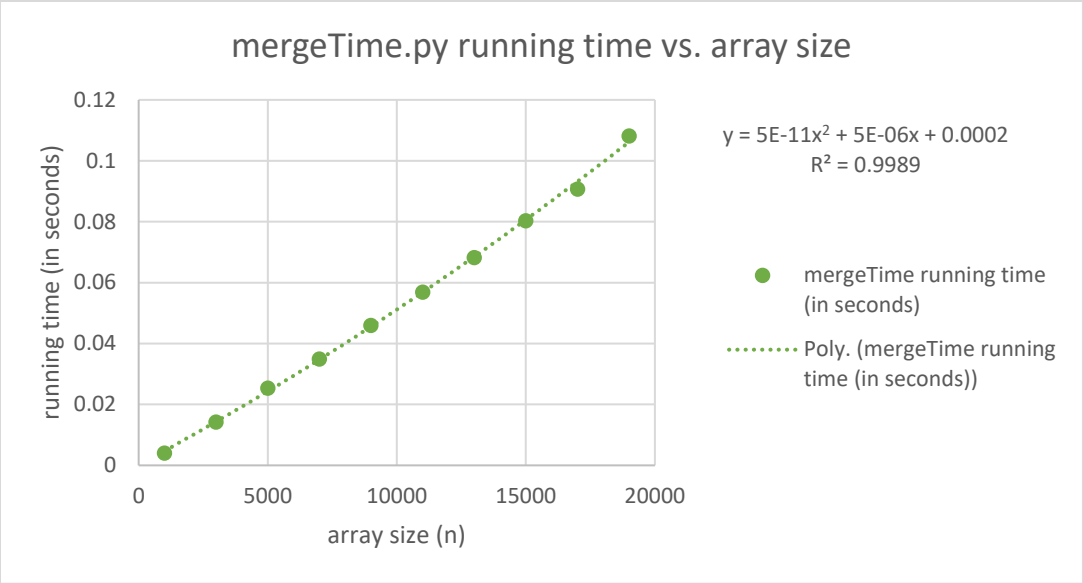
```
    for x in range(n):
        array_nums.append(random.randint(0,10001))
    return array_nums

def insertSort(integer_array):
    # for each number in the array
    for y in range(1, len(integer_array)):
        key = integer_array[y]

        # while the element is greater than the key, move one position
forward
        x = y - 1
        while x >= 0 and key < integer_array[x]:
            integer_array[x + 1] = integer_array[x]
            x = x - 1
        integer_array[x + 1] = key


for number in arraySizes:
        array_rand = genRandNum(number)
        start_time = timeit.default_timer()
        insertSort(array_rand)
        print("array size: " + str(number) + ", running time: " + "%s
seconds" % (timeit.default_timer() - start_time) + "\n")
```

b)

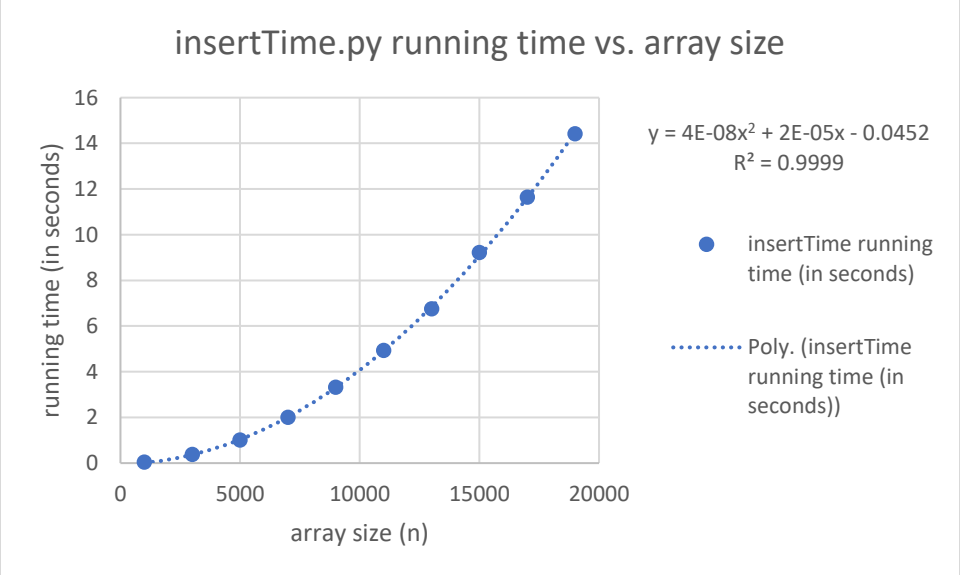
mergeTime.py	
array size (n)	**running time (in seconds)**
1000	0.004047155
3000	0.014290094
5000	0.025346041
7000	0.034968853
9000	0.045957088
11000	0.056860924
13000	0.068284988
15000	0.080347061
17000	0.09077096
19000	0.108187199

insertTime.py	
array size (n)	**running time (in seconds)**
1000	0.0436728
3000	0.369548082
5000	0.999078035
7000	2.002542973
9000	3.318465948
11000	4.926828861
13000	6.750751019
15000	9.208185196
17000	11.64162397
19000	14.4145031

c)



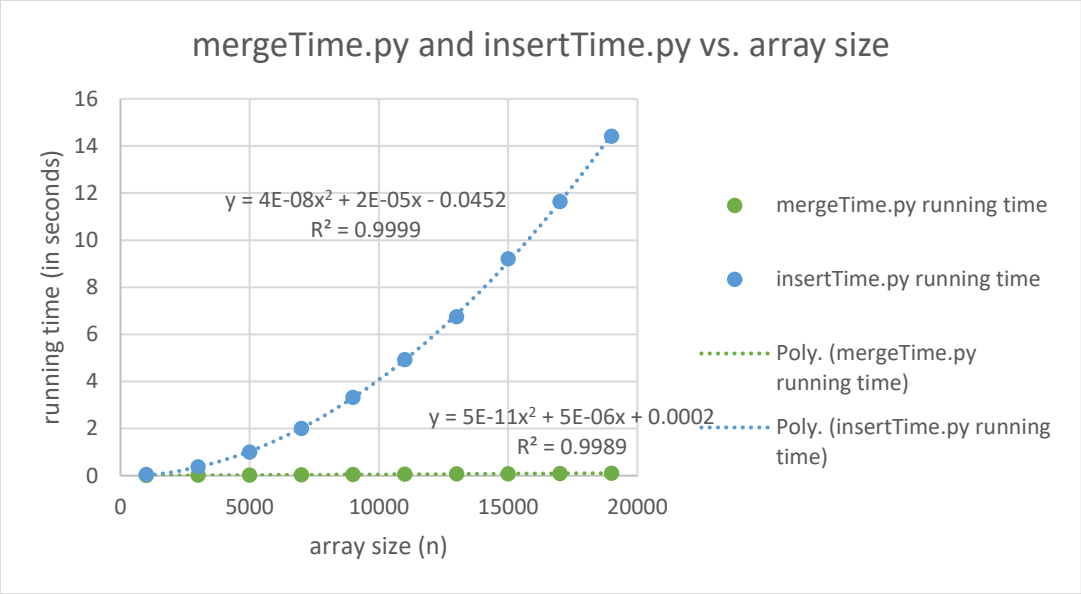
The curve that best fits the data for mergeTime.py is a polynomial line of degree 2. The equation that best fits the data is $y = 5E\text{-}11x^2 + 5E\text{-}06x + 0.0002$.



.

The curve that best fits the data for insertTime.py is a polynomial line of degree 2. The equation that best fits the data is y = 4E-08x2 + 2E-05x - 0.0452.

d)



e) For mergeTime.py, my experimental times fit the theoretical running time of the algorithm sort of close. The theoretical time is on average O(nlogn). By looking at my graph, because the last data point where I used an array size of 19000, has a longer run time than a linear model, I believe that my regression line may be O(nlogn) if I used bigger values of n. The R squared value is 0.9989 when using a polynomial line of degree 2, which was a better fit than when I tried a linear (which had a lower r squared value), so I believe my graph does look like O(nlogn) and it fits the theoretical line pretty well, and I would expect that if I use larger values of n that I would see it fitting even better.

For insertTime.py, my experimental running times fit the theoretical running time of the algorithm very nicely when using a polynomial trend line of degree 2. The R squared value is 0.9999 which is very close to 1, which shows that the data is very close to the fitted regression line. The average theoretical running time is O(n^2), which matches the experimental running time per my best fit line.