CS 325 Spring 2018 – HW 1

Sheng Bian

1)

	f(n)	g(n)	Relationship	Explanation
a	$n^{0.25}$	$\sqrt{n}$	$f(n)\ is\ O(g(n))$	$\lim_{n\to\infty} \frac{n^{0.25}}{\sqrt{n}} = \lim_{n\to\infty} \frac{1}{n^{0.25}} = 0$
b	log n^2	ln n	$f(n)$ is $\Theta(g(n))$	$\lim_{n\to\infty} \frac{log_{10}n^2}{log_e n} = \lim_{n\to\infty} \frac{2log_{10}n}{log_e n} = \lim_{n\to\infty} \frac{2log_{10}n}{log_{10}n/log_{10}e} = 2log_{10}e$
c	nlog n	$n\sqrt{n}$	$f(n)\ is\ O(g(n))$	$\lim_{n\to\infty} \frac{n\ logn}{n\sqrt{n}} = \lim_{n\to\infty} \frac{logn}{\sqrt{n}}$ then apply L’Hôpital Rule, $\lim_{n\to\infty} \frac{f'(n)}{g'(n)} = \lim_{n\to\infty} \frac{1/n}{0.5n^{-0.5}} = \lim_{n\to\infty} \frac{2}{\sqrt{n}} = 0$
d	2^n	3^n	$f(n)\ is\ O(g(n))$	$\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} (\frac{2}{3})^n = 0$
e	2^n	2^{n+2}	$f(n)$ is $\Theta(g(n))$	$\lim_{n\to\infty} \frac{2^n}{2^{n+2}} = \lim_{n\to\infty} \frac{1}{4} = \frac{1}{4}$
f	4^n	n!	$f(n)\ is\ O(g(n))$	$\lim_{n\to\infty} \frac{4^n}{n!} = 0$

2)

Base Case:

n = 2.

T(2) = 2lg 2 = 2.

For hypothesis, assume that T(n) =n* lg(n) for n = 2^k where k>1. We must show that

$T(2^{k+1}) = 2^{k+1} * \lg(2^{k+1})$

Inductive Case:

Let n = 2^{k+1}

$$T(2^{k+1}) = 2T\left(\frac{2^{k+1}}{2}\right) + 2^{k+1}$$

$$= 2T(2^k) + 2^{k+1}$$

$$= 2 * (2^k * \lg(2^k)) + 2^{k+1}$$

$= 2^{k+1} * \lg(2^k) + 2^{k+1}$

$= 2^{k+1}(\lg(2^k) + 1)$

$= 2^{k+1}(\lg(2^k) + \lg(2))$

$= 2^{k+1}(\lg(2^{k+1}))$

We have proved that $T(2^{k+1}) = 2^{k+1}(\lg(2^{k+1}))$. Therefore, when n is an exact power of 2, the solution of the recurrence is T(n) = n lg n.

3)

a. Disprove. The counter example is: $f_1(n) = n, f_2(n) = n^2, g(n) = n^3$

$f_1(n) = O(g(n))$ and $f_2(n) = O(g(n))$, however, $f_1(n) \neq \Theta(f_2(n))$

b. Prove. Since $f_1(n) = O\big(g_1(n)\big) and\ f_2(n) = O\big(g_2(n)\big)$, there exists c_1, c_2, n_1, $n_2 > 0$ such that

$f_1(n) \leq c_1 g_1(n)$ for $n \geq n_1$, $f_2(n) \leq c_2 g_2(n)$ for $n \geq n_2$. Then,

$f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n) \leq c_1 \max\{g_1(n), g_2(n)\} + c_2 \max\{g_1(n), g_2(n)\}$

$= (c_1 + c_2) \max\{g_1(n), g_2(n)\}$

Let t = $c_1 + c_2$, $n_0 = \max(n_1, n_2)$, then $f_1(n) + f_2(n) \leq t \max\{g_1(n), g_2(n)\}$ for $n \geq n_0$

Therefore, by the definition, $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$

5)

a)

The following code is for insert sort:

```
import random
import time

def insertionSort(arr):
    for j in range(1, len(arr)):

        key = arr[j]

        i = j - 1
        while i >= 0 and arr[i] > key:
            arr[i + 1] = arr[i]
            i = i - 1
        arr[i + 1] = key

def printRunningTime(n):
    arr = random.sample(range(0, 10000), n)
```

```
    startTime = time.time()
    insertionSort(arr)
    print("The running time for n = %s is %.5f seconds" % (n, ((time.time() -
startTime))))

printRunningTime(2000)
printRunningTime(2500)
printRunningTime(3000)
printRunningTime(3500)
printRunningTime(4000)
printRunningTime(5000)
printRunningTime(8000)
printRunningTime(10000)
```

The following code is for merge sort

```
import random
import time

def merge(arr, p, q, r):
    n1 = q - p + 1
    n2 = r - q
    L = [0] * n1
    R = [0] * n2

    for i in range(0, n1):
        L[i] = arr[p + i]

    for j in range(0, n2):
        R[j] = arr[q + 1 + j]

    i = 0
    j = 0
    k = p

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

def mergeSort(arr,p,r):
    if p < r:
        q = int((p+r)/2)
        mergeSort(arr, p, q)
        mergeSort(arr, q+1, r)
        merge(arr, p, q, r)

def printRunningTime(n):
```

```
    arr = random.sample(range(0, 10000), n)
    startTime = time.time()
    n = len(arr)
    mergeSort(arr, 0, n-1)
    print("The running time for n = %s is %.5f seconds" % (n, ((time.time() -
startTime))))

printRunningTime(2000)
printRunningTime(2500)
printRunningTime(3000)
printRunningTime(3500)
printRunningTime(4000)
printRunningTime(5000)
printRunningTime(8000)
printRunningTime(10000)
```

b)

The following running time is for insert sort:

The running time for n = 2000 is 0.18440 seconds

The running time for n = 2500 is 0.28559 seconds

The running time for n = 3000 is 0.43266 seconds

The running time for n = 3500 is 0.58802 seconds

The running time for n = 4000 is 0.77143 seconds

The running time for n = 5000 is 1.19138 seconds

The running time for n = 8000 is 3.03050 seconds

The running time for n = 10000 is 5.40661 seconds

The following running time is for merge sort:

The running time for n = 2000 is 0.01301 seconds

The running time for n = 2500 is 0.01601 seconds

The running time for n = 3000 is 0.02001 seconds

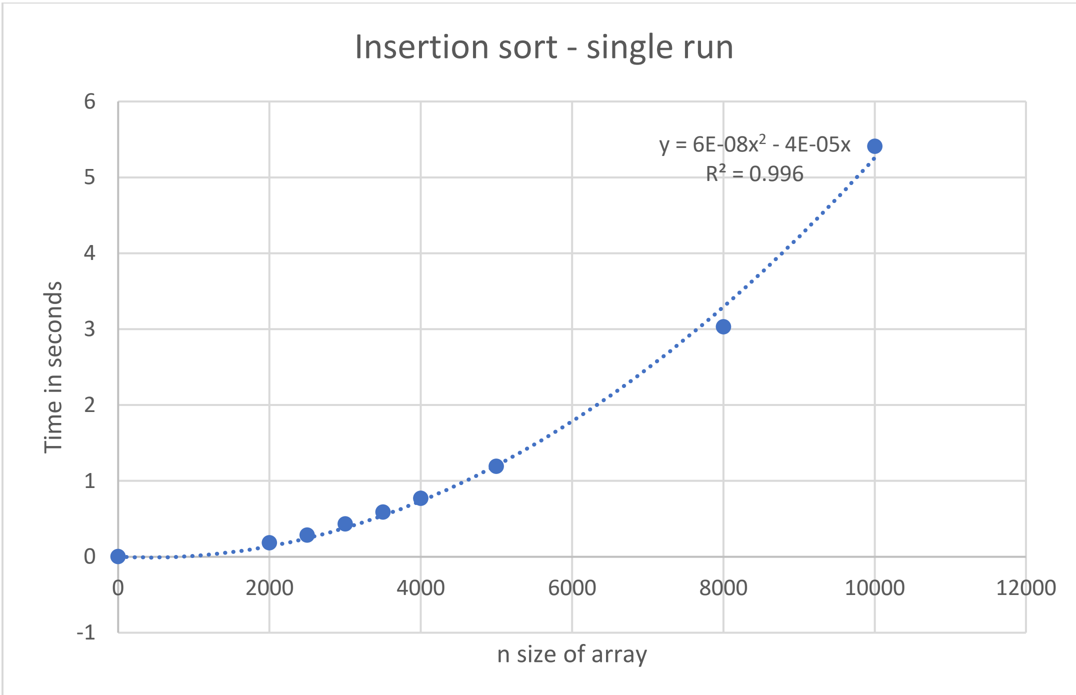
The running time for n = 3500 is 0.02302 seconds

The running time for n = 4000 is 0.02702 seconds

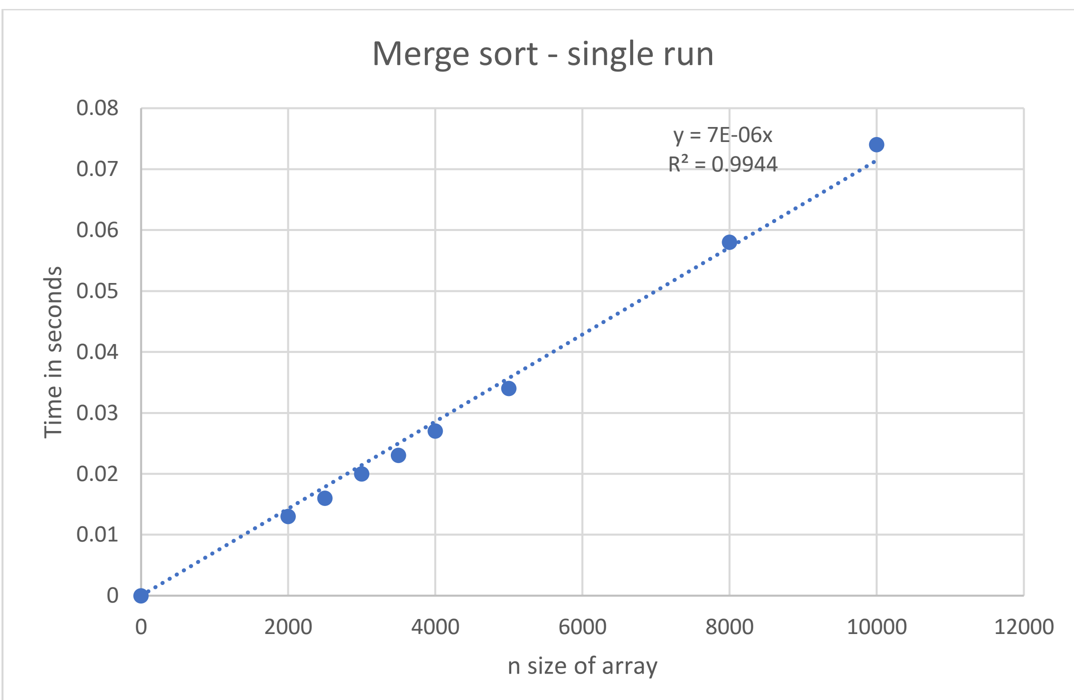
The running time for n = 5000 is 0.03402 seconds

The running time for n = 8000 is 0.05804 seconds

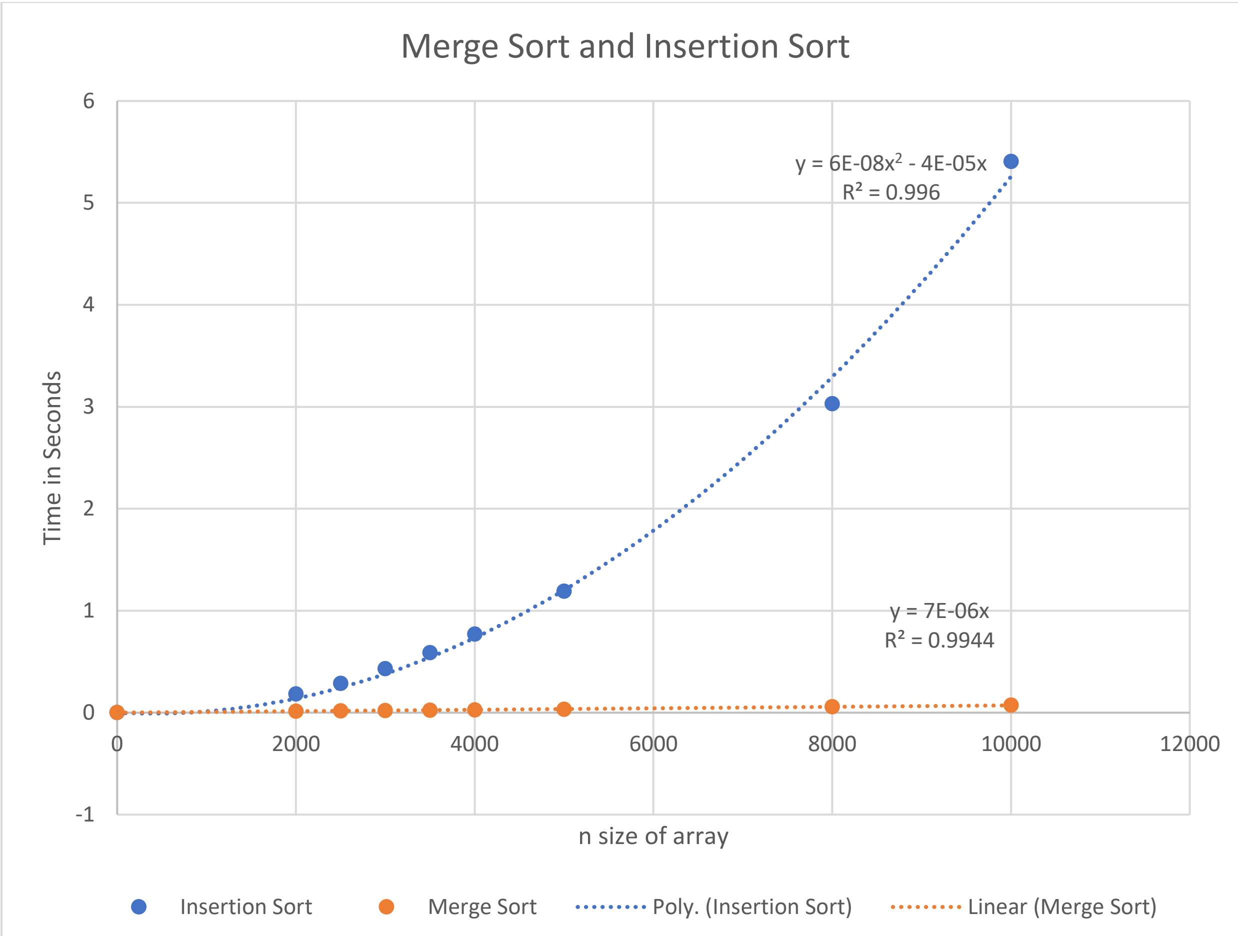
The running time for n = 10000 is 0.07405 seconds c)



Insertion sort - single run
y = 6E-08x² - 4E-05x
R² = 0.996
Time in seconds
n size of array
6
5
4
3
2
1
0
-1
0
2000
4000
6000
8000
10000
12000



Merge sort - single run
y = 7E-06x
R² = 0.9944
Time in seconds
n size of array
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
0
2000
4000
6000
8000
10000
12000



d)

The graph for insertion sort is polynomial, the equation for insertion sort is 6E – 08x^2 - 4E – 05x.

The graph for merge sort is linear, the equation for merge sort is 7E - 06x.

The curve has already drawn in c).

e)

The theoretical average complexity for insertion sort is $O(n^2)$, while in my graph, the equation for insertion sort is 6E – 08x^2 - 4E – 05x. The time complexity for my graph can also considered as $O(n^2)$. Therefore, the theoretical time complexity and the experimental time complexity are the same.

The theoretical average complexity for merge sort is O(n log(n)), while in my graph, the equation for insertion sort is 7E - 06x. The time complexity for my graph can be considered as O(n). Although the theoretical time complexity and the experimental time complexity are slightly different, they are very close.