CS 325 Spring 2018 – HW 2

Sheng Bian

Problem 1:

Algorithm A:

$T(n) = 5T\left(\frac{n}{2}\right) + O(n)$

Using Master theorem, a = 5, b = 2 => $\log_2(5) = 2.322$ => $n^{2.322}$; f(n) = n

Case 1: $f(n) = O(n^{2.322-\varepsilon})\ for\ \varepsilon = 1.322$

Then $T(n) = \Theta(n^{log_2 5})$

Algorithm B:

$T(n) = 2T(n-1) + O(1)$

Using Muster Method, a = 2, b = 1, f(n) = 1 so d = 0. $f(n) = \Theta(n^0)$

If a > 1, $T(n) = \Theta(n^d a^{n/b})$, then $T(n) = \Theta\left(n^0 2^{\frac{n}{1}}\right) = \Theta(2^n)$

Algorithm C:

$T(n) = 9T\left(\frac{n}{3}\right) + O(n^2)$

Using Master theorem, a = 9, b = 3 => $\log_3(9) = 2$ => n^2; f(n) = n^2

Case 2: $f(n) = \Theta(n^2)$

Then $T(n) = \Theta(n^2 \lg n)$

Since $n^2 \lg n < n^{log_2 5} < 2^n$, algorithm C is the best and I select algorithm C.

Problem 2:

a)

```
function ternarySearch(A, target, start, end){
	if (start > end) {
		return false;
	}
	mid1 = start + (end - start) / 3;
	mid2 = end - (end - start) / 3;
	if (A[mid1] == target) {
		return true;
	}
	if (A[mid2] == target) {
		return true;
	}
	if (target < A[mid1]) {
		return ternarySearch(A, target, start, mid1 - 1);
	}
	else if (target > A[mid2]) {
		return ternarySearch(A, target, mid2 + 1, end);
	}
	else {
		return ternarySearch(A, target, mid1 + 1, mid2 - 1);
	}
}
```

b)

$T(n) = T(n/3) + 2$

c)

$T(n) = T(n/3) + 2$

Using Master theorem: a = 1, b = 3 => $\log_3 (1) = 0$ => $n^0 = 1$

Case 2: $f(n) = \Theta(1)$, then $T(n) = \Theta(lgn)$.

Binary search is $\Theta(logn)$ and ternary search is $\Theta(lgn)$. They have the same running time.

Problem 3:

a)

```
function min_and_max(A, start, end){
    if(A.length == 1){
        min = A[0];
        max = A[0];
        return (min, max);
    }
    mid = A.length / 2;
    (min1, max1) =  min_and_max(A, start, mid);
    (min2, max2) =  min_and_max(A, mid+1, end);
    if(min1 <= min2) {
        min = min1;
    } else {
        min = min2;
    }
    If(max1 >= max2) {
        max = max1;
    } else {
        max = max2;
    }
    return (min, max);
```

}

b)

T(n) = 2T(n/2) + 2

c)

T(n) = 2T(n/2) + 2

Using Master theorem: a = 2, b = 2 => $\log_2$ (2) = 1 => n; f(n) = 2

Case 1: $f(n) = O(n)$. Then $T(n) = \Theta(n)$

The running time for iterative algorithm is also $\Theta(n)$. They are the same.

Problem 4:

a)

The base case is an array with two elements. If the first element is larger than the second element, their position will be swapped.

The inductive case: StoogeSort(A[0 ... m - 1]) sorts the first 2/3 of the array correctly. StoogeSort(A[n - m ... n - 1]) sorts the last 2/3 of the array correctly. After these two sorts, the last 1/3 of the array has been correctly sorted and the elements in the last 1/3 is larger than or equal to first 2/3 of A. At last, StoogeSort(A[0 ... m - 1]) sorts the first 2/3 of the array correctly. Therefore, the whole array has been sorted correctly.

b)

No, it wouldn't sort correctly. The counterexample is array [5, 9, 6, 7]. n = 4 so m = floor(2n/3) = 2. For StoogeSort(A[0 ... m - 1]), StoogeSort(A[0…1]). [5,9] has already been sorted and will not be changed. For StoogeSort(A[n - m ... n - 1]), StoogeSort(A[2…3]). [6,7] has already been sorted and will not be changed. Therefore, the array [5, 9, 6, 7] will not be changed.

c)

T(n) = 3T(2n/3) + 3

d)

Using Master theorem: a = 3, b = 3/2 = 1.5, $\log_{(1.5)}3 = 2.71 \Rightarrow n^{2.71}$; f(n) = 3

Case 1: $f(n) = O(n^{2.71-\varepsilon}) for\ \varepsilon = 1.71$

Then, $T(n) = \Theta(n^{log_{1.5}3})$

Problem 5:

b)

The following code is for stooge sort

```
import math
import random
import time

def stoogesort(arr, start, end):
    if end - start == 1 and arr[end] < arr[start]:
        arr[start], arr[end] = arr[end], arr[start]
    if end - start > 1:
        m = math.ceil((end - start + 1) * 2 / 3)
        stoogesort(arr, start, start + m - 1)
        stoogesort(arr, end - m + 1, end)
        stoogesort(arr, start, start + m - 1)

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

printRunningTime(50)
printRunningTime(100)
printRunningTime(200)
printRunningTime(300)
printRunningTime(350)
printRunningTime(450)
printRunningTime(500)
printRunningTime(800)
```

The following running time is for stooge sort:

The running time for n = 50 is 0.01501 seconds

The running time for n = 100 is 0.13209 seconds

The running time for n = 200 is 0.40527 seconds

The running time for n = 300 is 1.20981 seconds

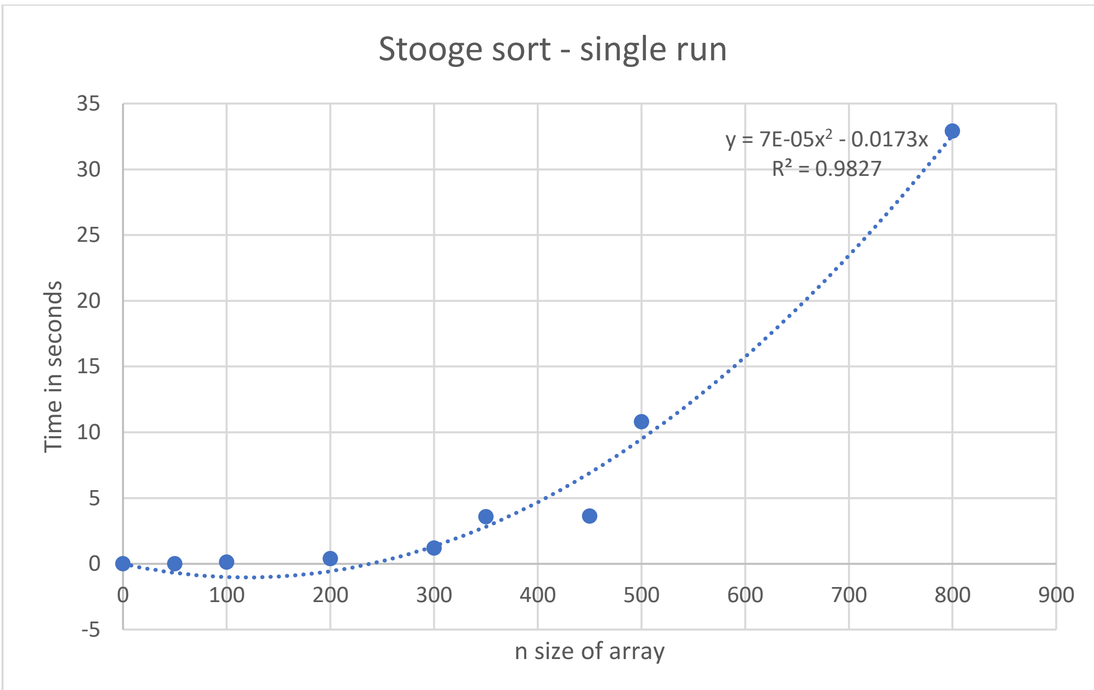
The running time for n = 350 is 3.57940 seconds

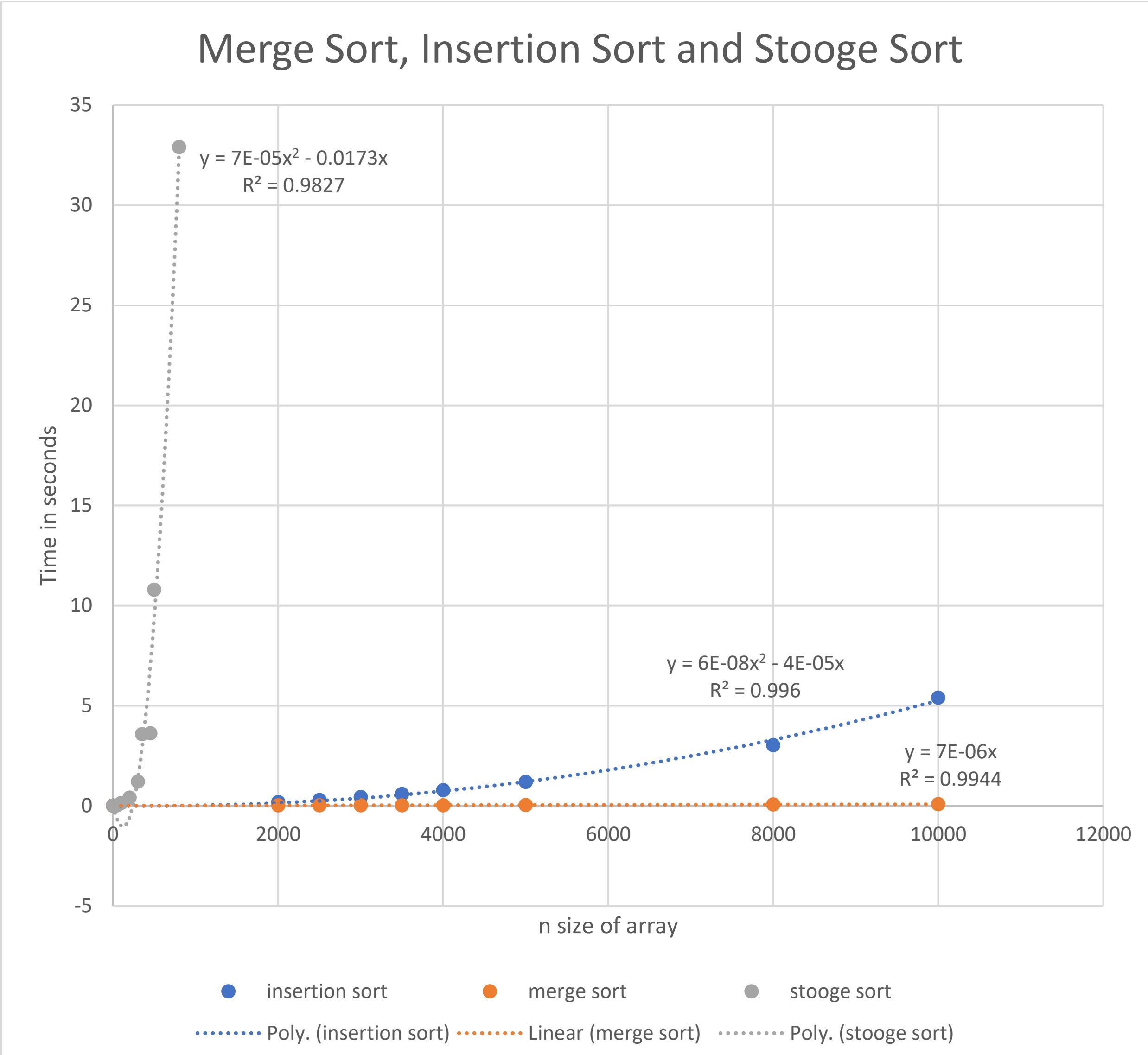
The running time for n = 450 is 3.62374 seconds

The running time for n = 500 is 10.79722 seconds

The running time for n = 800 is 32.91200 seconds

c)





d)

The graph for stooge sort is polynomial, the equation for stooge sort is y = 7E - 05x^2 – 0.0173x.

The curve has already been drawn in c).

The theoretical average complexity for stooge sort is $O(n^{log_{1.5}3})$, while in my graph, the equation for insertion sort is 7E - 05x^2 – 0.0173x. The time complexity for my graph can be considered as O(n^2). Although the theoretical time complexity and the experimental time complexity are slightly different, they are very close.