CS 325 Spring 2019, Homework 2

April 16, 2019

1. Give the asymptotic bounds for T(n) in each of the following recurrences. Make your bounds as tight as possible and justify your answers. *Assume the base cases T(0) = 1 and/or T(1) = 1.*

   a. $T(n) = 3T(n-1) + 1$

   Using the Muster Method:

   - $a = 3 > 1$
   - $b = 1$
   - $f(n) = 1 = O(n^0)$
   - $d = 0$

   For $a > 1$,

   $$T(n) = O(n^d \cdot a^{n/b}) \tag{1}$$
   $$= O(n^0 \cdot 3^{n/1}) \tag{2}$$
   $$= O(3^n) \tag{3}$$

   Since $f(n) = \Theta(n^d)$,

   $$T(n) = \Theta(3^n)$$

   b. $T(n) = T(n-2) + 2$

   Using the Muster Method:

   - $a = 1$
   - $b = 2$
   - $f(n) = 2 = O(n^0)$
   - $d = 0$

   For $a = 1$,

   $$T(n) = O(n^{d+1}) \tag{4}$$
   $$= O(n^{0+1}) \tag{5}$$
   $$= O(n) \tag{6}$$

   Since $f(n) = \Theta(n^d)$,

   $$T(n) = \Theta(n)$$

c. $T(n) = 9T(\frac{n}{3}) + 6n^2$
Using the Master Method:

- $a = 9$
- $b = 3$
- $n^{\log_b a} = n^2$
- $f(n) = 6n^2 = \Theta(n^2)$

This is a Case 2,

$$T(n) = \Theta(n^{\log_b a} \log n) \tag{7}$$
$$= \Theta(n^2 \log n) \tag{8}$$

d. $T(n) = 2T(\frac{n}{4}) + 2n^2$
Using the Master Method:

- $a = 2$
- $b = 4$
- $n^{\log_b a} = n^{1/2}$
- $f(n) = 2n^2 = \Omega(n^{1/2+\epsilon})$ for $\epsilon = 3/2$

This is a Case 3. We need to satisfy Regularity Condition,

$$af(\frac{n}{b}) \leq cf(n) \tag{9}$$
$$2f(\frac{n}{4}) \leq cf(n) \tag{10}$$
$$4\left(\frac{n}{4}\right)^2 \leq c(2n^2) \tag{11}$$
$$\frac{n^2}{4} \leq c(2n^2) \tag{12}$$
$$\tag{13}$$

Regularity Condition is satisfied for $c = \frac{1}{8}$. Therefore,

$$T(n) = \Theta(f(n)) \tag{14}$$
$$= \Theta(n^2) \tag{15}$$

2. The Ternary Search Algorithm is a modification of the binary search algorithm that splits the input not into two sets of almost-equal sizes, but into three sets of sizes approximately one-third.

   a. Verbally describe and write pseudo-code for the Ternary Search Algorithm.

   Ternary Search is a divide and conquer algorithm that can be used to find an element in an array. It is similar to binary search where we divide the array into two parts but in this algorithm, we divide the given array into three parts and determine which has the key (searched element). We can divide the array into three parts by taking the left third and the right third which can be calculated as shown below. Initially, l and r will be equal to 0 and n-1 respectively, where n is the length of the array.

```
left_third = l + (r-l)/3
right_third = r - (r-l)/3
```

   Steps to perform Ternary Search:

   1. First, we compare the key with the element at mid1. If found equal, we return mid1.
   2. If not, then we compare the key with the element at mid2. If found equal, we return mid2.
   3. If not, then we check whether the key is less than the element at mid1. If yes, then recur to the first part.
   4. If not, then we check whether the key is greater than the element at mid2. If yes, then recur to the third part.
   5. If not, then we recur to the second (middle) part.

   Here is the pseudocode for Ternary Search,

```
ternary-search(array A, query x)
    if length(A) = 0
        return false
    else
        if x = left third element of A
            return true
        else if x = right third element of A
            return true
        else
            if x < left third element of A
                B = left third of A
            else if x > right third element of A
                B = right third of A
            else
                B = middle third of A
            return ternarh-search(B,x)
```

b. Give the recurrence for the Ternary Search Algorithm
$T(n) = T\left(\frac{n}{3}\right) + c$

c. Solve the recurrence to determine the asymptotic running time of the algorithm. How does the running time of the Ternary Search Algorithm compare to that of the Binary Search Algorithm.
Using the Master Method:

- $a = 1$
- $b = 3$
- $n^{\log_b a} = n^0 = 1$
- $f(n) = c = \Theta(1)$

This is a Case 2,

$$T(n) = \Theta(n^{\log_b a} \log n) \tag{16}$$
$$= \Theta(logn) \tag{17}$$

The running time of Ternary Search Algorithm is similar to that of the Binary Search Algorithm.

3. Consider the following pseudocode for a sorting algorithm.

```
StoogeSort(A[0 ... n-1])
    if n = 2 and A[0] > A[1]
        swap A[0] and A[1]
    else if n > 2
        m = ceiling(2n/3)
        StoogeSort(A[0 ... m-1])
        StoogeSort(A[n-m ... n-1])
        StoogeSort(A[0 ... m-1])
```

a. State a recurrence for the number of comparisons executed by STOOGESORT.
$T(n) = 3T\left(\frac{2n}{3}\right) + c$

b. Solve the recurrence to determine the asymptotic running time.
Using Master Method:

- $a = 3$
- $b = 3/2$
- $n^{\log_b a} = n^{2.71}$
- $f(n) = c = O(n^{2.71-\epsilon})$ for $\epsilon = 2.71$

This is a Case 1,

$$T(n) = \Theta(n^{\log_b a}) \tag{18}$$
$$= \Theta(n^{2.71}) \tag{19}$$

4. Stooge Sort Running Time Analysis

   a. Implement STOOGESORT from Problem 3 to sort an array/vector of integers. Implement the algorithm in the same language you used for the sorting algorithms in HW1. Your program 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 (like in HW1). The output will be written to a file called "stooge.out".

   b. Now that you have verified 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 the file data.txt and sorting, you will now generate arrays of size n containing random integer values from 0 to 10,000 to sort. Use the system clock to record the running times of each algorithm for ten different values of n for example:

      n = 5000, 10000, 15000, 20000, . . . , 50000

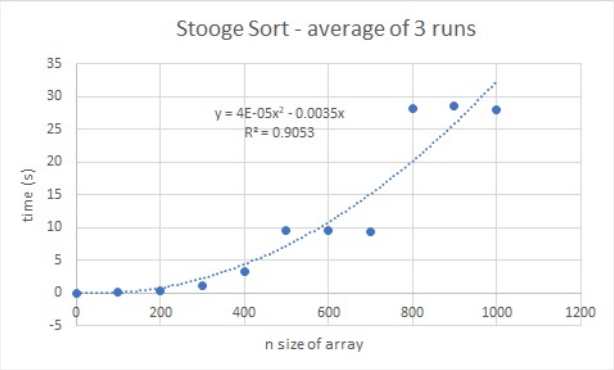
      You may need to modify the values of n if an algorithm runs too fast or too slow to collect the running time data (do not collect times over a minute). Output the array size n and time to the terminal. Name the new program stoogeTime.

   c. Collect running times - Collect your timing data on the engineering server. You will need at least eight 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. Create a table of running times for each algorithm.

n	Stooge Sort
0	0
100	0.12
200	0.35
300	1.06
400	3.15
500	9.44
600	9.53
700	9.38
800	28.26
900	28.58
1000	28.02

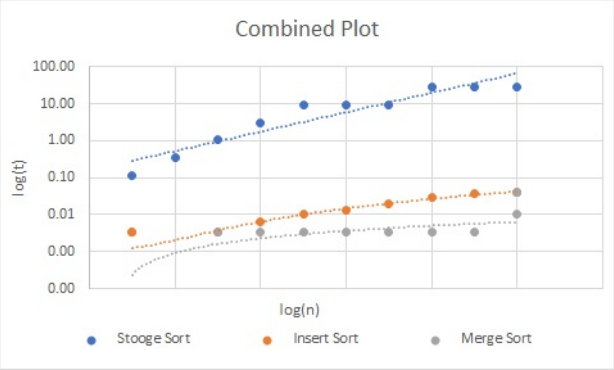
Running Time of Stooge Sort.

d. Plot data and fit a curve - 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. What type of curve best fits each data set? Give the equation of the curves that best "fits" the data and draw that curves on the graphs.



Stooge Sort.

e. Combine - Plot the data from both algorithms together on a combined graph. If the scales are different you may want to use a log-log plot.



Combined Plot.

f. Comparison - Compare your experimental running times to the theoretical running times of the algorithms? Remember, the experimental running times were the "average case" since the input arrays contained random integers.

The experimental running time of Stooge Sort closely resembles the theoretical running time. Stooge Sort has a theoretical average case of $O(n^3)$. Insertion Sort has a theoretical average case of $O(n^2)$. Merge Sort has a theoretical average case of $O(n \log n)$. n^3 grows a lot faster than n^2 and $n \log n$.