Name: Brian Allison
Date: 9/30/2017
Course: CS325 Online
Description: Assignment 2

Problem 1: *(5 points)* 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) $T(n) = 2T(n-2) + 1$

Solved with the muster theorem:

a = 2, b = 2, f(n) = n^0 = 1, d = 0

T(n) = $\Theta(n^0 2^{n/2}) = \Theta(2^{n/2})$

This answer makes sense because the rate of increase is dominated by the number of leaves.

b) $T(n) = T(n-1) + n^3$

Solved with the muster theorem:

a = 1 b = 1, f(n) = n^3, d = 3

T(n) = $\Theta(n^{3+1}) = \Theta(n^4)$

This answer makes sense, because the rate of increase is dominated by f(n)

c) $T(n) = 2T\left(\frac{n}{6}\right) + 2n^2$

Solved with the master theorem

a = 2, b = 6, f(n) = $2n^2$

$\log_b a = \log_6 2$

f(n) = $\Omega(n^{\log_6 2})$

T(n) = **$\Theta(n^2)$**

This answer makes sense because the rate of increase is dominated by f(n)

Problem 2: *(5 points)* The quaternary search algorithm is a modification of the binary search algorithm that splits the input not into two sets of almost-equal sizes, but into four sets of sizes approximately one-fourth.

a) Verbally describe and write pseudo-code for the quaternary search algorithm.

Description:

The quaternary search algorithm checks for matches at the midpoints of each quarter section of the list. If no match is found, it then tests to see if the search value is less than any of the midpoints or the last index. If the search value is less than one of the midpoints or the last index, quaternary search is called again on the subset between two significant points of the list, significant points being the beginning (included in subset), a quarter's midpoint, or the end of the list (included in subset). If a match is found at any point, the function returns the index of the matching value. If the subset being searched contains less than 4 terms, the function checks

to see if the value exists in the final subset and, if not, returns a -1 to indicate that the value was not found.

Note: Although I implemented the algorithm quite differently, I used http://ijirt.org/master/publishedpaper/IJIRT143908_PAPER.pdf as reference to understand what quaternary search does.

Pseudo-code:

```
q_search (Array, key)
        low = Array[0]
        high = Array.length – 1
        location = qsearch2 (Array, key, low, high)
        return location

q_search2 (Array, key, low, high)
        mid2 = high/2
        mid1 = mid2/2
        mid3 = mid2 + high/2

        if mid1, mid2, or mid3 == key
                return index of matching value
        if (high – low >= 2)  --- if not, skip to val not found
                 if the key is less than value at mid1
                        qsearch2(Array, key, low, mid1 – 1)
                else if the key is less than val at mid2
                        qsearch2(Array, key, mid1 + 1, mid2 -1)
                else if the key is less than val at mid3
                        qsearch2(Array, key, mid2 + 1, mid3 -1)
                else if the key is less than high
                        qsearch2(Array, key, mid3 + 1, high)
                else if the key is greater than high

        val not found: return -1
```

b) Give the recurrence for the quaternary search algorithm

   Ans: T(n) = T(n/4) + c

c) Solve the recurrence to determine the asymptotic running time of the algorithm. How does the running time of the quaternary search algorithm compare to that of the binary search algorithm.

   Ans:

$T(n) = aT(n/b) + f(n)$

$a = 1, b = 4, f(n) = c$

$\log_a b = \log_4 1 = 0$

$c = \Theta(n^0)$ because both are constants so we use case 2

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

Binary search is $\Theta(\lg n)$ as well, so they run at the same time. It would be tempting to say that quaternary search is $T(n) = \Theta(\log_4 n)$, because $\log_4 n$ is the number of levels of the recursion tree for quaternary search. But, for asymptotic analysis, the base of log does not matter because the base can be changed simply by dividing by a constant.

Problem 3: *(5 points)* Design and analyze a **divide and conquer** algorithm that determines the minimum and maximum value in an unsorted list (array).

a) Verbally describe and write pseudo-code for the min_and_max algorithm.

Description:

The min_and_max algorithm returns an array of length 2, containing the min value and the max value of the unordered array that is passed in. It does this by recursively calling itself twice within each call to the function. The first call passes in the first half of the array and returns an array of length two, the first value of which represents the lowest value and the second representing the highest value within that half of the array. The second call does the same. As the tree of calls gets deeper and deeper, eventually we hit the base case, which returns an array of length 2, each value being the only value in the array that was passed in. Since it's the only value in the array that was passed in, it represents both the lowest and highest value. To continue back up the tree, the first (smallest) and second (largest) item in the two resulting arrays are compared and a single array with the smallest and largest resulting value is returned. This continues until we are left with the first call finishes, returning an array of length 2 with the smallest and largest value.

Pseudocode:

```
min_max (array)
        if array.length == 1
                return [array[0], array[0]]

        mm1 = min_max(array[0]…(array.length/2)-1)
        mm2 = min_max(array.length/2…array.length-1)

        if (mm2[0] < mm1[0])
                mm1[0] = mm2[0]
```

```
if (mm2[1] > mm1[1])
        mm1[1] = mm2[1]

return mm1
```

b) Give the recurrence.

Ans: $T(n) = 2(n/2) + c$

c) Solve the recurrence to determine the asymptotic running time of the algorithm. How does the theoretical running time of the recursive min_and_max algorithm compare to that of an iterative algorithm for finding the minimum and maximum values of an array.

Ans:
$T(n) = aT(n/b) + f(n)$
$a = 2, b = 2, f(n) = c$
$\log_a b = \log_2 2 = 1$
$c = O(n^1)$
$T(n) = \Theta(n)$

An iterative algorithm for finding the minimum and maximum would have to go through the array a single time, comparing the current min and max to each value in the array. With 2 comparisons for each element of the array, this would result in $T(n) = 3n$. Since, the constant 3, is not important, the result is the same as the divide and conquer algorithm: $T(n) = \Theta(n)$

Problem 4: *(5 points)* 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) Verbally describe how the STOOGESORT algorithm sorts its input.

Ans:

If there are two elements in the array and the first element is greater than the second element, then the first two elements are swapped. Otherwise, if there are more than 2 elements, StoogeSort recursively calls itself on the first 2/3 of the array, then the last 2/3s of the array, then again on the first 2/3 of the array. Each call uses the ceiling method of division, and recursively sorts the subset of the array that is passed in. After the third call in the tree's root, the array is sorted. The third call Stoogesort is necessary because, after the second call, the first

2/3 of the array may contain unsorted elements if the last 1/3 of the array has elements that belong in the first 2/3s of the array. Each call to StoogeSort (at the same level of the tree) contains the middle 1/3 of the array elements. Intuitively, this does not seem like an efficient method of sorting an array.

b) Would STOOGESORT still sort correctly if we replaced k = ceiling(2n/3) with k = floor(2n/3)? If yes prove if no give a counterexample. (Hint: what happens when n = 4?)
Ans:
No. It would not sort correctly.
Counter Example:
Let the array be [6,2,4,1].
StoogeSort is called on the initial array
n > 2, so we execute the else block
k = floor((2*4)/3) = 2
call StoogeSort on the array from index 0 through index k – 1 = 1, which is [6, 2]
n = 2, A[0] > A[1], so the elements are swapped and we now have [2, 6]
Back at the root level our current array is now [2, 6, 4, 1]
call StoogeSort on the array from index n – m = 2 through index k – 1 = 3 which is [4, 1]
n = 2, A[0] > A[1], so the elements are swapped and we now have [1, 4]
Back at the root level our current array is now [2, 6, 1, 4]
call StoogeSort on the array from index 0 through index k – 1 = 1, which is [2, 6]
n = 2, but A[0] < A[1], so nothing else is done
Back at the root level, the execution is complete, but the array is now [2, 6, 1, 4]
Clearly, the array is not fully sorted. Therefore,StoogeSort will not sort correctly if k = ceiling(2n/3) was replaced with k = floor(2n/3).

c) State a recurrence for the number of comparisons executed by STOOGESORT.
$T(n) = 3T(2n/3) + c$

d) Solve the recurrence to determine the asymptotic running time.

$T(n) = aT(n/b) + f(n)$
$a = 3, b = 3/2, f(n) = c$
$\log_a b = \log_3 3/2 = 2.7$ (approx.)
$c = O(n^{2.7})$
$T(n) = \Theta(n^{2.7})$

Problem 5: *(10 points)*

a) Implement STOOGESORT from Problem 4 to sort an array/vector of integers. Implement the algorithm in the same language you used for the sorting algorithms in HW 1. 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 HW 1). The output will be written to a file called “stooge.out”.

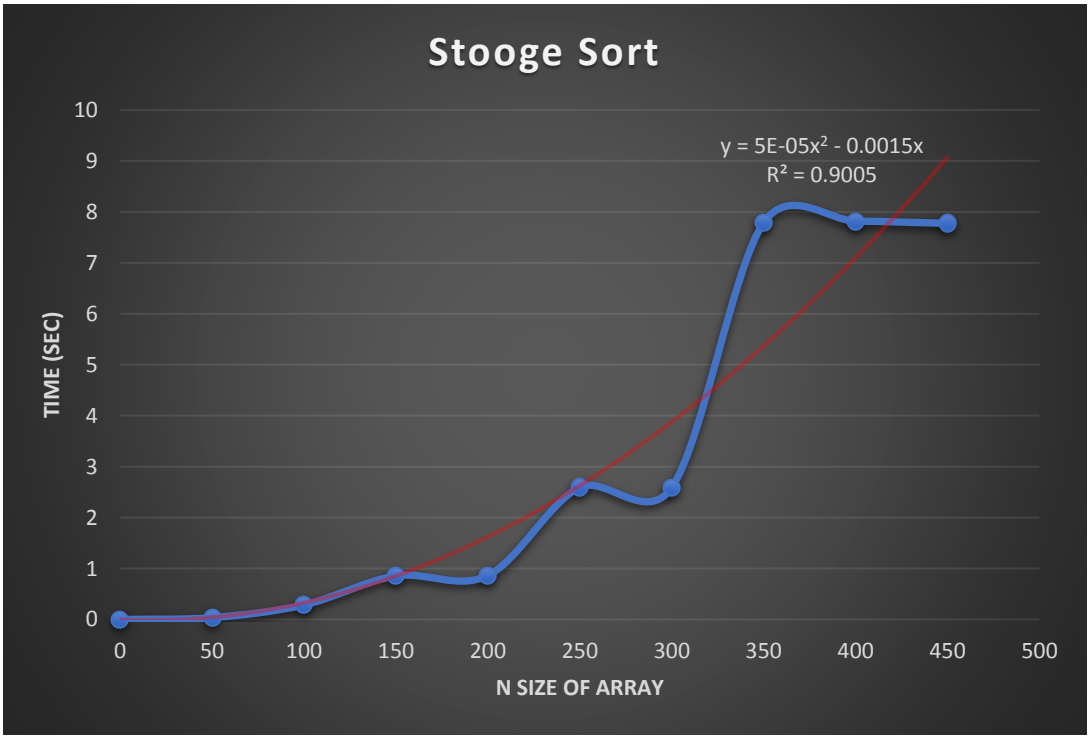
Submit a copy of all your code files and a README file that explains how to compile and run your code in a ZIP file to TEACH. We will only test execution with an input file named data.txt.

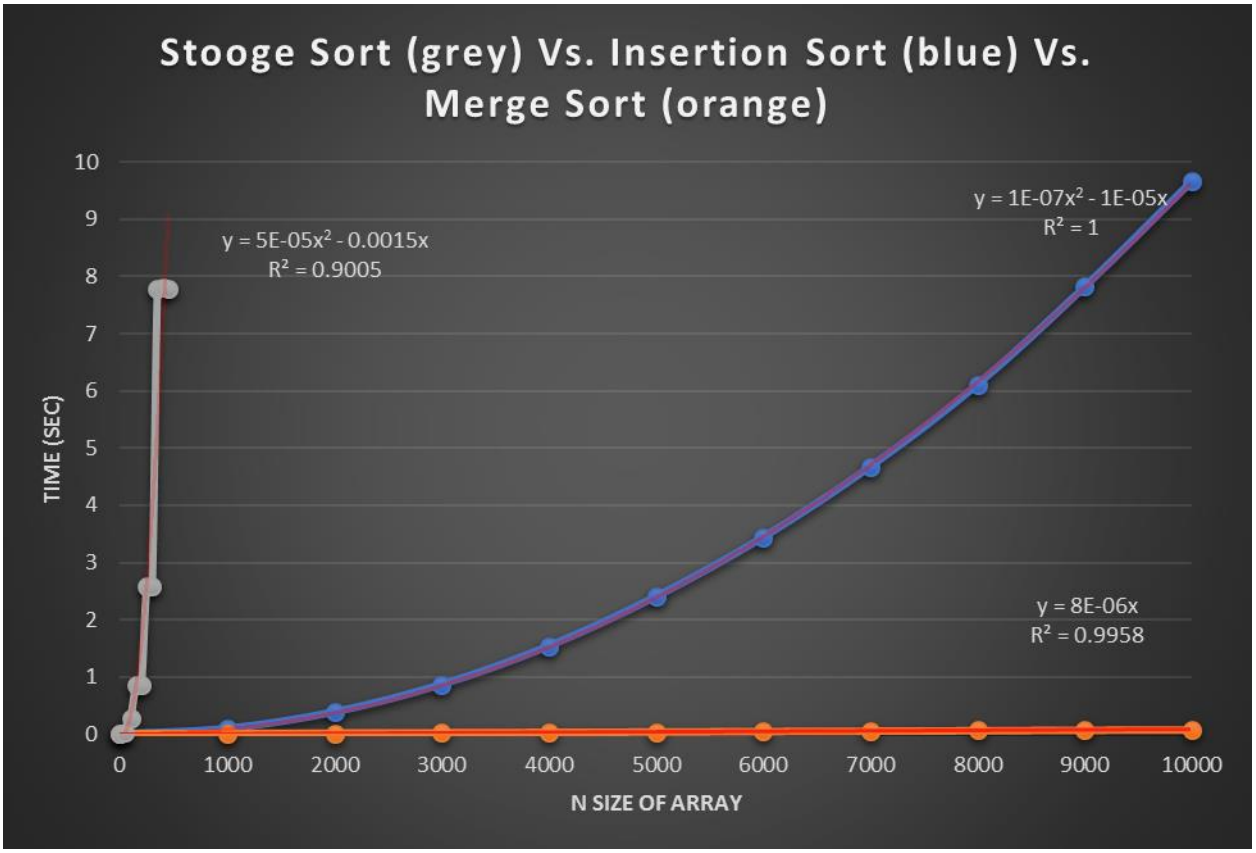
b) Now that you have proven that your code runs correctly using the data.txt input file, you can modify the code to collect running time data. Instead of reading arrays from a file to sort, you will now generate arrays of size n containing random integer values and then time how long it takes to sort the arrays. We will not be executing the code that generates the running time data so it does not have to be submitted to TEACH or even execute on flip. Include a "text" copy of the modified code in the written HW submitted in Canvas. You will need at least seven values of t (time) greater than 0. If there is variability in the times between runs of the algorithm you may want to take the average time of several runs for each value of n.

Ans: Code is included at the bottom of the document.

c) 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. Also plot the data from Stooge algorithm together on a combined graph with your results for merge and insertion sort from HW1.

Ans:





d) What type of curve best fits the StoogeSort data set? Give the equation of the curve that best "fits" the data and draw that curve on the graphs of created in part c). How does your experimental running time compare to the theoretical running time of the algorithm?

Ans:

A quadratic curve is the best fit for StoogeSort.
The equation is $f(n) = 5E\text{-}05x^2 - 0.0015x$
$R^2 = 0.9005$

The experimental running time seems to be very close to the theoretical running time. Stooge Sort is $T(n) = \Theta(n^{2.7})$. Overall, The time required is definitely increasing at a quadratic rate, but the rate of increase seems to change between data points that are in close proximity to each other. This sorting algorithm is clearly much worse than Insertion Sort and Merge Sort.

Code for time analysis of stooge sort:

```python
# Name: Brian Allison
# Date: 10/8/2017
# Course: CS325 Online
# Description: Assignment 2, stooge_sort

import time
import random


# sorts an array of ints in O(n^2.7) time
# below code was written with the hw2 pseudo code used as a reference
def stooge_sort(array):
    n = len(array)
    if n == 2:
        if array[0] > array[1]:
            array[0], array[1] = array[1], array[0]
        return array
    elif n > 2:
        r = (2 * n) % 3
        if r != 0:
            m = 2 * n // 3 + 1
        else:
            m = 2 * n // 3
        fthird = stooge_sort(array[:int(m)])
        array = fthird + array[int(m):]
        lthird = stooge_sort(array[int(n - m):int(n)])
        array = array[:int(n - m)] + lthird
        fthird = stooge_sort(array[:int(m)])
        array = fthird + array[int(m):]

    return array


# below running time code was written based on canvas lecture for hw1 problem 5
tot = 0
ave = 0
rand_array = []

file_out = open('stooge_time.txt', 'w')
for k in range(0, 500, 50):
    file_out.write(str(k) + ' , ')
    print(k, end=' , ')
    tot = 0
    ave = 0

    for l in range(3):
        rand_array = []
        for m in range(k):
            random_number = random.randint(0, 1000)
            rand_array.append(random_number)

        start_time = time.time()
        stooge_sort(rand_array)
        end_time = time.time()

        elapsed_time = end_time - start_time
        tot = tot + elapsed_time
        file_out.write(str(elapsed_time) + ' , ')
        print(elapsed_time, end=' , ')

    ave = tot/3
    print("average: " + str(ave))
    file_out.write("average: " + str(ave) + '\n')
```

```
file_out.close()
```