CS 325-400 Fall 17
HW 2
10/1/17

Peter Moldenhauer
moldenhp@oregonstate.edu

1. Give the asymptotic bounds for T(n):

   a. $T(n) = 2T(n-2) + 1$ is $\Theta(2^{n/2})$
   [Muster method] a = 2, b = 2, $f(n) = n^0 = 1$ so d = 0
   Since a > 1, We have the case of: $O(n^d a^{n/b}) = O(n^0 2^{n/2}) = O(2^{n/2})$
   Therefore, we can conclude that T(n) is $\Theta(2^{n/2})$

   b. $T(n) = T(n-1) + n^3$ is $\Theta(n^4)$
   [Muster method] a = 1, b = 1, d = 3
   Since a = 1, We have the case of: $O(n^{d+1}) = \Theta(n^4)$

   c. $T(n) = 2T(n/6) + 2n^2$ is $\Theta(n^2)$
   [Master method] a = 2, b = 6, $\log_6 2 = 0.387$
   compare $n^{0.387}$ with $f(n) = 2n^2$
   We know that f(n) is bounded below by $n^{0.387}$ because 0.387 is less than 1 (and $2n^2$ would be bounded below by n^1)
   Therefore, we have case 3 and need to check the regularity condition:
   $af(n/b) <= cf(n)$ for some $c < 1$ → $2*(2n^2/6) <= c*2n^2$
   c = 1/3 is a solution (c < 1)
   Therefore $T(n) = \text{Theta}(f(n)) = \text{Theta}(2n^2) = \Theta(n^2)$

2. 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:

   The quaternary search algorithm is very similar to binary search. However, instead of creating two subarrays (as in binary search), we would create four subarrays. Once the initial array is split up into four subarrays, three elements are obtained:

   Initial array:
   |----------|
   Four subarrays that produce 3 elements:
   |----------1----------2----------3----------|

   The value that is being searched on is them compared to those three elements. If any of these three elements match the searched value then the position of that element is returned (or also in some cases “true” is returned saying that the searched value is in the initial array). If any of the

three elements do not match the searched value, then the process is repeated between any of the four subarrays. Based on the value being searched on will determine which of the four subarrays to repeat the process on. This process is then repeated until the searched value is either found or determined not to be contained in the initial array. Similar to binary search, quaternary search requires the initial array to be sorted.

Pseudocode for quaternary search:

```
QuaternarySearch(array, value)
        if arraySize = 0
                return -1
        low = 0
        element1 = (arraySize/4) – 1
        element2 = (arraySize/2) – 1
        element3 = (arraySize*3/4) – 1
        high = arraySize – 1

        elementList = [element1, element2, element3]

        try
                array[x] for x in elementList.index(value)
                return 1
        except
                pass

        if value < array[element1]
                return QuaternarySearch(array[low:element1, value)
        else if value < array[element2]
                return QuaternarySearch(array[element1:element2, value)
        else if value < array[element3]
                return QuaternarySearch(array[element2:element3, value)
        else if value < array[high]
                return QuaternarySearch(array[element3:high, value)
        else
                return -1
```

b. Give the recurrence for the quaternary search algorithm.

$T(n) = T(n/4) + c$

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.

$T(n) = T(n/4) + c$ is $\Theta(\lg n)$

[Master method] a = 1, b = 4, f(n) = c, $\log_4 1 = 0$
Compare $n^0 = 1$with f(n) = c = 1
Since both are 1, we have case 2: f(n) = Theta($n^{\log_b a}$)
Therefore, T(n) = Theta($n^{\log_b a}$ lg n) = Theta(1 lg n) = Theta(lg n)

The running time of the quaternary search algorithm is the same as that of the binary search algorithm. Both of these algorithms have the asymptotic running time of Theta(lg n). The recurrence is different for the two algorithms but the asymptotic running time for them is the same.

3. 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 pseudocode for the min_and_max algorithm.

    The min_and_max algorithm could be described by simply running merge sort on the array and then returning the first and last elements of the array. Merge sort is a divide and conquer algorithm that would sort the unsorted input array. This would then make it easy to simply return the smallest and largest elements of the array.

    However, if using a sorting algorithm is not allowed then another description of a min and max algorithm could be described as follows: If the array only has 1 element, then it sets both min and max to the value of that one element and then returns min and max. (Note: for all of these, the min and max will be returned as a list of two elements, the min as the first element and the max as the second element) If the array has two elements then it will check which of the two is larger/smaller and then return the appropriate corresponding values for min and max. Lastly, if there are more than two elements in the array then the algorithm will first split the array in half (this is accomplished by calculating the size of the array / 2). After splitting the array into two subarrays, it will recursively call itself two times – once for the left side and once for the right side. It will then compare the min and max from both of the sides and return the appropriate values for min and max of the entire array.

    Pseudocode for min_and_max algorithm:

```
min_and_max(array)
        if arraySize = 1
                min = array[0]
                max = array[0]
                return min, max
        else if arraySize = 2
                if array[0] < array[1]
                        min = array[0]
```

```
                max = array[1]
                return min, max
        else
                min = array[1]
                max = array[0]
                return min, max
else
        middle = arraySize/2
        high = arraySize – 1
        (minLeft, maxLeft) = min_and_max(array[0:middle])
        (minRight, maxRight) =
                min_and_max(array[middle+1:high])
        if maxLeft < maxRight
                max = maxRight
        else
                max = maxLeft
        if minLeft < minRight
                min = minLeft
        else
                min = minRight
        return min, max
```

b. Give the recurrence.

T(n) = 2T(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.

T(n) = 2T(n/2) + c is Theta(n)
[Master method] a = 2, b = 2, f(n) = c, $\log_2 2 = 1$
Compare n^($\log_2 2$) with f(n), n^($\log_2 2$) = n and f(n) = 1
Therefore, we have case 1 so T(n) = Theta(n^($\log_2 2$)) = Theta(n)

The theoretical running time of the recursive min_and_max algorithm is (surprisingly) similar to that of an iterative algorithm for finding the minimum and maximum values of an array. An iterative algorithm would have a running time of T(n) = Theta(n) + 2. This is because we would make two comparisons with each step. We can also just “simplify” this as having a running time of Theta(n) because the 2 makes little to no difference with the total running time because it is just a constant. The Theta(n) running time of the iterative algorithm is the same complexity for the recursive algorithm (as shown above).

4. 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.

The STOOGESORT algorithm sorts its input in the following manner: The algorithm inputs an array of length n. If the array length is less than 2 then the algorithm does not sort (it does nothing). If the array length is 2 then it will check if the first element is greater than the second element. If so, it will swap these two elements in the array. This can be viewed as the "base case". If the array length is greater than 2, then it will first calculate the position of the element which is two thirds the length of the initial array. Note, it does this by taking the "ceiling" of 2/3*n so if a decimal value is the result then it will be rounded up. Once this "two thirds position" is calculated, the algorithm will recursively call StoogeSort on the first two thirds of the array, the last two thirds of the array and then the first two thirds of the array again. It will stop making these recursive calls of itself once the base case (array length of 2) is reached for each given call. In short, the algorithm will repeatedly make subarrays of the initial array until each of the subarrays contain only two elements. It should be noted that the algorithm "overlaps" with the first and last two thirds portion of the array. Due to this overlap, the recursive call to sort the last two thirds of the array may interfere with the previous ordering of the first two thirds of the array. This is why the recursive call to sort the first two thirds portion of the array is called a second time.

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?)

No, STOOGESORT would not still sort correctly if we replaced k = ceiling(2n/3) with k = floor(2n/3). We can see this clearly with an array of size 4 (as mentioned in the hint). In this case with an array of size 4, we would skip over the base case (since its greater than n = 2) and jump to calculating the "two thirds position" and making the recursive calls. However, when using "floor" instead of "ceiling", we could fall into the situation where there would be no overlap between the first two thirds

portion and the last two thirds portion. Then, with the starting array size of 4, the elements at index 1 and 2 would not ever get compared to each other. This would result in the initial array potentially not being properly sorted at the end.

Here is a specific example: The starting array size will be [6, 4, 2, 8]. As a result, m = floor(2n/3) = 2(4)/3 = 8/3 = 2. The first recursive call will create the array of [6, 4]. This elements will get swapped so now the array becomes [4, 6, 2, 8]. The second recursive call will create the array of [2, 8]. These elements will remain in sorted order with the total array of [4, 6, 2, 8] still. Finally, the third recursive call will create the array of [4, 6] again which needs no sway and then the algorithm finishes. However, since there was no overlap between both of the two thirds portions, the final array is not sorted properly.

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.

$T(n) = 3T(2n/3) + c$ is $\Theta(O(n^{2.71}))$
[Master method] a = 3, b = 3/2 (this is because 2n/3 is n * 2/3 or also n / 3/2), f(n) = c = 1
$\text{Log}_{3/2}3 = 2.71$
Compare $n^{2.71}$ with f(n) → $f(n) = O(n^{2.71})$
Therefore, $T(n) = 3T(2n/3) + c$ is $\Theta(O(n^{2.71}))$

5.

a. Implementation of the stoogesort algorithm was submitted separately to TEACH.

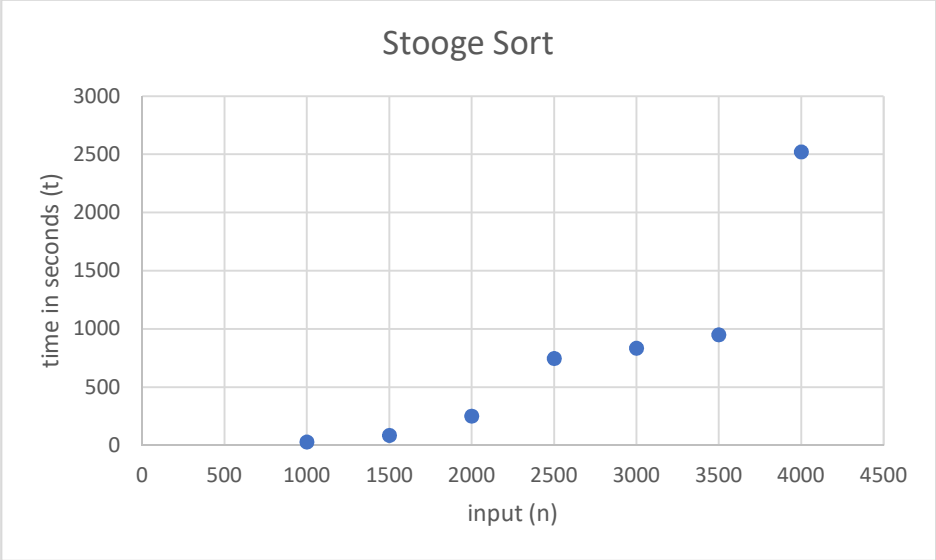
b. “Text” copy of modified code and collected running time data below…

```
#############################################################################################
# Name: Peter Moldenhauer
# Class: CS 325-400
# Date: 10/4/17
# Description: This program implements the stooge sort algorithm.
# It reads in input data from data.txt and outputs data to stooge.out
#
# I used the following the website as a resource: https://en.wikipedia.org/wiki/Stooge_sort
#############################################################################################

#used for random number generation
import random

# used to get programs execution time
import time
startTime = time.time()

# stoogeSort function
def stoogeSort(theArray, i, j):
    if theArray[j] < theArray[i]:
        theArray[i], theArray[j] = theArray[j], theArray[i]
    if (j - i + 1) > 2:
        m = (j - i + 1) / 3
        stoogeSort(theArray, i, j-m)
        stoogeSort(theArray, i+m, j)
        stoogeSort(theArray, i, j-m)
    return theArray

# generate an array of size n containing random integer values from 0 to 10000
# note: I will change the value of n each time I execute the program, n = 1000 in this example
n = 1000

# create the array of random integer values
theArray = []
for i in range(n):
    theArray.append(random.randrange(0, 10001, 1))

# call the stoogeSort function to sort the array
sortedArray = stoogeSort(theArray, 0, len(theArray) - 1)

# print the programs execution time
print("--- %s seconds ---" % (time.time() - startTime))
```

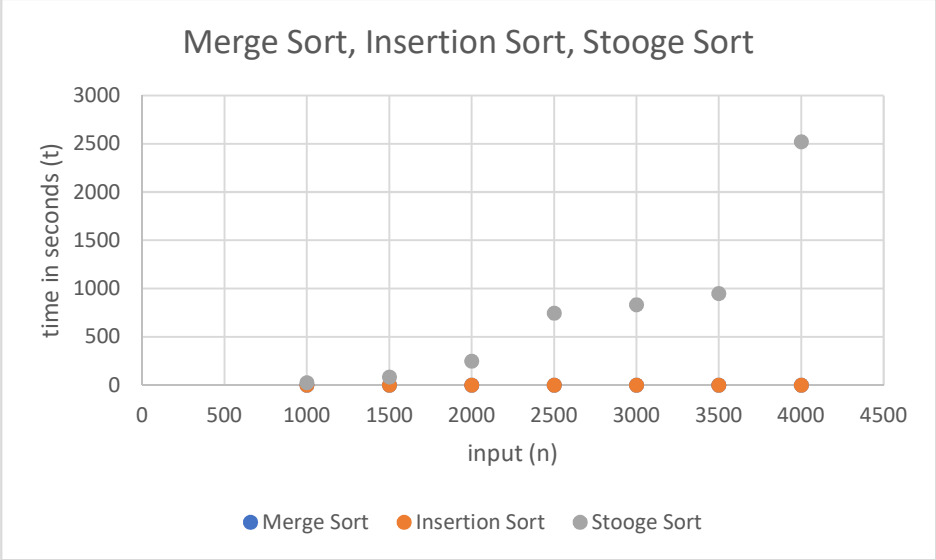
n value	t (time in seconds)
1000	26.78
1500	84.8
2000	248.9
2500	746.6
3000	834.4
3500	948.1
4000	2519.4

c. Graphs of collected running time data below…

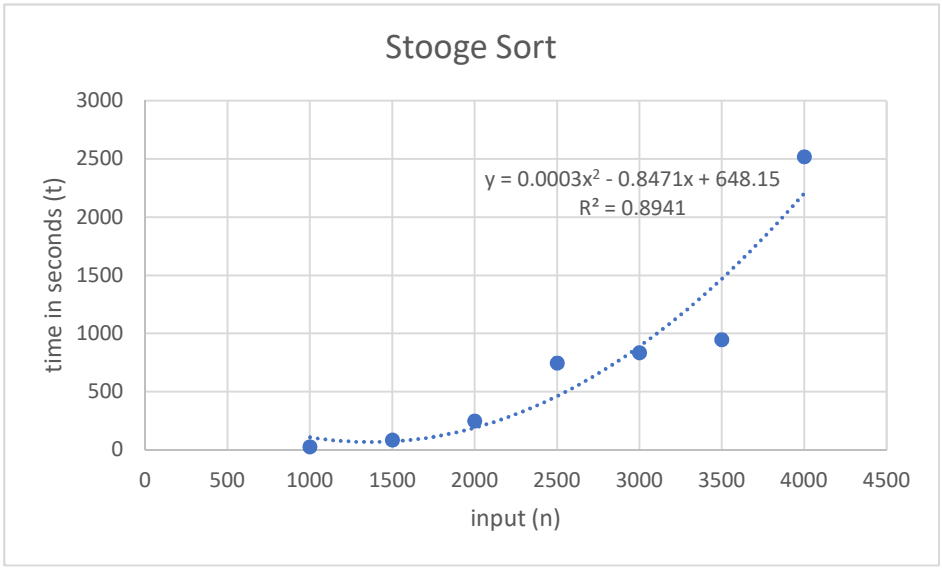
Individual graph of stooge sort:

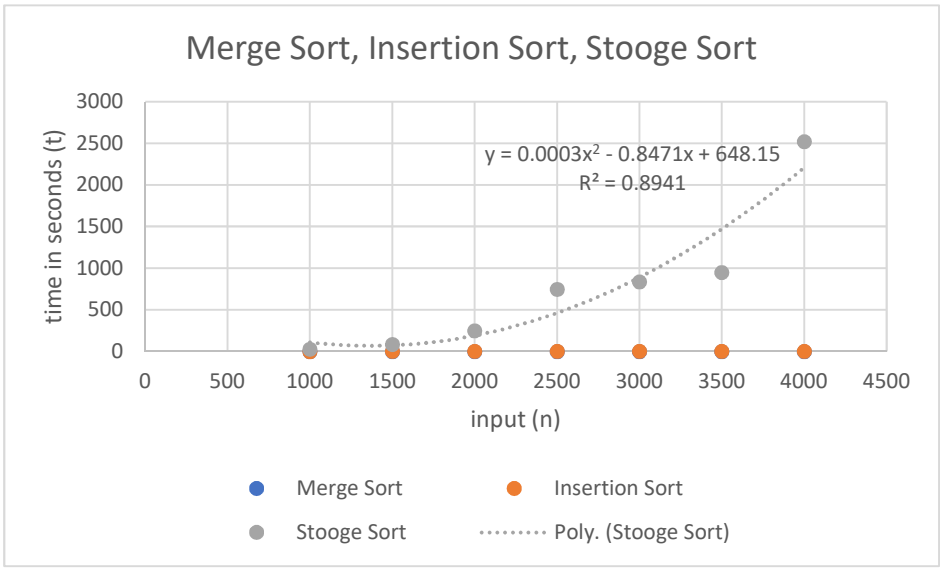


Combined graph (merge sort, insertion sort, stooge sort):
Note: Due to how small the time values are of merge sort and insertion sort compared to stooge sort, the plotted points of merge sort are hidden behind those of insertion sort.



d. Graphs with the equation curves drawn in:





The type of curve that best fits the stooge sort data set is a polynomial 2nd degree curve (n^2). The specific equation of the curve that best fits my data is: $y = 0.0003x^2 - 0.8471x + 648.15$ This curve has an R^2 of .89 My experimental running time is similar to that of the theoretical running time of this algorithm. Based on my data, the running time was roughly n^2 while the theoretical running time for stooge sort is $n^{2.71}$. With stooge sort, I feel that there probably is not a best/worst case. This is because all that would change is the number of constants and these do not make a difference with an algorithm of this complexity.