Student: Joaquin Saldana
Class: CS325 – Intro to Algorithms (Summer 2017)
Homework 2

1. Give 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) = T(n – 2) + n

   Using the Muster method:
   T(n) = a * T(n – b) + f(n)
   a = 1, b = 2, and f(n) = n, so d = 1 and f(n) = $\theta(n^0)$ therefore $T(n) = \theta(n^{d+1}) = \theta(n^2)$

   b. T(n) = 3T(n-1) + 1

   Using the Muster method:
   a = 3, b = 1, and f(n) = 1 so d = 0 so f(n) = $\theta(n^0)$ *therefore T(n) =* $\theta(n^d a_b^n) = \theta(n^0 3_1^n) = \theta(3^n)$

   c. T(n) = 2T($\frac{n}{8}$) + $4n^2$

   Using the Master method:
   a = 2, b = 8, $log_b a = \frac{1}{3} = n^{1/3}$, $f(n) = 4n^2$
   f(n) = $\Omega\left(n^{\frac{1}{3}}\right)$, $case\ 3$
   $2 * 4\left(\frac{n}{8}\right)^2 \leq c4n^2 = \frac{n^2}{8} \leq c4n^2, c = .5\ is\ a\ solution$
   T(n) = $\theta(n^2)$

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

   function quaternary_Search(array, value, left, right, high)

   first_quad = left + (right – 1) / 4
   second_quad = first_quad + (right – 1) / 4
   third_quad = second_quad + (right – 1) / 4

   if Array[first_quad] == value
   return Array[first_quad]
   if Array[second_quad] == value
   return Array[second_quad]
   if Array[third_quad] == value
   return Array[third_quad]

else if Array[first_quad] > value
    return quaternary_Search(Array, value, left, Array[first_quad - 1])
else if Array[second_quad] > value
    return quaternary_Search(Array, value, Array[first_quad + 1], Array[second_quad - 1])
else if Array[third_quad] > value > Array[second_quad]
    return quaternary_Search(Array, value, Array[second_quad + 1], Array[third_quad - 1])
else
    return quaternary_Search(Array, value, Array[third_quad + 1], high)

Notes: The quaternary search algorithm finds the 3 mid indexes of the array and checks if any of them are the value we are looking for. If not, then the function recursively calls itself w/ the left and right boundaries.

b. Give the recurrence of the quaternary search algorithm

The recurrence is $T(n) = T\left(\frac{n}{4}\right) + c$

c. Solve the recurrence to determine asymptotic running time

a =1, b = 4, f(n) = c
Next $n^{logb^{(a)}} = n^{log4^{(1)}} = n^0$
After $n^0 = 1 = \theta(1) = \theta(c)$, therefore case 2 applies
Hence $T(n) = \theta\left(n^{logb^{(a)}} * \log 4\,(n)\right) = \theta\big(1 * log4(n)\big) = \theta(log_4(n))$
The run time is $T(n) = \theta(log_4(n))$

d. When compared to the binary search, the worst case for both is O(lgn), however because the quaternary algorithm is making more comparisons so it may affect runtime.

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

a. Describe and write pseudo-code for the min and max algorithm

Find_min_max(array)

int min;
int max;

If array.length = 1

```
        Min = array[0]
        Max = array[0]
        Return min, max
Else if array.length  == 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
        Minleft , maxright = min_max(array[length / 2])
        Minright , maxleft = min_max (array[length /2 + 1 … n])

        If (maxleft < maxright)
                Max = maxright
        Else
                Max = maxleft

        If minleft < minright
                Min = minleft
        Else
                Min = minright

        Return min, max
```

The function takes an input and divides it until the array is of size 1 and/or 2. If the input is size 1 then the value at position 0 is both the max and the min.

If the size of array is >= 2, then the function will start breaking the array apart and assign the min and max of each array (left and right) until the it hits the base case.

b. Give the recurrence

The recurrence is $T(n) = 2T\left(\frac{n}{2}\right) + c$ for the case n > 2

c. Solve the recurrence and compare to iterative min max algorithm

a = 2, b = 2, f(n) = 2
as a result $n^{logb^{(a)}} = n^{log2^{(2)}} = n^1$
f(n) = 2 and f(n) = n^1

Case 1: f(n) = O(n^1)
f(n) = $O(n^{logb^{(a)-\epsilon}})$ where e = 1
Therefore $T(n) = \theta(n^{logb^{(a)}})$
And $\theta(n^{logb^{(a)}}) = \theta(n^1) = \theta(n)$

Also, an iterative example has a running time of O(n) at worst because it would have compare each element in the array plus compare the current min and max variables.

4. Analyze the Stoogesort algorithm

   a. The pseudo-code sorts the array because it passes sub-arrays that holds the array that was originally passed in. It is noted, that the recursive calls causes for subarrays to overlap. The obvious example is the first Stoogesort recursive call and the third Stoogesort recursive call in the else statement if n > 2
   b. No it will not sort correctly and the reason for it being that the third recursive call will not need to do a sort and two sub arrays will not overlap and as a result only sorts two sub arrays when the length of the array (n) is an even number like 4 or 8. The halves of the array will not overlap

      Example: Assume an array that holds the following integers [100, 12, 8, 99]

      We calculate the m variable which is floor(2 * 4 / 3) which equals 2
      The first recursive calls will pass the array w/ index 0 and 1, sub array [100, 12]
      And will sort it because 100 is greater than 12 so the array will return as [12, 100, 8, 99]

      The second recursive call will be passed the values in index 4 – 2 = 2 and n – 1 = 3, or subarray [8, 99]. No swap is done so the full array is [100, 12, 8, 99].

      The third recursive call will be passed the same subarray as in the first recursive call [100, 12] and no swap is performed. But had it been ceiling it would have been array indexes 0, 1, and 2.

      Therefore the array will remain as [100, 12, 8, 99] and remained unsorted.

   c. A recurrence for Stoogesort

      The recurrence for this algorithm is $T(n) = 3T\left(\frac{2n}{3}\right) + c$

   d. Solve the recurrence to determine the asymptotic running time (ignore ceiling)

      The recurrence is $T(n) = 3T\left(\frac{2n}{3}\right) + c$ so

a = 3, b = 3/2 and f(n) = n^0
and f(n) = $O(n^{2.7})$ and this is case 1
As a result $T(n) = \theta(n^{2.7})$

5. Implement the stooge sort algorithm
    a. Submitted via TEACH

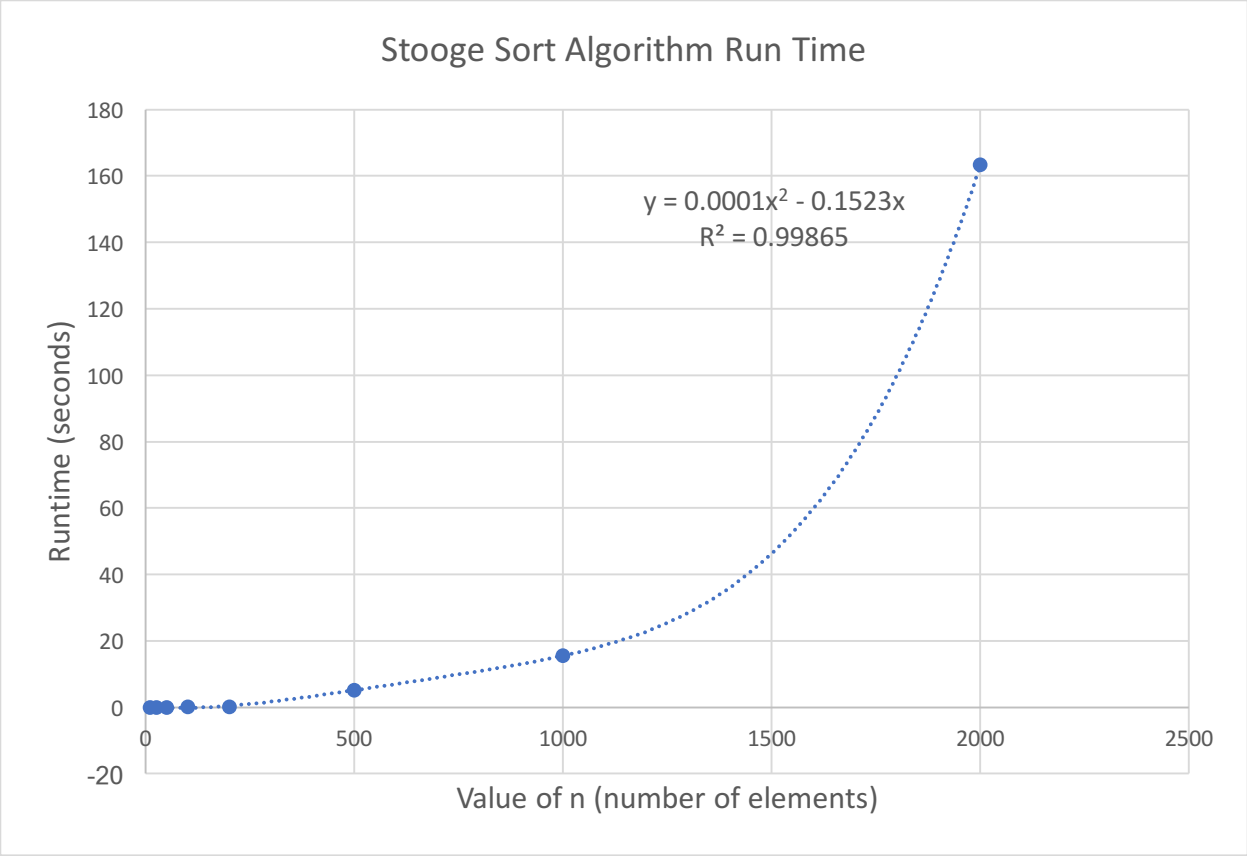
    b. In the spirit of keeping things the same, I tested the stoogesort.py program locally on my machine as I did w/ HW1. I was initially going to use the same n values as well, but I immediately realized that in the first value, n = 1000 (the same as HW1), the algorithm took 15.5 seconds to complete. This is strictly timing the algorithm and not the entire program (see code excerpt below). As a result I drastically decreased the n values to avoid extensive wait times.

    Code excerpt:

```
#!/usr/bin/python

# Student: Joaquin Saldana
# Assignment: Homework 2 / Stooge Sort program

import string
import io
import random
from random import randint
import time

# Much of my code was inspired from the following post:
# http://www.geeksforgeeks.org/stooge-sort/

def stooge_sort(arrayOfInts, i=0, j=None):

    if j is None:
            j = len(arrayOfInts) - 1
    if arrayOfInts[j] < arrayOfInts[i]:
            arrayOfInts[i], arrayOfInts[j] = arrayOfInts[j], arrayOfInts[i]
    if j - i > 1:
            t = (j - i + 1) // 3
            stooge_sort(arrayOfInts, i  , j - t)
            stooge_sort(arrayOfInts, i + t, j)
            stooge_sort(arrayOfInts, i  , j - t)

    return arrayOfInts
```

```
#=====================================================================
=

def main():

    randomIntArray = []

    # what will be the size of the array, this will contain the values 1000, 2000, 5000,
and 10,000
    n = 1000

    while len(randomIntArray) < n:
            randomIntArray.append(randint(0, 10000))

    # will start the timer now that we are about to enter the insertion sort
algorithm/function
    start_time = time.time()

    #insertion sort algorithm
    newArray = stooge_sort(randomIntArray)

    # to identify how many seconds have passed we substract from the
    # current time the start time
    print("--- Algorithm took %s seconds --- " % (time.time() - start_time))

main()
```

n	Stooge Sort Run Time (seconds)
10	0.000108957
25	0.000823021
50	0.007117987
100	0.064029932
200	0.203157902
500	5.216928959
1000	15.51646614
2000	163.3285179



It appears to me that the curve that best firsts the stoogesort data set is polynomial.

e. The theoretical run time of the sorting algorithm is $O(n^{2.7095\ldots})$ which is worst than the run time of $\theta(n^2)$.