Shannon Jeffers
CS 325 HW2

1. These will mostly be completed using the muster and master methods.

   a. Using the Muster Method
   $T(n) = 2T(n-2) +1$
   $T(n) = aT(n-b) + f(n)$
   Let a = 2, b = 2, and d = 0 because $f(n) = \Theta(n^d) = 1$
   Since a > 1 we will use $T(n) = \Theta(n^d a^{n/b})$
   In this case $n^d = n^0 = 1$ so we get
   $\Theta(2^{n/2})$ or alternatively $\Theta(\text{sqrt}(2^n))$

   b. Using Muster Method
   $T(n) = T(n-1) + n^3$
   $T(n) = aT(n-b) + f(n)$
   Let a = 1, b = 1, d = 3 because $f(n) = \Theta(n^d) = n^3$
   Since a = 1 we will use $T(n) = \Theta(n^{d+1})$
   So we get $\Theta(n^4)$

   c. Using the Master Method
   $T(n) = 2T(n/6) + 2n^2$
   Let a = 2, b = 6 and $f(n) = 2n^2$
   $n^{\log_6 2} = n^{.387}$ so f(n) is bounded below by this.
   $f(n) = \Omega(n^{.387 + \epsilon})$ where $\epsilon = 1.613$
   so $T(n) = \Theta(n^2)$

2. Quaternary Search

   a. The quaternary search is similar to the binary search, but would require a larger number of sub arrays be created at each step of the recursion (4 to be exact). With this version 4 values would be tested to see if any of these 4 values were the value we were looking for. If not, we would then check to see which of the 4 sub arrays the value would most likely be in and call the algorithm again on this subarray. Binary search can either return true or false or return the index of the location of the value in the list. I am going to implement my quaternary search to return 1 if found or 0 if it is not in the list.

   Psudocode

```
quaternarySearch(arr, value)
    if length of array = 0
            return 0
    low = 0
    part1 = len(arr) / 4
    part2 = len(arr) / 2
    part3 = (3 X len(arr)) / 4
    high = len(arr) -1

    if arr[part1] or arr[part2] or arr[part3] = value:
        return 1
    else if value < arr[part1]:
        return quaternarySearch(arr[low:part1, value)
    else if value < arr[part2]:
        return quaternarySearch(arr[part1:part2, value)
    else if value < arr[part3]:
        quaternarySearch(arr[part2:part3, value)
    else:
        quaternarySearch(arr[part3:high, value)
```

b. The recurrence for this would be $T(n) = T(n/4) + c$

c. Using the Master Method we get $a = 1$, $b = 4$ $f(n) = c$
$n^{\log_4 1} = n^0 = 1$. Comparing this to $f(n) = c = \Theta(1)$ gives us case 2
$T(n) = \Theta(n^{\log_4 1} \lg n) = \Theta(\lg n)$
It has the same theoretical run time as binary search. There are fewer constant comparisons involved in binary search, and there could be fewer recursive calls associated with quarerarysearch, but these constants matter far less when comparing large values of n.

3. Min-Max

   a. This algorithm can be achieved in a very similar manner to merge-sort, only it doesn't need the lop to merge the arrays back together. The base case's would be case 1 if the array only has 1 element then min and max are both set to that element. If it only has 2 elements, compare the two and set min and max accordingly, else split the array in half and call minmax on both the left and right side, the compare the min max of both sides, to get the min and max of the left and right side together and return that value. My min and max values will be stored in an array of 2 values. The first value will always be the min value, the second value the max value.

      Psudocode:

```
minMax(arr)
    minmax = []

    if len(arr)  = 1
        minmax[0] = arr[0]
        minmax[1] = arr[0]
    else if len(arr) = 2
        if arr[0] > arr[1]
            minmax[0] = arr[1]
            minmax[1] = arr[0]
        else
            minmax[0] = arr[0]
            minmax[1] = arr[1]
    else
        leftminmax = []
        rightminmax = []
```

```
mid = len(arr) / 2
leftminmax = minMax(arr[0:mid])
rightminmax = minMax(arr[mid+1:len(arr)-1)

if leftminmax[0]  < rightminmax[0]
        minmax[0] = leftminmax[0]
else
        minmax[0] = rightminmax[0]
if leftminmax[1] < rightminmax[1]
        minmax[1] = rightminmax[1]
else
        minmax[1] = leftminmax[1]
return minmax
```

b. $T(n) = 2T(n/2) +C$

c. Using the master method to solve this
   a= 2, b = 2 f(n) = C
   This would be case 1 because $n^{\log_2 2} = 1$
   $f(n) = O(n^{\log_2 2-\epsilon})$ where $\epsilon = 1$
   $T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$
   The runtime for the iterative method would also be Θ(n) as it would be a single for loop that needed to check every single value for n. They have the same run time negating and constants.

4. Stoogesort
   a. This sorting method is kind of similar to the merge sort with some important changes. The base cases are when the array only has 1 element, in which the function returns without doing anything, and when the array has 2 elements, in which it swaps those two elements. It then gets the value of 2/3rds of n rounded up, lets call this m. Using this value it recursively calls itself. The first call is on the array from 0…m-1, the next call does n-m…n-1 and the final call is a repeat of the first call. Because there is an overlap between the numbers used in the first call and the numbers used in the second call, the third call is able to integrate any smaller numbers from the back 1/3rd of the array into their proper places in the front of the array.
   b. No, it is important to use the ceiling rather than the floor because our base case stops at an array size of 3. As a counter example if we had an array of size 4 the first sub array would consist of the first 2 elements, and the second sub array would consist of the last 2 elements. There is no overlap here so if either of the numbers in the second sub array is smaller than the numbers in the first sub array, when the third call of stoogesort is made they will not be integrated in. For this example in numbers say we have the array [92, 12, 19, 3]. The first call would be passed the subarray [92, 12]. This would cause a swap leading the array to look like [12, 92, 19, 3]. The second call would receive the sub array [19. 3]. This would also cause a swap leading to [12, 92, 3, 19].. when the final call was made it would be passed [12,92] and the array would never end up in a sorted form.
   c. $T(n) = 3T(2n/3) + C$
   d. Using the master method
      a = 3 b = 3/2 and f(n) = C
      $n^{\log_{3/2} 3} = n^{2.71}$

so this would be case 1.

T(n) = $\Theta(n^{2.71})$

5. The code section of this homework has been submitted on teach

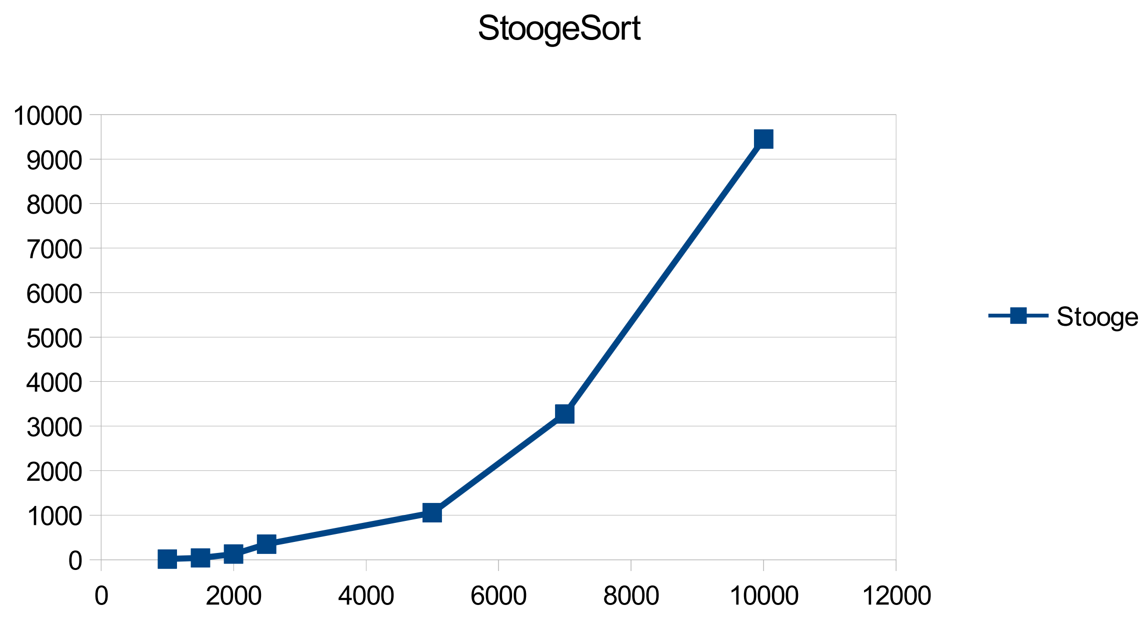
b. Here is the text copy of my modified code:

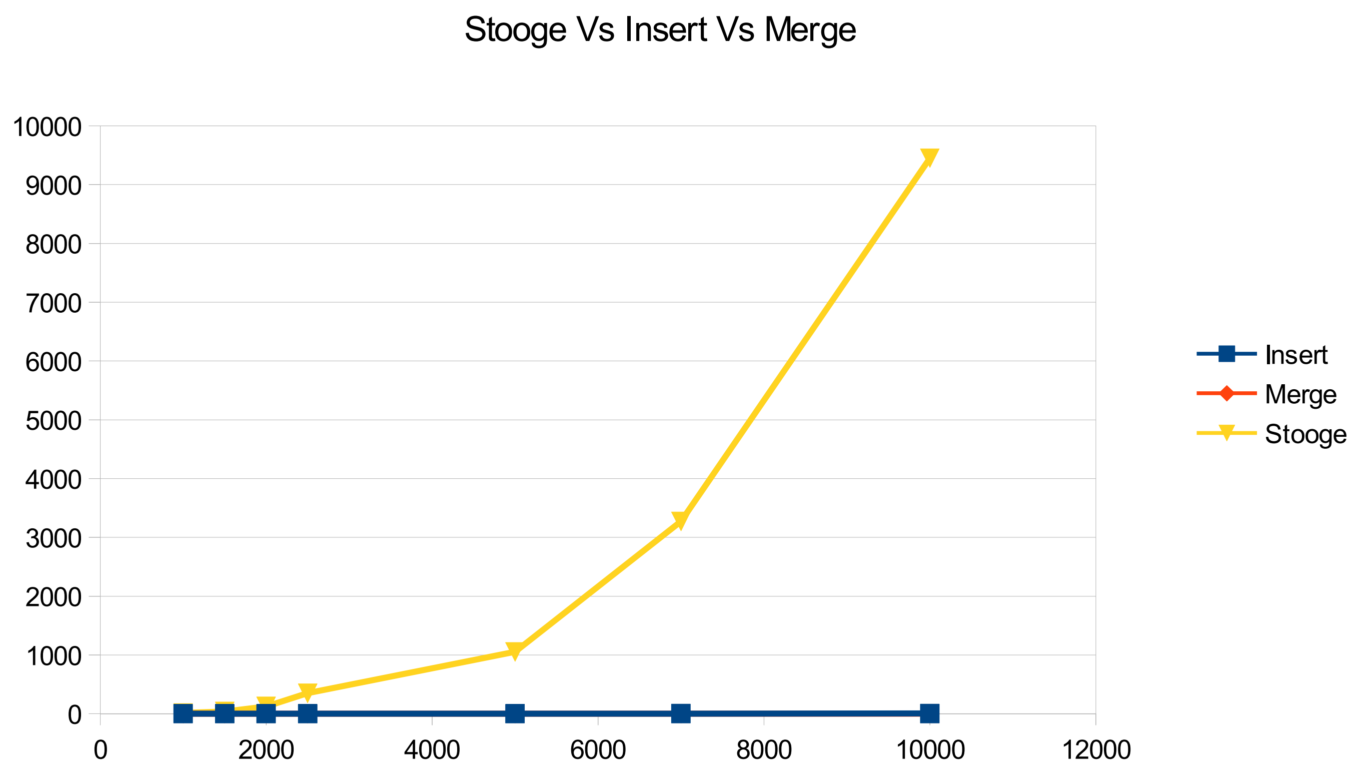
```
import time
6. from random import randint

#n = int(input("Enter a number for n: "))
n=7000
myList = []

for x in range(n):
    i = randint(1,10000)
    myList.append(i)


#insertSort sorts a list by insertion
def stoogesort(myList, low, high):
    n = high-low + 1
    if myList[low] > myList[high]:
        myList[low], myList[high] = myList[high], myList[low]
    if n > 2:
        m = int(n/3.0)
        stoogesort(myList, low, high-m)
        stoogesort(myList, low+m, high)
        stoogesort(myList, low, high-m)


#collects the time it takes to run
start = time.time()
stoogesort(myList, 0, (len(myList)-1))
end = time.time()

print(end - start)
```

Table of values can be found on the next page.

n	StoogeSort Time (seconds)
1000	12.92
1500	38.71
2000	123.93
2500	349.53
5000	1052.15
7000	3271.27
10000	9450.65

c.





Stooge sort was so incredibly slow that insertion and merge sort are laying on each other which causes you not to be able to see merge sort at all. Or maybe merge sort was so fast in comparison that it just disappeared all together. I had to run a couple additional data points for both merge and insert because there was no way in which I was going to be able to run stooge sort at the same data points in which I ran insert and merge. As you can see stooge sort already took almost 10k seconds to run at 10k n!!

d. The curve is definitely one of a power or polynomial, since it starts out increasing at a sort of steady rate and then suddenly starts to increase dramatically. The equation for this curve is :

$$f(x) = 0.000000069\ x\verb|^|2.7836637442$$

The theoretical run time listed above $\Theta(n^{2.71})$ which is so incredibly close to the actual run time that I achieved!! This just further solidifies the fact that the run time I gathered using the mast method above is the proper run time for this algorithm.