Erin Alltop
HW 2 – CS325 Fall 2017
10/2/17

1.

a. $T(n) = 2T(n - 2) + 1$

The asymptotic bounds for T(n) can be found using the Muster method for decrease and conquer. In this case, a=2, b=2, and f(n)=1 so d=0 because $n^0=1$. Since a > 1, then $T(n) = O(n^d a^{n/b})$.

So, $O(n^0 2^{n/2}) = O(2^{n/2})$. Since f(n) is $\theta(n^d)$ then we can see that **$T(n)$ is $\theta(2^{n/2})$.**

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

Using the muster theorem for this equation as it is a decrease and conquer also, we can see that a = 1, b = 1, and $f(n) = n^3$. Since a = 1, then $T(n) = O(n^{d+1})$ per the theorem. In this case d = 2. Now we can see that **$T(n)$ is $O(n^3)$.**

c. $T(n) = 2T^{(n/6)} + 2n^2$

For this recurrence we will use the master theorem to find the asymptotic bounds. Here we see that a = 2, b = 6, and $f(n) = 2n^2$. So we can see that $T(n) = \theta(n^{\log_b a}) = \theta(n^{\log_6 2}) =$ **$\theta(n^{0.37})$,** which is our answer.

2.

a. A quarternary search algorithm is an algorithm similar to the binary search algorithm but splits the input into four sets in approximately one fourth instead of one half. The binary search works by splitting the array in half and "tossing out" the half of the array that does not contain the desired value. A quarternary search would do much the same, but search four quarters instead of two. In my interpretation, I first find the three "mid points" of the array that would split it into four quarters. Then after determining that the values are not the desired term, I then go quarter by quarter recursively and splitting the array into quarters and testing the mid points again until the desired value is found.

Pseudocode for a quarternary search algorithm:

```
quarterSearch(A[0..n-1], value, low, high)
        if low == high //array of size 1
                return high;
        while high > 1
                midpoint = (low + high) / 2 //middle point
                quarter1 = ceiling(midpoint + high) / 2 //midpoint of first half
                quarter2 = ceiling(low + midpoint) /2  //midpoint of second half
                if(midpoint == value)
                        return midpoint;
                if(quarter1 == value)
```

```
            return quarter1;
        if(quarter2 == value)
            return quarter2;
        else if(A[quarter1] > value)
            return quarterSearch(A, value, low, quarter1-1)
        else if(A[quarter2] > value)
            return quarterSearch(A, value, quarter1, quarter2-1)
        else if(A[midpoint] > value)
            return quarterSearch(A, value, midpoint, midpoint-1)
        else if(A[high] < value) //should be found here
            return quarterSearch(A, value, quarter2, high-1)
```

b. The recurrence equation for the quarternary search algorithm would be T(n) = 4T(n/4) + O(1), where 4T is the number of subproblems (we are sorting four quarters of the array recursively), n/4 is the subproblem size, and O(1) is the work. In this case, it is a constant time because we are simply comparing the value and it is either equal or not.

c. Using the Master Theorem, we can see that the running time is **θ(lgn).** This can be found because a=4, b=4, and c = O(1). We compare $n^{\log_4 4} = n^1 = n$. The running time for the quarternary search surprisingly is the same as the binary search algorithm. Since both are dividing the array into equal parts to solve, they are solving the search in a logarithmic fashion.

3.

a. Our minMax algorithm will search through an unsorted array to find the largest and smallest values. We will use a divide and conquer approach which has three parts: Dividing the problem into subproblems, solving the subproblems, and combining them into a solution. This can be characterized by dividing our problem into two. We split the array into two parts and recursively look at each part to find the min and max numbers. Then we recombine the parts to find the overall min and max numbers.

   Pseudocode for the minMax algorithm:

```
Int min; //declare outside of function
Int max;
minMax(A[0…1], low, high) //initial low = 0, high = N-1
        if sizeof A = 2;
        if(A[0] > A[1])
                swap A[0] and A[1]
                min = A[0]
                max = A[1]
                break;
        else
        min = A[0]
        max = A[1]
        break;
else
```

midpoint = ceiling(low + high) / 2 //find the midpoint
minMax(A, low, midpoint) //first half
minMax(A, mid + 1, high) //second half
minMax(A, low, high) //whole array (combine)

b. The recurrence equation can be written as 2T(n/2) + 2, where 2T is the number of subproblems (searching two halves of the array), n/2 is the subproblem size, and 2 is the work to find the min and max of the whole array. Solving further, this will give you T(n)= 3/2n-2 in closed form.

c. The worst case complexity then would be O(n). This can be deduced by looking and thinking about the algorithm itself. Each number will need to be checked at some point so even with the divide and conquer approach the complexity will remain at O(n) and will not be able to be faster than that. In that same vein, whether we are iteratively solving the problem or using the divide and conquer approach, the running time will be the same.

4.

a. The STOOGESORT algorithm sorts its input by first detecting if the size of the array is equal to 2. If so, and if the element at index 0 is larger than the element at index 1, then it swaps these two. Otherwise, if the array is larger than 2, then it calculates the ceiling of 2n/3(m). It then recursively runs the STOOGESORT algorithm with three different inputs. First, it uses the array from 0 to m-1 which is the first 2/3rds of the array. Next, it does the same with the last 2/3rds of the array, and lastly it runs the algorithm with the first 2/3rds of the array again to verify that it is correctly sorted or complete sorting.

b. If we replaced k = ceiling(2n/3) with k = floor(2n/3) STOOGESORT would no longer work correctly. In larger sizes of n it might work for a while, but when we get to lower values of n, it will no longer work. For example, if we have an array of size 4 and run the algorithm, k would equal 2 in the first iteration. This would work for swapping two numbers if needed, but the algorithm will no longer go beyond that point, potentially leaving half of the array or more unsorted.

c. The recurrence equation for the STOOGESORT algorithm would be T(n) = 3T(2n/3) + O(1), where 3 is the number of subproblems (running STOOGESORT three times), 2n/3 is the subproblem size, and O(1) is the work which is constant time.

d. Using the Master Theorem, a= 3, b=3/2 (because 2n/3 = n/(3/2)), and d = 0. We can then compare $\log_b{}^a = \log_{3/2}{}^3 = n^{\log_3/\log_{3/2}} = \theta(n^{2.7})$.

5 a. Submitted to TEACH

b. STOOGESORT running time testing. Modified code and testing table follows:

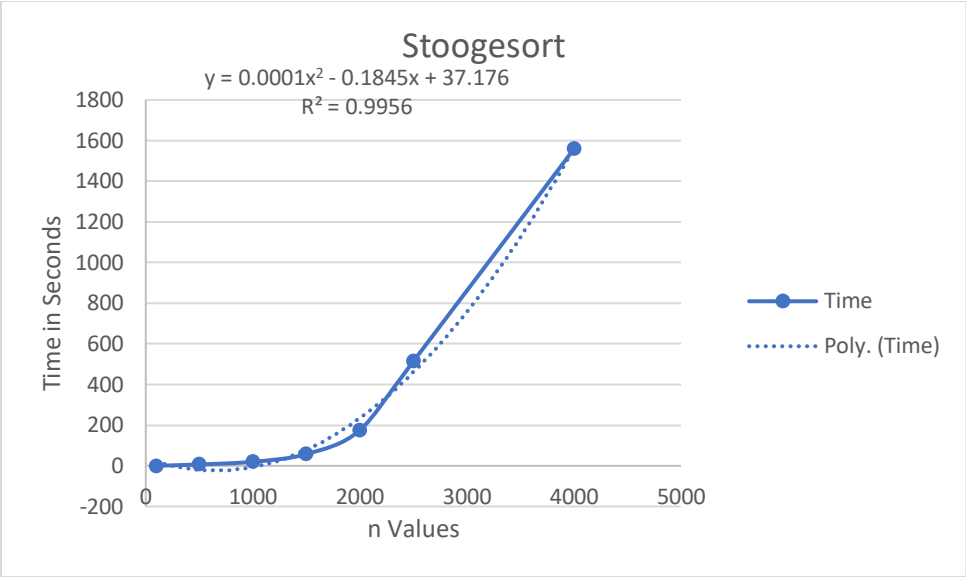
```
def stoogeSort(arr, low, high):
if arr[high] < arr[low]:
```

```
            arr[low], arr[high] = arr[high], arr[low]
    if high - low > 1:
            t = (high - low + 1) // 3
            stoogeSort(arr, low, high-t)
            stoogeSort(arr, low+t, high)
            stoogeSort(arr, low, high-t)
    return arr

    def stooge(arr): return stoogeSort(arr, 0, len(arr) - 1)

    import random
    import time
    arr=[]
    arr = random.sample(xrange(1, 10000), 4000)
    start = time.time()
    stooge(arr)
    end = time.time()
    print "I sorted this in "
    print(end-start)
    print " seconds"
```

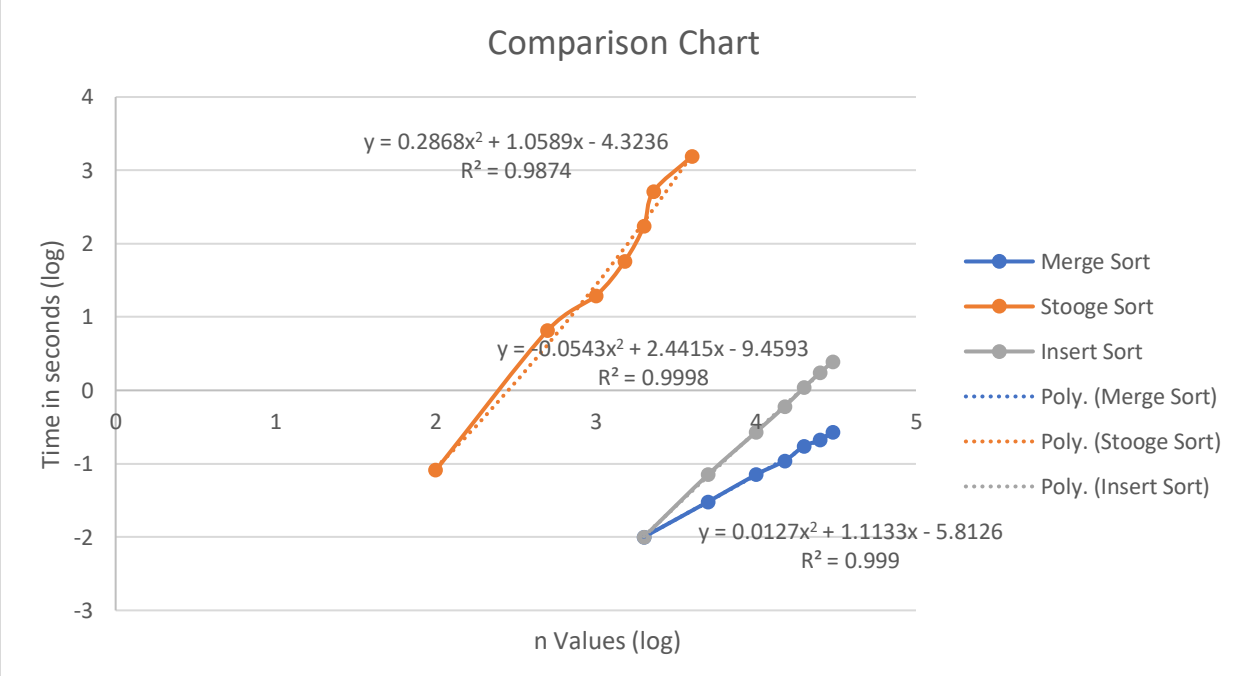
RUNNING TIME TESTS OF STOOGESORT

Number of Elements to Sort	Time to Sort (in seconds)
100	0.081
500	6.655
1000	19.85
1500	57.68
2000	175.26
2500	520.05
3000	514.03

c. Graphical representation of running time data of STOOGESORT:



Graphical representation of all three sorting algorithms, using a log on log chart:



d. The polynomial curve fits the StoogeSort data set best with an equation of $y = 0.0001x^2 - 0.1845x + 37.176$ and an R-value of $R^2 = 0.9956$. This fits the data very closely. This fits the theoretical running time fairly closely as they are both polynomials.