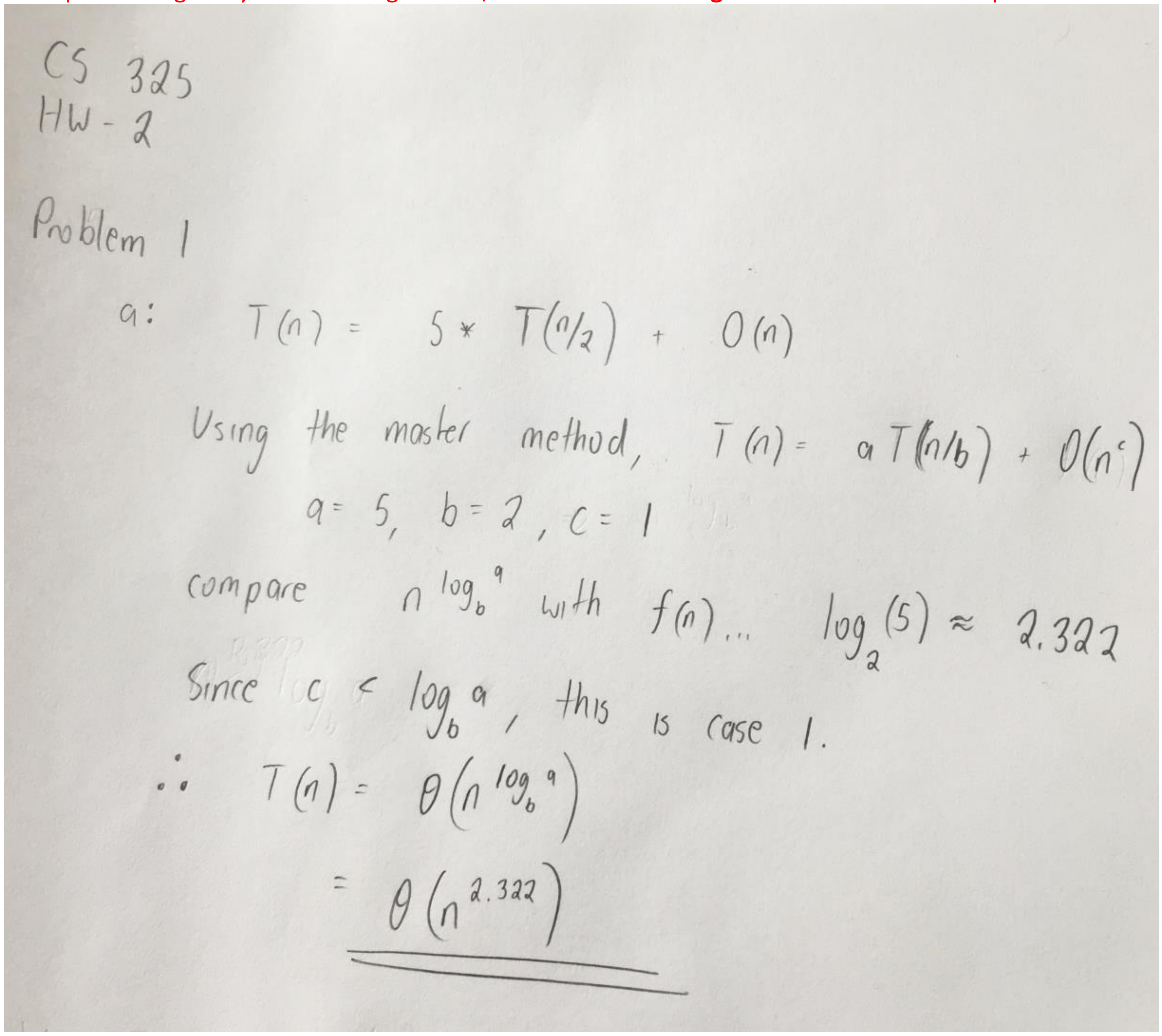
Christopher Ragasa
CS 325, HW 2
4/15/18

Problem 1: *(3 points)* Suppose you are choosing between the following three algorithms:

- *Algorithm A* solves problems of size n by dividing them into five sub problems of half the size (n/2), recursively solving each sub problem, and then combining the solutions in linear time.
- *Algorithm B* solves problems of size n by recursively solving two sub problems of size (n -1) and then combining the solutions in constant time.
- *Algorithm C* solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each sub problem, and then combining solutions in ø(n2) time.

After performing analysis on all 3 algorithms, the best choice is **Algorithm C**. Please see the pictures below:



CS 325
HW - 2

Problem 1

a: $T(n) = 5 * T(n/2) + O(n)$

Using the master method, $T(n) = a\,T(n/b) + O(n^c)$

$a = 5, \quad b = 2, \quad c = 1$

compare $n^{\log_b a}$ with $f(n)$... $\log_2(5) \approx 2.322$

Since $c < \log_b a$, this is case 1.

$\therefore \; T(n) = \Theta(n^{\log_b a})$

$= \underline{\underline{\Theta(n^{2.322})}}$

CS 325, HW-2

Problem 1

b: $T(n) = 2 * T(n-1) + O(1)$

$T(1) = O(1) = c$

$T(n) = 2 * T(n-1) + c$

$T(2) = 2 * T(2-1) + c$

$= 2 * T(1) + c$

$T(3) = 2 * T(3-1) + c = 2 * T(2) + c$

$= 2 * 2 * T(1) + c$

$T(4) = 2 * T(4-1) + c = 2 * T(3) + c$

$= 2 * 2 * 2 * T(1) + c$

General form: $T(n) = 2^{n-1} * T(1) + c$

$= O(2^n)$

CS 325, HW-2

Problem 1

C: $T(n) = 9 * T\left(\frac{n}{3}\right) + O(n^2)$

Using master method...

$a = 9,\ b = 3,\ d = 2$

compare $n^{\log_b a}$ with $f(n)$...

$\log_b a = \log_3 9 = 2$

$n^2 = n^2$

since $n^{\log_b a} = f(n)$.... use case 2

$\therefore\ T(n) = \theta(n^2 \log n)$

Problem 2: *(6 points)* The ternary search algorithm is a modification of the binary search algorithm that splits the input not into two sets of almost-equal sizes, but into three sets of sizes approximately one-third.

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

Ternary search is similar to that of a binary search, except that it splits the input into of three sets of sizes approximately one-third. The algorithm can be created with either an iterative or recursive approach.

To create this algorithm with an iterative approach, we would store 3 index values. Every element to the left of index2 will be the first third of the set, every element to the right of index3 will be the last third of the set, and every element in between the two indices will be the second third of the set.

Each iteration of the while loop will first check if either of the indices equal the target. If they do, return the index value. Otherwise, re-assign the values for left and right. See below for the Python code.

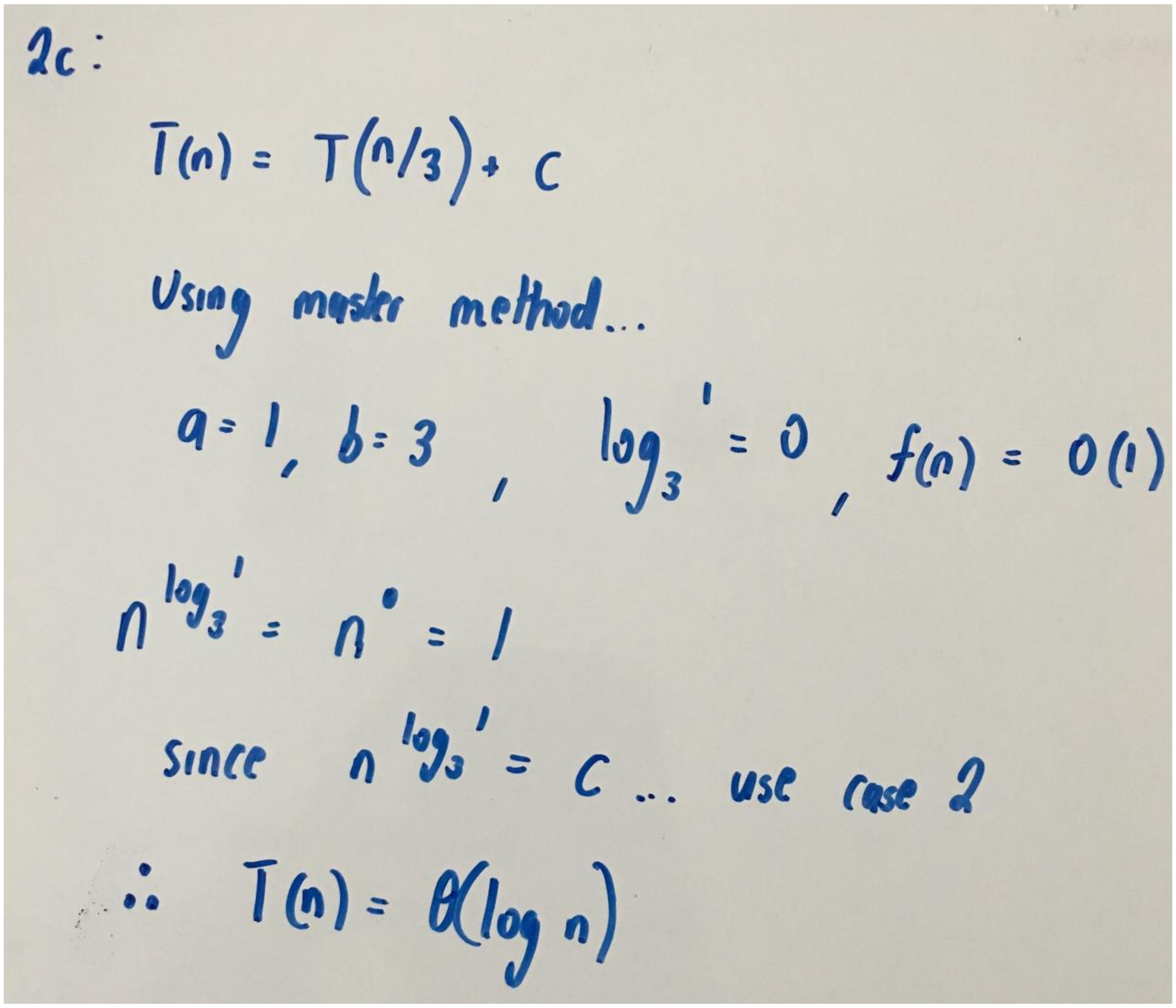
```
def ternary_search(arr, key):
    left = 0
    right = len(arr) - 1
    while left <= right:
        idx1 = left
        idx2 = left + (right - left) // 3
        idx3 = left + 2 * (right - left) // 3
        if key == arr[left]:
            return left
        elif key == arr[right]:
            return right
        elif key < arr[left] or key > arr[right]:
            return -1
        elif key <= arr[idx2]:
            right = idx2
        elif key > arr[idx2] and key <= arr[idx3]:
            left = idx2 + 1
            right = idx3
        else:
            left = idx3 + 1
    return
```

b) Give the recurrence for the ternary search algorithm

Since the recurrence for binary search is $T(n) = T(n/2) + c$, the recurrence for the ternary search algorithm is $T(n) = T(n/3) + c$.

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

Both ternary search and binary search have a running time of O(log n). Please see picture below:



2c:

$T(n) = T(n/3) + c$

Using master method...

$a = 1$, $b = 3$, $\log_3 1 = 0$, $f(n) = O(1)$

$n^{\log_3 1} = n^0 = 1$

since $n^{\log_3 1} = c$ ... use case 2

$\therefore T(n) = \theta(\log n)$

Problem 3: *(6 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.

The min and max algorithm uses recursion to divide the array into sub-arrays of half the size until the array reaches a length of 2 or 1. Once the array reaches a size <= 2, it finds the larger and smaller of the two elements and stores those in global variables. At the very end, the algorithm returns a tuple of min and max.

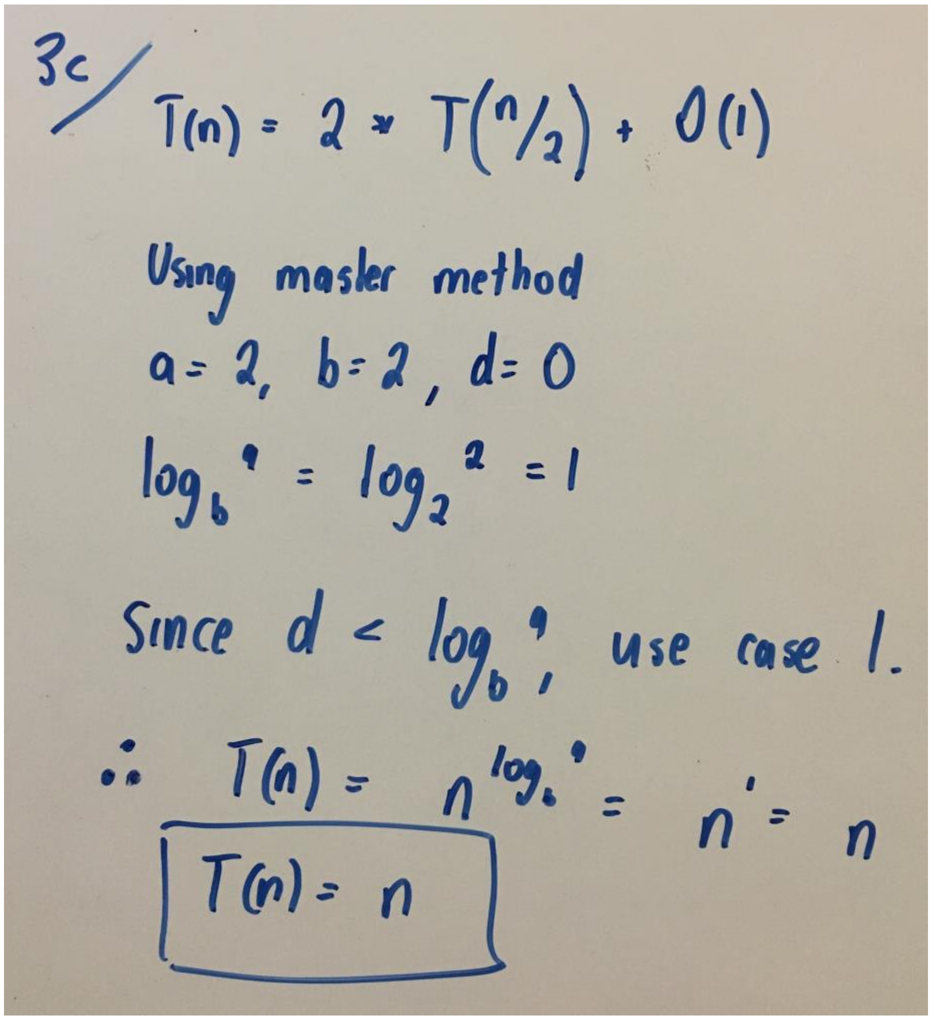
```
min = 100001
max = -1
def getLarger(a, b):
    if a > b:
        return a
    else:
        return b

def getSmaller(a, b):
    if a < b:
        return a
    else:
        return b

def min_and_max(array, start, end):
    global min
    global max
    mid = (start + end) / 2
    if end - start > 2:
        min_and_max(array, start, mid)
        min_and_max(array, mid, end)
    else:
        array = array[start:end]
        if len(array) == 1:
            array.append(array[0])
        max = getLarger(max, getLarger(array[0], array[1]))
        min = getSmaller(min, getSmaller(array[0], array[1]))
    return (max, min)
```

b) Give the recurrence.

The recurrence for min_and_max algorithm with a divide and conquer approach is T(n) = 2 * T(n/2) + O(1).

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.

Given T(n) in part b, we are able to solve for the recurrence using the master method. See picture below. After using the master method to solve for the recurrence, we see that the time complexity of the min_and_max algorithm using the divide and conquer approach is O(n). This is identical to the time complexity of finding the min and max using an iterative approach, which also has time complexity of O(n).



3c/ $T(n) = 2 * T(n/2) + O(1)$

Using master method

$a = 2, \; b = 2, \; d = 0$

$\log_b^a = \log_2^2 = 1$

Since $d < \log_b^a$, use case 1.

$\therefore \; T(n) = n^{\log_b^a} = n^1 = n$

$\boxed{T(n) = n}$

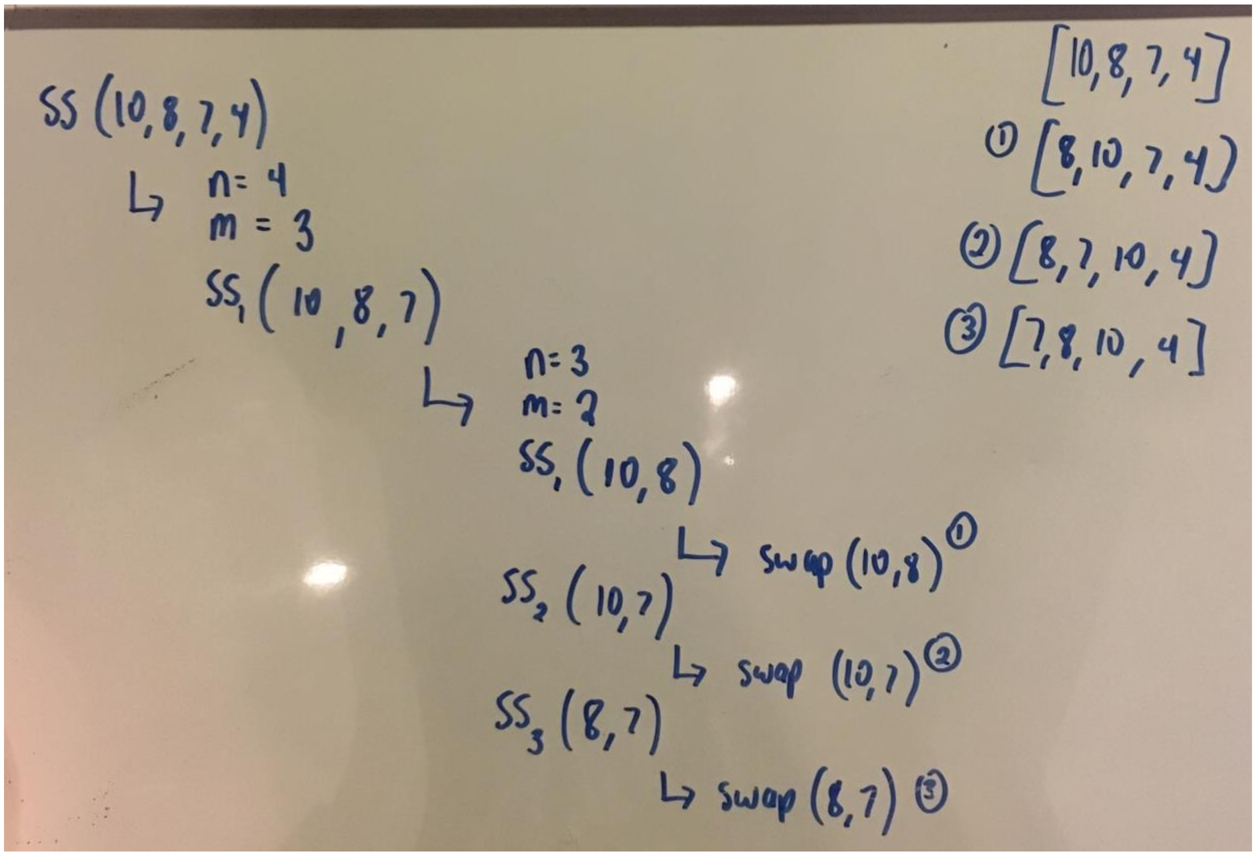
Problem 4: *(5 points)* Consider the following pseudocode for a sorting algorithm.

```
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.

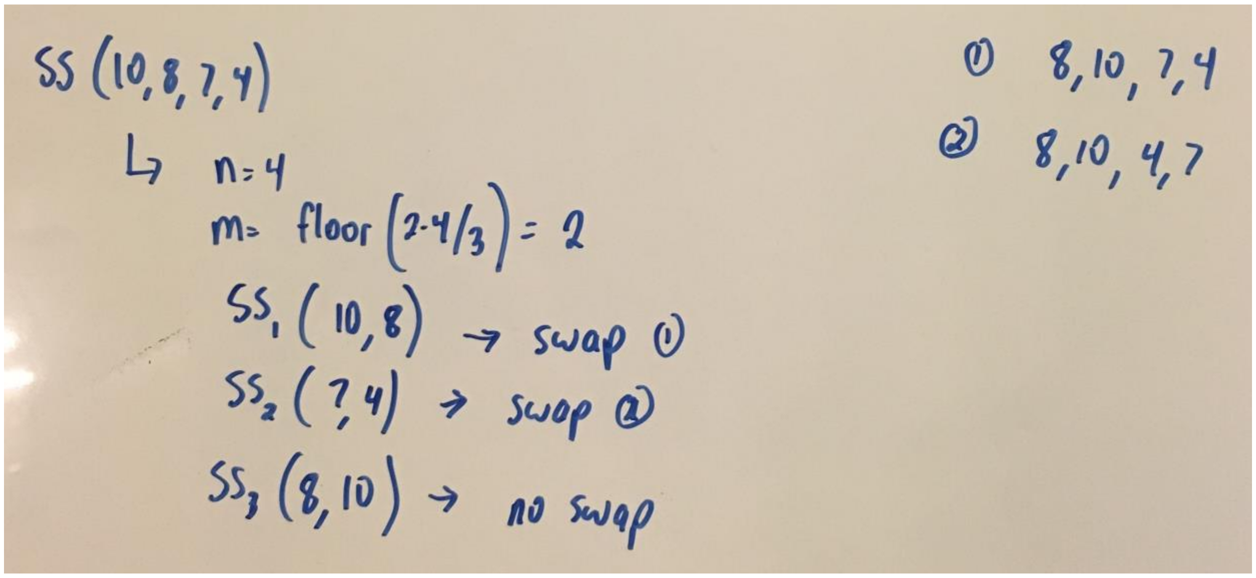
Stoogesort uses a recursive divide and conquer technique to sort a list. If the list passed is larger than size 2, the algorithm recursively breaks the list down into 2 sub-lists of sizes 2/3 and 1/3 of the original size.

Once the list is of size 2, the first and second elements are compared. If the first element is larger than the first, they are swapped. This processed is repeated for each recursive call of stoogesort until the entire list is sorted.



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 sort correctly if m was replaced with floor(2n/3). Let us use a counterexample while passing the same array that was passed in part a, [10, 8, 7, 4]. Please see the picture below:



SS(10, 8, 7, 4)

$\hookrightarrow$ n = 4

m = floor(2·4/3) = 2

SS_1(10, 8) → swap ①

SS_2(7, 4) → swap ②

SS_3(8, 10) → no swap

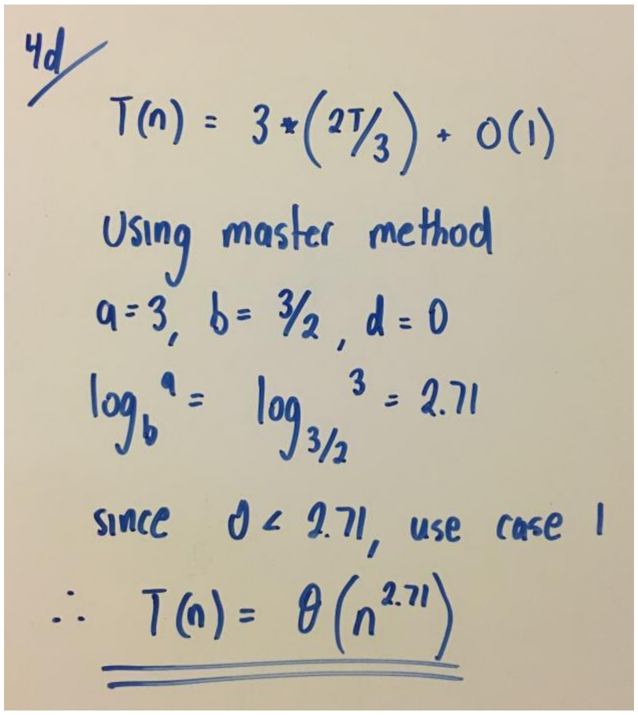
① 8, 10, 7, 4

② 8, 10, 4, 7

We can see that if we use floor instead of ceiling, m evaluates to 2 instead of 3. When going through the recursive calls to stoogesort, the list is not properly sorted.

c) State a recurrence for the number of comparisons executed by STOOGESORT.
T(n) = 3 * (2T/3) + O(1)

d) Solve the recurrence to determine the asymptotic running time.
Please see picture below:



4d/

$T(n) = 3 * (2T/3) + O(1)$

Using master method

$a = 3$, $b = 3/2$, $d = 0$

$\log_b a = \log_{3/2} 3 = 2.71$

since $0 < 2.71$, use case 1

$\therefore \underline{\underline{T(n) = \theta(n^{2.71})}}$

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".

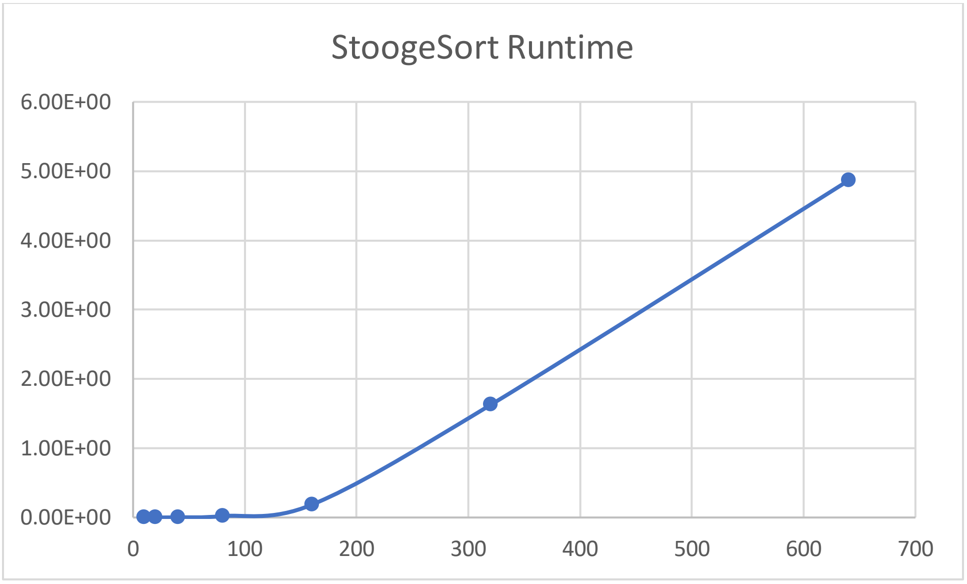
Please see file, stoogesort.py.

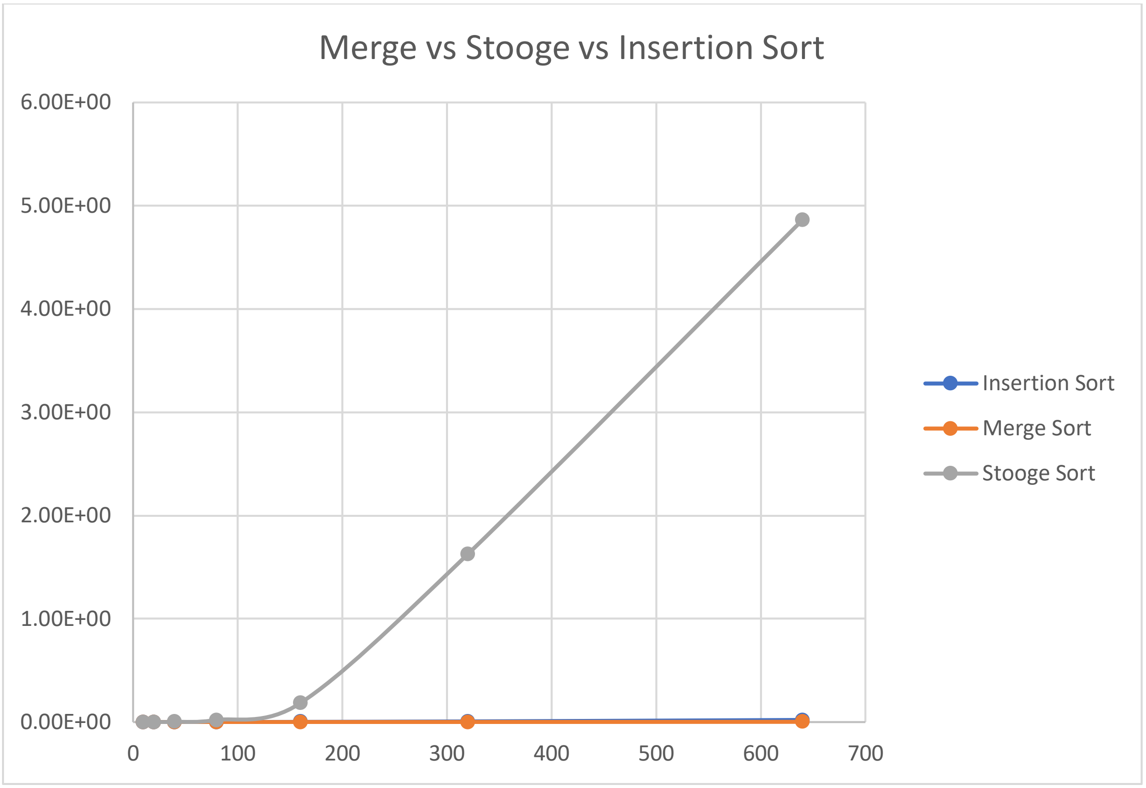
b) Now that you have proven that your code runs correctly using the data.txt input file, you can modify the code to collect running time data. Instead of reading arrays from a file to sort, you will now generate arrays of size n containing random integer values and then time how long it takes to sort 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.

```
def create_array(size = 10000, max = 10000):
        import random
        randoms = []
        for i in range(size):
                randoms.append(random.randrange(0, max))
        return randoms

def stoogesort(arr, l = 0, h = None):
  if h is None:
        h = len(arr) - 1

  # If first element is smaller
  # than last,swap them
  if arr[h]<arr[l]:
    arr[l], arr[h] = arr[h], arr[l]

  # If there are more than 2 elements in
  # the array
  if h - l > 1:
    t = (h - l + 1) // 3

    # Recursively sort first 2/3 elements
    stoogesort(arr, l, h-t)

    # Recursively sort last 2/3 elements
    stoogesort(arr, l+t, h)

    # Recursively sort first 2/3 elements
    # again to confirm
    stoogesort(arr, l, h-t)
  return arr

def write_to(arr_in, f_in):
	for i in arr_in:
		f_in.write(str(i) + " ")
	f_in.write("\n")

def print_execution_time(n, arr):
	import timeit
	start = timeit.default_timer()
	stoogesort(arr)
	stop = timeit.default_timer()
	print("Stoogesort, n = " + str(n) + ", time: " + str(stop - start))


def main():
	# create arrays for benchmarking
	array100 = create_array(100, 10000)
	array200 = create_array(200, 10000)
	array500 = create_array(500, 10000)
	array700 = create_array(700, 10000)
	array1000 = create_array(1000, 10000)
	array1200 = create_array(1200, 10000)
	array1500 = create_array(1500, 10000)

	# benchmark each stoogesort
	print_execution_time(100, array100)
	print_execution_time(200, array200)
	print_execution_time(500, array500)
	print_execution_time(700, array700)
	print_execution_time(1000, array1000)
	print_execution_time(1200, array1200)
	print_execution_time(1500, array1500)

if __name__ == "__main__":
	main()
```

d) 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.

Stooge Sort	
n	time
10	8.70E-05
20	0.000751
40	0.002257
80	0.020685
160	0.184731
320	1.626582
640	4.863745

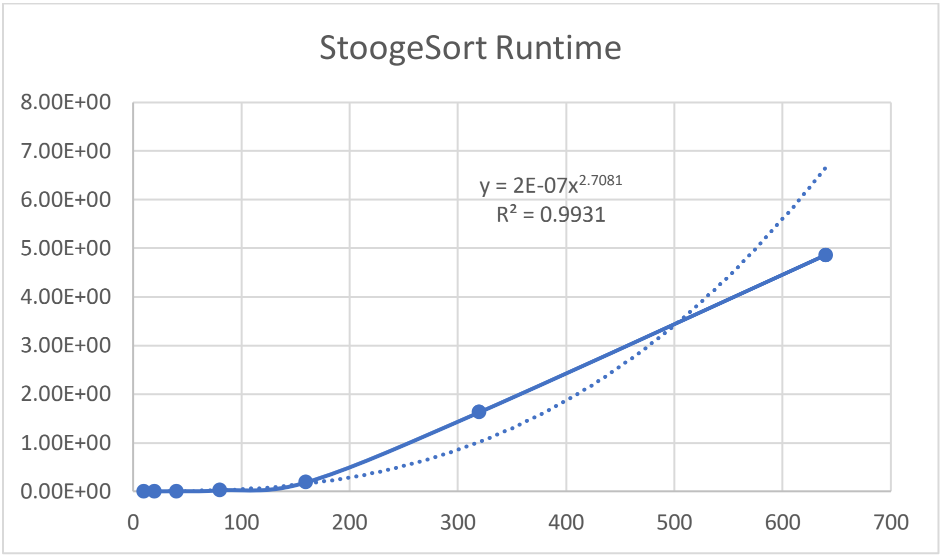




We can see that stooge sort runs significantly slower than merge sort and insertion sort. The difference in time between stooge sort and the other two algorithms is so great that the performance of merge sort and insertion sort looks constant relative to stooge sort. Stooge sort is probably not a preferred method of sorting because of how slow it is.

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?

Stoogesort best fits a line with f(x) = 2E-07x^2.7081 and has an R^2 value of 0.9931 when compared to a power line. Stoogesort has a theoretical complexity of $O(n^{\log 3 / \log 1.5}) = O(n^{2.7095})$. This is very close to my experimental runtime, as the equation derived from my timetable was (x) = 2E-07x^2.7081.


StoogeSort Runtime
y = 2E-07x^2.7081
R² = 0.9931
8.00E+00
7.00E+00
6.00E+00
5.00E+00
4.00E+00
3.00E+00
2.00E+00
1.00E+00
0.00E+00