Class: CS-325
Term: Fall 2017
Author: Jon-Eric Cook
Date: October 6, 2017
Homework: #2

1) (5 points) Give the 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 case T(0) = 1 and/or T(1) = 1.

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

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

c) T(n) = 2T(n/6) + 2n^2

Answers:

a) By the Muster Method: T(n) = aT(n - b) + f(n)
T(n) = 2T(n - 2) + 1
a = 2
b = 2
f(n) = 1 so d = 0
f(n) = theta(n^0)
Due to a being > 1,
T(n) = theta(n^d a^(n/b))

As a result,
T(n) = theta(n^0 2^(n/2))
Therefore,
T(n) = theta(sqrt(2^n))

b) By the Muster Method: T(n) = aT(n - b) + f(n)
T(n) = T(n - 1) + n^3
a = 1
b = 1
f(n) = n^3 so d = 3
f(n) = theta(n^3)
Due to a being = 1,
T(n) = theta(n^(d+1))

As a result,
T(n) = theta(n^(3+1))
Therefore,
T(n) = theta(n^4)

c) By the Master Method: T(n) = aT(n/b) + f(n)
T(n) = 2T(n/6) + 2n^2
a = 2
b = 6
f(n) = 2n^2
n^(log6(2)) = n^0.3868
f(n) = omega(n^(0.3868 + e)) for e = 1 and
2(cn/6) <= c2n^2 for c = 1/6
Therefore,
T(n) = theta(f(n))
T(n) = theta(n^2)

2) (5 points) 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.

b) Give the recurrence for the quaternary search algorithm

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.

Answers:

a) The Quaternary Search algorithm is similar to that of the Binary Search algorithm as it too splits up the input array into smaller arrays. Where the Binary Search only splits the input array into two separate arrays each iteration, the Quaternary Search splits the array into four smaller arrays. The Binary Search algorithm, going left to right, checks the middle element of the selected array and then determines if the value is less than or greater than the value being searched for. Once this is done, it recursively calls itself with the new indices of the smaller array, or returns that it found it. For the Quaternary Search algorithm, going left to right, checks the middle element of the selected array and then determines if the value is less than or greater than the value being searched for. Once this is done, it recursively calls itself with the new indices of the smaller array, or returns that it found it. The big difference between the two algorithms is that for each recursive call, the Quaternary Search algorithm breaks up the array into four arrays where the Binary Search algorithm only does two. This means that the Quaternary Search algorithm will get to an array with only one element faster.

Pseudo-code for the Quaternary Search algorithm is as follows:

```
QuaternarySearch(A[0…n-1], value, low, high)
    while high >= 1
        a = low + ((high - low) / 4)
        b = low + ((high - low) / 2)
        c = low + (3(high - low) / 4)
        if A[a] = value
            return A[a]
        else if A[b] = value
            return A[b]
        else if A[c] = value
            return A[c]
        else if A[a] > value
            return QuaternarySearch(A, value, low, a-1)
        else if A[b] > value
            return QuaternarySearch(A, value, a+1, b-1)
        else if A[c] > value
            return QuaternarySearch(A, value, b+1, c-1)

        else
            return QuaternarySearch(A, value, c+1, high)
    return not_found
```

b) Recurrence:

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

c) By the Master Method: $T(n) = aT(n/b) + f(n)$

$T(n) = T(n/4) + c$
$a = 1$
$b = 4$
$f(n) = c$
$n^{(\log_4(1))} = n^0 = 1$
if $f(n) = \theta(n^{(\log_b(a))})$, then $T(n) = \theta(n^{(\log_b(a))}\lg n)$
$f(n) = \theta(1)$
Therefore,
$T(n) = \theta(\lg n)$

The running time of the Quaternary Search algorithm is the same as the Binary Search Algorithm.

Resource: After having zero success at trying to figure out how to modify the Binary Search Algorithm to match that of the Quaternary Search algorithm, I resorted to using Google for help. I found this: https://stackoverflow.com/questions/39845641/quaternary-search-algorithm I spent around an hour sifting through the code and tracing on paper what happens. I feel a lot more confident about how the Quaternary Search algorithm works and how it draws from the Binary Search Algorithm.

3) (5 points) Design and analyze a divide and conquer algorithm that determines the minimum and maximum value of an unsorted array.

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

b) Give the recurrence.

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.

Answers:

a) The min_and_max algorithm utilizes some features from the Merge Sort algorithm. Essentially, it drills down until it has an array of length 2 and compares the two elements to determine which is the max and min. The max and min are stored and then compare to the next array of length 2.This comparison to find the min and max is done instead of "merging" the arrays back together as the Merge Sort algorithm does.

Pseudo-code for the divide and conquer algorithm that finds the min and max of an unsorted array.

```
# returns an array of format [min, max]
min_and_max(a):
        if len(a) == 1:
                return [a[0],a[0])
        if len(a) == 2:
                if (a[0] < a[1]:
                        return [a[0],a[1])
                else:
                        return [a[1],a[0]]
        else:
                mid = length(a)/2
                left_min_max = min_and_max(a[:mid])
                right_min_max = min_and_max(a[mid:])

                if left_min_max[1] > right_min_max[1]:
                        max = left_min_max[1]
                else:
                        max = right_min_max[1]

                if left_min_max[0] < right_min_max[0]:
                        min = left_min_max[0]
                else:
                        min = right_min_max[0]

                return [min, max]
```

b) $T(n) = 2T(n/2) + c$

c) By the Master Method: T(n) = aT(n/b) + f(n)
T(n) = 2T(n/2) + c
a = 2
b = 2
f(n) = c
n^(log2(2)) = n^1 = n
f(n) = theta(n^(logb(a-e)) for some e > 0, then T(n) = theta(n^(logb(a))
Therefore,
T(n) = theta(n)

The running time of the recursive min_and_max algorithm is the same as the iterative algorithm for finding the minimum and maximum values of an array.

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.

b) Would STOOGESORT still sort correctly if we replaced m = ceiling(2n/3) with m = floor(2n/3)? If yes prove if no give a counterexample. (Hint: what happens when n = 4?)

c) State a recurrence for the number of comparisons executed by STOOGESORT.

d) Solve the recurrence to determine the asymptotic running time.

Answers:

a) The Stoogesort algorithm starts out by checking if the first element is larger that the element at the end. If this is true then it swaps them. If there are 3 or more elements in the array, it proceeds to sort the first 2/3s of the array. It then sorts the final 2/3 of the array. It the goes back and sorts the first first 2/3 of the array again.

b) The STOOGESORT would NOT still sort correctly if we replaced m = ceiling(2n/3) with m = floor(2n/3). There would be no overlap between the first sort of the front 2/3rds of the array and the end 2/3rds of the array. This would mean that, in an array of 4, element at index 1 and element at index 2 would not be compared to each other and would not be properly sorted.

c) T(n) = 3T(2n/3) + c

d) By the Master Method: T(n) = aT(n/b) + f(n)
T(n) = 3T(2n/3) + c
a = 3
b = 3/2
f(n) = c
n^(logb(a)) = n^(log(3/2)(3)) = n^2.71
if f(n) = O(n^logb(a-e)) for some e > 0 then
T(n) = theta((n^logb(a))
Therefore,
T(n) = n^2.71

Resource: I used this link to better help me understand the algorithm.
https://en.wikipedia.org/wiki/Stooge_sort

5) (10 points)

a) Implement STOOGESORT from problem 4 to sort an array 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".

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.

c) 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.

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?

Answers:

a) See TEACH for code files.

b) Below is the code from stoogesort_graphs.py

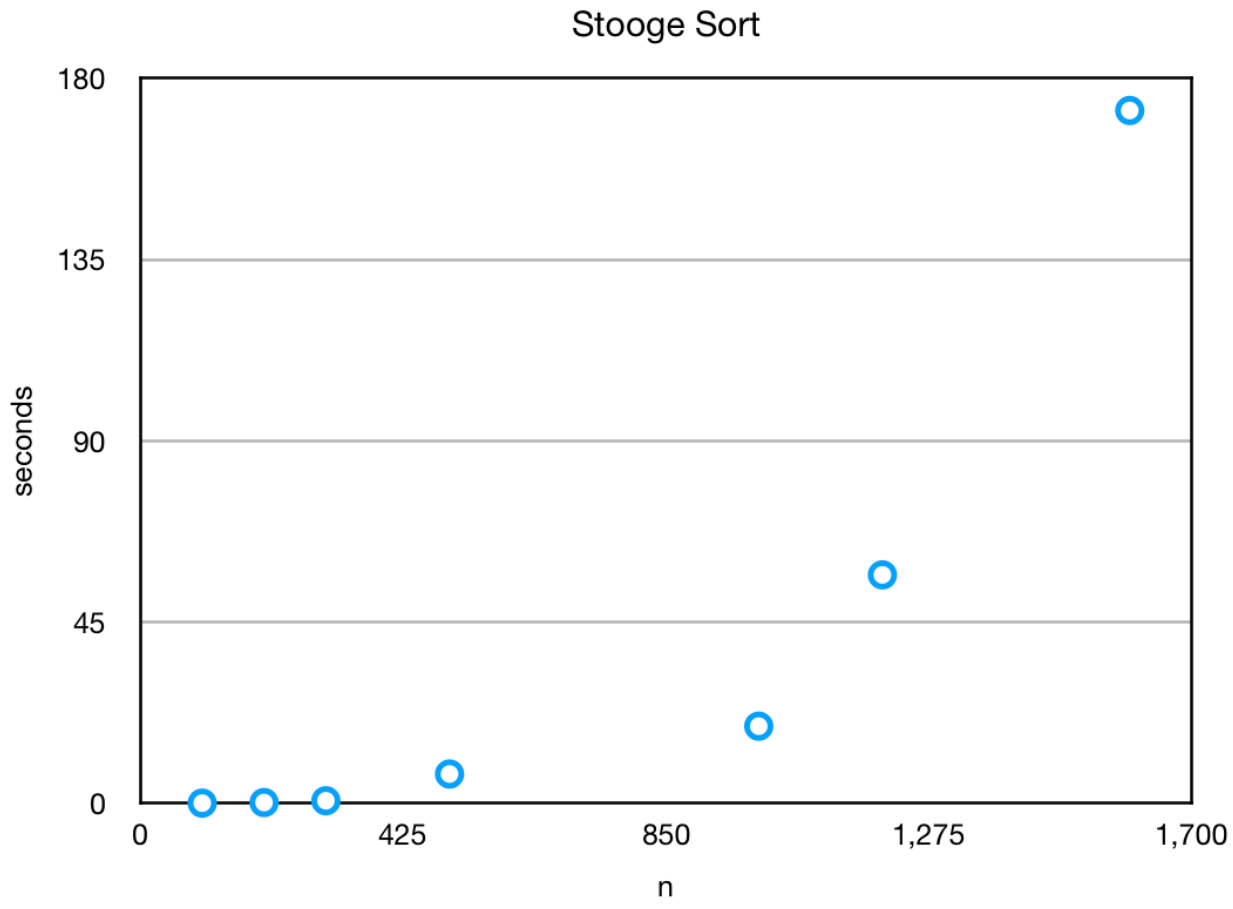
```
'''
Jon-Eric Cook
CS-325
Homework #2
This program demonstrates the stooge sort algorithm. It does this by first
checking if the first element is larger that the element at the end. If this is
true then it swaps them. If there are 3 or more elements in the array, it
proceeds to sort the first 2/3s of the array. It then sorts the final 2/3 of
the array. It the goes back and sorts the first first 2/3 of the array again.
This program will be sorting arrays of length n that contain random numbers.
'''
import random
import timeit

# wrapper function
def wrapper(func, *args, **kwargs):
    def wrapped():
        return func(*args, **kwargs)
    return wrapped

# stoogeSort algorithm
def stoogeSort(a,low,high):
    # swaps elements
    if a[low] > a[high]:
        temp = a[low]
        a[low] = a[high]
        a[high] = temp

    # if there are more than two elements in the array
    if (high - low + 1) > 2:
        fraction = (int)((high - low + 1.0)/3)
        stoogeSort(a,low,high-fraction)
        stoogeSort(a,low+fraction,high)
        stoogeSort(a,low,high-fraction)

    return a

# arrays to hold times from sort functions
stoogesort_data = []

# number of elements to be in random array
elements = [100,200,400,800,1600,3200,6400]

# loop 7 times, timing the run time of each sort algorithm for 7 different
# array sizes
for x in range(7):
    # randomly generate arrays to be sorted for mergesort and insertsort
    rand_list_stoogesort = [int(1000*random.random()) for i in xrange(elements[x])]

    # wrap mergesort function
    wrapped = wrapper(stoogeSort, rand_list_stoogesort, 0, len(rand_list_stoogesort) - 1)
    # time mergesort function
    time_taken = timeit.timeit(wrapped, number=1)
```

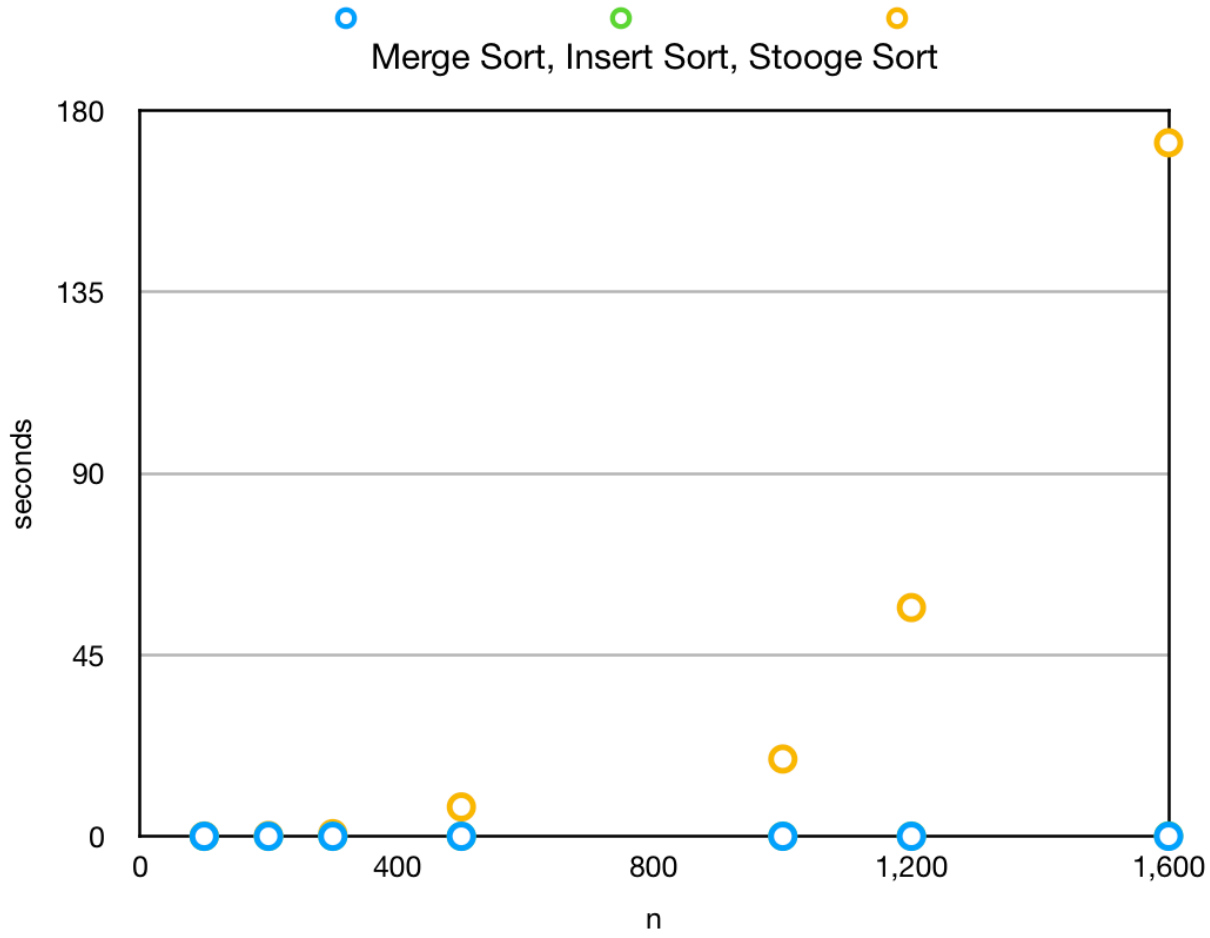
```
    # log number of elements and time taken for mergesort
    n = "n = "
    n += str(elements[x])
    stoogesort_data.append(n)
    sec = "sec = "
    sec += str(time_taken)
    stoogesort_data.append(sec)

print "Stooge Sort"
print stoogesort_data
```

c)

Answers:

| **MergeSort** | | **InsertSort** | | **StoogeSort** | |
|---|---|---|---|---|---|
| n | seconds | n | seconds | n | seconds |
| 100 | 0.00042 | 100 | 0.00043 | 100 | 0.07036 |
| 200 | 0.00086 | 200 | 0.00161 | 200 | 0.21421 |
| 300 | 0.00138 | 300 | 0.00306 | 300 | 0.65780 |
| 500 | 0.00215 | 500 | 0.00904 | 500 | 7.29266 |
| 1000 | 0.00589 | 1000 | 0.0335 | 1000 | 19.16655 |
| 1200 | 0.00584 | 1200 | 0.04952 | 1200 | 56.72244 |
| 1600 | 0.00946 | 1600 | 0.09154 | 1600 | 171.98292 |

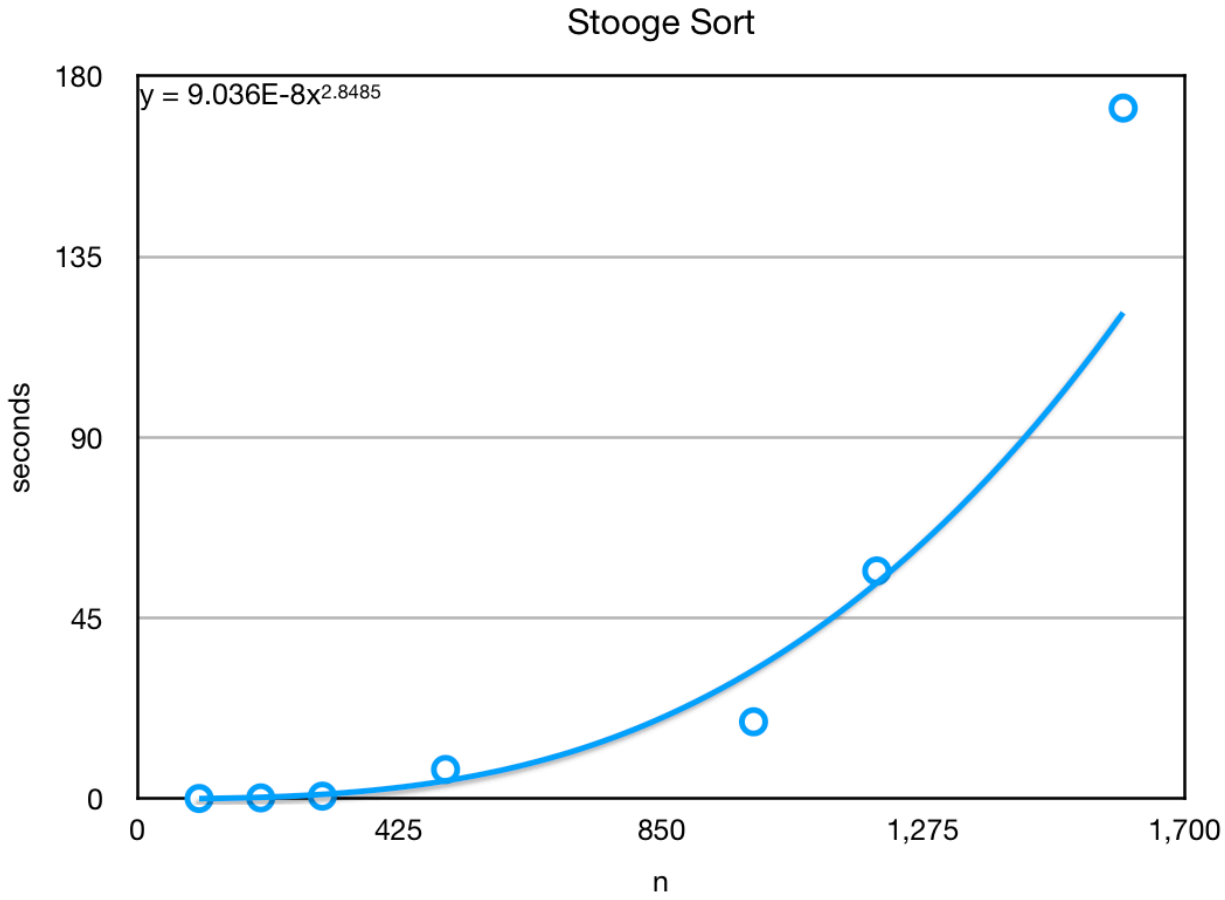





d)

Answers:

A power curve best fits the StoogeSort data set. The equation of the curve that best “fits” the data is y = x^2.8485. The curve can be seen below. Comparing the experimental running time to that of the theoretical running time of the Stooge Sort algorithm, they are pretty close. The experimental was y = x^2.8485 and the theoretical was y = x^2.71.




# RESOURCES:
# The below links were used to help me complete this problem.
# https://rosettacode.org/wiki/Sorting_algorithms/Stooge_sort#Python
# https://en.wikipedia.org/wiki/Stooge_sort
# http://www.geeksforgeeks.org/stooge-sort/