Josh Huff, huffj@oregonstate.edu
Professor Schutfort
CS 325 – Winter 2018
January 21, 2018

Homework 2

1.

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

aT(n-b) + f(n)
a = 2; b = 2; f(n) = 1 = n^0; d = 0
Muster Method – Case 3
T(n) = $\Theta(n^0 2^{n/2})$
T(n) = $\Theta(2^{n/2})$

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

aT(n-b) + f(n)
a = 1; b = 1; f(n) = n^3 ; d = 3
Muster Method – Case 2
T(n) = $\Theta(n^{d+1})$
T(n) = $\Theta(n^4)$

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

aT(n/b) + f(n)
a = 2; b = 6; f(n) = $2n^2$
Master Method – Case 3
T(n) = $\Theta(2n^2)$

2.

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

The quaternary search algorithm recursively establishes three bounds within an array until the subarray being considered has the sought element in one of those bounds, or until the array is expended (in which case the element is not in the array)

Start with a sorted array.
Find the midpoint of the array by adding the lowest and highest bound and dividing by two.
Find the low-mid point of the array by adding the lowest and midpoint bounds and dividing by two.
Find the mid-high point of the array by adding the midpoint and highest bounds and dividing by two.
Quarter 1 ranges from the lowest point up to (but not including) the low-mid point.
Quarter 2 ranges from the low-mid point up to (but not including) the mid point.
Quarter 3 ranges from the mid point up to (but not including) the mid-high point.
Quarter 4 ranges from the mid-high point up to the high point.
If the element at the midpoint is greater than the target, then it is either in Quarter 1 or Quarter 2.
    If the element at the low-mid point is greater than the target, begin at the start with Quarter 1.
    If the element at the low-mid point is less than the target, begin at the start with Quarter 2.
    If the low-mid point is neither greater nor less than the target, it is equal. Return this value.
If the element at the midpoint is less than the target, then it is either in Quarter 3 or Quarter 4.
    If the element at the mid-high point is greater than the target, begin at the start with Quarter 3.
    If the element at the mid-high point is less than the target, begin at the start with Quarter 4.
    If the mid-high point is neither greater nor less than the target, it is equal. Return this value.
If the element at the midpoint is neither greater nor less than the target, it is equal. Return this value.
If the highest bound is exceeded by the lowest bound, the element is not to be found and the search returns a False.

Pseudocode

```
def quaternary_search(in_array, target, low, high):
    if (high < low): return -1

    mid = (low + high) // 2
    low_mid = (low + mid) // 2
    mid_high = (mid + high) // 2

    if in_array[mid] > target:
        if in_array[low_mid] > target:
            return quaternary_search(in_array, target, low, low_mid - 1)
        elif in_array[low_mid] < target:
            return quaternary_search(in_array, target, low_mid + 1, mid)
        else: return low_mid
    elif in_array[mid] < target:
        if in_array[mid_high] > target:
            return quaternary_search(in_array, target, mid, mid_high - 1)
        elif in_array[mid_high] < target:
            return quaternary_search(in_array, target, mid_high + 1, high)
        else: return mid_high
    else: return mid
```

b) Give the recurrence for the quaternary search algorithm

$T(n) = T(n/4) + O(1)$

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.

$aT(n/b) + f(n)$
$a = 1; b = 4; f(n) = 1$
Master Method – Case 2
$T(n) = \Theta(\lg n)$

Quaternary search and binary search differ only by a constant factor and thus have asymptotically equal running times.

3.
a. Verbally describe and write pseudo-code for the min_and_max algorithm.

The D&C min/max algorithm recursively splits the given array in half until a given subarray is one element in length, then compares that array's sole value against the current minimum and maximum, naming the value the new min or max if appropriate.

Start with an array (sortedness is not assumed).
Find the length of the array.
If the length is greater than one, then it is must be divided further.
    Find the midpoint of the array by adding the lowest and highest bound and dividing by two.
    Half 1 ranges from the lowest point up to (but not including) the mid point.
    Half 2 ranges from the mid point up to the end.
    Begin at the start with half 1, then with half 2.
If the length is one, then the value is ready to be compared with the current min and max.
    If there is no established minimum, or if the value is lower than the current min, then the value becomes the minimum.
    If there is no established maximum, or if the value is higher than the current max, then the value becomes the maximum.

Pseudocode

```
def compare_min(curr_elem, curr_min):
    global result
    if curr_min == None or curr_elem < curr_min:
        curr_min = curr_elem
    result[0] = curr_min

def compare_max(curr_elem, curr_max):
    global result
    if curr_max == None or curr_elem > curr_max:
        curr_max = curr_elem
    result[1] = curr_max

def min_and_max(in_array, beginning, end):
    global result
    n = end - beginning + 1
    if n == 1:
        curr_min = compare_min(in_array[beginning], result[0])
        curr_max = compare_max(in_array[beginning], result[1])
    elif n > 1:
        middle = (beginning + end) // 2
        min_and_max(in_array, beginning, middle)
        min_and_max(in_array, middle + 1, end)
```

b) Give the recurrence.

$T(n) = 2T(n/2) + \Theta(1)$

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.

$aT(n/b) + f(n)$
$a = 2; b = 2; f(n) = 1$
Master Method – Case 1
$T(n) = \Theta(n)$

An iterative algorithm would take $\Theta(n)$ time to find the min and max values of an unsorted array. An iterative algorithm would work in constant time on a sorted array, however (because it could just retrieve the first and last array offsets), and the D&C implementation would run at the comparatively worse linear time.

4.

a) Verbally describe how the STOOGESORT algorithm sorts its input.

StoogeSort recursively passes the first two thirds of an array to itself as a subarray, until the subarray being called is two elements in length (that is, the base case is n = 2), and swaps the elements if they are not already in ascending order. Then the process repeats for the final two thirds of an array, followed up by a second recursive call on the first two thirds of an array.

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 the m variable were determined using the floor of 2n/3. Instead, it would properly sort adjacent elements but not the whole array. For instance, an array containing elements [9, 6, 8, 7, 10, 5] would be sorted as such: [6, 9, 7, 8, 5, 10].

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

$T(n) = 3T(2n/3) + \Theta(1)$

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

$T(n) = 3T(2n/3) + \Theta(1)$
$aT(n/b) + f(n)$
$a = 3; b = 3/2; f(n) = 1$
Master Method – Case 1
$T(n) = \Theta(n^{2.7})$

5.

Stooge Sort Timed

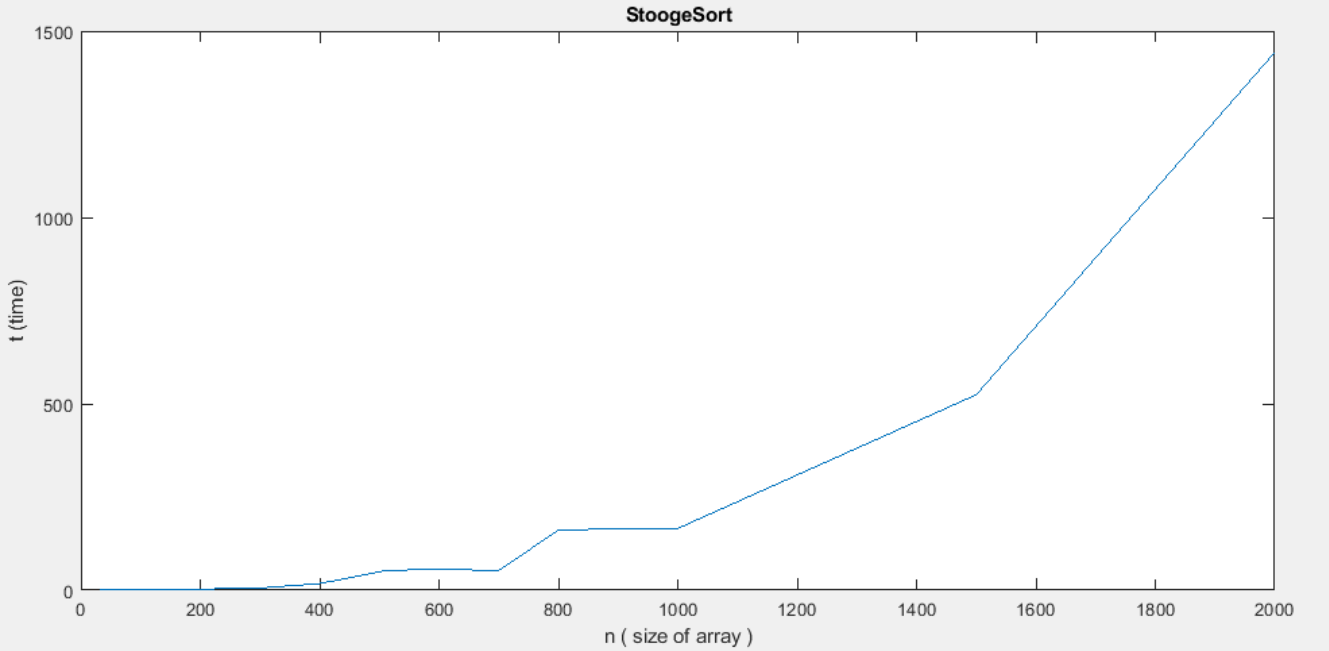
```
import math
import random
import sys
from timeit import Timer

def stoogesort(in_array, beginning, end):
    """Adapted from pseudocode found in HW2"""
    n = end - beginning + 1

    # if n = 2 and A[0] > A[1]
    if n == 2 and in_array[beginning] > in_array[end]:
        #Swap A[0] and A[1]
        in_array[beginning], in_array[end] = in_array[end], in_array[beginning]

    # else if n > 2
    elif n > 2:
        # m = ceiling(2n/3)
        m = math.ceil((2 * n) / 3)

        # StoogeSort(A[0 ... m - 1)
        stoogesort(in_array, beginning, beginning + (m - 1))

        # StoogeSort(A[n - m ... n - 1])
        stoogesort(in_array, beginning + (n - m), end)

        # StoogeSort(in_array[0 ... m-1)
        stoogesort(in_array, beginning, beginning + (m - 1))

if __name__ == "__main__":

    # First non-scriptname argument passed on the command line will serve as n
    n = int(sys.argv[1])

    # Populate an array of size n with zeroes,
    # Then fill it with random integers between zero and 10,000
    array = [0] * n
    for i in range(0, n):
        array[i] = random.randint(0, 10000)

    # Pass stoogesort call to Timer object, run it ten times, and store the average
    timer = Timer('stoogesort(array, 0, len(array) - 1)', globals=globals())
    average = timer.timeit(10)

    # Append the result to the stooge-runs file
    with open("stooge-runs.txt", "a") as f2:
        f2.write("Average run-time for " + str(n) + " inputs " + str(average))
        f2.write("\n")
```

5b.

Stooge Sort (selected run-times with each *n* 10 times; the times below are averages of these trials.)

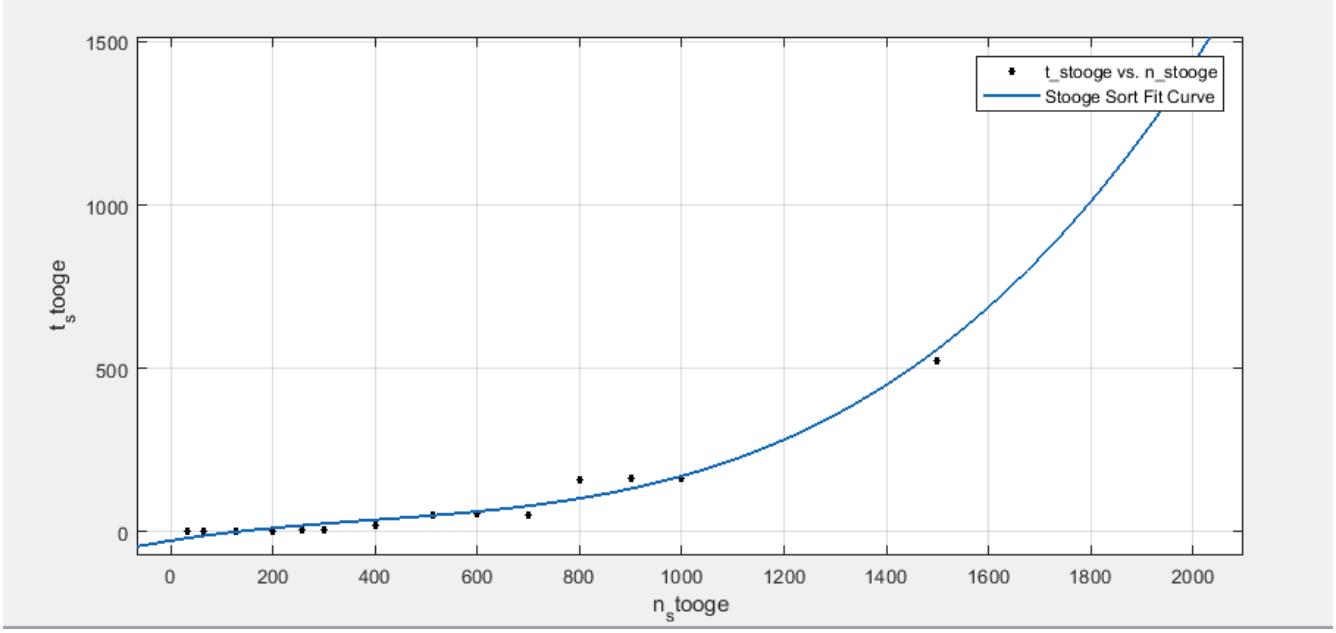
Average run-time for 200 inputs	2.0483002119872253
Average run-time for 400 inputs	17.821185598993907
Average run-time for 600 inputs	56.06648110298556
Average run-time for 800 inputs	161.16891549501452
Average run-time for 1000 inputs	165.7478428290051
Average run-time for 1500 inputs	524.3409846559807
Average run-time for 2000 inputs	1442.6982034709945

5c.



5d.

StoogeSort Curve Fit – f(x) = p1 * x^3 + p2 * x^2 + p3 * x + p4 (confidence >99%)



Though there is a “step-wise” trend in practice when running the StoogeSort algorithm as a result of the ceiling division in determining the m variable, the actual run-time is very close to the theoretical otherwise.