CS352 (Winter 2018) - Homework 2

Marc Tibbs (tibbsm@oregonstate.edu)

Due Date: January 21, 2018

Problem 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 cases T(0)=1 and/or T(1) = 1.

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

$a = 2, b = 2, f(n) = 1$, so $d = 0$

$T(n) = \mathcal{O}(n^d a^{\frac{n}{b}}) = \mathcal{O}(n^0 2^{\frac{n}{2}}) = \mathcal{O}(2^{\frac{n}{2}})$

$\in \mathcal{O}(2^{\frac{n}{2}})$

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

$a = 1, b = 1, d = 3$

$T(n) = \mathcal{O}(n^{d+1}) = \mathcal{O}(n^4)$

$\in \mathcal{O}(n^4)$

(c) $T(n) = 2T(\frac{n}{6}) + 2n^2$

$a = 2, b = 6 \Rightarrow n^{log_b a} = n^{log_6 2} = 0.387; f(n) = 2n^2$

$T(n) = \Theta(n^2)$

Problem 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.

A quaternary search algorithm would split the input into four sections by calculating the 1/4, 1/2, and 3/4 markers of the input. To do this the algorithm would need to be passed the sorted array of numbers to be searched, the minimum index of the array, the maximum index of the

array, and the search value. The algorithm would first check to see if any of the values at 1/4 index, 1/2 index, or 3/4 index in the array match the search value. If none of the values match, the algorithm would find the appropriate section of the array to continue the search in by comparing the search value with the 1/4, 1/2, 3/4 indexed values. It would continue splitting and searching the array until value is found or the array could not be split any more.

```
quatSearch(A[0...n-1], lo, hi, find)
    while hi ≥ 1
        onef = lo + ((hi - lo) / 4)
        half = lo + ((hi - lo) / 2)
        threef = lo + ((hi - lo) / 4)
        if A[onef] = value
            return A[onef]
        else if A[half] = value
            return A[half]
        else if A[threef] = value
            return A[threef]
        else if A[onef] > value
            return quatSearch(A, lo, onef-1, find)
        else if A[half] > value
            return quatSearch(A, onef+1, half-1, find)
        else if A[threef] > value
            return quatSearch(A, half+1, threef-1, find)
        else
            return quatSearch(A, threef+1, hi, find)
```

(b) Give the recurrence for the quaternary search algorithm

$$T(n) = T(n/4) + \Theta(1)$$

(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.

$T(n) = T(\frac{n}{4}) + c$
$T(n/4) = T(\frac{n}{16}) + 2c$
$T(n) = T(n/4^k) + kc$
$n = 4^k, T(\frac{n}{4^k}) = T(\frac{4^k}{4^k}) = T(1)$
$k = log_4 n$
$T(n) = T(1) + log_4 nc$
$T(n) = \Theta(log_n)$ which is the same as binary search.

Problem 3: *(5 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.

A divide and conquer algorithm that determines the min and max of an unsorted list could function in a very similar way to a binary search algorithm. The min and max algorithm would need to receive an array of the list of numbers, the minimum and maximum indexes of the array, and a struct holding the minimum and maximum variables. Using those values, the algorithm would then would split the list of numbers comparing and keeping track of the min and max variables until the sub-arrays cannot be split anymore. In other words the algorithms will recurse until there is only one number in the sub-array. The psuedo-code for such a algorithm is displayed below:

```
min_and_max(A[0...n-1], lo, hi, minmax)
    if (lo==hi)
        minmax.max = A[lo]
        minmax.min = A[lo]
        return minmax
    else if (lo==hi-1)
        if (A[lo]>A[hi])
            minmax.min = A[hi]
            minmax.max = A[lo]
        else
            minmax.min = A[lo]
            minmax.max = A[hi]
    else
        half = (lo + hi)/2
        minmax1 = min_and_max(A, lo, half, minmax)
        minmax2 = min_and_max(A, half+1, hi, minmax)
        if (minmax1.min < minmax2.min)
            minmax.min = minmax1.min
        else
            minmax.min = minmax2.min
        if (minmax1.max > minmax2.max)
            minmax.max = minmax1.max
        else
            minmax.max = minmax2.max
    return minmax
```

(b) Give the recurrence.

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

(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.

$T(n) = 2T(n/2) + 1$
$T(n/4) = 2(2T(n/2) + 1) + 1$
$T(n/4) = 2^2T(n/2) + 2^1 + 1$
$T(n) = 2^kT(n/2^k) + 2^{k-1} + 2^{k-2}... + 1$
$n = k^2, T(n/k^2) = T(k^2/k^2) = T(1)$
$k = log_2(n)$
$T(n) = 2^kT(1) + 2(2^{k-1}) + 1$
$T(n) = 2^k + 2(2^{k-1}) + 1$
$T(n) = n + 2(n-1) + 1$
$T(n) = n + n$
$T(n) = \Theta(n)$

Problem 4: *(5 points)* Consider the following pseudo-code 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([0...m − 1])
        StoogeSort([n − m...n − 1])
        StoogeSort([0...m − 1])
```

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

The STOOGESORT algorithm starts by checking the base case, which is if the array is only has two items. If the array only has two items then the array checks the two values and puts them in the correct position using swap. Otherwise, the algorithm will split the array into two parts: one portion representing 2/3's of the original array and the other representing the latter 1/3 of the original array. The algorithm will recurse on those two sections of the passed array. First it will recurse into the 2/3 portion of the array, then the 1/3 portion, and then the 2/3 portion once more. This will continue until the sub-arrays have been split into only two values.

(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 code interchanged k = ceiling(2n/3) with k = floor(2n/3). For example, when $n = 4$ using the floor method, STOOGESORT would recurse on the [0,1],[2,3],[0,1] sections of the array. Since there is no overlap on the sub-arrays the sort function would be unable to compare [1,2] and making it an uncompleted sort. In contrast using the ceiling the sort is run on the [0,2],[1,3], and [0,2] portions of the array allowing for a complete sort of the array.

(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.

$$T(n) = 3T(2n/3) + \Theta(1)$$
$$a = 3, b = 1.5, log_b a = 2.7...$$
$$T(n) = \Theta(n^{log_{1.5}3}) = \Theta(n^{2.7...})$$

Problem 5: *(5 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".

Copy of code files and README file that explains how to compile and run the code are compressed in a ZIP file and have been submitted to TEACH. Program runs with input file named data.txt.

(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.

Stooge Sort Running Time Analysis Code in Appendix A

(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.

integers(n)	Merge Sort time(secs)	Insertion Sort(secs)	Stooge Sort time(secs)
500	0.000091	0.000353	0.15898
1,000	0.000199	0.001365	0.46656
1,000	0.000317	0.002940	1.38275
2,000	0.000407	0.004948	4.28249
2,000	0.000495	0.008411	12.4292
3,000	0.000606	0.011236	12.2208
3,000	0.000719	0.017696	12.2005
4,000	0.000985	0.020637	36.8044
4,000	0.001121	0.026712	37.7713
5,000	0.001248	0.028282	38.2733

Graphs in Appendix B

(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?

The power trendline $y = 6E - 09x^{2.6947}$ seems to fit the data the best with a R^2 value of 0.98271. This equation is very close to the theoretical running time of $T(n) = \Theta(n^{2.7\cdots})$.

Appendix A: Modified Code for Testing Sorting Algorithm Times :

```
void stoogeSort(int A[], int lo, int hi) {

    int size = (hi - lo + 1);

    //Base case
    if (size == 2 && A[lo]>A[hi]) {
        std::swap(A[lo],A[hi]);
    }
    //Recurse
    else if (size > 2){
        int m = std::ceil(double(2*size)/3);
        stoogeSort(A, lo, lo+m-1);          //Sort lower array
        stoogeSort(A, hi-m+1, hi);         //Sort higher array
        stoogeSort(A, lo, lo+m-1);          //Sort lower array
    }
}

int main(int argc, const char * argv[]) {

    std::ofstream outfile("stoogeTimesMatch.txt");
    const int arrSize = 1000002;
    int array[arrSize];

    for (int k = 0; k <= 5000; k=k+500){
        float total = 0, average = 0;

        outfile << k << ", " ;
        std::cout << k << ", ";

        for (int i = 1; i <= 3; i++){
            //Create array with random ints
            srand(time_t(NULL));

            for (int j = 0; j < k; j++){
                array[j] = rand() % 1000;
            }
            clock_t t1, t2;
            t1 = clock();

            stoogeSort(array, 0, k-1);

            t2 = clock();

            float diff ((float)t2 - (float)t1);     //Total time
            float seconds = diff/CLOCKS_PER_SEC;
            total = total + seconds;
            outfile << seconds << ", ";
            std::cout << seconds << ", ";
        }
        average = total / 3;
        outfile << average << std::endl;
        std::cout << average << std::endl;
    }
    outfile.close();
    return 0;
}
```

Appendix B: Graphs of the Sorting Algorithm Times

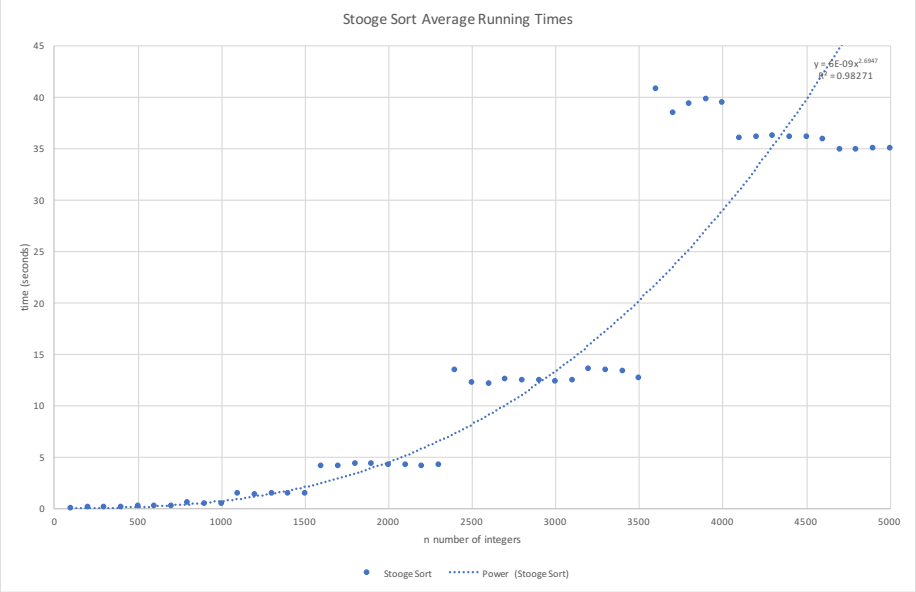

Figure 1: Stooge Sort Average Times

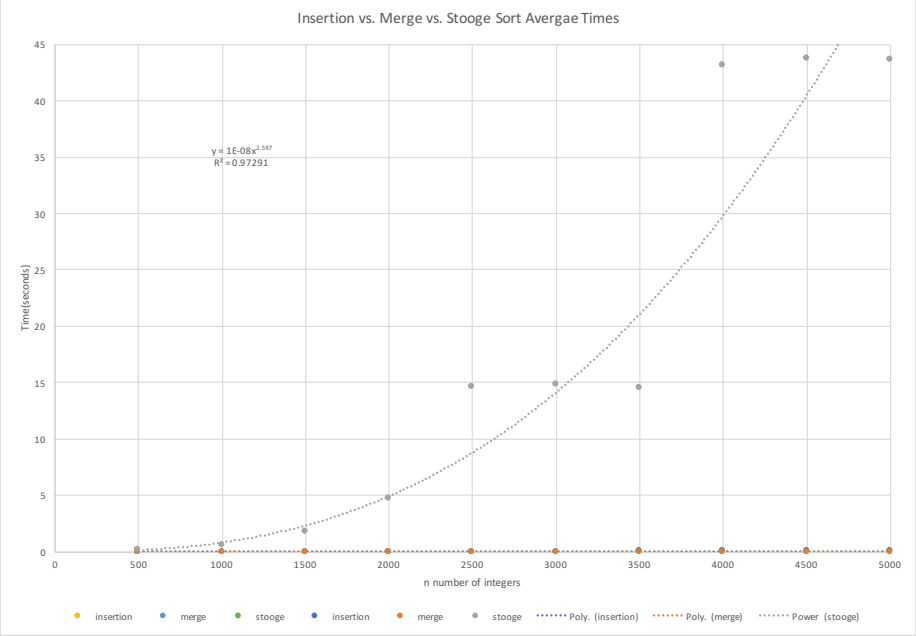

Figure 2: Insertion vs. vs. Merge vs. Stooge Sort Average Times

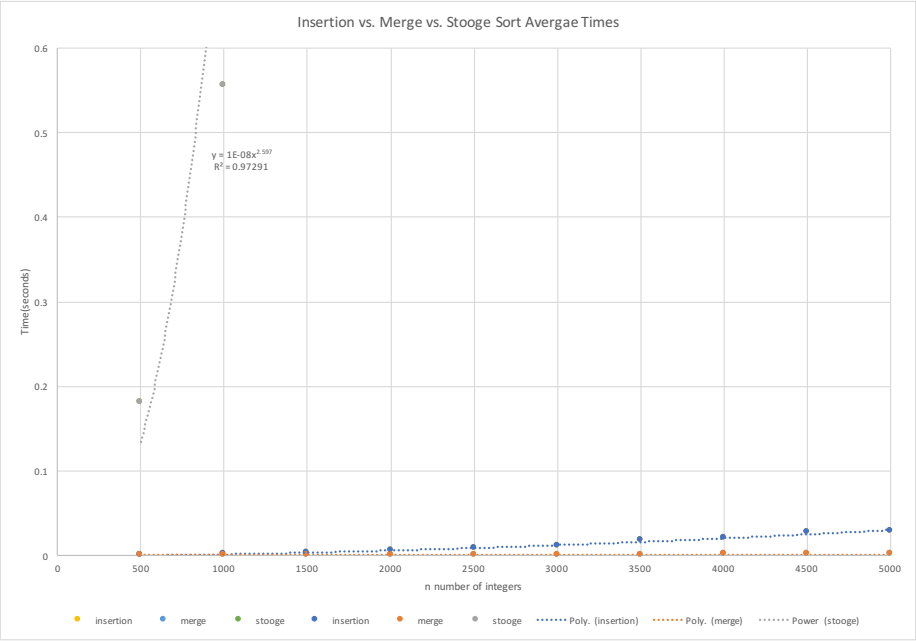

Figure 3: Insertion vs. vs. Merge vs. Stooge Sort Average Times

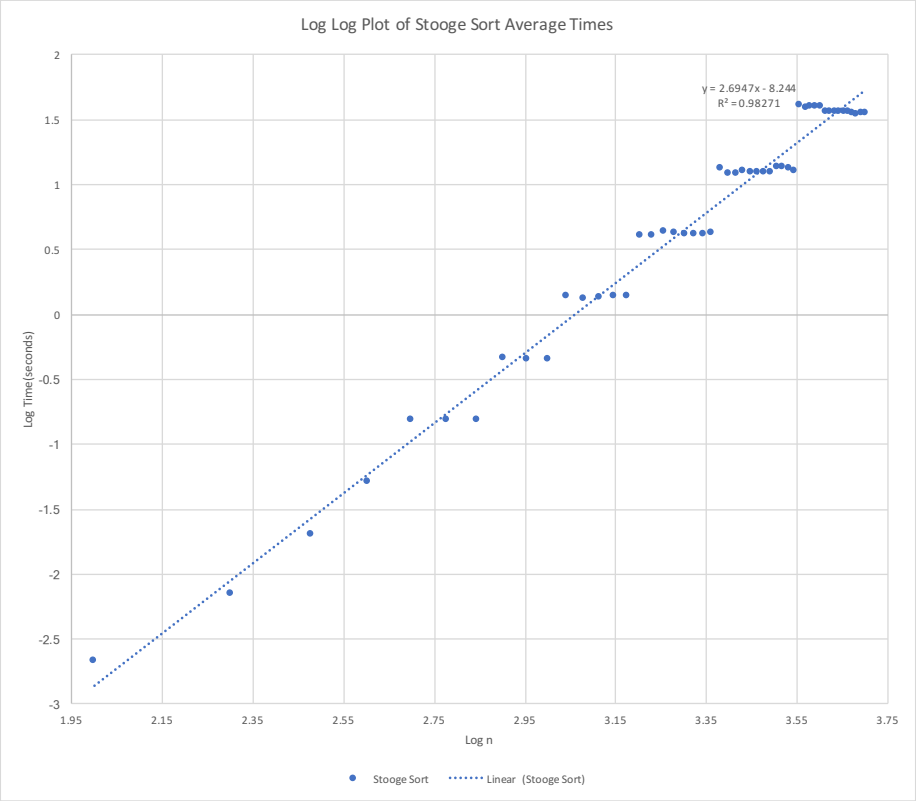

Figure 4: Log Log Plot of Stooge Sort Average Times

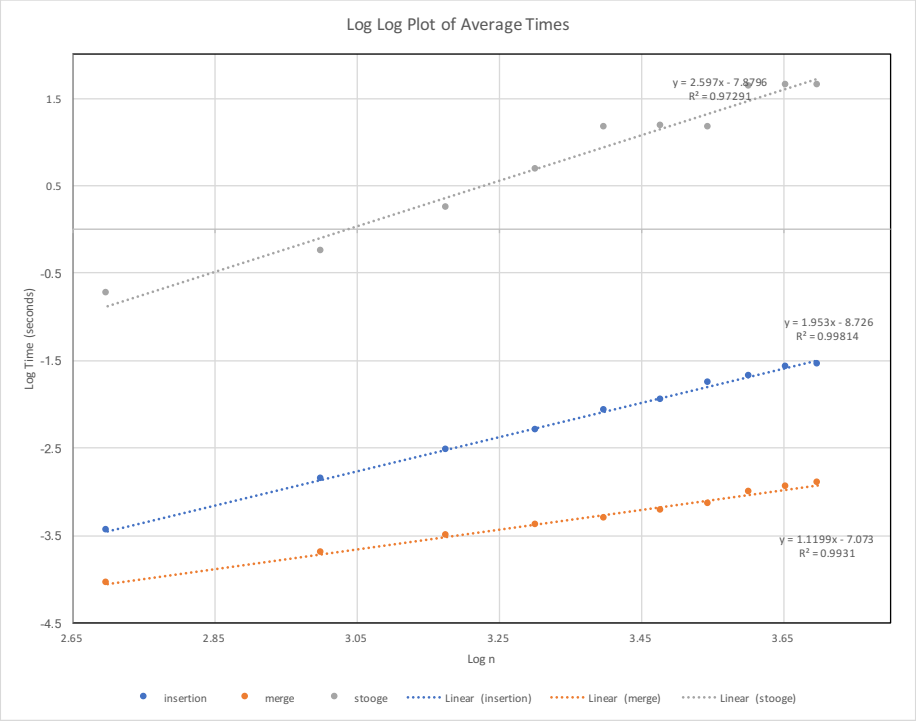

Figure 5: Log Log Plot of Insertion vs. vs. Merge vs. Stooge Sort Average Times