Gretel Rajamoney

Analysis of Algorithms

January 16, 2021

Homework #2

Problem 1:

a. In order to implement a divide and conquer min_and_max algorithm, we first start by determining the size of our given dataset. If the list contains only one element, size = 1, then the minimum as well as the maximum for that data set would be the same. If the list contains only two elements, size = 2, then we will find the minimum and maximum by comparing the two elements to find which one is bigger or smaller, if they are both of the same value then our maximum and minimum will be the same. If we are given a dataset containing more than two elements, size > 2, then we will find the maximum and minimum by dividing our list into two separate sections. We will transverse both separate lists comparing elements repeatedly until a minimum and maximum has been reached. Lastly, we will compare the minimum and maximum of both lists in order to arrive at the official minimum and maximum of the entire dataset.

```
size = number of elements in data.txt

if (size = 0)
{
	return "no minimum or maximum due to empty dataset"
}

if (size = 1)
{
	maximum = only element in data.txt
	minimum = only element in data.txt

	return maximum, minimum
}

if (size = 2)
{
	if (first element > second element)
	{
		maximum = first element in data.txt
		minimum = second element in data.txt
	}

	if (second element > first element)
	{
		maximum = second element in data.txt
		minimum = first element in data.txt
```

```
        }

        if (first element == second element)
        {
                maximum = first element in data.txt
                minimum = first element in data.txt
        }

        return maximum, minimum
}

if (size > 2)
{
        for (first half of data.txt)
        {
                transverse array until minimum and maximum are found
                return maximumFirstHalf, minimumFirstHalf
        }

        for (second half of data.txt)
        {
                transverse array until minimum and maximum are found
                return maximumSecondHalf, minimumSecondHalf
        }

        if (maximumFirstHalf > maximumSecondHalf)
        {
                maximum = maximumFirstHalf
        }
        else
        {
                maximum = maximumSecondHalf
        }

        if (minimumFirstHalf < minimumSecondHalf)
        {
                minimum = minimumFirstHalf
        }
        else
        {
                minimum = minimumSecondHalf
        }

        return maximum, minimum
}

print maximum & minimum

end & exit program
```

b. $T(n) = \begin{cases} 2 \cdot T\left(\frac{n}{2}\right) + c, n \geq 2 \\ T(1),\ n = 1 \end{cases}$

c. Assume c = 1 since it represents a constant time.

$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + 1$

$T(n) = 2 \cdot \left(2 \cdot T\left(\frac{n}{4}\right) + 1\right) + 1$

$T(n) = 2 \cdot \left(2 \cdot \left(2 \cdot T\left(\frac{n}{8}\right) + 1\right) + 1\right) + 1$

$T(n) = 2 \cdot \left(2 \cdot \left(2 \cdot \left(2 \cdot T\left(\frac{n}{16}\right) + 1\right) + 1\right) + 1\right) + 1$

… repeats recursively until complete

For the i^{th} recursion we get the formula $T_i(n) = 2^i \cdot T\left(\frac{n}{2^i}\right) + i$

Next, we must calculate for what values of i can $\frac{n}{2^i}$ be equal to the constant 1

$\frac{n}{2^i} = 1$

$n = 2^i$

$i = \log_2(n)$

Finally, we must submit $i = \log_2(n)$ into our recursion formula above

$T_i(n) = 2^{log_2 n} \cdot T\left(\frac{n}{2^{log_2 n}}\right) + \log_2 n$

$T_i(n) = n^{log_2 2} \cdot T\left(\frac{n}{n^{log_2 2}}\right) + \log_2 n$

$T_i(n) = n + \log_2 n$

The recursive formula can be compared to the iterative method, because the iterative method has a time complexity of calculating the min_and_max algorithm in O(n) time, meanwhile, the recursive method has a time complexity of calculating the min_and_max algorithm in O($n + \log_2 n$) time.

Problem 2:

```
MergeSort3 Function
{
	if (number of values in data.txt = 0)
	{
		return nothing
	}

	copy all elements of data.txt into new array

	if (number of values in data.txt = 1)
	{
		return nothing;
	}

	split1 = 1/3 point of array
	split2 = 2/3 point of array

	mergesort3(first third of array)
	mergesort3(second third of array)
	mergesort3(third third of array)
}

Merge3 Function
{
	min1, min2, min3 represent the minimums of all arrays

	if (min1 < min2 < min3)
	{
		return min1
	}

	else
	{
		sort to find the smallest value out of them all
	}

	transverse through all three sections until sorted
```

a. }

b. MergeSort3 calls itself 3 times, and then merges one time. Therefore, the recurrence relation can be represented by the equation:

$$T(n) = 3T\left(\frac{n}{3}\right) + n$$

c. $T(n) = 3T\left(\frac{n}{3}\right) + n$

$g(n) = n\,log_b\,a$

$g(n) = n\,log_3\,3$

$T(n) = O(n\,log_3\,n)$

Problem 3:

Completed, submitted as a zipped file named mergesort3.cpp on TEACH !!

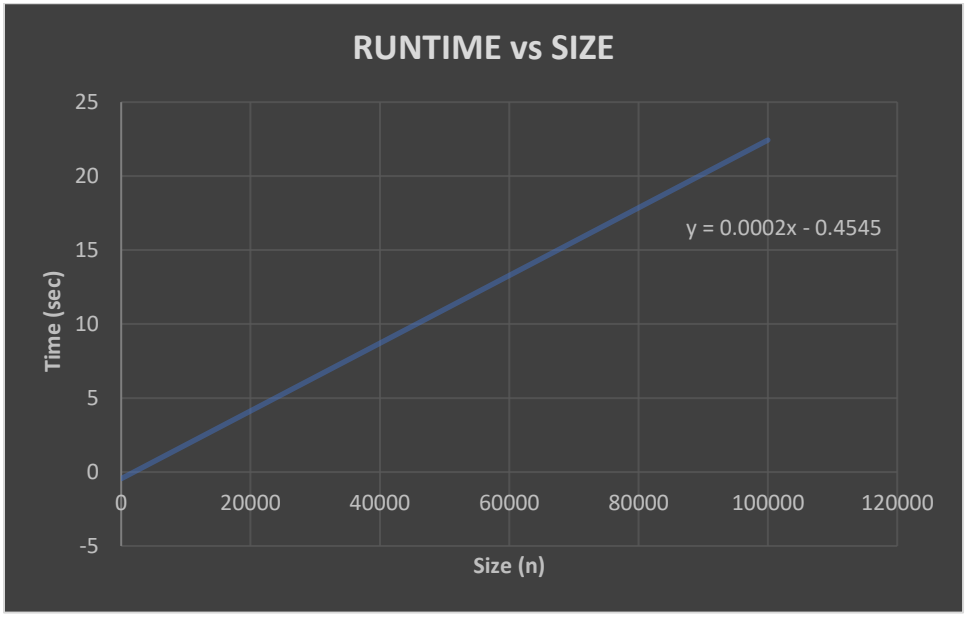
Problem 4:

a. Modified, submitted as a zipped file names mergesort3time.cpp on TEACH !!

b.

Size (n)	Runtime (sec)
0	0
10000	1.81818
20000	4.54545
30000	6.36364
40000	8.18182
50000	10.9091
60000	13.6364
70000	13.6364
80000	17.2727
90000	21.8182
100000	22.7273

c.



Linear Line of Best Fit: $y = 0.0002x - 0.4545$

d. According to my data plotted on the graph below, my three-way merge sort runs significantly faster than the original merge sort created for homework 1. Since both programs utilize the same master merge method, their theoretical runtimes simplify down to $\theta(n \log n)$. However according to our experimental runtimes, the three-way merge runtimes show that it is clearly much faster than the original merge sort, when the size of n gets larger the runtime differences between the two methods begins to show more apparently as well.