Problem 1:

a)

```
function getMaxMin(low,. high. array)
    If size_of_arr = 1
        return element as max and min
    else if size_of_arr = 2
        If arr[0] > arr[1]
            return max and min
        else
            return max and min
    else
        recurse getMaxMin for left
        recurse getMaxMin for right
    compare max and min of both left and right halves
    return true max and min
```

b)

$$T(n) = \begin{cases} 2 * T(n/2) + c, & n \geq 2 \\ T(1), & n = 1 \end{cases}$$

c)

$$O(n)$$

Problem 2:

a)

```
function mergesort3(arr, start, end)
    if length between start and end < 2
        return arr
    else
        m1 = calculate index that divide left and middle sections
        m2 = calculate index that divides middle and right sections

        recurse mergesort3 for left
        recurse mergesort3 for middle
        recurse mergesort3 for right

        merge the sections together

        return arr
```

b)

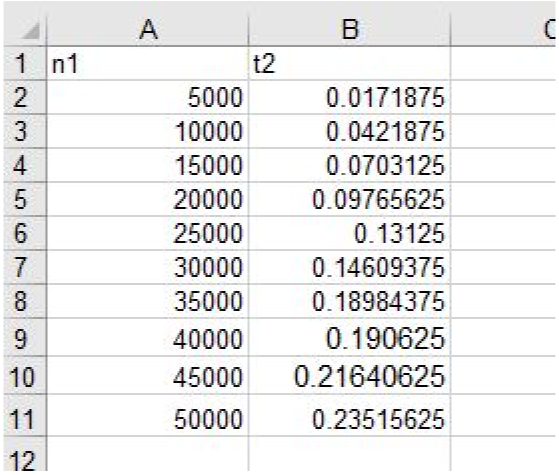
$$T(n) = 3T(\frac{n}{3}) + cn$$

c)

$$O(nlog_3n)$$

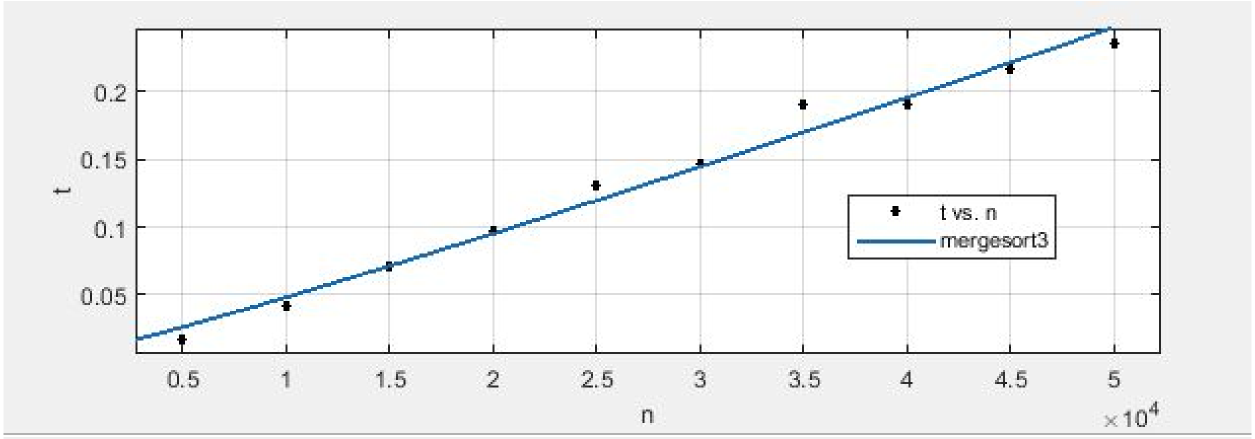
Problem 4:

b)



	A	B	C
1	n1	t2	
2	5000	0.0171875	
3	10000	0.0421875	
4	15000	0.0703125	
5	20000	0.09765625	
6	25000	0.13125	
7	30000	0.14609375	
8	35000	0.18984375	
9	40000	0.190625	
10	45000	0.21640625	
11	50000	0.23515625	
12			

c)



General model:

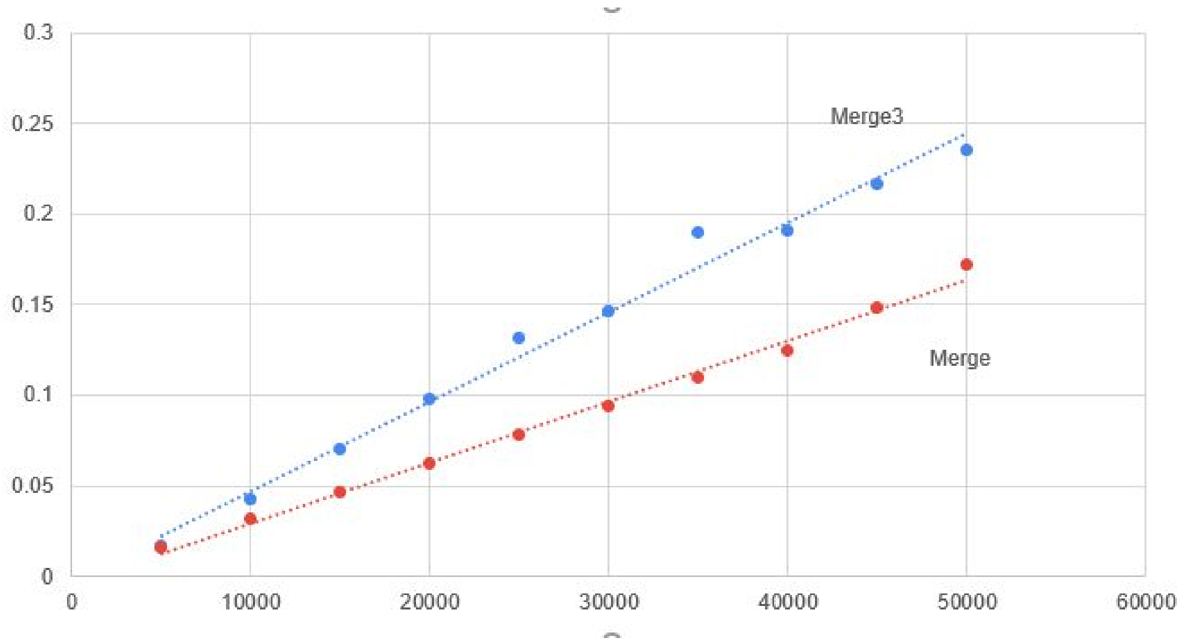
f(x) = a*x*(log(x)/log(3))+b

Coefficients (with 95% confidence bounds):

a = 4.867e-07

b = 0.00729

d)



The results from the runtimes collected show that the three way merge sort runs slower than the normal mergesort. This is different than what was expected based on the runtime complexity, as the three way merge sort has a lower complexity. This discrepancy is probably due to the constant “c” being much larger in the three way merge sort than the normal one, as there are far more constant time instructions in the three way. Even if these don’t match the runtime complexity when comparing them, looking at them individually they do fit with their respected nlogn curves.