	t_{max}/t_{min}	n ratio	n ln(n) ratio	n^2 ratio	Behavior
SC	54579	260	430	67600	Quadratic
SS	56276	260	430	67600	Quadratic
SR	62100	260	430	67600	Quadratic
IC	107	100	129	10000	Linear
IS	128	100	129	10000	n lg(n)
IR	71859	340	573	115600	Quadratic
MC	2595	2000	3158	4000000	n lg(n)
MS	2822	2000	3158	4000000	n lg(n)
MR	6000	4000	6669	16000000	n lg(n)
QC	20721	160	255	25600	Quadratic
QS	3754	3333	5477	11111111	Linear
QR	5951	4000	6669	16000000	n lg(n)

1. How your results compare to the theoretical analysis for the four algorithms

- For **Selection Sort**, they all have a quadratic time complexity which makes sense because the algorithm's best, average, and worst case complexity are all $\Theta(n^2)$.

- For **Insertion Sort**, a constant array takes linear time which is the best-case complexity for this algorithm. A sorted array has an unexpected behavior as n lg(n) is not theoretically correct. A random array has quadratic time complexity which is the average or worst-case complexity.

- For **Merge Sort**, all arrays have a time complexity of n lg(n) which is also the best, average, and worst-case complexity for Merge Sort.

- For **Quick Sort**, a constant array took quadratic time which is the worst-case complexity. Sorted array however, had an unexpected, non-theoretical time complexity. The random array took n lg(n) which is the best or average-case complexity.

2. Why your results make sense or are surprising

- The result for **Selection Sort** algorithm makes sense and meets the expected theoretical value. This is because Selection Sort has a best, average, and worst case complexity of $\Theta(n^2)$ regardless of the array type. This is because the algorithm must go through every element in the array to find the minimum value in the array ($\Theta(n)$) then swap that minimum value to the beginning which is also $\Theta(n)$. This makes the algorithm take $\Theta(n) * \Theta(n) = \Theta(n^2)$ all the time.

- For **Insertion Sort**, constant and random arrays make sense but the result for sorted array was different from the expected theoretical value. Every Insertion Sort algorithm have to go through all of the array and insert every value to the correct position. This takes $\Theta(n)$. The next step depends on the type of the array.

For constant arrays, you don't need to do anything as the array is already sorted. Therefore, the theoretical value for this constant array is $\Theta(n)$ and it matches the experimental value.

For random arrays, you need to shift the sorted elements to make space which will take $\Theta(n)$. This will make the theoretical value $\Theta(n) * \Theta(n) = \Theta(n^2)$ and it matches the experimental value.

The sorted array for Insertion Sort was the unexpected result that was surprising. I expected the value to be $\Theta(n)$ for similar reasons for the constant array. After going through all the elements in the array (which takes $\Theta(n)$), the elements still stay in the same position as before as the elements were already sorted. So, there is no need to shift the sorted elements. The result however was $\Theta(n \lg(n))$ and this is because there were some background tasks running on my PC that interfered with the process. You can see that the t_{max}/t_{min} value is in between n ratio and n ln(n) ratio and also that the n ratio and n ln(n) ratio have very similar values.

- The result for **Merge Sort** makes sense and meets the expected theoretical value. This is because, similar to Insertion Sort, the best, average, and the worst case complexity are all $\Theta(n \lg(n))$ for all types of array. Merge Sort is an algorithm that divides the array in half then sorts each divided array recursively. This takes $2T(n/2)$. After that the algorithm combines the two arrays which takes $\Theta(n)$. Using the master theorem, $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg(n))$. This process is for every array even though the array is constant or is sorted.

- For **Quick Sort**, constant and random arrays make sense but the result for sorted array was different from the expected theoretical value. This is similar to Insertion Sort. For constant array, the time complexity was the worst case and for random array, the time complexity was the average case. This surprised me at first because usually you would think a constant array would take more time than a random, unsorted array. But thinking about the algorithm, the time complexity all depends on the pivot selection. Random pivot selection always results in average case complexity. For the random, unsorted array here, a random pivot was selected making it the average case complexity. For the constant array however, there is no random pivot as the pivot value would be the same no matter which pivot is selected. This means that the pivot was also the worst pivot that could possibly be, and this resulted in the constant array to have the worst-case complexity.

  The unexpected result that surprised me was the sorted arrays. In this case I assume that random pivot selection was used which will usually result in $\Theta(n \lg(n))$. So, the expected theoretical complexity was $(n \lg(n))$ while the result I got from the experiment was (n). This is because the pivot that the algorithm selected happened to be a very "good" pivot. If a "good" pivot was luckily selected, the complexity would be lower compared to the normal average case. As you can see from the above graph, t_{max}/t_{min} value is in between n ratio and n ln(n) ratio and this time, because of the luckily selected "good" pivot, the time complexity went a bit down making the complexity linear.