To make our analysis of the two different sorting algorithms fair, it was important to fill the arrays with the same numbers. Of course, you needed to have two different arrays because you would not yield proper results if you had one sorting algorithm sort an array and then had another sorting algorithm try and sort that same already sorted array. After ensuring fair trials, we can now properly evaluate the results returned. As seen in the table below, quickSort blew mergeSort out of the water in terms of speed in almost every trial.

|  |  |  |  |
| --- | --- | --- | --- |
| n | nTrials | # mergeSort Wins | # quicksortWins |
| 10 | 20 | 0 | 20 |
| 100 | 20 | 8 | 12 |
| 1000 | 20 | 1 | 19 |
| 10000 | 20 | 0 | 20 |
| 100000 | 20 | 1 | 19 |
| 1000000 | 20 | 0 | 20 |
| 2000000 | 20 | 0 | 20 |

From these results, we can determine that quicksort is indeed the faster sorting algorithm. quickSort was faster in every trial run and sometimes even beat mergeSort out completely in every trial ran. I do not believe that our answer here is dependent on the size of the array due to the fact that we ran each sorting algorithm on arrays of drastically different sizes (ranging from 10 to 2,000,000) multiple different times.

Now that we have determined that quickSort is faster, it is necessary to see exactly by how much. We required the program to begin tracking the time that it took to run each of the sorts and then compare each trial individually. This is basically what we did in experiment one, except now we store the times somewhere to calculate the average run time for both quickSort and mergeSort.

Comparing the averages can assist in giving us a general idea as to which sort is faster. By using the average times and the definition of big Oh, we can potentially prove that each sort has the time complexity that is required or assumed of it. Big Oh is defined as f(n) is O(g(n)) if there is some n0 >= 1 and c > 0 such that f(n) <= cg(n) for all n >= n0.

Using the instructions provided, I was able to produce the following table of results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | mergeSort:  mean runtime (nanosecs) | mergeSort:  mean runtime/ (n\*log2(n) ) | quickSort:  mean runtime (nanosecs) | quickSort:  mean runtime / (n\*log2(n) ) |
| 10 | 14855 | 447.18005855884405 | 5125 | 154.27787277779035 |
| 100 | 56874 | 85.60389986696633 | 41141 | 61.923375258059245 |
| 1000 | 869750 | 87.27361290958254 | 135185 | 13.564913321278432 |
| 10000 | 3818853 | 28.739732550784538 | 1436361 | 10.809693640047792 |
| 100000 | 22646185 | 13.634361944711433 | 14499630 | 8.729647112058663 |
| 1000000 | 198762973 | 9.972269483391669 | 133064058 | 6.676045467128624 |
| 2000000 | 410177010 | 9.79804759116635 | 275834721 | 6.588964417225852 |

These results confirm that mergeSort and quickSort are O(n\*log(n)) because as the number of elements in the array increases the average sort time divided by n\*log(n) decreases. As seen by my graphs below, our values begin trending towards n\*log(n) and straying further and further away from n2.

This observation can constitute that both mergeSort and quickSort trend towards n\*log(n) and are therefore O(n\*log(n)).

I personally believe quickSort is faster because of how it works. mergeSort continues to break down an array into multiple different arrays and sorts the individual arrays and then merges the sorted arrays together in a sorted order. quickSort keeps the same array and partitions the array. Partitioning is not creating new arrays, but just creating little subarrays within the existing array. quickSort begins by picking a pivot and then swapping numbers around the pivot making sure that every number on the left side is smaller than the pivot and every number on the right is larger than the pivot and considering the pivot itself as “sorted”. It continues this process until the array is sorted.