



From looking at the graphs we can clearly see that the O(nlogn) sorting algorithms (Quick, Shell, Merge, and IterativeMerge sort) have a significantly faster runtime when sorting large data sets (greater than 1,000) when compared to the O(n2) sorting algorithms (Bubble and Insertion sort). We can see from the graphs the O(nlogn) sorts are able to sort lists of 100,000 integers faster than the O(n2) are able to sort lists of 5,000 integers. The increased efficiency is due to the divide and conquer approach.

When comparing Bubble Sort and Insertion Sort, which both having the same complexity, we see that Insertion Sort is significantly faster on data sets that have more than 1,000 integers. This is because in bubble sort swaps occur through all remaining unsorted values, moving from one to the bottom of the list. In insertion sort the value being sorted is placed directly in the sorted location without having to compare to every other value in the list. This leads less comparisons.

Comparing the O(nlogn) sorting algorithms, we see that all the sorts perform about the same time up to 25,000 integers. Once we the data sets become larger, we can see that QuickSort becomes the fastest, followed by IterativeMergeSort, ShellSort and then MergeSort. MergeSort is the slowest because each time we merge the vectors, we must allocate new memory in the form of a temporary array. The merge sort is recursively called and each time that happens the data needs to be copied to a new temporary array, taking up more time and space. This is improved upon in the IterativeMergeSort because rather than creating a temporary array for each recursive call, it only uses one temporary vector that then copies the data back and forth from the original vector until the list is sorted. It requires significantly less memory allocation and thus runs faster overall. ShellSort is quicker than MergeSort because it doesn’t require the extra memory allocation. Quick Sort is the fastest in this case because it is in-place (doesn’t require new memory allocation) and requires the least amount of comparisons. QuickSort does have a worst-case complexity of O(n2), but because of the way we chose the pivot it avoids this case in most scenarios.