Sorting Analysis

The results of my program already confirmed what I already knew about the following sorting algorithms. First, I knew that sorts like bubble sort, selection sort, or insertion sort are not as efficient as other sorts because they can only go so fast. Since they compare adjacent values consistently it takes a lot of time for them to sort lists with a large number of elements. As we can see from the output, with each digit added to our number of inputs our number of comparisons would increase in one or two digits as well. Due to the large amount of comparisons the program even excluded them for larger input above 5000.The next thing it confirmed for me was that the recursive sorts are extremely quick. For example, for inputs of 5000 elements it takes on average a little bit over than half a second to sort with our three slower sorts. Meanwhile, it takes on average a thousandth or a hundredth of a second to sort lists with the same amount of elements with recursive sorts. The only downside to recursive sorts is the number of resources it uses as it requires memory to allocate space of the same amount as the number of inputs. According to the outputs, the input was 500 then it required 499 mallocs. However, one thing that surprised me is the possibility of mixing two sorts like the merge-insertion recursive function which decreased the number of memory allocations. For reference, for 500 inputs it only requires 31 mallocs.

There are a few differences I noticed among the slow and fast sorting algorithms. First, there is a large difference in the number of comparison between the two types. The fast algorithms only have half sometimes less than half of the number of comparisons to make compared to the slow algorithms. Due to the fast algorithms' divide and conquer nature the comparisons are drastically decreased. I also noticed when comparing to the insertion sort, fast algorithms have drastically less copies as well. For the fast algorithms, the number of copies is also almost half of the slow algorithms or in other words the insertion sort. I also noticed that moving elements around are different for both types of algorithms. For slow algorithms, it is heavily dependent on swaps and shifting elements left or right within the existing list. This can be inefficient as we're constantly moving numbers we might not need to move. Meanwhile for fast algorithms, it is heavily dependent on the index of the list and its ability to divide and merge to put elements in order. Swapping isn't really utilized in fast algorithms, rather they are placed after being analyzed and divided. Lastly, I also noticed the difference in CPU times used. For the faster algorithms for input of 500 it would take 10,000th of a second to finish ,while slower algorithms would take a 1000th of a second. This difference is even larger with greater numbers of inputs. Overall there are many differences between the slower and faster algorithms of sorts.