Writeup

Compile

To run the program, unzip PA1.zip to a folder and open that folder from the command prompt/terminal. Then, run the command python main.py *or* python3 main.py

Estimate and Graph Runtime

To generate the below table, we created three different variables to track the runtime operation of each of the three sorting algorithms: insertion sort, merge sort, and quicksort. These variables were initialized in a global scope to the value zero and incremented for each case where an element from the array to be sorted 'array[index]' is compared.

Lists	Quicksort Runtime	Merge Sort Runtime	Insertion Sort Runtime
pokemonSortedSmall	13695	1238	165
pokemonSortedMedium	93096	3808	431
pokemonSortedLarge	319600	7776	799
pokemonReverseSortedSmall	9523	1238	13672
pokemonReverseSortedMedium	49429	3808	92330
pokemonReverseSortedLarge	157546	7776	316730
pokemonRandomSmall	1203	1238	6934
pokemonRandomMedium	4226	3808	46589
pokemonRandomLarge	12164	7776	163108

The runtime complexity of insertion sort is O(n) and O(n^2) in the best case and worst case respectively. We know that the best case for Insertion Sort is when an array is mostly sorted and from the above table, we can see that this holds true as the runtime we evaluated is very fast (see the first three rows for sorted values). On the other hand, for lists that contain unsorted elements, we can see that insertion sort needs to make a lot of comparisons as the number of elements in the list increases.

The runtime complexity of merge sort is O(n(log(n))) for best, average, as well as worst cases. This mean regardless of the ordering of the elements; merge sort will always have a constant runtime for a listen of given length. The table demonstrates that this is true as Merge Sort has the same runtime of 1238, 3808, and 7776 for the small, medium, and large lists respectively.

The runtime complexity of quicksort is O(n(log(n)) and O(n^2) in the best case and worst case respectively. Quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets. Analyzing the values in the table, this is true as for the small list of randomly ordered values, as quicksort makes only 1203 comparisons whereas merge sort makes 1238 comparisons. The worst case for quicksort occurs when the array to be sorted has values that are already somewhat sorted. And we can see that this holds as in the above table, quicksort must make a huge number of comparisons for sorted lists that are small, medium, and large.
Thus, from these deductions, we can conclude that the behaviors above are normal.

Some important details of my implementation:

1. The only external library used is “math” in order to use the floor() method required for the merge sort algorithm.
2. The index of the final element of a given array is passed to both merge_sort() and quick_sort() functions and not the entire length of the array.
3. To make sure that the value of runtime operation trackers are not lost within different scopes of recurrence calls in merge_sort() and quick_sort(), we use the global keyword within the sub-function scopes.
4. A copy of the original lists are made using the .copy() function instead of directly using the assignment operator (to achieve a deep copy instead of a shallow copy) such that consequent sorting algorithms don’t sort arrays that have already been pre-sorted.
5. Other than the sorting functions, some functions to help with printing have also been defined to reduce redundant code and make the printing process more convenient.