*Morgan Hardin*

*Dr. Johnson | CS-330 Algorithms*

*Algorithm Unique Integers and Sorting report*

This report shows the theoretical and empirical analysis of code that produces n unique integers and sorts them using merge sort and again using insertion sort.

The original algorithm takes an inputted empty array and integer and produces randomly generated numbers and puts them in the array. It checks to see if a number is duplicated, and if so, it generates another number into that index. This ensures that each element in the array is unique and not duplicated. To create the random numbers, the Random function is used to generate an integer between the inputted integer n and n3. It then loops through every element in the array and checks if the randomly generated integer is already in the array. If it isn’t, it will be put in the next null index. If it is, then it is labeled as a duplicate and the loop starts over with a new generate number.

For this original algorithm, there are two nested while loops, two if statements, and 1 basic operation to compare indexes. This means that the formula would be:

This shows that for the original algorithm of filling an array of n unique integers is O(n2). This matches up with the theoretical efficiency of O(n2) due to the two nested while loops. So, the theoretical and empirical efficiencies are O(n2).

The first sort algorithm is the merge sort algorithm. This algorithm separates the array into a left side and a right side and continuously separates them until the entire array has been separated into subarrays. They are then sorted in the subarrays and the subarrays and merged together and sorted until finally it is merged back into one array and the elements are all sorted.

For this algorithm, there are two for loops and three while loop. None of the loops are nested, so they each cycle through when the conditions are met, meaning there is a possibility that some loops will not be executed. The Master Theorem is used in order to find the efficiency for this algorithm through the following:

T(n) = aT(n/b) + O(nd) if a > 0, b > 1, and d >= 0

If a = 2, b = 2, and d = 1 because the code splits the array into halves and each is O(n)

T(n) = 2T(n/2) + O(n)

T(n) = O(nd \* log n)

T(n) = O(n log n)

For merge sort, the best, worst, and average efficiency is O(n log n). This is because no matter what, the algorithm has to loop through the array but also divides it in half each time to sort. This algorithm is extremely efficient with the average runtime being O(n log n). No CPU or wall time was collected. As n grows larger, it takes longer for the algorithm to execute so CPU and wall time are obvious to the user. Both the theoretical and empirical efficiencies are O(n log n) for the merge sort algorithm.

The second sort algorithm is the insertion sort algorithm. This algorithm compares two numbers in the array and swaps them if the second element is smaller than the first. It then continues with the second and third elements, all the way to the n-2 and n-1 elements. This is the first pass through the array, and the algorithm will continue to repeat these steps until there is a pass through the array where no elements need to be swapped.

For this algorithm, there is 1 for loop and 1 while loop as well as basic comparison operations to swap and sort the elements. The best case efficiency would be O(n) and that would mean that the array is already sorted. This means that the formula would be:

The worst case would be O(n2) because every element in the array would need to be compared and swapped, meaning both loops will be iterated through the n integers n times, or n \* n. The greatest exponent is n^2 meaning its efficiency is O(n2). This means that the formula would be:

Finally, the average case would be O(n2) because although some elements might not need to be swapped, both loops are being iterated through n times, so n \* n. The formula would be the same as the worst case:

Insertion sort has some differences in the theoretical and empirical efficiencies. By looking at the code, it looks as if the efficiency is O(n2). While this is the case for the average and worst cases, the best case is actually O(n) because if every element in the array is already in order, then the second loop will not be iterated through because no swapping needs to be done. Therefore, the best case has different theoretical and empirical efficiencies of O(n) and O(n2).

Overall, merge sort is a more efficient sorting algorithm than insertion sort. While in the best case insertion sort is O(n), it is very rarely that an array will already be sorted. Therefore, it is not likely that the best case will happen frequently. This means that merge sort is much more efficient since O(n log n) is quicker and more efficient than insertion sort’s efficiency of O(n2). This is because as n gets larger and larger, it takes more CPU and wall time for insertion sort to execute than merge sort. Merge sort is a more efficient approach to solving a large problem as n grows larger and larger.