Group 7

CSCI 313

HW1-- Question 5: Multi-Level Sort

In our main function, we generated an integer array of a random size, containing between one and twenty elements. We then created a for loop to iterate through the array and assign a random value between one and fifty to each of the elements. The array serves as our unsorted, random number array. This means that our program could randomly generate an array of size five, as five is an integer between one and twenty, that looks like this: [12, 49, 18, 6, 32]. The elements in the array are in no specific order, and the values are integers between one and fifty.

Next, we wanted to sort the array using two different sorting methods in order to create the multi-level sort. We achieved this by calling the QuickSort function from the main method. For arrays with a size less than ten, we used the InsertionSort function to sort the array; otherwise QuickSort would complete the task. We chose to use QuickSort for the larger arrays because we could use the pivot element and partitioning technique in order to rapidly complete the sorting task. We also found InsertionSort, although not as fast, was more efficient than QuickSort and would be better to use for the smaller sized arrays.

We began the QuickSort function with an if statement that checks the size of the array by subtracting the first position of the array from the last. If the difference was less than ten, we called the InsertionSort function, and if it was greater than or equal to ten, then we would continue using QuickSort. Assuming the value is less than ten, let us begin using InsertionSort. In InsertionSort, we used a for loop to iterate all the array's elements. The for loop stores the current element in an integer variable in order to create a reference value. This allows us to compare each of the elements in the array so that we can effectively sort them. We also created a

second integer variable to shift to the right position and effectively sort the array. Next, we created a for loop to iterate through the sorted elements and insert the reference value in the correct position. In the for loop, we set a condition that checks to see if the value of the current index is greater than the reference value. If it is true, then we move the current value to the right of the reference value, ensuring that our array has values in ascending order. If the condition is false, we move the reference value to the current index. We exit the loop when all the elements are correctly sorted.

Now if we assume that the array size is greater than or equal to ten, we will move forward with the QuickSort function to sort the array. QuickSort will only perform the sorting process if the difference of endIndex and startIndex is not less than 10. Proceeding with our QuickSort function, we have an if statement condition that checks whether the end index is higher than the starting index, only when this statement is true the sorting may be performed. We then create an integer variable that retrieves the pivot position through the Partition function. This new variable will already be in the correct position, allowing us two split the array into the two sub-arrays to be sorted. The Partition function works by making the first element of the array the pivot as well as a second integer variable called the "middleIndex." At this point, the left and right sub-lists are empty. Our next goal is to make sure that the middleIndex, which should be a part of the right sub-list, accurately divides the two lists. To achieve this, we created a for loop that iterates through all elements of the array, excluding the first and last. Within the loop, we set a condition that checks if the value of the current element is less than the pivot. If the condition is met, we shift the middleIndex and swap it with the current element. This process continues until we finish the array and exit the for loop. Then we can swap the starting index of the array with the middle index and return the middle index to QuickSort. Quicksort then does recursive calls

on itself two more times to sort through both the left and right sub-lists of the array. When each of those portions are complete, the array has been sorted. Finally, the main function prints the sorted array.