This program uses 3 functions in order to perform the quick sort algorithm, a main function, a partition function which does the sorting, and a quickSort helper function which makes function calls and splits the vector up. This program uses a vector instead of dynamically allocated arrays. I primarily chose a vector so we would not have to declare any set sizes, and vectors are also extremely versatile. Finally, this program has the option of reading through a file for numbers, or getting numbers through iostream's cin from the user.

The function starts in main, where the user is asked if they would like to enter a filename or test numbers entering them from the terminal. If neither option is selected, the program ends. If the user chooses 0, or to enter a filename, the user is asked to input the filename. The file is then opened using fstream, and all contents of the file are pushed into the vector. If the file can't be found, then the program outputs "Unable to open file". We then close the file and call the mergeSort function. If the user inputs 1, then they are asked to enter how many numbers they would like to input. The user can then enter the numbers they would like to be sorted. If the user continues to enter more numbers than what they put for the size, they aren't pushed to the vector. Then, we call quickSort.

We call the quickSort function with the following arguments: the numArray vector, 0 (the starting index of the array to be sorted) and the size of numArray - 1 (the ending index). We name the last two arguments low and high respectively in the rest of the program. The purpose of this function is to recursively split up numArray. This algorithm works very similarly to mergeSort, however one big change is that quickSort is an unstable algorithm. Meaning that two elements of the same value may be switched during sorting. Before sorting the vector, the function checks if the sub-array contains more than one element. If low is less than high, the sub-array has more than one element, and sorting is required. Otherwise if low is greater than or equal to high, the subarray contains either zero or one element, and no sorting is necessary. It starts by calling the partition function to rearrange the elements in the sub-array. The partition function selects a pivot element and places all elements less than or equal to the pivot on the left side, and all elements greater than the pivot on the right side. The partition function then returns the index of the pivot element after the partitioning process. I will dive into the partition function in the next paragraph. After the partitioning is done, the quickSort function recursively calls itself on the two subarrays created by the partition. It calls itself for the entire left side first, then the right side. This continues until we have finished all calls.

The partition function works as I said earlier, by taking in the arguments: the numArray vector, and the current lowest and highest indexes for the current sub-array. We first

declare a couple variables: the pivot element, which is used to make sure we don’t swap a higher number with a lower number, and the currentIndex which keeps track of the sub-array index in the function. Then we loop over the current sub-array size, and check if the current element is less than or equal to the pivot. If it is, then we increment the currentIndex and swap the element at the currentIndex with the one of the loop (j). We use the built in standard library swap call to swap the numArray element at the currentIndex with the element at the loop iteration (j) and vice versa. Finally, we place the pivot element in its correct sorted position using a final swap. After we have finished sorting, we return to main and print out the sorted vector.