Best case:

* In my code, the numbers in the array can be as high or low as one can call. The array is iterated once looks for duplicates, negatives and it also does not corrupt if the array is not to long since its meant to handle any size. Another best case scenario would be if the array is already sorted so the time complexity would be almost instantly.

Worst case/Average case

* One malfunction that can occur is that if the array within the code was 2D array instead of a 1D array. It wouldn’t understand how it can be iterated since it meant to be for a single array. Another scenario which I did not test was to check if decimal could be taken into the array. Another case that involved both worst and average cases is that no matter how long the array is it would take the same time since it has the same time complexity for the code to sorted.

SortOfSort

The focus of the code is that is capable to rearrange numbers in a 1D array to put them in a specific order. This method can take 1D array of integers that can contain negative numbers, duplicates numbers, and a single number. This method utilizes for loops to iterate the array and if statements to locate a number and set it in it corresponding position within the array. One variable that I utilize was named “skip” this variable was used in the if statement to skip the numbers in the array so it can organize the array to it can be in the specific order. For the tester, I utilize a JUnit to test a variety of way for the 1D array to function. This is where I came up with four way to see if the method would work with different numbers or ways it was given to it. I decided to go for duplicate numbers, negative numbers, and a long array of number to see the extent of the array. The time complexity for my code would be 8+20n