Method Performance

The sortOfSort method I have in place starts off by checking the size of the array, returning if it is less than 2 elements. It continues by entering a main for loop that will iterate through the entire size of whatever array is given, it sets an index value to keep track of what part of the array is being filled and a max value that is declared and changed after every iteration, i.e. every max element is sorted, this part of the method goes through linear time and has a O(n) time complexity. Then the method goes into another loop that will find the highest value in the array before anything is sorted, this part of the method is also linear and has a O(n) time complexity. The method is met by a conditional that compares the indexes of what’s been sorted on the side of the start of the array and on the side of the end of the array, this is done to avoid one unnecessary sorting step and it will also make sure not to pass the iteration out of bounds.

If this condition is met, there is another conditional that is dependent on a count, since we are sorting two of the highest elements on each end before switching sides in this algorithm, the algorithm starts off at the !front side and this is where the swap of the highest element is made at the end, this part of the array is now sorted and the variable storing the end index of the array decrements to leave it alone furthermore. After the two iterations of this part of the array, the front Boolean is negated and so it will enter the remaining conditional to sort the highest remaining element in descending order from the start of the array, again, for only two iterations as per the instructions of the method. The algorithm stops when the start variable incrementing from the front of the array is equal to the end variable incrementing from the end of the array, therefore it will not sort the last element since it should already be sorted. The explanation of how the method algorithm works gives us the runtime of the algorithm by terms of n (where n is the number of elements in the array) is O(n^2), except when there is an empty array or the array is of size 1, which requires no steps of the algorithm it only checks the length and returns the same array.

The best-case scenario of this method is that the array entered would have 0 or 1 elements and whatever is entered would be returned immediately. The average and worst-case would be considered the same in my opinion. This is because no matter the size or order of whatever array that is given, to find the max element of the array, the inner for loop will iterate through the entire element and the general runtime of the algorithm will always be n^2 excluding when n is 0. Nonetheless, the best case, average or worst-case scenarios are completely dependent on n. This would be considered a quadratic algorithm where O(n^2).

Figure 1 below demonstrates the parts of the algorithm that define its runtime complexity.

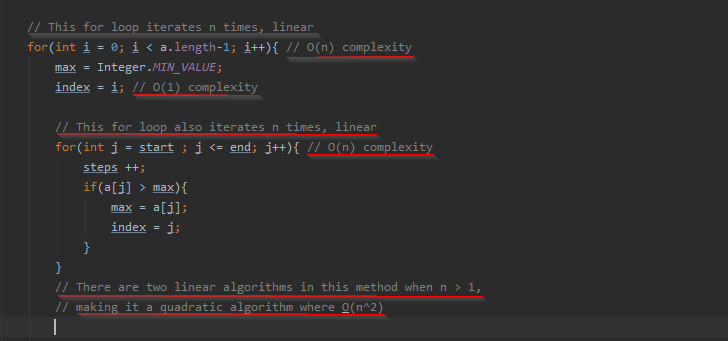


Figure 1