SortOfSort method Analysis

|  |  |
| --- | --- |
| **ONCE** | **REPEATING** |
| int count = 0; | first < last |
| int first = 0; | maxNum = array[first]; |
| int last = array.length – 1; | i <= last |
| int maxNum; | i++; |
| int maxFoundAt = 0; | array[i] >= array[maxFoundAt] |
| int temp = 0; | MaxNum = array[i]; |
| int i = first; | maxFoundAt = i; |
|  | count == 4 |
|  | count = 0; |
|  | count < 2 |
|  | temp = array[last]; |
|  | array[last] = maxNum; |
|  | array[maxFoundAt] = temp; |
|  | last--; |
|  | count++; |
|  | temp = array[first]; |
|  | array[first] = maxNum; |
|  | array[maxFoundAt] = temp; |
|  | first++; |

**Categorizing the instructions of the algorithm**

**NOTE:** All print lines and the printArray method calls were omitted from the computation of the algorithm’s time complexity as they do not affect the overall outcome of the final calculation. Print statements and print Array method calls were utilized strictly for visualization and debugging purposes.

**Time Complexity:** After analyzing and categorizing all the individual steps of the algorithm by the ones that happen ONCE and the ones that REPEAT, it is easier to determine what the time complexity could ultimately be. After categorization, we can conclude that only 7 of the instructions happen once thus leading to a total of 7. Expressed by the code, we can also conclude that the while loop will compute 13 instructions for some n times leading to 13n. Similarly, the for loop executes a total of 5 instructions for some n times resulting in 5n. To conclude, T(n) = 7 + 5n + 13n which can be reduced to T(n) = 7 + 18n expressed the time complexity of the algorithm.

**Big O(n):** With the calculation of the time complexity, we can take T(n) = 7 + 18n and from there exclude the constants. This then results in Big O(n) equaling O(n) for some n steps being the overall size of the array.