In order to analyze the performance of the method sortOfSort, I have compiled the number of times that each statement in the program contributes to the time complexity. This compilation was made with the notion that an array of length n was passed to the method. The leftmost column assigns a number to each statement in order to facilitate discussion. The middle column contains the statement itself and the rightmost column contains the number of times that specific statement contributes to the overall time complexity. Further discussion of this table will be found below.

|  |  |  |
| --- | --- | --- |
| # | Statement | Number of Contributions |
|  | sortOfSort(int[] integerArray) | |
| 1 | int maxIndex; | 1 |
| 2 | int endIndex = integerArray.length – 1; | 1 |
| 3 | int beginningIndex = 0; | 1 |
| 4 | int counter = 1; | 1 |
| 5 | int arrayEndLimit = 0; | 1 |
| 6 | for(int i = 0; i < integerArray; i++) | n |
| 7 | maxIndex = beginningIndex; | n – 1 |
| 8 | for(int i = beginningIndex; i < integerArray.length – arrayEndLimit; i++) | n(n+1)/2 |
| 9 | if(integerArray[i] > integerArray[maxIndex]) | n(n+1)/2 |
| 10 | maxIndex = I; | 0 to n(n+1)/2 |
| 11 | if(counter == 1) | n – 1 |
| 12 | swap(integerArray, maxIndex, endIndex); | 3(n – 1) |
| 13 | endIndex--; | n – 1 |
| 14 | counter++; | n – 1 |
| 15 | arrayEndLimit++; | n – 1 |
| 16 | else if(counter == 2) | n – 1 |
| 17 | swap(integerArray, maxIndex, endIndex); | 3(n – 1) |
| 18 | endIndex--; | n – 1 |
| 19 | counter = -1; | n – 1 |
| 20 | arrayEndLimit++; | n – 1 |
| 21 | else if(counter == -1) | n – 1 |
| 22 | swap(integerArray, maxIndex, beginningIndex); | 3(n – 1) |
| 23 | beginningIndex++ | n – 1 |
| 24 | counter--; | n – 1 |
| 25 | else | n – 1 |
| 26 | swap(integerArray, maxIndex, beginningIndex); | 3(n – 1) |
| 27 | beginningIndex++; | n – 1 |
| 28 | counter = 1; | n – 1 |
|  | swap(int[] integerArray, int largest, int toSwap) | |
| 29 | int temp = integerArray[largest]; | 1 |
| 30 | integerArray[largest] = integerArray[toSwap]; | 1 |
| 31 | integerArray[toSwap] = temp; | 1 |

**Reasoning for the Number of Contributions in each Statement**

For lines 1-5, each statement is only executed once, which explains the value of 1 in the number of times each statement contributes to the time complexity. For line 6, this for-loop will contribute n times since that is the length of the array. For line 7, this value is within the previous for-loop, so it will contribute n – 1 times. For line 8, the number of times it contributes is a summation because in the first loop, it will iterate n times, and in the next loop it will iterate n – 1 times, and so on. Therefore, the total steps for this statement is of n(n+1)/2. Line 9 will also iterate this same amount of times because it is within the for-loop and it only contributes once each iteration, so 1 times the summation is equal to the same summation. Line 10 can either contribute between 0 times to the previous summation amount of times. It will contribute 0 times if the max index was at the beginning index, so the if-statement is always false. On the other hand, the if-statement can always be true, so it will iterate n(n+1)/2 times. The lines 11, 16, 21, and 25 contain conditionals, which means that either they will all contribute once, or just one will contribute. If just one contributes, then the number of contributions will be (n – 1) because it is within a for-loop. If they all contribute, then (n – 1) will be added four times. Lines 13, 14, 15, 18, 19, 20, 23, 24, 27, and 28 all contribute once to the time complexity but since they are within a for-loop, they will each contribute (n – 1) times. Lines 29, 30, and 31 each contribute once to the time complexity since they are in a separate method. Lines 12, 17, 22, and 26 will each execute (n – 1) times because they are within the for-loop; however, since they each make a call to a method that contributes 3 times to the time complexity, their contribution will be 3(n – 1).

**Overall Time Complexity Equation**

The overall time complexity formula is created through the addition of each statement’s contributions.

T(n) = (3n2 + 21n – 10) / 2

This formula was taken given the worst-case scenarios where all of the if-statements contribute to the time complexity, with the exception of the content within the first three if-statements, and a different max value is found through each iteration.

**Time Complexity in Big O Notation**

The time complexity written in Big O notation would equal to O(n2) because that was the largest exponent value and the rest of the values are disregarded. This means that for an n length array, the method will go through n2 steps in order to sort it. This is because for each value at an index being evaluated, it will be compared n times for the n length of the array. Earlier, it was stated that each time the value will be compared one time less, which equaled to the final summation of n(n + 1) / 2, which when multiplied through, is where the n2 originates.

**Different Cases for the Performance**

For this method, the worst and average case for the performance are both O(n2). The worst case for this method would be to be given an array of integers that are directly opposite of how the method is sorting them, so if the first largest are the beginning and the next largest are at the end. The formula was computed given the worst cases, which led to additional values being added in order to account to additional contributions that would have otherwise been zero or one. This performance is also the average case because it was generated with a general length and not a specific case. The best-case performance is on an array that is already sorted the way that the method is sorting arrays. Although this is the best case, each value will still be checked a total of n(n+1)/2 times, which ultimately leads to a time complexity of O(n2). Therefore, regardless of the case, the time complexity will always be O(n2).