# Performance: sortOfSort Method

While performing my theoretical analysis, I created three tables to count the number of static and repeating instructions for each of my for-loops.

***First for-loop:***

|  |  |
| --- | --- |
| once | repeating |
| int i = 0  i < repeat | i < repeat  int tempHoldR =  **for(right side)** …  if()  break  int tempHoldL =  **for(left side)** …  if()  break  i++ |
| 2 instructions + | (8 + (56+36n))\**repeat* |

**= 2 + (64 + 36n)\**repeat***

***Nested for-loop: right side***

|  |  |
| --- | --- |
| once | repeating |
| int r = 0  r < 2 | r < 2  if()  break  max = ***findMax()***  maxInd = ***findIndexOfMax()***  arr[] =  if()  arr[] =  tempRight - -  if()  if()  tempHoldR =  r++ |
| 2 instructions + | (11 + (4n) + ( 2+5n))\*2 instructions |

**= 28 + 18n instructions**

***Nested for-loop: left side***

|  |  |
| --- | --- |
| once | repeating |
| int l = 0  l < 2 | l < 2  if()  break  max = ***findMax()***  maxInd = ***findIndexOfMax()***  arr[] =  if()  arr[] =  tempLeft - -  if()  if()  tempHoldL =  l++ |
| 2 instructions + | (11 + (4n) + ( 2+5n))\*2 instructions |

**= 28 + 18n instructions**

If **n** is defined as the length of the array, the value ***repeat*** is defined as **n**/4 or **n**/4 + 1 depending on whether the length is even or odd, respectively. In the case of **n** being an even number, the time complexity of sortOfSort is: **T(n) = 2 + (64 + 36n)\*(n/4).** Because of this, the bigO notation is: **O(n) = n^2**

# Performance: findMax and findIndexOfMax Methods

During my analysis, I also had to account for the helper methods that I used in my sortOfSort method.

***findMax Method:***

|  |  |
| --- | --- |
| once | repeating |
| int max = -1000  int i = left  i <= right  return max | i <= right  if()  max =  i++ |
| 4 instructions + | 4\*(right +1 – left) |

**= 4n**

***findIndexOfMax Method:***

|  |  |
| --- | --- |
| once | repeating |
| int max = -1000  int index = 0  int i = left  i <= right  if()  index =  return max | i <= right  if()  max =  index =  i++ |
| 7 instructions + | 5\*(right + 1 – left) |

**= 2+ 5n**

\*\*The above functions were found algebraically based off of the following logic.

In each of these cases, the time complexity is dependent on the values of ***right*** and ***left***. These values represent the indices of the left and right positions being looked at, which will always be dependent on **n**:

0 <= ***left*** <= (**n**/2 – 1) and, (**n**/2 – 1) <= ***right*** < **n**. So, the first time they are called, the for loops will iterate **(n-1) + 1 – (n-n) = n** times. On average, it will iterate **n – 1** times with every call. (right + 1 – left) = n - 1

# Best Case/Worst Case/Average Case

The best-case scenario is when **n** is 1, the worst case scenario is when **n** is very large, and the average case scenario might be when the array already has sorted elements in there corresponding position.