**Complexities**

***Bubble Sort***

Bubble Sort compares the adjacent elements.

|  |  |
| --- | --- |
| Cycle | Number of Comparisons |
| 1st | (n-1) |
| 2nd | (n-2) |
| 3rd | (n-3) |
| ....... | ...... |
| last | 1 |

Hence, the number of comparisons is

(n-1) + (n-2) + (n-3) +.....+ 1 = n(n-1)/2

nearly equals to n2

Hence, Complexity: O(n2)

Also, if we observe the code, bubble sort requires two loops. Hence, the complexity is

**n\*n = n2**

**1. Time Complexities**

***Worst Case Complexity: O(n2)***

If we want to sort in ascending order and the array is in descending order then the worst case occurs.

***Best Case Complexity: O(n)***

If the array is already sorted, then there is no need for sorting.

***Average Case Complexity: O(n2)***

It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

**2. Space Complexity**

Space complexity is O(1) because an extra variable is used for swapping.

***Selection Sort***

|  |  |
| --- | --- |
| Cycle | Number of Comparisons |
| 1st | (n-1) |
| 2nd | (n-2) |
| 3rd | (n-3) |
| ....... | ...... |
| last | 1 |

Number of comparisons: (n - 1) + (n - 2) + (n - 3) + ..... + 1 = n(n - 1) / 2 nearly equals to n2.

Hence, Complexity = O(n2)

Also, we can analyze the complexity by simply observing the number of loops. There are 2 loops so the complexity is **n\*n = n2.**

1. **Time Complexities:**

**Worst Case Complexity: O(n2)**

If we want to sort in ascending order and the array is in descending order then, the worst case occurs.

**Best Case Complexity: O(n2)**

It occurs when the array is already sorted

**Average Case Complexity: O(n2)**

It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

**2. Space Complexity**

Space complexity is O(1) because an extra variable is used for swapping.

***Insertion Sort***

1. **Time Complexities**

**Worst Case Complexity: O(n2)**

Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.

Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.

Thus, the total number of comparisons = n\*(n-1) ~ n2

**Best Case Complexity: O(n)**

When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.

**Average Case Complexity: O(n2)**

It occurs when the elements of an array are in jumbled order (neither ascending nor descending).

Space Complexity

1. **Space complexity**

Space Complexity is O(1) because an extra variable key is used.

***Quick Sort***

**1. Time Complexities**

**Worst Case Complexity [Big-O]: O(n2)**

It occurs when the pivot element picked is either the greatest or the smallest element.

This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains n - 1 elements. Thus, quicksort is called only on this sub-array.

However, the quicksort algorithm has better performance for scattered pivots.

**Best Case Complexity [Big-omega]: O(n\*log n)**

It occurs when the pivot element is always the middle element or near to the middle element.

**Average Case Complexity [Big-theta]: O(n\*log n)**

It occurs when the above conditions do not occur.

**2. Space Complexity**

The space complexity for quicksort is O(log n).

***Merge Sort***

1. **Time Complexity**

**Best Case Complexity**:

O(n\*log n)

**Worst Case Complexity**:

O(n\*log n)

**Average Case Complexity**:

O(n\*log n)

1. **Space Complexity**

The space complexity of merge sort is **O(n).**

***Heap Sort***

1. **Time Complexity**

**Best Case Complexity**:

O(n\*log n)

**Worst Case Complexity**:

O(n\*log n)

**Average Case Complexity**:

O(n\*log n)

1. **Space Complexity**

The space complexity of merge sort is **O(1).**