| **Sorting Algorithm** | **Best-Case Time** | **Worst-Case Time** | **Reasoning** |
| --- | --- | --- | --- |
| **Bubble Sort (non-recursive)** | **O(n)** | **O(n^2)** | The best-case time of O(n) is only when the array is sorted, as the number of swaps required is 0, with only one pass required. In the worst case, every element needs to be compared with every other element (requiring (n/2) passes and O(n) comparisons for every pass), making the algorithm run in quadratic time complexity. |
| **Bubble Sort (recursive)** | **O(n)** | **O(n^2)** | This is similar to the non-recursive bubble sort. Each recursive call makes a single pass through the array of elements. |
| **Selection Sort (non-recursive)** | **O(n^2)** | **O(n^2)** | For each n element, the algorithm makes n-1 comparisons to find the lowest element. The time complexity remains the same regardless of the initial order (since the minimum element cannot be pinpointed unless you go through the array (if it is not at the starting index). |
| **Insertion Sort (non-recursive)** | **O(n)** | **O(n^2)** | The best case time complexity is when the array is sorted since there is only a single iteration. The worst case is when the time to sort a list is proportional to the square of n (the number of elements), meaning the array is in reverse. |
| **Merge Sort (recursive)** | **O(n log n)** | **O(n log n)** | Because of divide-and-conquer, the array will still be divided, and all elements will be compared and merged. Hence, the time complexity stays O(n log n). |
| **Quick Sort (recursive)** | **O(n log n)** | **O(n^2)** | The best case is when the pivot is in the middle of the array, dividing it evenly. The worst case is when it is at the beginning or the end, resulting in an unbalanced partition. (When the pivot divides the array evenly, it is a balanced partition. ) |