**Comparison among Bubble Sort, Selection Sort and Insertion Sort**

1. Bubble Sort:

Bubble sort repeatedly compares and swaps(if needed) adjacent elements in every pass. In i-th pass of Bubble Sort (ascending order), last (i-1) elements are already sorted, and i-th largest element is placed at (N-i)-th position, i.e. i-th last position.

**Time Complexity:**

Best Case Sorted array as input. Or almost all elements are in proper place. [ O(N) ]. O(1) swaps.

Worst Case: Reversely sorted / Very few elements are in proper place. [ O(N2) ] . O(N2) swaps.

Average Case: [ O(N2) ] . O(N2) swaps.

Space Complexity: A temporary variable is used in swapping [ auxiliary, O(1) ]. Hence it is In-Place sort.

**Advantage:**

* It is the simplest sorting approach.
* Best case complexity is of O(N) [for optimized approach] while the array is sorted.
* Using optimized approach, it can detect already sorted array in first pass with time complexity of O(1).
* Stable sort: does not change the relative order of elements with equal keys.
* In-Place sort.

**Disadvantage:**

* Bubble sort is comparatively slower algorithm.

1. Selection Sort

Selection sort selects i-th smallest element and places at i-th position. This algorithm divides the array into two parts: sorted (left) and unsorted (right) subarray. It selects the smallest element from unsorted subarray and places in the first position of that subarray (ascending order). It repeatedly selects the next smallest element.

**Time Complexity:**

Best Case [ O(N2) ]. Also O(N) swaps.

Worst Case: Reversely sorted, and when inner loop makes maximum comparison. [ O(N2) ] . Also O(N) swaps.

Average Case: [ O(N2) ] . Also O(N) swaps.

Space Complexity: [ auxiliary, O(1) ]. In-Place sort.

**Advantage:**

* It can also be used on list structures that make add and remove efficient, such as a linked list. Just remove the smallest element of unsorted part and end at the end of sorted part.
* Best case complexity is of O(N) while the array is already sorted.
* Number of swaps reduced. O(N) swaps in all cases.
* In-Place sort.

**Disadvantage:**

* Time complexity in all cases is O(N2), no best case scenario.

1. Insertion Sort

Insertion Sort is a simple comparison based sorting algorithm. It inserts every array element into its proper position. In i-th iteration, previous (i-1) elements (i.e. subarray Arr[1:(i-1)]) are already sorted, and the i-th element (Arr[i]) is inserted into its proper place in the previously sorted subarray.

**Time Complexity:**

Best Case Sorted array as input, [ O(N) ]. And O(1) swaps.

Worst Case: Reversely sorted, and when inner loop makes maximum comparison, [ O(N2) ] . And O(N2) swaps.

Average Case: [ O(N2) ] . And O(N2) swaps.

Space Complexity: [ auxiliary, O(1) ]. In-Place sort.

**Advantage:**

* It can be easily computed.
* Best case complexity is of O(N) while the array is already sorted.
* Number of swaps reduced than bubble sort.
* For smaller values of N, insertion sort performs efficiently like other quadratic sorting algorithms.
* Stable sort.
* Adaptive: total number of steps is reduced for partially sorted array.
* In-Place sort.

**Disadvantage:**

* It is generally used when the value of N is small. For larger values of N, it is inefficient.

**Time and Space Complexity:**

| **SORTING ALGORITHM** | **TIME COMPLEXITY** | | | **SPACE COMPLEXITY** |
| --- | --- | --- | --- | --- |
|  | Best Case | Average Case | Worst Case | Worst Case |
| Bubble Sort | **O(N)** | **O(N2)** | **O(N2)** | **O(1)** |
| Selection Sort | **O(N2)** | **O(N2)** | **O(N2)** | **O(1)** |
| Insertion Sort | **O(N)** | **O(N2)** | **O(N2)** | **O(1)** |