Test Case Report

Input Array
[150, 397, 239, 192, 254, 995, 310]

Sorted array
150 192 239 254 310 397 995

The following run times are shown on the above Input array for different algorithms:

Bubble Sort : Runtime of the program is 0.0009490447998046875s.
Insertion Sort : Runtime of the program is 7.6253204345703125e-05s.
Selection Sort: Runtime of the program is 0.0009731250762939453s.
Exchange Sort: Runtime of the program is 0.0005978874206542968 8s.

Arranging in ascending order:

Insertion Sort < Exchange Sort < Bubble Sort < Selection Sort

Bubble Sort:

Bubble sort repeatedly compares and swaps(if needed) adjacent elements in every pass. In the 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:

Worst and Average Case Time Complexity: O(n*n).
Worst case occurs when array is reverse sorted. / Very few elements are in proper place.

- [O(N^2)] . O(N2) swaps

Best Case Time Complexity: O(N)
Best case occurs when array is already sorted. Or almost all elements are in proper place.

- [O(N)]. O(1) swaps.

Boundary Cases: Bubble sort takes minimum time (Order of n) when elements are already sorted.

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.

We can easily get the time complexity by examining the for loops in the Selection Sort algorithm.
For a list with n elements, the outer loop iterates n times.
The inner loop iterate n-1 when i is equal to 1, and then n-2 as i is equal to 2 and so forth.
The amount of comparisons are (n - 1) + (n - 2) + ... + 1, which gives Selection Sort a time complexity of O(n^2).

Time Complexity:

Best Case: [O(N^2)]. And O(1) swaps.

Worst Case: Reversely sorted, and when the inner loop makes a maximum comparison. [O(N^2)] . Also, O(N) swaps.

Average Case: [O(N^2)] . Also O(N) swaps.

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.

In the worst case scenario, an array would be sorted in reverse order. The outer for loop in Insertion Sort function always iterates n-1 times.
In the worst case scenario, the inner for loop would swap once, then swap two and so forth. The amount of swaps would then be 1 + 2 + ... + (n - 3) + (n - 2) + (n - 1) which gives Insertion Sort a time complexity of O(n^2).

Time Complexity:

Best Cas: 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(N^2) swaps.

Average Case: [O(N2)] . And O(N^2) swaps.

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Exchange Sort:

The exchange sort is almost similar as the bubble sort.
The exchange sort compares each element of an array and swaps those elements that are not in their proper position, just like a bubble sort does. The only difference between the two sorting algorithms is the manner in which they compare the elements.
The exchange sort compares the first element with each element of the array, making a swap where is necessary.
In some situations the exchange sort is slightly more efficient than its counterpart the bubble sort. The bubble sort needs a final pass to determine that it is finished, thus is slightly less efficient than the exchange sort, because the exchange sort doesn't need a final pass.

Time Complexity: O(n^2)