**Sort Algorithms**

[I. Introduction. 2](#_Toc80207794)

[II. Contents. 3](#_Toc80207795)

[1. Selection Sort. 3](#_Toc80207796)

[2. Insertion Sort. 3](#_Toc80207797)

[3. Binary Insertion Sort. 3](#_Toc80207798)

[4. Interchange Sort. 3](#_Toc80207799)

[5. Bubble Sort. 3](#_Toc80207800)

[6. Shaker Sort. 3](#_Toc80207801)

[7. Shell Sort. 3](#_Toc80207802)

[8. Heap Sort. 3](#_Toc80207803)

[9. Quick Sort. 3](#_Toc80207804)

[10. Merge Sort. 3](#_Toc80207805)

[11. Radix Sort. 3](#_Toc80207806)

[III. Applications. 3](#_Toc80207807)

# **I. Introduction.**

* Sorting algorithms are used in many data structures and applications to reorder the members of the list based on the comparison between elements.
* There are a variety of different sorting algorithms which will come with different time and space complexity and depend on the case of the data to use the suitable sorting algorithm for it.
* To be easier for illustration, the structure that we are going to sort in this topic is a consecutive sequence of integers (array of integers)
* They can also be classified in many way like in-place, recursion, non-recursion, stability,…

# **II. Contents.**

1. **Selection Sort.**

* Selection Sort proceeds by finding the smallest (or largest, depending on sorting order) element in the right unsorted subarray, exchanging it with the leftmost unsorted element (putting it in sorted order), and expand the sublist boundaries one element to the right.
* Selection sort is an in-place comparison sorting algorithm.
* Time Complexity: O(n2)
* Advantages:
  + It takes the elements to their final positions in the sorted array.
  + Selection sort is typically used in situations where the auxiliary memory is limited because it is a in-place sorting algorithm.
  + Easy to implement.
* Disadvantages: High time complexity.
* Example:

Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

Sort in acsending order:



Example 1 - 1

Sort in desending order:



Example 1 - 2

Array B = {**1, -678, 952, -357, -711**}.

Sort in acsending order:



Example 1 - 3

1. **Insertion Sort.**

* Insertion Sort is a simple sorting algorithm. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part by making some comparisons: if the element is smaller than its predecessor, compare it with the elements before. Move the greater elements one position up to make space for the swapped element.
* Insertion Sort is an in-place and stable comparison sorting algorithm.
* Advantages:
  + Real-time sorting, data may be incomplete or coming, but the array is still sortable.
  + Efficient for small data sets, much like other quadratic sorting algorithms and more efficient than most other simple quadratic algorithms such as *Selection Sort* or *Bubble Sort*
  + Easy to implement.
* Disadvantages:
  + High time complexity.
  + It is much less efficient on large lists than more advanced algorithms such as *Quick Sort*, *Heap Sort*, or *Merge Sort*.
* Example:

Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

Sort in acsending order:



Example 2 - 1

Sort in desending order:



Example 2 - 2

Array B = {**1, -678, 952, -357, -711**}.

Sort in desending order:



Example 2 - 3

1. **Binary Insertion Sort.**

* Binary insertion Sort is a variant of the *Insertion Sort*. Because of finding the insert position is sequential in *Insertion Sort*. For a better way, we use the *Binary Search* method to find this position.
* Binary insertion Sort employs a binary search to determine the correct location to insert new elements, and therefore performs O(log2n) comparisons in the worst case. When each element in the array is searched for and inserted this is O(nlogn).
* But because of the series of swaps required for each insertion. The algorithm as a whole still has a time complexity of O(n2) on average.
* Advantages: The number of swaps is reduced more than a normal *Insertion Sort*.
* Example:

Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

Sort in acsending order:



Example 3 -

Sort in acsending order:



Example 3 -

Array B = {**1, -678, 952, -357, -711**}.

Sort in acsending order:



Example 3 -

1. **Interchange Sort.**

* Interchange Sort works by comparing the first element with all elements behind it, swapping where needed, it then proceeds to do the same for the second element, and so on.
* Interchange Sort is an in-place comparison sort.
* Time complexity: O(n2)
* Advantages:
  + Like any simple O(n2) sort it can be reasonably fast over very small data sets, though in general insertion sort will be faster.
  + Easy to implement.
* Disadvantages:
  + High time complexity.
  + It is much less efficient on large lists.
* Example:

Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

Sort in acsending order:



Sort in desending order:

Array B = {**1, -678, 952, -357, -711**}.

Sort in desending order:

1. **Bubble Sort.**

* Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.
* Bubble Sort is an in-place comparison sort.
* Time complexity: O(n2)
* Advantages:
  + Easy to implement.
* Disadvantages:
  + High time complexity.
  + It is much less efficient on large lists.
* Example:

Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

1. **Shaker Sort.**

* Shaker Sort is an extension of *Bubble Sort*.
* The algorithm extends *Bubble Sort* by operating in two directions. While it improves on *Bubble Sort* by more quickly moving items to the beginning of the list.
* Shaker Sort is an in-place comparison sort.
* Time complexity: O(n2)
* Advantages:
  + If the list is mostly ordered before applying the sorting algorithm it becomes closer to O(n) time complexity.
* Disadvantages:
  + High time complexity.
  + It is much less efficient on large lists.
  + Even it is an extension of *Bubble Sort* but it still not be better than *Insertion Sort.*
* Example:

Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

1. **Shell Sort.**

* Shell Sort is a modified version of *Insertion Sort* that allows the exchange of items that are far apart.
* The sorting algorithm compares elements separated and using *Insertion Sort* methods but by a distance and decreases on each pass.
* Shell Sort is an in-place comparison sort.
* Time complexity: O(n2)
* Advantages:
  + Shell Sort has distinctly improved running times more than *Insertion Sort.*
* Example:

Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

1. **Heap Sort.**

* Heap Sort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region.
* The concept of it seems like it is very similar with *Selection Sort*, but Heap Sort does not waste time with a linear-time scan of the unsorted region.Rather, Heap Sort maintains the unsorted region in a *Heap Data Structure* or *Binary Heap* to more quickly find the largest element in each step.
* A *Binary Heap* is a complete *Binary Tree* where items are stored in a special order such that the value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called *Max Heap* and the latter is called *Min Heap*. The heap can be represented by a binary tree or array.
* Heap sort is an in-place comparison sorting algorithm, but it is not a stable sort.
* Time complexity: O(nlogn)
* Advantages:
  + Low time complexity.
* Disadventages:
  + The implementation of the *Quick Sort* is still better.
* Example:

Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

1. **Quick Sort.**

* It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. This can be done in-place.
* Quick Sort is an in-place and unstable sorting algorithm and also is a divide and conquer algorithm.
* Time complexity: O(nlogn)
* Advantages:
  + Low time complexity. Faster than *Merge Sort* and about 2 or 3 times faster than *Heap Sort*.
* Disadventages:
  + Requiring small additional amounts of memory to perform the sorting.
* Example:

Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

1. **Merge Sort.**

* Merge Sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.
* Merge Sort is an in-place and stable comparison sorting algorithm and also is a divide and conquer algorithm.
* Time complexity: O(nlogn)
* Advantages:
  + Low time complexity.
* Disadventages:
  + Requiring small additional amounts of memory to perform the sorting.
  + Goes through the whole process even if the list is sorted.
* Example:

Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

1. **Radix Sort.**

* The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.
* Radix Sort is a non-comparative stable sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix.
* Time complexity: O(n\*k) (k is the length of the data)
* Advantages:
  + Fast when the keys are short or when the range of the array elements is less.
* Disadvantages:
  + Since Radix Sort depends on digits or letters, Radix Sort is much less flexible than other sorts. So, for every different type of data it needs to be rewritten.
  + It takes more space compared to *Quick Sort* which is inplace sorting.
* Example:

Array A = {**84**, **65**, **148**, **1**, **49**, **26**}.

# **III. Applications.**