1. Selection sort: the selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

+ The subarray which is already sorted.

+ Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

1. Insertion sort: insertion sort is the sorting mechanism where the sorted array is built having one item at a time. The array elements are compared with each other sequentially and then arranged simultaneously in some particular order. The analogy can be understood from the style we arrange a deck of cards. This sort works on the principle of inserting an element at a particular position, hence the name Insertion sort. The first step involves the comparison of the element in question with its adjacent element. And if at every comparison reveals that the element in question can be inserted at a particular position, then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position. The above procedure is repeated until all the element in the array is at their apt position.
2. Bubble sort: bubble sort is one of the simplest sorting algorithms and really intuitive to understand. We compare adjacent elements and see if their order is wrong (a[i] > a[j] for 1 <= i < j <= size of array; if array is to be in ascending order, and vice-versa). If yes, then swap them. For example, we have an array of length ‘n’. To sort this array we do the above step(swapping) for n - 1 passes. In simple terms, in the ith pass, the ith largest element goes at its right place in the array by swappings. First, the largest element goes at its right place then, second largest and so on. In a bit of mathematical terms, in ith pass, at least one element from (n - i + 1) elements from start will go at its right place. That element will be the ith (for 1 <= i <= n - 1) largest element of the array. Because in the ith pass of the array, in the jth iteration (for 1 <= j <= n - i), we are checking if a[j] > a[j + 1], and a[j] will always be greater than a[j + 1] when it is the largest element in range [1, n - i + 1]. Now we will swap it. This will continue until ith largest element is at the (n - i + 1)th position of the array.
3. Heap sort: heap sort can be understood as the improved version of the binary search tree. It does not create a node as in case of binary search tree instead it builds the heap by adjusting the position of elements within the array itself. In which method a tree structure called heap is used where a heap is a type of binary tree. An ordered balanced binary tree is called a Min-heap, where the value at the root of any subtree is less than or equal to the value of either of its children. An ordered balanced binary tree is called a max heap where the value at the root of any subtree is more than or equal to the value of either of its children. It is not necessary that the two children must be in some order. sometimes the value in the left child may be more than the value at the right child and some other time it may be the other way round. Basically, there are two phases involved in the sorting of elements using heap sort algorithm they are as follows: First, start with the construction of a heap by adjusting the array elements. Once the heap is created repeatedly eliminate the root element of the heap by shifting it to the end of the array and then store the heap structure with remaining elements. Suppose an array consists of N distinct elements in memory, then the heap sort algorithm works as follows: To begin with, a heap is built by moving the elements to its proper position within the array. This means that as the elements are traversed from the array the root, its left child, its right child are filled in respectively forming a binary tree. In the second phase, the root element is eliminated from the heap by moving it to the end of the array. The balance elements may not be a heap. So again steps 1 and 2 are repeated for the balance elements. The procedure is continued until all the elements are eliminated. When eliminating an element from the heap we need to decrement the maximum index value of the array by one. The elements are eliminated in decreasing order for a max-heap and in increasing order for min-heap.
4. Binary insertion sort: binary Insertion sort is a variant of Insertion sorting in which proper location to insert the selected element is found using the binary search. Binary search reduces the number of comparisons in order to find the correct location in the sorted part of data. In normal insertion sort, it takes O(n) comparisons(at nth iteration) in worst case. We can reduce it to O(log n) by using binary search.
5. Shaker sort: shaker sort is also based on the principle of direct swapping but finds out the disadvantages of Bubble Sort. In each arrangement, browse the array in 2 turns from 2 different sides. Turn 1: push the small element to the beginning of the array, turn 2: push the large element to the end of the array. Record sorted sections to save redundant comparisons. The worst complexity of this algorithm is O(n^2).
6. Shell sort: shell sort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of shell sort is to allow exchange of far items. In shell sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h’th element is sorted. Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left. This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval. This interval is calculated based on Knuth's formula as – h = h \* 3 + 1 where h is interval with initial value 1. This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is Ο(n), where n is the number of items. And the worst case space complexity is O(n).
7. Mege sort: merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. 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.

+ Divide the unsorted list into N sublists, each containing 1 element.

+ Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. N will now convert into N/2 lists of size 2.

+ Repeat the process till a single sorted list of obtained.

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists. The list of size N is divided into a max of logN parts, and the merging of all sublists into a single list takes O(n) time, the worst case run time of this algorithm is O(NlogN).

1. Quick sort: quick sort is one of the most efficient sorting algorithms and is based on the splitting of an array into smaller ones. The name comes from the fact that, quick sort is capable of sorting a list of data elements significantly faster than any of the common sorting algorithms. And like Merge sort, Quick sort also falls into the category of divide and conquer approach of problem-solving methodology.

+ Taking the analogical view in perspective, consider a situation where one had to sort the papers bearing the names of the students, by name. One might use the approach as follows:

1. Select a splitting value, say L. The splitting value is also known as Pivot.
2. Divide the stack of papers into two. A-L and M-Z. It is not necessary that the piles should be equal.
3. Repeat the above two steps with the A-L pile, splitting it into its significant two halves. And M-Z pile, split into its halves. The process is repeated until the piles are small enough to be sorted easily.
4. Ultimately, the smaller piles can be placed one on top of the other to produce a fully sorted and ordered set of papers.

+ The approach used here is recursion at each split to get to the single-element array.

+ At every split, the pile was divided and then the same approach was used for the smaller piles.

+ Due to these features, quick sort is also called as partition exchange sort.

1. Counting sort: In counting sort, the frequencies of distinct elements of the array to be sorted is counted and stored in an auxiliary array, by mapping its value as an index of the auxiliary array. Let's assume that, array A of size N needs to be sorted.

+ Initialize the auxiliary array Aux[] as 0. Note: The size of this array should be >= max(A[]).

+ Traverse array A and store the count of occurrence of each element in the appropriate index of the Aux array, which means, execute Aux[A[i]]++ for each i, where i ranges from [0, N - 1].

+ Initialize the empty array sorted A[].

+ Traverse array Aux and copy i into sorted A for Aux[i] number of times where 0 <= i <= max(A[i]).

The array can be sorted by using this algorithm only if the maximum value in array A is less than the maximum size of the array Aux. Usually, it is possible to allocate memory up to the order of a million 106. If the maximum value of A exceeds the maximum memory- allocation size, it is recommended that you do not use this algorithm. Use either the quick sort or merge sort algorithm. The array A is traversed in O(n) time and the resulting sorted array is also computed in O(n) time.Aux[] is traversed in O(K) time. Therefore, the overall time complexity of counting sort algorithm is O(N + K).

1. Radix sort: radix sort is a sorting technique that sorts the elements by first grouping the individual digits of same place value. Then, sort the elements according to their increasing/decreasing order. Suppose, we have an array of 8 elements. First, we will sort elements based on the value of the unit place. Then, we will sort elements based on the value of the tenth place. This process goes on until the last significant place. Let the initial array be [121, 432, 564, 23, 1, 45, 788].

+ Find the largest element in the array, i.e. max. Let X be the number of digits in max. X is calculated because we have to go through all the significant places of all elements. In this array [121, 432, 564, 23, 1, 45, 788], we have the largest number 788. It has 3 digits. Therefore, the loop should go up to hundreds place (3 times).

+ Now, go through each significant place one by one. Use any stable sorting technique to sort the digits at each significant place. We have used counting sort for this. Sort the elements based on the unit place digits (X=0).

+ Now, sort the elements based on digits at tens place.

+ Finally, sort the elements based on the digits at hundreds place.

Since radix sort is a non-comparative algorithm, it has advantages over comparative sorting algorithms. For the radix sort that uses counting sort as an intermediate stable sort, the time complexity is O(d(n + k)). Here, d is the number cycle and O(n+k) is the time complexity of counting sort. Thus, radix sort has linear time complexity which is better than O(nlog n) of comparative sorting algorithms. If we take very large digit numbers or the number of other bases like 32-bit and 64-bit numbers then it can perform in linear time however the intermediate sort takes large space. This makes radix sort space inefficient. This is the reason why this sort is not used in software libraries.

1. Flash sort: flash sort is a distribution sorting algorithm showing linear computational complexity O(n) for uniformly distributed data sets and relatively little additional memory requirement. The original work was published in 1998 by Karl-Dietrich Neubert. The basic idea behind flash sort is that in a data set with a known distribution, it is easy to immediately estimate where an element should be placed after sorting when the range of the set is known. The basic idea behind flashsort is that in a data set with a known distribution, it is easy to immediately estimate where an element should be placed after sorting when the range of the set is known. For example, if given a uniform data set where the minimum is 1 and the maximum is 100 and 50 is an element of the set, it's reasonable to guess that 50 would be near the middle of the set after it is sorted. This approximate location is called a class. If numbered 1 to m, the class of an item Ai is the quantile, computed as:

K(Ai) = 1 + INT((m – 1) ) where A is the input set. The range covered by every class is equal, except the last class which includes only the maximum(s). The classification ensures that every element in a class is greater than any element in a lower class. This partially orders the data and reduces the number of inversions. Insertion sort is then applied to the classified set. As long as the data is uniformly distributed, class sizes will be consistent and insertion sort will be computationally efficient. The only extra memory requirements are the auxiliary vector L for storing class bounds and the constant number of other variables used. In the ideal case of a balanced data set, each class will be approximately the same size, and sorting an individual class by itself has complexity O(1). If the number {\displaystyle m}m of classes is proportional to the input set size n, the running time of the final insertion sort is m.O(1) = O(m) = O(n). In the worst-case scenarios where almost all the elements are in a few or one class, the complexity of the algorithm as a whole is limited by the performance of the final-step sorting method. For insertion sort, this is O(n2). Variations of the algorithm improve worst-case performance by using better-performing sorts such as quicksort or recursive flashsort on classes that exceed a certain size limit. Choosing a value for m, the number of classes, trades off time spent classifying elements (high m) and time spent in the final insertion sort step (low m). Based on his research, Neubert found m = 0.42n to be optimal. Due to the in situ permutation that flashsort performs in its classification process, flashsort is not stable. If stability is required, it is possible to use a second, temporary, array so elements can be classified sequentially. However, in this case, the algorithm will require O(n) space.