**Sorting Algorithm:**

1. **The selection sort** algorithm maintains two sub-arrays in a given array:

* The sub-array which is already sorted.
* Remaining sub-array which is unsorted.

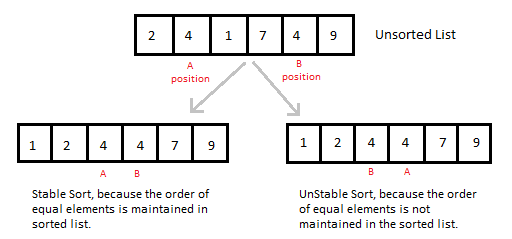
In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted sub-array is picked and moved to the sorted sub-array.

* Time Complexity: O(n\*n) as there are two nested loops.
* Auxiliary Space: O(1)

1. **Bubble Sort** is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

* Time Complexity: O(n\*n)
* Auxiliary Space: O(1)

1. **Insertion sort** is a simple sorting algorithm that works the way we sort playing cards in our hands. In this algorithm, we pick up a key and compare it with elements ahead of it, and put the key in the right place. It is Stable, as it does not change the relative order of elements with equal keys



* Time Complexity: O(n\*n)
* Auxiliary Space: O(1)

1. **Merge Sort** is a Divide and Conquer algorithm. It divides input array in two halves till leaf node (of a tree structure) and then merges the two **sorted** halves.

* Time Complexity: O(nlogn) in all 3 cases (worst, average and best).
* Auxiliary Space: O(n)

Merge Sort is useful for sorting linked lists in O(nLogn) time. Other nlogn algorithms like Heap Sort, Quick Sort (average case nLogn) cannot be applied to linked lists. It is used in External Sorting (for big data).

1. **Heap Sort** Algorithm for sorting in increasing order:

* Build a max heap (A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in its two children nodes. The former is called as max heap) from the input data.
* At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
* Repeat above steps until size of heap is greater than 1.

Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn). Priority queues can be efficiently implemented using Binary Heap.

1. **Quick Sort** is a Divide and Conquer algorithm. It picks an element as pivot (leftmost, rightmost, random, median) and partitions the given array around the picked pivot. Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

Time taken by Quick Sort in general can be written as following.

**T(n) = T(k) + T(n-k-1) + O(n)**

The first two terms are for two recursive calls and the last term is for the partition process. k is the number of elements which are smaller than pivot. The worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case. T(n) = T(0) + T(n-1) + O(n) which is equivalent to O(n^2). The best case occurs when the partition process always picks the middle element as pivot. T(n) = 2T(n/2) + O(n)

The solution of above recurrence is O(nLogn). Following is recurrence for average case. T(n) = T(n/9) + T(9n/10) + O(n) Solution of above recurrence is also O(nLogn)

1. **Counting/Bucket Sort** is a sorting technique based on keys between a specific range

* Take a count array to store the count of each unique object.
* Modify the count array such that each element at each index stores the sum of previous counts.
* Output each object from the input sequence followed by decreasing its count by 1.
* Time Complexity: O(n+k) where n is the number of elements in input array and k is the range of input.
* Auxiliary Space: O(n+k)

1. **Shell sort**, sometimes called the “diminishing increment sort,” improves on the insertion sort by breaking the original list into a number of smaller sub-lists, each of which is sorted using an insertion sort. For example, if we use an increment of three, there are three sub-lists, each of which can be sorted by an insertion sort. Although this list is not completely sorted, something very interesting has happened. By sorting the sub-lists, we have moved the items closer to where they actually belong. Final insertion sort will sort this list completely now.

* Time Complexity: O(n\*n)

**Searching Algorithm:**

1. **Binary Search** works with sorted arrays and reduces the time complexity to O(Logn). It basically ignores half of the elements just after one comparison.
2. **Interpolation search** also work with sorted arrays and tries to follow the way we search a name in a phone book, or a word in the dictionary. We, humans, know in advance that in case the name we’re searching starts with a “B”, like “Bond” for instance, we should start searching near the beginning of the phone book. We can define the following formula:

C = (x-L)/(R-L) where x is index of searched item and L/R are boundaries

On average the interpolation search makes about log(log(n)) comparisons (if the elements are uniformly distributed. In the worst case (for instance where the numerical values of the keys increase exponentially) it can make up to O(n) comparisons.

1. **Jump Search** works similar to Binary Sort but step size is √n and complexity is O(√n).

**Data Structure:**

1. A **trie** is a tree-like data structure in which each node contains an array of 26 pointers, one pointer for each character in the alphabet. The **ternary** **search** **tree** contains three types of arrows. First, traversing a down-arrow corresponds to “matching” the character from which the arrow starts. The left- and right- arrow are traversed when the current character does not match the desired character at the current position. We take the left-arrow if the character we are looking for is alphabetically before the character in the current node and the right-arrow in opposite case. This can be used for efficient auto-complete.
2. Quick Sort in its general form is an in-place sort whereas merge sort requires O(N) extra storage. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. For arrays, merge sort loses due to the use of extra O(N) storage space. In arrays, we can do random access as elements are continuous in memory. Quick Sort requires a lot of this kind of access. So **Quick Sort preferred for Arrays and Merge Sort for Linked Lists.**
3. To calculate middle element “m = (l+r)/2″, fails if the sum of low and high is greater than the maximum positive int value (231 – 1). The sum overflows to a negative value. So should use :

**int mid = (low + high) >>> 1;**