Searching and sorting

***Linear Search

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.
It compares the element to be searched with all the elements present in the array and when the element is **matched** successfully, it returns the index of the element in the array, else it return -1.
Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

Features of Linear Search Algorithm

1. It is used for unsorted and unordered small list of elements.
2. It has a time complexity of **O(n)**, which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.
3. It has a very simple implementation.

***Binary Search*

Binary Search is used with sorted array or list. In binary search, we follow the following steps:

1. We start by comparing the element to be searched with the element in the middle of the list/array.
2. If we get a match, we return the index of the middle element.
3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.
5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

Binary Search is useful when there is large number of elements in an array and they are sorted.
So a necessary condition for Binary search to work is that the list/array should be sorted.

Features of Binary Search

1. It is great to search through large sorted arrays.
2. It has a time complexity of **O(log n)** which is a very good time complexity.
3. It has a simple implementation

***Linear Search Algorithm**

Linear Search (Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit

***Binary search algorithm**

```
function binary_search(A, n, T) is
   L := 0
   R := n − 1
   while L ≤ R do
      m := floor((L + R) / 2)
      if A[m] < T then
         L := m + 1
      else if A[m] > T then
         R := m - 1
      else:
         return m
   return unsuccessful
```

Comparison between Linear search and Binary search

- A linear search scans one item at a time, without jumping to any item. In contrast, binary search cuts down your search to half as soon as you find the middle of a sorted list.
- In linear search, the worst case complexity is O(n), where binary search making O(log n) comparisons.
- Time taken to search elements keep increasing as the number of elements is increased when searching through linear process. But binary search compresses the searching period by dividing the whole array into two half.
- Linear search does the sequential access whereas Binary search access data randomly.

- Input data needs to be sorted in Binary Search and not in Linear Search.
- In linear search, performance is done by equality comparisons. In binary search, performance is done by ordering comparisons.
- Binary search is better and quite faster than linear search.
- Linear search uses sequential approach. But, binary search implements divide and conquer approach.
- Linear search is quick and easy to use, but there is no need for any ordered elements. Where binary search algorithm is tricky, and elements are necessarily arranged in order.
- The best case time in linear search is for the first element that is O(1). And the other side O(1) is the middle element in binary search.
- Linear search can be implemented in an array as well as in linked list, but binary search can't be implemented directly on linked list.
- Binary search is efficient for the larger array. If the amount of data is small, then linear search is preferable because this searching process is fast when data is small.

***Bubble sort** is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

***This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name**, insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted

***Selection sort** is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

Algorithm

Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted order.

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(nLogn) and image.png(n^2), respectively.

Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot

**Comparison of all sorting

Comparison Based Soring techniques are bubble sort, selection sort, insertion sort, Merge sort, quicksort, heap sort etc. These techniques are considered as comparison based sort because in these techniques the values are compared, and placed into sorted position in ifferent phases. Here we will see time complexity of these techniques.

Analysis Type	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Quick Sort	Heap Sort
Best Case	$O(n^2)$	$O(n^2)$	O(n)	O(log n)	O(log n)	O(logn)
Average Case	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(log n)	O(log n)	O(log n)
Worst Case	$O(n^{2)}$	$O(n^2)$	$O(n^2)$	O(log n)	$O(n^2)$	O(log n)

some sorting algorithms are non-comparison based algorithm. Some of them are Radix sort, Bucket sort, count sort. These are non-comparison based sort because here two elements are not compared while sorting. The techniques are slightly different.