Q.1	Explain Bubble sort and its algorithm.
Ans.	**Bubble sort:**

- Bubble Sort is an algorithm which is used to sort N elements that are given in a memory for eg: an Array with N number of elements. Bubble Sort compares all the element one by one and sort them based on their values.
- It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.
- Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

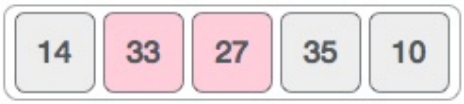
How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this –

tic_analysis.htm

Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –

Notice that after each iteration, at least one value moves at the end.

And when there's no swap required, bubble sorts learns that an array is completely sorted.

Now we should look into some practical aspects of bubble sort.

Algorithm for Bubble Sort

- algorithm Bubble_Sort(list)
- Pre: list != fi
- Post: list is sorted in ascending order for all values
- for i <- 0 to list:Count – 1
- for j <- 0 to list:Count – 1
- if list[i] < list[j]
- Swap(list[i]; list[j])
- end if
- end for
- end for
- return list
- end Bubble_Sort

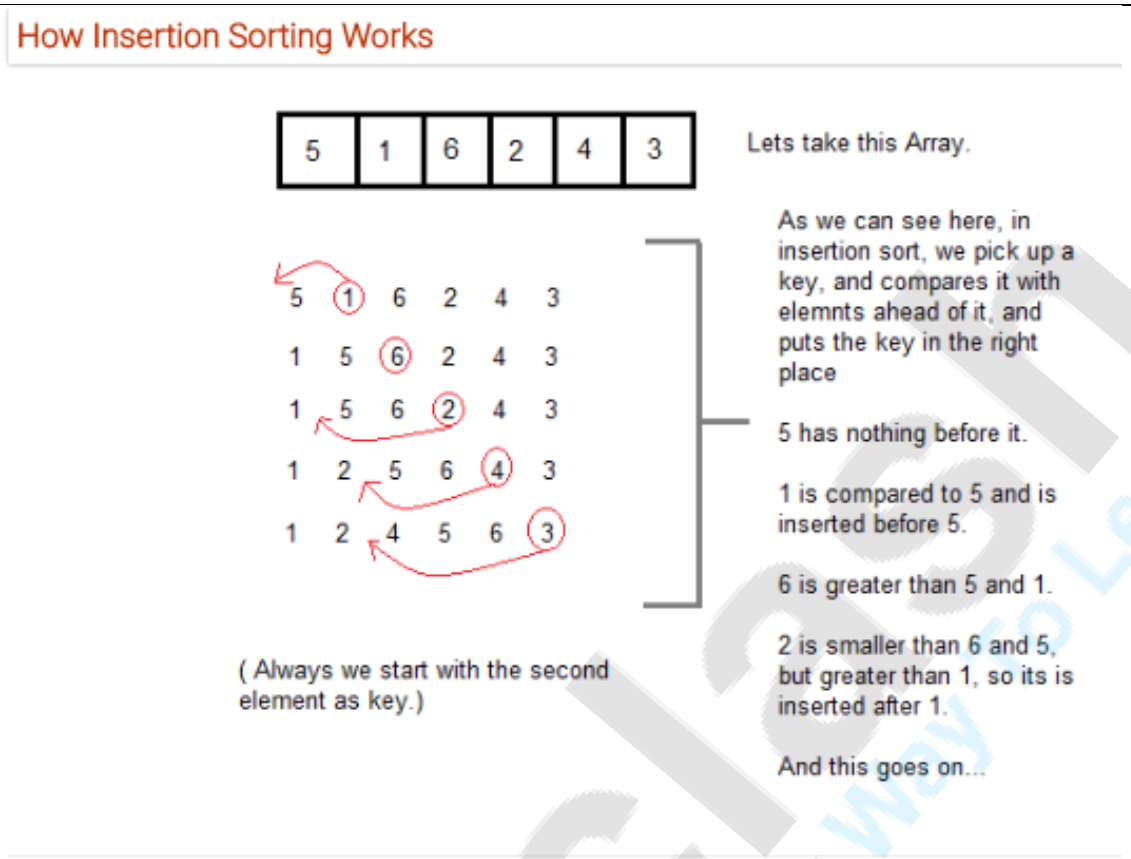
Q2. Explain Insertion sort with algorithm.

Ans.

Insertion sort:

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less, like Bubble Sorting, inerstion sort also requires a single additional memory space.
6. It is Stable, as it does not change the relative order of elements with equal keys



Algorithm:

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

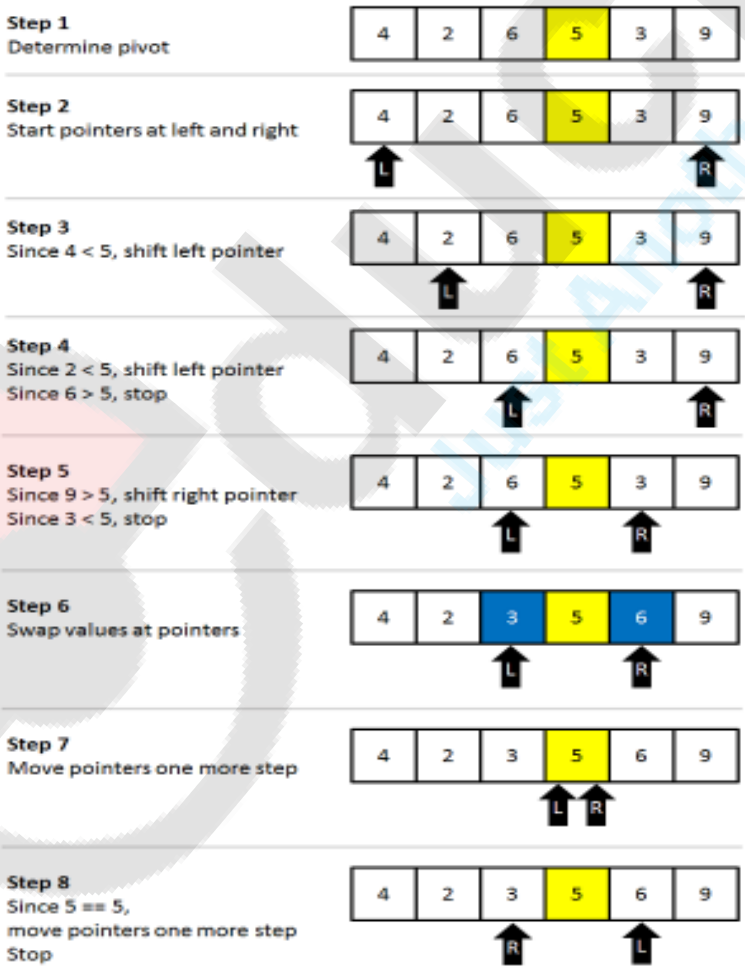
Q.3	Explain Quick sort.
Ans.	**Quick sort:** • Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less aditional space. It is based on the rule of Divide and Conquer(also called partition-exchange sort). This algorithm divides the list into three main parts : • Elements less than the Pivot element • Pivot element • Elements greater than the pivot element • In the list of elements, mentioned in below example, we have taken 25 as pivot. So after the first pass, the list will be changed like this.

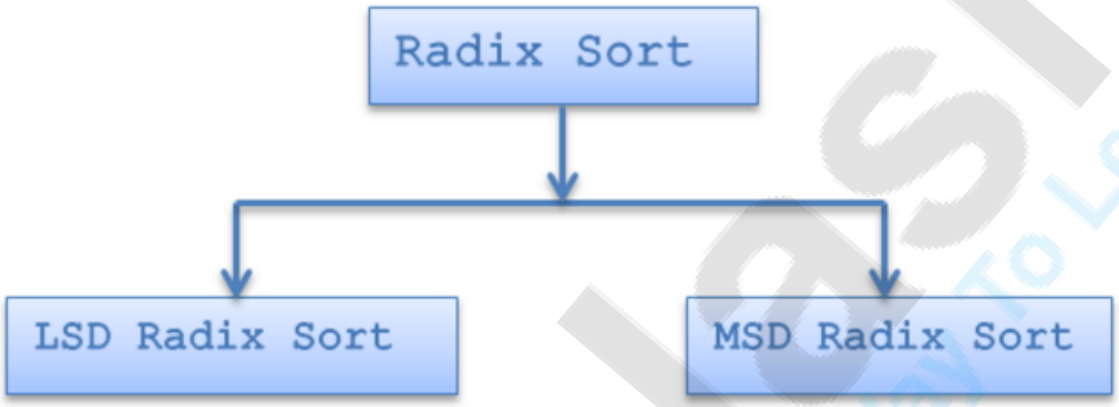
- 6 8 17 14 25 63 37 52

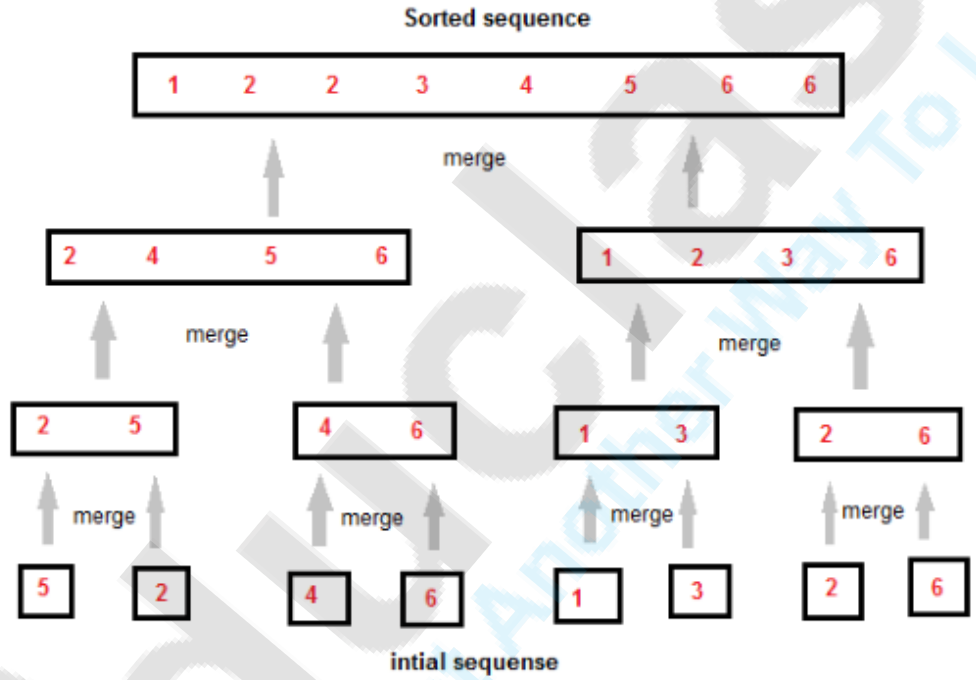
- Hnece after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.

- **Algorithm:**

- 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

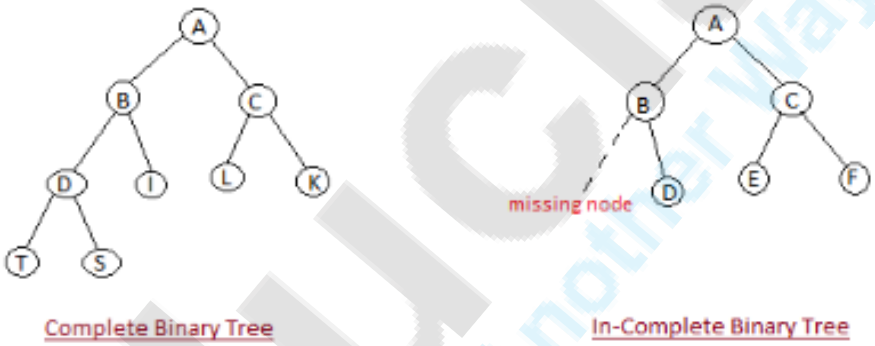


Q.5	What is Radix sort and its type?
Ans.	**Radix Sort:** • Radix sort was developed for sorting large integers, but it treats an integer as a string of digits, so it is really a string sorting algorithm. • Radix sort is a non-comparative sorting algorithm that sorts data with keys by grouping keys by the individual digits which share the same significant position and value. • Radix Sort arranges the elements in order by comparing the digits of the numbers  **LSD radix sort** • Least-significant-digit-first radix sort. • LSD radix sorts process the integer representations starting from the least significant digit and move the processing towards the most significant digit. **MSD radix sort** • Most-significant-digit-first radix sort. • MSD radix sort starts processing the keys from the most significant digit, leftmost digit, to the least significant digit, rightmost digit. This sequence is opposite that of least significant digit (LSD) radix sorts **Algorithm** • This sorting **algorithm** doesn't compare the numbers but distributes them, it works as follows: 1. Sorting takes place by distributing the list of number into a bucket by passing through the individual digits of a given number one-by-one beginning with the least significant part. Here, the number of buckets are a total of ten, which bare key values starting from 0 to 9. 2. After each pass, the numbers are collected from the buckets, keeping the numbers in order 3. Now, recursively redistribute the numbers as in the above step '1' but with a following reconsideration: take into account next most significant part of the number, which is then followed by above step '2'.

Q. 6	Explain Merge sort and how it works ?
Ans	**Merge sort:** • Merge Sort follows the rule of Divide and Conquer. But it doesn't divides the list into two halves. In merge sort the unsorted list is divided into N sub lists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sub lists, to produce new sorted sub lists, and at lasts one sorted list is produced. • Merge Sort is quite fast, and has a time complexity of O(n log n). It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list. How Merge Sort Works  **Complexity Analysis of Merge Sort** • Time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves. • It requires equal amount of additional space as the unsorted list. Hence its not at all recommended for searching large unsorted lists. • It is the best Sorting technique for sorting Linked Lists.

Algorithm for Merge Sort

- algorithm Merge_Sort(list)
- Pre: list 6= fı
- Post: list has been sorted into values of ascending order
- if list.Count = 1 // when already sorted
- return list
- end if
- m <- list.Count = 2
- left <- list(m)
- right <- list(list.Count – m)
- for i <- 0 to left.Count – 1
- left[i] <- list[i]
- end for
- for i <- 0 to right.Count -1
- right[i] <- list[i]
- end for
- left <- Merg_Sort(left)
- 17) right <- Merge_Sort(right)
- 18) return MergeOrdered(left, right)
- 19) end Merge_Sort

Q.7	Explain Heap sort.
Ans.	**Heap Sort Algorithm** Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts : • Creating a Heap of the unsorted list. • Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal. ## What is a Heap ? Heap is a special tree-based data structure, that satisfies the following special heap properties : **Shape Property :** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.  Complete Binary Tree In-Complete Binary Tree **Heap Property :** All nodes are either [greater than or equal to] or [less than or equal to] each of its children. If the parent nkodes are greater than their children, heap is called a Max-Heap, and if the parent nodes are smalled than their child nodes, heap is called Min-Heap.  Min-Heap Max-Heap In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements. In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

Implementing Heap-Sort In-Place

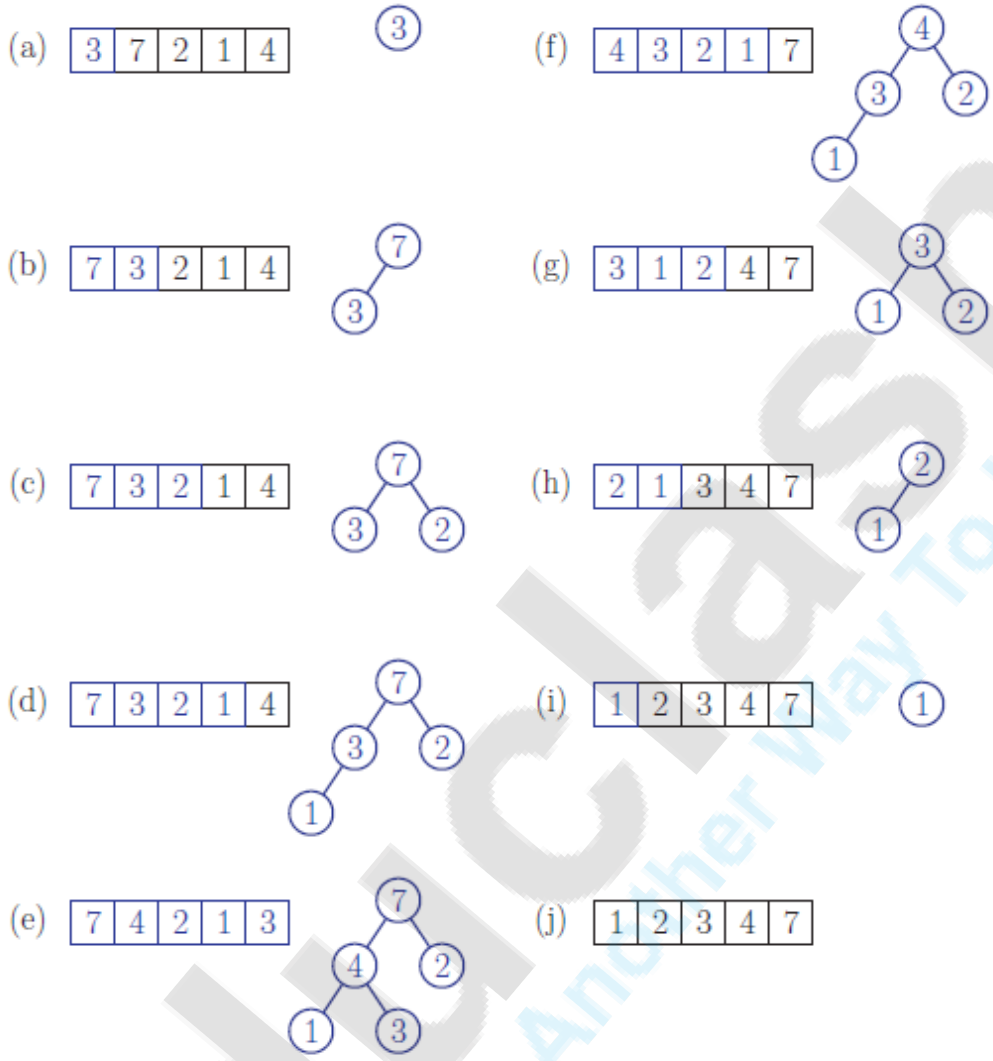
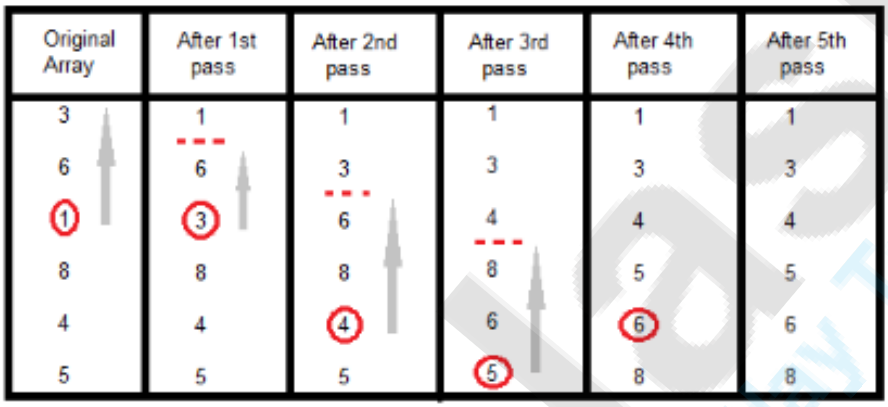
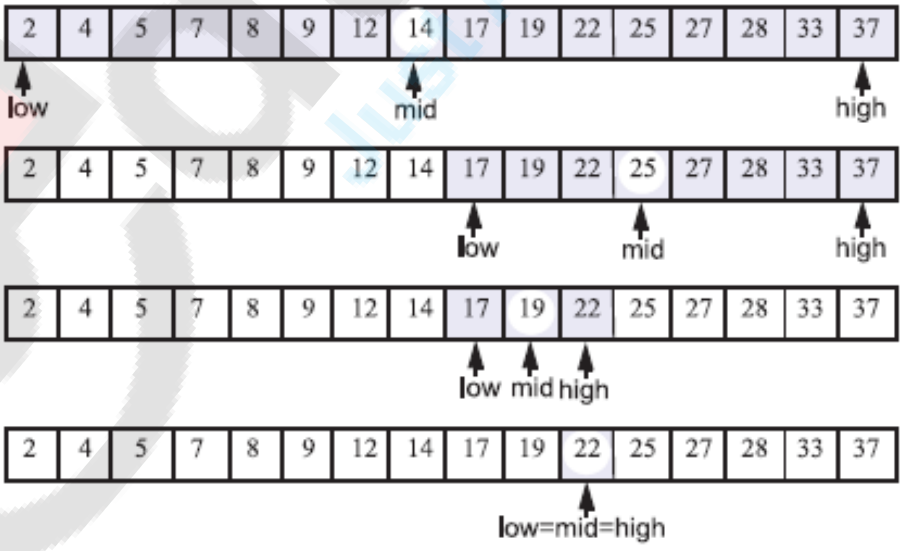


Figure 8.9: In-place heap-sort. Parts (a) through (e) show the addition of elements to the heap; (f) through (j) show the removal of successive elements. The portions of the array that are used for the heap structure are shown in blue.

Q. 8	Explain Selection sort.
Ans.	**Selection Sorting:** Selection sorting is conceptually the most simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted. How Selection Sorting Works  In the first pass, the smallest element found is 1, so it is placed at the first position, then leaving first element, smallest element is searched from the rest of the elements, 3 is the smallest, so it is then placed at the second position. Then we leave 1 nad 3, from the rest of the elements, we search for the smallest and put it at third position and keep doing this, until array is sorted. **The selection sort algorithm is performed using following steps...** Step 1: Select the first element of the list (i.e., Element at first position in the list). Step 2: Compare the selected element with all other elements in the list. Step 3: For every comparison, if any element is smaller than selected element (for Ascending order), then these two are swapped. Step 4: Repeat the same procedure with next position in the list till the entire list is sorted.

Q. 9	What is Shell sort ?
Ans.	**Shell sort:** SHELL SORT ALGORITHM- EXPLANATION, IMPLEMENTATION AND COMPLEXITY Shell Sort is a generalized version of insertion sort. It is an in–place comparison sort. Shell Sort is also known as diminishing increment sort, it is one of the oldest sorting algorithms invented by Donald L. Shell (1959.) **Here are some key points of shell sort algorithm –** • Shell Sort is a comparison based sorting. • Time complexity of Shell Sort depends on gap sequence . Its best case time complexity is O(n* log) and worst case is O(n* log2n). Time complexity of Shell sort is generally assumed to be near to O(n) and less than O(n2) as determining its time complexity is still an open problem. • The best case in shell sort is when the array is already sorted. The number of comparisons is less. • It is an in-place sorting algorithm as it requires no additional scratch space. • Shell Sort is unstable sort as relative order of elements with equal values may change. • It is been observed that shell sort is 5 times faster than bubble sort and twice faster than insertion sort its closest competitor. • There are various increment sequences or gap sequences in shell sort which produce various complexity between O(n) and O(n2). input: an array num of length n with array elements numbered 0 to n - 1 Shell.Sort(num,n,key) 1. Assign, span = int(n/2) 2. while span > 0 do: a) for i from span to n - 1, Repeat step b,c,e b) assign num[i] to key and i to j c) while j = span and num[j - span] > key, Repeat step d d) swap num[j] and num[j - span] e) Assign, span = int(span / 2.2) 3. Use Insertion Sort to sort remaining array of data

Q. 10	Explain Linear search.
Ans.	**Searching:** • An algorithm is a step-by-step procedure or method for solving a problem by a computer in a given number of steps. • The steps of an algorithm may include repetition depending upon the problem for which the algorithm is being developed. • The algorithm is written in human readable and understandable form. To search an element in a given array, it can be done in two ways Linear search and Binary search. **Linear search:** • A linear search is the basic and simple search algorithm. • A linear search searches an element or value from an array till the desired element or value is not found and it searches in a sequence order. • It compares the element with all the other elements given in the list and if the element is matched it returns the value index else it return -1. • Linear Search is applied on the unsorted or unordered list when there are fewer elements in a list. Example with Implementation To search the element 5 it will go step by step in a sequence order.  **Linear search is implemented using following steps...** Step 1: Read the search element from the user Step 2: Compare, the search element with the first element in the list. Step 3: If both are matching, then display "Given element found!!!" and terminate the function Step 4: If both are not matching, then compare search element with the next element in the list. Step 5: Repeat steps 3 and 4 until the search element is compared with the last element in the list. Step 6: If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.

Q.11	Explain Binary Search.
Ans.	**Binary search algo:** Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquers. For this algorithm to work properly, the data collection should be in the sorted form. • Binary search looks for a particular item by comparing the middle most item of the collection. • If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. • Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero **Implementation binary search:** First, we shall determine half of the array by using this formula – `mid = low + (high - low) / 2` We illustrate the binary search algorithm in Figure 9.7. 

Example of a binary search to perform operation find(22), in a map with integer keys, implemented with an ordered vector. For simplicity, we show the keys, not the whole entries.

Algorithm BinarySearch(*L*,*k*, *low*,*high*):
Input: An ordered vector *L* storing *n* entries and integers *low* and *high*
Output: An entry of *L* with key equal to *k* and index between *low* and *high*, if such an entry exists, and otherwise the special sentinel end
if *low* > *high* **then**
return end
else
mid←⌊(*low*+*high*)/2⌋ *e*←*L*.at(*mid*)
if *k* = *e*.key() **then**
return *e*
else if *k* < *e*.key() **then**
return BinarySearch(*L*,*k*, *low*,*mid*−1)
else
return BinarySearch(*L*,*k*,*mid*+1,*high*)

Q.12 Write a short note on Sequential (Linear) search.

Ans.

Sequential search:

- When data items are stored in a collection such as a list, we say that they have a linear or sequential relationship.
- Each data item is stored in a position relative to the others. In Python lists, these relative positions are the index values of the individual items.
- Since these index values are ordered, it is possible for us to visit them in sequence. This process gives rise to our first searching technique, the sequential search.

- Figure 1 shows how this search works. Starting at the first item in the list, we simply move from item to item, following the underlying sequential ordering until we either find what we are looking for or run out of items.
- If we run out of items, we have discovered that the item we were searching for was not present.

Linear search is implemented using following steps...

Step 1: Read the search element from the user
Step 2: Compare, the search element with the first element in the list.
Step 3: If both are matching, then display "Given element found!!!" and terminate the function
Step 4: If both are not matching, then compare search element with the next element in

	the list. Step 5: Repeat steps 3 and 4 until the search element is compared with the last element in the list. Step 6: If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.