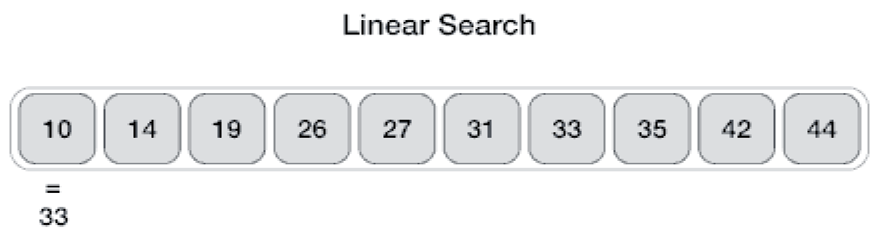
Chapter II
Arrays And Stack

Searching :-

- Searching is a process of locating a particular element present in a given set of elements.
- The element may be a record, a table, or a file.
- The search is said to be successful if the element is found i.e if the element is present in set, otherwise it is unsuccessful.
- There are two simple approaches of searching:-
  1.Linear Search
  2.Binary Search

1.Linear Search :-

- In Linear Search the list is searched sequentially and the position is returned if the key element to be searched is available in the list, otherwise -1 is returned.
- The search in Linear Search starts at the beginning of an array and move to the end, testing for a match at each item.
- All the elements preceding the search element are traversed before the search element is traversed. i.e. if the element to be searched is in position 10, all elements form 1-9 are checked before 10.



- Here we are searching for the element 33 .To search 33 the item 33 is compared with the element at A[0] then A[1] and so on .Until we find the key value or reach to the end of array.
- When the item is found it displays the location of an item else displays item not found

Algorithm :-

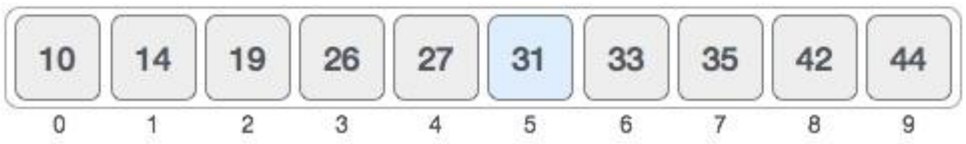
```
Algorithm
Linear Search ( Array A, Value x)

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:-

- The binary search technique requires the list to be sorted in an ascending order.
- Binary search is a fast search algorithm with run-time complexity of O(log n).
- Binary search a particular item by comparing the middle most item of the collection.
- If match occurs then index of item is returned.
- If middle item is greater than item then item is searched in sub-array to the right of the middle item otherwise item is search in sub-array to the left of the middle item..
- This process continues on sub-array as well until the size of sub-array reduces to zero.

How binary search works?

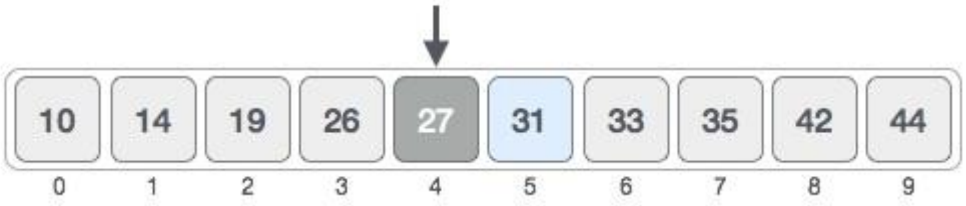
Consider for example **t**he below given is our sorted array and assume that we need to search location of value 31 using binary search.



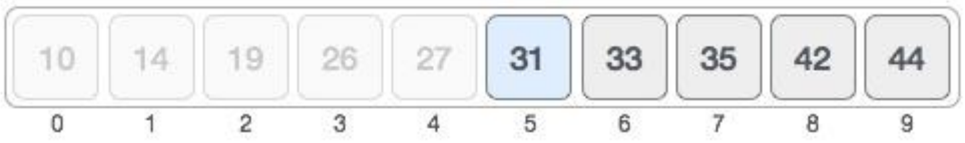
First, we shall determine the half of the array by using this formula –

mid = (low+high)/2

Here it is, (0+9) / 2 = 4 (integer value of 4.5). So 4 is the mid of array.

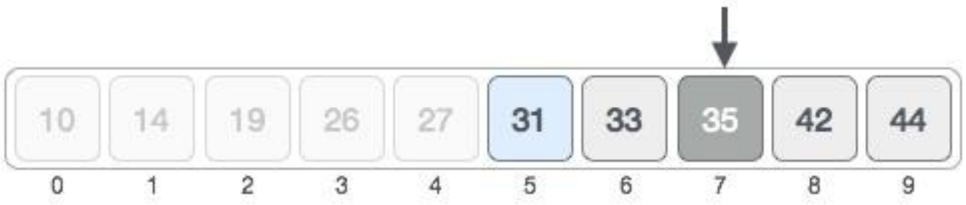


Now we compare the value stored at location 4, with the value being searched i.e. 31. We find that value at location 4 is 27, which is not a match. Because value is greater than 27 and we have a sorted array so we also know that target value must be in upper portion of the array.



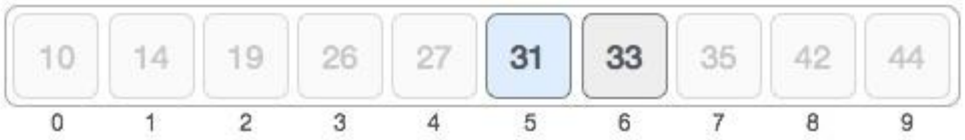
We change our low to mid + 1 and find the new mid value again.
low = mid + 1
mid = mid = (low + high)/2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

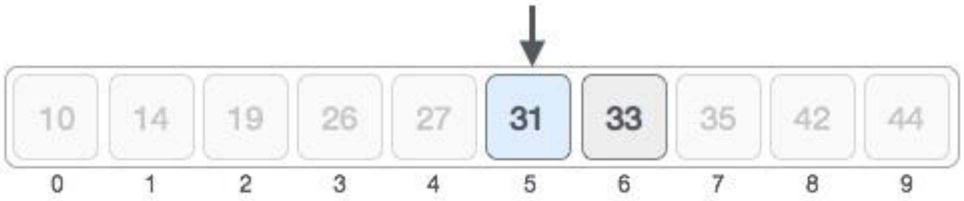


The value stored at location 7 is not a match, rather it is less that what we are looking for. So the value must be in lower part from this location.

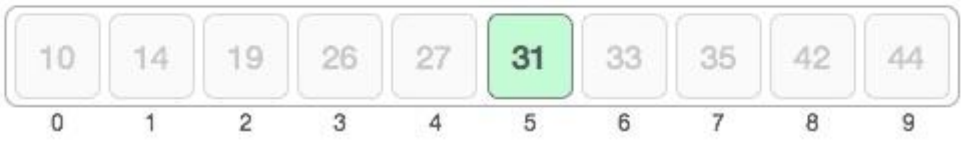
high = mid -1 ie high= 6 now 31<33



So we calculate the mid again. This time it is 5.



We compare the value stored ad location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Algorithm:-

Binary_Search(a, low, high, val,pos) // 'a' is the given array, 'low is the index of the first array element, ‘high’ is the index of the last array element, 'val' is the value to s earch ,pos is the position of number

```
Step 1: set  low=0,  high=n, pos = - 1
Step 2: repeat steps 3 and 4 while low <=high
Step 3: set mid = (low + high)/2
Step 4: if a[mid] = val
             set pos = mid
               print Number is found at   pos
               go to step 6
         else if a[mid] > val
                  set high = mid - 1
          else
                  set low = mid + 1
         [end of if]
         [end of loop]
Step 5: if pos = -1
              print "value is not present in the array"
          [end of if]
Step 6: exit
```

Sorting:-

- Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order.
- **Sorting** arranges data in a sequence which makes searching easier.
- Every record which is going to be sorted will contain one key. Based on the key the record will be sorted. For example, suppose we have a record of students, every such record will have the following data:

  Roll No.
  Name
  Age
  Class

Here Student roll no. can be taken as key for sorting the records in ascending or descending order.

- Types of sorting :-

1. Selection Sort
2. Bubble Sort
3. Insertion Sort
4. Merge Sort
5. Radix Sort

1.Selection sort :-

- 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.
- Consider ex. if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, leave the elements as it is.
- Then, again first element and third element are compared and swapped if necessary. This process goes on until first and last element of an array is compared. This completes the first step of selection sort.

- If there are *n* elements to be sorted then, the process mentioned above should be repeated *n-1* times to get required result.

Algorithm :-

```
SELECTION-SORT(A)
        n ← length[A]
        for j ← 1 to n - 1
                do smallest ← j
                    for i ← j + 1 to n
                          do if A[i] < A[smallest]
                                  then smallest ← i
                    exchange A[j] ↔ A[smallest]
```

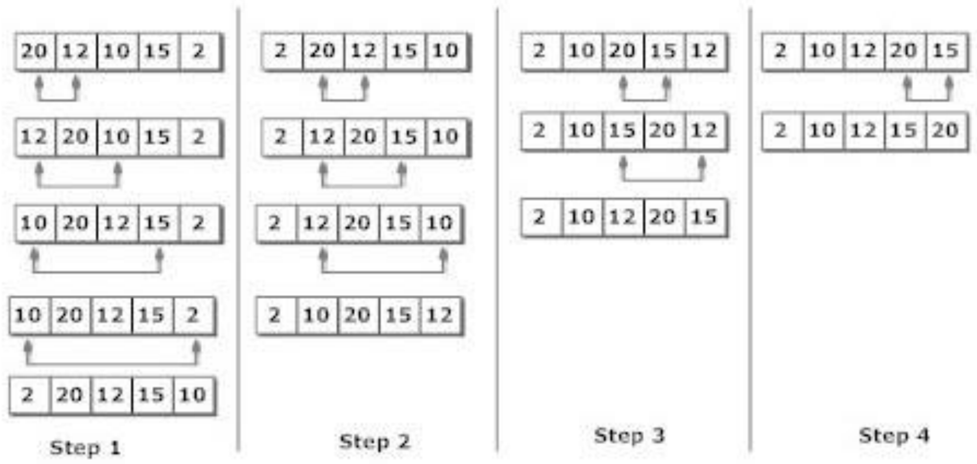


Figure: Selection Sort

➢ Complexity Analysis of Selection Sorting

Worst Case Time Complexity : $O(n^2)$
Best Case Time Complexity : $O(n^2)$
Average Time Complexity : $O(n^2)$
Space Complexity : O(1)

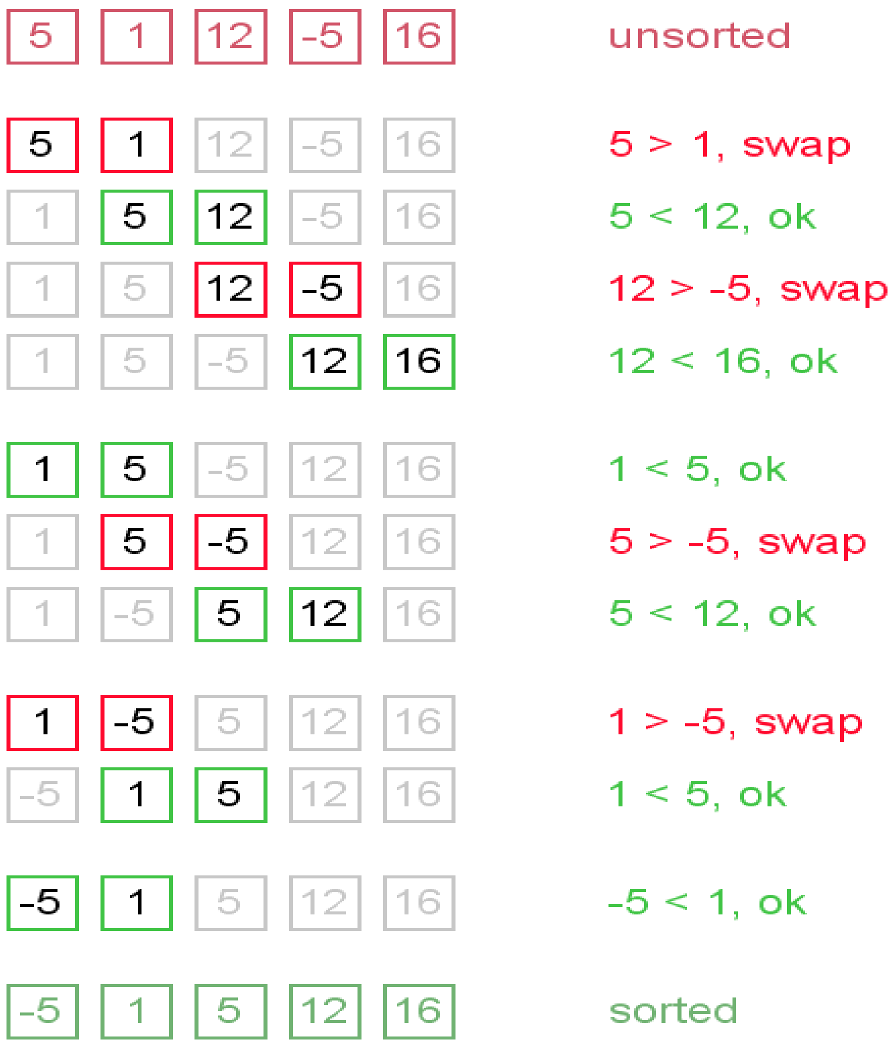
Bubble Sort:-

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.

Example. Sort {5, 1, 12, -5, 16} using bubble sort.



Average and worst case complexity of bubble sort is O(n^2). Also, it makes O(n^2) swaps in the worst case.

Algorithm:-

1. Start
2. Accept Array
3. Compare elements as

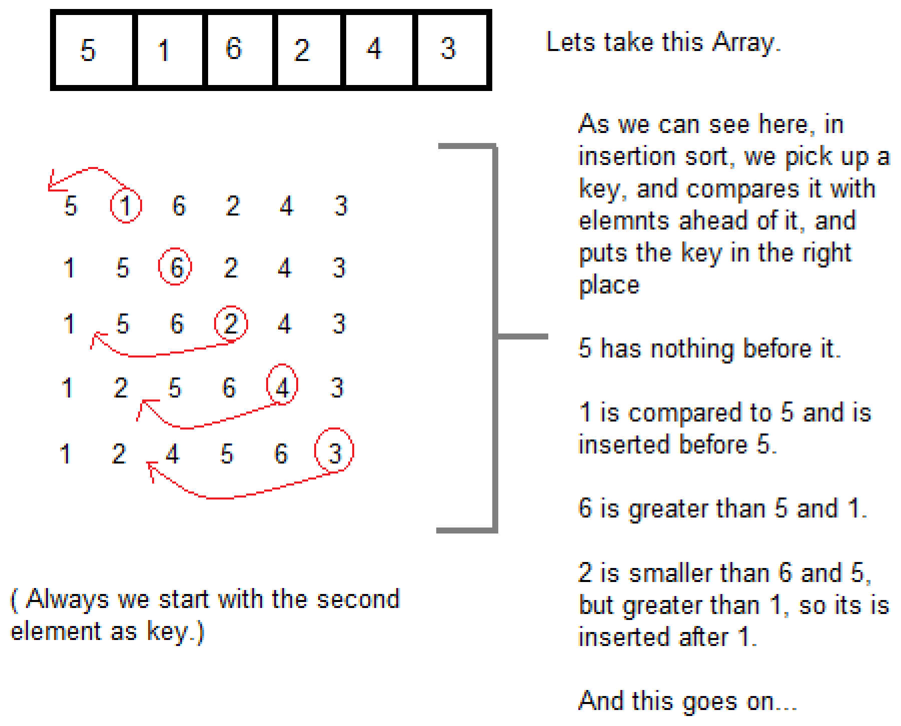
```
for(i=0;i<n-1;i++)
   {
     for(j=0;j<(n-(i+1));j++)
           {
                 If(a[j]>a[j+1])
                     {
                          temp=a[j];
                          a[j]=a[j+1];
                          a[j+1]=temp;
                     }
           }
```

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

- How Insertion Sorting Works



5 1 6 2 4 3
Lets take this Array.
5 1 6 2 4 3
1 5 6 2 4 3
1 5 6 2 4 3
1 2 5 6 4 3
1 2 4 5 6 3
As we can see here, in insertion sort, we pick up a key, and compares it with elemnts ahead of it, and puts the key in the right place
5 has nothing before it.
1 is compared to 5 and is inserted before 5.
6 is greater than 5 and 1.
2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.
And this goes on...
(Always we start with the second element as key.)

- ***Sorting using Insertion Sort Algorithm :-***

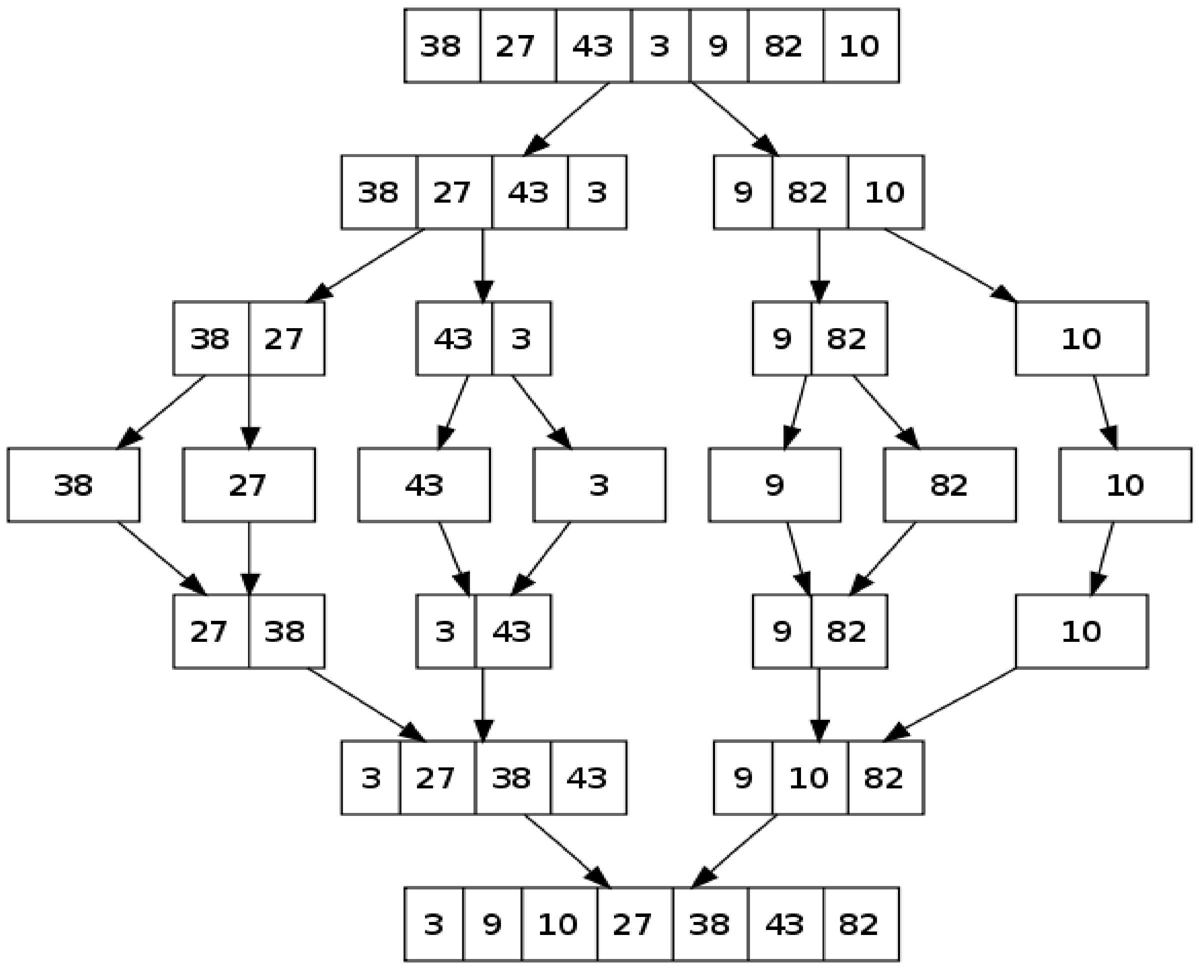
```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, key;
for(i=1; i<6; i++)
{
  key = a[i];
  j = i-1;
  while(j>=0 && key < a[j])
  {
    a[j+1] = a[j];
    j--;
  }
  a[j+1] = key;
}
```

- Complexity Analysis of Insertion Sorting

Worst Case Time Complexity : O(n^2)
Best Case Time Complexity : O(n)
Average Time Complexity : O(n^2)
Space Complexity : O(1)

Merge Sort :-

- MergeSort is a Divide and Conquer algorithm.
- In merge sort the unsorted list is divided into N sublists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sublists, to produce new sorted sublists, 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.



- Complexity Analysis of Merge Sort

Worst Case Time Complexity : O(n log n)
Best Case Time Complexity : O(n log n)
Average Time Complexity : O(n log n)
Space Complexity : O(n)

Radix Sort:-

Radix sort is one of the linear sorting algorithms for integers. It functions by sorting the input numbers on each digit, for each of the digits in the numbers. However, the process adopted by this sort method is somewhat counterintuitive, in the sense that the numbers are sorted on the least-significant digit first, followed by the second-least significant digit and so on till the most significant digit.

Consider the following 9 numbers:

493 812 715 710 195 437 582 340 385

We should start sorting by comparing and ordering the **one's** digits:

Digit	Sublist
0	340 710
1	
2	812 582
3	493
4	
5	715 195 385
6	
7	437
8	
9	

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again:

340   710   812   582   493   715   195   385   437

The **order** in which we divide and reassemble the list is **extremely important**, as this is one of the foundations of this algorithm.

Now, the sublists are created again, this time based on the **ten's** digit:

Digit	Sublist
0	
1	710 812 715
2	
3	437
4	340
5	
6	
7	
8	582 385
9	493 195

Now the sublists are gathered in order from 0 to 9:

710   812   715   437   340   582   385   493   195

Finally, the sublists are created according to the **hundred's** digit:

Digit	Sublist
0	
1	195
2	
3	340 385
4	437 493
5	582
6	
7	710 715
8	812
9	

At last, the list is gathered up again:

195   340   385   437   493   582   710   715   812

And now we have a fully sorted array! Radix Sort is very simple, and a computer can do it fast. When it is programmed properly, Radix Sort is in fact **one of the fastest sorting algorithms** for numbers or strings of letters.

Algorithm:-

```
RadixSort(arr)
1. Max = largest element in the given array
2. d = number of digits in the largest element (or, max)
3. Now, create d buckets of size 0 - 9
4. for i -> 0 to d
5. sort the array elements using counting sort (or any stable sort) according to the digits at  the ith place
```