Practical – 10

Sorting techniques

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.
Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order. For example, consider an array A = {A1, A2, A3, A4, ?? An }, the array is called to be in ascending order if element of A are arranged like A1 > A2 > A3 > A4 > A5 > ? > An .

Consider an array;

int A[10] = { 5, 4, 10, 2, 30, 45, 34, 14, 18, 9)

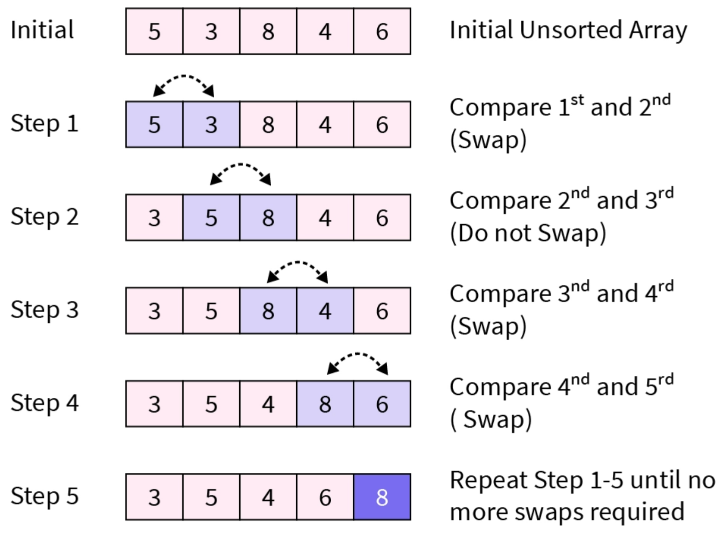
The Array sorted in ascending order will be given as;

A[] = { 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 }

There are many techniques by which sorting can be performed. In this section of the tutorial, we will discuss each method in detail.

Bubble Sort

Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.



Initial
5 3 8 4 6
Initial Unsorted Array
Step 1
5 3 8 4 6
Compare 1st and 2nd (Swap)
Step 2
3 5 8 4 6
Compare 2nd and 3rd (Do not Swap)
Step 3
3 5 8 4 6
Compare 3nd and 4rd (Swap)
Step 4
3 5 4 8 6
Compare 4nd and 5rd (Swap)
Step 5
3 5 4 6 8
Repeat Step 1-5 until no more swaps required

Bubble sort alogrithm

```
begin BubbleSort(arr)
 for all array elements
 if arr[i] > arr[i+1]
 swap(arr[i], arr[i+1])
 end if
 end for
 return arr
end BubbleSort
```

```
procedure start:
bubble_sort(arr)
   size = arr.length

   for i=0 to size-1:
      for j=0 to size-i-1:
         if arr[j] > arr[j+1] then:  //checking
adjacent elements
            swap(arr[j], arr[j+1])  //swap if
left_elem > right_elem
return arr
procedure end
```

Insertion sort

Insertion Sort is one of the simple sorting algorithms used to sort the array. Say we have two boxes, box1 and box2. box1 contains items in random order and box2 is empty. Now we pick each item from box1 and place this item in box2 such that the items in box2 are in sorted order after each addition.

This can be done by assuming the first element inserted is in its right position and from the next element, its right position is searched and the element at that position as well as elements placed after it is moved one step forward so that the current element is placed at the right position and items are in sorted order.

The same procedure is applied to the array as well. Instead of using two arrays, the given array is divided into two parts i.e. the sorted one and the unsorted one. Values from the unsorted section are picked one by one and placed in the sorted part at their right position.

Characteristics of Insertion Sort

- It is one of the simplest algorithms to sort an array. It is easy to understand and implement.
- It is efficient only for a small number of data values.
- Insertion sort is appropriate if the data is partially sorted i.e. it is adaptive in nature.
- It follows an incremental approach i.e. the number of elements in the sorted section of the array is increased one by one.
- It is a stable and in-place sorting algorithm.

Algorithm

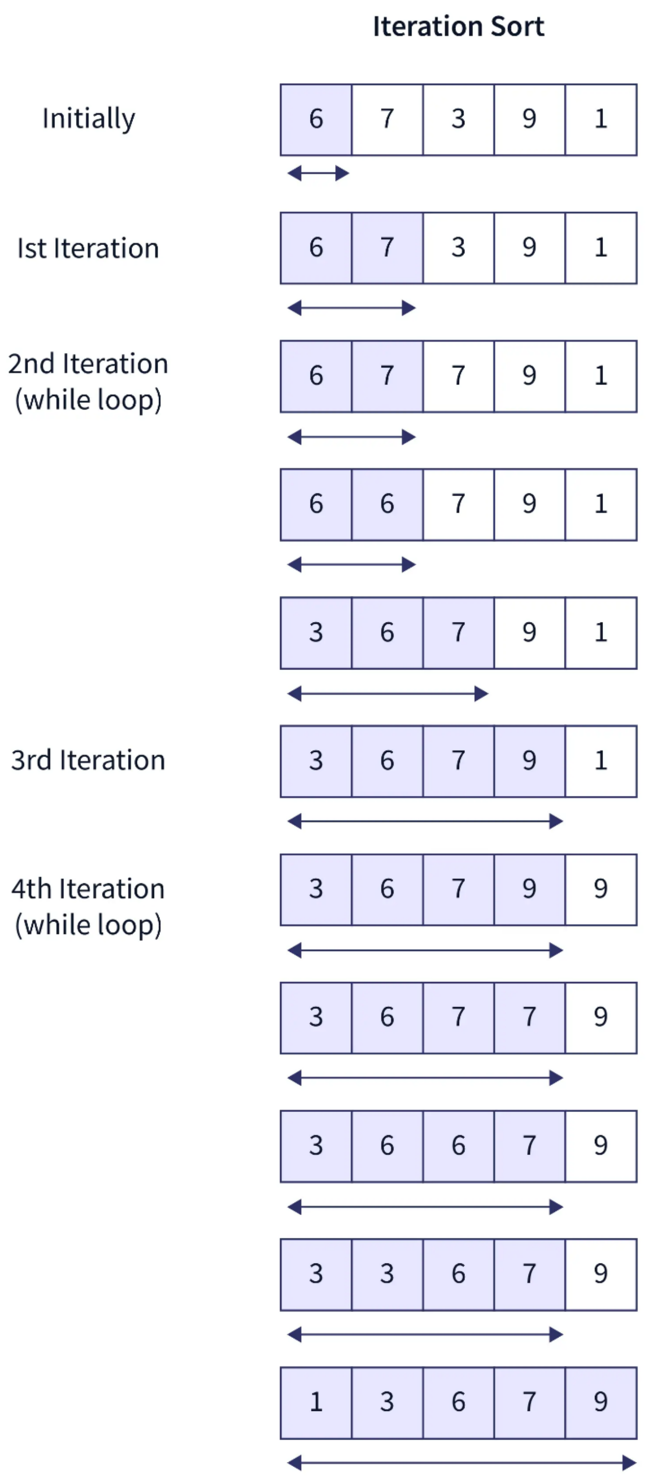
The steps to implement the insertion sort algorithm are described below:

1. We assume that the array is empty and on inserting one element into it will not affect the sorting order of the array. Thus, the first element is already sorted.
2. We run a loop from the second element to the last element in the array. In each iteration, the current element is compared with the previous elements, and its correct position is searched.
3. If the current element is greater than or equal to the previous element then we simply move toward the next iteration. If the current element is smaller than previous elements all the elements greater than the current are shifted one position to the right and the current element is placed at its correct position.
4. Step 3 is repeated for each element in the array.

OR

- **Step 1 -** If the element is the first element, assume that it is already sorted. Return 1.
- **Step2 -** Pick the next element, and store it separately in a **key.**

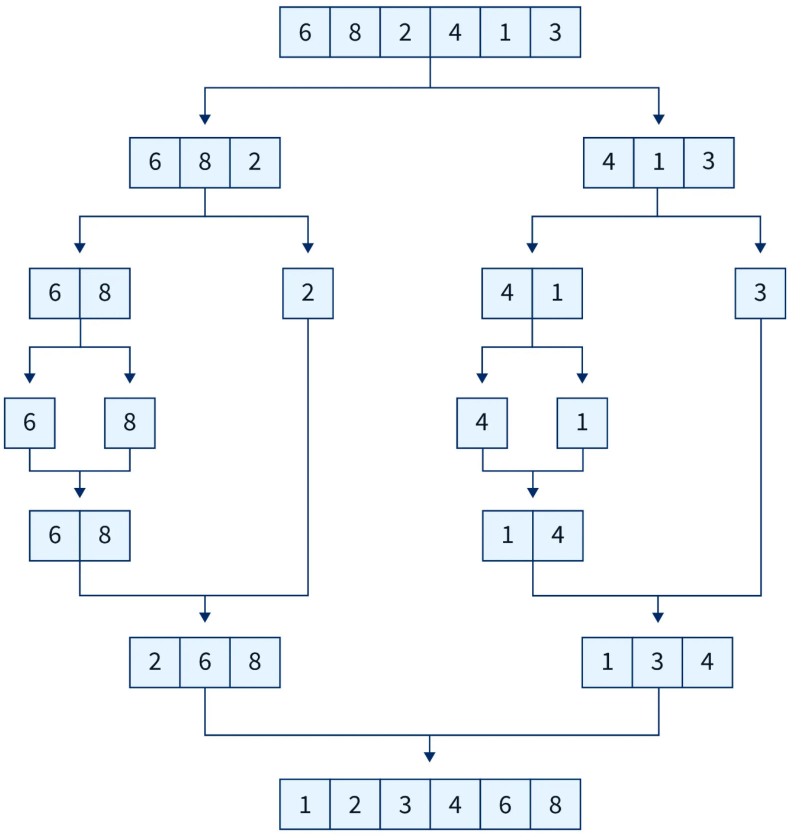
Step3 - Now, compare the **key** with all elements in the sorted array. **Step 4 -** If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right. **Step 5 -** Insert the value.
Step 6 - Repeat until the array is sorted.



```
arr = {6, 7, 3, 9, 1}
n = 5
void insertionSort(){
    cur = 0;
    j = 0;

    for(int i = 1; i < n; i++){
        cur = arr[i];
        j = i-1;

        // finding right position for the current and shifting
one position to the right
        while(j >= 0 && cur<arr[j]){
            arr[j+1] = arr[j];
            j--;
        }

        // placing current to its correct position
        arr[j+1] = cur;
    }
}
```

Merge Sort

Merge sort is a sorting technique based on divide and conquer technique. In Merge sort, we divide the array recursively in two halves, until each sub-array contains a single element, and then we merge the sub-array in a way that it results into a sorted array. merge() function merges two sorted sub-arrays into one, wherein it assumes that array[l .. n] and arr[n+1 .. r] are sorted.



```
void merge_sort(int arr[], int left, int right)
{
    if (left < right) {

        // finding the mid value of the array.
        int mid = l + (right - left) / 2;

        // Calling the merge sort for the first half
        merge_sort(arr, left, mid);

        // Calling the merge sort for the second half
        merge_sort(arr, mid + 1, right);

        // Calling the merge function
        merge(arr, left, mid, right);
    }
}

void merge(int ar[], int left, int mid, int right)
{
    int i, j, k;
    int s1 = mid - left + 1;
    int s2 = right - mid;
    /* It will create two temporary arrays */
    int left_arr[s1], right_arr[s2];
    /* It will copy data from arr to temporary arrays */
    for (i = 0; i < s1; i++)
        left_arr[i] = arr[left + i];

    for (j = 0; j < s2; j++)
        right_arr[j] = arr[mid + 1 + j];

    i = 0, j = 0;
    k = left;
    while (i < s1 && j < s2) {
        if (left_arr[i] <= right_arr[j]) {
            ar[k] = left_arr[i];
            i++;
        }
        else {
            ar[k] = right_arr[j];
            j++;
        }
        k++;
    }
    /* Copying the items of left_arr[] that have been left */
    while (i < s1) {
        arr[k] = left_arr[i];
        i++;
        k++;
    }
    /* Copying the items of right_arr[] that have been left */
    while (j < s2) {
        arr[k] = right_arr[j];
        j++;
        k++;
    }
}
```

Quick Sort

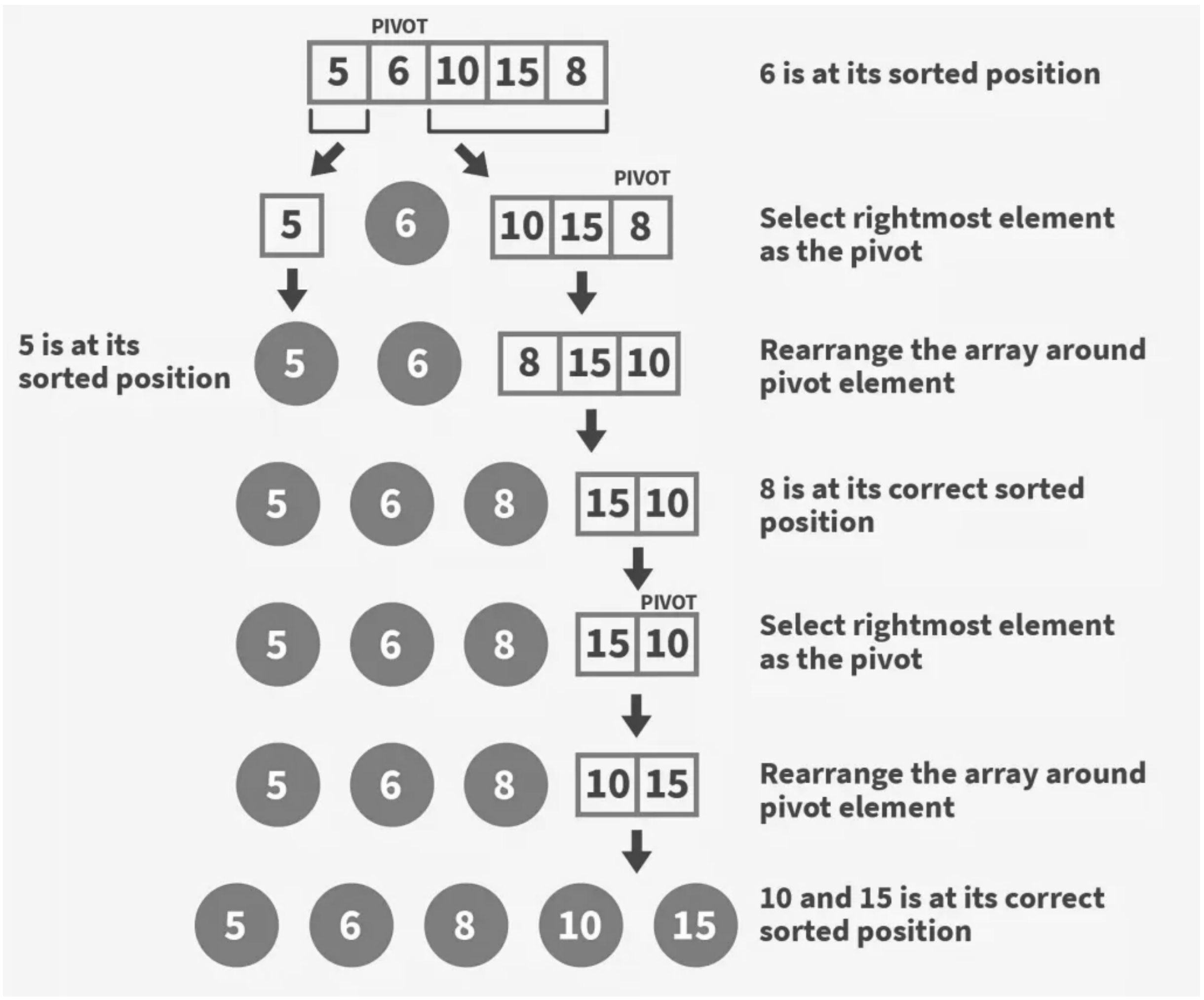
Quick Sort is a sorting algorithm that uses Divide and Conquer approach. The name itself portrays QuickSort which defines the Quick Sort algorithm is faster than all other algorithms. Quick Sort algorithm picks a pivot element and performs sorting based on the pivot element. In the Divide and Conquer approach we break down the array into subarrays and conquer them. The Quick Sort algorithm can be implemented in both iterative and Recursion modes. Quick Sort is also known as partition-exchange sort. The time complexity of the quick sort algorithm is O(n logn).

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting sub arrays.

```
FLAG □ true
IF LB < UB
Then
 I □ LB
 J □ UB + 1
 KEY □ K[LB]
 Repeat While FLAG = true
 I □ I+1
 Repeat While K[I] < KEY
 I □ I + 1
 J □ J – 1
 Repeat While K[J] > KEY
 J □ J – 1
 IF I<J
 Then K[I] □---□ K[J]
 Else FLAG □ FALSE

 K[LB] □---□ K[J]
```

OR

```
partition(array, low, high)
{
          pivot = arr[high]; // select last element of array as
pivot
      i = (low - 1);

                 // running a loop till the last element of the
array
     for (j = low; j <= high - 1; j++) {
          // comparing if jth index is less than pivot
           if (arr[j] <= pivot) {

                // increasing the value of i to perform swap
operation
           i++;

           // swapping jth index with ith index
               swap(&arr[i], &arr[j]);
          }
     }
     swap(&arr[i + 1], &arr[high]);
     return (i + 1);
}



quicksort(array, low, high)
{
     if (low < high)
           {
                       // this function returns where the array
is partitioned
          pi = partition(array, low, high);

                       // recursively calling left sub array
               quicksort(array, low, pi - 1);

                       // recursively calling right sub array
          quicksort(array, pi + 1, high);
     }
}
```


PIVOT
5 6 10 15 8
6 is at its sorted position
5
6
PIVOT
10 15 8
Select rightmost element as the pivot
5 is at its sorted position
5
6
8 15 10
Rearrange the array around pivot element
5 6 8
15 10
8 is at its correct sorted position
PIVOT
5 6 8
15 10
Select rightmost element as the pivot
5 6 8
10 15
Rearrange the array around pivot element
5 6 8 10 15
10 and 15 is at its correct sorted position

Exercise

1. Write a Program to collect an unsorted array from the user. Implement sorting of the array using following techniques.

- bubble sort
- quick sort.
- insertion sort
- Merge sort