# LAB NO. 5

# Sorting Algorithms: Quick, Merge & insertions Sort

## OBJECTIVE:

- To understand & implement the working of sorting algorithms using arrays in C++.

## SORTING:

It is often necessary to arrange the elements in an array in numerical order from highest to lowest values i.e.; from ascending to descending and vice versa. If an array contains string values or alphabetical order then arrays are need to be sorted. The process of sorting an array requires the exchanging of values. While this seems to be a simple process, a computer must be careful that no values are lost during this exchange.

## QUICK SORT:

Quicksort is a fast-sorting algorithm, which is used not only for educational purposes, but widely applied in practice. On the average, it has O(n log n) complexity, making quicksort suitable for sorting big data volumes. The idea of the algorithm is quite simple and once you realize it, you can write quicksort as fast as bubble sort.

### Quick Sort Steps:

The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:

1. Choose a pivot value. We take the value of the middle element as pivot value, but it can be any value, which is in range of sorted values, even if it doesn't present in the array.
2. Partition. Rearrange elements in such a way, that all elements which are lesser than the pivot go to the left part of the array and all elements greater than the pivot, go to the right part of the array. Values equal to the pivot can stay in any part of the array. Notice, that array may be divided in non-equal parts.
3. Sort both parts. Apply quicksort algorithm recursively to the left and the right parts.

### Partition algorithm in detail:

There are two indices i and j and at the very beginning of the partition algorithm i points to the first element in the array and j points to the last one. Then algorithm moves i forward, until an element with value greater or equal to the pivot is found. Index j is moved backward, until an element with value lesser

or equal to the pivot is found. If i ≤ j then they are swapped and i steps to the next position (i + 1), j steps to the previous one (j - 1). Algorithm stops, when ibecomes greater than j.

After partition, all values before i-th element are less or equal than the pivot and all values after j-th element are greater or equal to the pivot.

## Merge Sort

Merge sort is based on the divide-and-conquer paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with sub problems, we state each sub problem as sorting a sub arrayA[p .. r]. Initially, p = 1 and r = n, but these values change as we recurse through sub problems.

## Merge Sort Steps:

To sort A[p .. r]:

## 1. Divide Step

If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split A[p .. r] into two sub arrays A[p .. q] and A[q + 1 .. r], each containing about half of the elements of A[p .. r]. That is, q is the halfway point of A[p .. r].

## 2. Conquer Step

Conquer by recursively sorting the two sub arraysA[p .. q] and A[q + 1 .. r].

## 3. Combine Step

Combine the elements back in A[p .. r] by merging the two sorted sub arrays A[p .. q] and A[q + 1 .. r] into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

Note that the recursion bottoms out when the sub array has just one element, so that it is trivially sorted.

**Example: Writing a function to sort array elements using quick sort:**

```
#include<iostream>
Void quickSort(intarr[], int left, int right)
{
inti = left, j = right; inttmp; intpivot
= arr[(left + right) / 2];
/* partition */ while
(i<= j) {
while (arr[i] <pivot) i++;
while (arr[j] >pivot) j--;
if (i<= j)
{
```

```
tmp = arr[i];
arr[i] = arr[j];
arr[j] = tmp; i++;
j--; }
};
/* recursion */ if
(left<j)
quickSort(arr, left, j); if
(i<right) quickSort(arr,
i, right);
}
```

**Example: Writing a function to sort array elements using merge sort**

```
#include <iostream>
  Void mergeSort(int numbers[], int temp[], intarray_size)
{  m_sort(numbers, temp, 0, array_size - 1);
}  void m_sort(int numbers[], int temp[], int left, int right)
{  int mid; if (right > left)
 {  mid = (right + left) / 2;
m_sort(numbers, temp, left, mid);
```

```
m_sort(numbers, temp, mid+1, right);
merge(numbers, temp, left, mid+1, right);  } }
// m - size of A
// n - size of B
// size of C array must be equal or greater than
// m + n void merge(int m, int n, int A[], int B[], int
C[]) { inti, j, k; i = 0; j = 0; k = 0; while (i< m && j
< n)
{ if (A[i] <=
B[j])
{
C[k] = A[i]; i++;
} else {
C[k] = B[j];
j++; } k++;
}
if (i< m)
{ for (int p = i; p < m;
p++) {
57
C[k] = A[p]; k++;
}
} else { for (int p = j; p < n;
p++)
{
C[k] = B[p]; k++;
}
}
}
```

## Insertion Sort

```
#include <iostream> using
namespace std;  #define MAX 7
 int intArray[MAX] = {4,6,3,2,1,9,7};   void
printline(int count) {    int i;

   for(i = 0;i < count-1;i++) {       cout<<"=";
   }       cout<<"=\n";
}  void display() {
int i;    cout<<"[
";

   // navigate through all items     for(i = 0;i <
MAX;i++) {       cout<<intArray[i]<<" ";
   }       cout<<"]\n";
}  void
insertionSort() {     int
valueToInsert;    int
holePosition;    int
i;
   // loop through all numbers     for(i = 1; i <
MAX; i++) {

      // select a value to be inserted.
      valueToInsert = intArray[i];

      // select the hole position where number is to be inserted
```

```
      holePosition = i;

      // check if previous no. is larger than value to be inserted
while (holePosition > 0 && intArray[holePosition-1] > valueToInsert) {
intArray[holePosition] = intArray[holePosition-1];          holePosition--;
cout<<" item moved :
"<<intArray[holePosition]<<"\n";
      }        if(holePosition != i) {          cout<<"item inserted
:"<<valueToInsert<<" , at position
:"<<holePosition<<"\n";
         // insert the number at hole position
intArray[holePosition] = valueToInsert;
      }        cout<<"Iteration
"<<i<<"#: ";       display();


   }
}  int main() {
cout<<"Input Array: ";
display();    printline(50);
insertionSort();
cout<<"Output Array: ";
display();    printline(50);
return 0;  }
```

## Insertion Sort:

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 inserted 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.

## How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.


It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.


Insertion sort moves ahead and compares 33 with 27.


And finds that 33 is not in the correct position.


It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.


By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.


These values are not in a sorted order.


So we swap them.


However, swapping makes 27 and 10 unsorted.


Hence, we swap them too.


Again we find 14 and 10 in an unsorted order.


We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.


This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

## 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
```

## Lab Tasks:

**Task 1:** Writing a function to sort array elements using insertion sort**.**

[1,8,4,6,0,3,5,2,7,9]

**Task 2**: Write a code to sort the following arrays using Quick sort method.

[10, 34, 45, 33, 23, 47, 31, 23, 45, 69, 2, 56,7,67, 88, 42]

**Task 3**: Write a code to sort the following arrays using Merge sort method**.**

[10, 34, 33, 22, 77, 98, 2, 56, 7, 55, 56, 67, 88, 42]

**Task 4:** Write a code to sort the following arrays using Insertion sort method**.**

[12, 31, 41, 37, 49, , 45, 69, 2, 56,76,67, 98, 52]