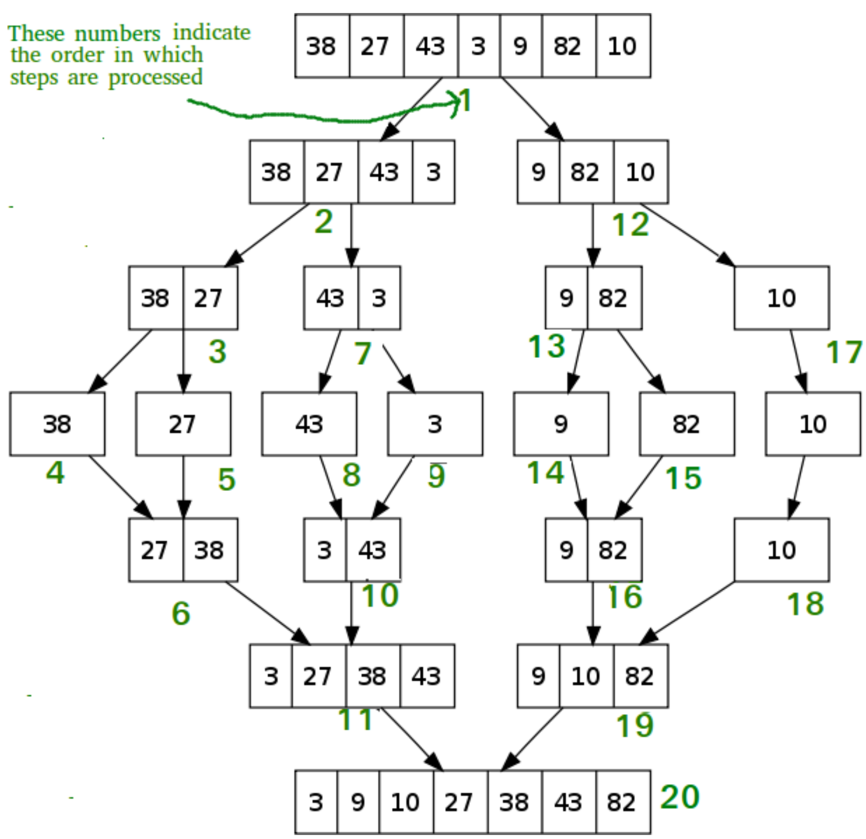
PROGRAM 3: IMPLEMENTATION OF MERGE SORT AND QUICK SORT

Aim: To write a program in C++ for implementation of merge sort and quick sort.

Description *Merge Sort*:

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.



Algorithm *Merge Sort*:

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.

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:

   middle m = l+ (r-l)/2

2. Call mergeSort for first half:

   Call mergeSort(arr, l, m)

3. Call mergeSort for second half:

   Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

   Call merge(arr, l, m, r)

Program *Merge Sort*:

```
#include<iostream>
using namespace std;
void display(int *array, int size) {
   for(int i = 0; i<size; i++)
      cout << array[i] << " ";
   cout << endl;
}
void merge(int *array, int l, int m, int r) {
```

```
int i, j, k, nl, nr;
//size of left and right sub-arrays
nl = m-l+1; nr = r-m;
int larr[nl], rarr[nr];
//fill left and right sub-arrays
for(i = 0; i<nl; i++)
   larr[i] = array[l+i];
for(j = 0; j<nr; j++)
   rarr[j] = array[m+1+j];
i = 0; j = 0; k = l;
//marge temp arrays to real array
while(i < nl && j<nr) {
   if(larr[i] <= rarr[j]) {
      array[k] = larr[i];
      i++;
   }else{
      array[k] = rarr[j];
      j++;
   }
   k++;
}
while(i<nl) {       //extra element in left array
   array[k] = larr[i];
   i++; k++;
}
while(j<nr) {     //extra element in right array
```

```
      array[k] = rarr[j];
      j++; k++;
   }
}
void mergeSort(int *array, int l, int r) {
   int m;
   if(l < r) {
      int m = l+(r-l)/2;
      // Sort first and second arrays
      mergeSort(array, l, m);
      mergeSort(array, m+1, r);
      merge(array, l, m, r);
   }
}
int main() {
   int n;
   cout << "Enter the number of elements: ";
   cin >> n;
   int arr[n];      //create an array with given number of elements
   cout << "Enter elements:" << endl;
   for(int i = 0; i<n; i++) {
      cin >> arr[i];
   }
   cout << "Array before Sorting: ";
   display(arr, n);
   mergeSort(arr, 0, n-1);     //(n-1) for last index
```

```
  cout << "Array after Sorting: ";
  display(arr, n);
}
```

Output

Enter the number of elements: 6

Enter elements:

14 20 78 98 20 45

Array before Sorting: 14 20 78 98 20 45

Array after Sorting: 14 20 20 45 78

Description *Quick Sort*:

Quick sort is a fast sorting algorithm used to sort a list of elements. Quick sort algorithm is invented by **C. A. R. Hoare**. The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. That means it use **divide and conquer** strategy. In quick sort, the partition of the list is performed based on the element called **pivot**.

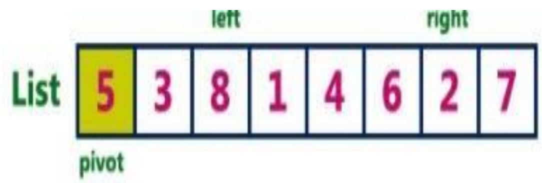
Here pivot element is one of the elements in the list. The list is divided into two partitions such that **"all elements to the left ofpivot are smaller than the pivot and all elements to the right of pivot are greater than or equal to the pivot"**.

Consider the following unsorted list of elements...

Define pivot, left & right. Set pivot = 0, left = 1 & right = 7. Here '7' indicates 'size-1'.

Compare List[left] with List[pivot]. If **List[left]** is greater than **List[pivot]** then stop left otherwise move left to the next.
Compare List[right] with List[pivot]. If List[right] is smaller than List[pivot] then stop right otherwise move right to the previous.
Repeat the same until **left>=right**.
If both left & right are stoped but left<right then swap List[left] with List[right] and countinue the process.
If left>=right then swap List[pivot] with List[right].

Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 8.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 2.



Here left & right both are stoped and left is not greater than right so we need to swap List[left] and List[right]

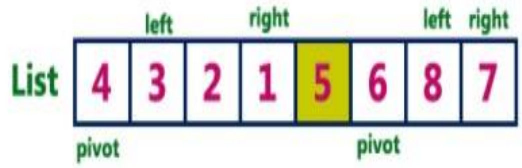
Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 6.

Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 4.

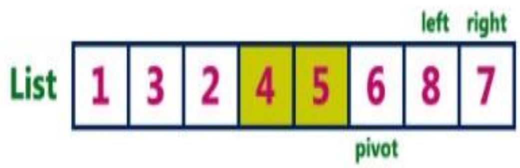
Here left & right both are stoped and left is greater than right so we need to swap List[pivot] and List[right]

Here we can observe that all the numbers to the left side of 5 are smaller and right side are greater. That means 5 is placed in its correct position.

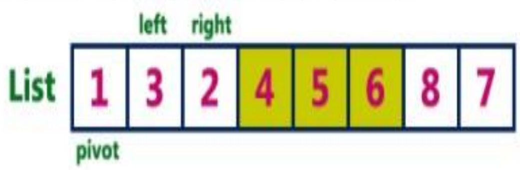
Repeat the same process on the left sublist and right sublist to the number 5.



In the left sublist as there are no smaller number than the pivot left will keep on moving to the next and stops at last number. As the List[right] is smaller, right stops at same position. Now left and right both are equal so we swap pivot with right.



In the right sublist left is grester than the pivot, left will stop at same position.
As the List[right] is greater than List[pivot], right moves towords left and stops at pivot number position.
Now left > right so we swap pivot with right. (6 is swap by itself).



Repeat the same recursively on both left and right sublists until all the numbers are sorted.
The final sorted list will be as follows...

Algorithm *Quick Sort*:

Quick sort:

Step 1 − Make the right-most index value pivot

Step 2 − partition the array using pivot value

Step 3 − quicksort left partition recursively

Step 4 − quicksort right partition recursively

Quick sort partition:

Partitioning of the list is performed using following steps...

Step 1 - Consider the first element of the list as pivot (i.e., Element at First position in the list).

Step 2 - Define two variables i and j. Set i and j to first and last elements of the list respectively.

Step 3 - Increment i until list[i] > pivot then stop.

Step 4 - Decrement j until list[j] < pivot then stop.

Step 5 - If i < j then exchange list[i] and list[j].

Step 6 - Repeat steps 3,4 & 5 until i > j.

Step 7 - Exchange the pivot element with list[j] element.

Program *Quick Sort*:

```
// Quick sort in C++
#include  <iostream>
using namespace std;
// function to swap elements
void swap(int *a, int *b) {
  int t = *a;
```

```
  *a = *b;
  *b = t;
}
// function to print the array
void printArray(int array[], int size) {
  int i;
  for (i = 0; i < size; i++)
    cout << array[i] << " ";
  cout << endl;
}
// function to rearrange array (find the partition point)
int partition(int array[], int low, int high) {
  // select the rightmost element as pivot
  int pivot = array[high];
  // pointer for greater element
  int i = (low - 1);
  // traverse each element of the array
  // compare them with the pivot
  for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {
      // if element smaller than pivot is found
      // swap it with the greater element pointed by i
      i++;
     // swap element at i with element at j
      swap(&array[i], &array[j]);
    }
```

```
  }
    // swap pivot with the greater element at i
  swap(&array[i + 1], &array[high]);
   // return the partition point
  return (i + 1);
}
void quickSort(int array[], int low, int high) {
  if (low < high) {
     // find the pivot element such that
    // elements smaller than pivot are on left of pivot
    // elements greater than pivot are on righ of pivot
    int pi = partition(array, low, high);
    // recursive call on the left of pivot
    quickSort(array, low, pi - 1);
    // recursive call on the right of pivot
    quickSort(array, pi + 1, high);
  }
}

// Driver code
int main() {
  int data[] = {8, 7, 6, 1, 0, 9, 2};
  int n = sizeof(data) / sizeof(data[0]);
  cout << "Unsorted Array: \n";
  printArray(data, n);
```

```
  // perform quicksort on data
  quickSort(data, 0, n - 1);
  cout << "Sorted array in ascending order: \n";
  printArray(data, n);
}
```

Output:

Unsorted array

8 7 9 1 0 9 2

Sorted array in ascending order:

0 1 2 7 8 9 9

Result: Thus a programs are written and executed successfully in C++ for implementation of merge sort and quick sort.