AOA EXPERIMENT 2

Name: Mohammed Arhaan Ali
Div: S11
Roll No: 02

AIM: To study and implement Merge sort and Quick sort algorithms.

THEORY:

A) MERGE SORT:

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

Now, let's see the algorithm of merge sort.

In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.

Algorithm:

```
MERGE_SORT(arr, beg, end)

if beg < end

set mid = (beg + end)/2

MERGE_SORT(arr, beg, mid)

MERGE_SORT(arr, mid + 1, end)
```

```
MERGE (arr, beg, mid, end)

end of if

END MERGE_SORT
```

The important part of the merge sort is the **MERGE** function. This function performs the merging of two sorted sub-arrays that are **A[beg…mid]** and **A[mid+1…end]**, to build one sorted array **A[beg…end]**. So, the inputs of the **MERGE** function are **A[], beg, mid,** and **end**.

Merge sort complexity:

Now, let's see the time complexity of merge sort in best case, average case, and in worst case. We will also see the space complexity of the merge sort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n*logn)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n*logn)**.

- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n*logn)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **O(n*logn)**.

2. Space Complexity

Space Complexity	O(n)
Stable	YES

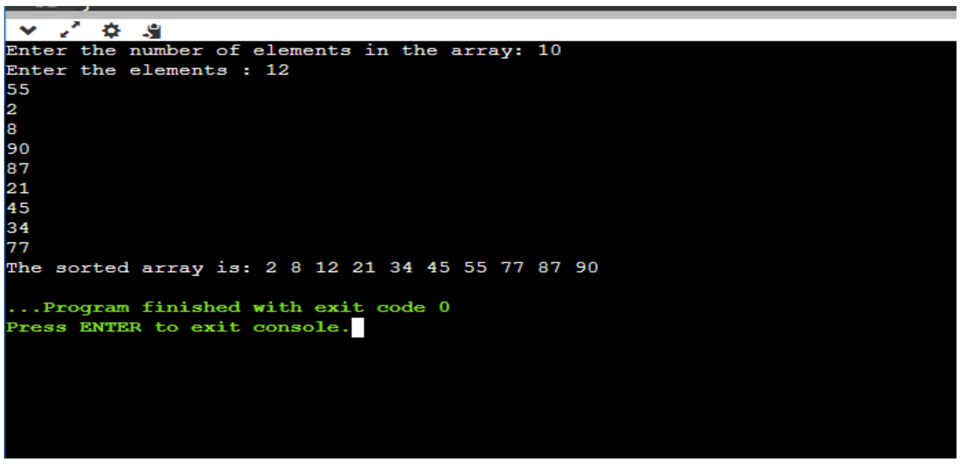
- The space complexity of merge sort is O(n). It is because, in merge sort, an extra variable is required for swapping.

Code:

```
#include<stdio.h>

void merge(int arr[],int p,int q,int r)
{
   int i,j,k;
   int n1=q-p+1;
   int n2=r-q;
   int L[100],M[100];
   for(i=0;i<n1;i++)

      L[i]=arr[p+i];
      for(j=0;j<n2;j++)
      M[j]=arr[q+1+j];
```

```
        i=0;
        j=0;
        k=p;

        while(i<n1 && j<n2)
        {
            if(L[i]<=M[j])
            {
                arr[k]=L[i];
                i++;
            }
            else{
                arr[k]=M[j];
                j++;
            }
            k++;
        }
        while(i<n1){
            arr[k]=L[i];
            i++;
            k++;

        }
        while(j<n2){
            arr[k]=M[j];
            j++;
            k++;

        }



}

void mergeSort(int arr[],int l,int r)
{
    if(l<r)
    {
        int m=l+(r-l)/2;
        mergeSort(arr,l,m);
        mergeSort(arr,m+1,r);
        merge(arr,l,m,r);

    }
}

void main(){
    int arr[100];
    int n,i;
    printf("Enter the number of elements in the array: ");
    scanf("%d",&n);
    printf("Enter the elements : ");
    for(i=0;i<n;i++)
    {
        scanf("%d",&arr[i]);

    }
    mergeSort(arr,0,n-1);
    printf("The sorted array is: ");
    for(i=0;i<n;i++)
    {
        printf("%d ",arr[i]);
    }
}
```

Output:



```
Enter the number of elements in the array: 10
Enter the elements : 12
55
2
8
90
87
21
45
34
77
The sorted array is: 2 8 12 21 34 45 55 77 87 90

...Program finished with exit code 0
Press ENTER to exit console.
```

B) QUICK SORT:

Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Algorithm:

```
QUICKSORT (array A, start, end)
{
 1 if (start < end)
 2 {
3 p = partition(A, start, end)
4 QUICKSORT (A, start, p - 1)
5 QUICKSORT (A, p + 1, end)
6 }
}
```

Partition Algorithm:

The partition algorithm rearranges the sub-arrays in a place.

```
PARTITION (array A, start, end)
{
 1 pivot ? A[end]
 2 i ? start-1
 3 for j ? start to end -1 {
 4 do if (A[j] < pivot) {
```

```
5 then i ? i + 1
6 swap A[i] with A[j]
7  }}
8 swap A[i+1] with A[end]
9 return i+1
}
```

Quicksort complexity

Now, let's see the time complexity of quicksort in best case, average case, and in worst case. We will also see the space complexity of quicksort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n^2)

- **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n*logn)**.

- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n*logn)**.
- **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **$O(n^2)$**.

Though the worst-case complexity of quicksort is more than other sorting algorithms such as **Merge sort** and **Heap sort**, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.

2. Space Complexity

Space Complexity	O(n*logn)
Stable	NO

- The space complexity of quicksort is O(n*logn).

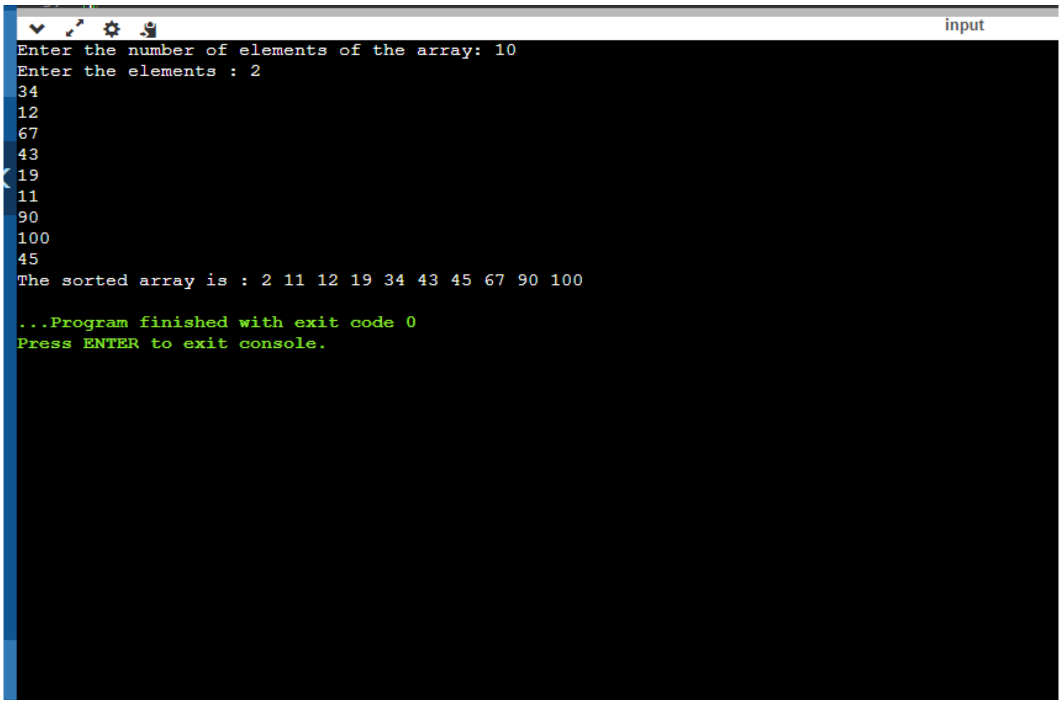
Code:

```
#include<stdio.h>

void swap(int *a,int *b){

   int pivot,i,j;

   int temp=*a;

   *a=*b;

   *b=temp;

}

int partition(int arr[],int l,int h){

   int pivot=arr[h];

   int i=l-1;

   int j;

   for(j=l;j<h;j++)

   {

      if(arr[j]<pivot)

      {
            i++;

            swap(&arr[i],&arr[j]);

        }

    }

    swap(&arr[i+1],&arr[h]);

    return(i+1);

}



void Quicksort(int arr[],int l,int h){

    int p;

    if(l<h)

    {

        p=partition(arr,l,h);

        Quicksort(arr,l,p-1);

        Quicksort(arr,p+1,h);

    }
```

```
}



void main(){

    int arr[100];

    int n,i;





    printf("Enter the number of elements of the array: ");

    scanf("%d",&n);

    printf("Enter the elements : ");

    for(i=0;i<n;i++)

    {

        scanf("%d",&arr[i]);



    }

    Quicksort(arr,0,n-1);
```

```
printf("The sorted array is : ");

for(i=0;i<n;i++)

{

    printf("%d ",arr[i]);

}

}
```

Output:



```
Enter the number of elements of the array: 10
Enter the elements : 2
34
12
67
43
19
11
90
100
45
The sorted array is : 2 11 12 19 34 43 45 67 90 100

...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion:

Thus we have successfully implemented Merge sort and Quick sort algorithms.