Experiment No. : 2

Title: Divide and Conquer Strategy

Aim: To implement and analyse time complexity of Quick-sort and Merge sort and compare both.

Explanation and Working of Quick sort & Merge sort:

Quick Sort:

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.

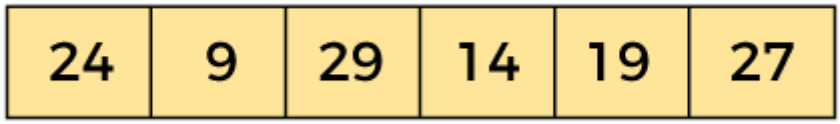
After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

Working of Quick Sort:

Now, let's see the working of the Quicksort Algorithm.

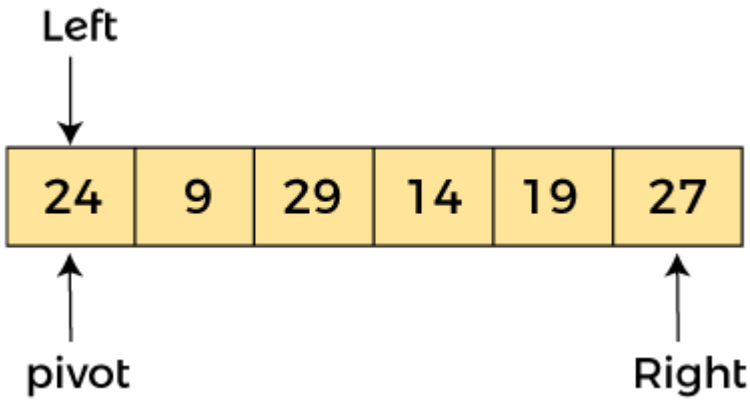
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

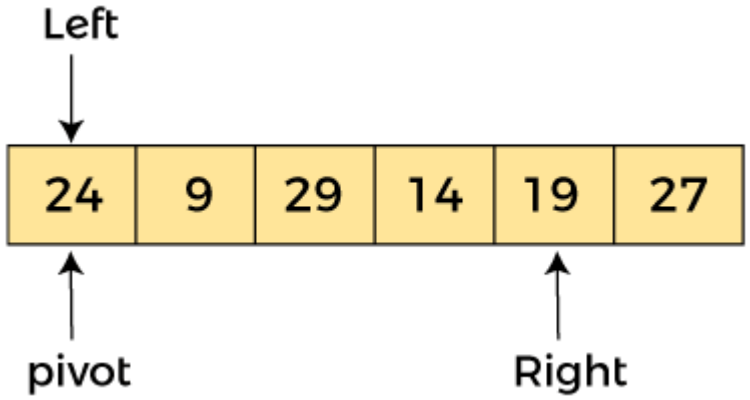


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

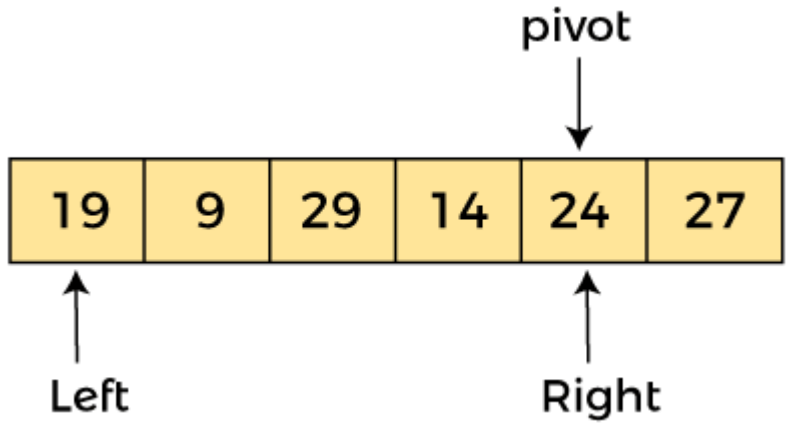


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. –



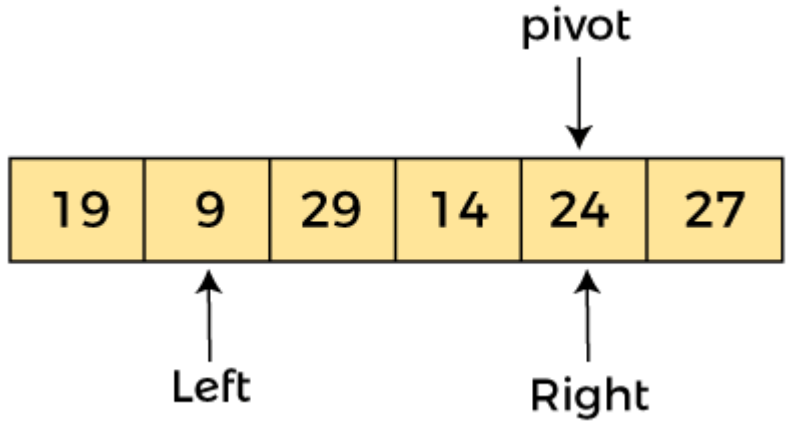
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

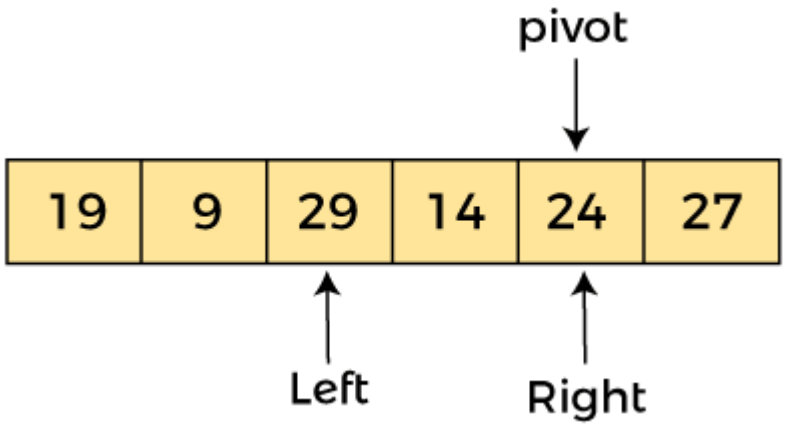


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

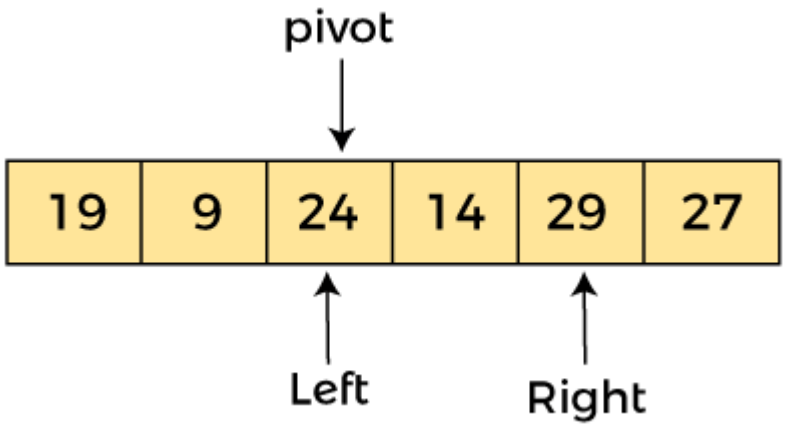
As a[pivot] > a[left], so algorithm moves one position to right as -



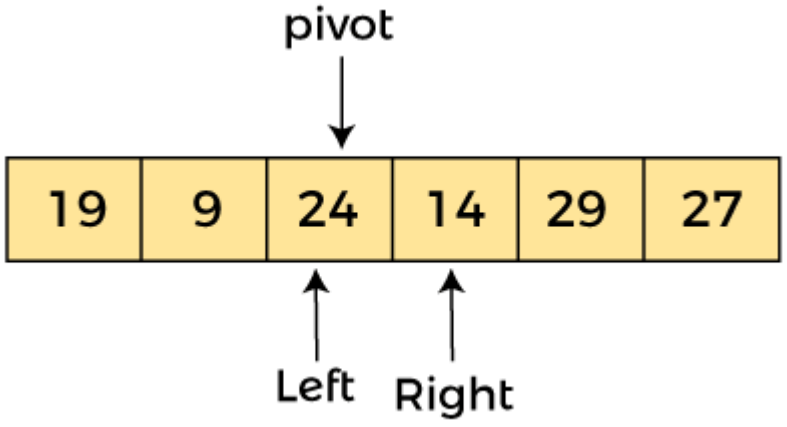
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



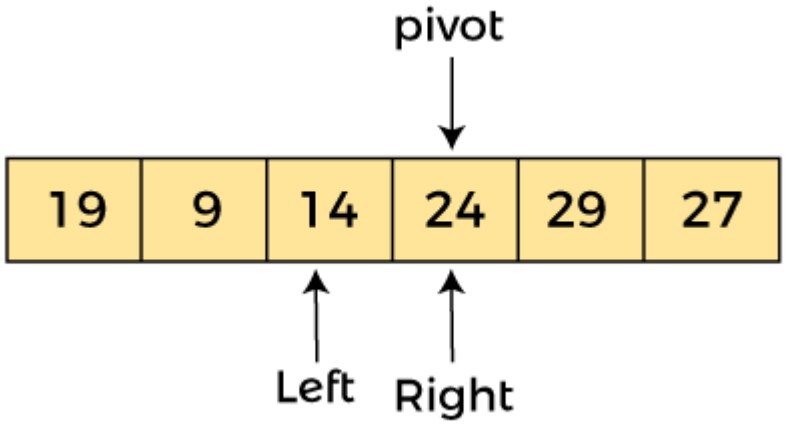
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



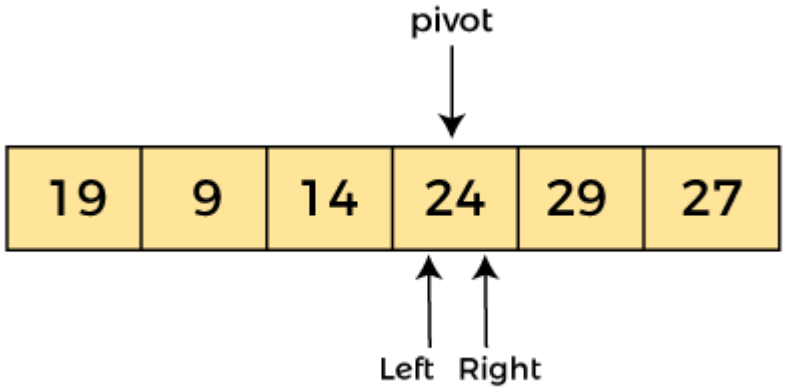
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



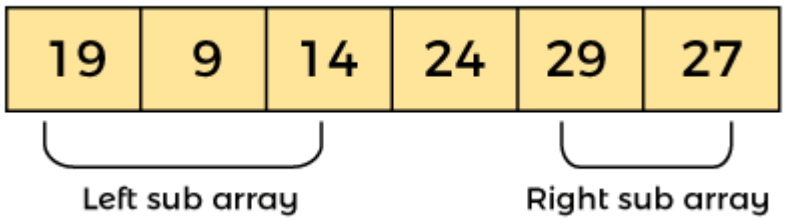
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -

9	14	19	24	27	29

Merge Sort:

Explanation:

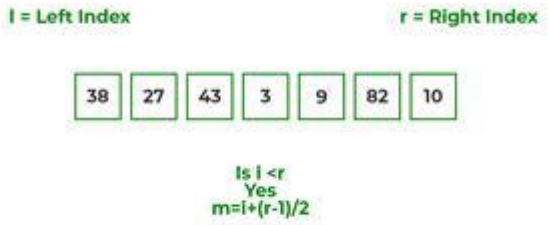
Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

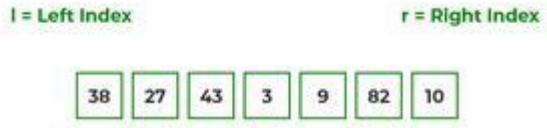
Working of Merge Sort:

To know the functioning of merge sort, lets consider an array arr[] = {38, 27, 43, 3, 9, 82, 10}

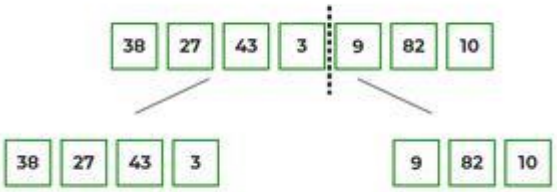
- At first, check if the left index of array is less than the right index, if yes then calculate its mid point



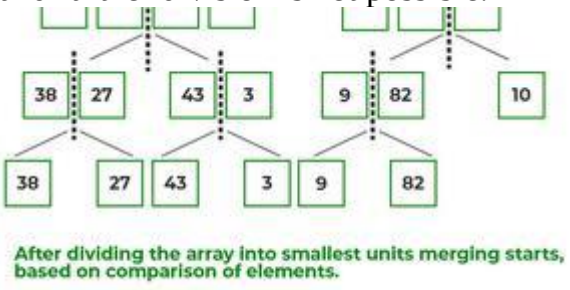
- Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.
- Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.



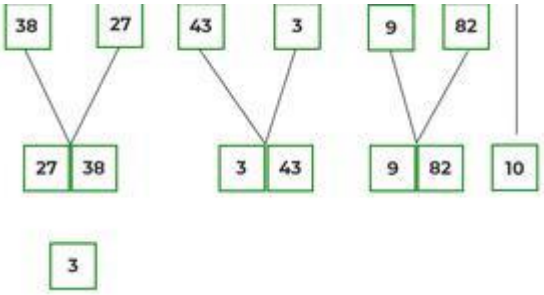
- Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.



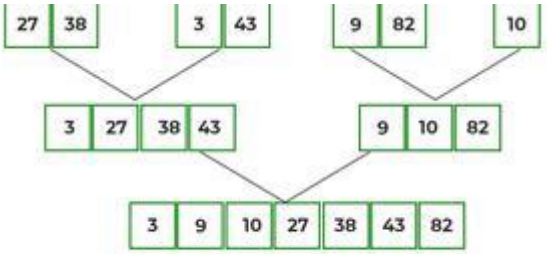
- Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.



- After dividing the array into smallest units, start merging the elements again based on comparison of size of elements
- Firstly, compare the element for each list and then combine them into another list in a sorted manner.



- After the final merging, the list looks like this:



Algorithm of Quick sort & Merge sort:

Quick Sort:

```
QUICKSORT (array A, start, end)
{
    if (start < end)
    {
     p = partition(A, start, end)
    QUICKSORT (A, start, p - 1)
    QUICKSORT (A, p + 1, end)
    }
}
```

```
PARTITION (array A, start, end)
{
    pivot ? A[end]
    i ? start-1
    for j ? start to end -1 {
    do if (A[j] < pivot) {
    then i ? i + 1
    swap A[i] with A[j]
```

```
        }}
        swap A[i+1] with A[end]
        return i+1
    }
```

Merge Sort:

1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

```
if left > right
    return
mid= (left+right)/2
mergesort(array, left, mid)
mergesort(array, mid+1, right)
merge(array, left, mid, right)
```

step 4: Stop

Derivation of Analysis Quick sort & Merge sort:

Quick sort:

Worst Case Analysis

- O(N^2)

- lets T(n) ne total time complexity for worst case
- n = total number of elements
- T(n) = T(n-1) + constant*n
- as we are dividing array into two parts one consist of single element and other of n-
- 1and we will traverse individual array
- T(n) = T(n-2) + constant*(n-1) + constant*n = T(n-2) + 2*constant*n - constantT(n)
- = T(n-3) + 3*constant*n - 2*constant - constant
- T(n) = T(n-k) + k*constant*n - (k-1)*constant ..... - 2*constant - constant
- T(n) = T(n-k) + k*constant*n - constant*[(k-1) ....+ 3 + 2 + 1]
- T(n) = T(n-k) + k*n*constant - constant*[k*(k-1)/2]
- put n=k

Best Case Analysis

- O(Nlog(N))

- Lets T(n) be the time complexity for best cases

- then
- T(n) = 2*T(n/2) + constant*n
- 2*T(n/2) is because we are dividing array into two array of equal size
-
- therefore,
- T(n) = 2*T(n/2) + constant*n
- further we will devide arrai in to array of equalsize so
- T(n) = 2*(2*T(n/4) + constant*n/2) + constant*n == 4*T(n/4) + 2*constant*n
-
- for this we can say that
- T(n) = 2^k * T(n/(2^k)) + k*constant*n
- then n = 2^k
- k = log2(n)

Average Case Analysis

- O(Nlog(N)

- lets T(n) be total time
- then for average we will consider random element as
- pivotlets index i be pivot
- then time complexity will be
- T(n) = T(i) + T(n-i)
- T(n) = 1/n *[\sum_{i=1}^{n-1} T(i)] + 1/n*[\sum_{i=1}^{n-1} T(n-i)]

- therefore
- T(n) = 2/n*[\sum_{i=1}^{n-1} T(i)]
-
- multiply both side by n
- n*T(n) = 2*[\sum_{i=1}^{n-1} T(i)] .....(1)
-
- put n = n-1
- (n-1)*T(n-1) = 2*[\sum_{i=1}^{n-2} T(i)] ..... (2)
-
- substract 1 and 2
- then we will get
- n*T(n) - (n-1)*T(n-1) = 2*T(n-1) + c*n^2 + c*(n-1)^2
- n*T(n) = T(n-1)[2+n-1] + 2*c*n - c
- n*T(n) = T(n-1)*(n+1) + 2*c*n [removed c as it was constant]
-
- divide both side by n*(n+1),
- T(n)/(n+1) = T(n-1)/n + 2*c/(n+1).....(3)
-
- put n = n-1,
- T(n-1)/n = T(n-2)/(n-1) + 2*c/n ..... (4)
-
- put n = n-2,
- T(n-2)/n = T(n-3)/(n-2) + 2*c/(n-1) ..... (5)
-
- by putting 4 in 3 and then 3 in 2 we will get
- T(n)/(n+1) = T(n-2)/(n-1) + 2*c/(n-1) + 2*c/n + 2*c/(n+1)
-
- also we can find equation for T(n-2) by putting n = n-2 in (3)
-
- at last we will get
-
- T(n)/(n+1) = T(1)/2 + 2*c * [1/(n-1) + 1/n + 1/(n+1) +.....]
-
- T(n)/(n+1) = T(1)/2 + 2*c*log(n) + C
-

- **T(n) = 2*c*log(n) * (n+1)**
-
- **now by removing constants,**

-
- T(n) = log(n)*(n+1)
- therefore,
-
-

Merge sort:

O(n*Log n) for all the 3 cases (best,worst,average).

Program(s) of Quick sort & Merge sort:

Quick sort:

```
#include <stdio.h>
#include<math.h>


void swap(int *a, int *b);
int partition(int arr[], int low, int high) ;
void quickSort(int arr[], int low, int high);
void user();
void R(int n);

void swap(int *a, int *b)
{
     int t = *a;
    *a = *b;

    *b = t;

}

int partition(int arr[], int low, int high)
{
```

```
    int pivot = arr[high]; int i = (low - 1);
    for (int j = low; j < high; j++)
     {
         if (arr[j] <= pivot)
        {
            i++;

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

        }

    }

    swap(&arr[i + 1], &arr[high]); return (i + 1);
}



void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int i = partition(arr, low, high);
        quickSort(arr, low, i - 1);
        quickSort(arr, i + 1, high);
    }

}
void user()
{

    int i,j,n,a[1000];
    printf("No of elements: ");
    scanf("%d",&n);
    printf("Elements:\n ");
    for(i=0;i<=n-1;i++)
    {
        scanf("%d", &a[i]);
    }

    quickSort(a, 0, n - 1);
     printf("Sorted array:\n");

    {

        for (int j = 0; j <= n-1; j++)
        {
            printf("%d ", a[j]);
```

```
        }

    }

}

void R(int n)
{
    printf("Time complexity of the best case : %f \n",n*log2(n));
    printf("Time complexity of the worst case : %d \n",n);
    printf("Time complexity of the average case : %f \n",n*log2(n));
}




int main()
{
    int p,n;

    printf("1.User Input\n2.Random value\nEnter your choice: ");
    scanf("%d",&p);
    switch (p)
    {
        case 1:
            user();
        break;

        case 2:


            printf("Enter the size: ");
            scanf("%d",&n);

            R(n);
        break;

        default:
            printf("Invalid input!");

    }

}
```

Merge sort:

```c
#include <stdio.h>
#include<stdlib.h>
#include<math.h>

void merge(int arr[], int l, int m, int r)
{
    int i,j;

    int n1 = m - l + 1;
    int n2 = r - m;

    int R[n2],L[n1];

    for (i = 0; i < n1; i++)
    {
         L[i] = arr[l + i];
    }
    for (j = 0; j < n2; j++)
    {
        R[j] = arr[m + 1 + j];
    }

    int k = l;
    i=0;
    j=0;
    while (i < n1 && j < n2)
    {
         if (L[i] <= R[j])
         {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
```

```
    while (j < n2)
    {
        arr[k] = R[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 user()
{
    int a[1000],i,n,l,r;
    printf("Enter the number of elements: ");
    scanf("%d",&n);
    printf("Elements: ");
    for(i=0;i<=n-1;i++)
    {
        scanf("%d",&a[i]);
    }
    l=0;
    r=n-1;
    mergeSort(a,l,r);
    printf("Sorted: ");
    for(i=0;i<=n-1;i++)
    {
        printf(" %d",a[i]);
    }
}

void R(int n)
{

    printf("Time complexity of the best case : %f \n",n*log2(n));
    printf("Time complexity of the worst case : %d \n",n);
```

```
    printf("Time complexity of the average case : %f \n",n*log2(n));
}

int main()
{
    int c,n;
    printf("1.User input\n2.Random number\nEnter your choice: ");
    scanf("%d",&c);

    switch(c)
    {
        case 1:
            user();
        break;

        case 2:
            printf("Enter the number of elements: ");
            scanf("%d",&n);
            R(n);
        break;

        default:
            printf("Invalid input!");
    }
}
```

Output(o) of Quick sort & Merge sort:

Quick sort:

```
1.User Input
2.Random value
Enter your choice: 1
No of elements: 5
Elements:
 12 4 56 90 22
Sorted array:
4 12 22 56 90
Process returned 0 (0x0)   execution time : 17.516 s
Press any key to continue.
```

```
1.User Input
2.Random value
Enter your choice: 2
Enter the size: 1000
Time complexity of the best case : 9965.784285
Time complexity of the worst case : 1000
Time complexity of the average case : 9965.784285

Process returned 0 (0x0)   execution time : 111.126 s
Press any key to continue.
```

```
1.User Input
2.Random value
Enter your choice: 2
Enter the size: 10000
Time complexity of the best case : 132877.123795
Time complexity of the worst case : 10000
Time complexity of the average case : 132877.123795

Process returned 0 (0x0)   execution time : 4.446 s
Press any key to continue.
```



```
1.User Input
2.Random value
Enter your choice: 2
Enter the size: 20000
Time complexity of the best case : 285754.247591
Time complexity of the worst case : 20000
Time complexity of the average case : 285754.247591

Process returned 0 (0x0)   execution time : 4.083 s
Press any key to continue.
```

Merge sort:

```
1.User input
2.Random number
Enter your choice: 1
Enter the number of elements: 5
Elements: 12 3 56 8 90
Sorted:  3 8 12 56 90
Process returned 0 (0x0)   execution time : 11.811 s
Press any key to continue.
```

```
1.User input
2.Random number
Enter your choice: 2
Enter the number of elements: 1000
Time complexity of the best case : 9965.784285
Time complexity of the worst case : 1000
Time complexity of the average case : 9965.784285

Process returned 0 (0x0)   execution time : 3.707 s
Press any key to continue.
```

```
1.User input
2.Random number
Enter your choice: 2
Enter the number of elements: 10000
Time complexity of the best case : 132877.123795
Time complexity of the worst case : 10000
Time complexity of the average case : 132877.123795

Process returned 0 (0x0)   execution time : 6.358 s
Press any key to continue.
```

```
1.User input
2.Random number
Enter your choice: 2
Enter the number of elements: 20000
Time complexity of the best case : 285754.247591
Time complexity of the worst case : 20000
Time complexity of the average case : 285754.247591

Process returned 0 (0x0)   execution time : 4.750 s
Press any key to continue.
```

Results:

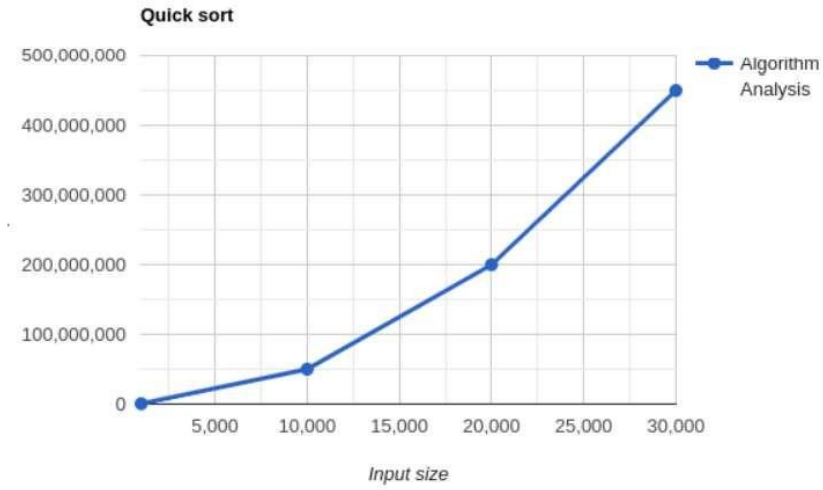
Time Complexity of Quick sort:

Worst Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis

1	1000	499500	6907
2	10000	49995000	92103
3	20000	1999900000	198069

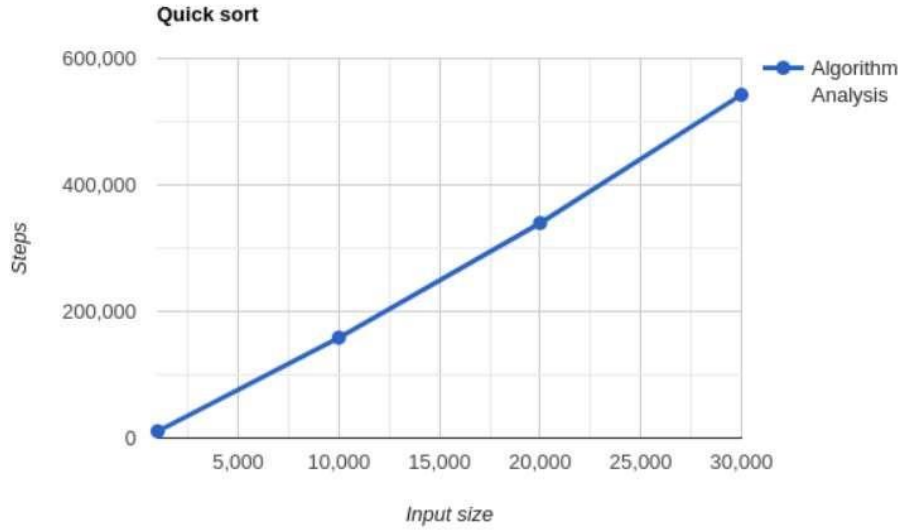
GRAPH:



Best Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
1	1000	10676	6907
2	10000	158562	92103
3	20000	339756	198069

GRAPH

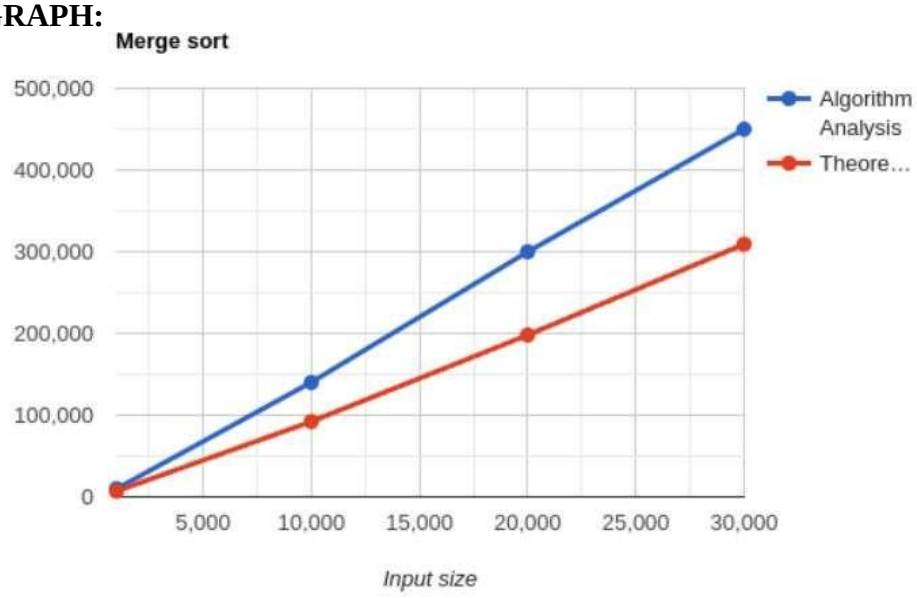


Time Complexity of Merge sort:

Worst Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
1	1000	10000	6907
2	10000	140000	92103
3	20000	300000	198069

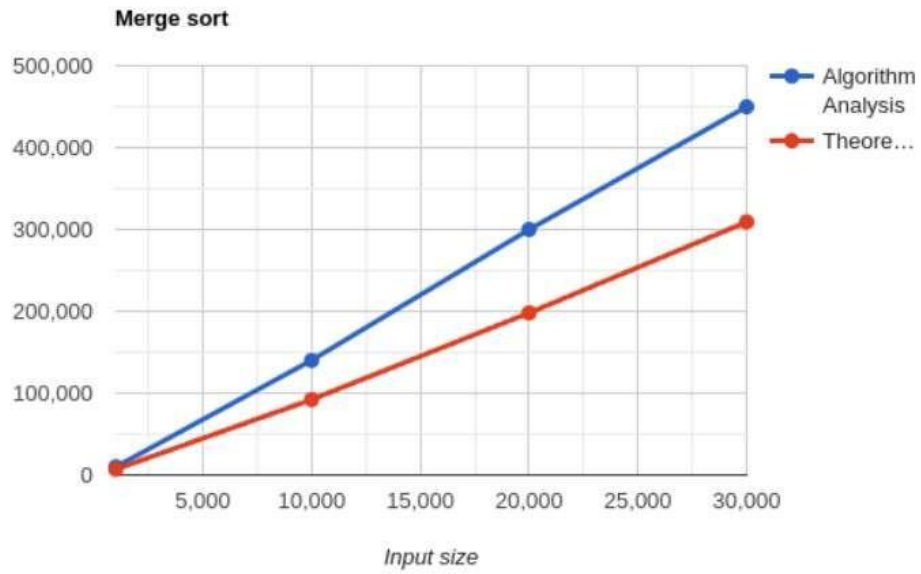
GRAPH:



Best Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
1	1000	10000	6907
2	10000	140000	92103
3	20000	300000	198069

GRAPH



Conclusion: (Based on the observations):

Thus we implemented merge sort and quick sort and also did the detailed analysis of the time complexity for both.

Outcome:

CO1 : Analyze time and space complexity of basic algorithms.

References:

1. Richard E. Neapolitan, " Foundation of Algorithms ", 5th Edition 2016, Jones & Bartlett Students Edition
2. Harsh Bhasin , " Algorithms : Design & Analysis", 1st Edition 2013, Oxford Higher education, India

3. T.H. Coreman ,C.E. Leiserson,R.L. Rivest, and C. Stein, " Introduction to algorithms", 3rd Edition 2009, Prentice Hall India Publication
4. Jon Kleinberg, Eva Tardos, " Algorithm Design", 10th Edition 2013, Pearson India Education Services Pvt. Ltd.