NAME: Meet Raut

DIV: S21

ROLL.NO: 2201084

Experiment 2:

- **Aim: To study and implement Quick Sort & Merge Sort algorithms.**

- **Theory:**

1. **Quick Sort Algorithm:** Sorting is a way of arranging items in a systematic manner. 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.

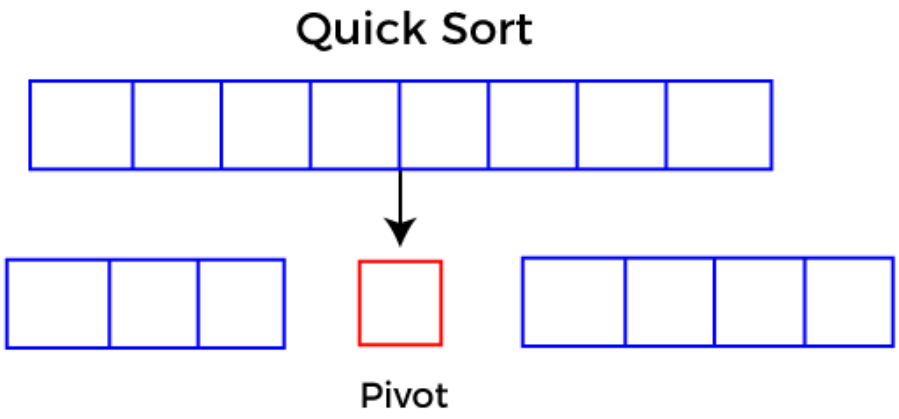
   Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

   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 }
}
```

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

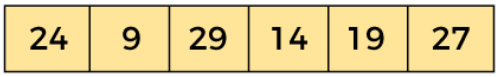
```
PARTITION (array A, start, end)
{
 1 pivot ? A[end]
 2 i ? start-1
 3 for j ? start to end -1 {
 4 do if (A[j] < pivot) {
 5 then i ? i + 1
 6 swap A[i] with A[j]
 7 }}
 8 swap A[i+1] with A[end]
 9 return i+1
}
```

➢ Working of Quick Sort Algorithm:

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

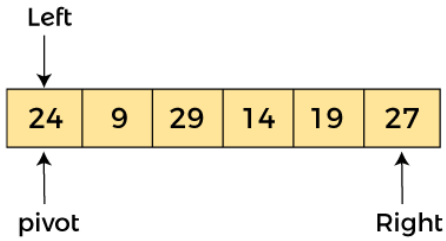
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear.

Let the elements of array are -

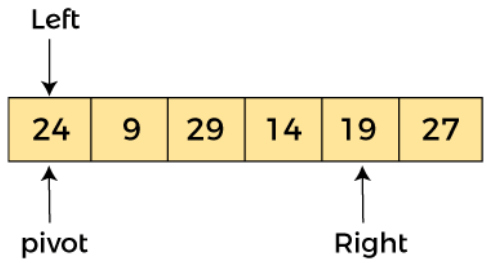


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

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

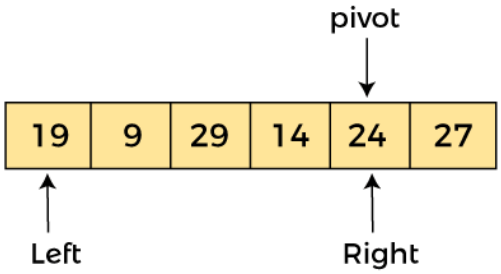


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. –



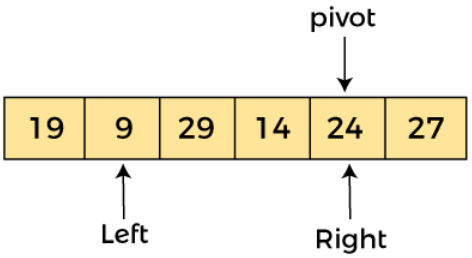
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

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

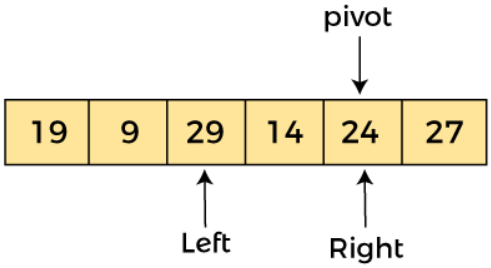


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

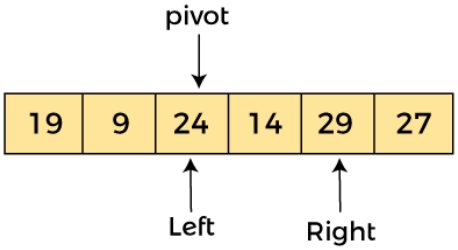
As a[pivot] > a[left], so algorithm moves one position to right as -



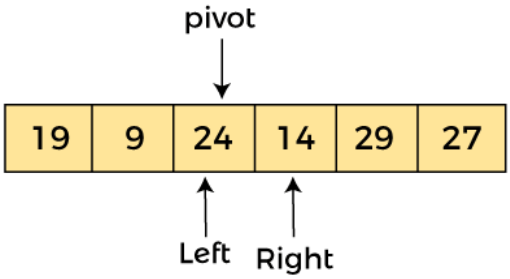
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



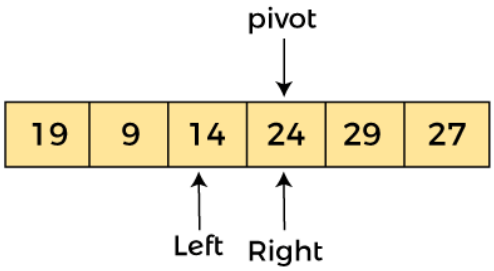
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



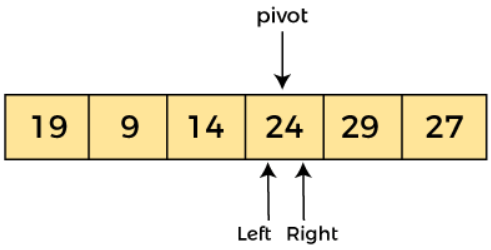
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



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. -



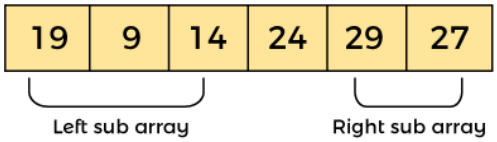
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



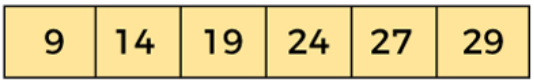
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

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 -



➢ 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(n2).

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).

➢ C PROGRAM (Quick Sort):

```
#include <stdio.h>

int partition (int a[], int low, int high)
{
   int pivot = a[high]; // pivot element
   int i = (low - 1);

   for (int j = low; j <= high - 1; j++)
   {
      // If current element is smaller than the pivot
      if (a[j] < pivot)
      {
         i++; // increment index of smaller element
         int t = a[i];
         a[i] = a[j];
```

```
            a[j] = t;
        }
    }

    int t = a[i+1];
    a[i+1] = a[high];
    a[high] = t;
    return (i + 1);
}

/* function to implement quick sort */
void quick(int a[], int low, int high)
{
    if (low < high)
    {
        int p = partition(a, low, high); //p is the partitioning index
        quick(a, low, p - 1);
        quick(a, p + 1, high);
    }
}

/* function to print an array */
void printArr(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}


int main()
{
    int a[] = { 24, 9, 19, 34, 15, 27, 1, 56 };
    int n = sizeof(a) / sizeof(a[0]);

    printf("\nBefore sorting array elements are - \n");
    printArr(a, n);
```

```
    quick(a, 0, n - 1);
    printf("\n\nAfter sorting array elements are - \n");
    printArr(a, n);

    return 0;
}
```

OUTPUT:

```
Before sorting array elements are -
24 9 19 34 15 27 1 56

After sorting array elements are -
1 9 15 19 24 27 34 56

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

---

2. **Merge Sort Algorithm:** 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.

Algorithm

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

```
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
```

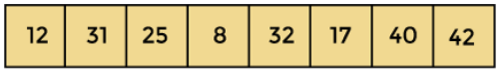
The important part of the merge sort is the MERGE function. This function performs the merging of two sorted sub-arrays that are A[beg…mid] and A[mid+1…end], to build one sorted array A[beg…end]. So, the inputs of the MERGE function are A[], beg, mid, and end.

➢ Working of Merge sort Algorithm:

Now, let's see the working of merge sort Algorithm.

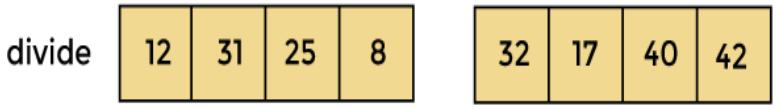
To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are -

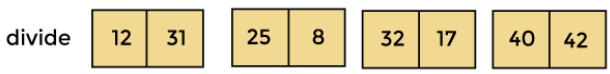


According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

As there are eight elements in the given array, so it is divided into two arrays of size 4.



Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.

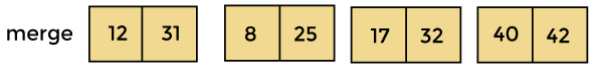


Now, again divide these arrays to get the atomic value that cannot be further divided.

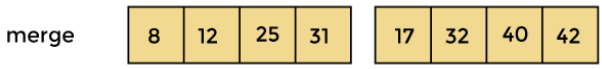
Now, combine them in the same manner they were broken.

In combining, first compare the element of each array and then combine them into another array in sorted order.

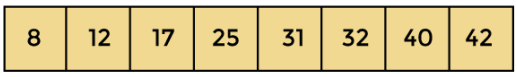
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



Now, the array is completely sorted.

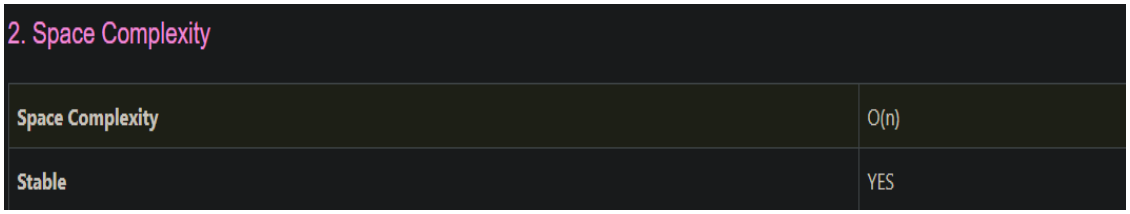
➢ MergeSort complexity:

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).



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

- **C PROGRAM (Merge Sort):**

```
#include <stdio.h>


void merge(int a[], int beg, int mid, int end)
{
   int i, j, k;
   int n1 = mid - beg + 1;
   int n2 = end - mid;

   int LeftArray[n1], RightArray[n2]; //temporary arrays

   /* copy data to temp arrays */
   for (int i = 0; i < n1; i++)
   LeftArray[i] = a[beg + i];
   for (int j = 0; j < n2; j++)
   RightArray[j] = a[mid + 1 + j];

   i = 0;               /* initial index of first sub-array */
   j = 0;               /* initial index of second sub-array */
   k = beg;             /* initial index of merged sub-array */

   while (i < n1 && j < n2)
   {
      if(LeftArray[i] <= RightArray[j])
      {
         a[k] = LeftArray[i];
         i++;
```

```
        }

        else
        {
            a[k] = RightArray[j];
            j++;
        }
        k++;
    }

    while (i<n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }

    while (j<n2)
    {
        a[k] = RightArray[j];
        j++;
        k++;
    }
}

void mergeSort(int a[], int beg, int end)
{
    if (beg < end)
    {
        int mid = (beg + end) / 2;
        mergeSort(a, beg, mid);
        mergeSort(a, mid + 1, end);
        merge(a, beg, mid, end);
    }
}
```

```
/* Function to print the array */
void printArray(int a[], int n)
{
   int i;
   for (i = 0; i < n; i++)
      printf("%d ", a[i]);
   printf("\n");
}


int main ()
{
   int a[] = { 12, 31, 15, 8, 32, 27, 4, 42, 1, 45, 60 };
   int n = sizeof(a) / sizeof(a[0]);
   printf("\nBefore sorting array elements are - \n");
   printArray(a, n);
   mergeSort(a, 0, n - 1);
   printf("\n\nAfter sorting array elements are - \n");
   printArray(a, n);
   return 0;
}
```

OUTPUT:

```
Before sorting array elements are -
12 31 15 8 32 27 4 42 1 45 60


After sorting array elements are -
1 4 8 12 15 27 31 32 42 45 60


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

Basis for comparison	Quick Sort	Merge Sort
The partition of elements in the array	The splitting of a array of elements is in any ratio, not necessarily divided into half.	In the merge sort, the array is parted into just 2 halves (i.e. n/2).
Worst case complexity	O(n^2)	O(nlogn)
Works well on	It works well on smaller array	It operates fine on any size of array
Speed of execution	It work faster than other sorting algorithms for small data set like Selection sort etc	It has a consistent speed on any size of data
Additional storage space requirement	Less(In-place)	More(not In-place)
Efficiency	Inefficient for larger arrays	More efficient
Sorting method	Internal	External
Stability	Not Stable	Stable
Preferred for	for Arrays	for Linked Lists
Locality of reference	good	poor
Major work	The major work is to partition the array into two sub-arrays before sorting them recursively.	Major work is to combine the two sub-arrays after sorting them recursively.
Division of array	Division of an array into sub-arrays may or may not be balanced as the array is partitioned around the pivot.	Division of an array into sub array is always balanced as it divides the array exactly at the middle.
Method	Quick sort is in- place sorting method.	Merge sort is not in – place sorting method.

➢ Quick Sort vs Merge Sort :

1. **Partition of elements in the array:** In the merge sort, the array is parted into just 2 halves (i.e. n/2). whereas in case of quick sort, the array is parted into any ratio. There is no compulsion of dividing the array of elements into equal parts in quick sort.

2. **Worst case complexity:** The worst case complexity of quick sort is O(n^2) as there is need of lot of comparisons in the worst condition. whereas In merge sort, worst case and average case has same complexities O(n log n).

3**. Usage with datasets:** Merge sort can work well on any type of data sets irrespective of its size (either large or small). whereas the quick sort cannot work well with large datasets.

4. **Additional storage space requirement:** Merge sort is not in place because it requires additional memory space to store the auxiliary arrays. whereas the quick sort is in place as it doesn't require any additional storage.

5. **Efficiency:** Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets. whereas Quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets.

6. **Sorting method:** The quick sort is internal sorting method where the data is sorted in main memory. whereas the merge sort is external sorting method in which the data that is to be sorted cannot be accommodated in the memory and needed auxiliary memory for sorting.

7. **Stability:** Merge sort is stable as two elements with equal value appear in the same order in sorted output as they were in the input unsorted array. whereas Quick sort is unstable in this scenario. But it can be made stable using some changes in code.

8. **Preferred for:** Quick sort is preferred for arrays. whereas Merge sort is preferred for linked lists.

9. **Locality of reference:** Quicksort exhibits good cache locality and this makes quicksort faster than merge sort (in many cases like in virtual memory environment).

- **<u>CONCLUSION</u>: Hence, we have successfully implemented Quick Sort & Merge Sort algorithm; LO1, LO2.**