**Name**: Ravi Jamanbhai Makwana

**Roll No**: 21BCP418

Div-6, G-12

**Assignment-2**

**Aim:**

To understand the concept and implementation of Merge sort and Quick sort.

**Theory:**

**Merge Sort:**

**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.

Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement. It is often used in conjunction with other algorithms, such as quicksort, to improve the overall performance of a sorting routine.

**Quick Sort:**

Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes **nlog 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.

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.

**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.

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

**Quick Sort:**

**Quick Sort Algorithm:**

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 Pivot Algorithm:**

Step 1 − Choose the highest index value has pivot

Step 2 − Take two variables to point left and right of the list excluding pivot

Step 3 − left points to the low index

Step 4 − right points to the high

Step 5 − while value at left is less than pivot move right

Step 6 − while value at right is greater than pivot move left

Step 7 − if both step 5 and step 6 does not match swap left and right

Step 8 − if left ≥ right, the point where they met is new pivot

quickSort(array, leftmostIndex, rightmostIndex)

if (leftmostIndex < rightmostIndex)

pivotIndex <- partition(array,leftmostIndex, rightmostIndex)

quickSort(array, leftmostIndex, pivotIndex - 1)

quickSort(array, pivotIndex, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex)

set rightmostIndex as pivotIndex

storeIndex <- leftmostIndex - 1

for i <- leftmostIndex + 1 to rightmostIndex

if element[i] < pivotElement

swap element[i] and element[storeIndex]

storeIndex++

swap pivotElement and element[storeIndex+1]

return storeIndex + 1

**Program:**

**Merge Sort:**

#include<stdio.h>

void merge(int arr[],int l, int mid, int r){

int size1 = mid-l+1;

int size2 = r-mid;

int arrr1[size1];

int arrr2[size2];

//copy elements of right and left sub-array to the temporary arrays

for (int i=0;i<size1;i++){

arrr1[i] = arr[l+i];

}

for (int i=0;i<size2;i++){

arrr2[i] = arr[mid+1+i];

}

int i=0, j=0, k=l;

//merging

while (i<size1 && j<size2){

if (arrr1[i] <= arrr2[j]){

arr[k++] = arrr1[i++];

}

else{

arr[k++] = arrr2[j++];

}

}

//if one of the arrr1 or arrr2 is completely copied while other one is left

while (i<size1){

arr[k++] = arrr1[i++];

}

while (j<size2){

arr[k++] = arrr2[j++];

}

}

void merge\_sort(int arr[], int l, int r){

if (l<r){

int mid = (l+r)/2;

merge\_sort(arr,l,mid);

merge\_sort(arr,mid+1,r);

merge(arr,l,mid,r);

}

}

int main(){

int array[6] = {6,3,2,8,7,4};

merge\_sort(array,0,5);

for (int i=0;i<=5;i++){

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

}

}

**Quick Sort:**

#include<stdio.h>

void quick\_sort(int arr[], int l, int r){

int counter = l-1; //initialize the value from -1

int pivot = arr[r];

for (int i=l;i<=r;i++){

if (arr[i] <= pivot){

//increment the counter and swap arr[counter] and arr[i]

counter++;

int temp = arr[counter];

arr[counter] = arr[i];

arr[i] = temp;

}

}

//for left sub-array

if (l != counter){

quick\_sort(arr, l, counter-1);

}

//for right sub-array

if (counter != r){

quick\_sort(arr, counter+1, r);

}

}

int main(){

int size = 10;

int array[] = {6,3,2,8,7,4,2,1,5,10};

quick\_sort(array,0,size-1);

for (int i=0;i<size;i++){

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

}

}

//ALGORITHM

//quick sort is done using recursion

//select last element of array as pivot and rearrange the rest of the array

//rearrange in such a way that all elements <= pivot are to the left and rest are rearranged to the right

//this is done using swapping (using a counter (variable) and the for (loop variable))

//once this is done, call the function for the left sub-array (left to the pivot) and for the right sub-array (right to the pivot)

//during each call, make the pivot as the last element of that sub array

//stopping condition is until counter becomes equal to l (for left array) or equal to r (for right array)

//i.e. when the sub-array has exactly 2 elements in it

**Output:**

**Merge Sort:**



**Quick Sort:**



**Analysis of Algorithm:**

**Merge Sort:**

**Time Complexity:**

A merge sort consists of several passes over the input. The first pass merges segments of size 1, the second merges segments of size 2, and the pass merges segments of size *2i-1*. Thus, the total number of passes is [*log2n*]. As merge showed, we can merge two sorted segments in linear time, which means that each pass takes *O(n)* time. Since there are [*log2n*] passes, the total computing time is **O(nlogn)**.

Best Case Complexity: O(n\*log n)

Worst Case Complexity: O(n\*log n)

Average Case Complexity: O(n\*log n)

**Space Complexity:**

The space complexity of merge sort is O(n).

**Quick Sort:**

**Time Complexity:**

**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)**.