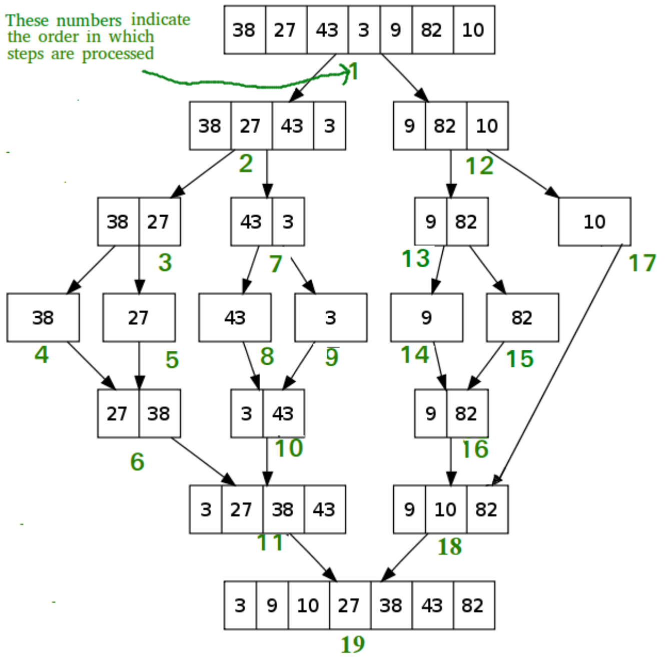
Sardar Patel Institute of Technology
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058, India
(Autonomous College Affiliated to University of Mumbai)

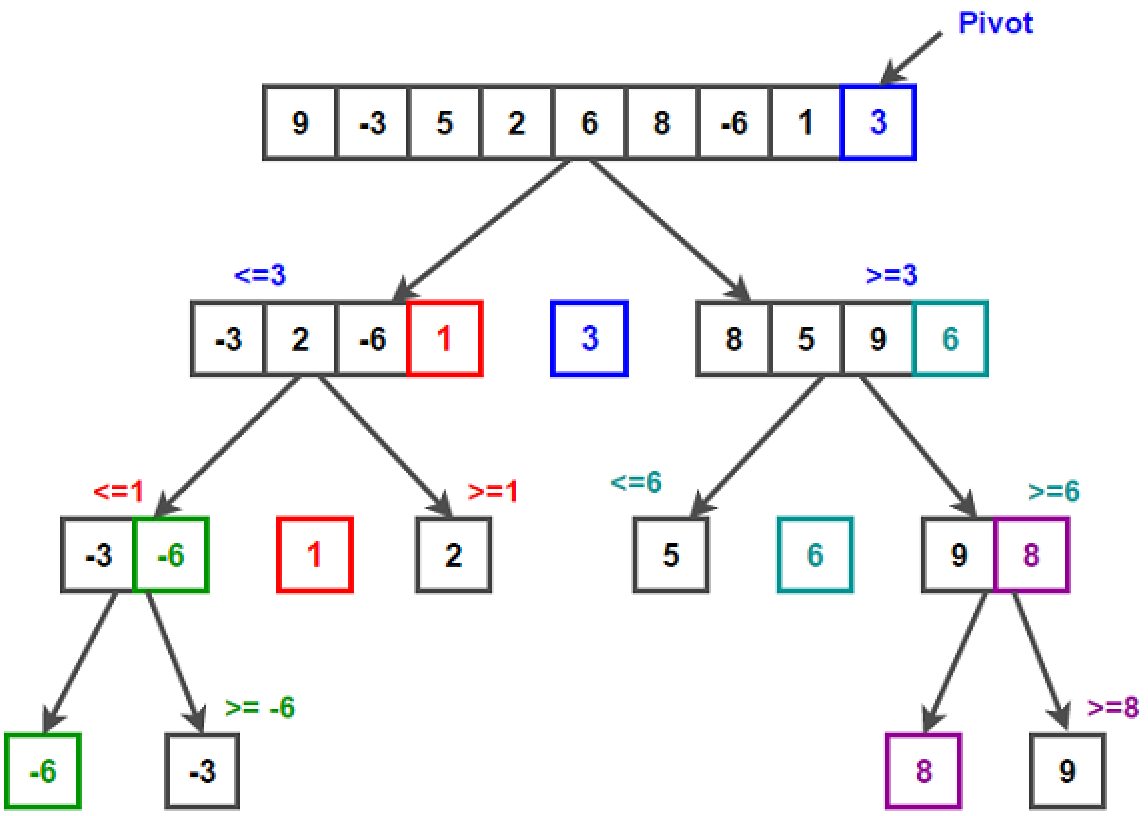
Experiment No.	2
Aim	Merge sort and quick sort
Name	Vishaka hole
UID No.	2021300043
Class & Division	Comps A(A3)

Theory:

1. 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.



2. Quick Sort: QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.
   - Always pick the first element as a pivot.
   - Always pick the last element as a pivot (implemented below)
   - Pick a random element as a pivot.
   - Pick median as the pivot.



Algorithm:

1. Merge Sort:

```
step 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.
```

```
That is: MergeSort(arr[], l,  r)
            If r > l
            Find the middle point to divide the array into two halves:
            middle m = l + (r – l)/2
          Call mergeSort for first half:
          Call mergeSort(arr, l, m)
          Call mergeSort for second half:
          Call mergeSort(arr, m + 1, r)
          Merge the two halves sorted in steps 2 and 3:
          Call merge(arr, l, m, r)
```

2. Quick Sort:

```
/* low  –> Starting index,  high  –> Ending index */

quickSort(arr[], low, high) {

    if (low < high) {

        /* pi is partitioning index, arr[pi] is now at right place */

        pi = partition(arr, low, high);

        quickSort(arr, low, pi – 1);  // Before pi

        quickSort(arr, pi + 1, high); // After pi

    }

}
```

Experiment: MERGE SORT:

```c
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
    are any */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
    are any */
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
```

```c
{
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
int main(){

    FILE *fptr, *sPtr;
    int index=99;
    int arrNums[100000];
    clock_t t;
    fptr = fopen("numbers.txt", "r");
    sPtr = fopen("mTimes.txt", "w");
    for(int i=0; i<=999; i++){
        for(int j=0; j<=index; j++){
            fscanf(fptr, "%d", &arrNums[j]);
        }
        t = clock();
        mergeSort(arrNums,0,index);
        //mergeSort(int arr[], 0, arr_size - 1);
        t = clock() - t;
        double time_taken = ((double)t)/CLOCKS_PER_SEC;
        fprintf(sPtr, "%lf\n", time_taken);
        printf("%d\t%lf\n", (i+1), time_taken);
        index = index + 100;
        fseek(fptr, 0, SEEK_SET);
    }
    fclose(sPtr);
    fclose(fptr);

    return 0;
}
```

QUICK SORT:

```c
#include<stdio.h>
#include<math.h>
#include<conio.h>
#include<stdlib.h>
#include<time.h>
void swap(int *x, int *y){
    int temp= *x;
    *x = *y;
    *y = temp;
}
int partition(int arr[], int low, int high){
    int pivot = arr[high];
    int i= (low-1);
    for(int j=low; j<=high-1; 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 pivot= partition(arr,low, high);
        quickSort(arr, low, pivot-1);
        quickSort(arr,pivot+1, high);
    }
}
int main(){

    FILE *fptr, *sPtr;
    int index=99;
    int arrNums[100000];
    clock_t t;
    fptr = fopen("numbers.txt", "r");
    sPtr = fopen("qTimes.txt", "w");
    for(int i=0; i<=999; i++){
        for(int j=0; j<=index; j++){
            fscanf(fptr, "%d", &arrNums[j]);
        }
        t = clock();
        quickSort(arrNums,0,index);
        //mergeSort(int arr[], 0, arr_size - 1);
        t = clock() - t;
        double time_taken = ((double)t)/CLOCKS_PER_SEC;
        fprintf(sPtr, "%lf\n", time_taken);
        printf("%d\t%lf\n", (i+1), time_taken);
        index = index + 100;
        fseek(fptr, 0, SEEK_SET);
    }
    fclose(sPtr);
    fclose(fptr);

    return 0;
}
```

Conclusion: The time complexity of merge sort is O(n log n) which is faster than many other sorting algorithm such as insertion sort, that have a time complexity of O(n^2). Merge sort is also a stable sorting algorithm, meaning that it maintains the relative order of equal elements in the array. However, merge sort does require additional memory to store the sub arrays during the sorting process, which can be a disadvantage for large arrays with limited memory. Additionally the recursive nature of the algorithm can lead to a large number of function call and a high overhead cost.

The average time complexity of quick sort algorithm is O(n log n), making it one of the fastest sorting algorithm available. However, the worst-case time complexity of O(n^2) can occur when the pivot is chosen poorly, leading to an unbalanced partition. To avoid this, various optimizations such as randomized pivot selection and three-way partitioning can be used.