Name	RITIK SINGH
UID no.	20201700061
Experiment No.	2

AIM:	Experiment based on divide and conquer approach.
Program 1	
PROBLEM STATEMENT :	– you need to implement two sorting algorithms namely Quicksort and Merge sort methods. Compare these algorithms based on time and space complexity. Time required for sorting algorithms can be performed using high_resolution_clock::now() under namespace std::chrono. You have to generate 1,00,000 integer numbers using C/C++ Rand function and save them in a text file. Both the sorting algorithms uses these 1,00,000 integer numbers as input as follows. Each sorting algorithm sorts a block of 100,200,300,...,100000 integer numbers with array indexes numbers A[0..99], A[100..199], A[200..299],…, A[99900..99999]. You need to use high_resolution_clock::now() function to find the time required for 100, 200, 300…. 100000 integer numbers. Finally, compare two algorithms namely Quicksort and Merge sort by plotting the time required to sort integers using LibreOffice Calc/MS Excel. The x-axis of 2-D plot represents the block no. of 1000 blocks. The y-axis of 2-D plot represents the tunning time to sort 1000 blocks of 100,200,300,...,100000 integer numbers.
ALGORITHM/ THEORY:	**What is Time complexity?** Time complexity is defined as the amount of time taken by an algorithm to run, as a function of the length of the input. It measures the time taken to execute each statement of code in an algorithm. It is not going to examine the total execution time of an algorithm. Rather, it is going to give information about the variation (increase or decrease) in execution time when the number of operations (increase or decrease) in an algorithm. Yes, as the definition says, the amount of time taken is a function of the length of input only. **What is Quick Sort?** Quicksort is based on a divide and conquer strategy. It works in the following steps:

1. It selects an element from within the array known as the **pivot element**.
2. Then it makes use of the **partition algorithm** to divide the array into two sub-arrays. One sub-array has all the values less than the pivot element. The other sub-array has all the values higher than the pivot element.
3. In the next step, the quicksort algorithm calls itself recursively to sort these two sub-arrays.
4. Once the sorting is done, we can combine both the sub-arrays into a single sorted array.

The most important part of quicksort is the partition algorithm. The partition algorithm puts the element into either of the two subarrays depending on the pivot point. We can choose pivot point in many ways:

- Take the first element as the pivot point
- Take the last element of the array as the pivot point
- Take the middle element of the array as the pivot element.
- Take random element as the pivot element in every recursive call

PARTITION ALGORITHM:-

Step 1: Choose the highest index value i.e. the last element of the array as a pivot point

Step 2: Point to the 1st and last index of the array using two variables.

Step 3: Left points to the low index and Right points to the high

Step 4: while Array[Left] < pivot

Move Right

Step 5: while Array[Right] > pivot

Move Left

Step 6: If no match found in step 5 and step 6, swap Left and Right

Step 7: If Left ≥ Right, their meeting point is the new pivot

QUICKSORT ALGORITHM:-

Step 1 − Array[Right] = pivot

Step 2 − Apply partition algorithm over data items using pivot element

Step 3 − quicksort(left of pivot)

Step 4 − quicksort(right of pivot)

What is Merge Sort?

The merge sort algorithm is an implementation of the divide and conquer technique. Thus, it gets completed in three steps:

1. Divide: In this step, the array/list divides itself recursively into sub-arrays

	until the base case is reached. **2. Recursively solve:** Here, the sub-arrays are sorted using recursion. **3. Combine:** This step makes use of the **merge( ) function** to combine the sub-arrays into the final sorted array. **MERGESORT ALGORITHM:-** Step 1: Find the middle index of the array. Middle = 1 + (last – first)/2 Step 2: Divide the array from the middle. Step 3: Call merge sort for the first half of the array MergeSort(array, first, middle) Step 4: Call merge sort for the second half of the array. MergeSort(array, middle+1, last) Step 5: Merge the two sorted halves into a single sorted array.
PROGRAM:	(code below)

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

void getInput()
{
  FILE *fp;
  fp = fopen("inputexp2.text","w");
  for(int i=0;i<100000;i++)
  fprintf(fp,"%d ",rand()%100000);
  fclose(fp);
}
void merge(int arr[], int p, int q, int r) {

  // Create L ← A[p..q] and M ← A[q+1..r]
  int n1 = q - p + 1;
  int n2 = r - q;

  int L[n1], M[n2];

  for (int i = 0; i < n1; i++)
    L[i] = arr[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = arr[q + 1 + j];
```

	`` // Maintain current index of sub-arrays and main array int i, j, k; i = 0; j = 0; k = p; // Until we reach either end of either L or M, pick larger among // elements L and M and place them in the correct position at A[p..r] while (i < n1 && j < n2) { if (L[i] <= M[j]) { arr[k] = L[i]; i++; } else { arr[k] = M[j]; j++; } k++; } // When we run out of elements in either L or M, // pick up the remaining elements and put in A[p..r] while (i < n1) { arr[k] = L[i]; i++; k++; } while (j < n2) { arr[k] = M[j]; j++; k++; } } // Divide the array into two subarrays, sort them and merge them void mergeSort(int arr[], int l, int r) { if (l < r) { // m is the point where the array is divided into two subarrays ```

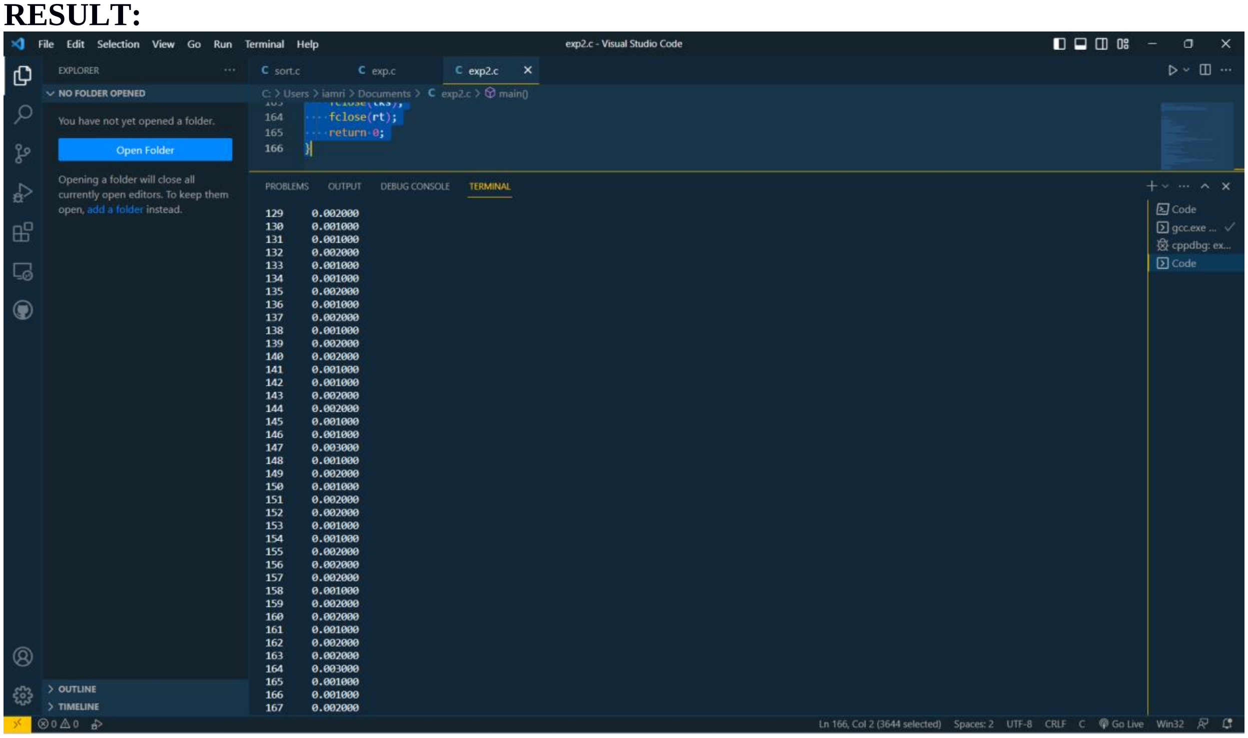
	`` int m = l + (r - l) / 2; mergeSort(arr, l, m); mergeSort(arr, m + 1, r); // Merge the sorted subarrays merge(arr, l, m, r); } } int partition(int A[], int low, int high) { int pivot = A[low]; int i = low + 1; int j = high; int temp; do { while (A[i] <= pivot) { i++; } while (A[j] > pivot) { j--; } if (i < j) { temp = A[i]; A[i] = A[j]; A[j] = temp; } } while (i < j); // Swap A[low] and A[j] temp = A[low]; A[low] = A[j]; A[j] = temp; return j; } ```

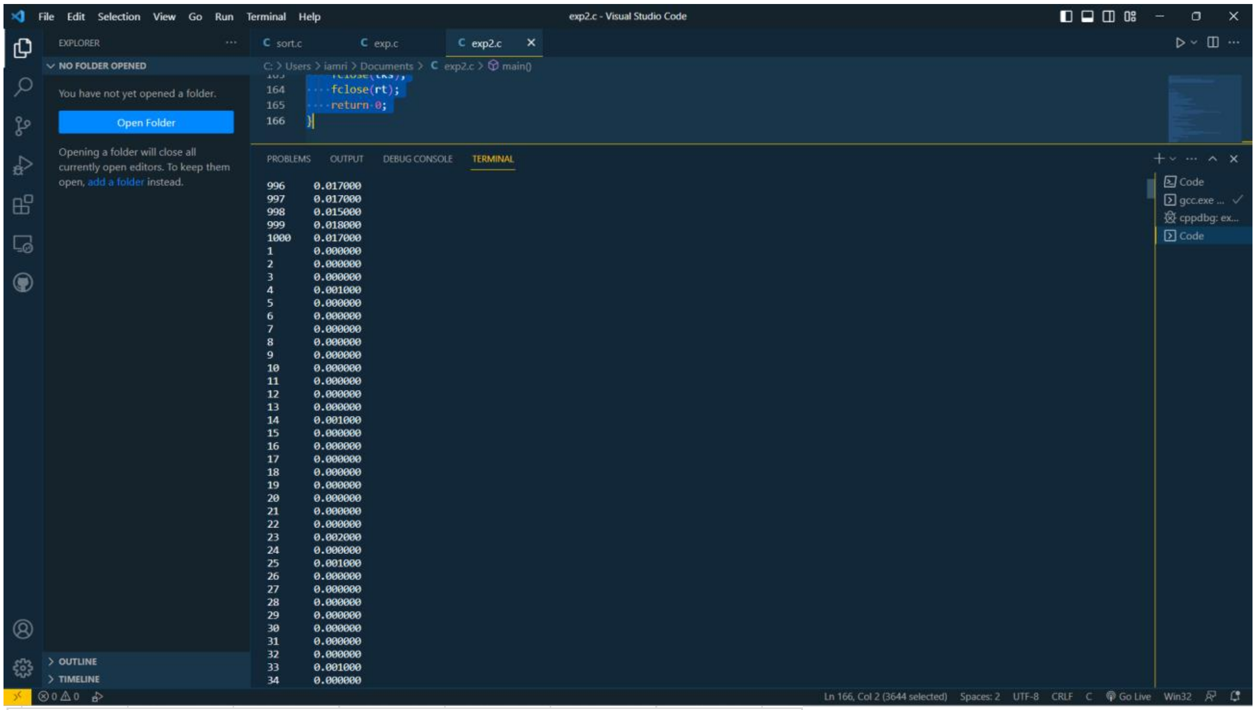
```
    int m = l + (r - l) / 2;

    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);

    // Merge the sorted subarrays
    merge(arr, l, m, r);
  }
}

int partition(int A[], int low, int high)
{
    int pivot = A[low];
    int i = low + 1;
    int j = high;
    int temp;

    do
    {
        while (A[i] <= pivot)
        {
            i++;
        }

        while (A[j] > pivot)
        {
            j--;
        }

        if (i < j)
        {
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    } while (i < j);

    // Swap A[low] and A[j]
    temp = A[low];
    A[low] = A[j];
    A[j] = temp;
    return j;
}
```

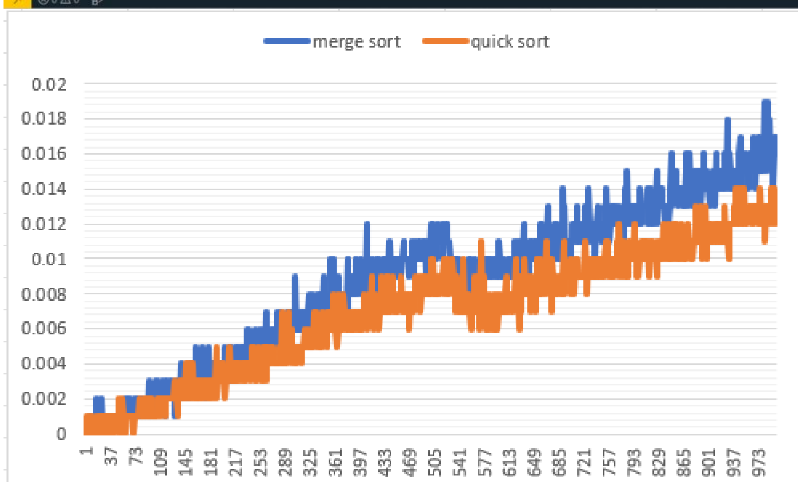
```
void quickSort(int A[], int low, int high)
{
    int partitionIndex; // Index of pivot after partition

    if (low < high)
    {
        partitionIndex = partition(A, low, high);
        quickSort(A, low, partitionIndex - 1);  // sort left
subarray
        quickSort(A, partitionIndex + 1, high); // sort right
subarray
    }
}



int main(){

    getInput();
    FILE *rt, *tks;
    int a=99;
    int arrNums[100000];
    clock_t t;
    rt = fopen("inputexp2.text", "r");
    tks = fopen("mTimes.txt", "w");
    for(int i=0; i<1000; i++){
        for(int j=0; j<=a; j++){
            fscanf(rt, "%d", &arrNums[j]);
        }
        t = clock();
        mergeSort(arrNums,0, a+1);
        t = clock() - t;
        double time_taken = ((double)t)/CLOCKS_PER_SEC;
        fprintf(tks, "time taken for %d iteration is %Lf\n",
(i+1), time_taken);
        printf("%d\t%lf\n", (i+1), time_taken);
        a = a + 100;
        fseek(rt, 0, SEEK_SET);
    }
    fclose(tks);
    tks = fopen("qTimes.txt", "w");
    a=99;
    for(int i=0; i<1000; i++){
        for(int j=0; j<=a; j++){
```

```c
            fscanf(rt, "%d", &arrNums[j]);
        }
        t = clock();
        quickSort(arrNums,0, a+1);
        t = clock() - t;
        double time_taken = ((double)t)/CLOCKS_PER_SEC;
        fprintf(tks, "time taken for %d iteration is %Lf\n",
(i+1), time_taken);
        printf("%d\t%lf\n", (i+1), time_taken);
        a = a + 100;
        fseek(rt, 0, SEEK_SET);
    }
    fclose(tks);
    fclose(rt);
    return 0;
}
```

RESULT:







CONCLUSION:	WE HAVE USED TWO ALGORITHM TECHNIQUES i.e MERGESORT AND QUICKSORT TO SORT THE RANDOM NO.s . BOTH THE ALGORITHMS HAVE LESS TIME COMPLEXITY. I HAVE SEEN BEHAVIOUR OF THE ALGORITHMS WITH TIME USING OF GRAPH . IT IS SEEN THAT QUICK HAS BETTER TIME COMPLEXITY THAN MERGE SORT.