Name:	Omkar Deshmukh
UID:	2021700018
Branch:	CSE(DS)D1
Exp:	02

*Aim:-Sorting Methods- Divide and Conquer

*Problem statement:-Experiment Based on Divide and Conquer Approach

***ALGORITHM THEORY:Merge Sort**

Merge sort is a divide-and-conquer algorithm that sorts an array or list by breaking it down into smaller and smaller subarrays, sorting those subarrays, and then merging them back together in the correct order. The algorithm has a worst-case time complexity of O(n log n) and a space complexity of O(n).

Here are the steps to perform a merge sort:

Divide the array or list into two halves.
Recursively sort each half.
Merge the two sorted halves back together.

Quick sort is also a divide-and-conquer algorithm that sorts an array or list by selecting a "pivot" element and partitioning the array or list around that pivot. Elements smaller than the pivot are moved to its left, and elements larger than the pivot are moved to its right. The pivot is

then placed in its correct position and the process is repeated for the two subarrays on either side of the pivot. The algorithm has a worst-case time complexity of O(n^2) and a space complexity of O(log n).

Mergesort

mergeSort(arr[], l, r)

if l < r

1. Find the middle point to divide the array into two halves:

   middle = l + (r - l) / 2

2. Call mergeSort for the first half:

   mergeSort(arr, l, middle)

3. Call mergeSort for the second half: mergeSort(arr, middle + 1, r)

4. Merge the two halves sorted in step 2 and 3:

   merge(arr, l, middle, r)

Quick Sort

quickSort(arr[], low, high) if low < high

1. partitionIndex = partition(arr, low, high)

2. Call quickSort for the first partition: quickSort(arr, low, partitionIndex - 1)

3. Call quickSort for the second partition:

   quickSort(arr, partitionIndex + 1, high)

PROGRAM:-

```
#include <stdio.h>
#include<time.h>
#include<stdlib.h>


void merge(int arr[], int l, int m, int r) {
int i, j, k;
int n1 = m - l + 1;


int n2 = r - m;


    int L[n1], R[n2];
```

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

        i = 0;  j = 0;     k = l;

   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 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 - 1; j++) {
        if (arr[j] < pivot) {
        i++;
        swap(&arr[i], &arr[j]);
        }
        }
        swap(&arr[i + 1], &arr[high]);
        return (i + 1);
```

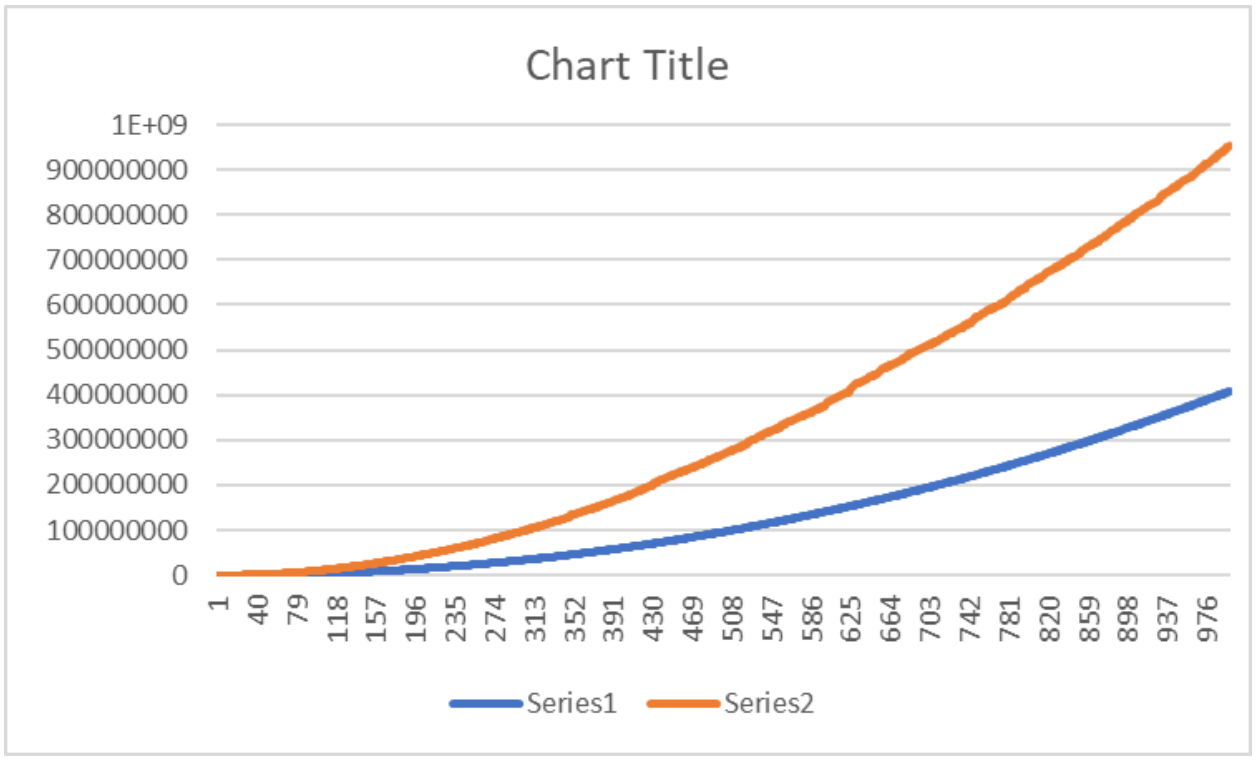
```
}

void quickSort(int arr[], int low, int high) {     if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
        }
}
 int main(void) {     clock_t  a,b,c;
        int arr[100000],arr2[100000];
double d,e;   FILE *fptr;
  int num;



  fptr = fopen("integers", "w");

  if (fptr != NULL) {
        printf("File created successfully!\n");
  }   else {
        printf("Failed to create the file.\n");
```

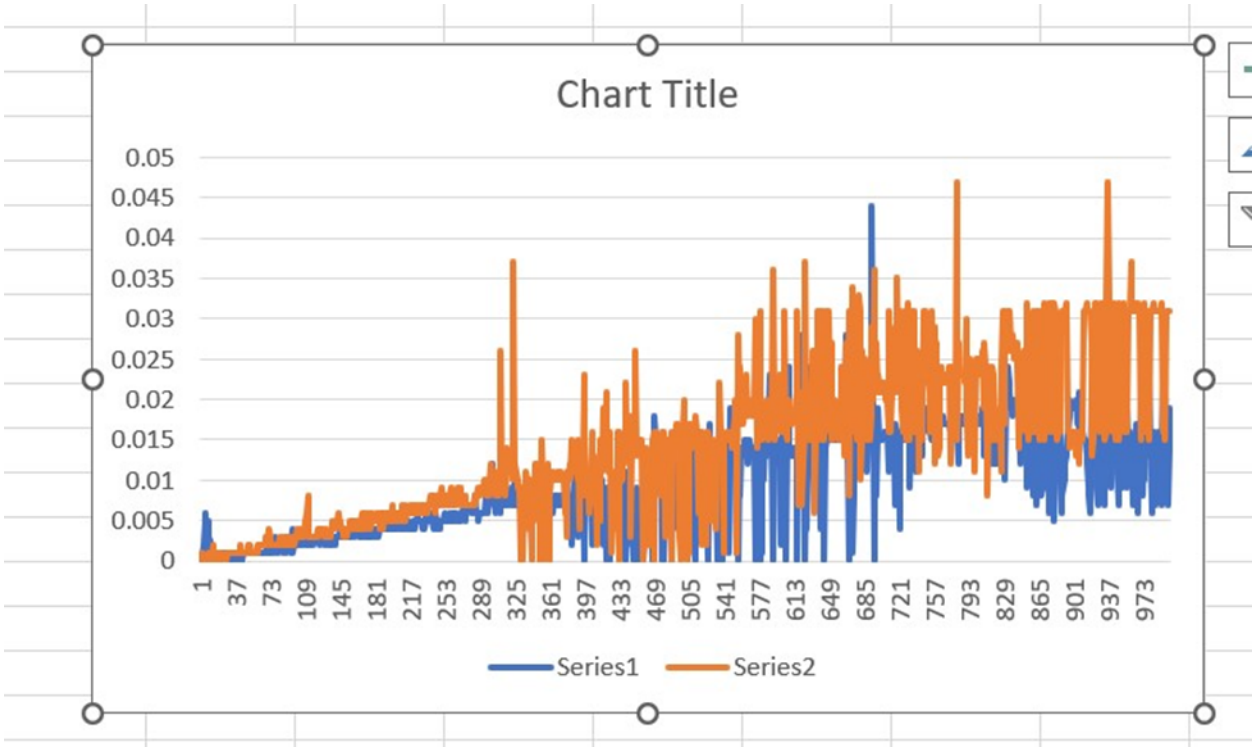
```
        return -1;
  }



int i=0;   while (i!=100000) {    num=rand()%100000;      if (num
!= -1) {
        putw(num, fptr);
    }   else {          break;
        }
        ++i;
  }
  fclose(fptr);
    fptr = fopen("integers", "r");
i=0;
  while ( (num = getw(fptr)) != EOF ) {
arr[i]=num;          arr2[i]=num;
        i++;
  }


  printf("\nEnd of file.\n");



  fclose(fptr);
```

```
  for(int j=100;j<=100000;j+=100){
a=clock();    mergeSort(arr,0,j-1);        b=clock();


     d=(double)(b-a)/CLOCKS_PER_SEC;
printf("%f",d);       printf("  ");
         quickSort(arr2,0,j-1);       c=clock();

         e=(double)(c-b)/CLOCKS_PER_SEC;

         printf("%f\n",e);




  }



  return 0;

}
```

Result: Blue: Merge sort

Red : quick sort





Observations:

Both merge sort and quick sort have an average-case time complexity of O(n log n), which makes them highly efficient for sorting large datasets. However, quick sort has a worst-case time complexity of O(n^2) if the pivot element is chosen poorly, while merge sort has a consistent worst-case time complexity of O(n log n). This means that merge sort is a safer choice for datasets with unpredictable distribution, while quick sort can be faster for datasets with a known distribution.

No of Comparisons in mergesort are far less than quick sort

Conclusion:-In this experiment, I have understood the divide and conquer algorithms like merge sort and quick sort behind them.