Name	Harsh Chandra
UID no.	2021700013
Experiment No.	2

AIM:	EXPERIMENT BASED ON DIVIDE AND CONQUER APPROACH
Program 1	
PROBLEM STATEMENT :	Experiment on finding the running time of sorting algorithms(Merge and Quick sort).
ALGORITHM/ THEORY:	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. Merge Sort Algorithm. 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 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. The key process in quickSort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

PROGRAM:	#include <stdio.h>
	#include <stdlib.h>
	#include <time.h>
	#include <stdbool.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])
	{

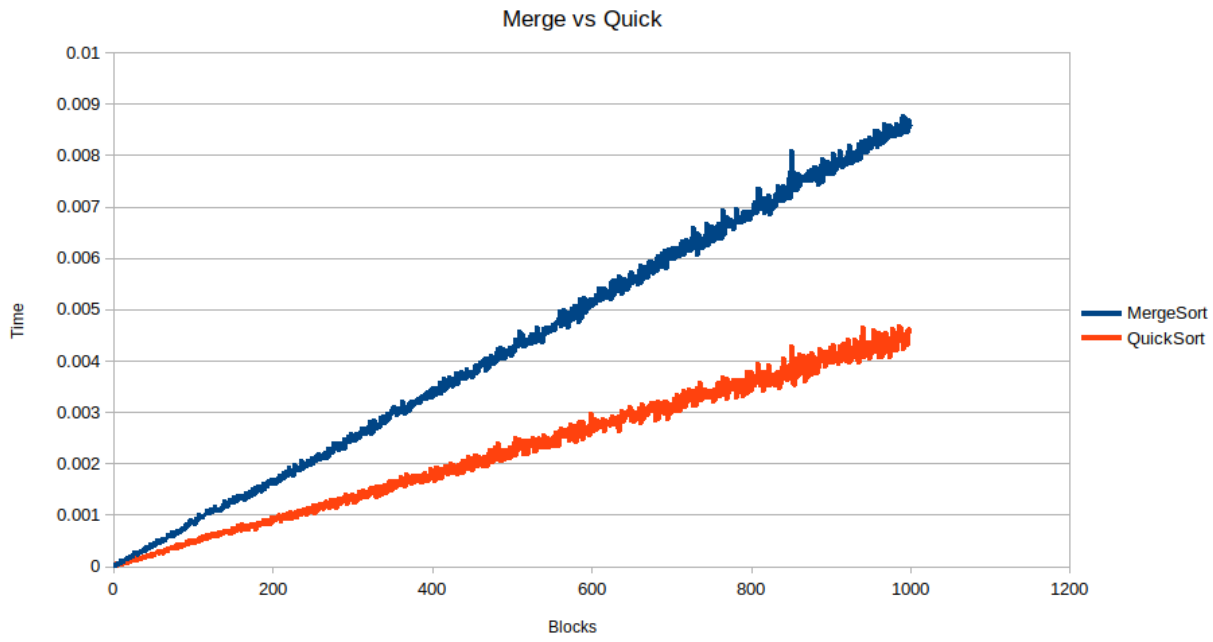
```
        arr[k] = L[i];
        i++;
        }
        else
        {
        arr[k] = R[j];
        j++;
        }
        k++;
        }

        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;
```

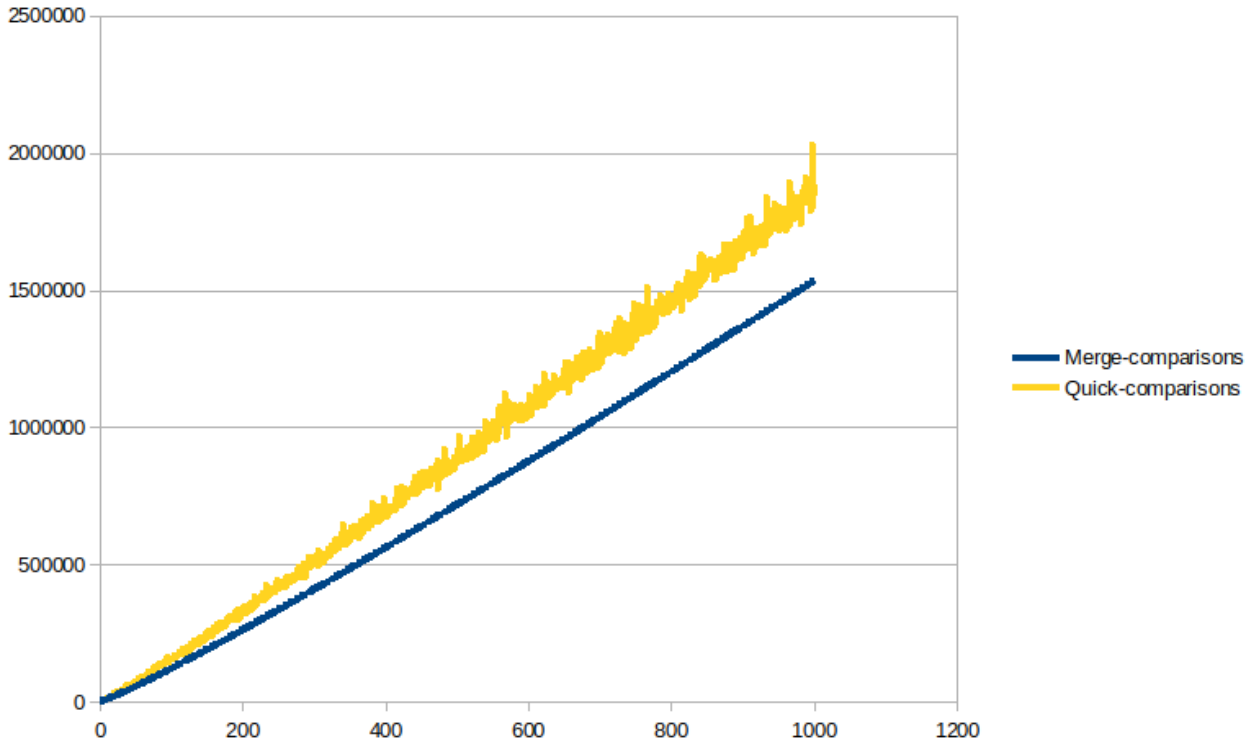
```
}

int partition(int arr[], int low, int high)
{
        int pivot = arr[low];    // first element as pivot.
        int i = low - 1, j = high + 1;

        while (true) {
        do {
        i++;
        } while (arr[i] < pivot);

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

        if (i >= j)
        return j;

        swap(&arr[i], &arr[j]);
        }
}


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(){

  FILE *fptr;
  int n = 100000;
  fptr = fopen("randomm.txt", "w");
  if (fptr == NULL){
        printf("ERROR Creating File!");
```

```
        exit(1);
}


srand(time(0));
for (int i = 1; i <= n; i++){
        int r = rand() % 100;
        fprintf(fptr, "%d\n", r);
}
fclose(fptr);


int block = 1;
printf("Block\tMergeSort\tQuickSort\n");
for (int i = 100; i <= n; i += 100){
        fptr = fopen("randomm.txt", "r");
        int arr[i];

        for (int j = 0; j < i; j++){
        fscanf(fptr, "%d", &arr[j]);
        }
        clock_t t;
        t = clock();
        mergeSort(arr,0,i-1);   // merge sort
        t = clock() - t;
        double time_takenms = ((double)t) / CLOCKS_PER_SEC;
        fclose(fptr);

        fptr = fopen("randomm.txt", "r");
        int arr2[i];
        for (int j = 0; j < i; j++){
        fscanf(fptr, "%d", &arr2[j]);
        }
        clock_t t2;
        t2 = clock();
        quickSort(arr2,0,i-1);  // quick sort
        t2 = clock() - t2;
        double time_takenqs = ((double)t2) / CLOCKS_PER_SEC;
  fclose(fptr);
```

```
  fptr = fopen("randomm.txt", "r");
        int arr3[i];
        for (int j = 0; j < i; j++){
        fscanf(fptr, "%d", &arr3[j]);
        }
  fclose(fptr);

        printf("%d\t%f\t%f\n", block, time_takenms, time_takenqs);
        block++;

 }
 return 0;
}
```

RESULT:





Quick sort algorithm (implemented using Lomuto partitioning scheme) is faster than merge sort for almost all input sizes.
Number of comparison operations required by merge sort is exactly linearly proportional to the input size. When quick sort is implemented using Lomuto partitioning scheme, number of comparisons required is higher than merge sort.

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.

CONCLUSION:	From this experiment I learnt how to perform merge sort and quicksort algorithms. I used the sorting techniques on 100000 random numbers and found out that runtime for merge sort is less than quicksort which helped me analyse that merge sort is the algorithm more efficient between the two, when working on a large set of data.