NAME:	Arnav Hoskote
UID:	2021300044
SUBJECT	Design and Analysis of Algorithm
EXPERIMENT NO :	02
DATE OF PERFORMANCE	13/02/2023
DATE OF SUBMISSION	20/02/2023
AIM:	To sort 100,000 randomly generated numbers using merge and quick sort.
PROBLEM STATEMENT 1:	
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. One of the main advantages of merge sort is that it has a time complexity of O(n log n), which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort. 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:** **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. There can be many ways to do partition, following pseudo-code adopts the method given in the CLRS book. The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.
ALGORITHM:	**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

QUICK SORT:

```
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
   }
}


partition (arr[], low, high)
{
   // pivot (Element to be placed at right position)
   pivot = arr[high];
   i = (low – 1)  // Index of smaller element and indicates the
   // right position of pivot found so far
   for (j = low; j <= high- 1; j++){
      // If current element is smaller than the pivot
      if (arr[j] < pivot){
         i++;    // increment index of smaller element
         swap arr[i] and arr[j]
      }
   }
   swap arr[i + 1] and arr[high])
   return (i + 1)
}
```

PROGRAM:

```
#include<stdio.h>
#include<time.h>
#include<stdlib.h>
void merge(int a[],int beg,int end,int mid)
{
        int i,j,k;
        int n1=mid-beg+1;
        int n2=end-mid;
        int leftarr[n1],rightarr[n2];
        for(i=0;i<mid;i++)
        {
                leftarr[i]=a[mid+i];
        }
        for(j=0;j<end;j++)
        {
                rightarr[j]=a[mid+1+j];
        }
        i=0;
        j=0;
        k=beg;
        while(i<n1&&j<n2)
        {
                if(leftarr[i]<rightarr[j])
                {
                        a[k]=leftarr[i];
                        i++;
                }
                else
                {
                        a[k]=rightarr[j];
                        j++;
                }
                k++;
        }
```

```
        while(i<n1)
        {
                a[k]=leftarr[i];
                i++;
                k++;
        }
        while(j<n2)
        {
                a[k]=rightarr[j];
                j++;
                k++;
        }
}
void mergesort(int a[],int l,int u)
{
        if(l<u)
        {
                int mid=(u-l)/2;
                mergesort(a,l,mid);
                mergesort(a,mid+1,u);
                merge(a,l,u,mid);
        }
}
int partition(int a[],int l,int u)
{
        int temp,t;
        int i=l-1;
        int pivot=a[u];
        for(int j=l;j<u;j++)
        {
                if(a[j]<pivot)
                {
                        t=a[j];
                        a[j]=a[i];
```

```
                        a[i]=t;
                }
        }
        temp=a[i+1];
        a[i+1]=a[u];
        a[u]=temp;
        return (i+1);
}
void quick(int a[],int l,int u)
{
        if(l<u)
        {
                int p=partition(a,l,u);
                quick(a,l,p);
                quick(a,p+1,u);
        }
}
void main()
{
        long int n=0;
        for(int k=0; k<(100000/100); k++)
        {
                n=n+100;
                int num[n];
                int quicksort[n];
                int merge[n];
                int j, min;
                clock_t start_t, end_t;
                double total_t;
                printf("%ld\t",n);
                for(int i=0; i<n; i++)
                {
                        num[i]=rand() % 10;
                        merge[i]=num[i];
```

	quicksort[i]=num[i];
	}
	start_t = clock();
	mergesort(merge, 0 ,n-1);
	end_t = clock();
	total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
	printf("%lf\t", total_t);
	start_t = clock();
	quicksort(quicksort, 0 ,n-1);
	end_t = clock();
	total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
	printf("%lf\n", total_t);
	}
	}

CONCLUSION:	By performing above experiment I have understood the time complexities of merge sort and quick sort.