	Bansilal Ramnath Agarwal Charitable Trust's Vishwakarma Institute of Information Technology **Department of** **Artificial Intelligence and Data Science**		
Name: Vikram Ram Biradar			
Class: TY	Division: c		Roll No:372007
Semester: V		Academic Year:2023-24	
Subject Name & Code: Design and Analysis of Algorithm			
Title of Assignment: Write a program to perform binary search on an unsorted random list of at least 5000 elements. The key element should be user input. Use the Divide & Conquer method to implement this program.			

Aim:

Implementation the following algorithm using Divide & Conquer method.
(a)Merge sort (b) Quick Sort.
Also display execution time for different size of input and perform the analysis.

Problem Statement-

To Implement merge sort and Quick sort and to perform the analysis of algorithms

Background Information:

Quick sort is an internal algorithm which is based on divide and conquer strategy. In this:

- The array of elements is divided into parts repeatedly until it is not possible to divide it further.
- It is also known as **“partition exchange sort”**.
- It uses a key element (pivot) for partitioning the elements.
- One left partition contains all those elements that are smaller than the pivot and one right partition contains all those elements which are greater than the key element.

Merge sort is an external algorithm and based on divide and conquer strategy. In this:

- The elements are split into two sub-arrays (n/2) again and again until only one element is left.
- Merge sort uses additional storage for sorting the auxiliary array.
- Merge sort uses three arrays where two are used for storing each half, and the third external one is used to store the final sorted list by merging other two and each array is then sorted recursively.
- At last, the all-sub arrays are merged to make it ‘n’ element size of the array.

Performance Analysis

1. **Partition of elements in the array**:
   In the merge sort, the array is parted into just 2 halves (i.e. n/2).
   whereas
   In case of quick sort, the array is parted into any ratio. There is no compulsion of dividing the array of elements into equal parts in quick sort.

2. **Worst case complexity**:
   The worst-case complexity of quick sort is O(n2) as there is need of lot of comparisons in the worst condition.
   whereas
   In merge sort, worst case and average case has same complexities

O (n log n).

3. **Usage with datasets**:
   Merge sort can work well on any type of data sets irrespective of its size (either large or small).
   whereas
   The quick sort cannot work well with large datasets.

4. **Additional storage space requirement**:
   Merge sort is not in place because it requires additional memory space to store the auxiliary arrays.
   whereas
   The quick sort is in place as it doesn't require any additional storage.

5. **Efficiency**:
   Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets.
   whereas
   Quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets.

6. **Sorting method**:
   The quick sort is internal sorting method where the data is sorted in main memory.
   whereas
   The merge sort is external sorting method in which the data that is to be sorted cannot be accommodated in the memory and needed auxiliary memory for sorting.

7. **Stability**:
   Merge sort is stable as two elements with equal value appear in the same order in sorted output as they were in the input unsorted array.
   whereas
   Quick sort is unstable in this scenario. But it can be made stable using some changes in code.

8. **Preferred for**:
   Quick sort is preferred for arrays.
   whereas

Merge sort is preferred for linked lists.

9. **Locality of reference**:
   Quicksort exhibits good cache locality, and this makes quicksort faster than merge sort (in many cases like in virtual memory environment).

Software Requirements:

VS CODE, GCC compiler

Program Code:

Merge Sort Code –

```
/*
Implementation the following algorithm using Divide & Conquer method.
(a)Merge sort
(b) Quick Sort
Also display execution time for different size of input and perform the analysis.
*/


#include <bits/stdc++.h>

using namespace std;

void merge(int *arr,int start,int end)
{
    int mid=start+(end-start)/2;

    // since we are going to merge two arrays we will create two tempapry arrays

    // the first temp array to store the half values i.e till mid and the second temp
array to store the values from mid to end


   // size of the temp arrays
    int len1= mid-start+1;
    int len2=end-mid;
```

```
//creating the temp arrays

int *first=new int[len1];
int *second=new int[len2];

// coping values into the temporary arrays
int mainArrayIndex=start;
for(int i=0;i<len1;i++)
{
    first[i]=arr[mainArrayIndex++];

}

mainArrayIndex=mid+1;

for(int i=0;i<len2;i++)
{
    second[i]=arr[mainArrayIndex++];

}

//merging two sorted arrays

int index1=0;
int index2=0;
mainArrayIndex=start;

while(index1<len1 && index2<len2)
{
    if(first[index1]<second[index2])
    {
       arr[mainArrayIndex++] = first[index1++];

    }

    else{
                   arr[mainArrayIndex++] = second[index2++];
```

```
        }



    }


    while(index1<len1)
    {
         arr[mainArrayIndex++] = first[index1++];

    }

     while(index2<len2)
    {
         arr[mainArrayIndex++] = second[index2++];

    }

    delete []first;
    delete []second;
}



void mergesort(int arr[],int start,int end)
{

        // base case
        if(start>=end)
        {
            return;
        }
    int mid=start+(end-start)/2;

        // recalling the function for the left part of the array
        mergesort(arr,start,mid);

        // recalling the fucntion for the right part of the array
        mergesort(arr,mid+1,end);

        //calling the merge fucntion
        merge(arr,start,end);



}

void print(int *arr,int size)
{
      for(int i=0;i<size;i++)
    {
        cout<<arr[i]<<endl;
    }
}

int main()
{
    time_t start, end;
    time(&start);
    ios_base::sync_with_stdio(false);
    //taking size as input
    cout<<"Size of array"<<endl;
    int size;
    cin>>size;
    int arr[size];
    cout<<"elements of array"<<endl;
    for(int i=0;i<size;i++)
    {
        cin>>arr[i];
    }

cout<<"given array is"<<endl;
 print(arr,size);


mergesort(arr,0,size-1);
```

```
cout<<"sorted array is"<<endl;
    print(arr,size);



    /*for(int i=0;i<size;i++)
    {
        cout<<arr[i]<<endl;
    }*/

   time(&end);
    double time_taken = double(end - start);
    cout << "Time taken by program is : " << fixed
         << time_taken << setprecision(5);
    cout << " sec " << endl;
    return 0;

}
```

Quick Sort Code

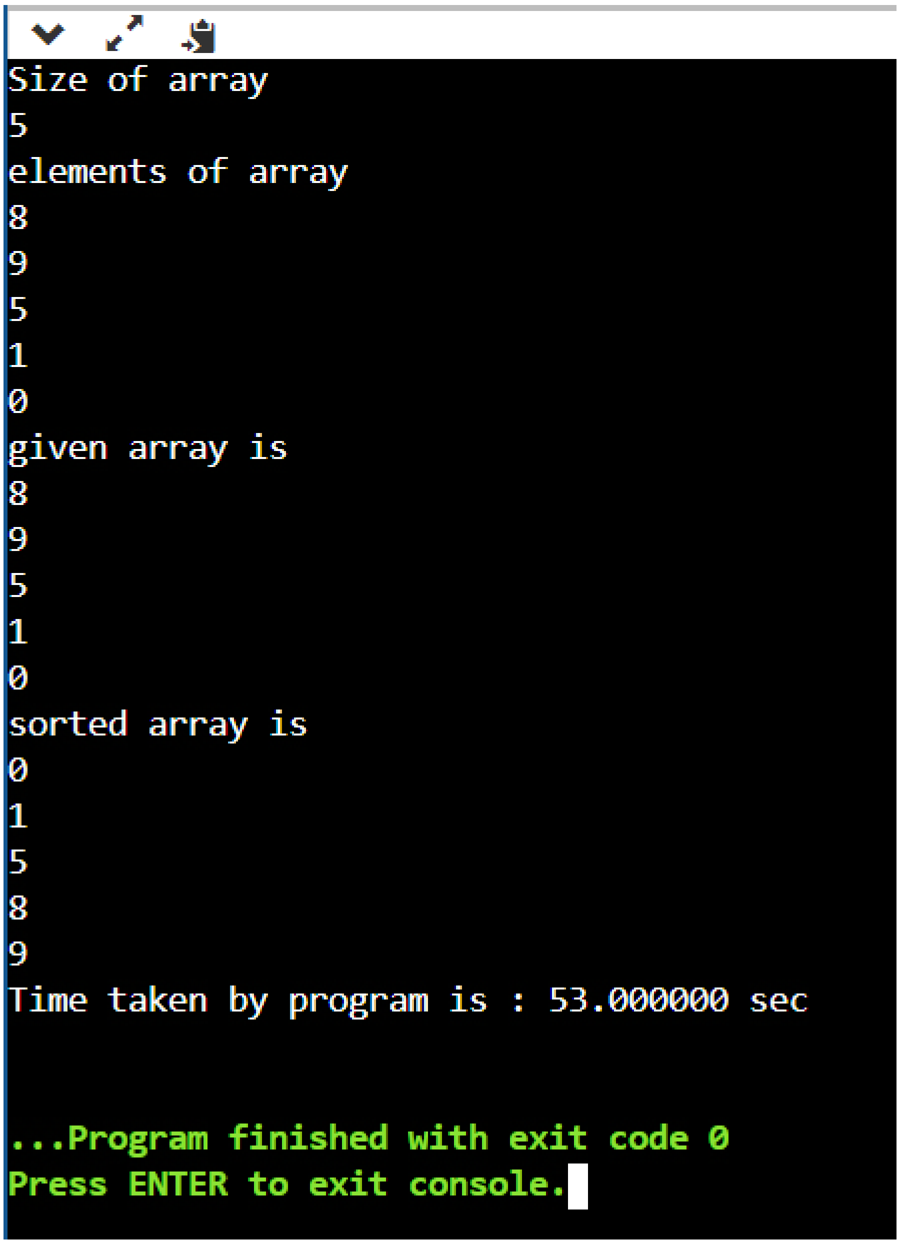
```
/*
Implementation the following algorithm using Divide & Conquer method.
(a)Merge sort
(b) Quick Sort
Also display execution time for different size of input and perform the analysis.
*/

#include<bits/stdc++.h>

using namespace std;

void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
```

```
}

/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element and indicates the right position of
pivot found so far

    for (int 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], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}







void print(int *arr,int size)
{
      for(int i=0;i<size;i++)
    {
        cout<<arr[i]<<endl;
    }
}

int main()
{
    time_t start, end;
    time(&start);
    ios_base::sync_with_stdio(false);

    //taking size as input
    cout<<"Size of array"<<endl;
    int size;
    cin>>size;
    int arr[size];
    cout<<"elements of array"<<endl;
    for(int i=0;i<size;i++)
    {
        cin>>arr[i];
    }

cout<<"given array is"<<endl;
 print(arr,size);
```

```
quickSort(arr,0,size-1);

cout<<"Sorted Array is"<<endl;
print(arr,size);



    /*for(int i=0;i<size;i++)
    {
        cout<<arr[i]<<endl;
    }*/

   time(&end);
    double time_taken = double(end - start);
    cout << "Time taken by program is : " << fixed
         << time_taken << setprecision(5);
    cout << " sec " << endl;
    return 0;

}
```

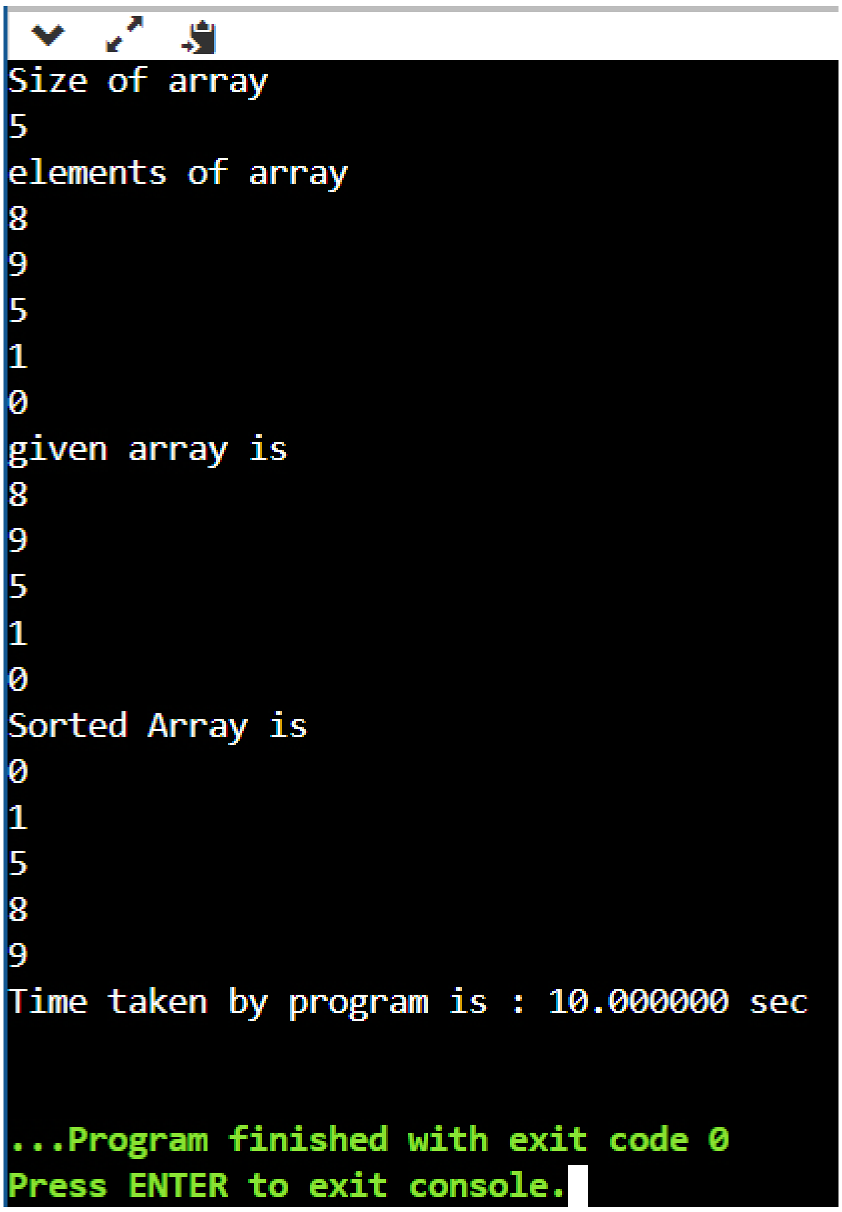
Results or Experimentation:

Merge Sort Output-



```
Size of array
5
elements of array
8
9
5
1
0
given array is
8
9
5
1
0
sorted array is
0
1
5
8
9
Time taken by program is : 53.000000 sec


...Program finished with exit code 0
Press ENTER to exit console.
```

Quick Sort Output-



```
Size of array
5
elements of array
8
9
5
1
0
given array is
8
9
5
1
0
Sorted Array is
0
1
5
8
9
Time taken by program is : 10.000000 sec


...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion:

Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets.
whereas
Quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets.