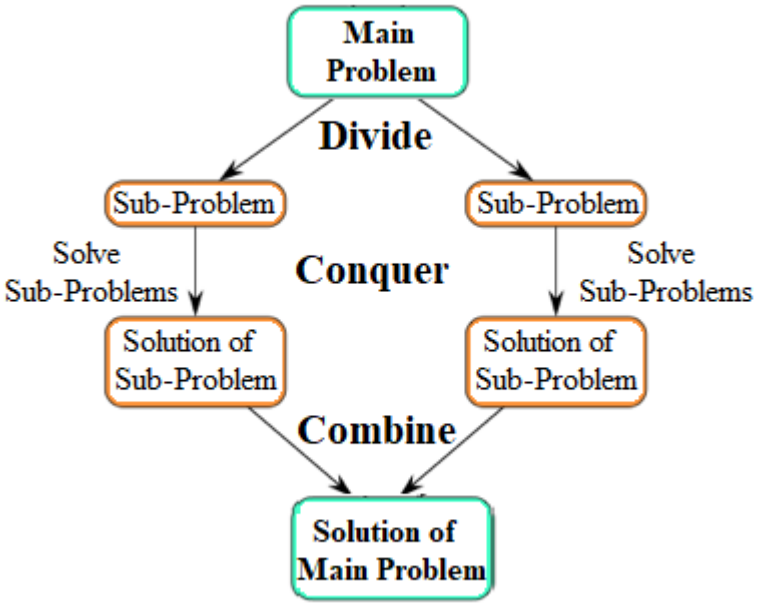
NAME :	APULKI DUBEY
EXP NO:	2
CLASS :	SE COMPS A
DATE OF PERFORMANCE:	14/02/23
AIM:	TO IMPLEMENT DIVIDE AND CONQUER TECHNIQUES
THEORY:	**Divide And Conquer** This technique can be divided into the following three parts: ● Divide: This involves dividing the problem into smaller sub-problems. ● Conquer: Solve sub-problems by calling recursively until solved. ● Combine: Combine the sub-problems to get the final solution of the whole problem. **1. QUICKSORT** Quicksort is a sorting algorithm. The algorithm picks a pivot element and rearranges the array elements so that all elements smaller than the picked pivot element move to the left side of the pivot, and all greater elements move to the right side. Finally, the algorithm recursively sorts the subarrays on the left and right of the pivot element.

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	$O(n^2)$

2.**MERGESORT**

Merge Sort is also a sorting algorithm. The algorithm divides the array into two halves, recursively sorts them, and finally merges the two sorted halves.



Time Complexity:

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n*logn)

ALGORITHM :

QUICKSORT

```
1. QUICKSORT (array A, start, end)
2. {
3.  1 if (start < end)
4.  2 {
5. 3 p = partition(A, start, end)
6. 4 QUICKSORT (A, start, p - 1)
7. 5 QUICKSORT (A, p + 1, end)
8. 6 }
9. }
```

MERGESORT

```
1. MERGE_SORT(arr, beg, end)
2.
3. if beg < end
4. set mid = (beg + end)/2
5. MERGE_SORT(arr, beg, mid)
6. MERGE_SORT(arr, mid + 1, end)
7. MERGE (arr, beg, mid, end)
8. end of if
9.
10. END MERGE_SORT
```

```cpp
#include <iostream>
#include <array>
#include <ctime>
#include <fstream>

using namespace std;
//  keeping the block size constant i.e 100 and array size 100000
const int limit = 100000;
const int block = 100;

// declaring the partition function
int partition(int arr[], int low, int high)
{

    int min = 0;
    // setting the pivot at staring element
    int pivot = arr[low];
    //  checking the count of numbers which are less than pivot
    for (int i = low + 1; i <= high; i++)
    {
        if (arr[i] < pivot)
            min++;
    }

    // setting the pivot index
    int pivotindex = low + min;
    swap(arr[pivotindex], arr[low]);
    // adjusting the smaller elements than pivot at left and
larger right of pivot
    for (int i = low, j = high; i<(pivotindex), j>(pivotindex);
i++, j--)

    {
        if (arr[i] > pivot && arr[j] < pivot)
        {
            swap(arr[i], arr[j]);

            i++;
            j--;
        }
        else if (arr[i] > pivot && arr[j] > pivot)
        {
            j--;
        }
        else if (arr[i] < pivot && arr[j] < pivot)
        {
            j--;
```

```
        }
        else
        {
            i++;
            j--;
        }
    }
    //   returning the pivot index
    return pivotindex;
}

void quicksort(int arr[], int low, int high)
{
    clock_t start, end;

    start =clock();
    // base case
    if (low >= high)
    {
        return;
    }
    // partition
    int p = partition(arr, low, high);

    // recursively calling the function to sort the left and
right part of array
    quicksort(arr, low, p - 1);
    quicksort(arr, p + 1, high);



}
void quick_sort (int arr[100000]) {
    int size = 0;
    int arr1[100];
     fstream f;
    f.open("Timesort.txt");

    for (int times = 0; times<limit/block; times++) {
        size+=block;

        int arr1 [size];
        for (int i = 0; i<size; ++i)

            arr1[i]=arr[i];
        // now our array is ready, we will perform quick sort
        // before that we will start the clock
```

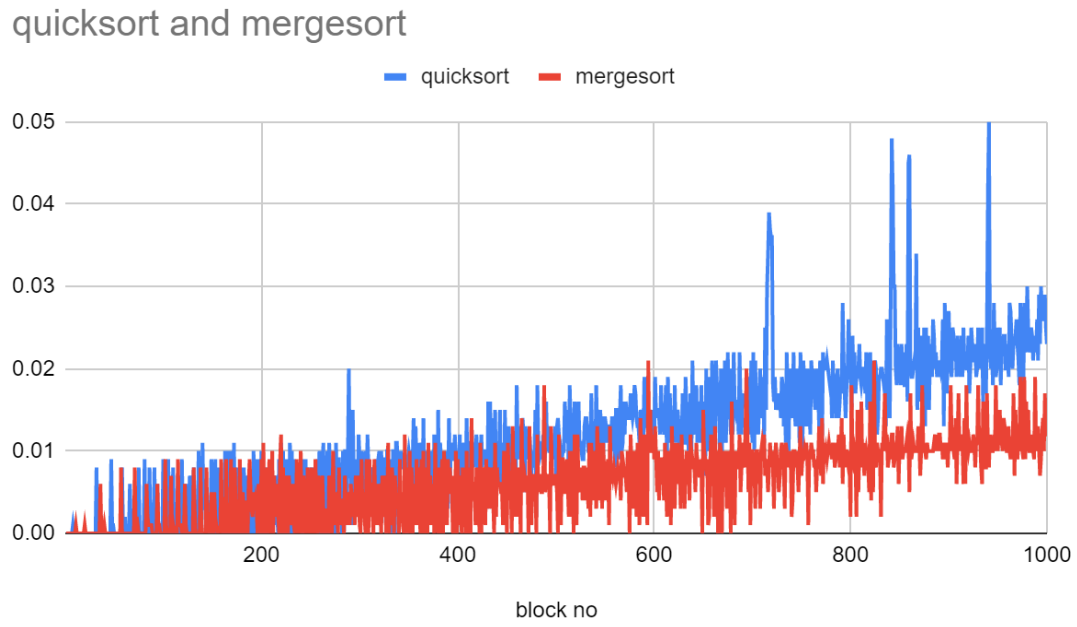
```
        clock_t t;
        t = clock();
        quicksort(arr1, 0, size-1);
            t = clock()-t;
            double time_taken = ((double)t)/CLOCKS_PER_SEC;
            // storing the result in a file
            cout<<times+1<<"\t\t"<<time_taken<<"\t\t"<<"\n";
          f<<time_taken;
          f<<"\n";
    }

}

int main () {
    int arr[100000];

    ifstream File;
    File.open("r.txt");
    int i=0;
    while(!File.eof())
    {
        File >> arr[i];
        i++;
    }


    File.close();

    quick_sort(arr);


    return 0;
}
```

```
#include <iostream>
using namespace std;

void print(int arr[10]){
    for(int i=0;i<10;i++)
    cout<<arr[i]<<" ";
}


void merge(int arr[10],int low,int mid,int high){
    int left=low;
    int right=mid+1;
```

```
    int temp[10];
    int k=0;
    int swap=0;
    while(left<=mid && mid+1<=high){
        if(arr[left]<=arr[right]){
            temp[k]=arr[left];
            left++;
            k++;
            swap++;

        }
        else{
            temp[k]=arr[right];
            right++;
            k++;
            swap++;
        }

        while(left==mid && right!=high){
        temp[k]=arr[right];
    }
     while(left!=mid && right==high){
        temp[k]=arr[left];
    }


    }
    print(temp);




}
void mergesort(int arr[10],int low ,int high ){
    if(low==high)
    return;
    int mid=(low+high)/2;
    mergesort(arr,low,mid);
    mergesort(arr,mid+1,high);
    merge(arr,low,mid,high);

}
```

```
int main()
{

 int arr[10];
 cout<<"Enter the values in array\n";
 int i=10;
 while(i--){
     cin>>arr[i];
 }
mergesort(arr,0,9);
    return 0;
}
```

OBSERVATION:



CONCLUSION:

From the graph for the given input, we saw that for larger input merge sort is better than quick sort as quick sort is slower for large input.This is because merge sort has a worst-case time complexity of O(nlogn), which is better than quicksort's worst-case time complexity of O(n^2)