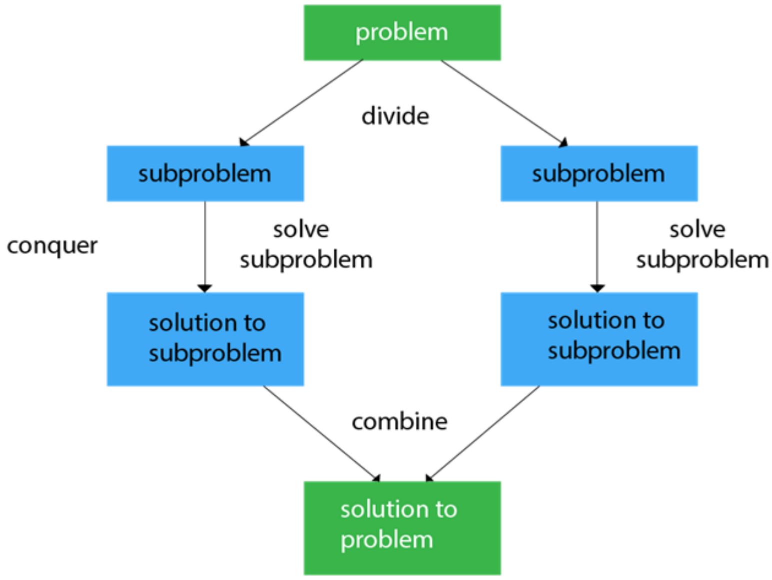
Name	**Ameya Dabholkar**
UID	**2021300023**
Subject	**Data Analysis Algorithm**
Experiment No	**2**

Aim-To implement the various sorting algorithms using divide andconquer technique.

Algorithm-

Both merge sort and quicksort employ a common algorithmic paradigm based on recursion. This paradigm, **divide-and-conquer**, breaks a problem into subproblems that are similar to the original problem, recursively solves the subproblems, and finally combines the solutions to the subproblems to solve the original problem. Because divide-and-conquer solves subproblems recursively, each subproblem must be smaller than the original problem, and there must be a base case for subproblems. You should think of a divide-and-conquer algorithm as having three parts:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. **Combine** the solutions to the subproblems into the solution for the original problem. You can easily remember the steps of a divide-and-conquer algorithm as *divide, conquer, and combine*. Here's how to view one step, assuming that each divide step creates two subproblems (though some divide-and-conquer algorithms create more than two):



Merge Sort –

1. Start
2. if the array is 0 or 1, it is already sorted, so return the array.
   Otherwise, split the array into two halves
3. Recursively sort the left and right halves using Merge Sort.

Merge the two sorted halves:
a. Initialize two pointers, one for each half.
b. While both pointers are within their respective halves, compare the values at the current positions of the pointers and add the smaller one to the sorted array.
c. When one pointer reaches the end of its half, add the remaining values from the other half to the sorted array.

4. Return the sorted array.

4. Stop

Quick Sort –

1. If the length of the array is 0 or 1, it is already sorted, so return the array.
2. Choose a pivot element from the array. This element will be used to partition the array into two parts.
3. Partition the array into two parts:

a. Rearrange the elements such that all elements less than the pivot are on the left, and

4. all elements greater than the pivot are on the right. The pivot itself should be in its final sorted position.

b. Return the index of the pivot.

5. Recursively sort the left and right partitions:

a. Call Quick Sort on the left partition (elements less than the pivot).

b. Call Quick Sort on the right partition (elements greater than the pivot).

6. Return the sorted array.
7. stop

Code-

```
#include <stdio.h>
#include<stdlib.h>
#include<time.h>
void merge(int mrgsort[], int l, int m, int r)
{
   int i, j, k;
   int n1 = m - l + 1;
   int n2 = r - m;
   int Left[n1], Right[n2];
   for (i = 0; i < n1; i++)
      Left[i] = mrgsort[l + i];
   for (j = 0; j < n2; j++)
      Right[j] = mrgsort[m + 1 + j];
   i = 0;
   j = 0;
   k = l;
   while (i < n1 && j < n2) {
      if (Left[i] <= Right[j]) {
         mrgsort[k] = Left[i];
         i++;
      } else {
         mrgsort[k] = Right[j];
         j++;
      }
      k++;
   }
   while (i < n1) {
      mrgsort[k] = Left[i];
      i++;
      k++;
   }
   while (j < n2) {
      mrgsort[k] = Right[j];
```

```
        j++;
        k++;
    }
}
void mergesort(int mrgsort[], int count, int n)
{
    if (count < n) {
        int temp = count + (n - count) / 2;
        mergesort(mrgsort, count, temp);
        mergesort(mrgsort, temp + 1, n);
        merge(mrgsort, count, temp, n);
    }
}
void display(int mrgsort[], int quicksort[], int n)
{
    for(int i=0; i<n; i++) {
        printf("%d\t%d\n",mrgsort[i],quicksort[i]);
    }
}
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
int partition(int array[], int low, int high)
{
    int pivot = array[high];
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (array[j] <= pivot) {
            i++;
            swap(&array[i], &array[j]);
        }
    }
    swap(&array[i + 1], &array[high]);
    return (i + 1);
}

void quickSort(int array[], int low, int high)
{
    if (low < high) {
        int pi = partition(array, low, high);
        quickSort(array, low, pi - 1);
        quickSort(array, pi + 1, high);
    }
}
void printArray(int array[], int size)
{
    for (int i = 0; i < size; ++i) {
        printf("%d  ", array[i]);
    }
    printf("\n");
}
void main()
```

```
{
        int n=0;
        for(int j=0; j<(10000/100); j++)
        {
           n=n+100;
           int num[n];
           int mrgsort[n];
           int quicksort[n];
           clock_t start_t, end_t;
           double total_t;
           for(int i=0; i<n; i++) {
              num[i]=rand() % 10;
              mrgsort[i]=num[i];
              quicksort[i]=num[i];
           }
           printf("%d\t",n);
           start_t = clock();
           mergesort(mrgsort, 0, n - 1);
           end_t = clock();
           total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
           printf("%f\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("%f\n", total_t  );
           //display(mrgsort, quicksort, n);
        }
}
```

Conclusion-

Merge sort is more efficient as its worst-case time complexity is O(logn) while in case of quick sort, it remains constant throughout all operations as we can see from its graph which is linear in nature.