Course Name:	**Analysis of Al-gorithms**	**Semester:**	**IV**
Date of Performance:	**12 / 02 / 2024**	**Batch No:**	**EXCP B1**
Faculty Name:	**Prof. Payal Varangoankar**	**Roll No:**	**16014022096**
Faculty Sign & Date:		**Grade/Marks:**	

Experiment No: 3

Title: Merge Sort Analysis /Quick Sort Analysis.

Aim and Objective of the Experiment:
To learn the divide and conquer strategy of solving the problems of different types

COs to be achieved:
CO2: Describe various algorithm design strategies to solve different problems.

Apparatus / Software tools used:

Theory:

Historical Profile:
Quicksort and merge sort are divide-and-conquer sorting algorithms in which division is dynamically carried out. They are one the most efficient sorting algorithms.

New Concepts to be learned:
Number of comparisons, Application of algorithmic design strategy to any problem, Classical problem solving vs Divide-and-Conquer problem solving.

Link for Merge Sort and Quick Sort in Virtual Labs
https://ds1-iiith.vlabs.ac.in/List%20of%20experiments.html

Code:

QUICK SORT:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int A[], int left, int right, int *iterations) {
    int pivot = A[left];
    int lo = left + 1;
    int hi = right;

    while (lo <= hi) {
        while (A[hi] > pivot)
            hi--;
        while (lo <= hi && A[lo] <= pivot)
            lo++;
        if (lo <= hi) {
            swap(&A[lo], &A[hi]);
            (*iterations)++;
        }
    }

    swap(&A[left], &A[hi]);
    return hi;
}

void quicksort(int A[], int left, int right, int *iterations) {
    if (left < right) {
        int q = partition(A, left, right, iterations);
        quicksort(A, left, q - 1, iterations);
        quicksort(A, q + 1, right, iterations);
    }
}

int main() {
    int n = 10;
    int arr[n];
    clock_t t1, t2;
```

```
    printf("Original array: ");
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
        printf("%d ", arr[i]);
    }
    printf("\n");

    t1 = clock();
    int iterations = 0;
    quicksort(arr, 0, n - 1, &iterations);
    t2 = clock();


    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    printf("Number of iterations: %d\n", iterations);

    double t = ((double)(t2 - t1)) / CLOCKS_PER_SEC;
    printf("TIME : %f \n", t);
    return 0;
}
```

MERGE SORT:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>

int merge(int A[], int p, int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int L[n1 + 1], R[n2 + 1];
    int iterations = 0;

    for (int i = 0; i < n1; i++)
        L[i] = A[p + i];
    for (int j = 0; j < n2; j++)
        R[j] = A[q + 1 + j];
```

```
    L[n1] = INT_MAX;
    R[n2] = INT_MAX;

    int i = 0, j = 0;
    for (int k = p; k <= r; k++) {
        iterations++;
        if (L[i] <= R[j]) {
            A[k] = L[i];
            i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
    return iterations;
}

int mergeSort(int A[], int p, int r) {
    int iterations = 0;
    if (p < r) {
        int q = (p + r) / 2;
        iterations += mergeSort(A, p, q);
        iterations += mergeSort(A, q + 1, r);
        iterations += merge(A, p, q, r);
    }
    return iterations;
}

int main() {
    int n = 10;
    int arr[n];
    clock_t t1, t2;

    printf("Original array: ");
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
        printf("%d ", arr[i]);
    }
    printf("\n");

    t1 = clock();
    int iterations = mergeSort(arr, 0, n - 1);
    t2 = clock();
```

```
    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    printf("Number of iterations: %d\n", iterations);

    double t = ((double)(t2 - t1)) / CLOCKS_PER_SEC;
    printf("TIME : %f \n", t);

    return 0;
}
```

Stepwise-Procedure / Algorithm:

Algorithm Recursive Quick Sort:

void quicksort(Integer A[ ], Integer left, Integer right)
//sorts A[left.. right] by using partition() to partition A[left.. right], and then //calling itself // twice to sort the two subarrays.
{ IF (left < right) then
{ q = partition(A, left, right);
quicksort(A, left, q–1);
quicksort(A, q+1, right);
}
}

Integer partition(integer AT[], Integer left, Integer right)
//This function rearranges A[left..right] and finds and returns an integer q, such that A[left], ..., //A[q–1] <~□pivot, A[q] = pivot, A[q+1], ..., A[right] > pivot, where pivot is the first element of //a[left…right], before partitioning.
{
pivot = A[left]; lo = left+1; hi = right;
WHILE (lo ≤ hi)
{ WHILE (A[hi] > pivot) hi = hi – 1;
WHILE (lo ≤ hi and A[lo] <~pivot) lo = lo + 1;
IF (lo ≤ hi) then swap(A[lo], A[hi]);
}
swap(pivot, A[hi]);
RETURN hi;
}

The space complexity of Quick Sort:

Derivation of best case and worst-case time complexity (Quick Sort)
Algorithm Merge Sort
MERGE-SORT (A, p, r)
// To sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT (A, //1, n). Array A and indices p, q, r such that p ≤ q ≤ r and sub array A[p .. q] is sorted and sub array //A[q + 1 .. r] is sorted. By restrictions on p, q, r, neither sub array is empty.
//OUTPUT: The two sub arrays are merged into a single sorted subarray in A[p .. r].

```
    IF p < r                                   // Check for base case
       THEN q = FLOOR [(p + r)/2]              // Divide step
             MERGE (A, p, q)                   // Conquer step.
             MERGE (A, q + 1, r)               // Conquer step.
             MERGE (A, p, q, r)                // Conquer step.

MERGE (A, p, q, r)
{
     n1 ← q − p + 1
     n2 ← r − q
     Create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
     FOR i ← 1 TO n1
          DO L[i] ← A[p + i − 1]
     FOR j ← 1 TO n2
          DO R[j] ← A[q + j ]
     L[n1 + 1] ← ∞
     R[n2 + 1] ← ∞
   i ← 1
   j ← 1
   FOR k ← p TO r
        DO IF L[i ] ≤ R[ j]
            THEN A[k] ← L[i]
                 i ← i + 1
            ELSE A[k] ← R[j]
                 j ← j + 1

}
```

Output:

QUICK SORT:

```
Enter value of n: 5
Original array: 3 6 7 5 3
Sorted array: 3 3 5 6 7
Number of iterations: 1
TIME : 0.000002
```

```
Enter value of n: 5
Original array: 83 86 77 15 93
Sorted array: 15 77 83 86 93
Number of iterations: 1
TIME : 0.000002
```

MERGE SORT:

```
Enter value of n: 5
Original array: 3 6 7 5 3
Sorted array: 3 3 5 6 7
Number of iterations: 12
TIME : 0.000001
```

```
Enter value of n: 5
Original array: 83 86 77 15 93
Sorted array: 15 77 83 86 93
Number of iterations: 12
TIME : 0.000002
```

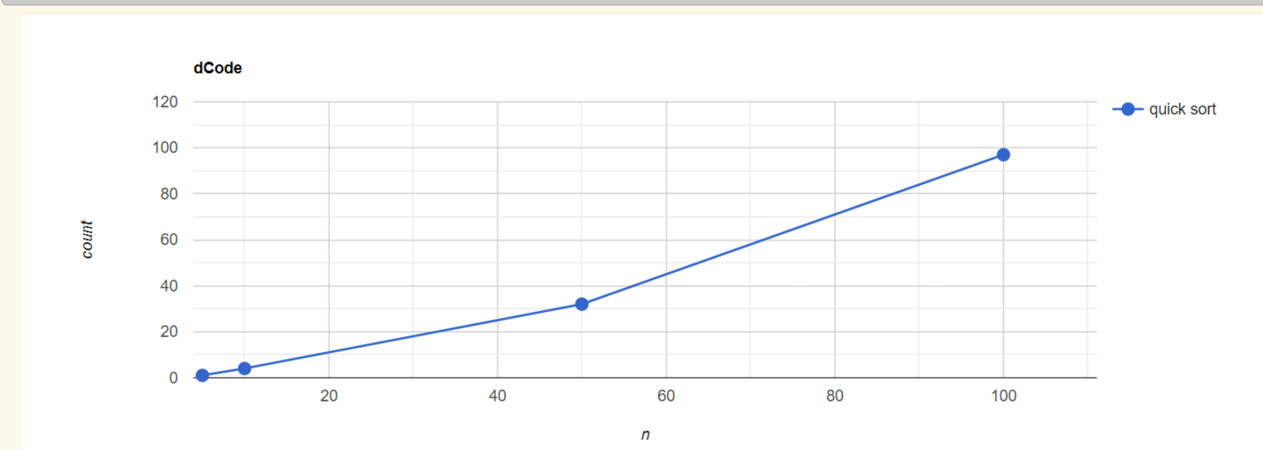
Observation Table:

QUICK SORT:

	abscissa x →	ordinate y or f(x) ↑
1	5	1
2	10	4
3	50	32
4	100	97
5	...	...
6	...	...

★ ORIGINS ◉ AUTOMATICALLY CALCULATED
○ SET TO (0,0)
★ HORIZONTAL X-AXIS NAME n
★ VERTICAL Y-AXIS NAME count
★ LEGEND quick sort

dCode



OK

MERGE SORT:

	abscissa x →	ordinate y or f(x) ↑
1	5	12
2	10	34
3	50	286
4	100	672
5	...	...
6	...	...

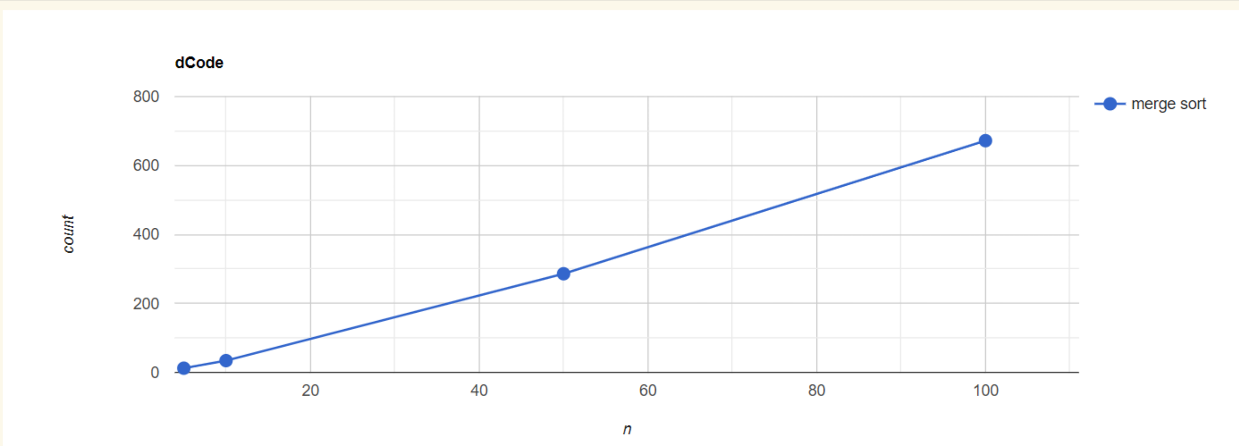
★ ORIGINS ◉ AUTOMATICALLY CALCULATED
○ SET TO (0,0)

★ HORIZONTAL X-AXIS NAME: n

★ VERTICAL Y-AXIS NAME: count

★ LEGEND: merge sort

dCode



OK

Calculation:

MERGE SORT:

```
Algorithm Merge (low, mid, high)
  //a[low:high] is gobal array
  // subset in a[low:mid] and in a[mid+1,high]. The goal
  // is to merge these two. b[] is auxillary global array
{
  h: high ; i:= low ; j := mid + 1
  While ((h< mid) and (j≤ high)) do
   { if ( a[h] ≤ a[j])then
        {
           b[i] = a[h] ; h=h+1 ;
        }
     else{
           b[i] = a[j] ; j=j+1 ;
        }
           i : i+1
   }
   if (h > mid) then
       for k :=j to high do
        {
           b [i]= a[k]; i=i+1
        }
   else
       for k :=h to mid do
        {
           b[i]:= a[k]; i=i+1 ;
        }
   for k: low to high do a[k] := b[k]
}
```

Algorith Merge Sort (low, high)
// a[low:high] is a global array
// Small(P) is true if there is only one element to sort
in this case the list is already sorted.
{
if (low < high) then
{
// Divide p into subproblems.
// Find where to split the set
mid:= [(low+high)/2] ;
// Sove the subproblem
MergeSort (low, mid)
Merge sort (mid+1, high)
// Combine solution
Merge (low, mid, high)

Master Method

$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^c)$

a = no of time called
b = no of time breaking
Θ = wd outside

$T(n) = 2T\left(\frac{n}{2}\right) + n$

$a=2 \quad b=2 \quad c=1$

$\log_b a = \log_2 2 = 1 \equiv c$

$\therefore T(n) = \Theta(n \log n)$

Space complexity $O(n)$

QUICK SORT:

```
Algorithm Partition (a,m,p)
// Within a[m], a[m+1], ... a[p-1] elements are
// rearranged in such a manner that if initially
// t = a[m], then after completion a[q] = t for
// some q between m and p-1, a[k] ≤ t for
// m ≤ k ≤ q and a[k] ≥ t for q < k < p.
// q is returned. Set a[p] = ∞.

{
  v := a[m]; i = m ; j := p;
  repeat
  {
     repeat
        i = i+1;
     until [a[i] ≥ v];

     repeat
        j = j+1;
     until (a[j] ≤ v);

     if (i<j) then interchange (a,i,j)
  }
     until (i>j);
  am := a[j]; a[j] := v; return j;
}
Algorithm Interchange (a,i,j)
{ p := a[i];
  a[i] := a[j]; a[j] = p;
}
```

```
Algorithm QuickSort(p, q)
    // sorts elements a[p]...a[q] in global array a[1:n]
{
    if (p<q) then
    {
        // divide P into two subproblems:
            j := Partition(a, p, q+1)

            QuickSort(p, j-1)
            QuickSort(j+1, q)
        // There is no need for combining solutions.
    }
}
```

Master Method

$T(n) = aT\left(\frac{n}{b}\right) + \theta(n^c)$

a = no of time called
b = no of time breaking
θ = wd outside

$T(n) = 2T\left(\frac{n}{2}\right) + n$

$a = 2 \quad b = 2 \quad c = 1$

$\log_b a = \log_2 2 = 1 \equiv c$

$\therefore T(n) = \theta(n \log n)$

Space complexity $O(\log n)$

Post Lab Subjective/Objective type Questions:

Conclusion:

We have successfully implemented quick sort algorithm and merge sort algorithm in the above experiment and came to a conclusion

QUICK SORT:

	TIME COMPLEXITY	SPACE COMPLEXITY
Best Case	O(n log n)	O(log n)
Average Case	O(n log n)	O(log n)
Worst Case	O(n^2)	O(n)

MERGE SORT:

TIME COMPLEXITY	SPACE COMPLEXITY
O(n log n)	O(n)

Signature of faculty in-charge with Date: