Rajan Gautam

19BCP101

Div. II, CE 19

SOT, PDPU

Pandit Deendayal Petroleum University

School of Technology

Design & Analysis of Algorithm (20CP209P)

B. Tech - Computer Science & Engineering (Sem-IV)

Table of Contents

Lab 2 Assignment: Comparison of Quick Sort & Merge Sort.

AIM: To write a C/C++ Program to implement Merge Sort & Quick Sort.

ALGORITHMS:

1. Merge Sort Algorithm (Pseudocode)

```
INSERTION-SORT (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
                R[j] ← A[q + j]
        L[n1 + 1] ← infinite
        R[n2 + 1] ← infinite
        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
MERGE-SORT(A, p, r)
        if p < r
                then q ← L(p + r) / 2⌋
                        MERGE-SORT(A, p, q)
                        MERGE-SORT(A, q + 1, r)
                        MERGE-SORT(A, p, q, r)
```

2. Quick Sort Algorithm (Pseudocode)

```
PARTITION(A, p, r)
        X = A[r]
        I = p – 1
        For j = p to r – 1
                if A[j] ≤ x
                        i = i + 1
                        exchange A[i] with A[j]
        exchange A[i + 1] with A[r]
        return i + 1
QUICKSORT (A, p, r)
        if p < r
                q = PARTITION (A, p, r)
                QUICKSORT (A, p, q – 1)
                QUICKSORT (A, q + 1, r)
```


CODE:

```c
/* ------------------------- 19BCP101 -----------------------*/
/* ---------------------- Rajan Gautam ---------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>          // For Time Calculation

void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;

    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
```

```
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[], if there are any
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[], if there are any
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for large l and h
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

/* FOR QUICK SORT */

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

int partition (int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
```

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

void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}


int main()
{
    printf("<--------------- Sorting --------------->\n\n");

    int n = 1000, it = 0;
    double time1[20], time2[20];        // To store the time values

    printf(" Array \t Merge(s) \t Quick(s) \n\n");

    while(it++ < 10)
    {
        long int a[n], b[n];
        for (int i = 0; i < n ; i++)
        {
            // Generating Random Integer Array for each algorithm

            a[i] = (rand() % n);
            b[i] = (rand() % n);
        }


        // For time calculation
        clock_t start, end;


        // For Merge Sort Algorithm
        start = clock();
        mergeSort(a, 0, n-1);
        end = clock();

        time1[it] = ((double)(end - start)/CLOCKS_PER_SEC);


        // For Quick Sort Algorithm
        start = clock();
        quickSort(b, 0, n-1);
        end = clock();

        time2[it] = ((double)(end - start)/CLOCKS_PER_SEC);

```

```
        // Printing the table of array size, time taken by Bubble Sort and Insertion Algorithm
        printf(" %d \t %f \t %f\n", n, time1[it], time2[it]);


        // Incrementing the value of n by 1000
        n += 1000;
    }
    return 0;
}
```

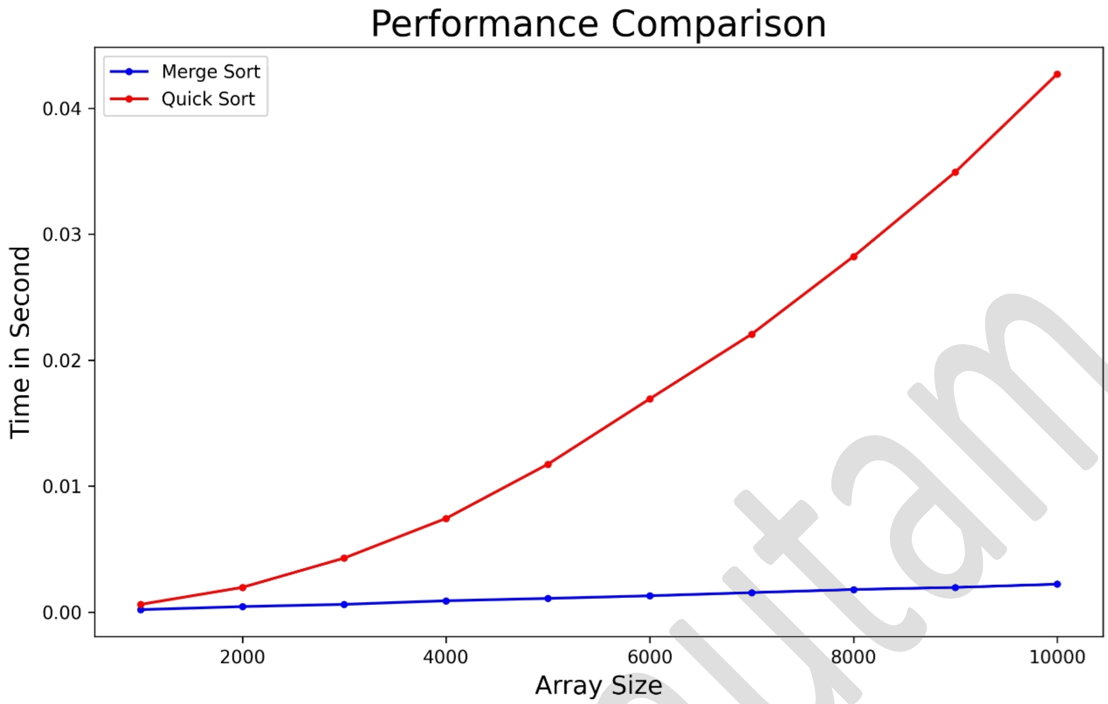
OUTPUT:

```
<--------------- Sorting --------------->

Array     Merge(s)          Quick(s)

1000      0.000186          0.000600
2000      0.000430          0.001953
3000      0.000606          0.004280
4000      0.000894          0.007431
5000      0.001078          0.011735
6000      0.001287          0.016924
7000      0.001538          0.022059
8000      0.001786          0.028225
9000      0.001953          0.034935
10000     0.002208          0.042718
```

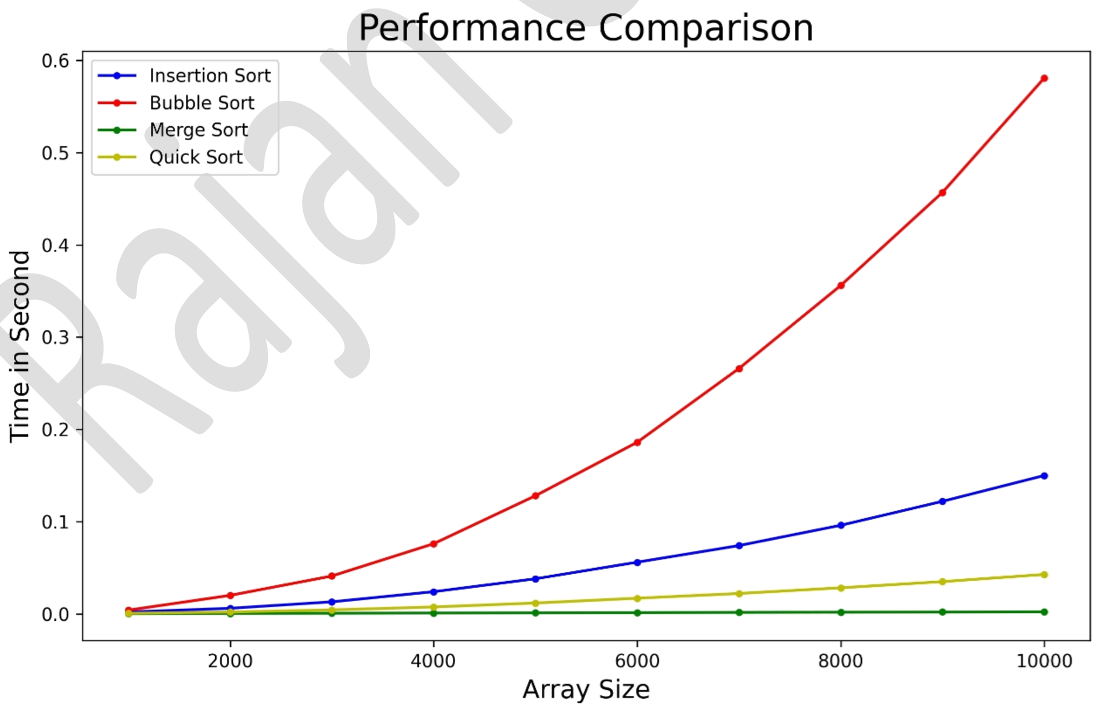
ALL OUTPUT:

	Array	Bubble	Insertion	Merge	Quick
0	1000	0.004	0.002	0.000186	0.000600
1	2000	0.020	0.006	0.000430	0.001953
2	3000	0.041	0.013	0.000606	0.004280
3	4000	0.076	0.024	0.000894	0.007431
4	5000	0.128	0.038	0.001078	0.011735
5	6000	0.186	0.056	0.001287	0.016924
6	7000	0.266	0.074	0.001538	0.022059
7	8000	0.356	0.096	0.001786	0.028225
8	9000	0.457	0.122	0.001953	0.034935
9	10000	0.581	0.150	0.002208	0.042718

Performance Comparison of Merge Sort and Quick Sort



Performance Comparison of All Four Sorting Algorithms



Link: https://github.com/rgautam320/Design-and-Analysis-of-Algorithm-Lab/tree/master/Lab_2_Sorting