Experiment Number: 3

Aim: - Write a program to compare Quick Sort and Merge Sort.

Problem Statement: - Doing the comparative analysis of Quick Sort and Merge Sort on the basis of comparisons required for sorting list of large value of n.

Theory:-

Quick Sort

Quicksort, like merge sort, is a divide-and-conquer recursive algorithm. The basic divide-and-conquer process for sorting a sub-array S [p…r] is summarized in the following three easy steps:

Divide:

Partition S [p...r] into two different sub-arrays S [p...q-1] and S [q+1...r] such that each element of S [p..q-1] is less than or equal to S[q], which is, in turn, less than or equal to each element of S [q+1...r]. Compute the index q as part of this partitioning procedure

Conquer:

Sort the two sub-arrays S[p...q-1] and S[q+1..r] by recursive calls to quicksort.

Combine:

Since the sub-arrays are sorted in place, no work is needed to combing them. The entire array S is now sorted.

Algorithm: -

```
quicksort (a, low, high)
    {
    if ( high > low )
        {
        pivot = partition( a, low, high );
        quicksort( a, low, pivot-1 );
        quicksort( a, pivot+1, high );
        }
    }
partition (a, low, high)
    {
    v=a[low]; i=a[m]; j=high;
    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)
    a[m]=a[j]; a[j]=v; return j;
    }
interchange (a, i, j)
    {
    p=a[i];   a[i]=a[j];  a[j]=p;
    }
```

While calculating the complexity for Quick Sort we count only the number of element comparisons. The recurrence equation of the Quick Sort can be given as,

$$T(n) = T(i) + T\ (n - i - 1) + n$$

Best case:

In best case of the Quick Sort the pivot element always partition the array into two equal size sub-arrays. Hence we can rewrite the recurrence equation as follows,

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

If we solve above recurrence equation using Master Method then we have,

$$\boldsymbol{T(n) = O\ (n \log n)}$$

Worst Case:

In the worst case the partitioning algorithm partition the array in such a way that on one of the side of array there are zero elements while on the other side of pivot there are (n - 1) elements. Hence we can rewrite the recurrence equation as follows:

$$T(n) = T\ (n - 1) + n$$

We can write as,

$$T(n-1) = T\ (n-2) + (n-1)$$
$$T(n-2) = T\ (n-3) + (n-2)$$
$$T(n-3) = T\ (n-4) + (n-3)$$
$$\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots$$
$$T(2) = T\ (1) + 2$$

Adding all above equations we get.

$$T(n) = T\ (1) + (2 + 3 + 4 + \cdots + n)$$
$$\therefore T(n) = 1 + \sum_{j=2}^{n} j$$
$$\therefore T\ (n) = 1 + \frac{n(n-1)}{2}$$
$$\therefore \boldsymbol{T(n) = O\ (n^2)}$$

Average case:

The average value of T (i) is 1/n times the sum of T (0) through T (n-1). Hence we can rewrite the recurrence equation as follows:

$$T(n) = \frac{2}{n}\left(\sum_{j=0}^{n-1} T(j)\right) + n$$

Multiplying both the sides by 'n' we get

$$nT(n) = 2\left(\sum_{j=0}^{n-1} T(j)\right) + n^2$$

To remove the summation we rewrite the equation for (n -1)

$$(n-1)T(n-1) = 2\left(\sum_{j=0}^{n-2} T(j)\right) + (n-1)^2$$

Subtracting above equation from the previous equation of nT(n) we get,

$$nT(n) - (n-1)T(n-1) = 2\ T(n-1) + 2n - 1$$
$$\therefore\ nT(n) = (n+1)T(n-1) + 2n$$

Dividing both the sides by n(n+1) we get,

$$\frac{\mathrm{T(n)}}{(n+1)} = \frac{\mathrm{T(n-1)}}{\mathrm{n}} + \frac{2}{n+1}$$

We can write as,

$$\therefore \frac{\mathrm{T(n-1)}}{(n)} = \frac{\mathrm{T(n-2)}}{(\mathrm{n-1})} + \frac{2}{n}$$

$$\therefore \frac{\mathrm{T(n-2)}}{(n-1)} = \frac{\mathrm{T(n-3)}}{(\mathrm{n-2})} + \frac{2}{(n-1)}$$

$$\therefore \frac{\mathrm{T(2)}}{(3)} = \frac{\mathrm{T(1)}}{(2)} + \frac{2}{3}$$

Adding all above terms we get,

$$\therefore \frac{\mathrm{T(n)}}{(n+1)} = \frac{\mathrm{T(1)}}{(2)} + 2\sum_{j=3}^{n+1} \frac{1}{j}$$

$$\therefore T(n) = (n+1)\left(\frac{1}{2} + \log n\right)$$

$$\therefore \boldsymbol{T(n) = O\ (n\ log\ n)}$$

Merge Sort

The merge sort algorithm is based on the classical divide and conquer paradigm. It operates as follows:

DIVIDE:

Partition the n-element array into two sub-arrays of (n/2) elements each.

CONQUER:

Sort the two sub-arrays recursively using merge sort.

COMBINE:

Merge the two sorted subsequences of size n/2 each to produce the sorted sequence consisting of n elements**.**

Algorithm:

```
merge_sort(a,low,high)
        {
        if(high > low)
                {
                mid= (low + high)/2 ;
                merge_sort(a, low, mid);
                merge_sort(a, mid+1, high);
                merge(a, low, mid, high);
                }
        }
merge (a, low, mid, high)
        {
        n1 = mid-low+1;
        n2 = high-mid;
        for i = 1 to n1
                L[i] = a[low+i-1];
        for j = 1 to n2
                R[j] = a[mid+j];
        L[n1+1]= R[n2+1] = infinity;
        i = j = 1;
        for k = low to high
                {
                do if L[i]≤ R[j]
                        then a[k] = L[i]; i = i +1;
```

```
else
        a[k] = R[j]; j = j+1;
}
```

If there is only one element in the array then the complexity of merge sort is constant and it is given by T (1) = 1.

If there more than 1 element in the array then the array of size 'n' is divided into sub-arrays of size 'n/2' each, and then it is sorted recursively calling merge_sort. At the last to combine two sub-arrays of size 'n/2',the complexity is linear and is given as 'n'. Hence we can write the recurrence equation for merge sort when n>1 as follows:

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

Dividing on both sides by 'n' we get,

$$\frac{T(n)}{n} = \left(\frac{T\left(\frac{n}{2}\right)}{\left(\frac{n}{2}\right)}\right) + 1$$

As n is in the powers of 2 i.e. $n = 2^k$ where k=1,2,3, …
We can write,

$$\frac{T\left(\frac{n}{2}\right)}{\left(\frac{n}{2}\right)} = \left(\frac{T\left(\frac{n}{4}\right)}{\left(\frac{n}{4}\right)}\right) + 1$$

$$\frac{T\left(\frac{n}{4}\right)}{\left(\frac{n}{4}\right)} = \left(\frac{T\left(\frac{n}{8}\right)}{\left(\frac{n}{8}\right)}\right) + 1$$

$$\frac{T\left(\frac{n}{8}\right)}{\left(\frac{n}{8}\right)} = \left(\frac{T\left(\frac{n}{16}\right)}{\left(\frac{n}{16}\right)}\right) + 1$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

Adding all above equations we get,

$$\frac{T(n)}{n} = \frac{T(1)}{1} + \log n$$

$$\therefore T(n) = n(\,1 + \log n)$$

$$\therefore \boldsymbol{T(n) = O(n \log n)}$$

Code:

```cpp
// SE3-C 52 Raj Singh
#include <bits/stdc++.h>
#include <chrono>
using namespace std;
using namespace std::chrono;

vector<int> qS_data(2);
int mS_Count = 0;
int partition(vector<int> &a, int l, int r)
{
    int pivot = a[l];
    int i = l + 1;
    int j = r;
    while (i < j)
    {
        while (a[i] <= pivot)
        {
            i++;
            qS_data[0]++;
        }
        while (a[j] > pivot)
        {
            qS_data[0]++;
            j--;
        }
        if (i < j)
        {
            swap(a[i], a[j]);
            qS_data[1]++;
        }
    }
    swap(a[j], a[l]);
    qS_data[1]++;
    return j;
}

   void quickSort(vector<int> &a, int l, int r)
   {
       if (l < r)
       {
           int pivot = partition(a, l, r);
           quickSort(a, l, pivot - 1);
           quickSort(a, pivot + 1, r);
       }
   }
```

```cpp
void merge(vector<int> &b, int l, int mid, int r)
{
    int i = l, j = mid, idx = 0;
    int len_ = r - l + 1;
    vector<int> mergeArr(len_);
    while (i < mid && j <= r)
    {
        if (b[i] < b[j])
        {
            mergeArr[idx++] = b[i++];
            mS_Count++;
        }
        else
        {
            mergeArr[idx++] = b[j++];
            mS_Count++;
        }
    }

    while (i < mid)
    {
        mergeArr[idx++] = b[i++];
    }

    while (j <= r)
    {
        mergeArr[idx++] = b[j++];
    }

    for (int k = 0; k < len_; k++)
    {
        b[l + k] = mergeArr[k];
    }
}

void mergeSort(vector<int> &b, int l, int r)
{
    if (l < r)
    {
        int mid = l + (r - l + 1) / 2;
        mergeSort(b, l, mid - 1);
        mergeSort(b, mid, r);
        merge(b, l, mid, r);
    }
}
```

```
int main(void)
{
    // For Average Case
    vector<int> a{20, 15, 17, 24, 13, 65, 1};

    // For Best Case
    vector<int> b{1, 5, 13, 14, 17, 26, 41};

    // For Worst Case
    vector<int> c{75, 55, 50, 44, 37, 26, 11};

    for (int i = 0; i <= 2; i++)
    {
        qS_data.clear();
        mS_Count = 0;

        // Average Case
        if (i == 0)
        {
            auto start = high_resolution_clock::now();
            quickSort(a, 0, a.size() - 1);
            auto end = high_resolution_clock::now();
            auto time_taken = duration_cast<nanoseconds>(end - start);
            cout << "Quick Sort (Average Case) : ";
            for (auto &n : a)
                cout << n << " ";
            cout << "\n";
            cout << "Swaps : " << qS_data[1] << "\n";
            cout << "Comparisons : " << qS_data[0] << "\n";
            cout << "Time taken to run Quick Sort in average case : "
                 << time_taken.count() << " ns" << endl;
            cout << "\n";
            a = {20, 15, 17, 24, 13, 65, 1};
            start = high_resolution_clock::now();
            mergeSort(a, 0, a.size() - 1);
            end = high_resolution_clock::now();
            time_taken = duration_cast<nanoseconds>(end - start);
            cout << "Merge Sort (Average Case): ";
            for (auto &n : a)
                cout << n << " ";
            cout << "\n";
            cout << "Comparisons : " << mS_Count << "\n";
            cout << "Time taken to run Merge Sort in average case : "
                 << time_taken.count() << " ns" << endl;
            cout << "\n";
        }
```

```
// Best Case
else if (i == 1)
{
    auto start = high_resolution_clock::now();
    quickSort(b, 0, b.size() - 1);
    auto end = high_resolution_clock::now();
    auto time_taken = duration_cast<nanoseconds>(end - start);
    cout << "Quick Sort (Best Case) : ";
    for (auto &n : b)
        cout << n << " ";
    cout << "\n";
    cout << "Swaps : " << qS_data[1] << "\n";
    cout << "Comparisons : " << qS_data[0] << "\n";
    cout << "Time taken to run Quick Sort in best case : "
         << time_taken.count() << " ns" << endl;
    cout << "\n";
    b = {1, 5, 13, 14, 17, 26, 41};
    start = high_resolution_clock::now();
    mergeSort(b, 0, b.size() - 1);
    end = high_resolution_clock::now();
    time_taken = duration_cast<nanoseconds>(end - start);
    cout << "Merge Sort (Best Case): ";
    for (auto &n : b)
        cout << n << " ";
    cout << "\n";
    cout << "Comparisons : " << mS_Count << "\n";
    cout << "Time taken to run Merge Sort in best case : "
         << time_taken.count() << " ns" << endl;
    cout << "\n";
}
```

```
// Worst Case
else
{
    auto start = high_resolution_clock::now();
    quickSort(c, 0, c.size() - 1);
    auto end = high_resolution_clock::now();
    auto time_taken = duration_cast<nanoseconds>(end - start);
    cout << "Quick Sort (Worst Case) : ";
    for (auto &n : c)
        cout << n << " ";
    cout << "\n";
    cout << "Swaps : " << qS_data[1] << "\n";
    cout << "Comparisons : " << qS_data[0] << "\n";
    cout << "Time taken to run Quick Sort in worst case : "
         << time_taken.count() << " ns" << endl;
    cout << "\n";
```

```
            c = {75, 55, 50, 44, 37, 26, 11};
            start = high_resolution_clock::now();
            mergeSort(c, 0, c.size() - 1);
            end = high_resolution_clock::now();
            time_taken = duration_cast<nanoseconds>(end - start);
            cout << "Merge Sort (Worst Case): ";
            for (auto &n : c)
                cout << n << " ";
            cout << "\n";
            cout << "Comparisons : " << mS_Count << "\n";
            cout << "Time taken to run Merge Sort in worst case : "
                 << time_taken.count() << " ns" << endl;
            cout << "\n";
        }
    }

    return 0;
}
```

Output:

```
ayusi@LAPTOP-64SMCRE0 MINGW64 ~/OneDrive/Desktop/Data/RajDoc
$ ./a.exe
Quick Sort (Average Case) : 1 13 15 17 20 24 65
Swaps : 6
Comparisons : 9
Time taken to run Quick Sort in average case : 0 ns

Merge Sort (Average Case): 1 13 15 17 20 24 65
Comparisons : 13
Time taken to run Merge Sort in average case : 0 ns

Quick Sort (Best Case) : 1 5 13 14 17 41 26
Swaps : 12
Comparisons : 29
Time taken to run Quick Sort in best case : 0 ns

Merge Sort (Best Case): 1 5 13 14 17 26 41
Comparisons : 9
Time taken to run Merge Sort in best case : 0 ns

Quick Sort (Worst Case) : 11 26 44 37 50 55 75
Swaps : 18
Comparisons : 49
Time taken to run Quick Sort in worst case : 0 ns

Merge Sort (Worst Case): 11 26 37 44 50 55 75
Comparisons : 11
Time taken to run Merge Sort in worst case : 0 ns
```

Comparisons:

Quick Sort :

1) Average Case :
# of Swaps : 6
# of Comparisons : 9
# of Operations : 15

2) Best Case :
# of Swaps : 12
# of Comparisons : 29
# of Operations : 41

3) Worst Case :
# of Swaps : 18
# of Comparisons : 49
# of Operations : 67

Merge Sort :

1) Average Case :
# of Comparisons : 13
# of Operations : 13

2) Best Case :
# of Comparisons : 9
# of Operations : 9

3) Worst Case :
# of Comparisons : 11
# of Operations : 11

Conclusion:-

In Average Case or small datasets Quick Sort performs better. Also, one can avoid using extra memory space, which is required in case of Merge Sort, for small and average datasets. However, in Best and Worst case, Merge Sort is always preferable as the time complexity of Quick Sort in both the cases is $O(N^2)$ because in every recursive step the pivot is either always the greatest element or the smallest element. Merge Sort has time complexity O(N*logN) in all the cases in contrast to Quick Sort which has time complexity O(N*logN) in Average Case and $O(N^2)$ in the remaining Best and Worst cases. One must always prefer Merge Sort for large datasets as it is time efficient though not space efficient as it requires a space complexity of O(N) while in case of small datasets one must try to avoid extra space and prefer Quick Sort.