**Experiment No. – 2.1**

**Aim**:

Write a program to implement merge sort and quick sort and compare their worst case complexities.

1. **Problem Description:**

We have given array of size n we have to implement Merge Sort and understand its time complexity in all three cases best case, average case & worst case and also space complexity.

1. **Algorithm:**

# **Algorithm for Merge Sort:**

**Mergesort(a, beg, end)**

Step 1: Repeat steps 2 to 3 when mergesort is recursively called.

Step 2: Check if beg < end then:

Calculate mid i.e. mid = (beg + end)/2;

mergesort(a, beg, mid);

mergesort(a,mid+1,end);

mergingsortedsubarrays(a,beg,mid,mid+1,end);

Step3: Return

**Mergingsortedsubarrays(a,beg,mid,mid+1,end)**

Step 1: Repeat steps 2 to 11 when mergingsortedsubarrays is called.

Step 2: Set lb = beg, lr = mid, rb = mid+1, rr = end, na = lb, nb = rb, nc = lb

Step 3: Repeat steps 4 & 5 while(na<=lr) & (nb<=rr)

Step 4: if a[na] < a[nb] then:

set c[nc] = a[na] and na = na+1

else set c[nc] = a[nb] and nb = nb+1

Step 5: nc = nc+1

Step 6: execute step 7 if(na > lr)

Step 7: Repeat while nb <= rr

Set c[nc] = a[nb] and nc = nc+1 and nb = nb+1

Step 8: Execute step 9 if (na < lr)

Step 9: Repeat while na <= lr

Set c[nc]=a[na] and nc=nc+1 and na=na+1

Step 10: Repeat for k=lb to rr

Set a[k] = c[k]

Step 11: Return

1. **Complexity Analysis**

**Time complexity:**

Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation. **T(n) = 2T(n/2) + O(n)** The solution of the above recurrence **is O(nLogn)**  as the mergesort always divides the array into two halves and takes linear time to merge two halves.

**Best Case: O(n\*logn)** when array is already sorted.

**Average Case: O(n\*logn)**

**Worst Case: O(n\*logn)** when array is not sorted.

**Space Complexity**: O(n) In merge sort, an extra variable is required for swapping.

1. **Pseudo Code**

Pseudo Code for Merge Sort

Procedure Mergesort(arr, l, r)

    Set mid = (l + r) / 2

    if (l < r)

       Mergesort(arr, l, mid)

        Mergesort(arr, mid + 1, r)

        Merge(arr, l, mid, r)

End procedure

Procedure Merge(L, R, A)

nL = length(L) // Size of Left Sub Array

nR = length(R) // Size of Right Sub Array

i = j = k = 0

while(i<nL && j<nR)

/\* When both i and j are valid i.e. when both the sub arrays have elements to insert in A \*/

if(L[i] <= R[j])

A[k] = L[i]

k = k+1

i = i+1

else

A[k] = R[j]

k = k+1

j = j+1

// Adding Remaining elements from left sub array to array A

while(i<nL)

A[k] = L[i]

i = i+1

k = k+1

// Adding Remaining elements from right sub array to array A

while(j<nR)

A[k] = R[j]

j = j+1

k = k+1

End procedure

1. **Source Code (C/C++):**

**Code for Merge Sort**

#include <bits/stdc++.h>

using namespace std;

void Merge(int arr[], int l, int mid, int r)

{

    int left = mid - l + 1;

    int right = r - mid;

    int a[left];

    int b[right];

    for (int i = 0; i < left; i++)

    {

        a[i] = arr[l + i];

    }

    for (int i = 0; i < right; i++)

    {

        b[i] = arr[mid + 1 + i];

    }

    int i = 0, j = 0, k = l;

    while (i < left && j < right)

    {

        if (a[i] < b[j])

        {

            arr[k] = a[i];

            k++;

            i++;

        }

        else

        {

            arr[k] = b[j];

            k++;

            j++;

        }

    }

    while (i < left)

    {

        arr[k] = a[i];

        k++;

        i++;

    }

    while (j < right)

    {

        arr[k] = b[j];

        k++;

        j++;

    }

}

void Mergesort(int arr[], int l, int r)

{

    int mid = (l + r) / 2;

    if (l < r)

    {

        Mergesort(arr, l, mid);

        Mergesort(arr, mid + 1, r);

        Merge(arr, l, mid, r);

    }

}

void print(int arr[], int n)

{

    for (int i = 0; i < n; i++)

    {

        cout << arr[i] << " ";

    }

}

int main()

{

    cout << "Name: Saurabh Kumar \nUID: 23MAI10004\n";

    cout << "Merge Sort\n";

    int n;

    cout << "Enter Size of array\n";

    cin >> n;

    int arr[n];

    cout << "Enter array elements \n";

    for (int i = 0; i < n; i++)

    {

        cin >> arr[i];

    }

    cout << "Array before sorting: ";

    print(arr, n);

    Mergesort(arr, 0, n - 1);

    cout << "\nArray after sorting : ";

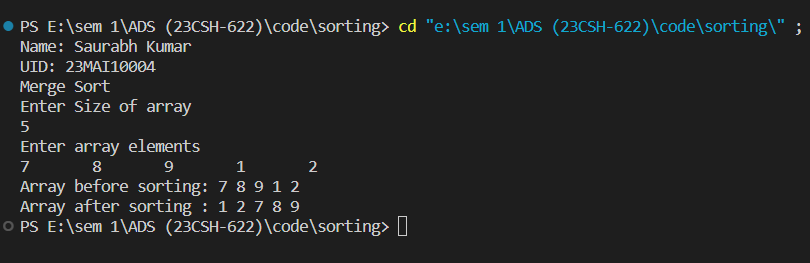
    print(arr, n);

    return 0;

}

1. **Screenshot of Outputs:**

**Output for Merge Sort:**

****

1. **Problem Description:**

We have given unsorted array of size n we have to implement quick sort and understand its time and space complexity and understand its worst time complexity.

1. **Algorithm:**

# **Algorithm for Quick Sort:**

QUICKSORT (array A, start, end)

**if** (start < end)

 p = partition(A, start, end)

 QUICKSORT (A, start, p - 1)

 QUICKSORT (A, p + 1, end

1. [Initialize] Set LEFT = BEG, RIGHT=END, and LOC = BEG.
2. [Scan from right to left]
   1. Repeat while A[LOC] <= A[RIGHT] and LOC != RIGHT:

RIGHT = RIGHT-1

[End of loop]

* 1. If LOC = RIGHT, then Return.
  2. If A[LOC] > A[RIGHT], then
     1. Interchange A[LOC] and A[RIGHT]
        1. TEMP = A[LOC] , A[LOC] = A[RIGHT].
        2. A[RIGHT] = TEMP.
     2. Set LOC = RIGHT.
     3. Go to Step 3.

[End of loop]

1. [Scan from left to right]
2. Repeat while A[LEFT] <= A[LOC] and LEFT != LOC:

LEFT = LEFT-1

[End of loop]

1. If LOC = LEFT, then Return.
2. If A[LEFT] > A[LOC], then
3. Interchange A[LEFT] and A[LOC]

TEMP = A[LOC] , A[LOC] = A[LEFT].

A[LEFT] = TEMP.

1. Set LOC = LEFT.
2. Go to Step 3.
3. **Complexity Analysis**

**Time complexity**

* **Best Case Complexity -** **O(n\*logn)**. In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element
* **Average Case Complexity - O(n\*logn)**. It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending.
* **Worst Case Complexity -** **O(n2)**. In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order.

**Space Complexity**: O(logn) The height of the recursion tree is always at least log n.

1. **Pseudo Code**

**Pseudo Code for Quick Sort**

function quickSort(arr, l, r)

if l < r

pivotIndex = partition(arr, l, r)

quickSort(arr, l, pivotIndex - 1)

quickSort(arr, pivotIndex + 1, r)

function partition(arr, l, r)

pivot = arr[r]

i = l - 1

for j = l to r - 1

if arr[j] < pivot

i = i + 1

swap arr[i] and arr[j]

swap arr[i + 1] and arr[r]

return i + 1

procedure quickSort(left, right)

if right-left <= 0

return

else

pivot = A[right]

partition = partitionFunc(left, right, pivot)

quickSort(left,partition-1)

quickSort(partition+1,right)

end if

end procedure

1. **Source Code (C/C++):**

**Code for Quick Sort:**

#include <bits/stdc++.h>

using namespace std;

void swapping(int &a, int &b){

    int temp;

    temp = a;

    a = b;

    b = temp;

}

int partition(int arr[], int start, int end){

    int pivot = arr[start];

    int count = 0;

    for (int i = start + 1; i <= end; i++){

        if (arr[i] <= pivot)

            count++;

    }

    int pivotIndex = start + count;

    swapping(arr[pivotIndex], arr[start]);

    int i = start, j = end;

    while (i < pivotIndex && j > pivotIndex){

        while (arr[i] <= pivot){

            i++;

        }

        while (arr[j] > pivot){

            j--;

        }

        if (i < pivotIndex && j > pivotIndex){

            swapping(arr[i++], arr[j--]);

        }

    }

    return pivotIndex;

}

void quickSort(int arr[], int start, int end){

    if (start >= end)

        return;

    int p = partition(arr, start, end);

    quickSort(arr, start, p - 1);

    quickSort(arr, p + 1, end);

}

void print(int arr[], int n){

    for (int i = 0; i < n; i++){

        cout << arr[i] << " ";

    }

}

int main(){

    cout << "Name: Saurabh Kumar \nUID: 23MAI10004\n";

    cout << "Quick Sort\n";

    int n;

    cout << "Enter Size of array\n";

    cin >> n;

    int arr[n];

    cout << "Enter array elements \n";

    for (int i = 0; i < n; i++)    {

        cin >> arr[i];

    }

    cout << "Array before sorting: ";

    print(arr, n);

    quickSort(arr, 0, n - 1);

    cout << "\nArray after sorting : ";

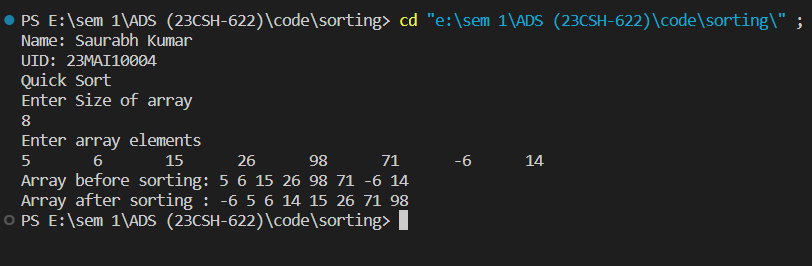
    print(arr, n);

    return 0;

}

1. **Screenshot of Outputs:**

**Output for Quick Sort:**

****

1. **Learning Outcomes**
2. Learnt about how to write code in c++ and take array as input.
3. Learnt about Merge sort and Quick sort.
4. Learnt about time and space complexity of Merge sort.
5. Learnt about time and space complexity of Quick sort.
6. Learnt what is the worst case time complexity of quick sort and why.