

Experiment-2.1

Aim of the Experiment :

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

1. Problem Description :

Merge Sort: Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array. It is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

2. Algorithm :

Mergesort(a,beg,end)

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

Step 2: Check if $beg < end$ then:

1. Calculate mid i.e. $mid = (beg + end) / 2$;
2. mergesort(a,beg,mid);
3. mergesort(a,mid+1,end);
4. merging sorted subarrays(a,beg,end);

Step 3: Return

Merging sorted subarrays(arr,s,e)

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

Step 2: Find mid and Set $len1 = mid - s + 1$, $len2 = e - mid$, $k = s$.

Course Name: ADSA Lab

Course Code: 23CSH-622

Step 3: Repeat for i=0 to len1

Set first[i]=arr[k] and k=k+1

Step 4: Repeat for i=0 to len2

Set second[i]=arr[k] and k=k+1

Step 5: Set index1=0, index2=0, k=s.

Step 6: Repeat steps 7 & 8 while(index1<len1)&(index2<len2)

Step 7: if first[index1]<second[index2] then:

set arr[k]=first[index1] and index1=index1+1

else

set arr[k]=second[index2] and index2=index2+1

Step 8: k=k+1

Step 9: Repeat while index1<=len1

Set arr[k]=first[index1] and k=k+1 and index1=index1+1

Step 10: Repeat while index2<=len2

Set arr[k]=second[index2] and k=k+1 and index2=index2+1

Step 11: Return

3. Complexity Analysis:

Time Complexity: $O(N\log(N))$

Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation:

$$T(n) = 2T(n/2) + \theta(n)$$

It can be solved using Recurrence Tree method or Master method. It falls in case II of the Master Method and the solution of the recurrence is $\theta(N\log(N))$. The time complexity of Merge Sort is $\theta(N\log(N))$ in all 3 cases (worst, average, best) as merge sort always divides array into 2 halves and takes linear time to merge two halves.

Space Complexity: $O(N)$, In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.

4. Pseudo Code :

MERGE_SORT(arr, beg, end)

if beg < end

 set mid = (beg + end)/2

 MERGE_SORT(arr, beg, mid)

 MERGE_SORT(arr, mid + 1, end)

 MERGE (arr, beg, mid, end)

end of if

END MERGE_SORT

5. Source Code for Experiment :

```
#include <iostream>
```

```
using namespace std;
```

```
void printArray(int arr[], int n){
```

```
    for(int i=0; i<n; i++){
```

```
        cout<<arr[i]<<" ";
```

```
    }
```

```
}
```

```
void merge(int arr[], int s, int e){
    //Create 2 new arrays to copy data
    int mid= s+(e-s)/2;
    int len1=mid-s+1;
    int len2=e-mid;
    int *first=new int[len1];
    int *second=new int[len2];

    // copy data into arrays
    int k=s;
    for(int i=0; i<len1; i++){
        first[i]=arr[k];
        k++;
    }
    for(int i=0; i<len2; i++){
        second[i]=arr[k];
        k++;
    }

    //merge sorted arrays
    int index1=0;
    int index2=0;
    k=s;
    while(index1<len1 && index2<len2){
        if(first[index1]<second[index2]){
            arr[k]=first[index1];
            k++;
            index1++;
        }
        else{
            arr[k]=second[index2];
            k++;
            index2++;
        }
    }
}
```

```
while(index1<len1){
    arr[k]=first[index1];
    k++;
    index1++;
}
while(index2<len2){
    arr[k]=second[index2];
    k++;
    index2++;
}
delete []first;
delete []second;
}

void mergeSort(int arr[], int s, int e){
    // Base Case
    if(s>=e){
        return;
    }
    int mid=s+(e-s)/2;
    // left part sort
    mergeSort(arr,s,mid);
    // right part sort
    mergeSort(arr,mid+1,e);
    // merge
    merge(arr,s,e);
}

int main() {
    cout<<"\nExperiment-2.1 (Ashish Kumar, 23MAI10008)"<<endl<<endl;
    cout<<"Performing Merge Sort ..."<<endl;
    int n;
    int arr[100];
    cout<<"Enter size of array: ";
    cin>>n;
```

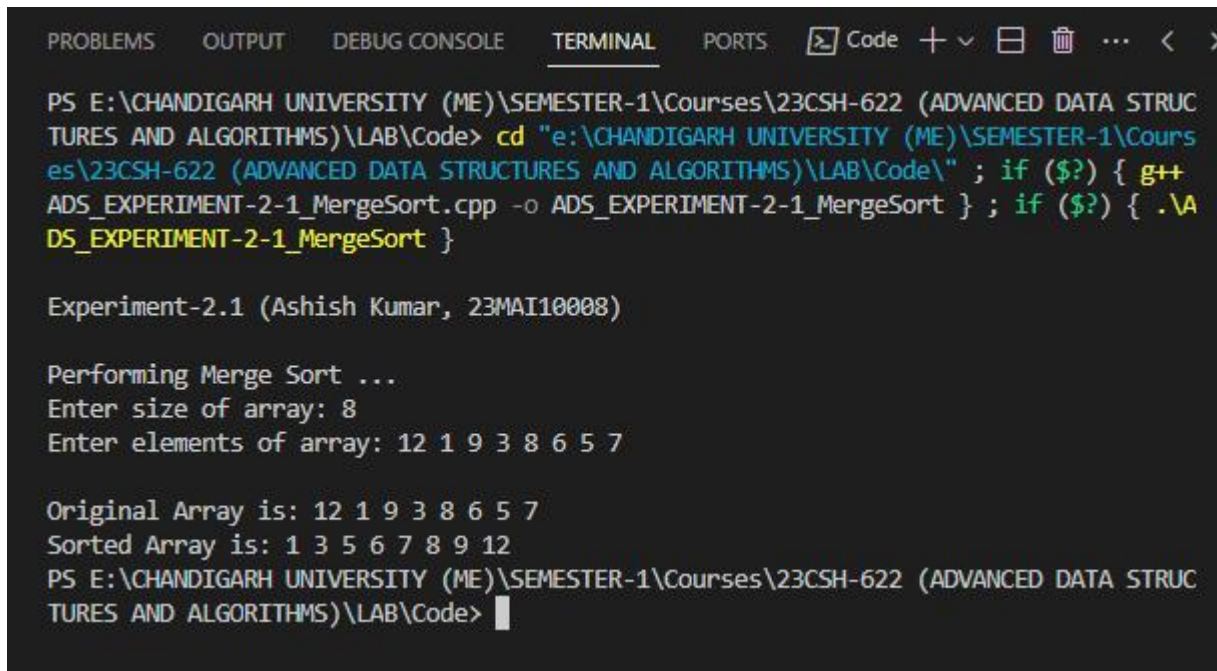
```
cout<<"Enter elements of array: ";
for(int i=0; i<n; i++){
    cin>>arr[i];
}

cout<<"\nOriginal Array is: ";
printArray(arr,n);
cout<<endl;

// Merge Sort Function
mergeSort(arr,0,n-1);

cout<<"Sorted Array is: ";
printArray(arr,n);
return 0;
}
```

6. Result/Output :



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Code + - [ ] [X] ... < >

PS E:\CHANDIGARH UNIVERSITY (ME)\SEMESTER-1\Courses\23CSH-622 (ADVANCED DATA STRUC
TURES AND ALGORITHMS)\LAB\Code> cd "e:\CHANDIGARH UNIVERSITY (ME)\SEMESTER-1\Cours
es\23CSH-622 (ADVANCED DATA STRUCTURES AND ALGORITHMS)\LAB\Code\" ; if ($?) { g++
ADS_EXPERIMENT-2-1_MergeSort.cpp -o ADS_EXPERIMENT-2-1_MergeSort } ; if ($?) { .\A
DS_EXPERIMENT-2-1_MergeSort }

Experiment-2.1 (Ashish Kumar, 23MAI10008)

Performing Merge Sort ...
Enter size of array: 8
Enter elements of array: 12 1 9 3 8 6 5 7

Original Array is: 12 1 9 3 8 6 5 7
Sorted Array is: 1 3 5 6 7 8 9 12
PS E:\CHANDIGARH UNIVERSITY (ME)\SEMESTER-1\Courses\23CSH-622 (ADVANCED DATA STRUC
TURES AND ALGORITHMS)\LAB\Code> █
```

1. Problem Description :

Quick Sort: Quick Sort is a recursive algorithm which divides the given array into two sections using a partitioning element called as pivot. The division performed is such that all the elements to the left side of pivot are smaller than pivot, all the elements to the right side of pivot are greater than pivot. After dividing the array into two sections, the pivot is set at its correct position. Then, sub arrays are sorted separately by applying quick sort algorithm recursively.

2. Algorithm :

QuickSort(A,N,BEG,END,LOC)

1.[Initialize.] Set LEFT: = BEG, RIGHT: = END and LOC:=BEG.

2.[Scan from right to left.]

(a)Repeat while $A[LOC] \leq A[RIGHT]$ and $LOC \neq RIGHT$;

RIGHT: = RIGHT -1

(b)It $LOC = RIGHT$, then: Return.

(c)It $A[LOC] > A[RIGHT]$, then:

(i)[Interchange $A[RIGHT]$ and $A[LOC]$.]

TEMP: = $A[LOC]$, $A[LOC]$: = $A[RIGHT]$.

$A[RIGHT]$: TEMP.

(ii)Set LOC : = $RIGHT$.

(iii)Go to Step 3.

[End of if structure.]

3.[Scan from left to right.]

(a)Repeat while $A[LEFT] \leq A[LOC]$ and $LEFT \neq LOC$:

$LEFT := LEFT + 1$.

[End of loop.]

(b)If $LOC = LEFT$, then: Return.

(c)If $A[LEFT] > A[LOC]$, THEN

(i)[Interchange $A[LEFT]$ and $A[LOC]$.]

$TEMP := A[LOC]$, $A[LOC] := A[LEFT]$,

$A[LEFT] := TEMP$.

(ii)Set $LOC := LEFT$.

(iii)Go to step 2.

[End of structure.]

3. Complexity Analysis:

Time Complexity:

1) Best Case: $O(N \log(N))$. When the pivot divides the array into roughly equal halves.

2) Worst Case: $O(N^2)$. When array is already sorted and pivot is always chosen as smallest or largest element.

3) Average Case: $O(N \log(N))$.

Space Complexity: $O(1)$. As no extra space is used to sort the elements.

4. Pseudo Code :

QUICKSORT (array A, start, end)

if (start < end)

 p = partition(A, start, end)

 QUICKSORT (A, start, p - 1)

 QUICKSORT (A, p + 1, end)

end of if

END QUICKSORT

5. Source Code for Experiment :

```
#include <iostream>
```

```
using namespace std;
```

```
void printArray(int arr[], int n) {
```

```
    for(int i=0; i<n; i++) {
```

```
        cout<<arr[i]<<" ";
```

```
    }
```

```
}
```

```
// Partition the array using the last element as the pivot
```

```
int partition(int arr[], int s, int e) {
```

```
    // Choose the pivot element
```

```
    int pivot = arr[e];
```

```
    int i = (s - 1);
```

```
for (int j = s; j <= e - 1; j++) {  
    // If current element is smaller than the pivot  
    if (arr[j] < pivot) {  
        // Increment index of smaller element  
        i++;  
        int t = arr[i];  
        arr[i] = arr[j];  
        arr[j] = t;  
    }  
}  
int t = arr[i+1];  
arr[i+1] = arr[e];  
arr[e] = t;  
  
return (i + 1);  
  
}
```

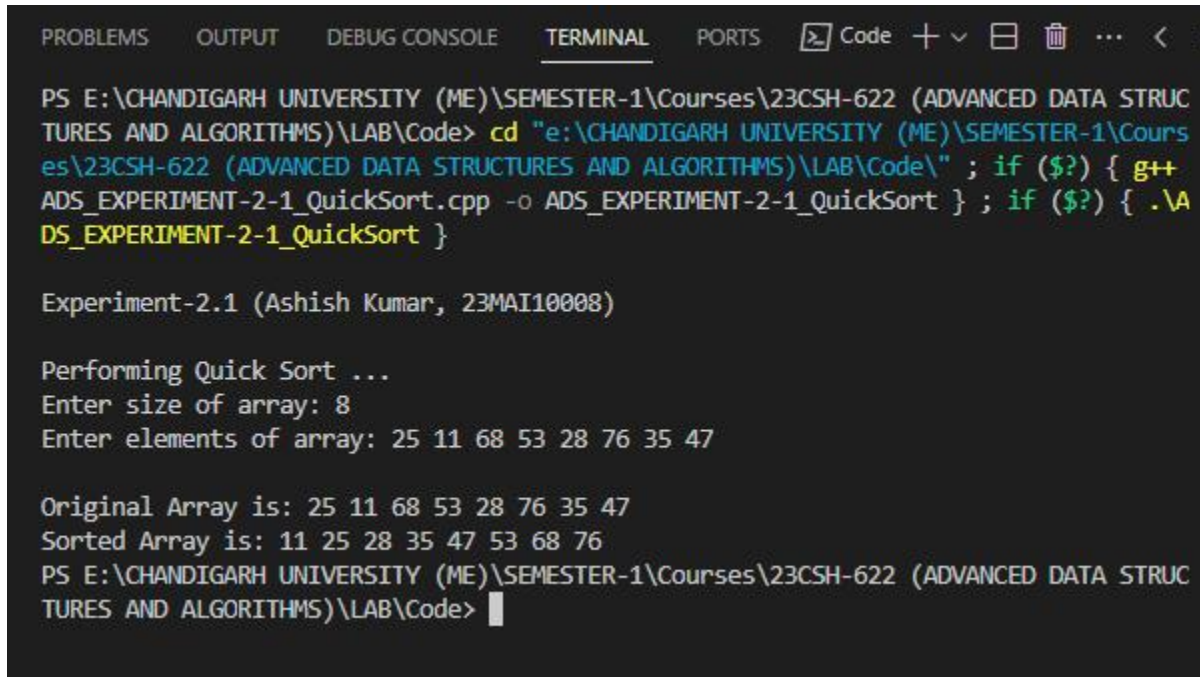
```
void quickSort(int arr[], int s, int e) {
```

```
    // Base Case  
    if(s>=e){  
        return;  
    }  
  
    // p is the partitioning index  
    int p = partition(arr,s,e);  
    // left part sort  
    quickSort(arr,s,p-1);  
    // right part sort  
    quickSort(arr,p+1,e);  
  
}
```



```
int main() {  
  
    cout<<"\nExperiment-2.1 (Ashish Kumar, 23MAI10008)"<<endl<<endl;  
    cout<<"Performing Quick Sort ..."<<endl;  
  
    int n;  
    int arr[100];  
    cout<<"Enter size of array: ";  
    cin>>n;  
  
    cout<<"Enter elements of array: ";  
    for(int i=0; i<n; i++){  
        cin>>arr[i];  
    }  
  
    cout<<"\nOriginal Array is: ";  
    printArray(arr,n);  
    cout<<endl;  
  
    // Quick Sort Function  
    quickSort(arr,0,n-1);  
  
    cout<<"Sorted Array is: ";  
    printArray(arr,n);  
  
    return 0;  
}
```

6. Result/Output :



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  Code + - [ ] [X] ... < >

PS E:\CHANDIGARH UNIVERSITY (ME)\SEMESTER-1\Courses\23CSH-622 (ADVANCED DATA STRUCTURES AND ALGORITHMS)\LAB\Code> cd "e:\CHANDIGARH UNIVERSITY (ME)\SEMESTER-1\Courses\23CSH-622 (ADVANCED DATA STRUCTURES AND ALGORITHMS)\LAB\Code\" ; if ($?) { g++ ADS_EXPERIMENT-2-1_QuickSort.cpp -o ADS_EXPERIMENT-2-1_QuickSort } ; if ($?) { .\ADS_EXPERIMENT-2-1_QuickSort }

Experiment-2.1 (Ashish Kumar, 23MAI10008)

Performing Quick Sort ...
Enter size of array: 8
Enter elements of array: 25 11 68 53 28 76 35 47

Original Array is: 25 11 68 53 28 76 35 47
Sorted Array is: 11 25 28 35 47 53 68 76
PS E:\CHANDIGARH UNIVERSITY (ME)\SEMESTER-1\Courses\23CSH-622 (ADVANCED DATA STRUCTURES AND ALGORITHMS)\LAB\Code> 
```

Learning outcomes (What I have learnt):

1. I learnt about how to input elements in an array.
2. I learnt about how to perform merge sort in an array.
3. I learnt about how to perform quick sort in an array.
4. I learnt about comparison between merge sort and quick sort.
5. I learnt about time and space complexity of merge sort and quick sort.

Comparison between Merge Sort and Quick Sort:

Merge Sort	Quick Sort
In the Merge sort, the array is parted into just 2 halves (i.e. $n/2$).	In the Quick sort, splitting of array is in any ratio, not necessarily divided into half.
For merging of sorted sub-arrays, it needs a temporary array with the size equal to the number of input elements.	Quicksort does not require additional array space.
The major work is to combine the two sub-arrays after sorting them recursively.	The major work is to partition the array into two sub-arrays before sorting them recursively.
Merge sort operates fine on any size of array	Quick sort works well on smaller array
Merge sort is more efficient.	Quick sort is inefficient for larger arrays.
Merge sort is a stable algorithm.	Quick sort is not a stable algorithm.
Merge sort is not in – place sorting method.	Quick sort is in- place sorting method.
Merge sort has a consistent speed on any size of data	Quick sort works faster than other sorting algorithms for small data set like Selection sort etc