K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

Batch: C2	**Roll No.:16010122257**
Experiment No.	3
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

Title: Implementation of Quick sort/Merge sort algorithm

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

CO to be achieved:

CO 2 Describe various algorithm design strategies to solve different problems and analyze Complexity.

Books/ Journals/ Websites referred:

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran," Fundamentals of computer algorithm", University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein," Introduction to algortihtms",2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://en.wikipedia.org/wiki/Quicksort**
4. **https://www.cs.auckland.ac.nz/~jmor159/PLDS210/qsort.html**
5. **http://www.cs.rochester.edu/~gildea/csc282/slides/C07-quicksort.pdf**
6. **http://www.sorting-algorithms.com/quick-sort**
7. **http://www.cse.ust.hk/~dekai/271/notes/L01a/quickSort.pdf**
8. **http://en.wikipedia.org/wiki/Merge_sort**
9. **http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/mergeSort.htm**
10. **http://www.sorting-algorithms.com/merge-sort**
11. **http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Merge_sort.html**

Pre Lab/ Prior Concepts:

Data structures, various sorting techniques

Historical Profile:

Quicksort and merge sort are divide-and-conquer sorting algorithm 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.

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);

        }
}
```

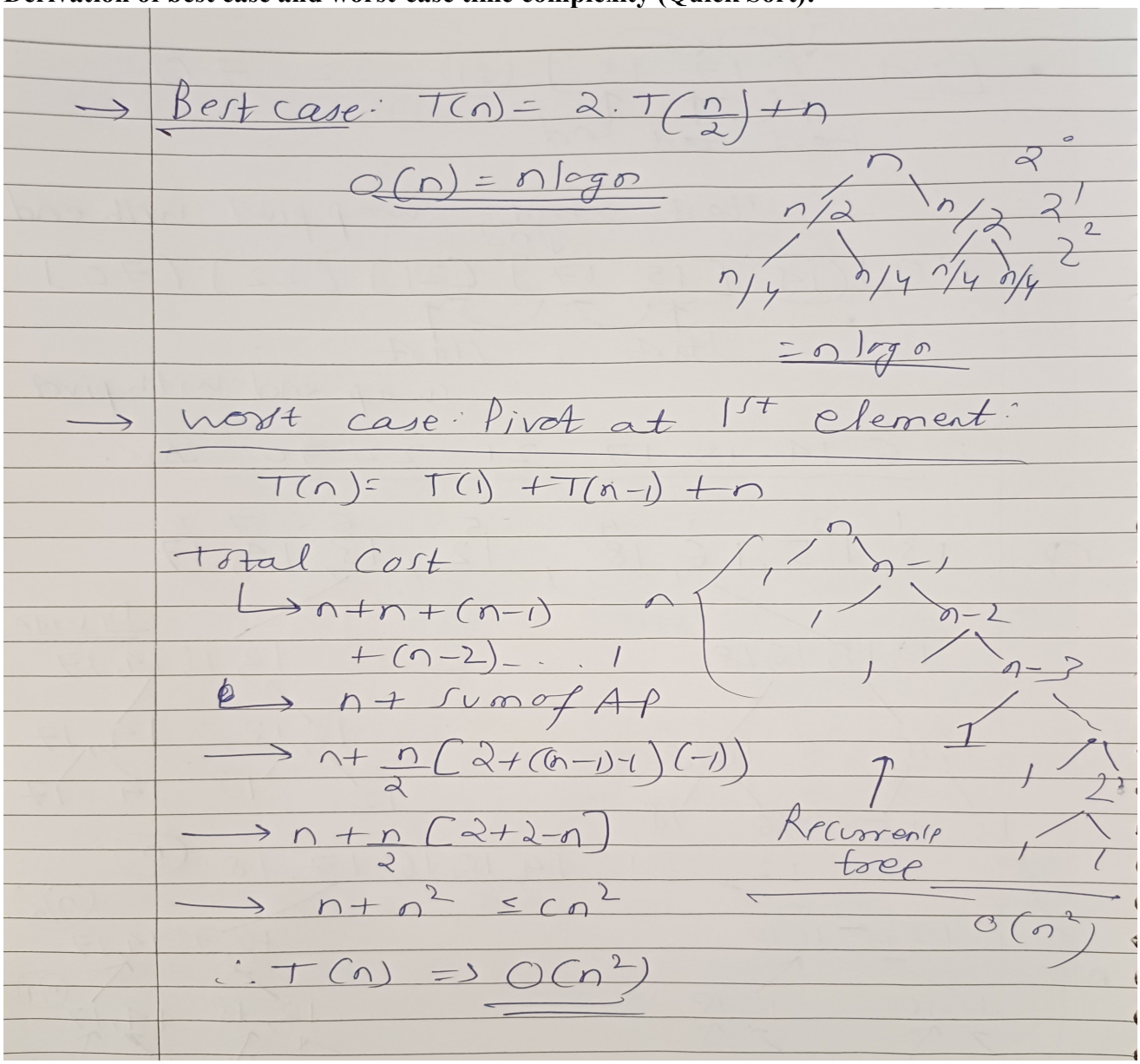
Integer *partition(integer A*T[], Integer *left*, Integer *right*)
//This function rarranges *A*[*left*..*right*] and finds and returns an integer *q*, such that *A*[*left*], ..., //*A*[*q*–1] <~ *pivot*, *A*[*q*] = *pivot*, *A*[*q*+1], ..., *A*[*right*] > *pivot*, where *pivot* is the first element 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:

- The space complexity of quick sort is O(logn).

Derivation of best case and worst-case time complexity (Quick Sort):



CODE:

```
#include <bits/stdc++.h>
using namespace std;
void quicksort(int Array[], int l, int r);
int partition(int Array[], int l, int r);
void swap(int &A, int &B);
int main()
{
int Array[10], num;
cout<<"Enter the number of array elements: ";
cin>>num;
cout<<"Input the array elements: ";
for(int i=0; i<num; i++)
{
cin>>Array[i];
}
quicksort(Array, 0, num-1);
cout<<"The sorted array is: ";
for(int i=0; i<num; i++)
{
cout<<Array[i]<<" ";
}
return 0;
}
void quicksort(int Array[], int l, int r)
{
if(l<r)
{
int Q = partition(Array, l, r);
quicksort(Array, l, Q-1);
quicksort(Array, Q+1, r);
}
}
```

```
int partition(int Array[], int l, int r)
{
int p = Array[l];
int lo = l+1;
int hi = r;
while(lo<=hi)
{
while(Array[hi]>p)
{
hi -= 1;
}
while(lo<=hi && Array[lo]<=p)
{
lo += 1;
}
if(lo<=hi)
{
swap(Array[lo], Array[hi]);
}
}
swap(Array[l], Array[hi]);
return hi;
}
void swap(int &A, int &B)
{
int t = A;
A = B;
B = t;
}
```

OUTPUT:

```
input
Enter the number of elements in the array: 10
Enter the elements of the array: 1 7 14 16 19 6 2 12 23 21
The sorted array is: 1 2 6 7 12 14 16 19 21 23

...Program finished with exit code 0
Press ENTER to exit console.
```

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 $p \leq q \leq r$ 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 sub array 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.

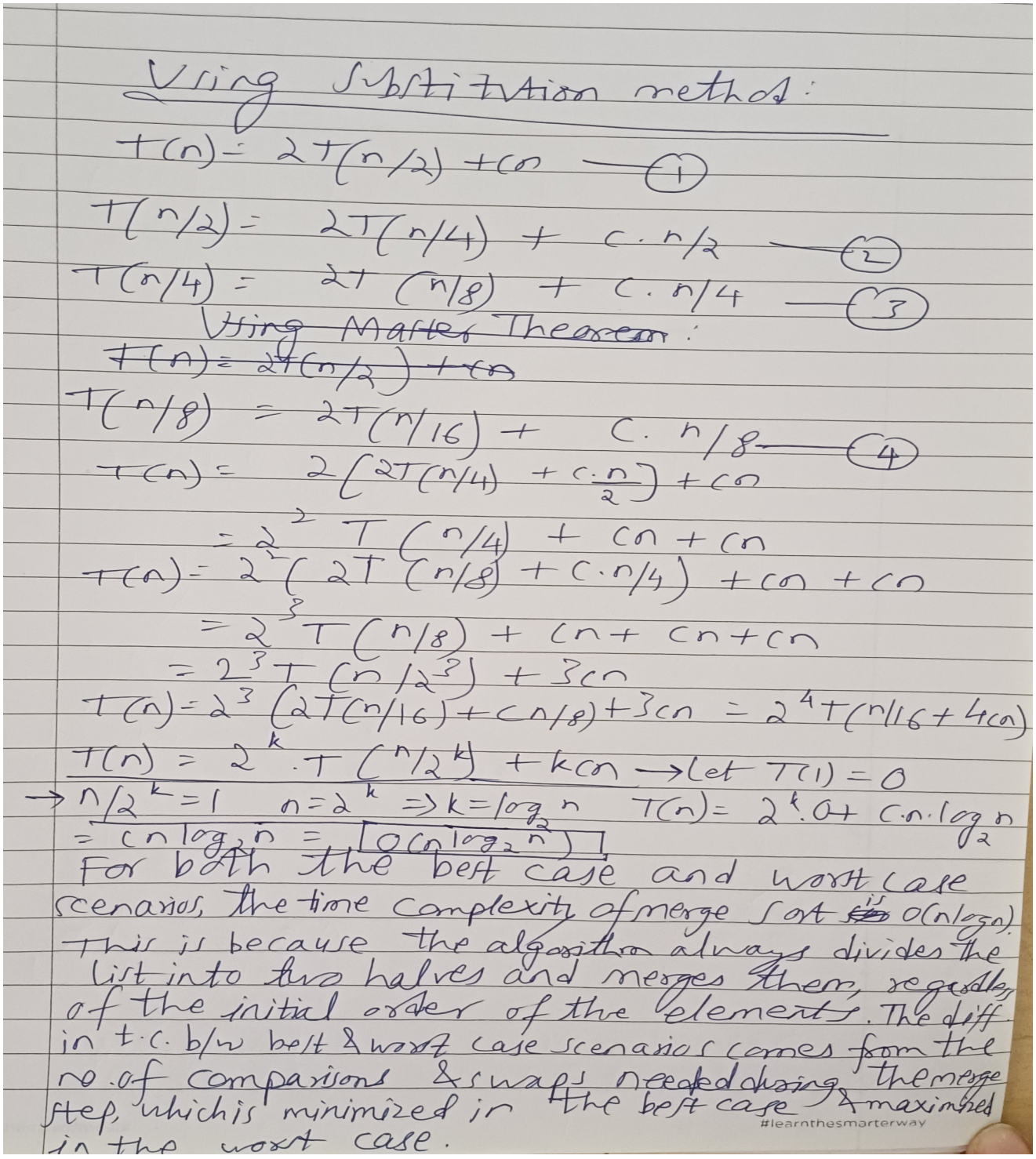
MERGE (A, $q + 1$, r) // Conquer step.
MERGE (A, p, q, r) // Conquer step.

```
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
}
```

The space complexity of Merge sort:

- The space complexity of merge sort is O(n).

Derivation of best case and worst-case time complexity (Merge Sort):



CODE:

```
#include <bits/stdc++.h>
using namespace std;
void mergeSort(int Array[], int p, int r);
void merge(int Array[], int p, int q, int r);
int main()
{
int Array[10], num;
cout<<"Enter the number of array elements: ";
cin>>num;
cout<<"Input the array elements: ";
for(int i=0; i<num; i++)
{
cin>>Array[i];
}
mergeSort(Array, 0, num-1);
cout<<"The sorted array is: ";
for(int i=0; i<num; i++)
{
cout<<Array[i]<<" ";
}
return 0;
}
void mergeSort(int Array[], int p, int r)
{
if(p<r)
{
int q = (p+r)/2;
mergeSort(Array, p, q);
mergeSort(Array, q+1, r);
merge(Array, p, q, r);
}
}
```

```
void merge(int Array[], int p, int q, int r)
{
int A = q-p+1;

int B = r-q;
int Left[A+1], Right[B+1];
for(int i=0; i<A; i++)
{
Left[i] = Array[p+i];
}
for(int j=0; j<B; j++)
{
Right[j] = Array[q+j+1];
}
Left[A] = 9999;
Right[B] = 9999;
int i=0, j=0;
for(int k=p; k<=r; k++)
{
if(Left[i]<=Right[j])
{
Array[k] = Left[i];
i++;
}
else
{
Array[k] = Right[j];
j++;
}
}
}
```

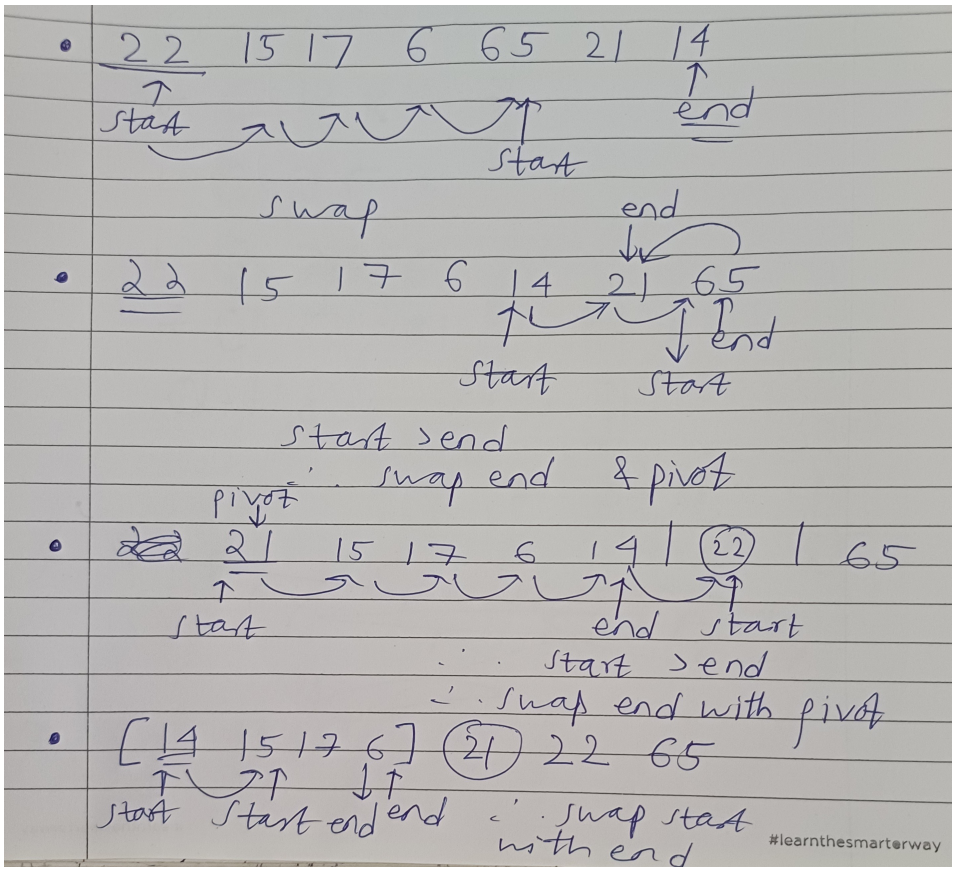
OUTPUT:

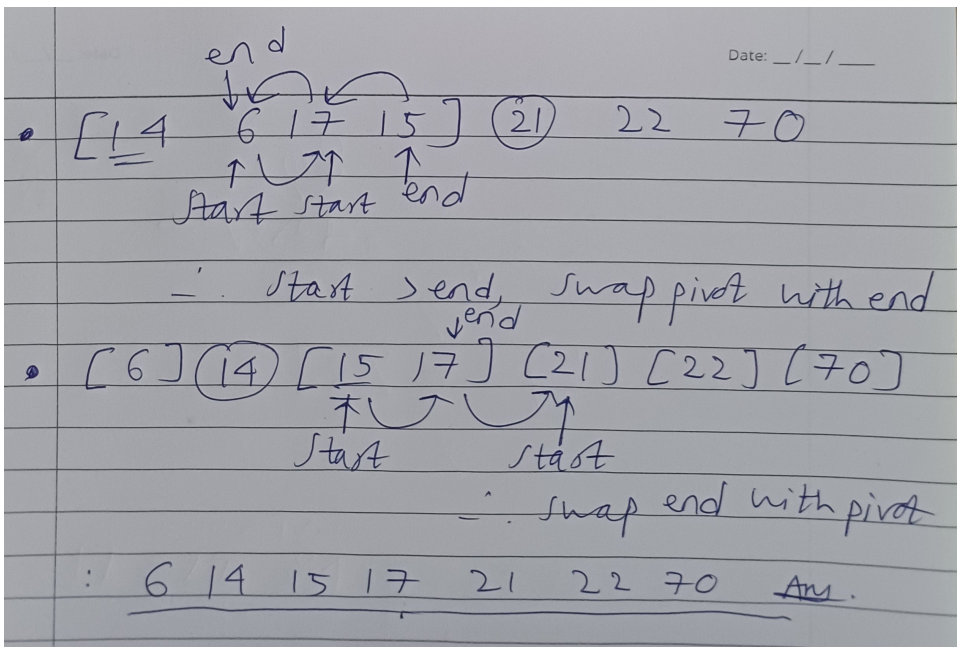
```
Enter the number of elements in the array: 6
Enter the elements of the array: 4 5 2 22 99 66
The sorted array is: 2 4 5 22 66 99

...Program finished with exit code 0
Press ENTER to exit console.
```

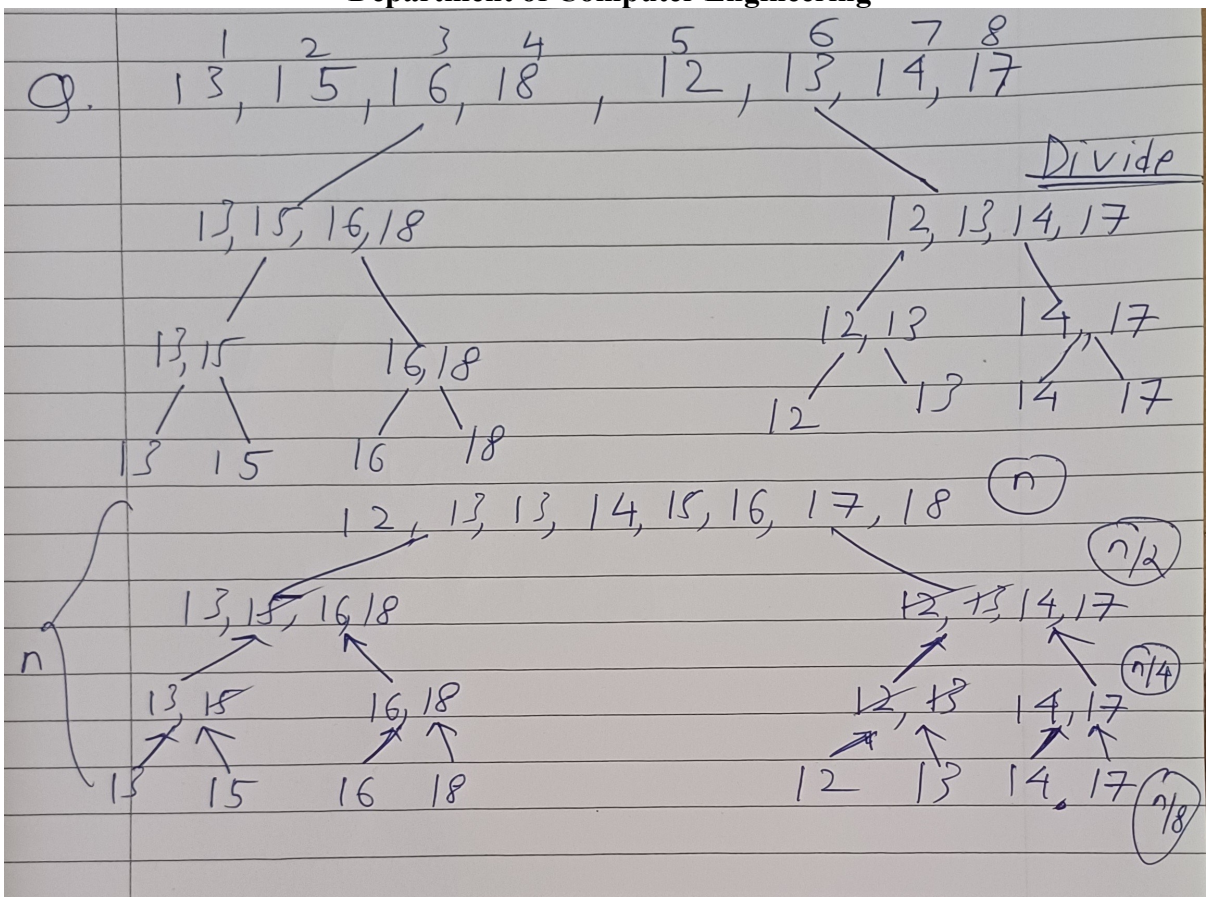
Example for quicksort/Merge tree for merge sort:

Quicksort:





Merge sort:



CONCLUSION: In this experiment,we delved into the fundamentals of two sorting algorithms namely, quick sort and merge sort,which are useful for sorting efficiently.From this experiment,we conclude that ,there are few differences between quick sort and merge sort techniques,which are as follows:

Basis for comparison	Quick Sort	Merge Sort
The partition of elements in the array	The splitting of a array of elements is in any ratio, not necessarily divided into half.	In the merge sort, the array is parted into just 2 halves (i.e. n/2).
Worst case complexity	O(n^2)	O(nlogn)
Works well on	It works well on smaller array	It operates fine on any size of array
Speed of execution	It work faster than other sorting algorithms for small data set like Selection sort etc	It has a consistent speed on any size of data

Basis for comparison	Quick Sort	Merge Sort
Storage space reqrmnt.	Less(In-place)	More(not In-place)
Efficiency	Inefficient for larger arrays	More efficient
Sorting method	Internal	External
Stability	Not Stable	Stable
Preferred for	for Arrays	for Linked Lists
Locality of reference	good	poor
Major work	The major work is to partition the array into two sub-arrays before sorting them recursively.	Major work is to combine the two sub-arrays after sorting them recursively.
Division of array	Division of an array into sub-arrays may or may not be balanced as the array is partitioned around the pivot.	Division of an array into sub array is always balanced as it divides the array exactly at the middle.
Method	Quick sort is in- place sorting method.	Merge sort is not in – place sorting method.
Merging	Quicksort does not need explicit merging of the sorted sub-arrays; rather the sub-arrays rearranged properly during partitioning.	Merge sort performs explicit merging of sorted sub-arrays.
Space	Quicksort does not require additional array space.	For merging of sorted sub-arrays, it needs a temporary array with the size equal to the number of input elements.