Gaurav Yadav
Roll No: 11911038
Branch: CSE

Data Structures Assignment.

Q1. Analysis of time complexity of any list in insertion sort.

→ For best case, list should be in ascending order, which will have time complexity lesser than $O(n^2)$.

Algorithm:-

```
for (int x=1; x<n; x++)
{
    temp = arr[x];
    for (int y=x-1; y>=0; y--)
        if (temp < arr[y])
        {
            arr[y+1] = arr[y];
            arr[y] = temp; }
```

```
            else
                break;
        }
    }
```

Consider list is {7, 9, 11, 13, 16};

Loop 1 →

temp = 9 ;

loop eq. (9 < 7)

false ⟶ break condition

Means at x = 1 loop → 2 has been called 1 time;

Similarly

(11) for x = 2 temp = 11 ;

loop 2 y = 1 ✓ y > 0 true

if (11 < arr[1])

false ⟶

means at $x=2$. loop 2 has been called once again.

So, we can say - [In ascending order]

Loop 1	Loop 2	No of calls
~~$x=1$~~	$y=0$	1
$x=2$	$y=1$	1
$x=3$	$y=2$	1
$x=4$	$y=3$	1
$x=5$	$y=4$	1
⋮	⋮	⋮
$x=n-1$	$y=n-2$	1

Total f^n calls = $n-1$

Time complexity = $O(n-1) \simeq O(n)$

⊛ Bubble Sort :-

Algorithm

```
for ( int a=0 ; a<n-1 ; a++)
{  for(int b=0 ; b<n-1 ; a++ )
   {  if( a[b] > a[b+1] )
      {
         temp = a[b];
         a[b] = a[b+1];
         a[b+1] = temp;
      }
   }
}
```

2.

Time Complexity :-

$T(n) = O(n-1) \times O(n-1)$

$= [O(n-1)]^2 \simeq O(n^2)$

loop 1 :-

loop 1 operates $(n-1)$ times

loop 2 :-

loop 2 operates $(n-1)^2$ times

So, $T(n) = O(n-1) \times O(n-1) \simeq O(n^2)$

Space complexity :- $O(1) \simeq$ constant

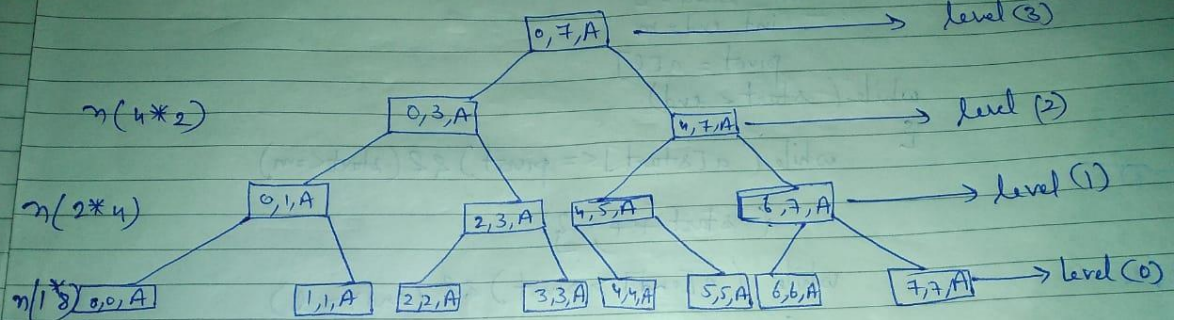
Merge Sort Algorithm :-

```
void merge_sort (int l, int r, int *a)
{
    if(l<r)
    { int m= (l+r)/2;

        merge_sort (l, m, a);
        merge_sort (m+1, r, a);
        merge (l, m, r, a);
    }
}
void merge( int l, int m, int r, int *p)
{   int n1= m-l+1;
    int n2= r-m;

}
```

let us consider a element:-



Numbes of level $o(n) = \log_2(n) = \log_2(8) = 3$

Time complexity $[T(n)] \rightarrow n \log n$

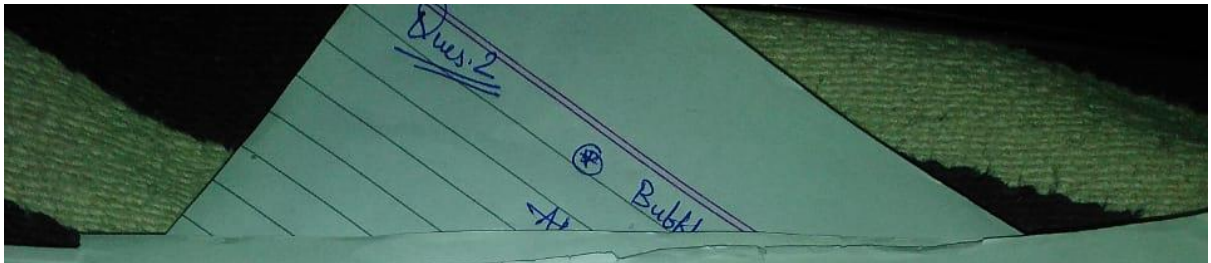
Merge sort space complexity = $o(n)$.

* Quick sort Algorithm:-

```
void sorting( l, m, array)
{
```

// l=0 ; m=n-1

initial le




```
int start = l
    int end = m
    pivot = a[l]
while( start < end)
{   while ( a[start] <= pivot) && (start <=m)
        { start++; }
    while ( a[end] > pivot) && (end >= l)
      { end--; }
      if (start < end)
      { swap (array [start, array [end]);
      }
  return end;
}
```