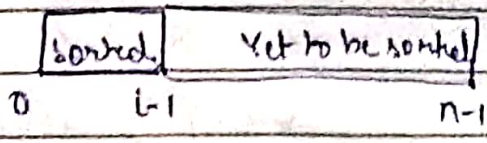
Explaination-

Q1→ Insertion sort comprise of two processes -
(i) comparing of pair of element.
(ii) shifting of element if not in order.

So, these processes effects the time complexity of the sorting algorithm.

```
void isort ( int arr[], int n )
{
    for ( int i=1 ; i<n; i++ )
    {
        int key = arr[i];
        j = i-1;
        while (j>=0 && arr[j]> key ) - (1)
        {
            arr [j+1] = arr [j];
            j--;
        }
        arr [j+1] = key;
    }
}
```



If sorted part becomes the whole array, then array is sorted.

* Now for the best case, the array would be sorted, so in this case there would be no shifting but there would be "n-1" comparision. So the complexity will turn out to be $O(n-1) \approx O(n)$.

It is because in sorted array -

$a[i] > a[i-1], a[i-2], \ldots a[0]$

so,

the while condition - (1)
would never be satisfied so no shifting would take place.

* Now incase of worst case, that is array is sorted in descending order the time complexity comes out to be $O(n^2)$.
To reduce it we use binary search instead of regular search to reduce the searching time from $O(n^2)$ to $O(n \log n)$ but the shifting time remains the same, so the complexity is still $O(n^2)$.

Now, there is difference between data structures also -
in array → shifting takes $O(n^2)$ + searching takes $O(n \log n)$
while
in linked list → shifting takes $O(1)$ + searching takes $O(n^2)$

So, no effect on the time complexity of algorithm, it still remains the same that is $O(n^2)$.

* Now to reduce the time complexity we take a space of n, & store the sorted array in reverse order.

in that array. By this method the shifting complexity is reduced to $O(n \log n)$ & thus the total complexity is reduced to $O(n \log n)$ in worst case.

Q2 →

* Bubble sort :-

Algorithm.

```
void bsort(int arr[], int n) {
for(int i=0; i<=n-2; i++)
{
    for(int j=0; j<=n-1-i; j++)
    {
        if(arr[j] > arr[j+1])
        {
            int temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
        }
    }
}
```

Now in worst case

$T(n) = O(n-1)$ (For external loop) $\times O(n-1)$ (For internal loop)

$\Rightarrow O(n-1)^2 \simeq O(n^2)$ //

& space complexity is O(constant) or O(1).

* So, it is a inplace algo.

Date __/__/__
Page _____

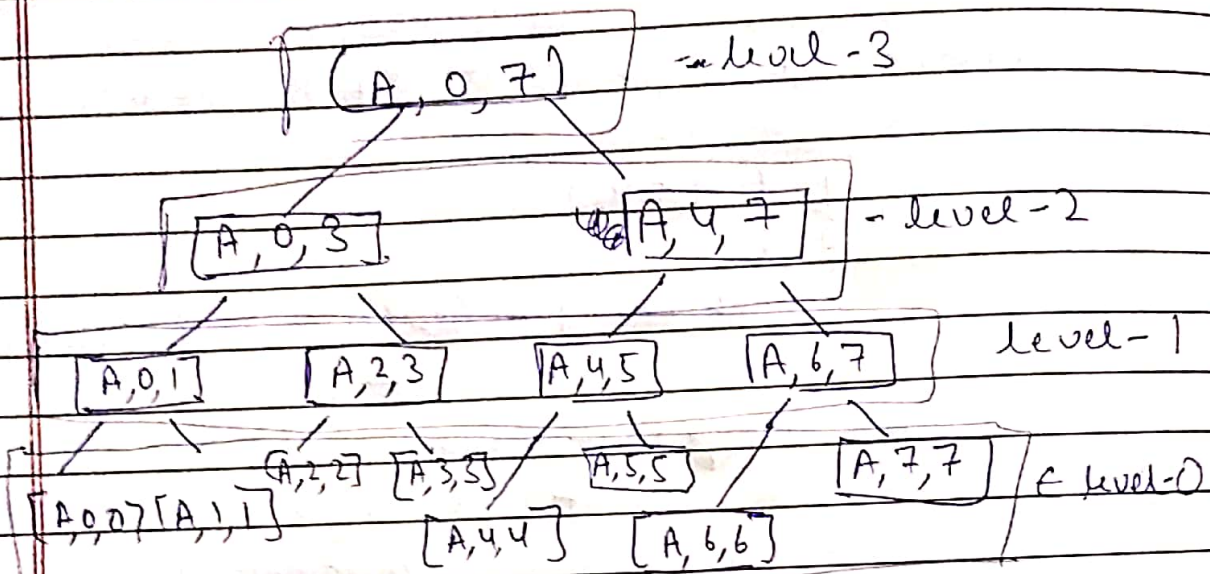
Merge sort Algorithm -

```
void msort ( int arr[], int l, int r )
{
    if ( l < r )
    {
        int m = l + (r - l) / 2;

        msort (arr, l, m);
        msort (arr, m+1, r);
        merge ( arr, l, m, r);
    }
}
void merge (int arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int arr1[n1], arr2[n2];
    int arr3[n1+n2];
/* Now these two arrays are sorted by
merging the array and storing it in
arr3[n1+n2]; This how the merge
sort algo works */
}
```

↑ time complexity of sorting is O(n).

Now consider 8 elements-



Now number of level = 3

that $\log_2 n = \log_2 8 = 3$

So Total time complexity is $n \log n$

& space complexity is $O(n)$

as a space of n element is created in merge function.

So, it is a outplace algo.

☆ In insertion sort algorithm -

→ Already discussed in Q-1-

time complexity worst case - $O(n^2)$

best case - $O(n)$

space complexity = $O(1)$ so, it is inplace algo.

* Quick sort algo -

A pivot element is selected from array, it bypart the array into two array, in which first sub array contains all value less than pivot & other sub array will contain all value greater than pivot.

```
partition (l, m, arr)   // l=0; m=n-1; initially
{
  int start = l;
  int end = m;
  pivot = a[l];
  while (start < end)
  {
    while ((arr[start] <= pivot) && (start <= m))
    {
      start++;
    }
    while (arr[end] > pivot) && (end >= l)
    {
      end--;
    }
    if (start < end)
    {
      swaping (arr[start], arr[end]);
    }
  }
  swapping (arr[l], arr[end])
  return end;
}
```