1) What is the time complexity of the following code?

```
int a = 0;
for (int i = 0; i < n; i++)
{
   for (int j = n; j > i; j--)
   {
      a = a + i + j;
   }
}
```

Output: O(N*N)

Explanation:

The above code runs total no of times,

= N + (N – 1) + (N – 2) + … 1 + 0
= N * (N + 1) / 2
= 1/2 * N^2 + 1/2 * N = (N^2).

Ans: The time complexity is O(N^2).

2) What is the time complexity of the following code?

```
int count = 0;
for (int i = 1; i <= n; i = i * 2)
{
for (int j = 1; j <= i; j ++)
{
count = count + 1;
}
}
```

Output: O(n)

Explanation:

The above code runs total no of times,

$=1(\frac{1-2^{(logn+1)}}{1-2})$

After doing all manipulation, we get

$=2^{logn} = n$

Ans: The time complexity is O(n).

3) Find the best case, average case and the worst case of Linear Search Algorithm.

Ans :

Worst Case Algorithm (Usually Done): -

In real life, most of the time we do the worst-case analysis of an algorithm. Worst case running time is the longest running time for any input of size nn.

In the linear search, the worst case happens when the item we are searching is in the last position of the array or the item is not in the array. In both the cases, we need to go through all nn items in the array. The worst-case runtime is, therefore, 0(n)O(n). Worst case performance is more important than the best-case performance in case of linear search because of the following reasons.

1. The item we are searching is rarely in the first position. If the array has 1000 items from 1 to 1000. If we randomly search the item from 1 to 1000, there is 0.001 percent chance that the item will be in the first position.
2. Most of the time the item is not in the array (or database in general). In which case it takes the worst-case running time to run.

Similarly, in insertion sort, the worst-case scenario occurs when the items are reverse sorted. The number of comparisons in the worst case will be in the order of n2n2 and hence the running time is 0(n2)O(n2).
Knowing the worst-case performance of an algorithm provides a guarantee that the algorithm will never take any time longer.

Average Case Algorithm (Sometimes done): -

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So, we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

Average Case Time =

$$\frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$

$$= \frac{\theta((n+1)*(n+2)/2)}{(n+1)}$$

= 0(n)

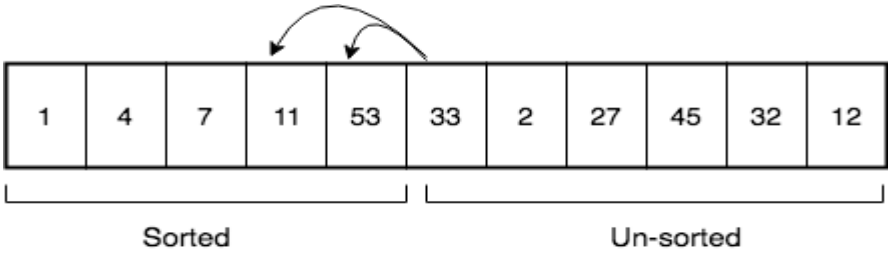
Best Case Algorithm (Bogus): -

Consider the example of Linear Search where we search for an item in an array. If the item is in the array, we return the corresponding index, otherwise, we return -1. The code for linear search is given below.

```
int search(int a, int n, int item) {
    int i;
    for (i = 0; i < n; i++) {
        if (a[i] == item) {
            return a[i]
        }
    }
    return -1
}
```

Variable `a` is an array, `n` is the size of the array and `item` is the item we are looking for in the array. When the item we are looking for is in the very first position of the array, it will return the index immediately. The `for` loop runs only once. So the complexity, in this case, will be O(1)O(1). This is the called the best case.

Consider another example of insertion sort. Insertion sort sorts the items in the input array in an ascending (or descending) order. It maintains the sorted and un-sorted parts in an array. It takes the items from the un-sorted part and inserts into the sorted part in its appropriate position. The figure below shows one snapshot of the insertion operation.



In the figure, items [1, 4, 7, 11, 53] are already sorted and now we want to place 33 in its appropriate place. The item to be inserted are compared with the items from right to left one-by-one until we found an item that is smaller than the item we are trying to insert. We compare 33 with 53 since 53 is bigger we move one position to the left and compare 33 with 11. Since 11 is smaller than 33, we place 33 just after 11 and move 53 one step to the right. Here we did 2 comparisons. It the item was 55 instead of 33, we would have performed only one comparison. That means, if the array is already sorted then only one comparison is necessary to place each item to its appropriate place and one scan of the array would sort it. The code for insertion operation is given below.

```
void sort(int a, int n) {
    int i, j;
    for (i = 0; i < n; i++) {
       j = i-1;
```

```
        key = a[i];
        while (j >= 0 && a[j] > key)
        {
            a[j+1] = a[j];
            j = j-1;
        }
        a[j+1] = key;
    }
}
```

When items are already sorted, then the `while` loop executes only once for each item. There are total `n` items, so the running time would be O(n)O(n). So, the best-case running time of insertion sort is O(n)O(n).
The best case gives us a lower bound on the running time for any input. If the best case of the algorithm is O(n)O(n) then we know that for any input the program needs *at least* O(n)O(n) time to run. In reality, we rarely need the best case for our algorithm. We never design an algorithm based on the best-case scenario.