1. For a sorted array, which of the following algorithms will take less time? Explain with examples.
2. Bubble sort
3. Insertion sort
4. Counting sort

**Answer:**

Before coming to conclusion let’s see how all these sorting algorithms works and what’s their time complexities. To keep the answer brief I won’t be presenting visual analysis.

**Bubble Sort**

In bubble sort suppose we have 5 elements. In each iteration start from beginning and we take element “i" and “i+1” and compare them. We swap them if requires and follow this process till last element. After ith iteration our last i number of elements are sorted. Now how many iterations do we need that depends on how many elements are sorted in original form.

In best case scenario we get time complexity of **O(n)**But in worst case we end up with time complexity of **O(n2)**

**Insertion Sort**

In insertion sort we consider the first element as sorted. We put the first element on the sorted side and consider the rest elements on the unsorted side. Then we start with the second one as **i**. We take ith element and put it on the sorted side on its place. We follow this process till the end of the array. In each iteration our ith element might not even travel to the left, or in worst case it can travel to the most left position of the array.

In best case scenario we get time complexity of **O(n)**But in worst case we end up with time complexity of **O(n2)**

**Counting Sort**

In counting sort we count the frequency of elements first for which we require an extra array which will be of size of the max valued element from the array. Then we convert the frequency array to a prefix sum array. We need another array where our sorted values will be stored. No we start from the end of the original array and after calculated references from frequency array we get the position where the value should be put on the final array.

In this method we get time complexity of **O(n)** on steps of frequency array creation then prefix sum array and then positioning on final array.  
So in worst case we end up with time complexity of **O(n)**

**Conclusion:**

It’s clear that of above three algorithms we get the best time complexity result from **Counting Sort** which is **O(n)**

But it is to be noted that counting sort cannot be applied where memory space is a concern. Because while creating frequency array if the size of max is too large we will end up creating an unnecessarily large frequency array which will consume large memory space unnecessarily.