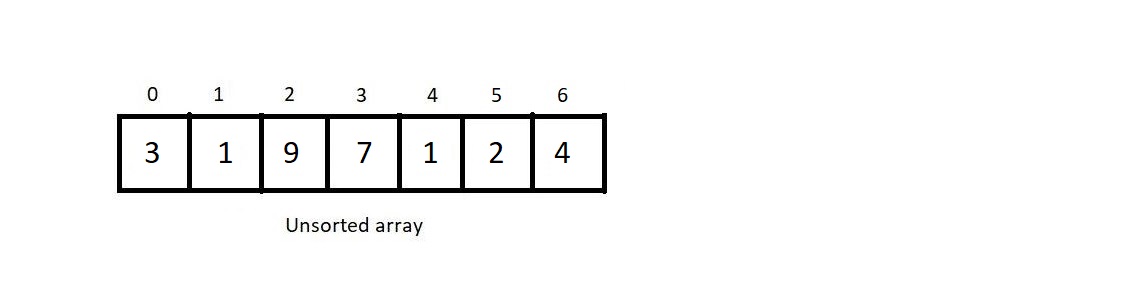
**Count Sort Algorithm**

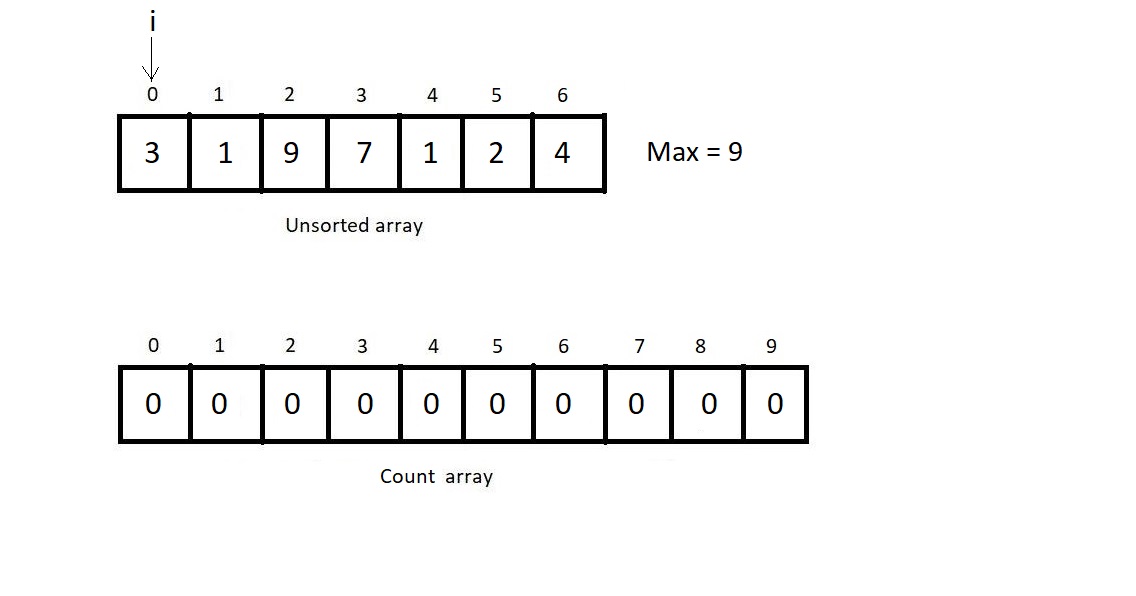
In the last lecture, we finished learning the merge sort algorithm and its implementation in C. Finally, we have moved to our last sorting algorithm. I know things must have felt tough at times, but believe me, you have finished learning a few of the most important algorithms of all. Today, we’ll start with the most intuitive and easiest sorting algorithm, known as the **count sort algorithm**.

Suppose we are given an array of integers and are asked to sort them using any sorting algorithm of our choice, then it is not difficult to generate the resultant array, which is just the sorted form of the given array. Still, the method you choose to reach the result matters the most.  Count Sort is one of the fastest methods of all. We will discuss constraints later, which would make you wonder why we don't use just this. We will do all the analysis, but before that, let’s see what count sort is. The below figure shows the array given.

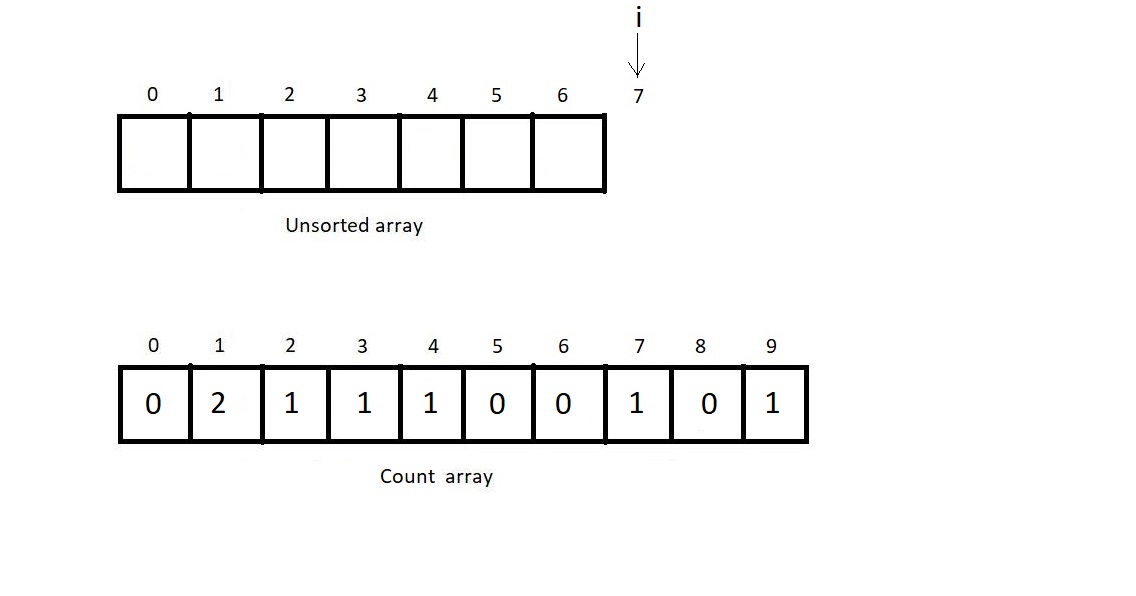


1. The algorithm first asks you to find the largest element from all the elements in the array and store it in some integer variable *max*. Then create a count array of size *max+1.*This array would count the no. of occurrences of some number in the given array. We will have to initialize all count array elements with 0 for that to work.
2. After initializing the count array, traverse through the given array, and increment the value of that element in the count array by 1. By defining the size of the count array as the maximum element in the array, you ensure that each element in the array has its own corresponding index in the count array. After we traverse through the whole array, we’ll have the count of each element in the array.
3. Now traverse through the count array, and look for the nonzero elements. The moment you find an index with some value other than zero, fill in the sorted array the index of the non-zero element until it becomes zero by decrementing it by 1 every time you fill it in the resultant array. And then move further. This way, you create a sorted array. Let’s take the one we have as an example above and use the count sort algorithm to sort it.

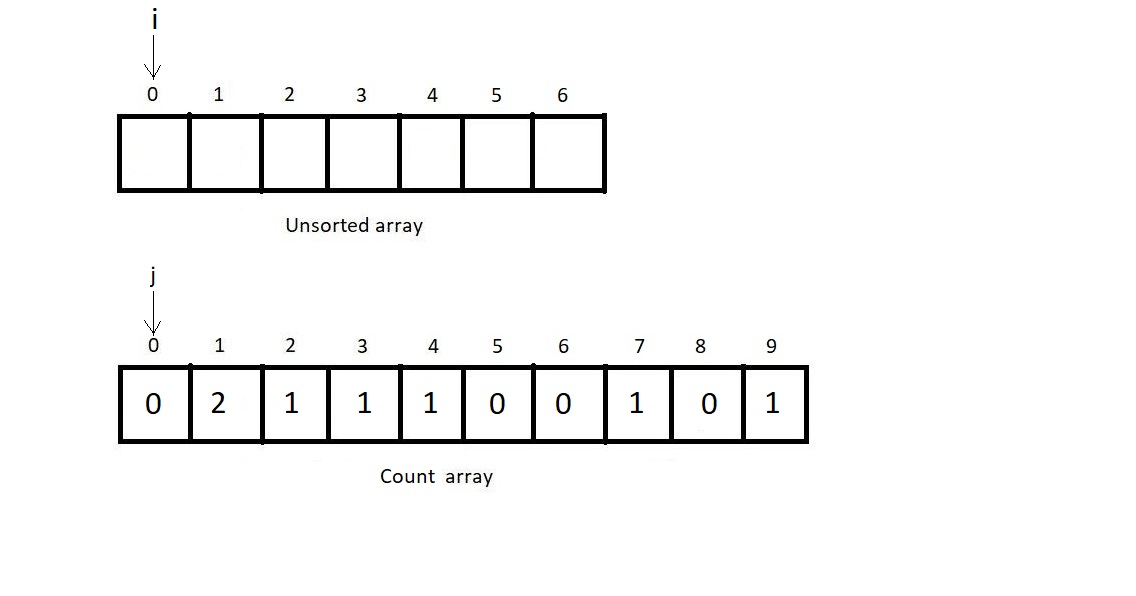
First of all, find the maximum element in the array. Here, it is 9. So, we’ll create a count array of size 10 and initialize every element by 0.



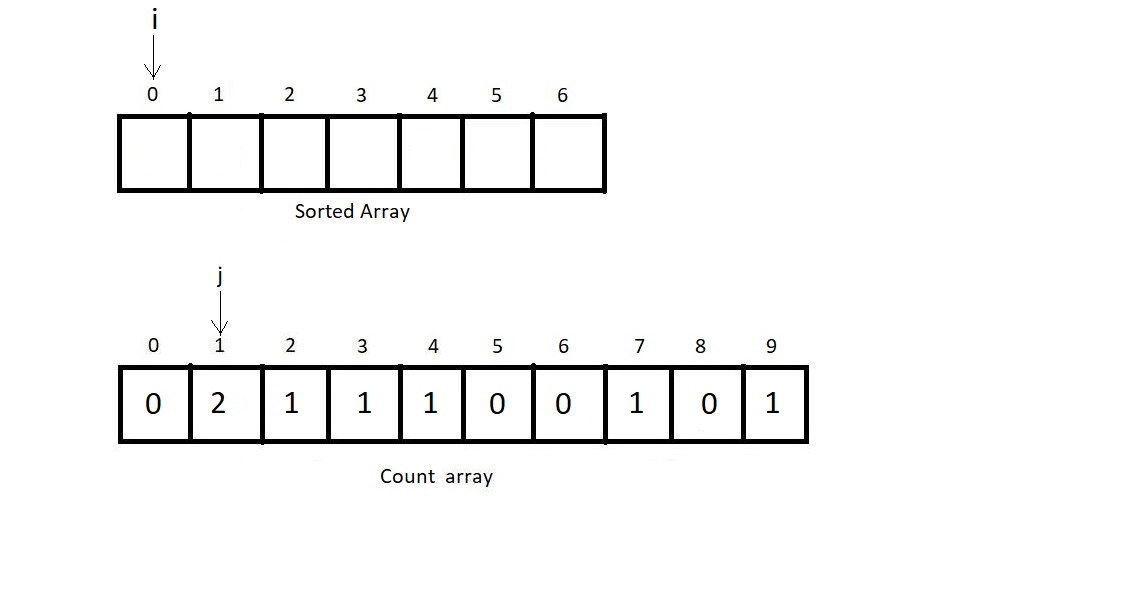
Now, let’s iterate through the given array and count the no. of occurrences of each number less than equal to the maximum element.



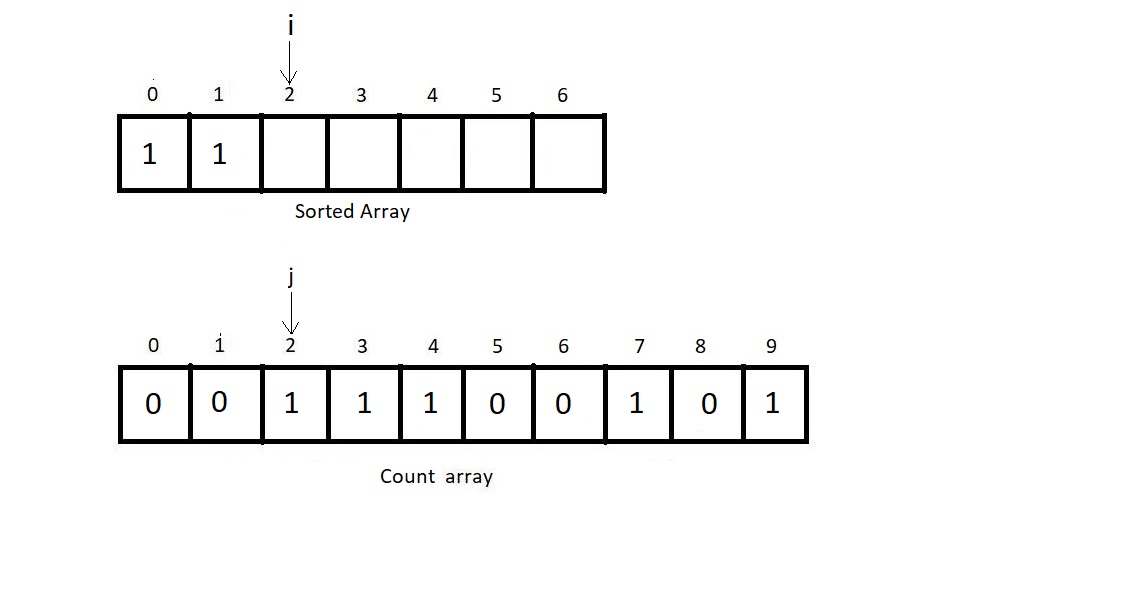
We would iterate through the count array and fill the unsorted array with the index we encounter having a non-zero number of occurrences.



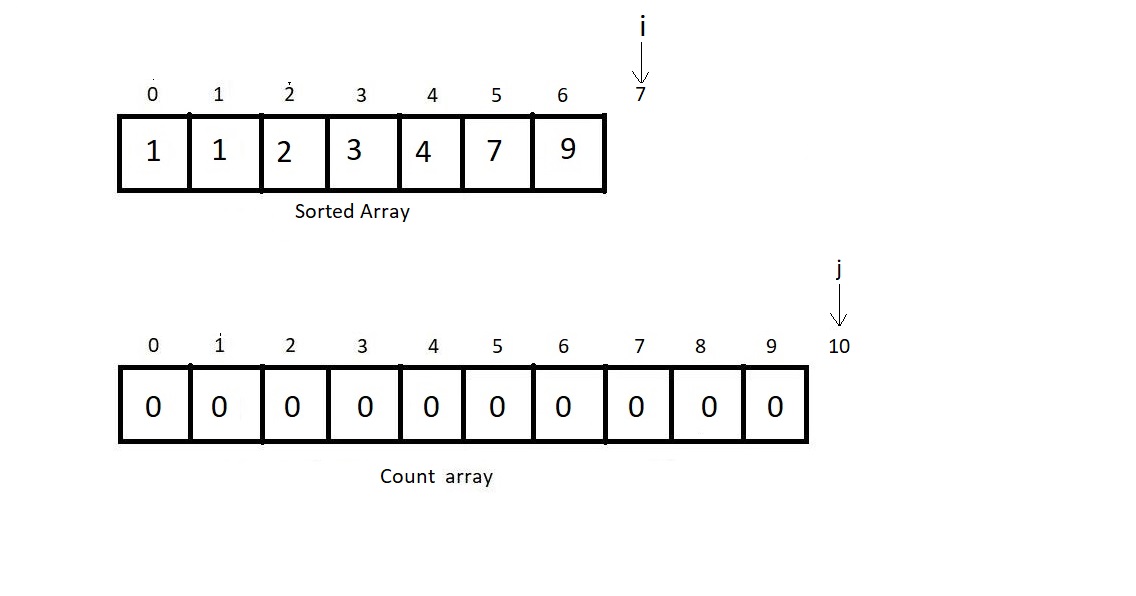
Now since there are zero numbers of zeros in the given array, we move further to index 1.



And since there were two ones in the array we were given, we push two ones in the sorted array and move our iterator to the next empty index.



And following a similar procedure for all the elements in the count array, we reach our sorted array in no time.



And it was this easy to sort the array. Tell me you found it as easy as I promised in the beginning. Having finished discussing the functioning of the count sort algorithm, let’s now move to the programming part. I have attached the source code below. Follow it as we proceed.

Time Complexity of Count Sort:

Calculating the time complexity of the count sort algorithm is one of the easiest jobs to do. If you carefully observe the whole process, we only ran two different loops, one through the given array and one through the count array, which had the size equal to the maximum element in the given array. If we suppose the maximum element to be m, then the algorithm's time complexity becomes**O(n+m),**and for an array of some huge size, this reduces to just **O(n)**, which is the most efficient by far algorithm. However, there is a negative point as well. The algorithm uses extra space for the count array.  And this linear complexity is reachable only at the cost of the space the count array takes.