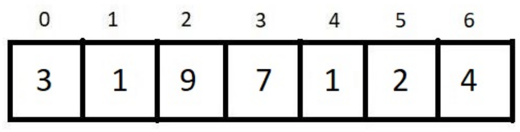
Count Sort Algorithm

codewithharry.com/videos/data-structures-and-algorithms-in-hindi-60

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**.

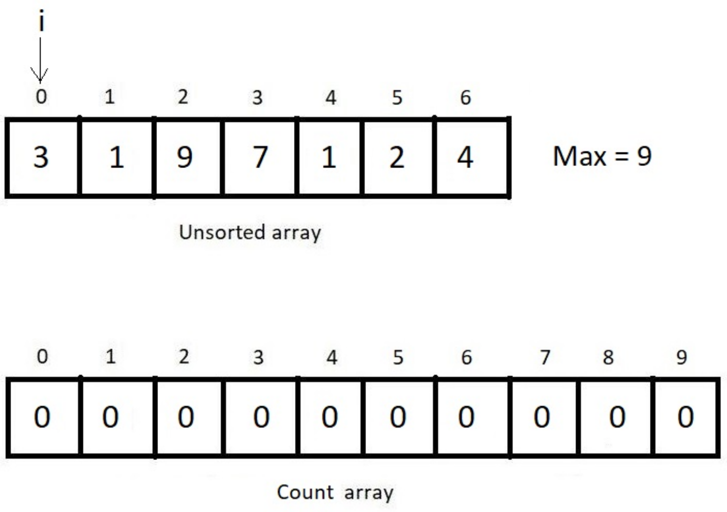
Suppose we are given an array of integers and are asked to sort them using any sorting algorithm of our choice, then it is not difficult to generate the resultant array, which is just the sorted form of the given array. Still, the method you choose to reach the result matters the most. Count Sort is 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.



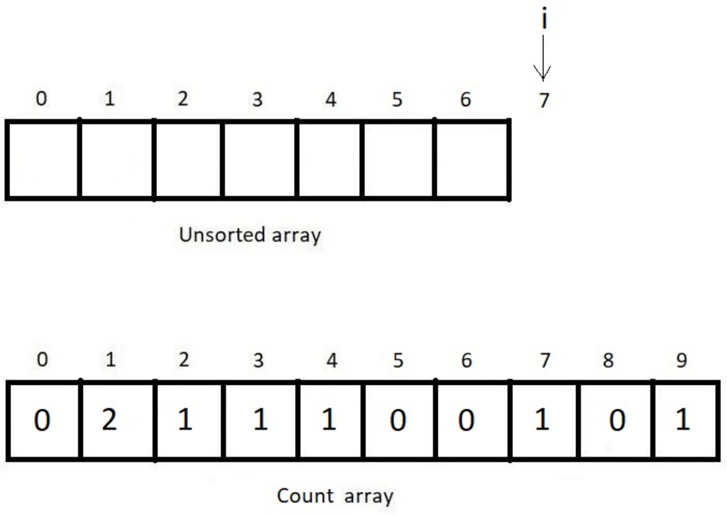
Unsorted array

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.

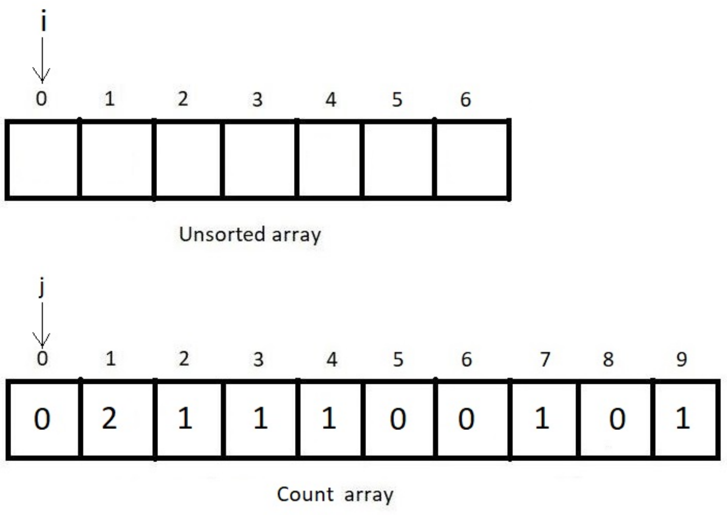
First of all, find the maximum element in the array. Here, it is 9. So, we'll create a count array of size 10 and initialize every element by 0.



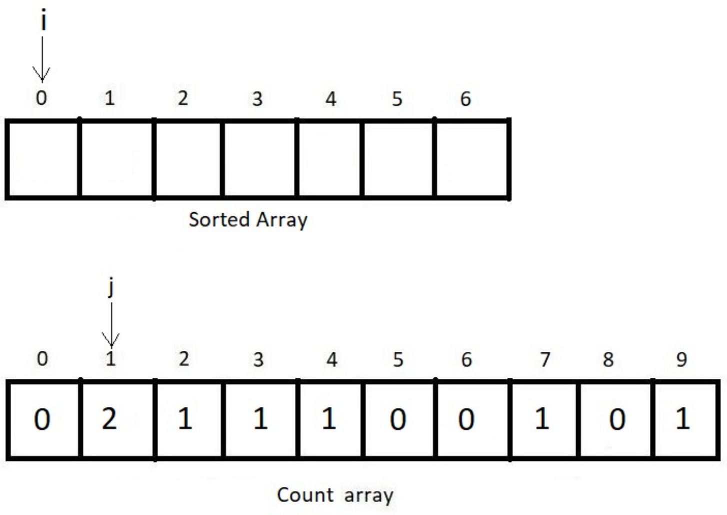
Now, let's iterate through the given array and count the no. of occurrences of each number less than equal to the maximum element.



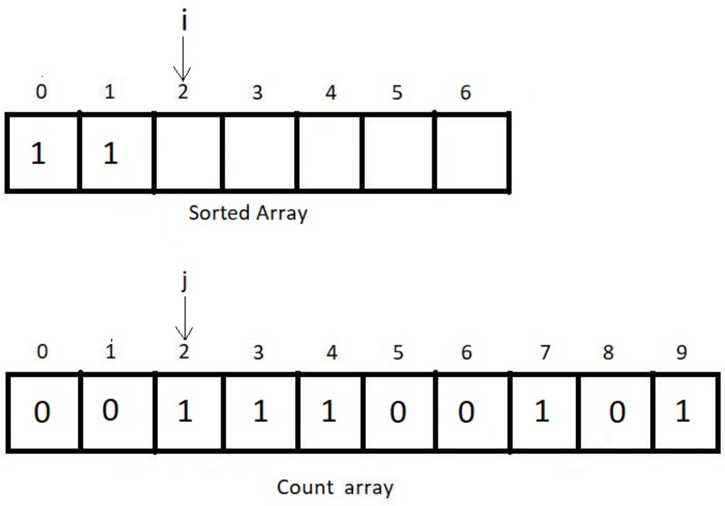
We would iterate through the count array and fill the unsorted array with the index we encounter having a non-zero number of occurrences.



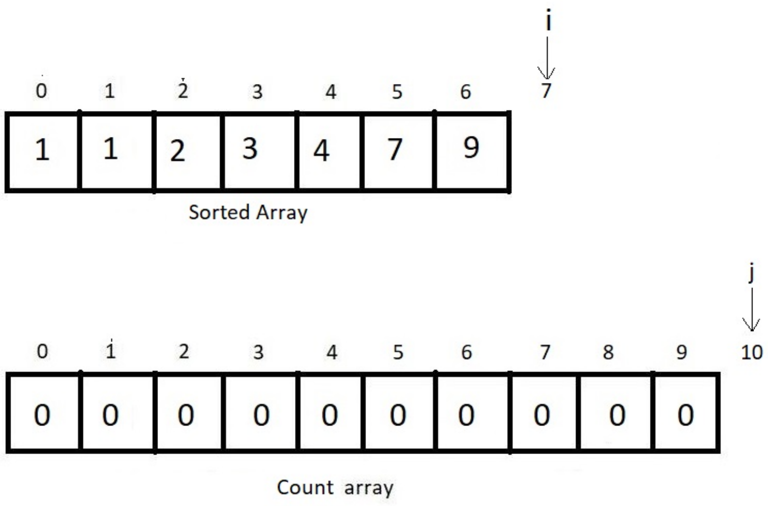
Now since there are zero numbers of zeros in the given array, we move further to index 1.



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.

Understanding the code snippet below:

1. Before we proceed with the actual function related to count sort, let's copy the *printArray* and the array declaration part from the previous lecture in our current program as well. This saves us time and helps us a lot to see the contents of the array before and after the sorting. I have attached the snippet for *printArray*

```
void printArray(int* A, int n){
    for (int i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
}
```

Code Snippet 1: Creating the *printArray* function

2. Now, to proceed with the *countSort* function, we would need a *maximum* function, which would return the maximum of all elements in the array given.

Creating the maximum function:

3. Create an integer function maximum and pass the array and its length as its parameters. Create an integer variable *max* to store the maximum element. Initialize this *max* with the least possible number we have, which is To use this identifier; you must include <limits.h>. Now iterate through the whole array using a for loop, and if you find an element greater than the current *max*, update *max*. At the end, return *max*.

```
int maximum(int A[], int n){
    int max = INT_MIN;
    for (int i = 0; i < n; i++)
    {
        if (max < A[i]){
            max = A[i];
        }
    }
    return max;

}
```

Code Snippet 2: Creating the *maximum* function

Creating the countSort function:

4. Create a void function *countSort* and pass the array and its length as its parameters. Create an integer variable *max* to store the maximum element which you get by calling the *maximum* function we made above. Next, create an integer array *count* and assign it memory dynamically using malloc of the size *max+1*. Don't forget to include <stdlib.h> to be able to use malloc.

Initialize the whole *count* array by simply using a for a loop.

Run another for loop to iterate through the given array and increase the value of the corresponding element index in the *count* array by 1.

5. Now, since the count array has been populated, create two index variables, *i and j,* to iterate through the count and the given array, respectively. Run a while loop until we reach the end of the count array. Inside the loop, check if the element at

the current index in the count array is non-zero. If it is, insert i at the j^{th} index of the given array and decrement the element in the count array at i^{th} index by 1 and simultaneously increase the value of j by 1. Repeat this until the element at the i^{th} index becomes zero or if it is already zero, increase i by 1.

The array becomes sorted once all the processes listed above are complete.

```
void countSort(int * A, int n){
    int i, j;
    // Find the maximum element in A
    int max = maximum(A, n);

    // Create the count array
    int* count = (int *) malloc((max+1)*sizeof(int));

    // Initialize the array elements to 0
    for (i = 0; i < max+1; i++)
    {
        count[i] = 0;
    }

    // Increment the corresponding index in the count array
    for (i = 0; i < n; i++)
    {
        count[A[i]] = count[A[i]] + 1;
    }

    i =0; // counter for count array
    j =0; // counter for given array A

    while(i<= max){
        if(count[i]>0){
            A[j] = i;
            count[i] = count[i] - 1;
            j++;
        }
        else{
            i++;
        }
    }
}
```

Code Snippet 3: Creating the *countSort* function

Here is the whole source code:

```c
#include<stdio.h>
#include<limits.h>
#include<stdlib.h>

void printArray(int *A, int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
}


int maximum(int A[], int n){
    int max = INT_MIN;
    for (int i = 0; i < n; i++)
    {
        if (max < A[i]){
            max = A[i];
        }
    }
    return max;

}
void countSort(int * A, int n){
    int i, j;
    // Find the maximum element in A
    int max = maximum(A, n);

    // Create the count array
    int* count = (int *) malloc((max+1)*sizeof(int));

    // Initialize the array elements to 0
    for (i = 0; i < max+1; i++)
    {
        count[i] = 0;
    }

    // Increment the corresponding index in the count array
    for (i = 0; i < n; i++)
    {
        count[A[i]] = count[A[i]] + 1;
    }

    i =0; // counter for count array
    j =0; // counter for given array A

    while(i<= max){
        if(count[i]>0){
            A[j] = i;
            count[i] = count[i] - 1;
            j++;
        }
        else{
            i++;
        }
    }
}
```

```
int main(){
    int A[] = {9, 1, 4, 14, 4, 15, 6};
    int n = 7;
    printArray(A, n);
    countSort(A, n);
    printArray(A, n);
    return 0;
}
```

Code Snippet 4: Program to implement the count Sort Algorithm

Let us now check if our functions work well. Consider an array A of length 7.

```
    int A[] = {9, 1, 4, 14, 4, 15, 6};
    int n = 7;
    printArray(A, n);
    countSort(A, n);
    printArray(A, n);
```

Code Snippet 5: Using the *countSort* function

And the output we received was:

```
9 1 4 14 4 15 6
1 4 4 6 9 14 15
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 1: Output of the above program

So, our array got sorted. That was easy for you to understand, I believe. And it was indeed as easy as pie. We'll now quickly move to the analysis part and wrap up our sorting algorithm series.

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.

Now that we have completed all our sorting algorithms, I expect you all to take your own unsorted array and use all the algorithms we have learned to sort it. That would make you confident about using them. Do refer to the lectures whenever you feel like revising things. Several more interesting topics are coming up, so stay tuned.

I appreciate your support throughout. I hope you enjoyed the tutorial. If you genuinely appreciate my work, please let your friends know about this course too. If you haven’t checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial, where we'll start with our new topic, another data structure known as **trees.** Till then, keep coding.