Counting Sort Algorithm

Counting sort is a very time-efficient (and somewhat space-inefficient) algorithm for sorting that avoids comparisons and exploits the $O(1)$ time insertions and lookups in an array.

The idea is simple: if you're sorting integers and you know they all fall in the range $1..100$, you can generate a sorted array this way:

- Allocate an array `numCounts` where the indices represent numbers from our input array and the values represent how many times the index number appears. Start each value at 0.
- In one pass of the input array, update `numCounts` as you go, so that at the end the values in `numCounts` are correct.
- Allocate an array `sortedArray` where we'll store our sorted numbers.
- In one in-order pass of `numCounts` put each number, the correct number of times, into `sortedArray`.

C# (beta)

```
public int[] CountingSort(int[] theArray, int maxValue)
{
    // Array of 0's at indices 0...maxValue
    int[] numCounts = new int[maxValue + 1];

    // Populate numCounts
    foreach (var num in theArray)
    {
        numCounts[num]++;
    }

    // Populate the final sorted array
    int[] sortedArray = new int[theArray.Length];
    int currentSortedIndex = 0;

    // For each num in numCounts
    for (int num = 0; num < numCounts.Length; num++)
    {
        int count = numCounts[num];

        // For the number of times the item occurs
        for (int i = 0; i < count; i++)
        {
            // Add it to the sorted array
            sortedArray[currentSortedIndex] = num;
            currentSortedIndex++;
        }
    }

    return sortedArray;
}
```

Counting sort takes $O(n)$ time and $O(n)$ additional space (for the new array that we end up returning).

> **Wait, aren't we nesting two loops towards the bottom? So shouldn't it be $O(n^2)$ time?** Notice *what those loops iterate over*. The *outer* loop runs once for each *unique* number in the array. The *inner* loop runs once for each *time that number occurred*.

> So in essence we're just looping through the n numbers from our input array, except we're splitting it into two steps: (1) each unique number, and (2) each time that number appeared.
>
> Here's another way to think about it: in each iteration of our two nested loops, we append one item to `sortedArray`. How many numbers end up in `sortedArray` in the end? Exactly how many were in our input array! n!

There are some rare cases where even though our input items aren't integers bound by constants, we can write a function that *maps* our items to integers from 0 to some constant such that different items will always map to different integers. This allows us to use counting sort.

What's next?

If you're ready to start applying these concepts to some problems, check out our mock coding interview questions (/next).

They mimic a real interview by offering hints when you're stuck or you're missing an optimization.

Try some questions now ➜