Theory: Counting sort

20 minutes 7 / 7 problems solved

Start practicing

484 users solved this topic. Latest completion was about 2 hours ago.

Counting sort is a non-comparison-based algorithm for sorting small integer numbers. The algorithm counts the number of occurrences for each element in the input array, storing the counts in a separate array. Then the algorithm returns array elements from the minimum to the maximum accordingly to their counts.

§1. Properties of the counting sort

- Counting sort is efficient when a range of input data is not significantly bigger than the size of input data;
- The algorithm can also sort other data types mappable to integers;
- An extended counting sort can work for negative integers;
- The time complexity is $O(n+k)$ where n is the number of elements in the input array and k is the range of the input;
- the total space usage of the algorithm is also $O(n+k)$.

The algorithm can be **stable** and **unstable**. We will consider both versions.

§2. The simplest unstable algorithm

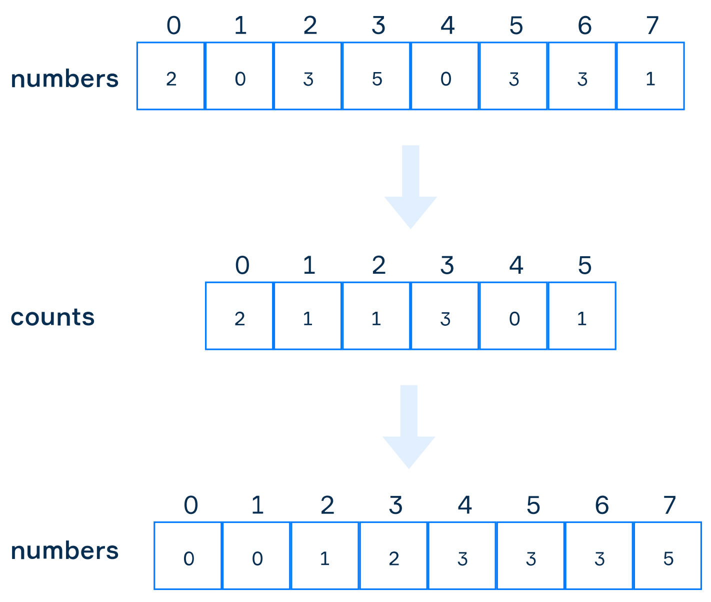
The simplest version of the algorithm is **unstable**. It works as follows.

The input array **numbers** of the length **n** contains integer numbers from the range $[0, k-1]$. Also, there is an additional array **counts** of the length k.

1. We go through the numbers array and write in $counts[numbers[i]]$ the count of array elements equal to $numbers[i]$.
2. Now it's enough to go through the **counts** array and write each number **num** in an array exactly $counts[num]$ times.

To spend less additional memory, let's write the result in the input array **numbers**.

The following picture demonstrates the algorithm:



§3. The stable counting sort

Often we need a **stable** sorting algorithm to sort complex objects by keys.

To make the counting sort stable, after filling the **counts** array, we will modify it by adding the previous count to the next. We will store our **cumulative counts** in the same **counts** array.

Here is an algorithm with the same notation as above.

1. We go through **numbers** and write in $counts[i]$ the count of array elements equal to i.
2. We modify the array **counts**: $counts[i]$ stores the count of elements from 0 to i.

Current topic:

✓ Counting sort

Topic depends on:

✓ The sorting problem

Topic is required for:

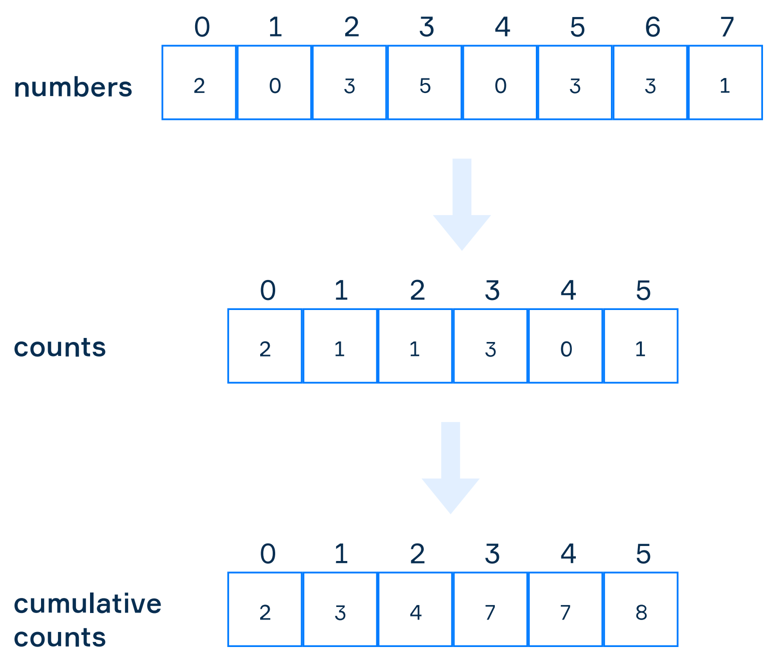
Counting sort in Java

Table of contents:

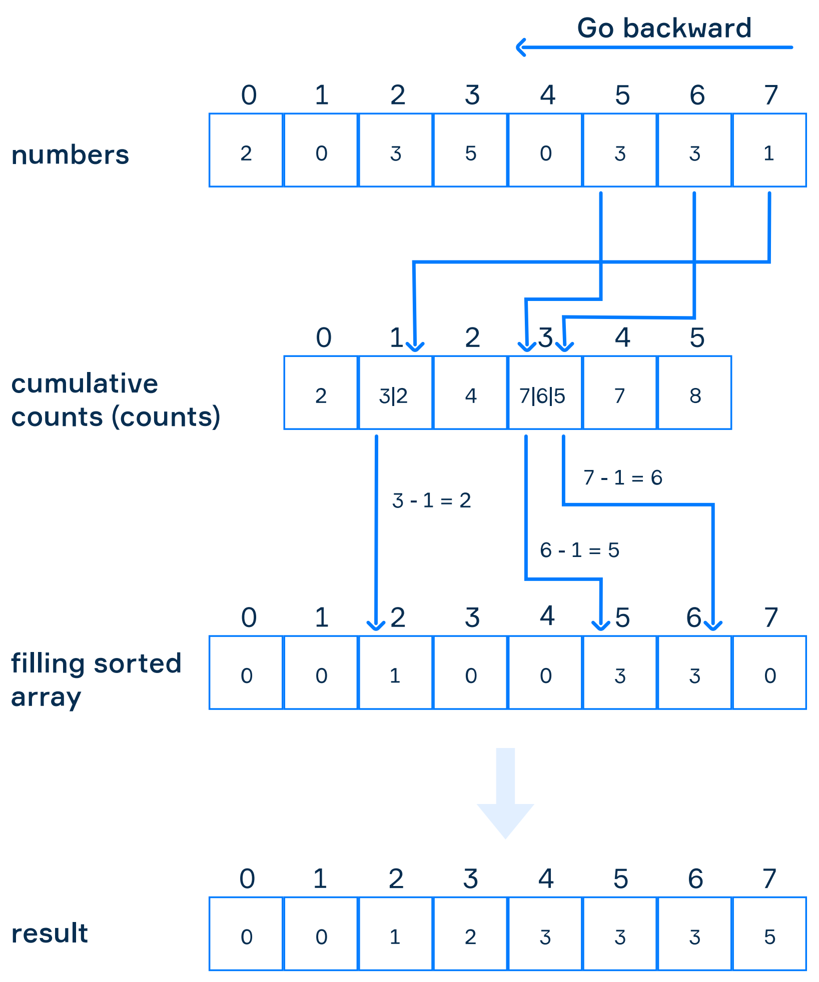
3. We go through **numbers** backward and write $numbers[i]$ on the $counts[numbers[i]] - 1$ position in a new array. At the same time, we decrease $counts[numbers[i]]$ by one after each writing.

Actually, $counts[numbers[i]] - 1$ is the last occurrence of $numbers[i]$ in the sorted array (the rightmost index). When we decrease it, it means that the next $numbers[i]$ will have the previous index in the sorted array.

The following pictures demonstrates the stable counting sort algorithm. The first one shows calculating cumulative **counts** (modified counts), and the second one shows how to write elements to the sorted array.



After we have calculated cumulative counts and modified the array **counts** to store them, we can write elements to the sorted array.



To write elements to the sorted array, we go through the input array backward and get the rightmost index of elements in the sorted array. To get the rightmost index, we access the modified **counts** array.

1) We want to know if the index of the element $numbers[7]$ is equal to 1 in the sorted array. We know that the value $counts[1]$ is 3 and so, we calculate $3 - 1 = 2$. It means that the element 1 has the index 2 in the sorted array. Then we decrease $counts[numbers[7]]$ by one.

2) Then we want to know if the index of the element $numbers[6]$ is equal to 3 in the sorted array. We know that value $counts[3]$ is equal to 7 and so, we calculate $7 - 1 = 6$, which means that the element has the index 6 in the sorted array. Do not forget to decrease $numbers[6]$ by one. The following elements equal to 3 will have a smaller index because we decrease $counts[numbers[6]]$ by one.

We repeat the same steps for all other elements from the input array.

This version of the algorithm may be more challenging to understand. For better understanding, here is a visualization of the stable counting sort.

§4. When the counting sort is used

Counting sort is the correct choice of an algorithm, if:

- the input array consists of integers or objects mappable to integers (characters, key-value pairs with small keys, and so on);
- we know the range of elements;
- most of the range elements are existing in the input array;
- additional memory usage is not an issue.

Stable counting sort is used to implement other non-comparison-based sorting algorithms such as **radix sort.**

§5. Finding the indices range

If the range of elements is not known, we can find it using the linear search of the minimum and the maximum.

If the minimum is negative, we can increase all values by the minimum and then decrease them after sorting.