<https://www.happycoders.eu/algorithms/counting-sort/>

# Counting Sort – Algorithm, Source Code, Time Complexity

All sorting methods discussed so far in this [article series](https://www.happycoders.eu/algorithms/sorting-algorithms/) are based on comparing whether two numbers are smaller, larger or equal. Counting Sort is based on a completely different, non-comparison approach.

This article answers the following questions:

* How does Counting Sort work?
* What is the difference between the simplified form of Counting Sort and its general form?
* What does the source code of Counting Sort look like?
* How to determine the time complexity of Counting Sort?
* Why is Counting Sort almost ten times faster for presorted number sequences than for unsorted ones despite the same number of operations?

**Contents**  [hide](https://www.happycoders.eu/algorithms/counting-sort/)

**[1 Counting Sort Algorithm (Simplified Form)](https://www.happycoders.eu/algorithms/counting-sort/" \l "Counting_Sort_Algorithm_Simplified_Form)**

[1.1 Counting Sort Algorithm – Phase 2: Counting the Elements](https://www.happycoders.eu/algorithms/counting-sort/" \l "Counting_Sort_Algorithm_-_Phase_2_Counting_the_Elements)

[1.2 Counting Sort Algorithm – Phase 2: Rearranging the Elements](https://www.happycoders.eu/algorithms/counting-sort/" \l "Counting_Sort_Algorithm_-_Phase_2_Rearranging_the_Elements)

**[2 Counting Sort Java Code Example (Simplified Form)](https://www.happycoders.eu/algorithms/counting-sort/" \l "Counting_Sort_Java_Code_Example_Simplified_Form)**

[2.1 Counting Sort Source Code Also for Negative Numbers](https://www.happycoders.eu/algorithms/counting-sort/" \l "Counting_Sort_Source_Code_Also_for_Negative_Numbers)

**[3 Counting Sort Algorithm (General Form)](https://www.happycoders.eu/algorithms/counting-sort/" \l "Counting_Sort_Algorithm_General_Form)**

[3.1 General Algorithm – Phase 1: Counting the Elements](https://www.happycoders.eu/algorithms/counting-sort/" \l "General_Algorithm_-_Phase_1_Counting_the_Elements)

[3.2 General Algorithm – Phase 2: Aggregating the Histogram](https://www.happycoders.eu/algorithms/counting-sort/" \l "General_Algorithm_-_Phase_2_Aggregating_the_Histogram)

[3.3 General Algorithm – Phase 3: Writing Back Sorted Objects](https://www.happycoders.eu/algorithms/counting-sort/" \l "General_Algorithm_-_Phase_3_Writing_Back_Sorted_Objects)

**[4 Counting Sort Java Code Example (General Form)](https://www.happycoders.eu/algorithms/counting-sort/" \l "Counting_Sort_Java_Code_Example_General_Form)**

**[5 Counting Sort Time Complexity](https://www.happycoders.eu/algorithms/counting-sort/" \l "Counting_Sort_Time_Complexity)**

[5.1 Runtime of the Java Counting Sort Example](https://www.happycoders.eu/algorithms/counting-sort/" \l "Runtime_of_the_Java_Counting_Sort_Example)

[5.2 Why Is Counting Sort Faster for Presorted Elements Than for Unsorted Ones?](https://www.happycoders.eu/algorithms/counting-sort/" \l "Why_Is_Counting_Sort_Faster_for_Presorted_Elements_Than_for_Unsorted_Ones)

[5.3 Why Doesn't Counting Sort Achieve Linear Time Complexity for Unsorted Output Sequences in Practice?](https://www.happycoders.eu/algorithms/counting-sort/" \l "Why_Doesnt_Counting_Sort_Achieve_Linear_Time_Complexity_for_Unsorted_Output_Sequences_in_Practice)

[5.4 Why Is Counting Sort Faster for Items Sorted in Ascending Order Than for Items Sorted in Descending Order?](https://www.happycoders.eu/algorithms/counting-sort/" \l "Why_Is_Counting_Sort_Faster_for_Items_Sorted_in_Ascending_Order_Than_for_Items_Sorted_in_Descending_Order)

**[6 Further Characteristics of Counting Sort](https://www.happycoders.eu/algorithms/counting-sort/" \l "Further_Characteristics_of_Counting_Sort)**

[6.1 Space Complexity of Counting Sort](https://www.happycoders.eu/algorithms/counting-sort/" \l "Space_Complexity_of_Counting_Sort)

[6.2 Stability of Counting Sort](https://www.happycoders.eu/algorithms/counting-sort/" \l "Stability_of_Counting_Sort)

[6.3 Parallelizability of Counting Sort](https://www.happycoders.eu/algorithms/counting-sort/" \l "Parallelizability_of_Counting_Sort)

**[7 Summary](https://www.happycoders.eu/algorithms/counting-sort/" \l "Summary)**

## Counting Sort Algorithm (Simplified Form)

Instead of comparing elements, Counting Sort counts how often which elements occur in the set to be sorted.

A simplified form of Counting Sort can be used when sorting numbers (e.g., int primitives). To sort objects according to their keys, you will learn about Counting Sort's general form afterward.

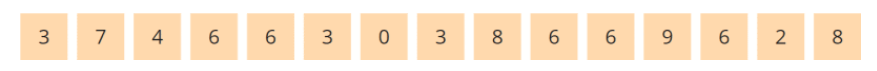
The simplified form consists of two phases:

### Counting Sort Algorithm – Phase 2: Counting the Elements

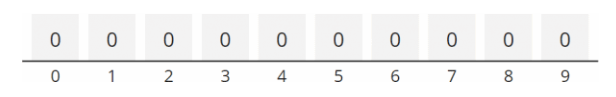
First, an auxiliary array is created whose length corresponds to the number range (e.g., an array of size 256 to sort bytes). Then you iterate once over the elements to be sorted, and, for each element, you increment the value in the array at the position corresponding to the element.

Here is an example with the number range 0–9 (i.e., the array to be sorted contains only numbers from 0 to 9).

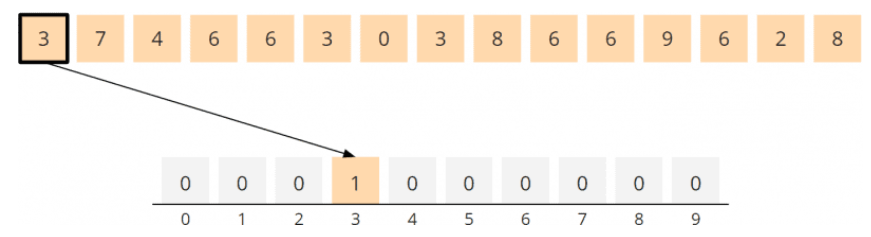
The following array shall be sorted:



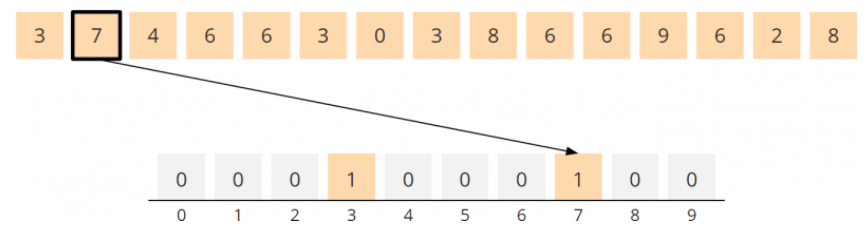
We create an additional array of length 10, initialized with zeros. In the diagram, the array index is displayed below the line:



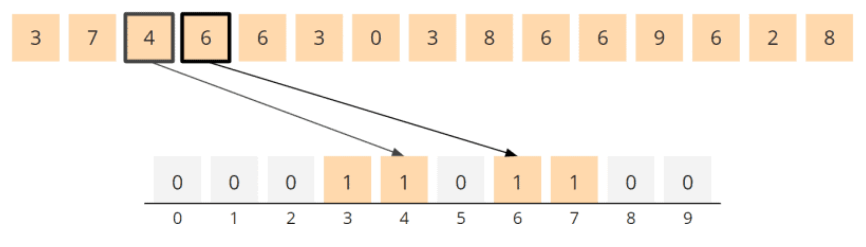
Now we iterate over the array to be sorted. The first element is a 3 – accordingly, we increase the value in the auxiliary array at position 3 by one:



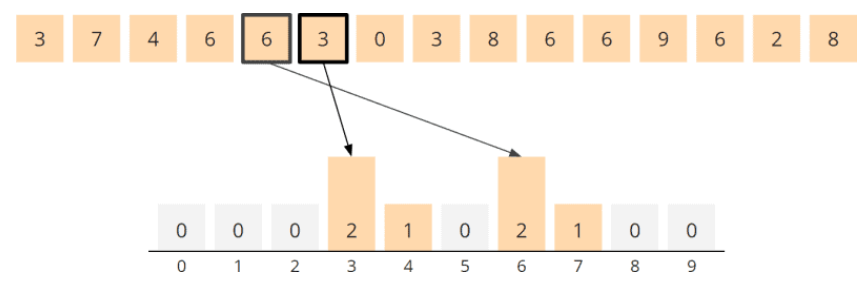
The second element is a 7. We increment the field at position 7 in the helper array:



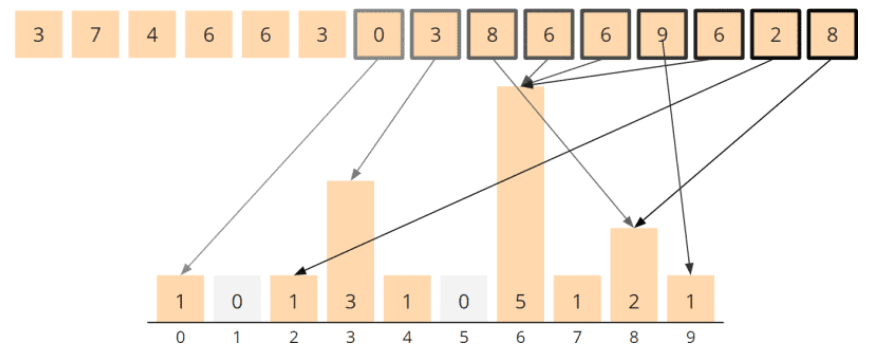
Elements 4 and 6 follow – thus, we increase the values at positions 4 and 6 by one each:



The next two elements – the 6 and the 3 – are two elements that have already occurred before. Accordingly, the corresponding fields in the auxiliary array are increased from 1 to 2:



The principle should be clear now. After also increasing the auxiliary array values for the remaining elements, the auxiliary array finally looks like this:



This so-called histogram tells us the following:

The elements to be sorted contain:

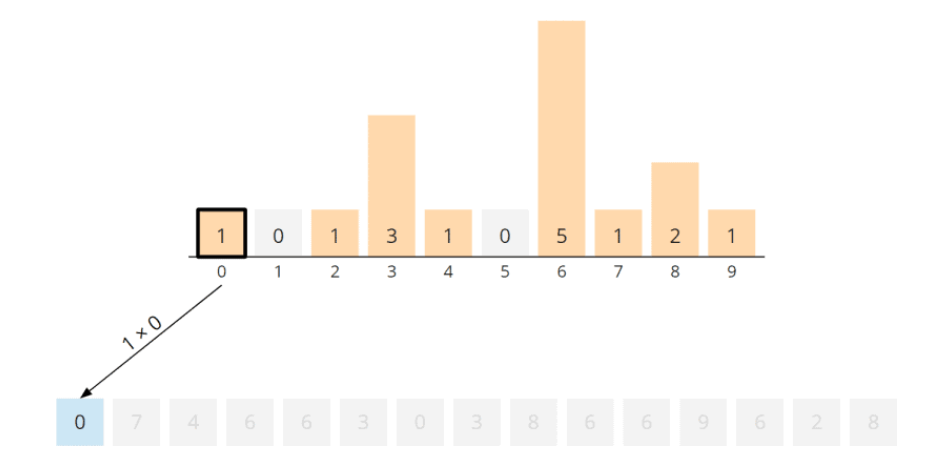
* 1 time the 0,
* 0 times the 1,
* 1 time the 2,
* 3 times the 3,
* 1 time the 4,
* 0 times the 5,
* 5 times the 6,
* 1 time the 7,
* 2 times the 8 and
* 1 time the 9.

We will use this information in phase 2 to rearrange the array to be sorted.

### Counting Sort Algorithm – Phase 2: Rearranging the Elements

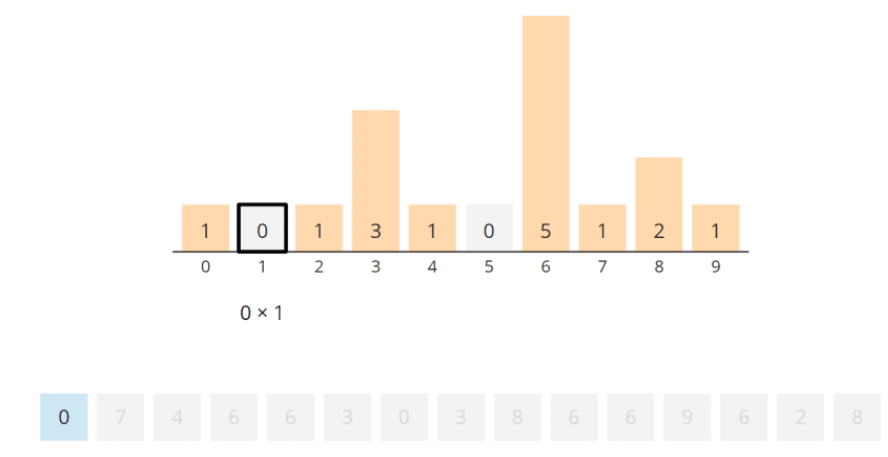
In phase two, we iterate once over the histogram array. We write the respective array index into the array to be sorted as often as the histogram indicates at the corresponding position.

In the example, we start at position 0 in the auxiliary array. That field contains a 1, so we write the 0 exactly once into the array to be sorted.

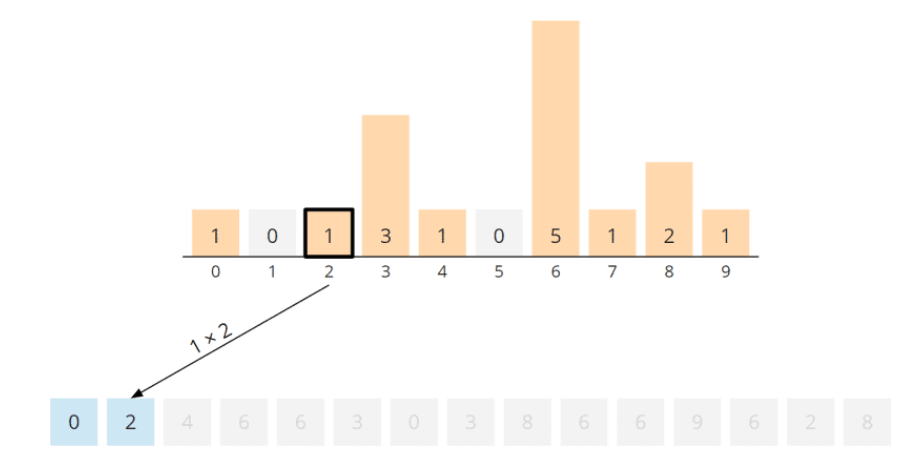


(I grayed out the rest of the numbers because they are still in the array, but we don't need them anymore. We now have this information entirely in the histogram.)

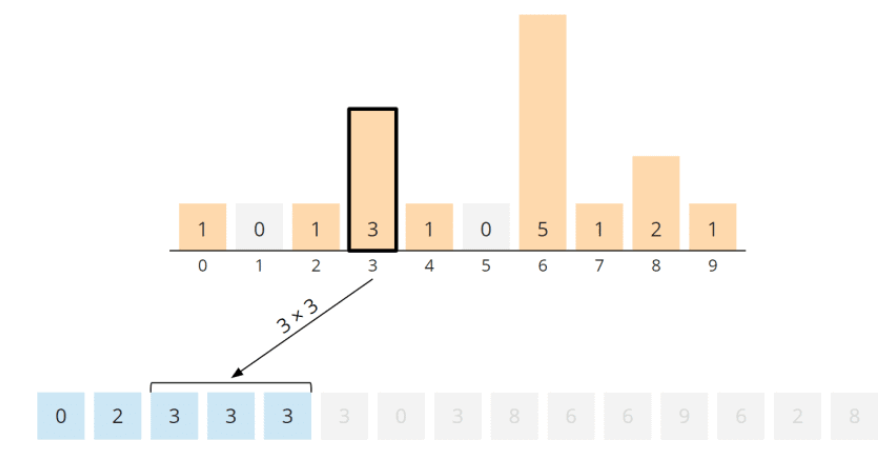
At position 1 in the histogram, there is a 0, meaning we skip this field – no 1 is written to the array to be sorted.



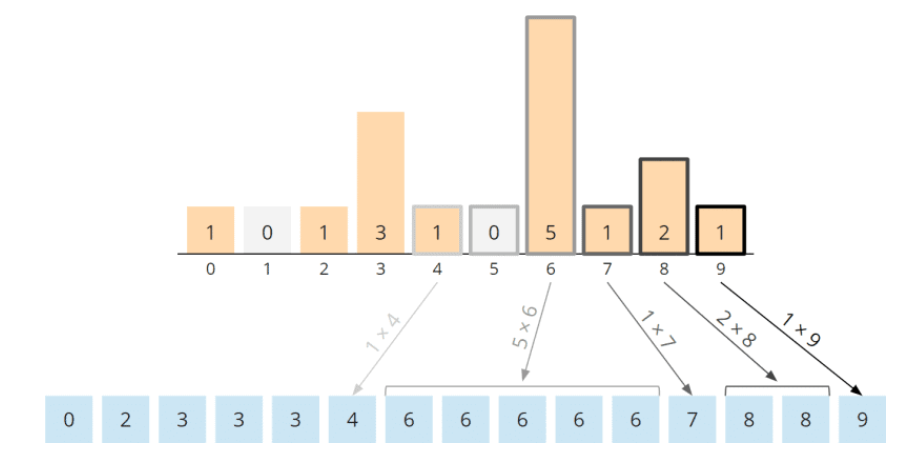
Position 2 of the histogram again contains a 1, so we write one 2 into the array to be sorted:



We come to position 3, which contains a 3; so we write three times a 3 into the array:



And so it goes on. We write once the 4, five times the 6, once the 7, twice the 8 and finally once the 9 into the array to be sorted:



The numbers are sorted; the algorithm is completed.

## Counting Sort Java Code Example (Simplified Form)

Below you'll find a simple form of the Counting Sort source code – it only works for non-negative int primitives (e.g., for the array from the example above).

First, the findMax() method is used to find the largest element in the array. Then the auxiliary array counts is created of the corresponding size, where the size is one greater than the largest element so we can count the 0 as well.

(For smaller number ranges like byte and short, you can omit to determine the maximum and directly create an array in the size of the corresponding number range.)

In the block commented with "Phase 1", the elements are counted so that the counts array eventually contains the histogram.

In the block commented with "Phase 2", the elements are written back to the array to be sorted in ascending order and according to the histogram's frequency.

public class CountingSortSimple {

  public void sort(int[] elements) {

    int maxValue = findMax(elements);

    int[] counts = new int[maxValue + 1];

    // Phase 1: Count

    for (int element : elements) {

      counts[element]++;

    }

    // Phase 2: Write results back

    int targetPos = 0;

    for (int i = 0; i < counts.length; i++) {

      for (int j = 0; j < counts[i]; j++) {

        elements[targetPos++] = i;

      }

    }

  }

  private int findMax(int[] elements) {

    int max = 0;

    for (int element : elements) {

      if (element < 0) {

        throw new IllegalArgumentException("This implementation does not support negative values.");

      }

      if (element > max) {

        max = element;

      }

    }

    return max;

  }

}

You could also determine the maximum using Arrays.stream(elements).max().getAsInt(). But then we would either have to omit the check for negative values or do it in a separate step.

You can find the code in the GitHub repository in the class [CountingSortSimple](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/countingsort/CountingSortSimple.java" \t "https://www.happycoders.eu/algorithms/counting-sort/_blank).

### Counting Sort Source Code Also for Negative Numbers

If you want to allow negative numbers too, the code gets a bit more complicated because we have to work with a so-called offset to map the number to be sorted to the auxiliary array position.

#### Calculating the Offset

The offset is: zero minus the smallest number to sort.

If, for example, -5 is the smallest number to be sorted, then the offset is 5, i.e., the index in the auxiliary array is always the number to be sorted plus 5.

For example, the -5 is counted at position -5+5 = 0; the 0 is counted at position 0+5 = 5; the 11 is counted at position 11+5 = 16.

#### Source Code

You can find the following source code in the [CountingSort](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/countingsort/CountingSort.java) class in the GitHub repository. It is similar to the source code shown above, except for the following differences:

* The method findMax() is replaced by the method findBoundaries(), which returns not only the maximum but also the minimum value (for small number ranges like byte and short, you can omit  to determine the boundaries and directly create an array in the size of the number range).
* When accessing the counts array during the counting phase, the -boundaries.min offset is added to the corresponding index (or -Byte.MIN\_VALUE or -Short.MIN\_VALUE).
* When writing back the sorted numbers into the array, the offset is subtracted again by adding boundaries.min (or Byte.MIN\_VALUE or Short.MIN\_VALUE).

public class CountingSort {

  private static final int MAX\_VALUE\_TO\_SORT = Integer.MAX\_VALUE / 2;

  private static final int MIN\_VALUE\_TO\_SORT = Integer.MIN\_VALUE / 2;

  public void sort(int[] elements) {

    Boundaries boundaries = findBoundaries(elements);

    int[] counts = new int[boundaries.max - boundaries.min + 1];

    // Phase 1: Count

    for (int element : elements) {

      counts[element - boundaries.min]++;

    }

    // Phase 2: Write results back

    int targetPos = 0;

    for (int i = 0; i < counts.length; i++) {

      for (int j = 0; j < counts[i]; j++) {

        elements[targetPos++] = i + boundaries.min;

      }

    }

  }

  private Boundaries findBoundaries(int[] elements) {

    int min = Integer.MAX\_VALUE;

    int max = Integer.MIN\_VALUE;

    for (int element : elements) {

      if (element > MAX\_VALUE\_TO\_SORT) {

        throw new IllegalArgumentException("Element " + element +

              " is greater than maximum " + MAX\_VALUE\_TO\_SORT);

      }

      if (element < MIN\_VALUE\_TO\_SORT) {

        throw new IllegalArgumentException("Element " + element +

              " is less than minimum " + MIN\_VALUE\_TO\_SORT);

      }

      if (element > max) {

        max = element;

      }

      if (element < min) {

        min = element;

      }

    }

    return new Boundaries(min, max);

  }

  private static class Boundaries {

    private final int min;

    private final int max;

    public Boundaries(int min, int max) {

      this.min = min;

      this.max = max;

    }

  }

}

This variant not only has the advantage of being able to count negative numbers but also occupies less additional memory than the first variant if the number range does not start at 0: For numbers from 1,000 to 2,000, for example, the first variant would need an auxiliary array with 2,001 fields, whereas variant 2 only needs 1,001 fields.

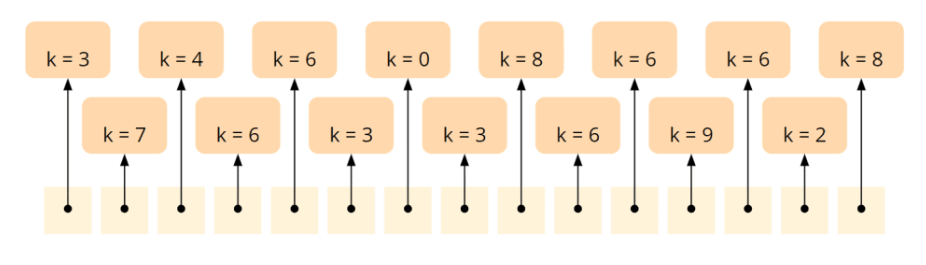
## Counting Sort Algorithm (General Form)

You can not only use Counting Sort to sort arrays of primitives (i.e., bytes, ints, longs, doubles, etc.) but also for arrays of objects. For this purpose, we have to extend the algorithm, as described in the following section.

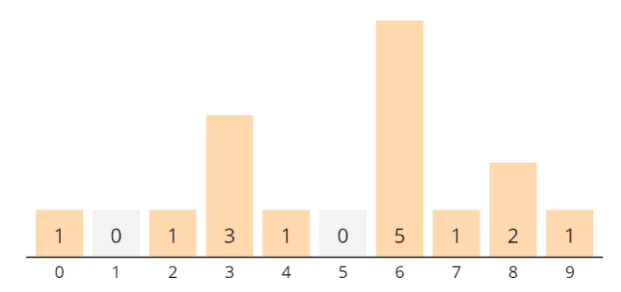
### General Algorithm – Phase 1: Counting the Elements

Phase 1, the counting phase, remains more or less unchanged. Instead of the objects themselves, their keys (determined by a getKey() method, for example) are now counted.

The array in the following image references objects whose keys correspond to the numbers in the previous example, i.e., 3, 7, 4, 6, 6, etc.:



Accordingly, the resulting histogram resembles the one from the first example:

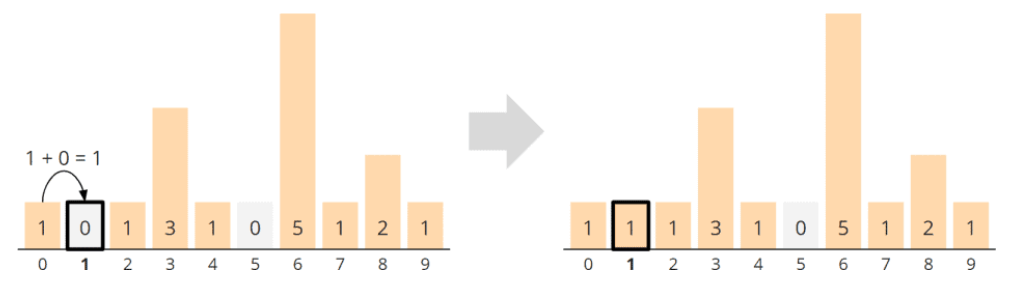


### General Algorithm – Phase 2: Aggregating the Histogram

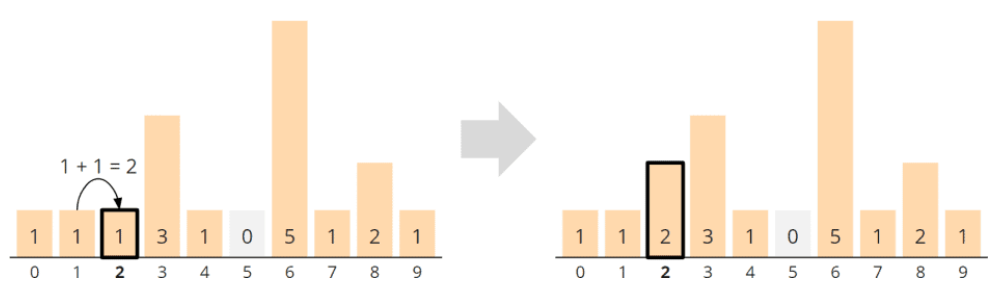
Here the difference to the simplified algorithm becomes obvious: We now know that the element with the key 0 occurs once, but we cannot merely write a 0 into the array to be sorted – we instead need the object with the key 0!

To find this efficiently, we first aggregate the values in the histogram. For this purpose, we iterate, starting at index 1, over the auxiliary array and add to each field the left neighboring field's value.

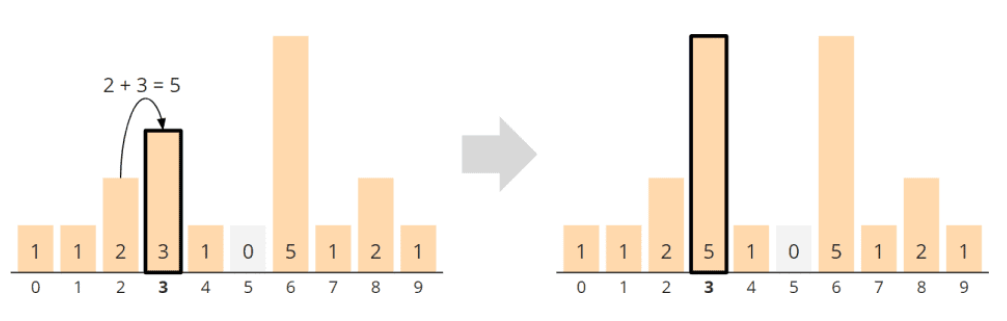
At position 1, we add to the 0 the value of field 0, the 1. The sum is 1:



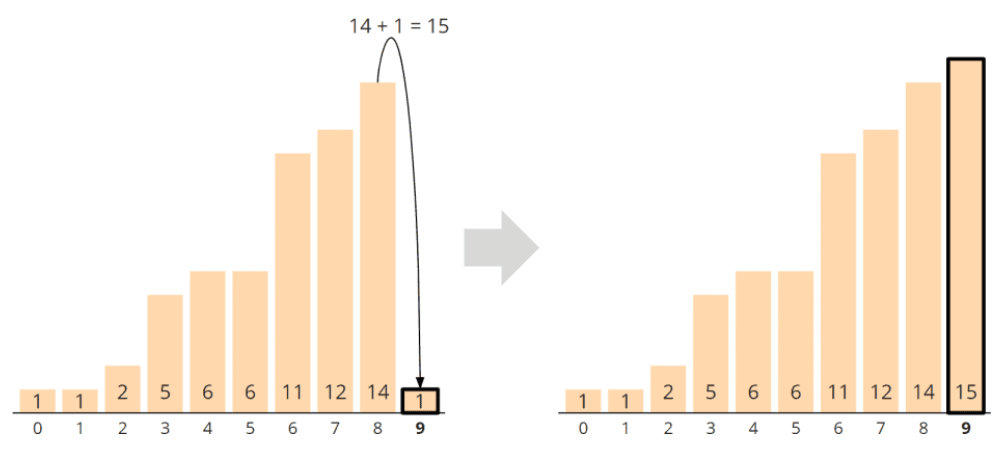
At position 2, we add to the 1 the 1 from field 1 and get a 2:



To the 3 at position 3, we add the 2 of field 2 – the sum is 5:



And so we continue until we finally add to the 1 in field 9 the 14 from field 8 to get 15:

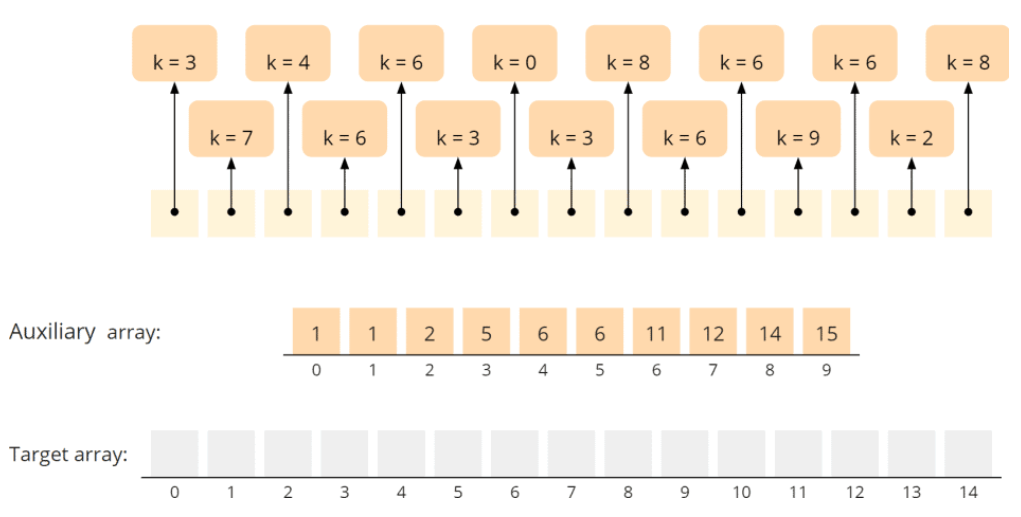


This aggregated histogram now no longer tells us how often the objects with a specific key occur, but at which position the last element with the corresponding key belongs. The position is 1-based, not 0-based.

For example, the object with key 0 belongs at position 1 (corresponds to index 0 in the array), the object with key 2 at position 2 (array index 1), and the three objects with key 3 at positions 3, 4, and 5 (array indexes 2, 3, 4).

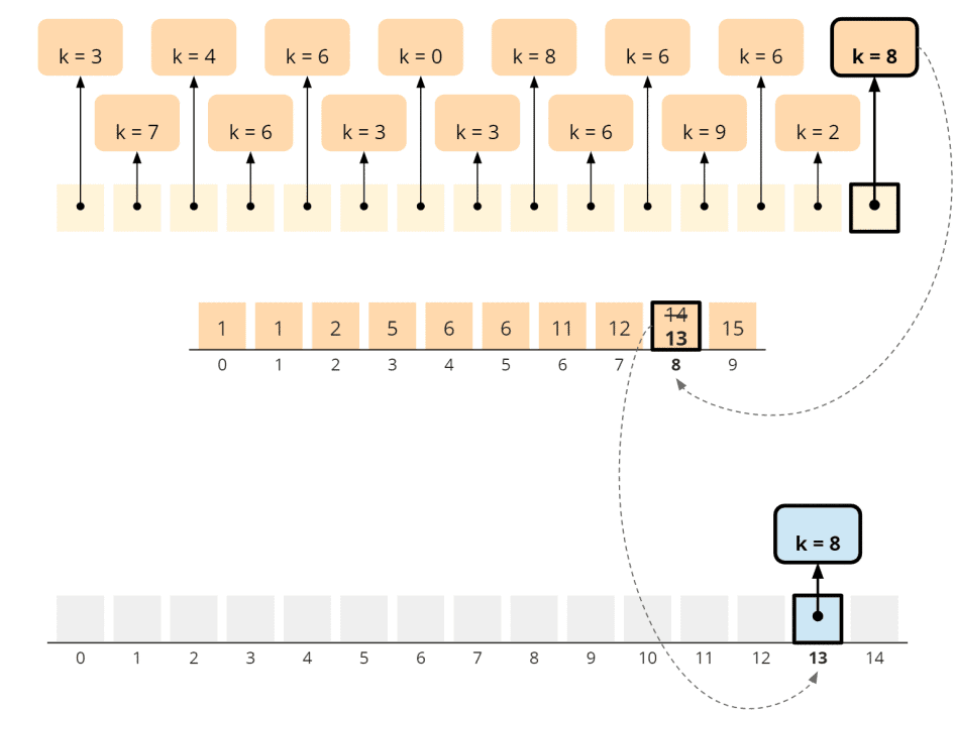
### General Algorithm – Phase 3: Writing Back Sorted Objects

To sort the objects, we need an additional array in the size of the input array:

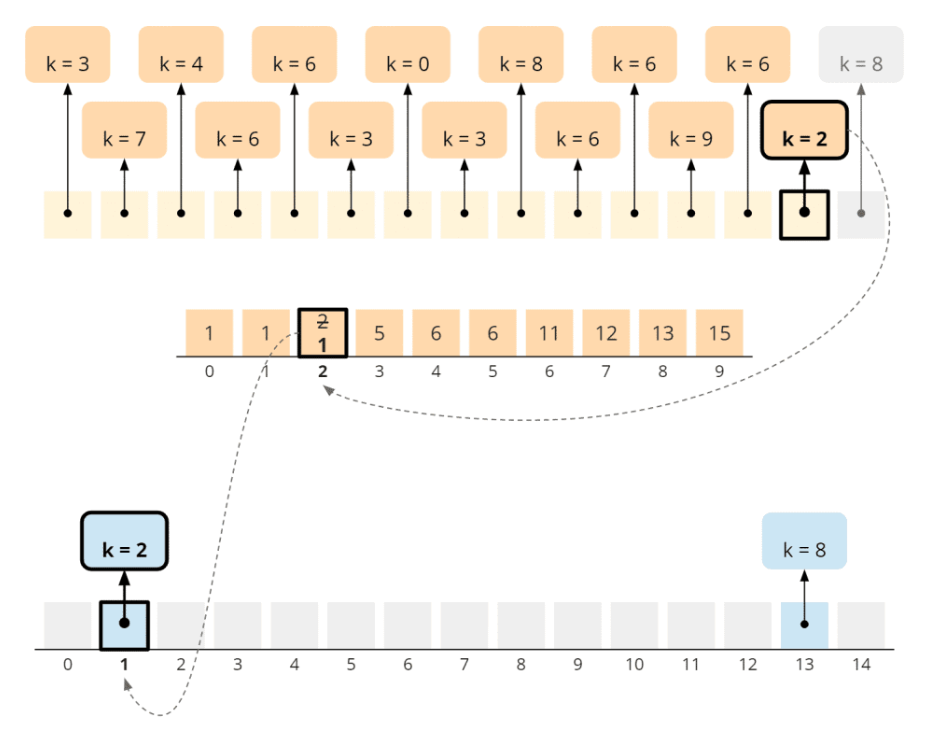


We now iterate backward over the array to be sorted and write each object into the target array to the position indicated by the auxiliary array. We decrement the corresponding value in the auxiliary array by 1 to put the next object with the same key one field further to the left.

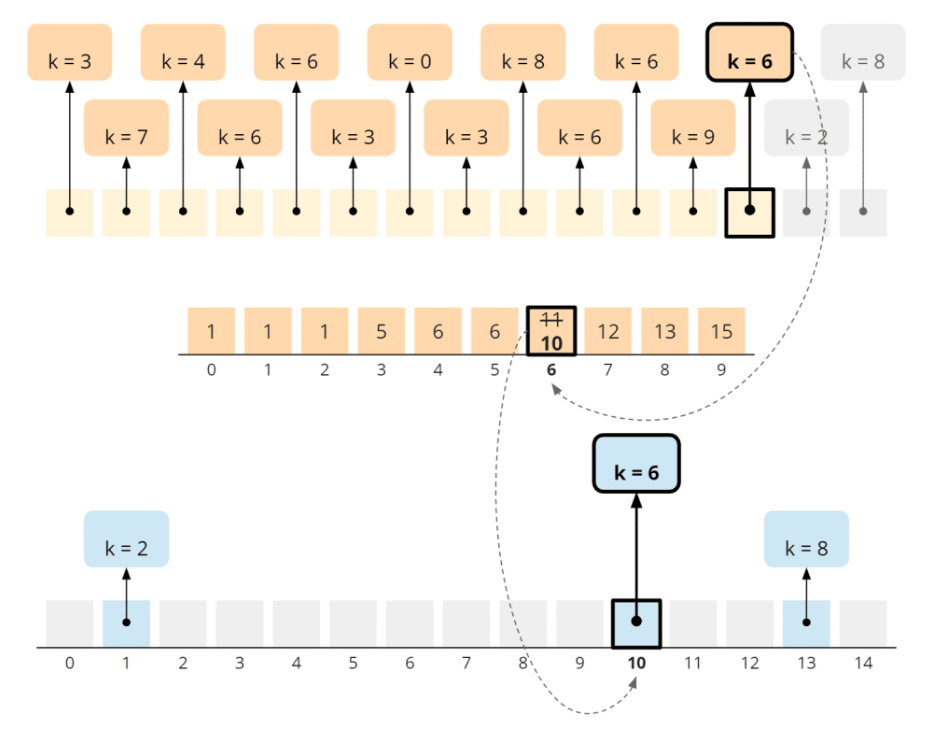
Let's start at the far right in the input array – with the object with key 8. In the auxiliary array, position 8 has the value 14. We decrement the value to 13 and copy the object with key 8 to the target array at position 13 (remember: the position information in the auxiliary array is 1-based, so we write at position 13, not 14).



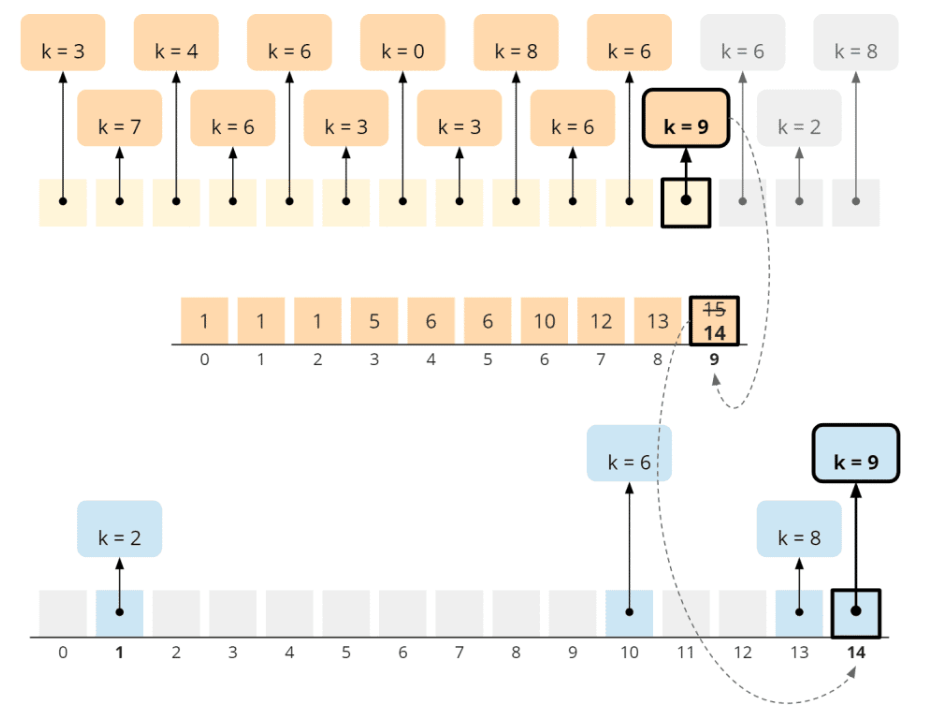
The second object from the right has the key 2. In the auxiliary array, position 2 has the value 2. We decrement the value in the auxiliary array to 1 and copy the object to the target array's corresponding position:



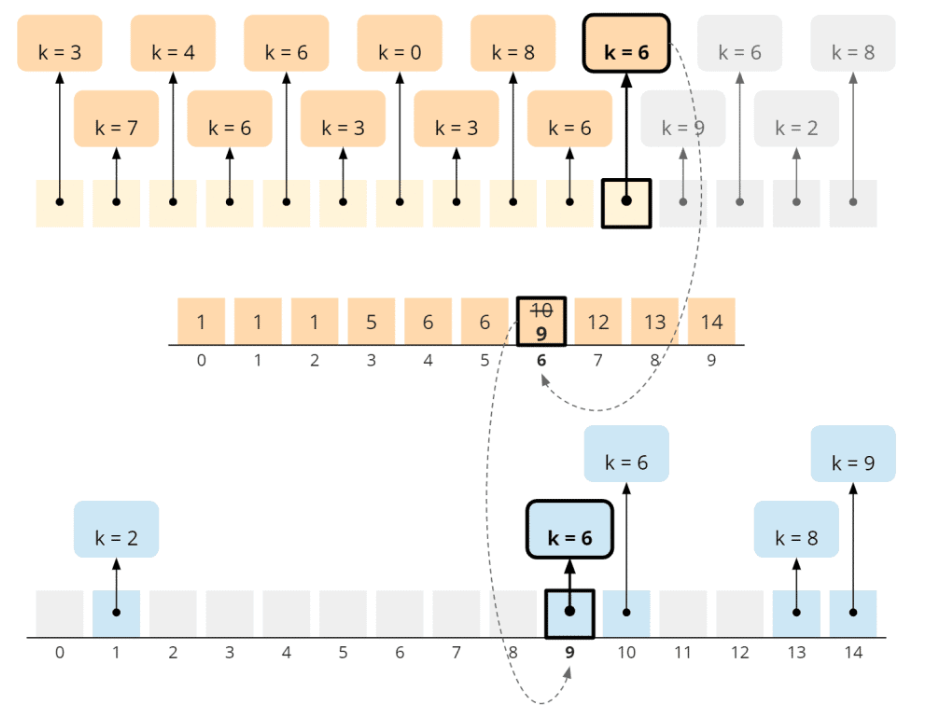
The next object has the key 6. In the auxiliary array, position 6 contains 11. We decrement the value to 10 and copy the object to field 10 in the target array:



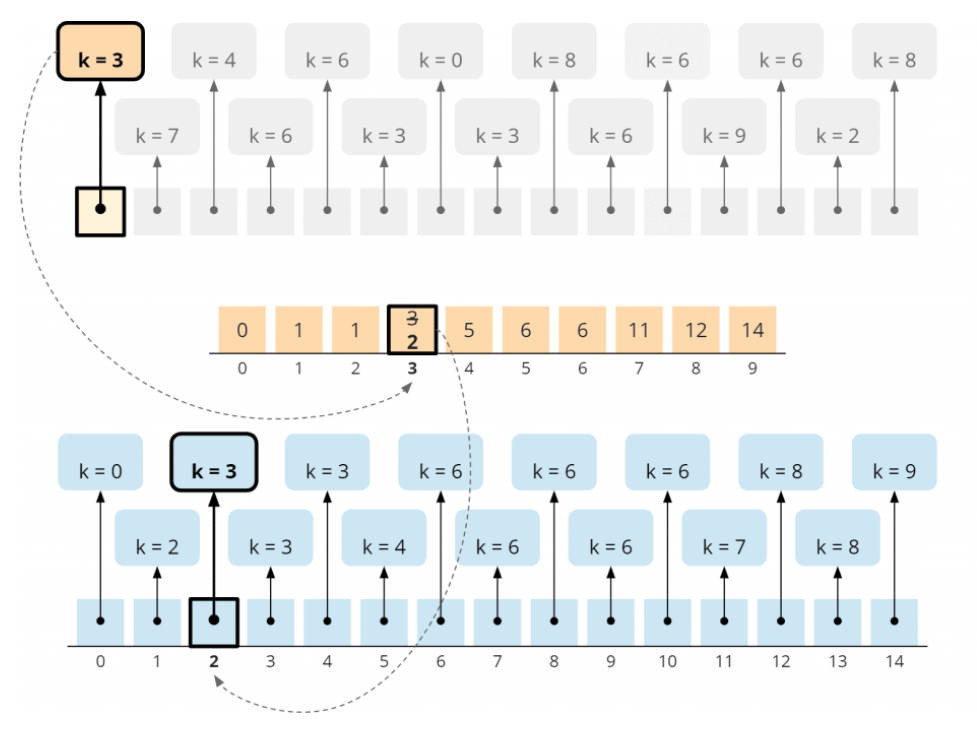
Following the same logic, we copy the object with the key 9 to position 14 in the target array:



An additional six follows. In the auxiliary array, position 6 now contains the 10 (after we had decremented the 11). We decrement the value again to 9 and copy the object to position 9 in the target array, i.e., to the left of the other object with key 6:



We repeat these steps for all elements and finally reach the object with the key 3. Field 3 in the auxiliary array now contains a 3. We decrement this to 2 and copy the object to position 2, the target array's last free position:



The objects are sorted; the algorithm is finished.

## Counting Sort Java Code Example (General Form)

The following code demonstrates the general form of Counting Sort for simplicity's sake using int primitives. The findMax() method is equal to the one in the first source code example, so I omitted it here.

public class CountingSortGeneral {

  public void sort(int[] elements) {

    int maxValue = findMax(elements);

    int[] counts = new int[maxValue + 1];

    // Phase 1: Count

    for (int element : elements) {

      counts[element]++;

    }

    // Phase 2: Aggregate

    for (int i = 1; i <= maxValue; i++) {

      counts[i] += counts[i - 1];

    }

    // Phase 3: Write to target array

    int[] target = new int[elements.length];

    for (int i = elements.length - 1; i >= 0; i--) {

      int element = elements[i];

      target[--counts[element]] = element;

    }

    // Copy target back to input array

    System.arraycopy(target, 0, elements, 0, elements.length);

  }

  [...]

}

You can find the source code in the [CountingSortGeneral](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/countingsort/CountingSortGeneral.java" \t "https://www.happycoders.eu/algorithms/counting-sort/_blank) class in the GitHub repository..

## Counting Sort Time Complexity

The [time complexity](https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/) of Counting Sort is easy to determine due to the very simple algorithm.

Let n be the number of elements to sort and k the size of the number range.

The algorithm contains one or more loops that iterate to n and one loop that iterates to k.

Constant factors are irrelevant for the time complexity; therefore:

The time complexity of Counting Sort is: O(n + k)

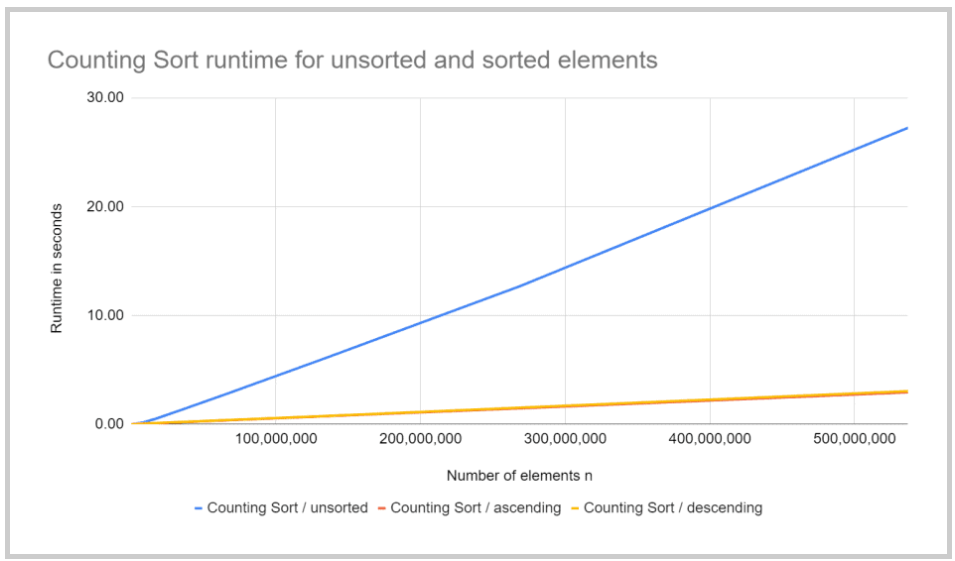
### Runtime of the Java Counting Sort Example

The GitHub repository contains the [UltimateTest](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/demos/UltimateTest.java" \t "https://www.happycoders.eu/algorithms/counting-sort/_blank) program, which allows us to measure the speed of Counting Sort (and all the other sorting algorithms in this article series).

The following table shows the time needed to sort unsorted and ascending and descending presorted elements for the given number of elements n, which in these measurements also corresponds to the size of the number range k:

| **n, k** | **random** | **ascending** | **descending** |
| --- | --- | --- | --- |
| ... | ... | ... | ... |
| 33,554,432 | 1,276 ms | 195 ms | 210 ms |
| 67,108,864 | 2,857 ms | 381 ms | 388 ms |
| 134,217,728 | 6,087 ms | 745 ms | 766 ms |
| 268,435,456 | 12,684 ms | 1,477 ms | 1,529 ms |
| 536,870,912 | 27,249 ms | 2,945 ms | 3,039 ms |

You can find the complete result in the file [Test\_Results\_Counting\_Sort.log](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/results/2020-08-30/Test_Results_Counting_Sort.log" \t "https://www.happycoders.eu/algorithms/counting-sort/_blank). The following diagram shows the measurements graphically:



You can see the following:

* Pre-sorted output sequences with half a billion elements are sorted about nine times faster than unsorted ones.
* For presorted input sequences, the measurements correspond to the expected linear time complexity O(n + k).
* For unsorted input sequences, the measurements are slightly higher: When the array size doubles, the time required increases by a factor of about 2.1 to 2.2.
* Input sequences sorted in descending order are sorted minimally slower than those pre-sorted in ascending order.

If elements are not actually sorted but counted and entirely rearranged, shouldn't the initial order do not affect the time needed for sorting!?

Using the program [CountOperations](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/demos/CountOperations.java" \t "https://www.happycoders.eu/algorithms/counting-sort/_blank), we can measure how many operations are needed for sorting. And indeed, the result confirms the assumption (see file [CountOperations\_Counting\_Sort.log](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/results/2020-08-30/CountOperations_Counting_Sort.log" \t "https://www.happycoders.eu/algorithms/counting-sort/_blank)):

* The number of operations is independent of the initial order of the elements.
* The number of operations corresponds to the expected time complexity O(n + k), thus increasing linearly with the number of elements to sort and the size of the number range.

Then what causes these deviating measurements? You will find explanations in the following sections.

### Why Is Counting Sort Faster for Presorted Elements Than for Unsorted Ones?

An auxiliary array with half a billion elements is 2 GB in size. If its elements are incremented in random order, a new cache line (typically 64 bytes) must be exchanged between RAM and CPU cache for almost every element. The larger the array, the lower the probability that the required cache line is in the CPU cache.

In contrast, if the array is incremented from front to back (or from back to front), 16 consecutive int values can be loaded from and written to the RAM in a single 64-byte block.

This does not quite achieve an acceleration by factor 16, but at least one by factor nine.

### Why Doesn't Counting Sort Achieve Linear Time Complexity for Unsorted Output Sequences in Practice?

The larger the array to be sorted, the higher the ratio of cache misses to cache hits when accessing the auxiliary array (because the size of the CPU cache remains the same).

So with an array twice as big, we don't have twice as many cache misses, but a little more than twice as many. Accordingly, the time required increases by a little more than a factor of two.

### Why Is Counting Sort Faster for Items Sorted in Ascending Order Than for Items Sorted in Descending Order?

If elements are sorted in ascending order, they are not changed and do not have to be written back to RAM. With elements sorted in descending order, every element of the array changes, so the whole array has to be written back into RAM once.

## Further Characteristics of Counting Sort

In this chapter, we determine the space complexity, stability, and parallelizability of Counting Sort.

### Space Complexity of Counting Sort

The simplified algorithm requires an additional array of size k; therefore:

The space complexity of the simplified counting sort algorithm is: O(k)

In addition to the auxiliary array of size k, the general algorithm requires a temporary target array of size n; thus:

The space complexity of the general counting sort algorithm is: O(n + k)

### Stability of Counting Sort

In Phase 3, the general form of the Counting Sort algorithm iterates from right to left over the input array, copying objects with the same key also from right to left into the output array. Thus:

Counting Sort is a stable sorting algorithm.

### Parallelizability of Counting Sort

Counting Sort can be parallelized by dividing the input array into as many partitions as there are processors available.

In phase 1, each processor counts the elements of "its" partition in a separate auxiliary array.

In phase 2, all auxiliary arrays are added up to one.

In phase 3, each processor copies the elements of "its" partition to the target array. The decrementing and reading of the fields in the auxiliary array must be done atomically.

Due to parallelization, it can no longer be guaranteed that elements with the same key are copied to the target array in their original order.

Parallel Counting Sort is therefore not stable.

## Summary

Counting Sort is a very efficient, stable sorting algorithm with a time and space complexity of O(n + k).

Counting Sort is mainly used for small number ranges. In the JDK, for example, for:

* byte arrays with more than 64 elements (for fewer elements, Insertion Sort is used)
* short or char arrays with more than 1,750 Elementen (for fewer elements, Insertion Sort or Dual-Pivot Quicksort is used)

If you liked the article, feel free to share it using one of the share buttons at the end. Would you like to be informed by e-mail when I publish a new article? Then [click here](https://www.happycoders.eu/algorithms/counting-sort/) to subscribe to the HappyCoders newsletter.