# Algorithmen und Datenstrukturen

Prof. Dr. Thomas Wiemann - FB AI

Hochschule Fulda
University of Applied Sciences

# Inhalt

## Gliederung

1. Laufzeit und Komplexität
2. **Sortieren**
3. Abstrakte Datentypen
4. Hashing
5. Suchbäume
6. Graphen- und Graphenalgorithmen
7. Ausblick

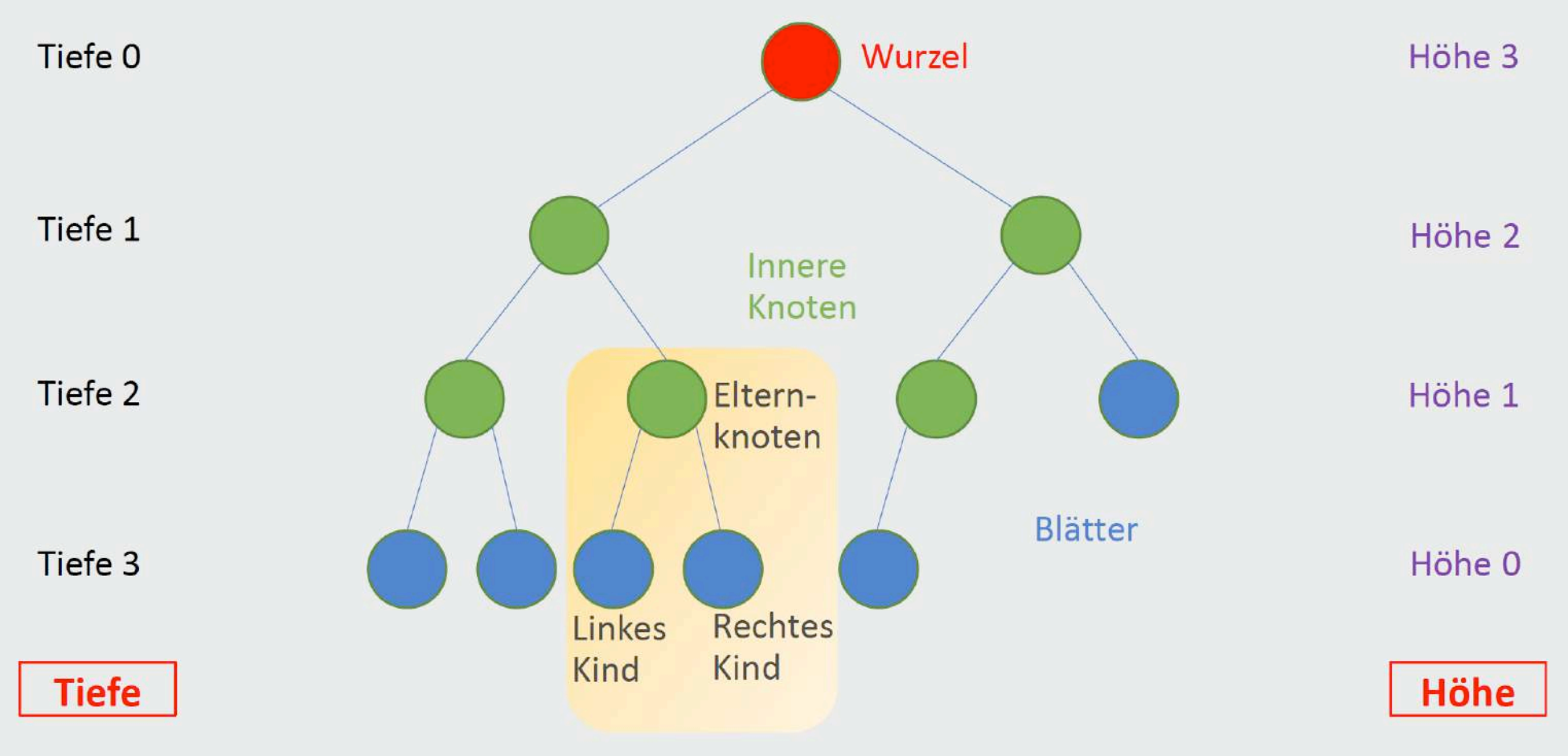
## 2. Sortieren

1. Simple Sortierverfahren
2. Komplexe Sortierverfahren
3. **Untere Schranke für vergleichsbasiertes Sortieren**
4. Sortieren in linearer Zeit

# Erinnerung: Das Sortierproblem

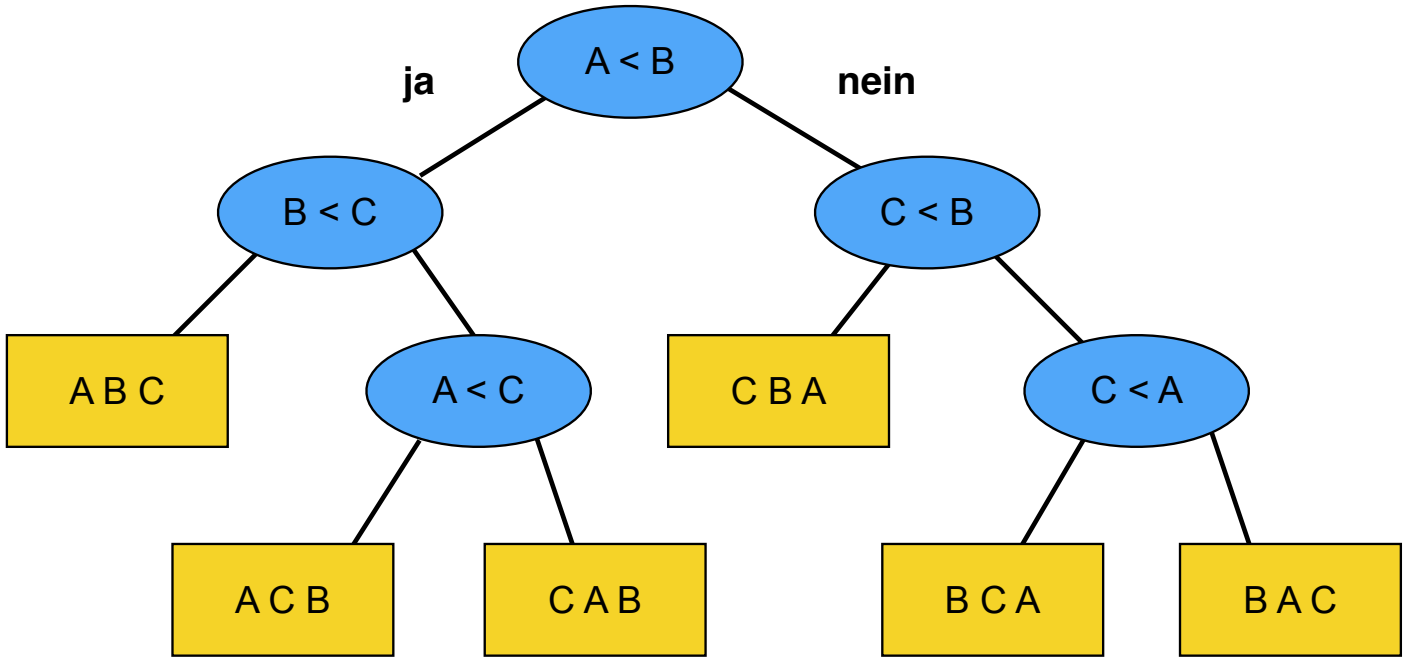
Sei $A$ ein Alphabet oder eine geordnete Menge von Elementen. Eine Folge $v = v_1, \ldots, v_n$ heißt **geordnet**, falls $v_i \leq v_{i+1}$ für $i = 1 \ldots n$. **Sortierproblem**: Gegeben seine eine Folge $v = v_1, \ldots, v_n$ in $A$. Ordne $v$ so um, dass eine geordnete Folge entsteht. Gesucht wird also eine **Permutation** $\pi : \{1,\ldots,n\} \rightarrow \{1,\ldots,n\}$ so dass die Folge $w = v_{\pi 1} \ldots v_{\pi n}$ geordnet ist

# Bäume - Definitionen



# Entscheidungsbäume (1)

- Entscheidungsbaum zur Sortierung von drei Elementen A,B,C




- Der Entscheidungsbaum zur Sortierung von $n$ Elementen hat mindestens $n!$ Blätter

# Entscheidungsbäume (2)

- Abstraktion eines Sortierverfahrens durch einen Binärbaum
- Darstellung der Vergleiche von irgendeinem Suchverfahren und Eingaben vorgegebener Größe
- Läßt alles andere (Kontrollstrukturen, Datenverschiebungen) außer acht
- Es werden nur vergleiche betrachtet
- Innere Knoten repräsentieren die Elemente im Array, die verglichen werden
- Blätter enthalten die Permutationen, die der Algorithmus bestimmt
- Die Operationen jedes Algorithmus können als Verfolgung eines bestimmten Weges durch den Entscheidungsbaum betrachtet werden
- Ein korrekter Suchalgorithmus muss alle Permutationen erzeugen können
- Jede der $n!$ Permutationen muss in einem Blatt vorkommen

# Untere Schranke für vergleichsbasiertes Sortieren

**Satz:** Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche im Worst-Case

- Beweis:
  - Es reicht die Höhe des Entscheidungsbaums zu bestimmen
  - $h =$ Höhe, $l =$ Anzahl der Blätter im Entscheidungsbaum
  - Im Entscheidungsbaum für $n$ Elemente gilt: $l \geq n!$
  - Im Binärbaum der Höhe $h$ gilt: $l \leq 2^h$
  - Entsprechend folgt: $n! \leq l \leq 2^h \Rightarrow n! \leq 2^h \Rightarrow h \geq \log(n!)$
  - Weiterhin gilt $n! \geq \left(\frac{n}{e}\right)^n$

# Untere Schranke für vergleichsbasiertes Sortieren

**Satz:** Jeder vergleichsbasierte Sortieralgorithmus benötigt $\Omega(n \log n)$ Vergleiche im Worst-Case

- Beweis (Fortsetzung):
  - Es gilt $n! \leq l \leq 2^h \Rightarrow n! \leq 2^h \Rightarrow h \geq \log(n!)$
  - Weiterhin gilt $n! \geq \left(\frac{n}{e}\right)^n$
  - Somit: $h \geq \log(n!) \geq n \log \left(\frac{n}{e}\right) = n \log n - n \log e$
  - Insgesamt also $h \in \Omega(n \log n)$

**Average Case ist ebenfalls in $\Omega(n \log n)$.**

# Inhalt

## Gliederung

1. Laufzeit und Komplexität
2. **Sortieren**
3. Abstrakte Datentypen
4. Hashing
5. Suchbäume
6. Graphen- und Graphenalgorithmen
7. Ausblick

## 2. Sortieren

1. Simple Sortierverfahren
2. Komplexe Sortierverfahren
3. Untere Schranke für vergleichsbasiertes Sortieren
4. **Sortieren in linearer Zeit**

# Lineare Sortierverfahren

- Bisher: Sortierverfahren basierend auf den Operationen *vergleichen* und *vertauschen*
- Im besten Fall kann damit eine Laufzeit von $\mathcal{O}(n \log n)$ erreicht werden
- Weitere Klasse von Sortieralgorithmen: Distribution Sort
- Neben Vergleichen werden weitere Operationen, z.B. Zählen, arithmetische Operationen usw.. angewendet
- Allgemeines Schema:
  - Verteile die Daten auf Fächer (distribute)
  - Einsammeln der Daten aus den Fächern (wobei die Ordnung der Fächer eingehalten werden muss) (gather)
- Erster Vertreter: Counting Sort

# Inhalt

## Gliederung

1. Laufzeit und Komplexität
2. **Sortieren**
3. Abstrakte Datentypen
4. Hashing
5. Suchbäume
6. Graphen- und Graphenalgorithmen
7. Ausblick

## 2. Sortieren

1. Simple Sortierverfahren
2. Komplexe Sortierverfahren
3. Untere Schranke für vergleichsbasiertes Sortieren
4. **Sortieren in linearer Zeit**

# Sortieren in linearer Zeit - Counting Sort

- Counting-Sort kann eine gegebene Folge von $n$ Elementen in linearer Zeit durch einfaches Zählen sortieren
- Voraussetzung: Es gibt ein beschränktes Intervall $k$ mit möglichen Werten oder eine Abbildung von $n$ auf $k$
- Position des Elements in der sortierten Menge wird durch ein Histogramm erstellt
- Gutes Beispiel: Sortierung aller Bundesbürger nach ihrem Alter (Abbildung 80 Mio. Menschen auf 0..130)
- Schlechtes Beispiel: Alphabetische Sortierung aller Namen der Länge $l$ ($k \approx 26^l$)

# Counting Sort - Ablauf

| 3 | 7 | 2 | 6 | 1 | 0 | 4 | 8 | 8 | 9 | 0 | 3 | 4 | 6 | 7 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Zähler | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

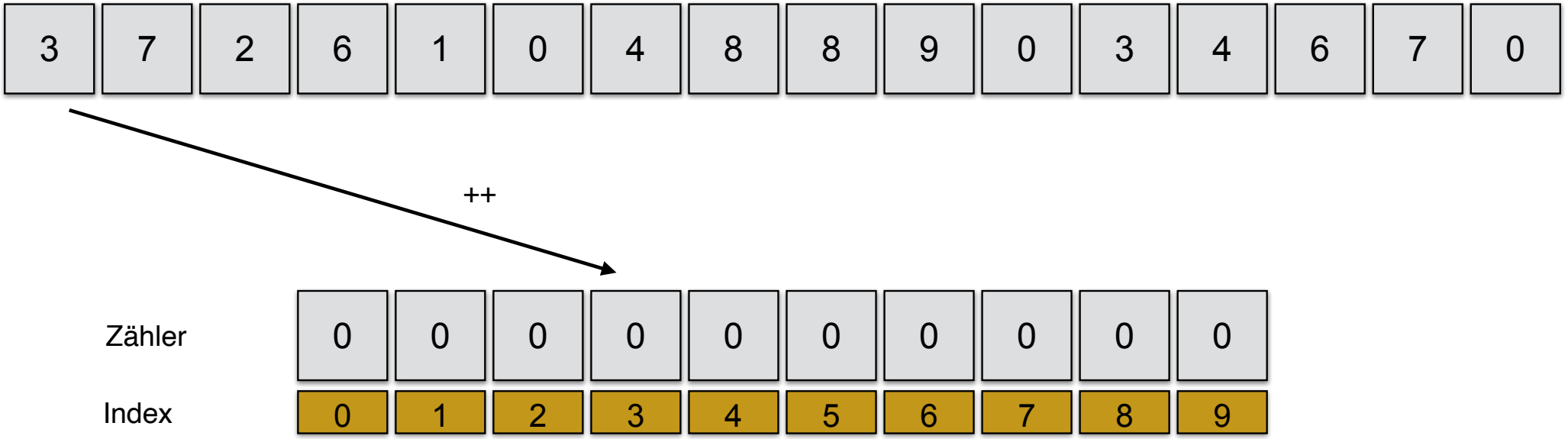
# Counting Sort - Ablauf

| 3 | 7 | 2 | 6 | 1 | 0 | 4 | 8 | 8 | 9 | 0 | 3 | 4 | 6 | 7 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Zähler | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

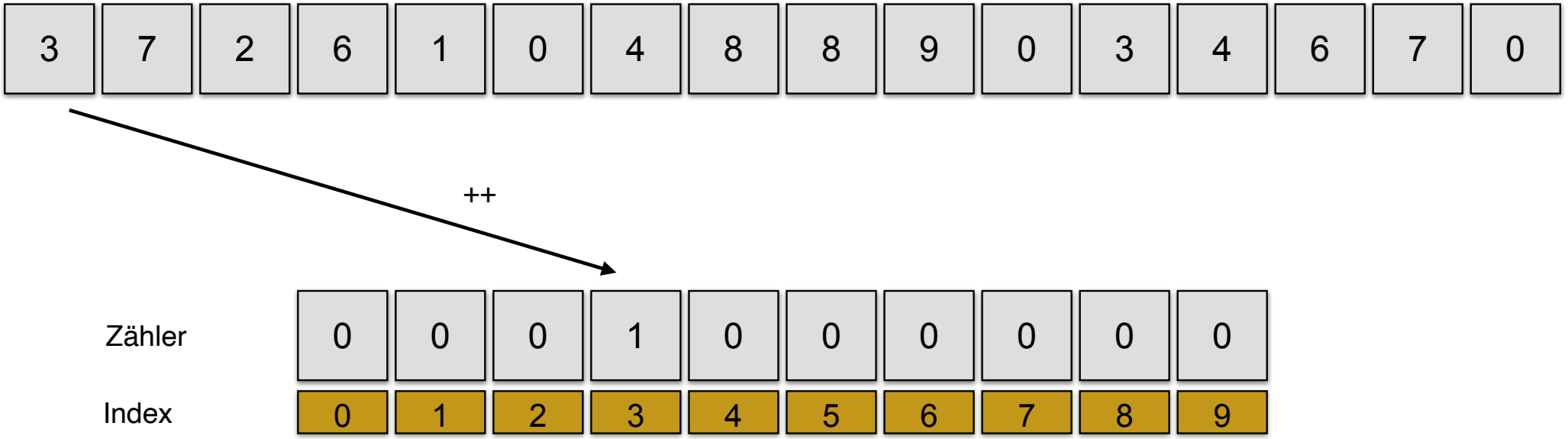
- Schritt 1: Iteriere durch Eingabearray und inkrementiere die Zähler in den entsprechenden Buckets

# Counting Sort - Ablauf



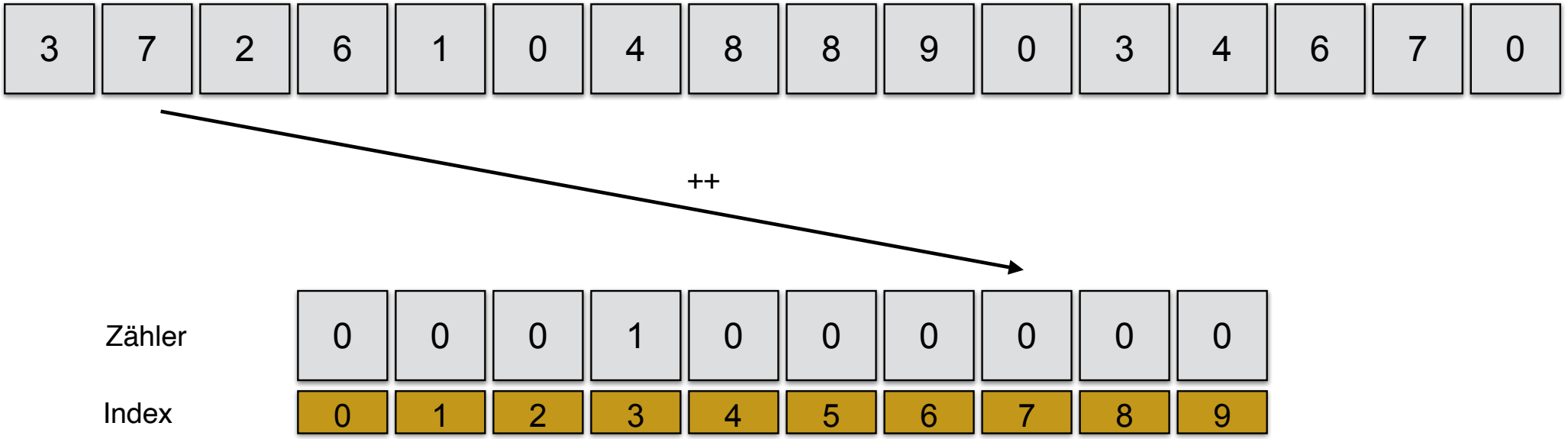

- Schritt 1: Iteriere durch Eingabearray und inkrementiere die Zähler in den entsprechenden Buckets

# Counting Sort - Ablauf



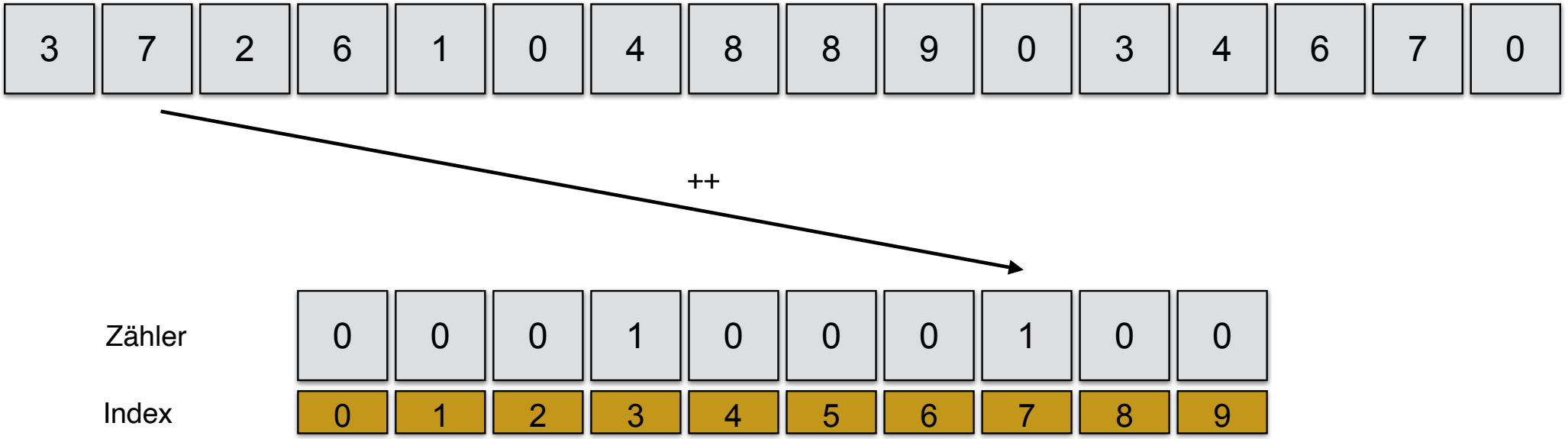

- Schritt 1: Iteriere durch Eingabearray und inkrementiere die Zähler in den entsprechenden Buckets

# Counting Sort - Ablauf




- Schritt 1: Iteriere durch Eingabearray und inkrementiere die Zähler in den entsprechenden Buckets

# Counting Sort - Ablauf




- Schritt 1: Iteriere durch Eingabearray und inkrementiere die Zähler in den entsprechenden Buckets

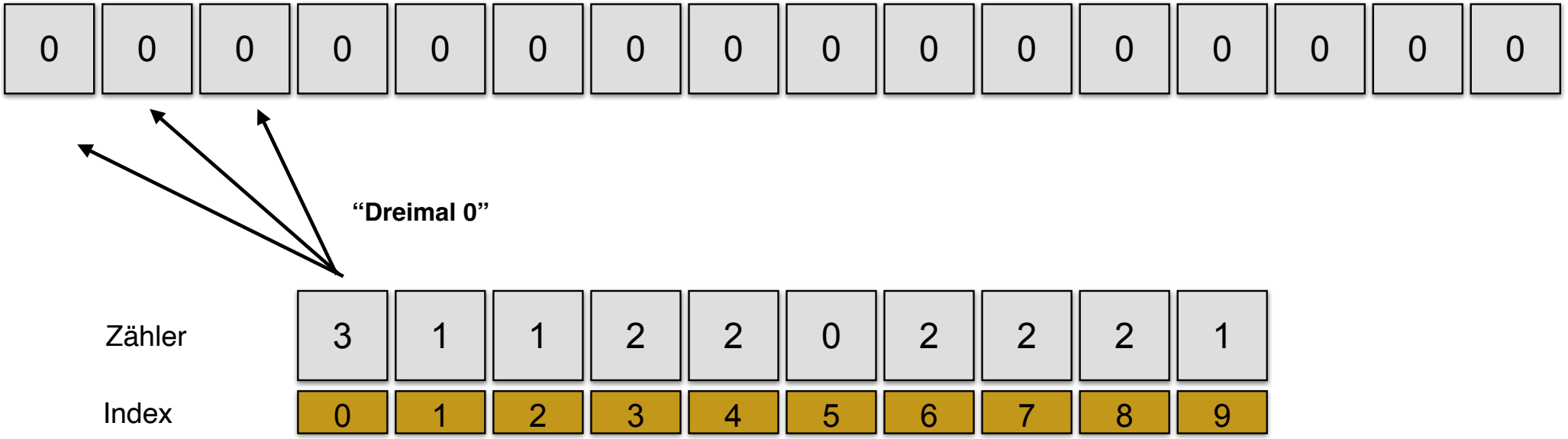
# Counting Sort - Ablauf

| 3 | 7 | 2 | 6 | 1 | 0 | 4 | 8 | 8 | 9 | 0 | 3 | 4 | 6 | 7 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Zähler | 3 | 1 | 1 | 2 | 2 | 0 | 2 | 2 | 2 | 1 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

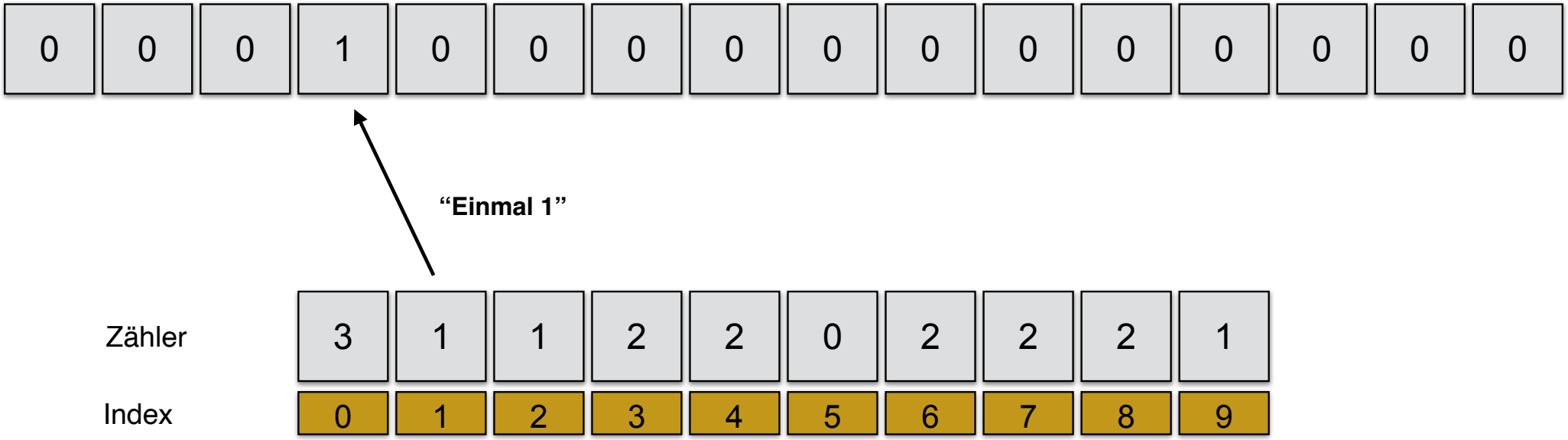
- Schritt 1 Ende: Histogramm der Vorkommen der Ziffern

# Counting Sort - Ablauf



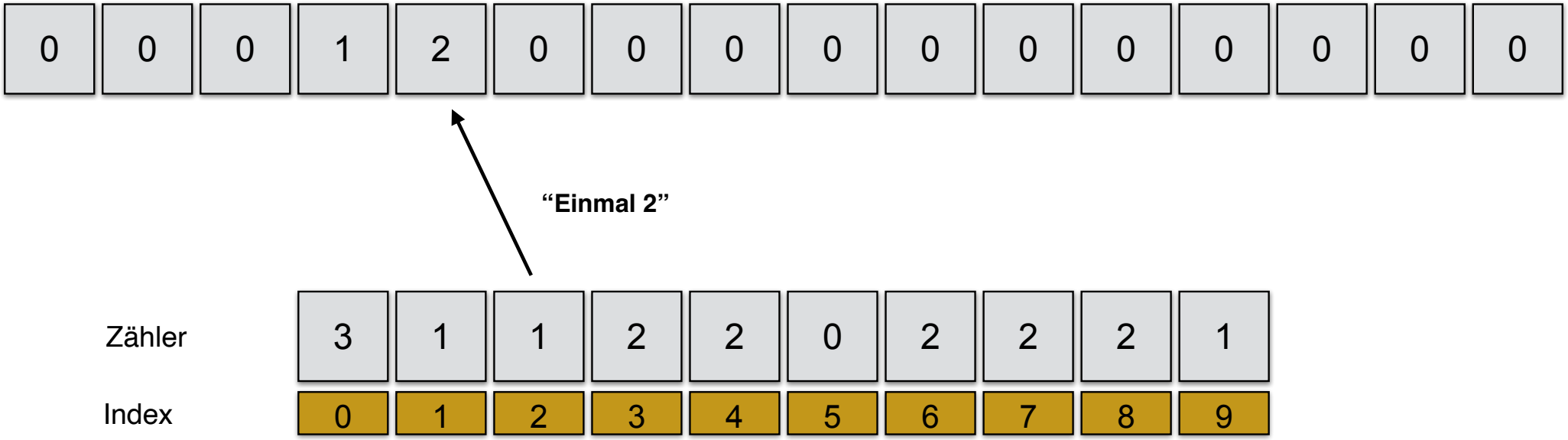

- Schritt 2: Durchlaufe die Buckets und verteile die Inhalte

# Counting Sort - Ablauf



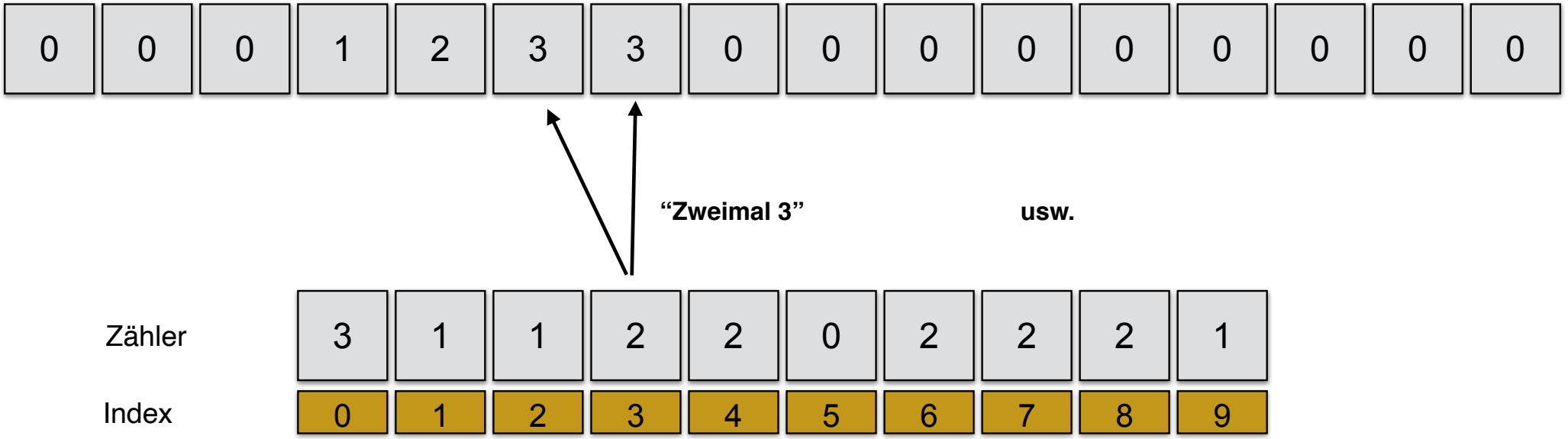

- Schritt 2: Durchlaufe die Buckets und verteile die Inhalte

# Counting Sort - Ablauf




- Schritt 2: Durchlaufe die Buckets und verteile die Inhalte

# Counting Sort - Ablauf




- Schritt 2: Durchlaufe die Buckets und verteile die Inhalte

# Counting Sort - Ablauf

| 0 | 0 | 0 | 1 | 2 | 3 | 3 | 4 | 4 | 6 | 6 | 7 | 7 | 8 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Zähler | 3 | 1 | 1 | 2 | 2 | 0 | 2 | 2 | 2 | 1 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- Schritt 2: Ende

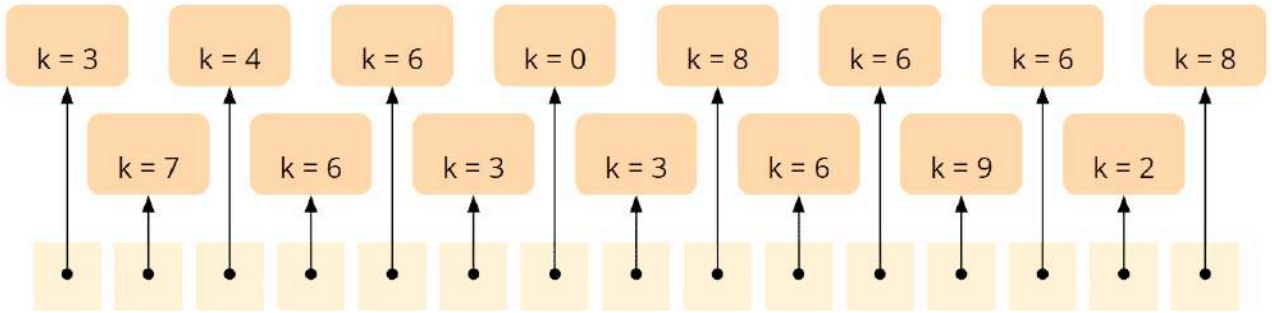
# Einfacher Counting Sort in Java

```
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 back results
  int targetPos = 0;
  for (int i = 0; i < counts.length; i++) {
    for (int j = 0; j < counts[i]; j++) {
      elements[targetPos++] = i;
    }
  }
}
```

Counting Sort für positive Zahlen!

# Algemeiner Counting Sort (1)

- Phase 1: Wie im einfachen Fall ➠ Histogramm erstellen

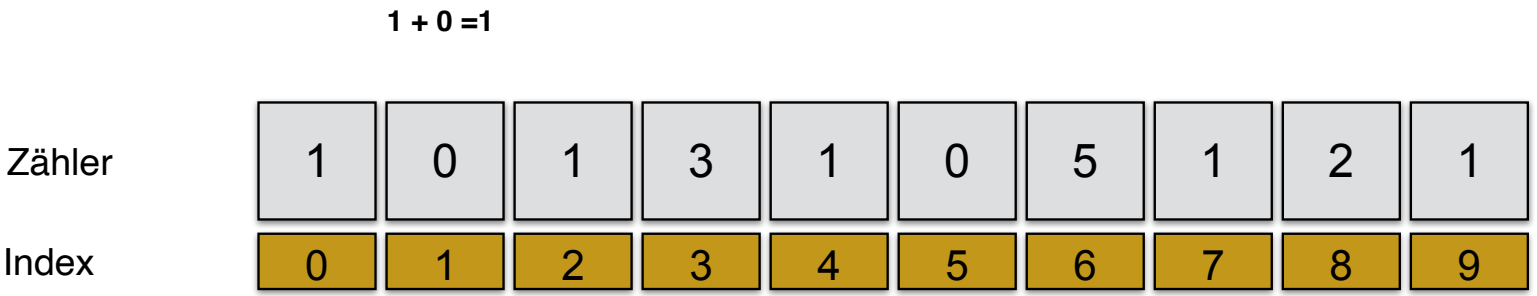



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Zähler | 1 | 0 | 1 | 3 | 1 | 0 | 5 | 1 | 2 | 1 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Quelle: https://www.happycoders.eu/de/algorithmen/counting-sort/

# Allgemeiner Counting Sort (2)

- Phase 2: Histogramm aggregieren
- Wird benötigt, um zu bestimmen, wo die Adressen im Zielarray sind
- Summe der Elemente links vom aktuellen Element



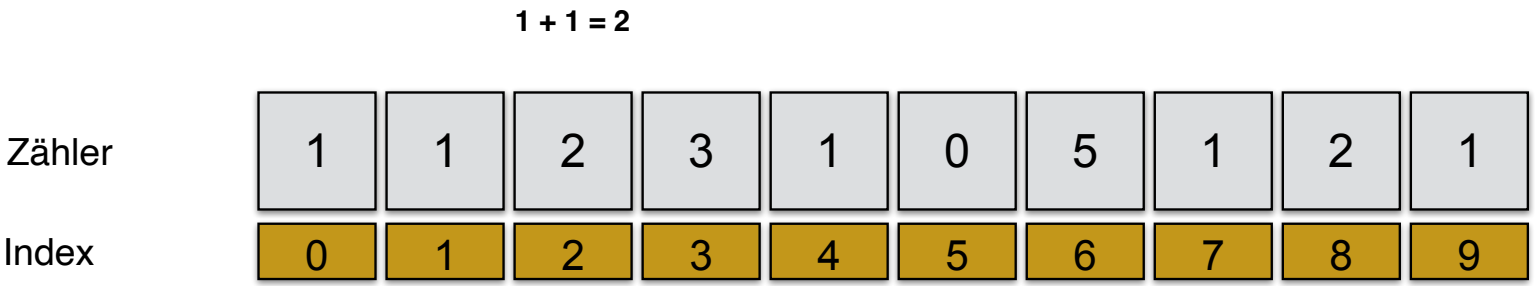
# Allgemeiner Counting Sort (2)

- Phase 2: Histogramm aggregieren
- Wird benötigt, um zu bestimmen, wo die Adressen im Zielarray sind
- Summe der Elemente links vom aktuellen Element

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Zähler | 1 | 1 | 1 | 3 | 1 | 0 | 5 | 1 | 2 | 1 |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

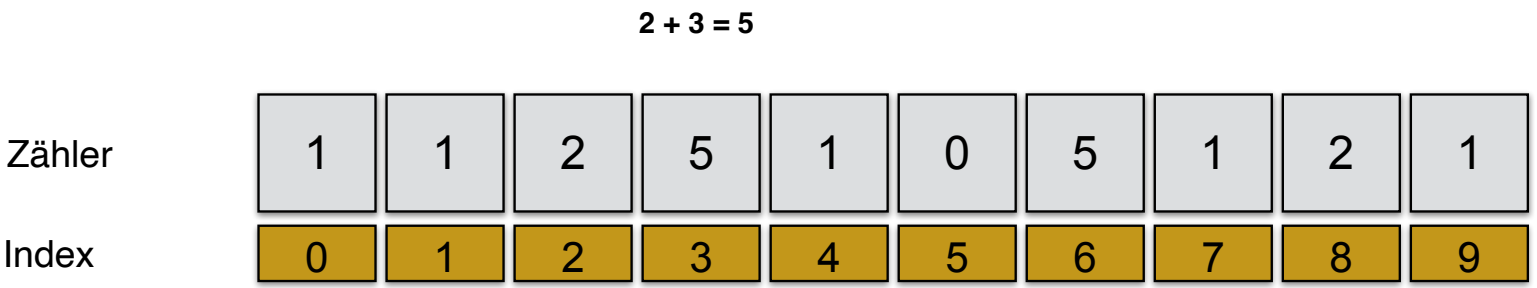
# Allgemeiner Counting Sort (2)

- Phase 2: Histogramm aggregieren
- Wird benötigt, um zu bestimmen, wo die Adressen im Zielarray sind
- Summe der Elemente links vom aktuellen Element



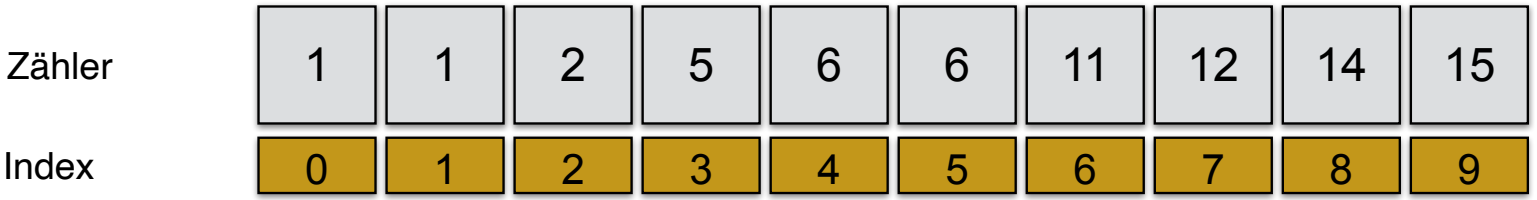
# Allgemeiner Counting Sort (2)

- Phase 2: Histogramm aggregieren
- Wird benötigt, um zu bestimmen, wo die Adressen im Zielarray sind
- Summe der Elemente links vom aktuellen Element

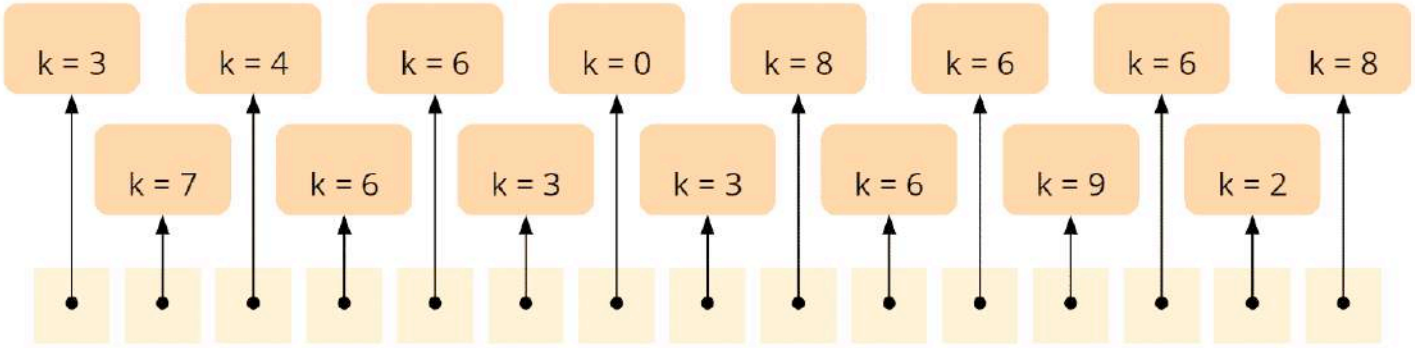


# Allgemeiner Counting Sort (2)

- Phase 2: Histogramm aggregieren
- Wird benötigt, um zu bestimmen, wo die Adressen im Zielarray sind
- Summe der Elemente links vom aktuellen Element



# Allgemeiner Counting Sort - Zurückschreiben




Hilfsarray:

| 1 | 1 | 2 | 5 | 6 | 6 | 11 | 12 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

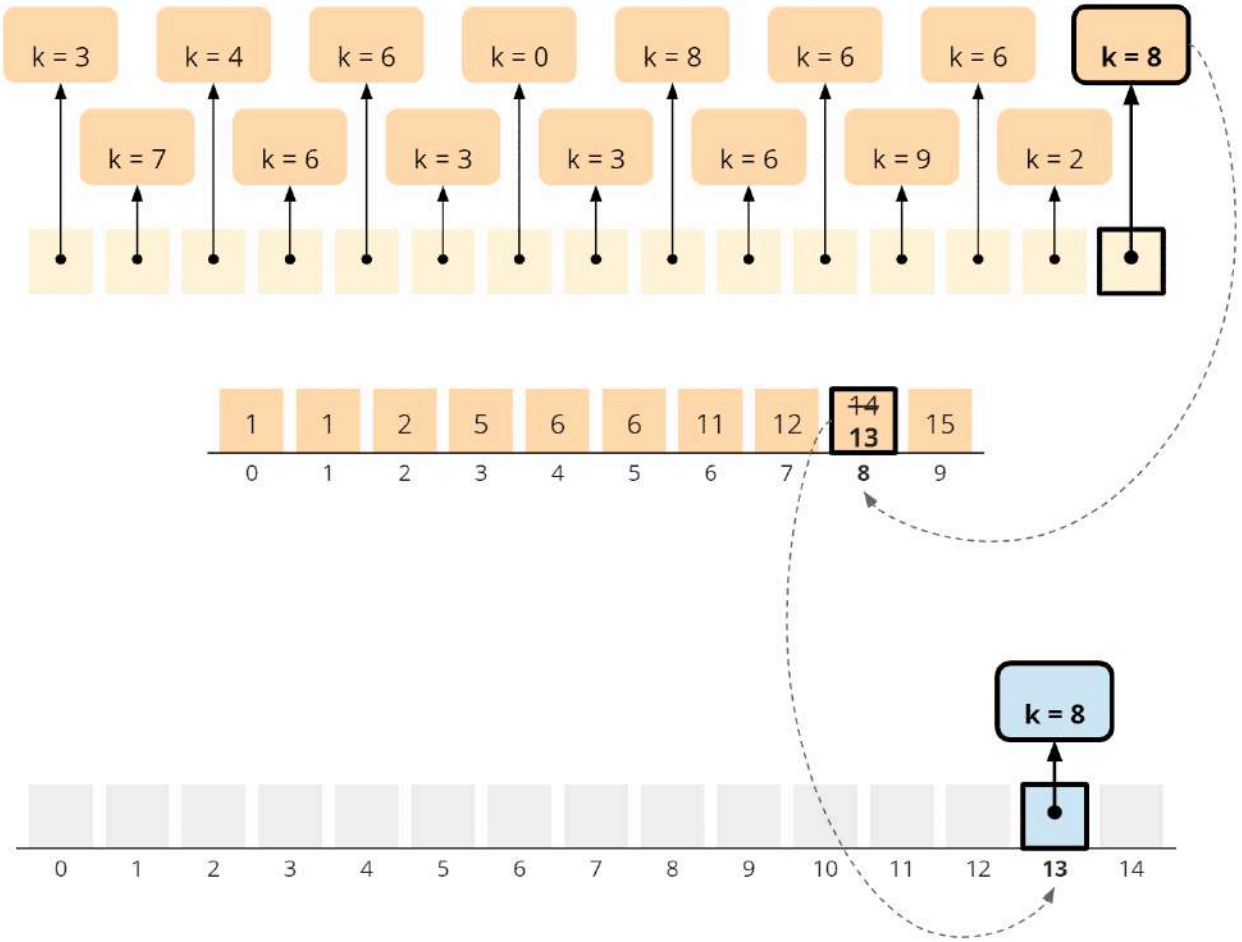
Zielarray:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

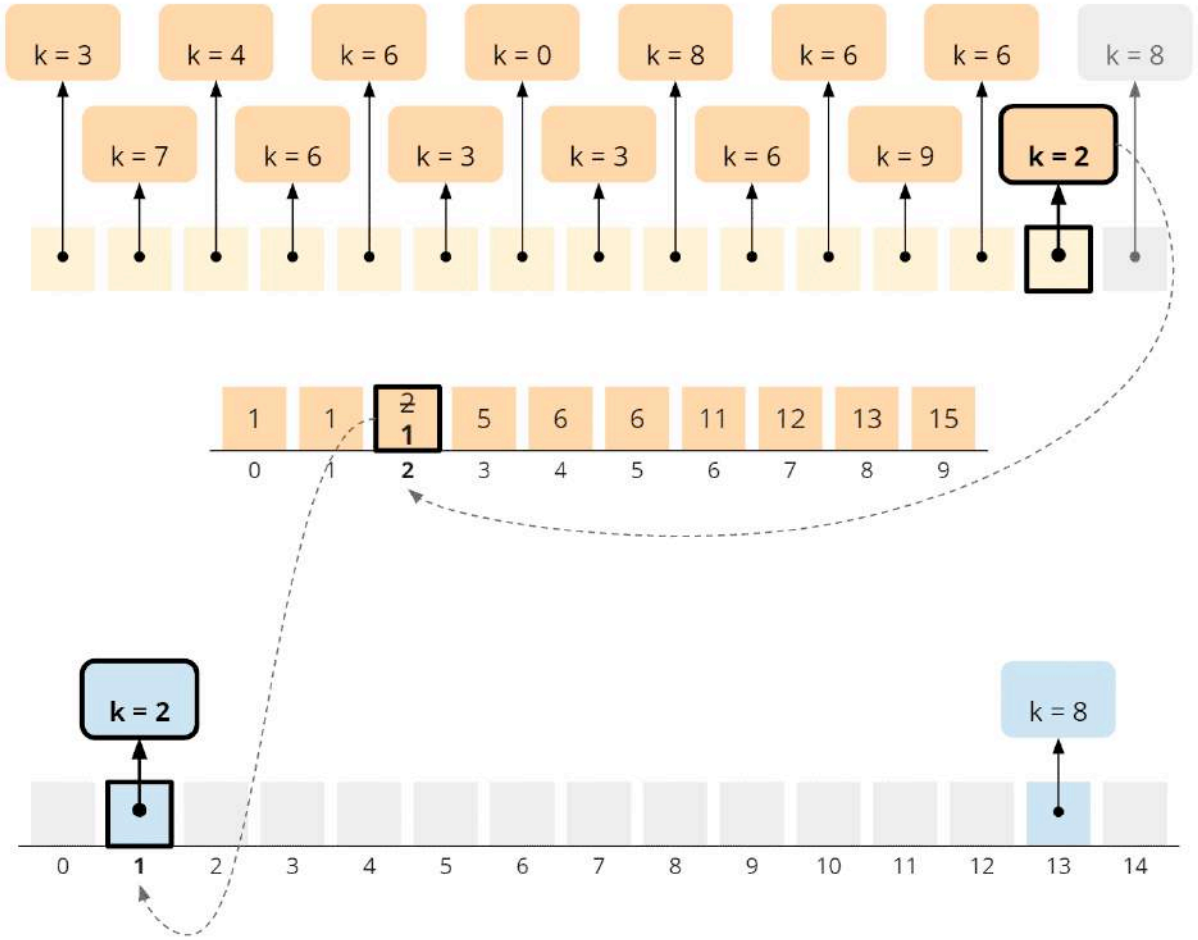
**Achtung bei der Indizierung!**
**Häufigkeit ≠ Array Index**

Quelle: https://www.happycoders.eu/de/algorithmen/counting-sort/

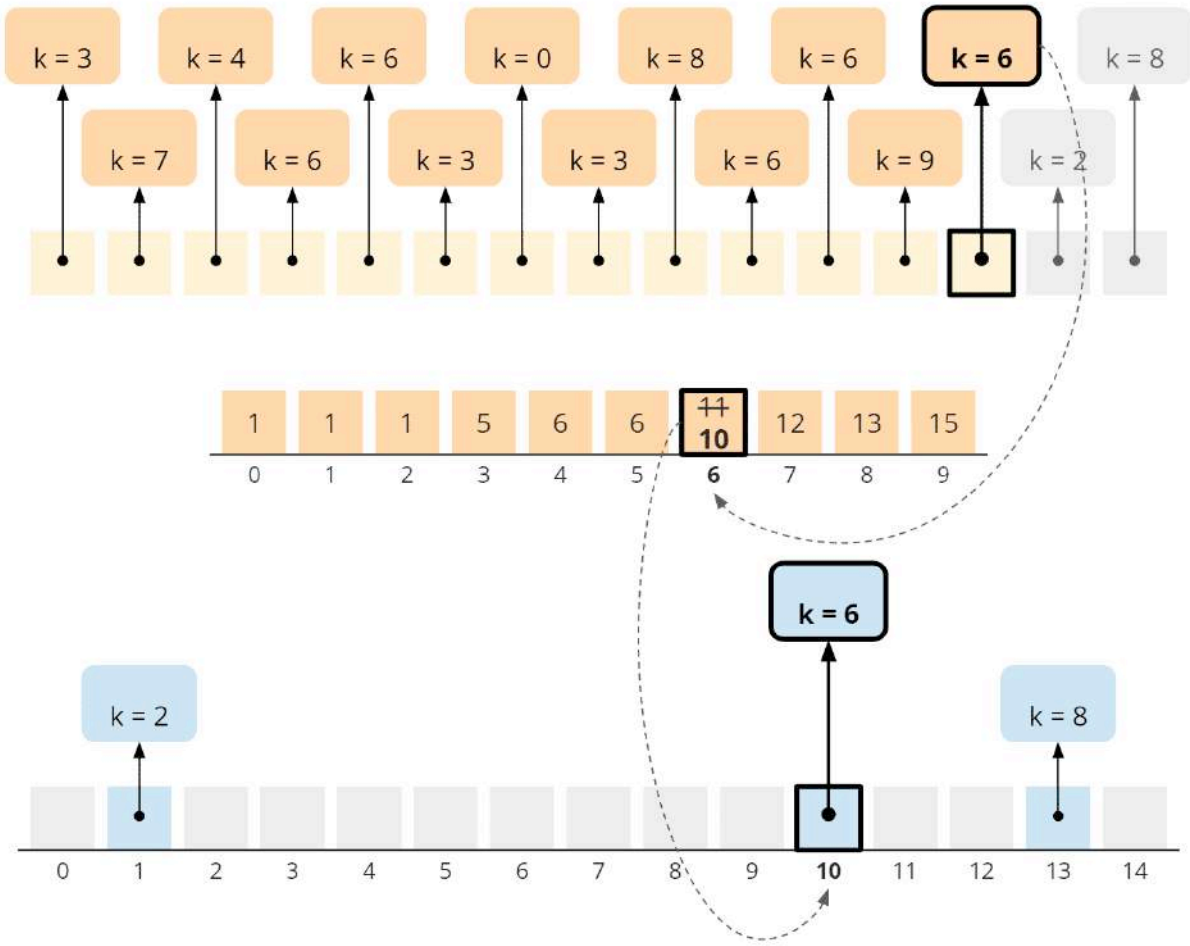
# Allgemeiner Counting Sort - Zurückschreiben



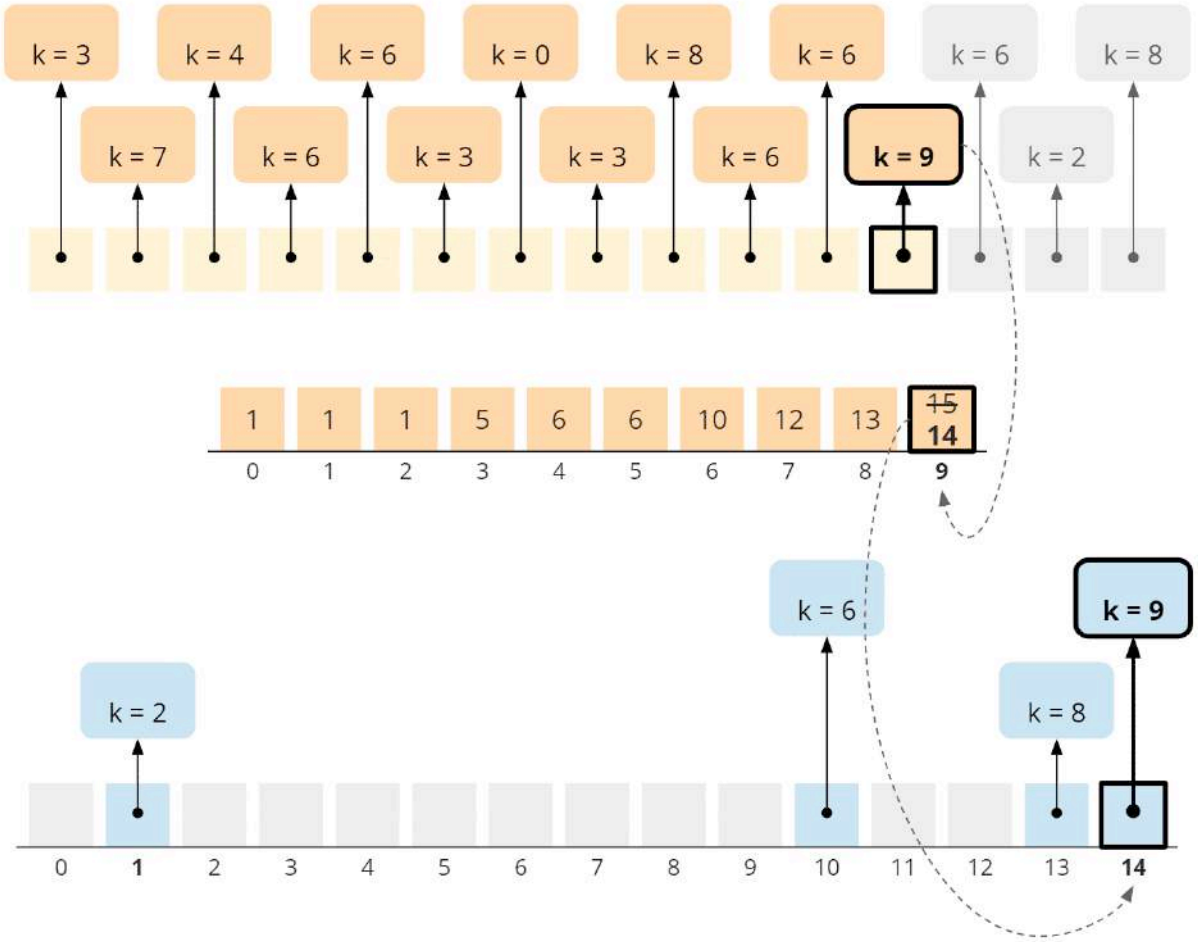
# Allgemeiner Counting Sort - Zurückschreiben



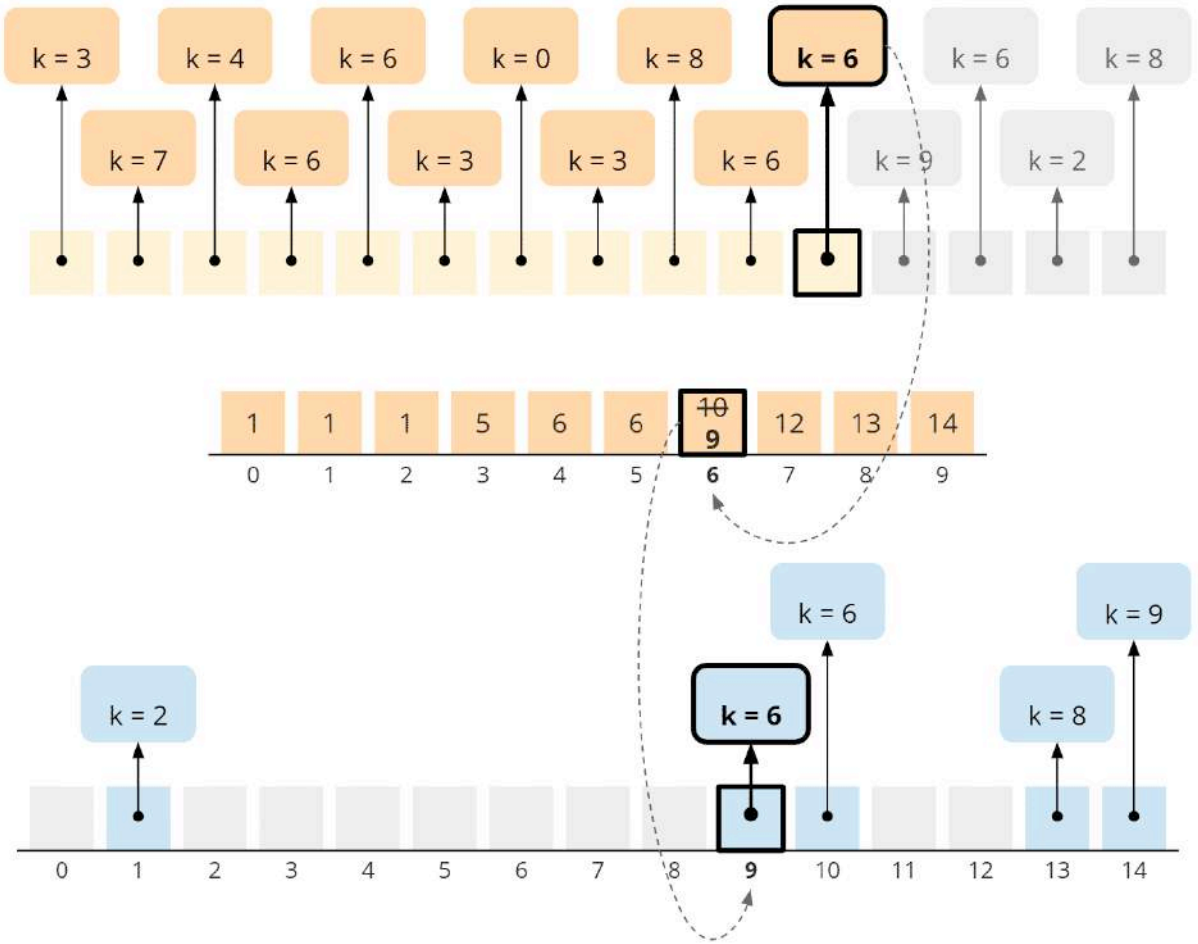
# Allgemeiner Counting Sort - Zurückschreiben



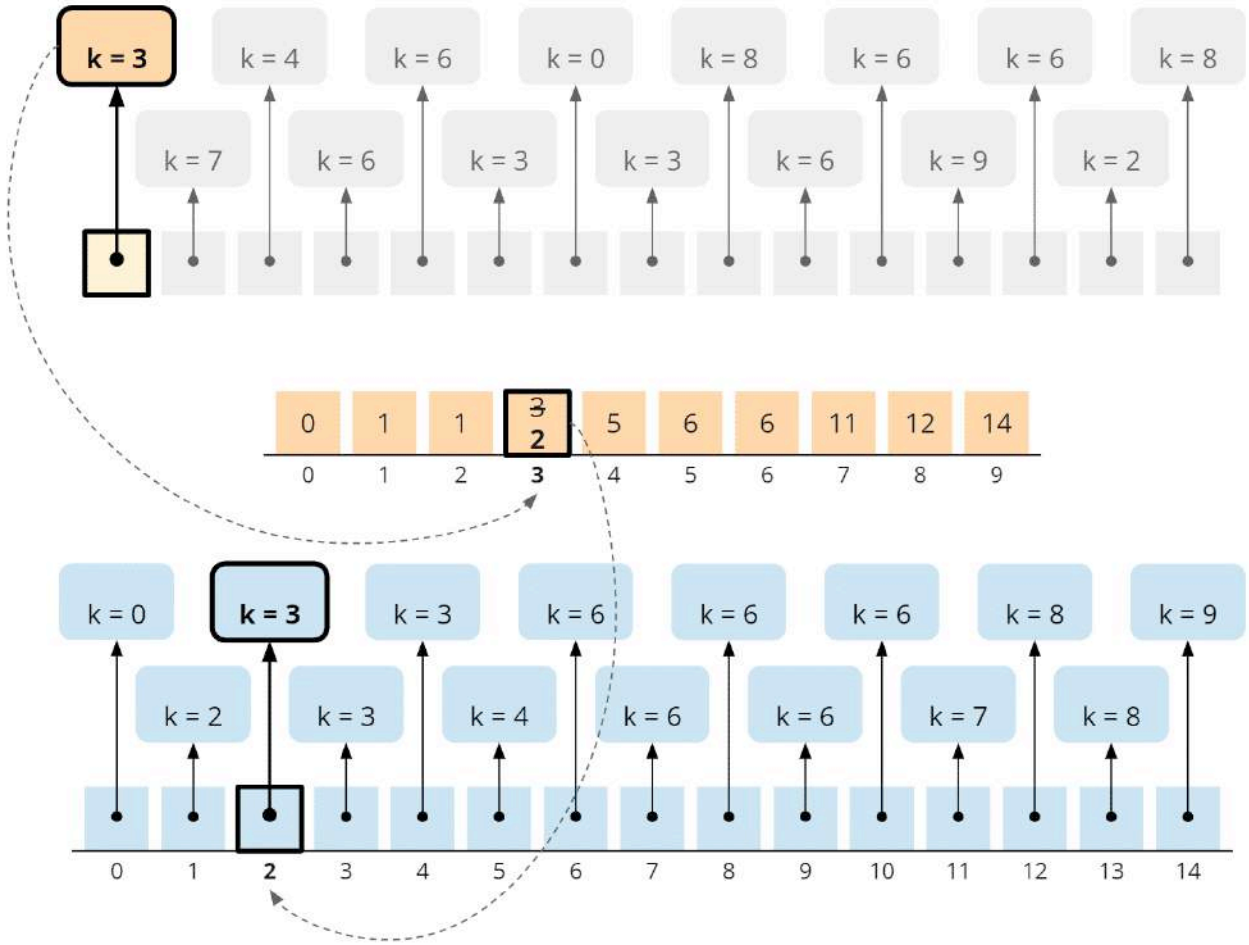
# Allgemeiner Counting Sort - Zurückschreiben



# Allgemeiner Counting Sort - Zurückschreiben



# Allgemeiner Counting Sort - Zurückschreiben



# Allgemeiner Counting Sort in Java

```java
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);
 }
```

**Laufzeit von Counting Sort:** $\mathcal{O}(n + k)$

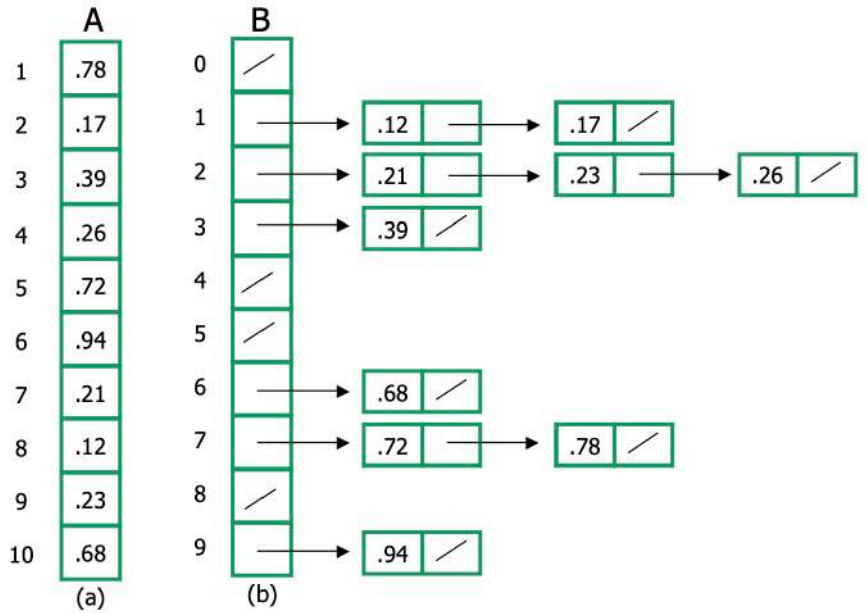
# Einfacher Counting Sort in Java

```
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;
    }
  }
}
```

Verschiebe um kleinste Zahl, um Intervall positiv zu machen

# Bucket Sort - Idee

- Verteile die Eigabedaten in $k$ gleich große Buckets
- Sortiere jeden Bucket
- Gehe durch die Buckets und sammle die Werte




| 0.12 | 0.17 | 0.21 | 0.23 | 0.26 | 0.39 | 0.68 | 0.72 | 0.78 | 0.94 |
|---|---|---|---|---|---|---|---|---|---|

# Bucket Sort - Laufzeit

- Wir gehen davon aus, dass kein Bucket “zu voll” wird
- Verteilen auf die Buckets geht in $\mathcal{O}(n)$
- Intuitiv: Wenn jeder Bucket eine konstante Anzahl an Elementen hat, ist die Laufzeit zum sortieren für jedes Bucket in etwa konstant ➠ $\mathcal{O}(1)$
- Also $\mathcal{O}(n)$ für das sortieren aller Buckets
- Plausibel, aber umständlich zu beweisen…

# Sortieren in linearer Zeit - Radix Sort

- Sortieren von Strings über einem endlichen Alphabet durch mehrmaliges Anwenden von Bucket Sort
- Angenommen: Alle Wörter haben die Länge $l$
- Z.b. Postleitzahlen
- Sortieren Sie die Strings `hefe`, `bach`, `gaga`, `cafe`, `geha`
- Beispiel an der Tafel

**Laufzeit von Radix Sort: $\mathcal{O}(n)$, weil $l \cdot n$ Durchläufe**