Sortierverfahren

Bucketsort, Heapsort, Radixsort

M. Heckel

2019–08–15

Gliederung

Allgemeines

Bucketsort

Heapsort

Radixsort

Quellen

Interne und externe Verfahren

Interne und externe Verfahren

- Intern
  - Zu sortierende Elemente liegen im RAM
  - Lesen/Schreiben spielen keine besonders große Rolle

Interne und externe Verfahren

- Intern
  - Zu sortierende Elemente liegen im RAM
  - Lesen/Schreiben spielen keine besonders große Rolle
- Extern
  - Zu sortierende Elemente liegen auf externen Medien (z.B. Festplatte, Band)
  - Lesen/Schreiben spielen eine relativ große Rolle

Stabilität

Stabilität

- Instabile Verfahren
  - Behalten die Reihenfolge der Elemente mit gleichen Wert nicht bei
  - Für Datensätze, die nur nach einem Kriterium sortiert werden können, geeignet

Stabilität

- Instabile Verfahren
  - Behalten die Reihenfolge der Elemente mit gleichen Wert nicht bei
  - Für Datensätze, die nur nach einem Kriterium sortiert werden können, geeignet
- Stabile Verfahren
  - Behalten die Reihenfolge der Elemente mit gleichen Wert bei
  - Auch für Datensätze, die nach mehreren Kriterien sortiert werden können, geeignet

in-place

in-place

- in-place
  - Sortieren auf dem gegebenen Array
  - Zusätzlich benötigter Speicher konstant

in-place

- in-place
  - Sortieren auf dem gegebenen Array
  - Zusätzlich benötigter Speicher konstant
- Nicht in-place
  - Kopieren in anderes Array (ggf. mehrere)
  - Zusätzlich benötigter Speicher hängt von Anzahl der zu sortierenden Elemente ab

Komplexität

Komplexität

- Die Komplexität wird mit der Landau-Notation $\mathcal{O}$ angegeben

Komplexität

- Die Komplexität wird mit der Landau-Notation $\mathcal{O}$ angegeben
- Häufig werden zwei Eigenschaften betrachtet:
  - Laufzeitkomplexität
  - Speicherkomplexität

Komplexität

- Die Komplexität wird mit der Landau-Notation $\mathcal{O}$ angegeben
- Häufig werden zwei Eigenschaften betrachtet:
  - Laufzeitkomplexität
  - Speicherkomplexität
- Meist wird die Komplexität in Bezug auf die Problemkomplexität n angegeben
  - Beim Sortieren oder Suchen: Anzahl der Elemente, die sortiert bzw. durchsucht werden müssen
  - Beim Multiplizieren: Anzahl der Stellen der Zahl
  - Bei Approximationen: Genauigkeit (Anzahl richtiger Nachkommastellen)

Komplexität

Komplexität

- Häufig werden drei Fälle betrachtet:
  - Best Case
  - Average Case (in der Praxis meistens verwendet)
  - Worst Case

Komplexität

- Häufig werden drei Fälle betrachtet:
  - Best Case
  - Average Case (in der Praxis meistens verwendet)
  - Worst Case
- Vergleichsbasierte Sortieralgorithmen erreichen im Average Case bestenfalls eine Laufzeitkomplexität von $\mathcal{O}(n \log(n))$
  - Besser als $\mathcal{O}(n^2)$
  - Schlechter als $\mathcal{O}(n)$

Bucketsort — Funktionsweise

Bucketsort — Funktionsweise

- Eimer (“Buckets”) erstellen
  - Jeder Bucket deckt einen bestimmten Bereich ab (z.B. 0–10, 10–20, etc.)
  - Gesamter Bereich der Sortiergröße muss durch Buckets abgedeckt sein

Bucketsort — Funktionsweise

- Eimer (“Buckets”) erstellen
  - Jeder Bucket deckt einen bestimmten Bereich ab (z.B. 0–10, 10–20, etc.)
  - Gesamter Bereich der Sortiergröße muss durch Buckets abgedeckt sein
- Elemente des Arrays in die entsprechenden Buckets sortieren

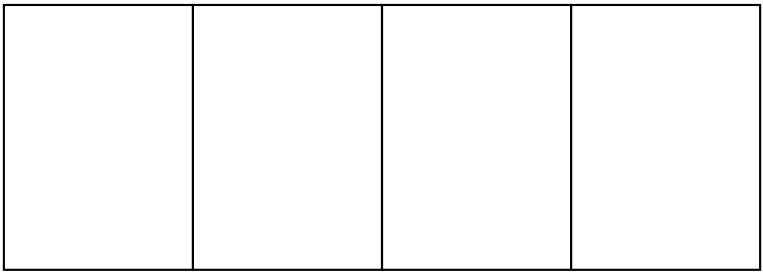
Bucketsort — Funktionsweise

- Eimer (“Buckets”) erstellen
  - Jeder Bucket deckt einen bestimmten Bereich ab (z.B. 0–10, 10–20, etc.)
  - Gesamter Bereich der Sortiergröße muss durch Buckets abgedeckt sein
- Elemente des Arrays in die entsprechenden Buckets sortieren
- Buckets vergleichsbasiert sortieren

Bucketsort — Funktionsweise

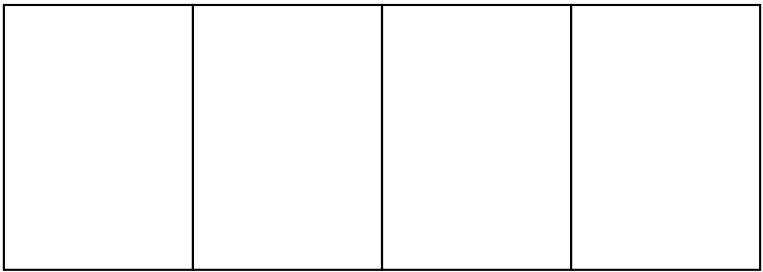
- Eimer (“Buckets”) erstellen
  - Jeder Bucket deckt einen bestimmten Bereich ab (z.B. 0–10, 10–20, etc.)
  - Gesamter Bereich der Sortiergröße muss durch Buckets abgedeckt sein
- Elemente des Arrays in die entsprechenden Buckets sortieren
- Buckets vergleichsbasiert sortieren
- Bucket-Elemente in Ausgangsarray kopieren

Bucketsort — Funktionsweise



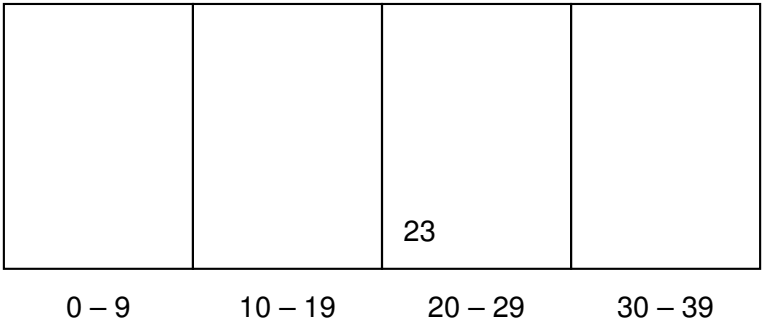
23	10	27	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



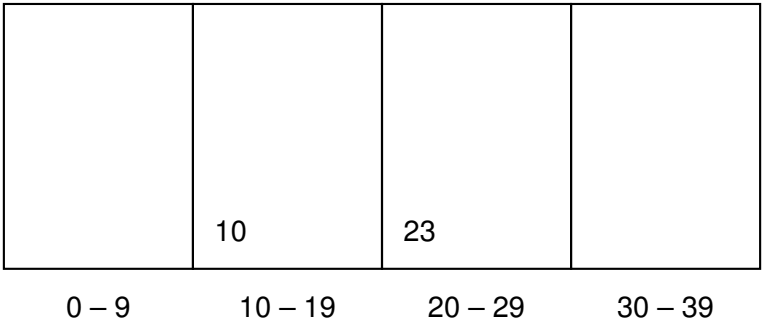
23	10	27	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



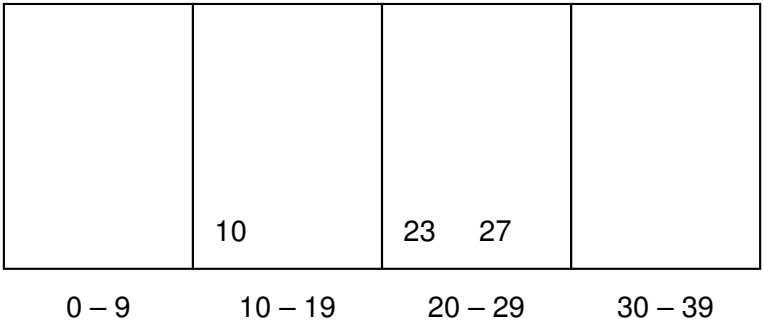
23	10	27	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



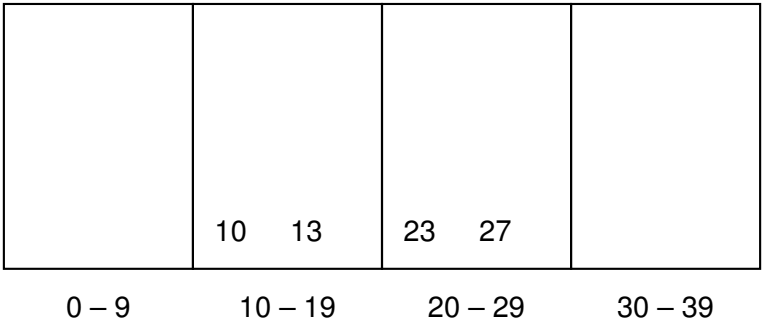
23	10	27	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



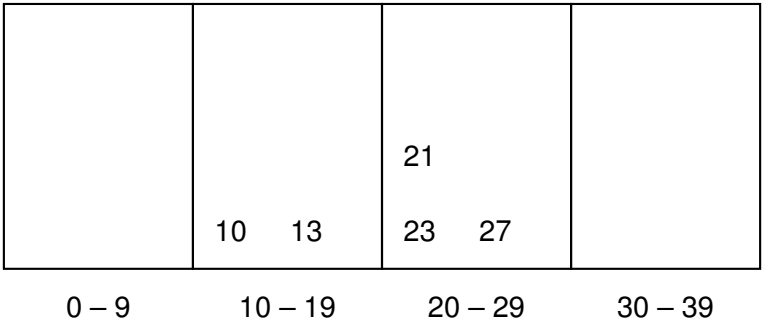
23	10	27	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



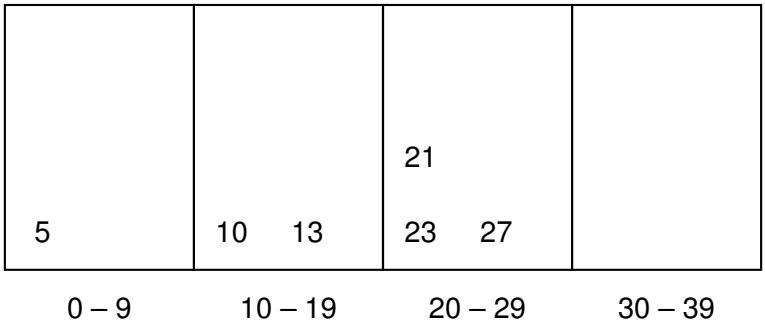
23	10	27	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



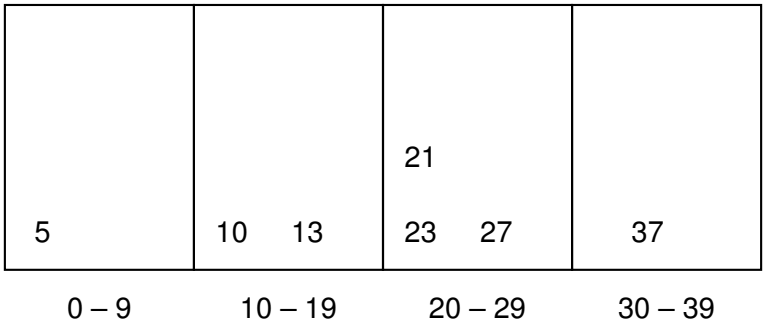
23	10	27	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



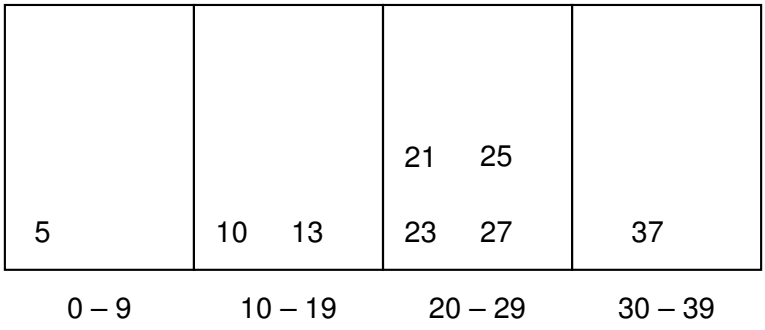
23	10	27	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



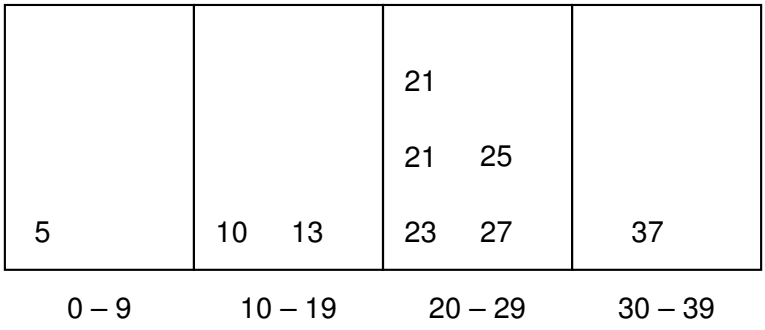
23	10	27	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



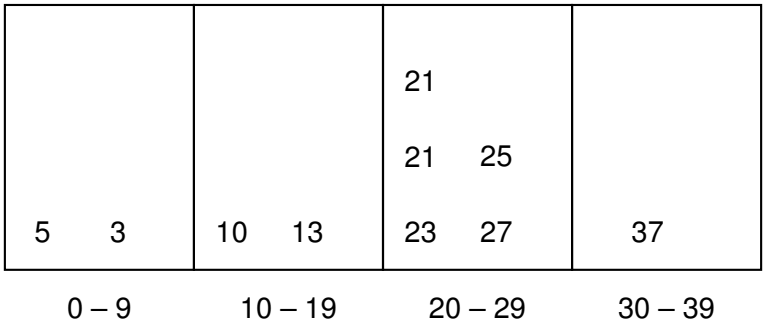
23	10	27	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



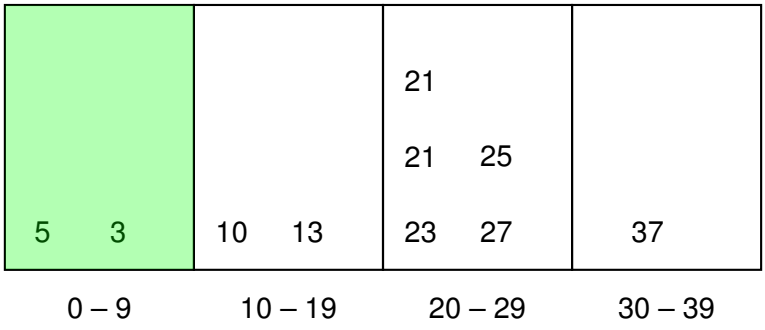
23	10	27	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



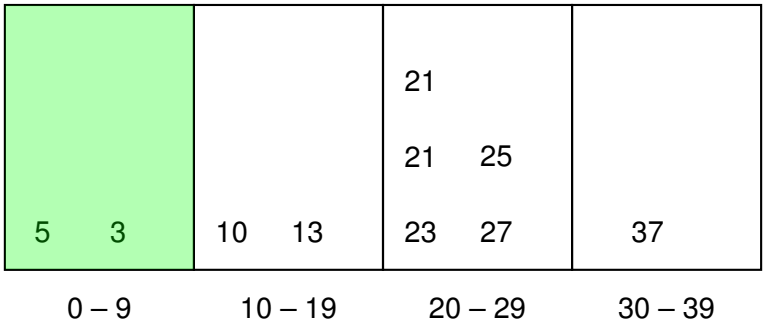
23	10	27	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



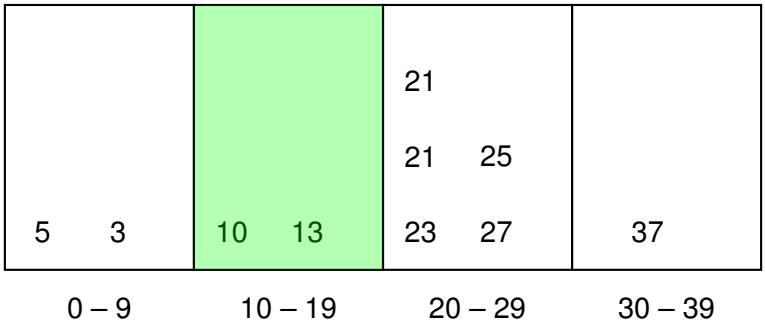
23	10	27	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



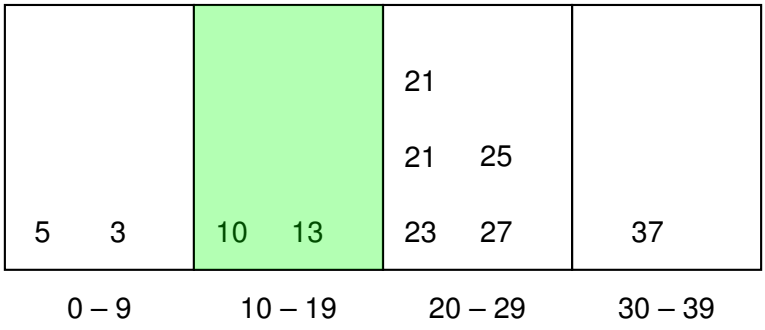
3	10	27	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



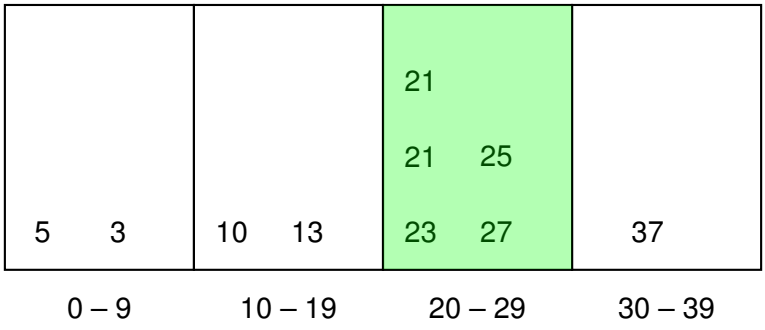
3	5	27	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



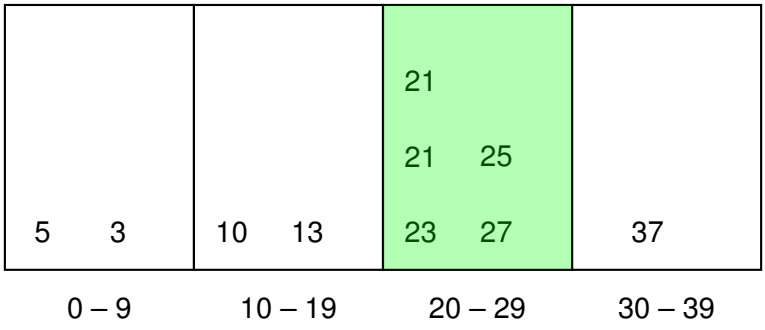
3	5	10	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



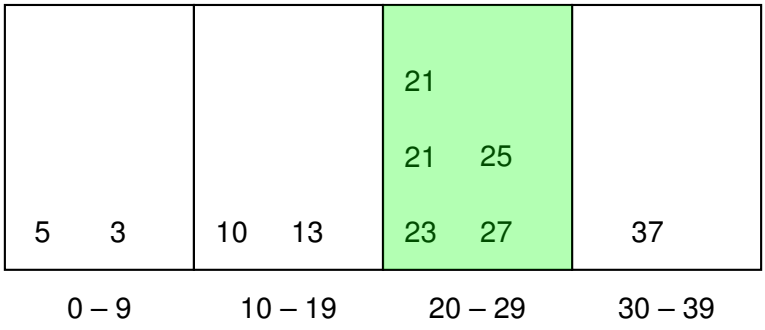
3	5	10	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



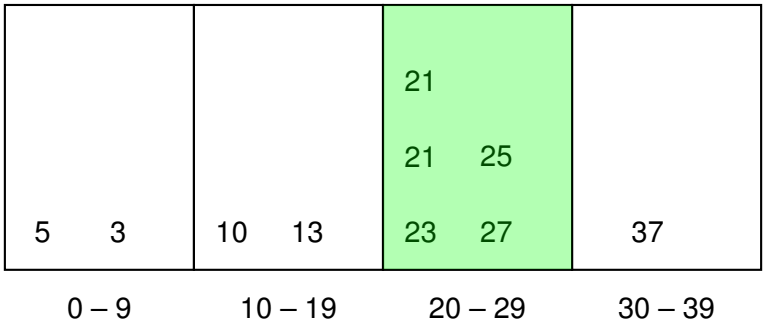
3	5	10	13	21	5	37	25	21	3

Bucketsort — Funktionsweise



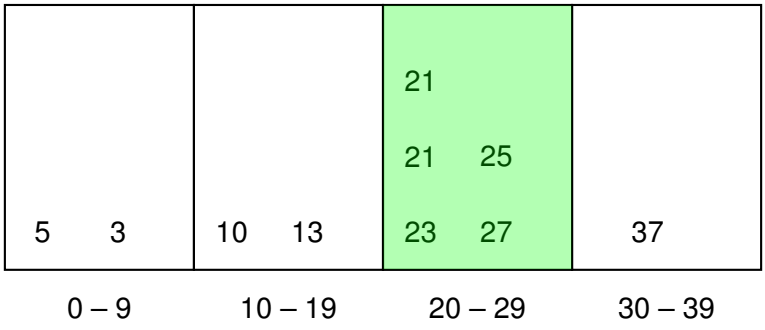
3	5	10	13	21	21	37	25	21	3

Bucketsort — Funktionsweise



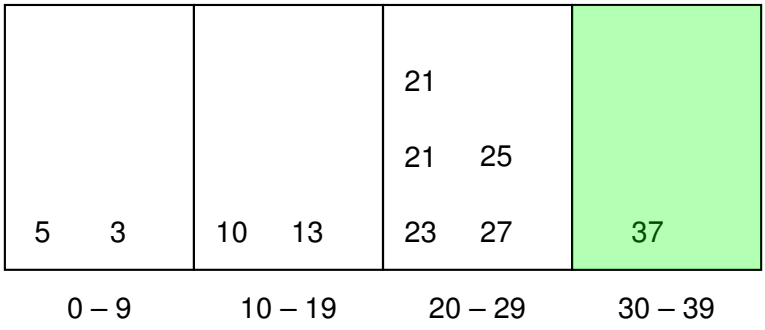
3	5	10	13	21	21	23	25	21	3

Bucketsort — Funktionsweise



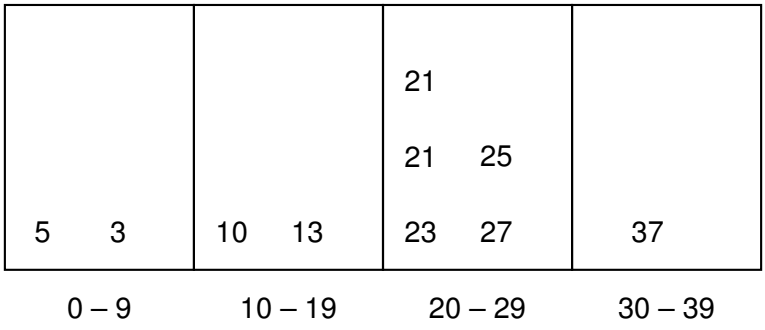
3	5	10	13	21	21	23	25	21	3

Bucketsort — Funktionsweise



3	5	10	13	21	21	23	25	27	3

Bucketsort — Funktionsweise



3	5	10	13	21	21	23	25	27	37

Bucketsort — Code (1)

```
int[] bucketsort(int[] array, int bucketnum) {
    int[][] buckets = int[bucketnum];
    int max = 0;
    for(int i = 0; i < array.lenght; i++) {
        if(array[i] > max) {
            max = array[i];
        }
    }

    for(int i = 0; i < array.length; i++) {
        buckets[array[i] * bucketnum / (max + 1)].add(array[i])
    }
```

Bucketsort — Code (2)

```
    int [] sorted;

    for(int i = 0; i < bucketnum; i++) {
        compareSort(buckets[i])
        for(int k = 0; k < buckets[i].length; k++) {
            sorted.add(buckets[i][k])
        }
    }
    return sorted;
}
```

Bucketsort — Steckbrief

Laufzeitkomplexität (best case)	$\mathcal{O}(n)$
Laufzeitkomplexität (avarage case)	$\mathcal{O}(n + \frac{n^2}{k} + k)$ (k ist die Anzahl der Buckets)
Laufzeitkomplexität (worst case)	$\mathcal{O}(n^2)$
Speicherkomplexität	$\mathcal{O}(n)$ (Abhängig vom verwendeten Sub-Verfahren)
Stabilität	Abhängig vom verwendeten Sub-Verfahren
Vergleichsbasiertes Verfahren	Nein

Heap

Heap

- Ein Heap ist eine Datenstruktur, die folgende Eigenschaften erfüllt:
  - Baumstruktur, bei der jeder Knoten maximal zwei Nachfolger hat
  - Max-Heap: Alle Nachfolger eines Knotens sind maximal so groß wie der Knoten selbst
  - Min-Heap: Alle Nachfolger eines Knotens sind mindestens so groß wie der Knoten selbst

Heap

- Ein Heap ist eine Datenstruktur, die folgende Eigenschaften erfüllt:
  - Baumstruktur, bei der jeder Knoten maximal zwei Nachfolger hat
  - Max-Heap: Alle Nachfolger eines Knotens sind maximal so groß wie der Knoten selbst
  - Min-Heap: Alle Nachfolger eines Knotens sind mindestens so groß wie der Knoten selbst
- Der erste Knoten heißt **Wurzel**.

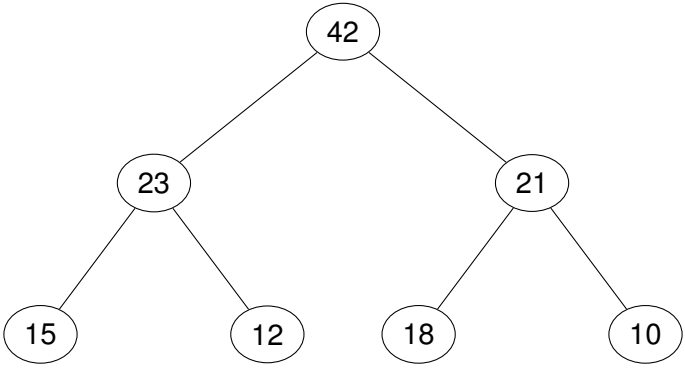
Heap

- Ein Heap ist eine Datenstruktur, die folgende Eigenschaften erfüllt:
  - Baumstruktur, bei der jeder Knoten maximal zwei Nachfolger hat
  - Max-Heap: Alle Nachfolger eines Knotens sind maximal so groß wie der Knoten selbst
  - Min-Heap: Alle Nachfolger eines Knotens sind mindestens so groß wie der Knoten selbst
- Der erste Knoten heißt **Wurzel**.
- Knoten ohne Nachfolger heißen **Blatt**.

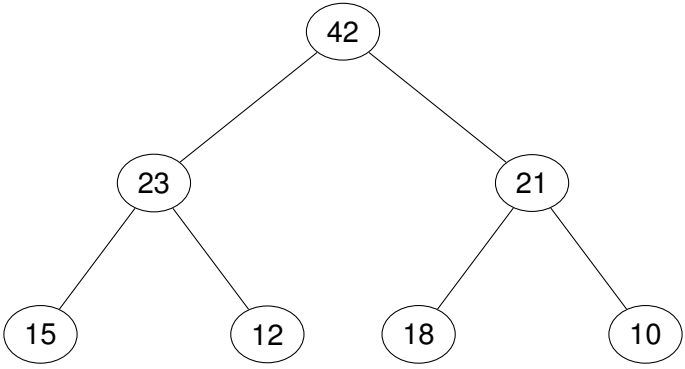
Heap

- Ein Heap ist eine Datenstruktur, die folgende Eigenschaften erfüllt:
  - Baumstruktur, bei der jeder Knoten maximal zwei Nachfolger hat
  - Max-Heap: Alle Nachfolger eines Knotens sind maximal so groß wie der Knoten selbst
  - Min-Heap: Alle Nachfolger eines Knotens sind mindestens so groß wie der Knoten selbst
- Der erste Knoten heißt **Wurzel**.
- Knoten ohne Nachfolger heißen **Blatt**.
- Jeder einzelne Knoten ist per Definition ein Heap

Heap — Funktionsweise



Heap — Funktionsweise

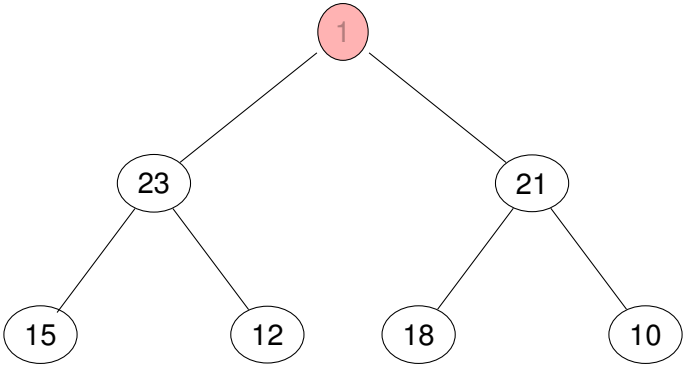


42	23	21	15	12	18	10

Heap — Code

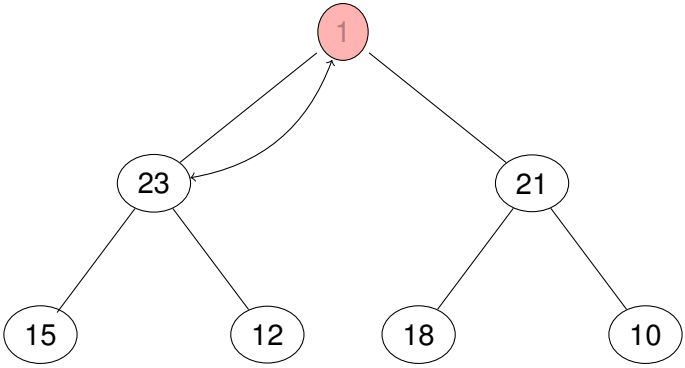
```
int getLeftChildIndex(int parent_index) {
    return 2 * parent_index + 1;
}
int getRightChildIndex(int parent_index) {
    return 2 * parent_index + 2;
}
```

Heapify — Funktionsweise



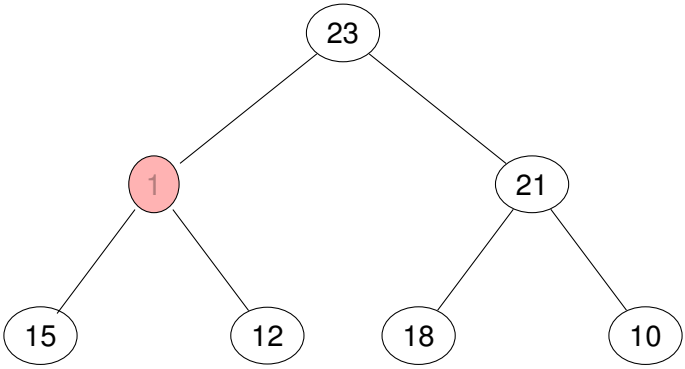
1	23	21	15	12	18	10

Heapify — Funktionsweise



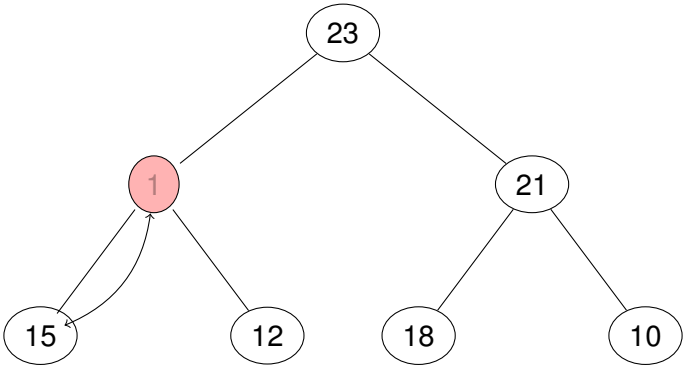
1	23	21	15	12	18	10

Heapify — Funktionsweise



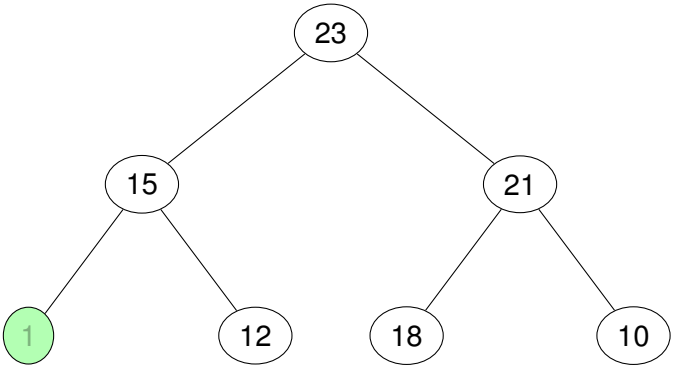
23	1	21	15	12	18	10

Heapify — Funktionsweise



23	1	21	15	12	18	10

Heapify — Funktionsweise



23	15	21	1	12	18	10

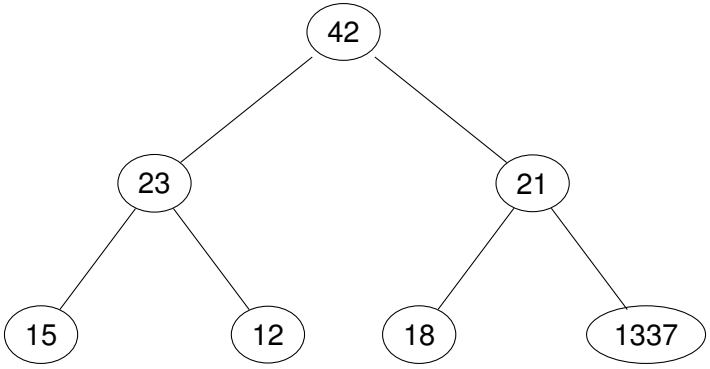
Heapify — Code (1)

```
void heapify(int[] array, int elementIndex, int lastIndex) {
    int index = elementIndex;
    while(index < lastIndex) {
        int left = getLeftChildIndex(index);
        int right = getRightChildIndex(index);
        if(left > lastIndex) {
            return;
        }
        if(right > lastIndex || array[left] > array[right]) {
            if(array[left] > array[index]) {
                int tmp = array[left];
                array[left] = array[index];
                array[index] = tmp;
                index = left;
            }
```

Heapify — Code (2)

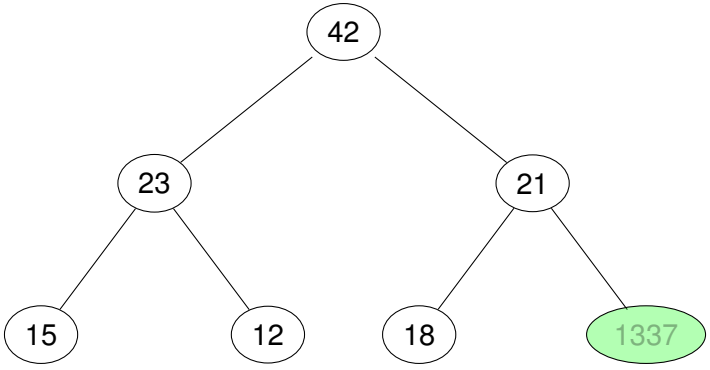
```
            else {
                break;
            }
        }
        else {
            if(array[right] > array[index]) {
                int tmp = array[right];
                array[right] = array[index];
                array[index] = tmp;
                index = right;
            }
            else {
                break;
            }
        }
    }
}
```

BuildHeep — Funktionsweise



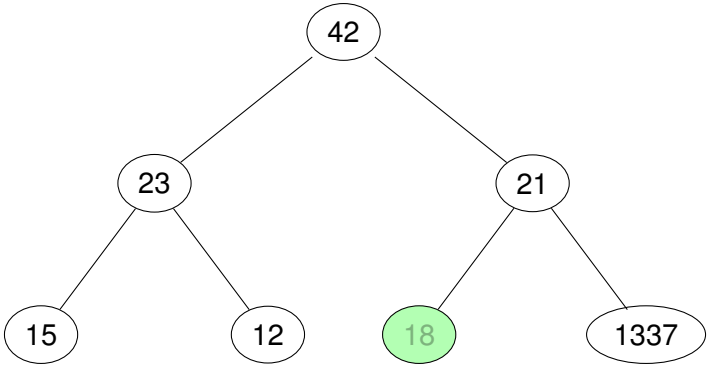
42	23	21	15	12	18	1337

BuildHeep — Funktionsweise



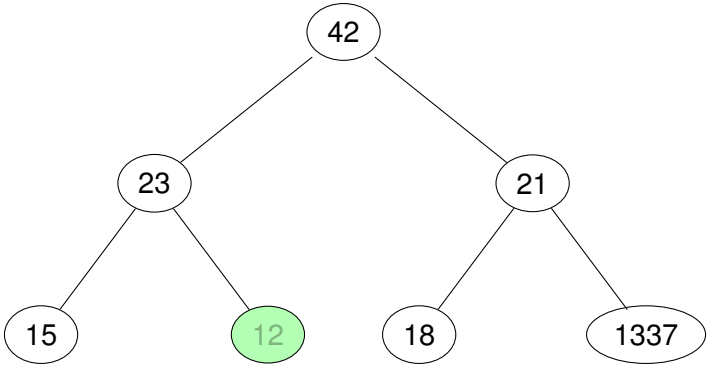
42	23	21	15	12	18	1337

BuildHeep — Funktionsweise



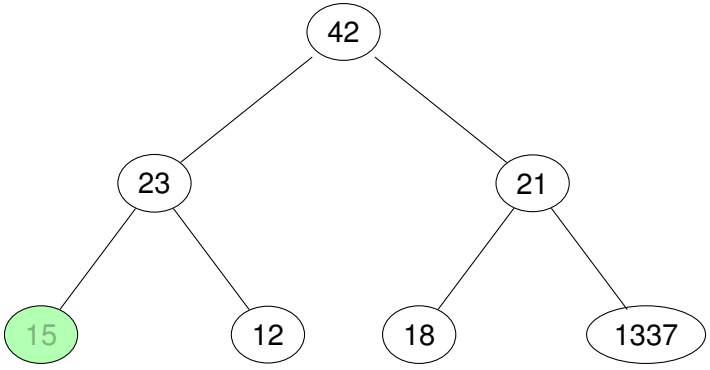
42	23	21	15	12	18	1337

BuildHeep — Funktionsweise



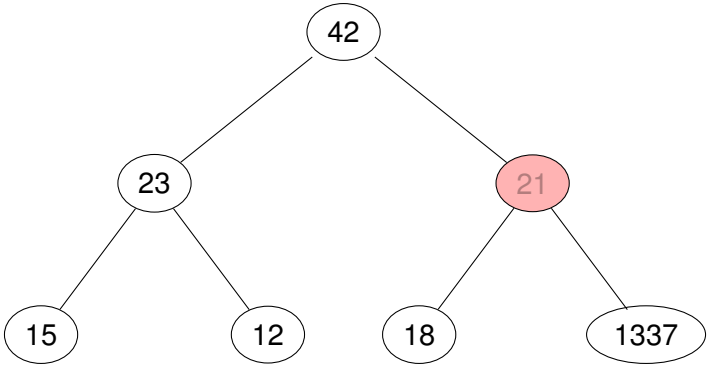
42	23	21	15	12	18	1337

BuildHeep — Funktionsweise



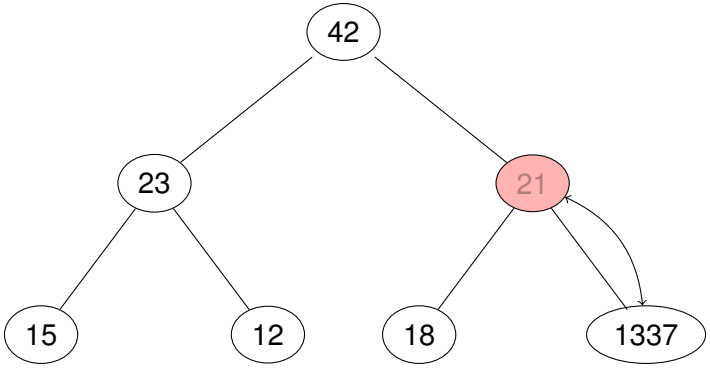
42	23	21	15	12	18	1337

BuildHeep — Funktionsweise



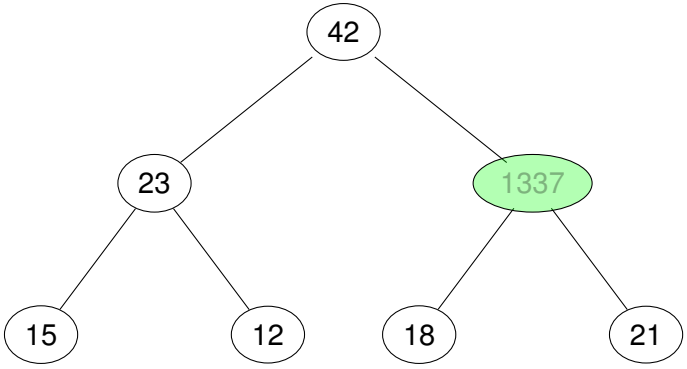
42	23	21	15	12	18	1337

BuildHeep — Funktionsweise



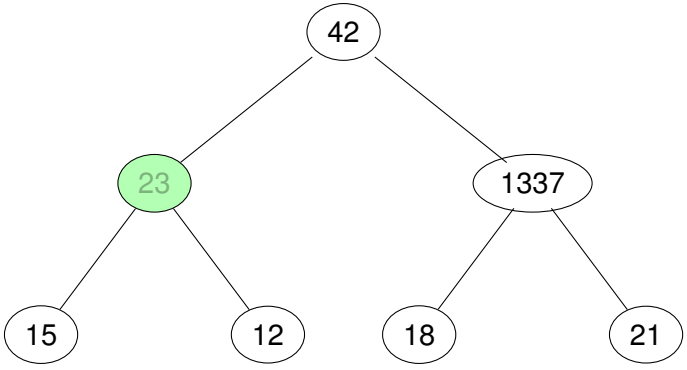
42	23	21	15	12	18	1337

BuildHeep — Funktionsweise



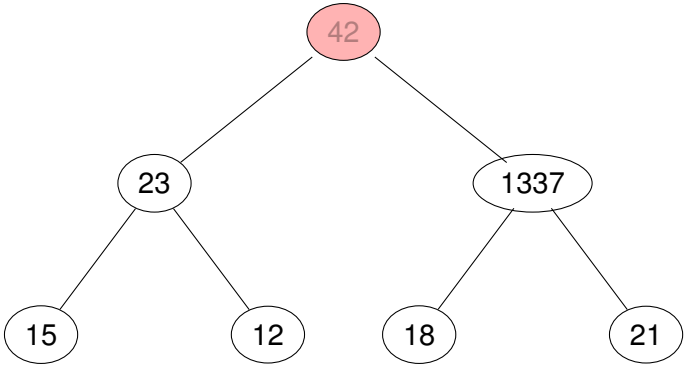
42	23	1337	15	12	18	21

BuildHeep — Funktionsweise



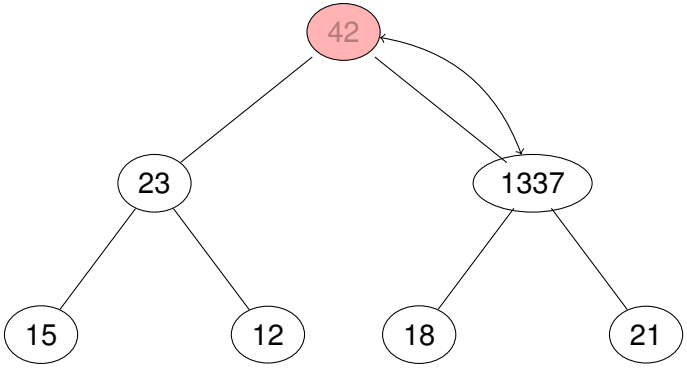
42	23	1337	15	12	18	21

BuildHeep — Funktionsweise



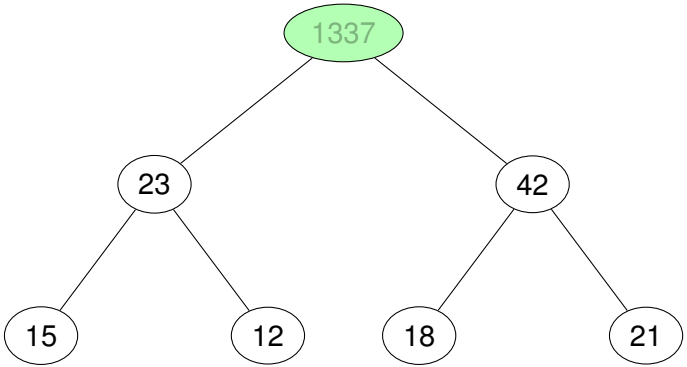
42	23	1337	15	12	18	21

BuildHeep — Funktionsweise



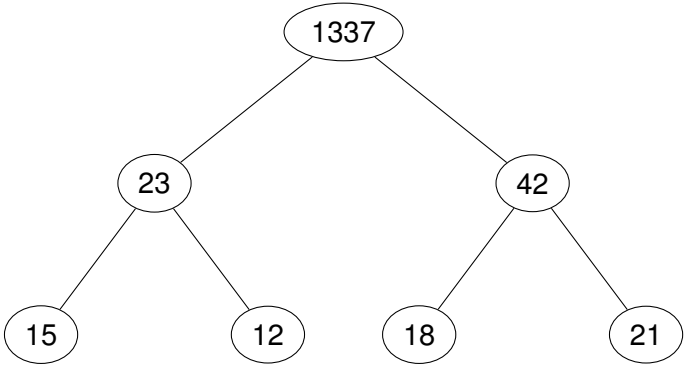
42	23	1337	15	12	18	21

BuildHeep — Funktionsweise



1337	23	42	15	12	18	21

BuildHeep — Funktionsweise



1337	23	42	15	12	18	21

BuildHeep — Code

```
void buildHeap(int[] array, int lastIndex) {
    for(int index = array.length / 2; index >= 0; index--) {
        int left = getLeftChildIndex(elementIndex);
        int right = getRightChildIndex(elementIndex);
        if(left > lastIndex) {
            continue;
        }
        if(array[left] > array[index] || (right <= lastIndex && array[
            right] > array[index])) {
            heapify(array, index, lastIndex);
        ]
    }
}
```

Heapsort — Funktionsweise

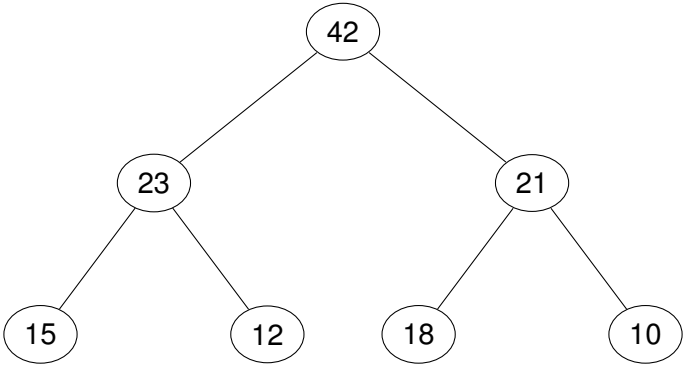
Heapsort — Funktionsweise

- Elemente im Array so anordnen, dass die Heap-Eigenschaft erfüllt ist

Heapsort — Funktionsweise

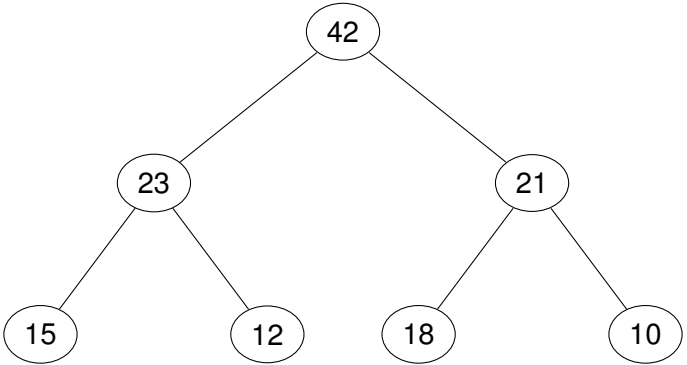
- Elemente im Array so anordnen, dass die Heap-Eigenschaft erfüllt ist
- So lange Elemente auf dem Heap vorhanden sind:
  - Das erste Element mit dem Element am Ende des Arrays tauschen und Heap um 1 verkleinern
  - Heap-Eigenschaft wiederherstellen, indem Heapify auf das erste Element aufgerufen wird

Heapsort — Funktionsweise

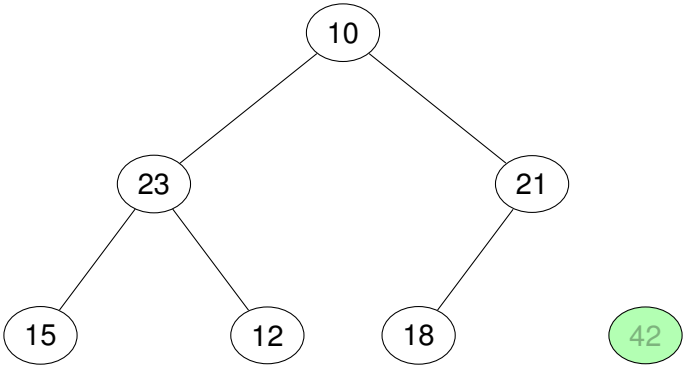


42	23	21	15	12	18	10

Heapsort — Funktionsweise

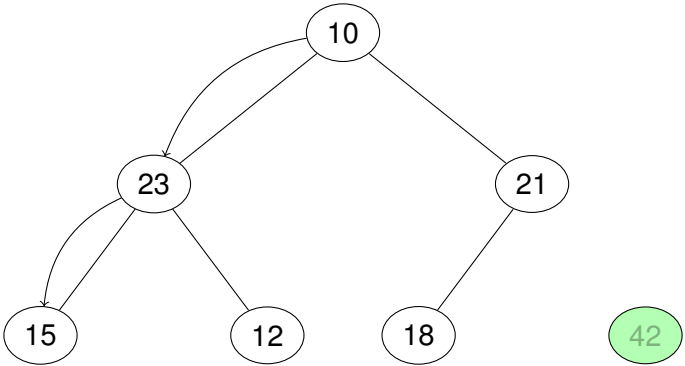


Heapsort — Funktionsweise



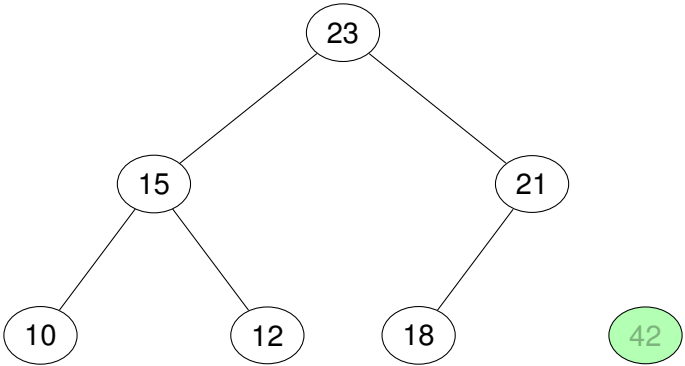
10	23	21	15	12	18	42

Heapsort — Funktionsweise



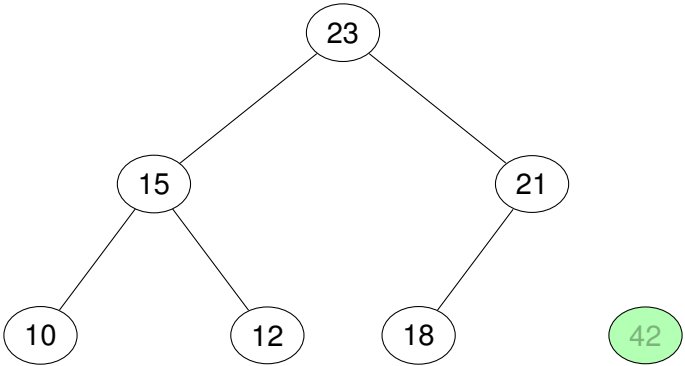
10	23	21	15	12	18	42

Heapsort — Funktionsweise

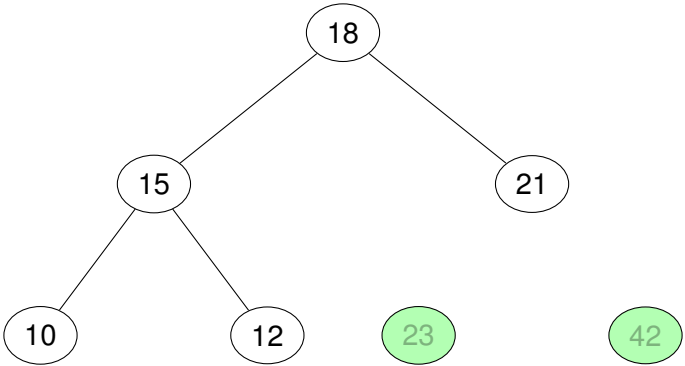


23	15	21	10	12	18	42

Heapsort — Funktionsweise

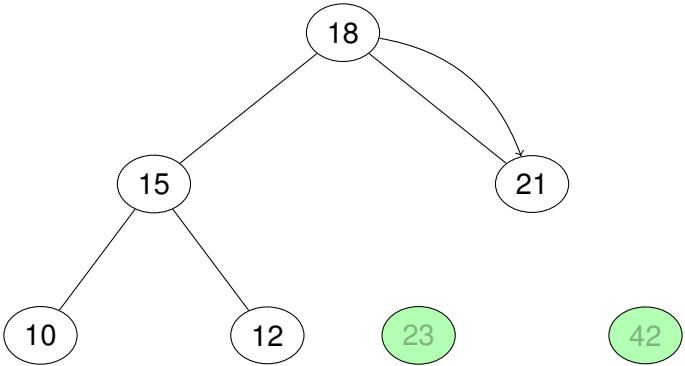


Heapsort — Funktionsweise



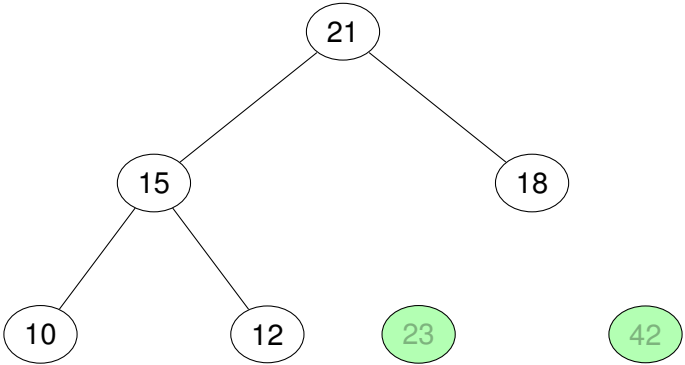
18	15	21	10	12	23	42

Heapsort — Funktionsweise



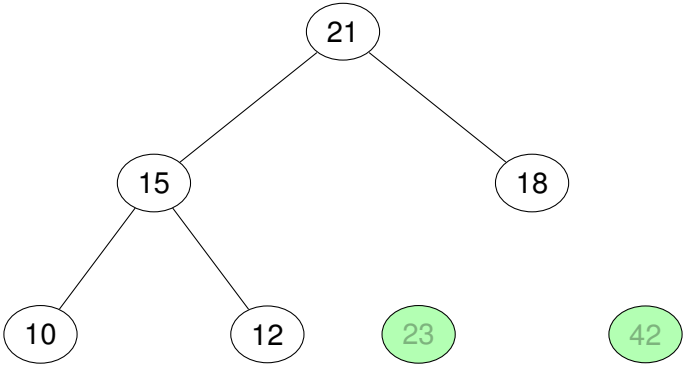
18	15	21	10	12	23	42

Heapsort — Funktionsweise

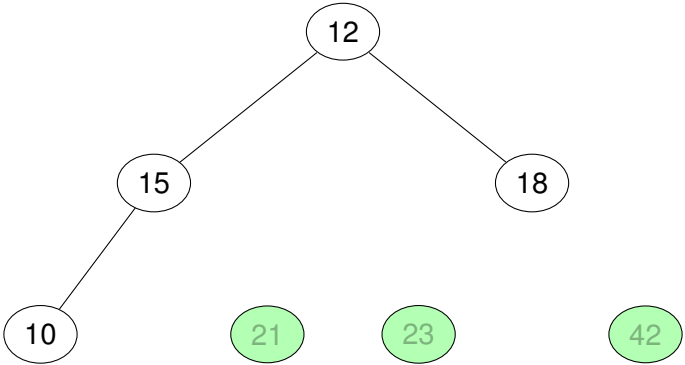


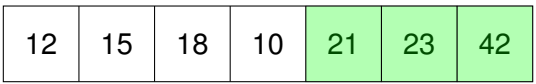
21	15	18	10	12	23	42

Heapsort — Funktionsweise

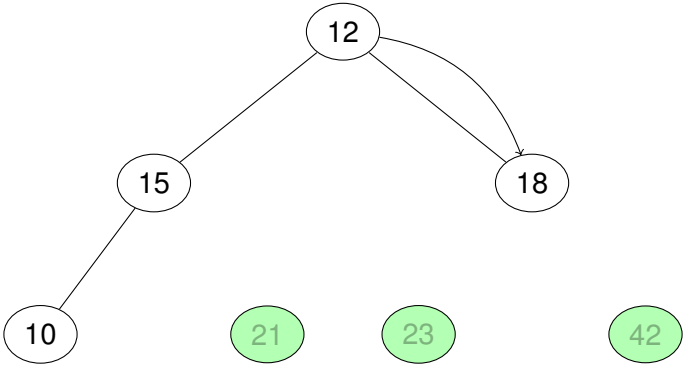


Heapsort — Funktionsweise



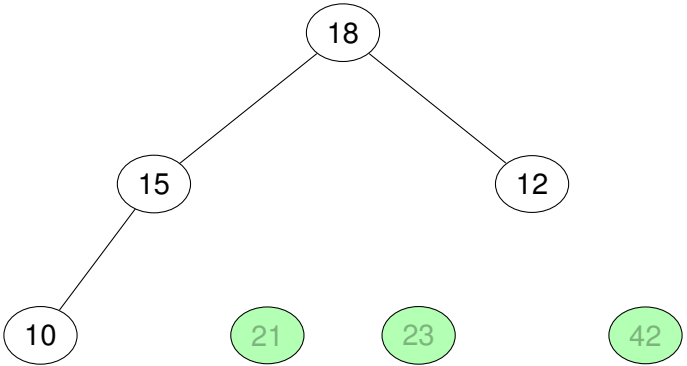


Heapsort — Funktionsweise



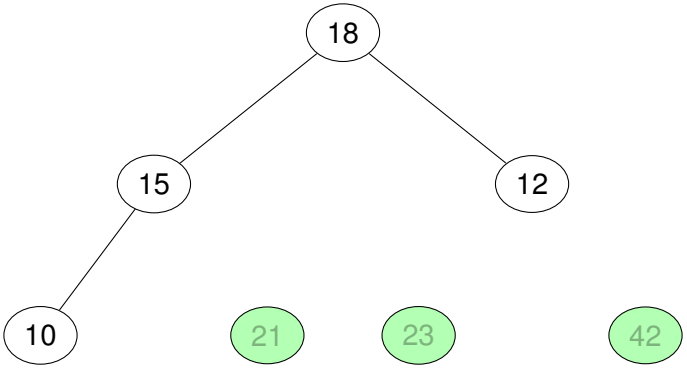
12	15	18	10	21	23	42

Heapsort — Funktionsweise

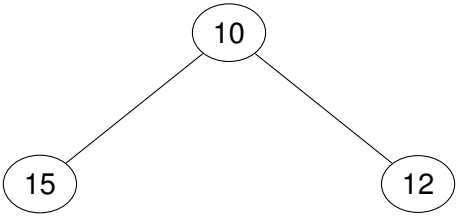


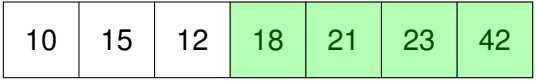
18	15	12	10	21	23	42

Heapsort — Funktionsweise

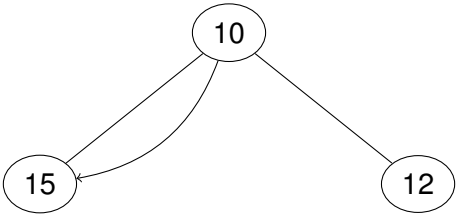


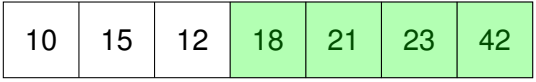
Heapsort — Funktionsweise



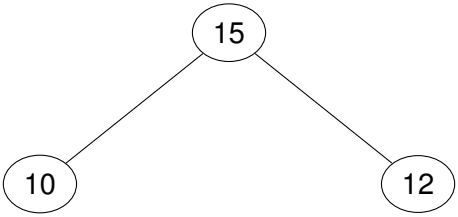


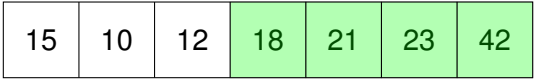
Heapsort — Funktionsweise



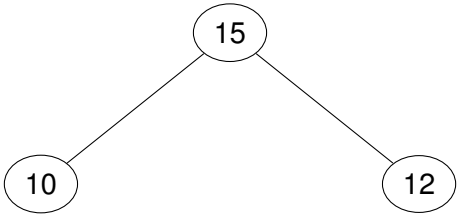


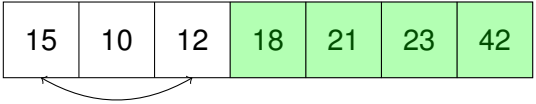
Heapsort — Funktionsweise



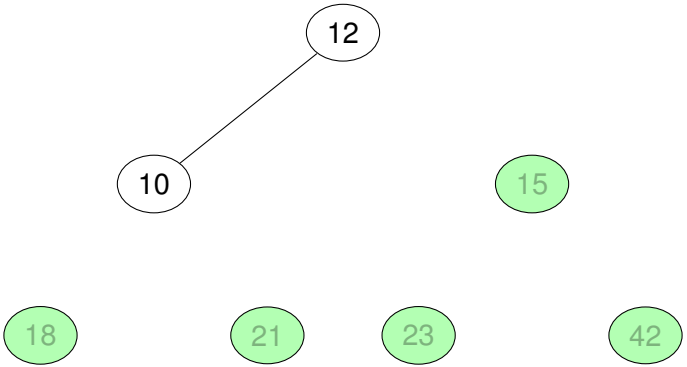


Heapsort — Funktionsweise

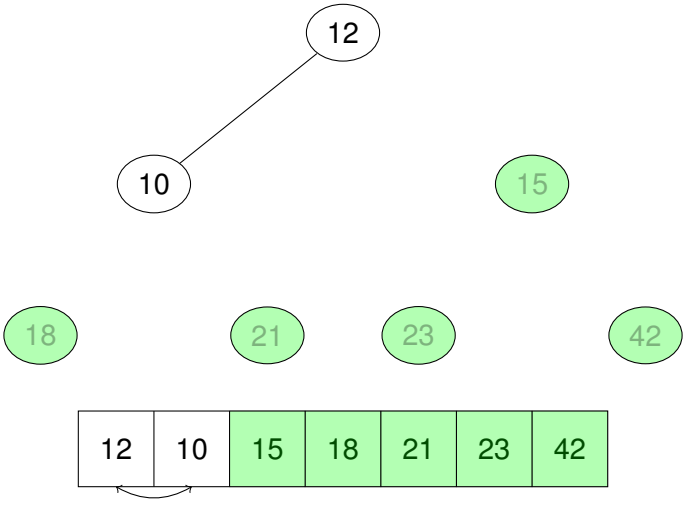




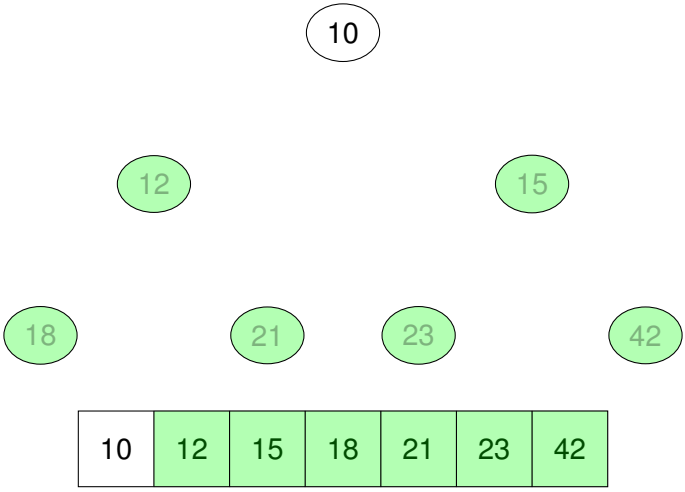
Heapsort — Funktionsweise



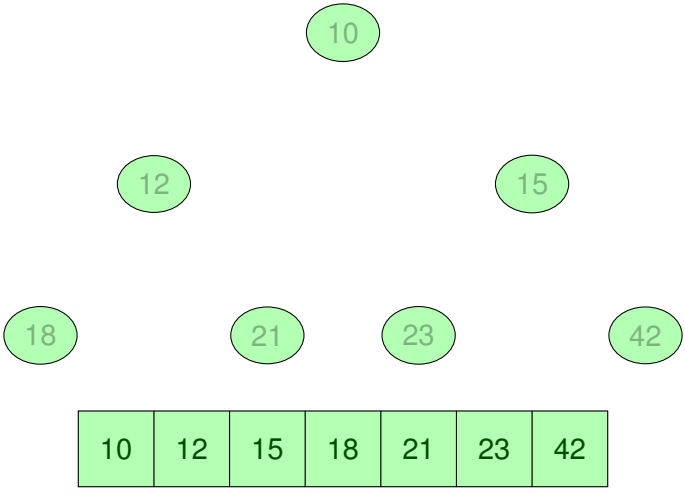
Heapsort — Funktionsweise



Heapsort — Funktionsweise



Heapsort — Funktionsweise



Heapsort — Code

```
int[] heapsort(int[] array) {
    int lastIndex = array.length - 1;
    buildHeap(array, lastIndex);
    while(lastIndex > 0) {
        int tmp = array[lastIndex];
        array[lastIndex] = array[0];
        array[0] = tmp;
        lastIndex = lastIndex - 1;
        heapify(array, 0, lastIndex);
    }
    return array;
}
```

Heapsort — Steckbrief

Laufzeitkomplexität (best case)	$\mathcal{O}(n \log(n))$
Laufzeitkomplexität (avarage case)	$\mathcal{O}(n \log(n))$
Laufzeitkomplexität (worst case)	$\mathcal{O}(n \log(n))$
Speicherkomplexität	$\mathcal{O}(1)$
Stabilität	Nicht stabil
Vergleichsbasiertes Verfahren	Ja

Radixsort — Funktionsweise

Radixsort — Funktionsweise

- Basiert auf Bucketsort

Radixsort — Funktionsweise

- Basiert auf Bucketsort
- Gut geeignet zum Sortieren von positiven Ganzzahlen, andere Schlüssel eher schwierig

Radixsort — Funktionsweise

- Basiert auf Bucketsort
- Gut geeignet zum Sortieren von positiven Ganzzahlen, andere Schlüssel eher schwierig
- Generell wird ein Stellenwertsystem gewählt, z.B. Dualsystem, Dezimalsystem, etc.

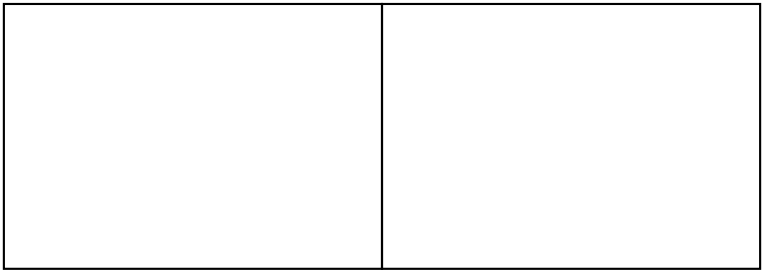
Radixsort — Funktionsweise

- Basiert auf Bucketsort
- Gut geeignet zum Sortieren von positiven Ganzzahlen, andere Schlüssel eher schwierig
- Generell wird ein Stellenwertsystem gewählt, z.B. Dualsystem, Dezimalsystem, etc.
- Es gibt so viele Buckets wie Ziffern in dem gewählten Stellenwertsystem

Radixsort — Funktionsweise

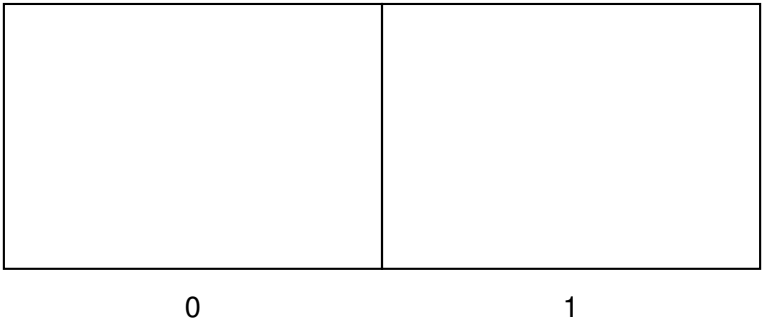
- Basiert auf Bucketsort
- Gut geeignet zum Sortieren von positiven Ganzzahlen, andere Schlüssel eher schwierig
- Generell wird ein Stellenwertsystem gewählt, z.B. Dualsystem, Dezimalsystem, etc.
- Es gibt so viele Buckets wie Ziffern in dem gewählten Stellenwertsystem
- Vorgehensweise:
  - Iterieren über die Stellen der Zahl (von niederwertigster zu höchstwertigster Stelle)
  - Sortieren der Zahl in den entsprechenden “Eimer”
  - Kopieren der Zahlen in das Ausgangsarray

Radixsort — Funktionsweise



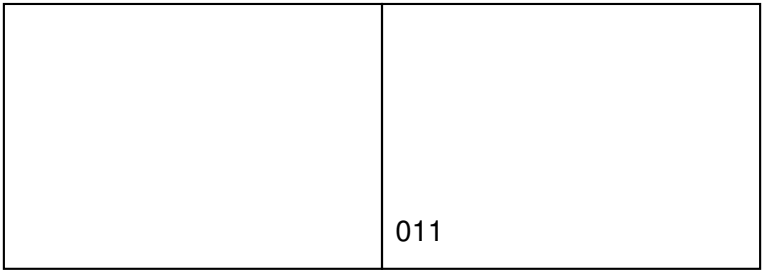
011	001	100	010	111	101	000	011	110	101

Radixsort — Funktionsweise



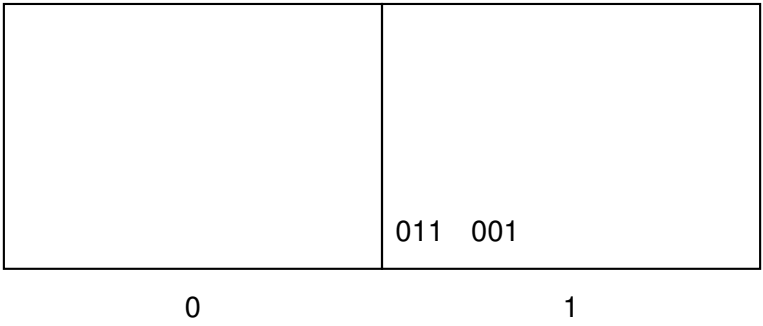
011	001	100	010	111	101	000	011	110	101

Radixsort — Funktionsweise



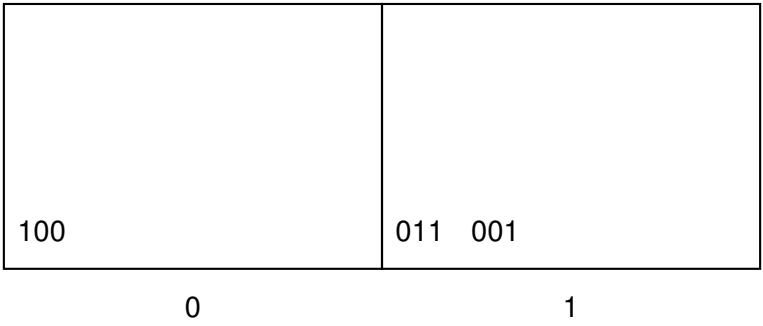
011	001	100	010	111	101	000	011	110	101

Radixsort — Funktionsweise



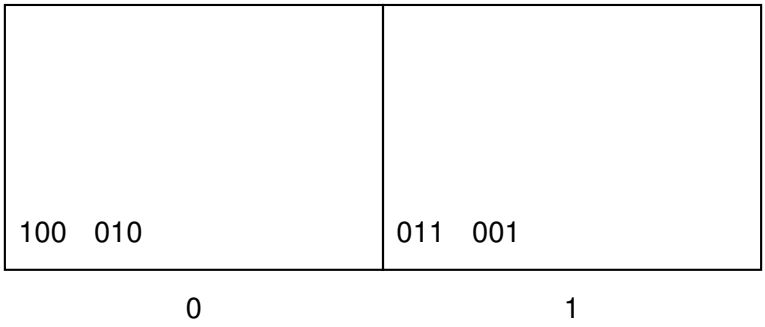
011	001	100	010	111	101	000	011	110	101

Radixsort — Funktionsweise



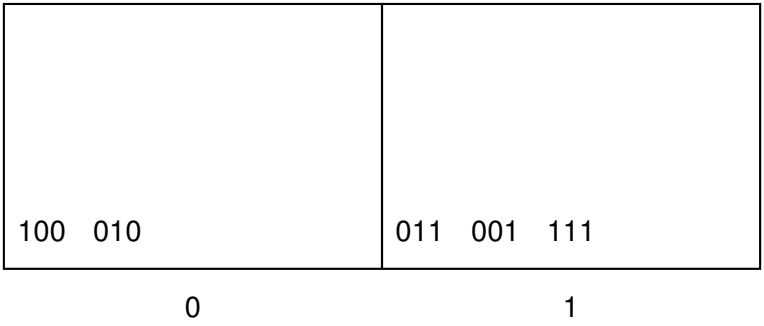
011	001	100	010	111	101	000	011	110	101

Radixsort — Funktionsweise



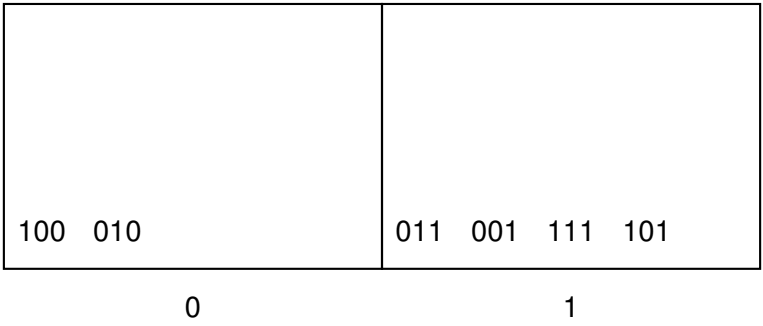
011	001	100	010	111	101	000	011	110	101

Radixsort — Funktionsweise



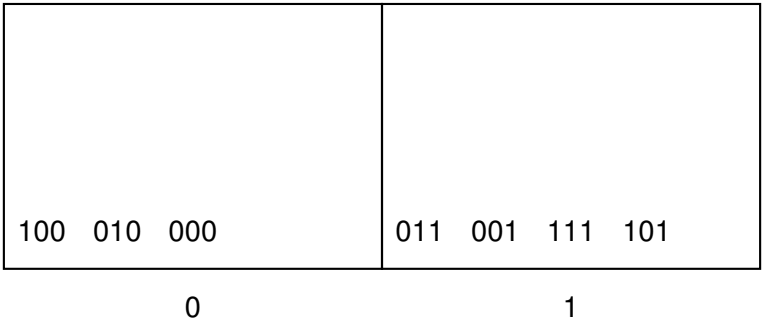
011	001	100	010	111	101	000	011	110	101

Radixsort — Funktionsweise



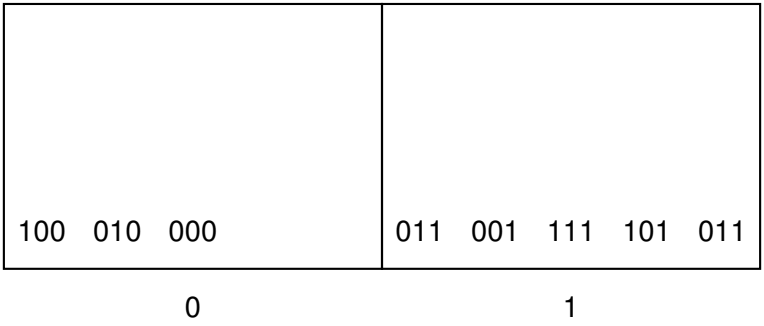
011	001	100	010	111	101	000	011	110	101

Radixsort — Funktionsweise



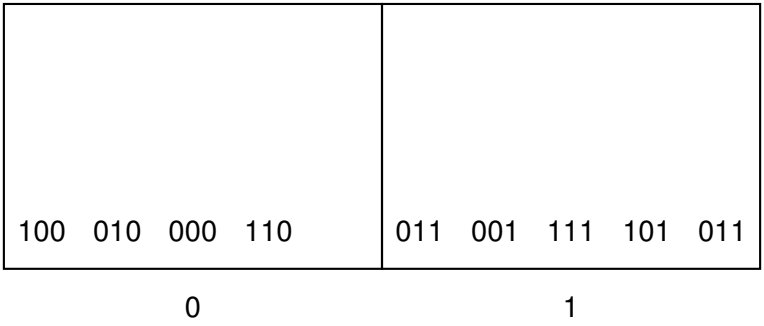
011	001	100	010	111	101	000	011	110	101

Radixsort — Funktionsweise



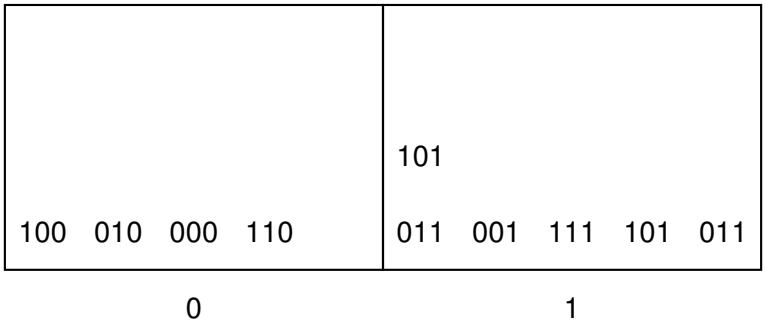
011	001	100	010	111	101	000	011	110	101

Radixsort — Funktionsweise



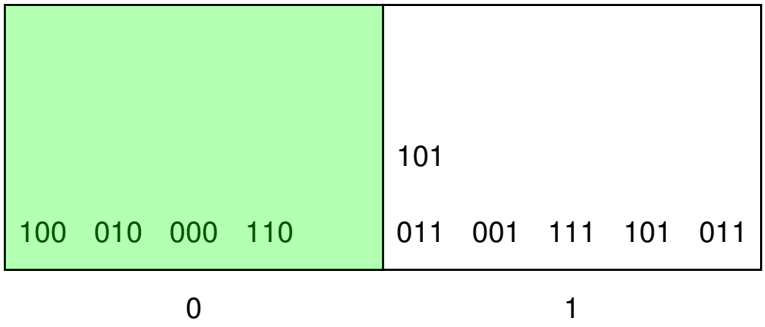
011	001	100	010	111	101	000	011	110	101

Radixsort — Funktionsweise



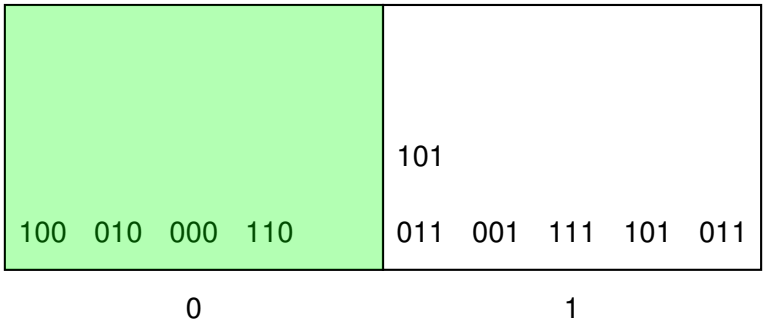
011	001	100	010	111	101	000	011	110	101

Radixsort — Funktionsweise



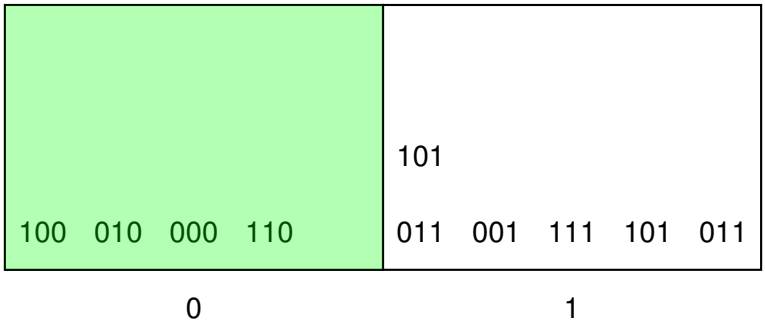
011	001	100	010	111	101	000	011	110	101

Radixsort — Funktionsweise



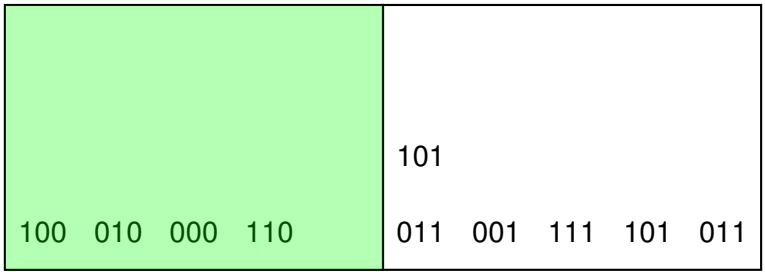
100	001	100	010	111	101	000	011	110	101

Radixsort — Funktionsweise



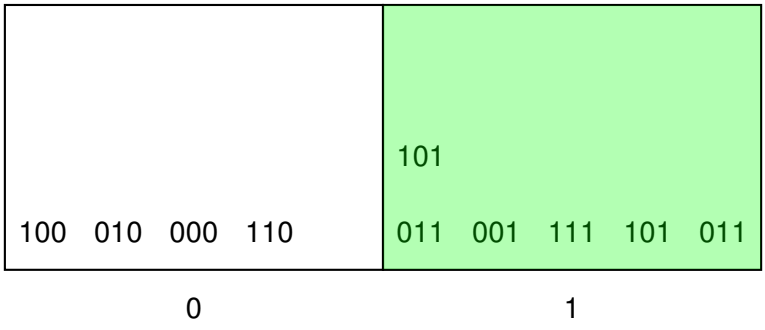
100	010	100	010	111	101	000	011	110	101

Radixsort — Funktionsweise



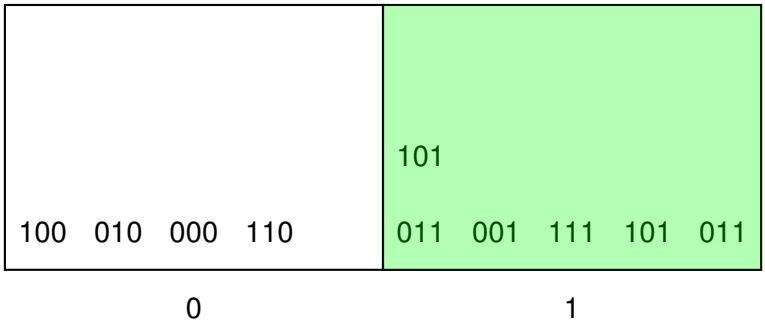
100	010	000	010	111	101	000	011	110	101

Radixsort — Funktionsweise



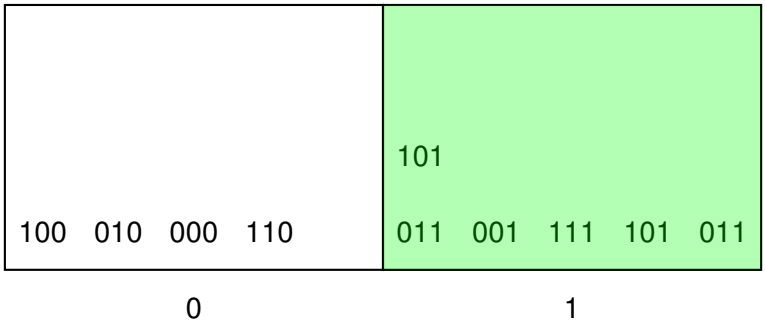
100	010	000	110	111	101	000	011	110	101

Radixsort — Funktionsweise



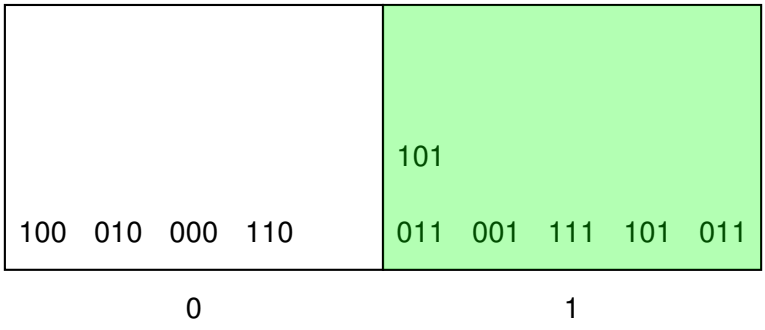
100	010	000	110	011	101	000	011	110	101

Radixsort — Funktionsweise



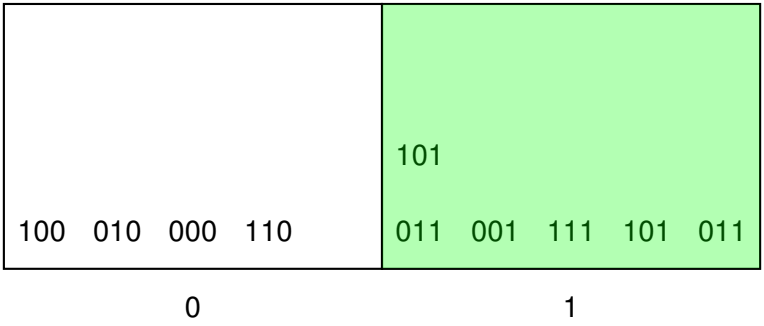
100	010	000	110	011	001	000	011	110	101

Radixsort — Funktionsweise



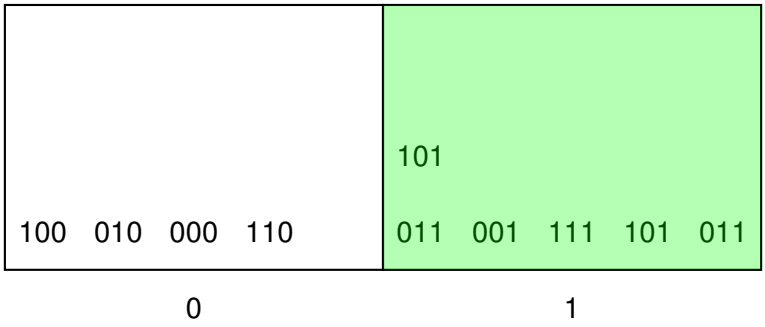
100	010	000	110	011	001	111	011	110	101

Radixsort — Funktionsweise



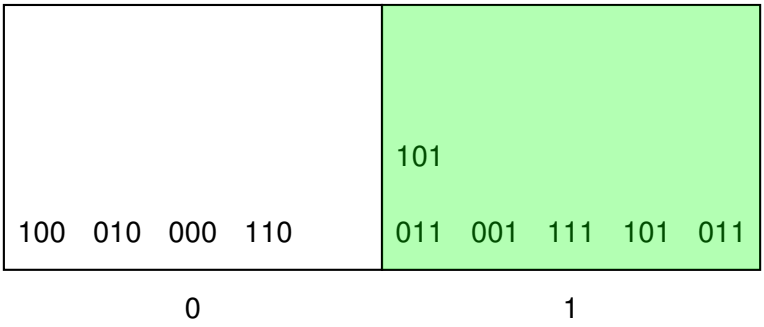
100	010	000	110	011	001	111	101	110	101

Radixsort — Funktionsweise



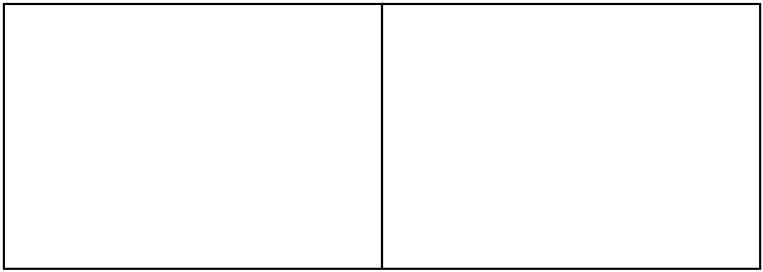
100	010	000	110	011	001	111	101	011	101

Radixsort — Funktionsweise



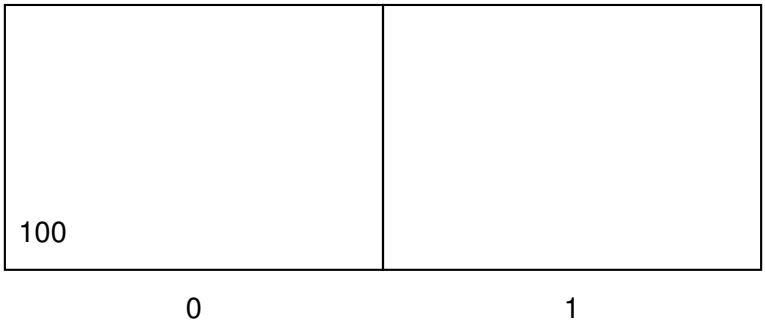
100	010	000	110	011	001	111	101	011	101

Radixsort — Funktionsweise



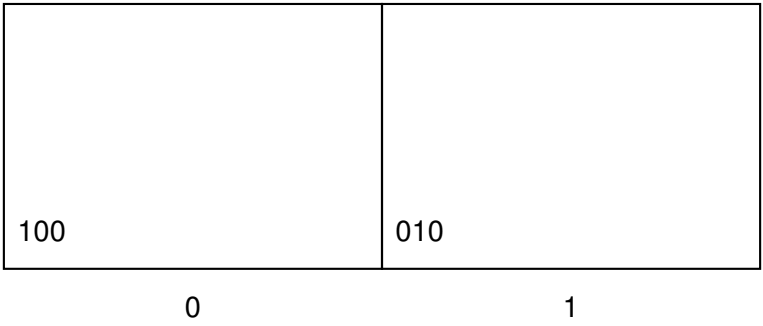
100	010	000	110	011	001	111	101	011	101

Radixsort — Funktionsweise



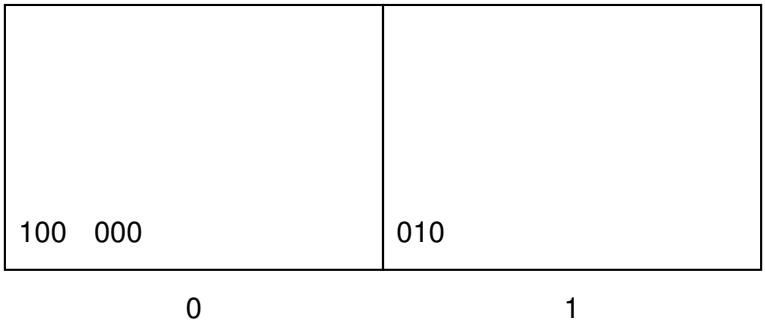
100	010	000	110	011	001	111	101	011	101

Radixsort — Funktionsweise



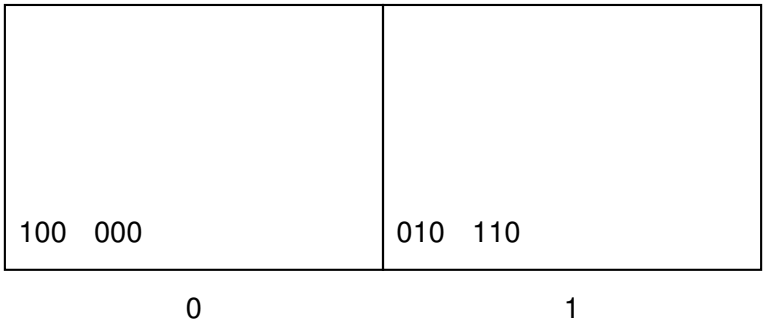
100	010	000	110	011	001	111	101	011	101

Radixsort — Funktionsweise



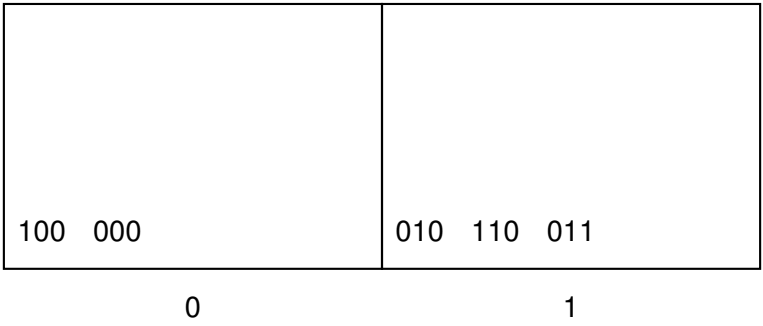
100	010	000	110	011	001	111	101	011	101

Radixsort — Funktionsweise



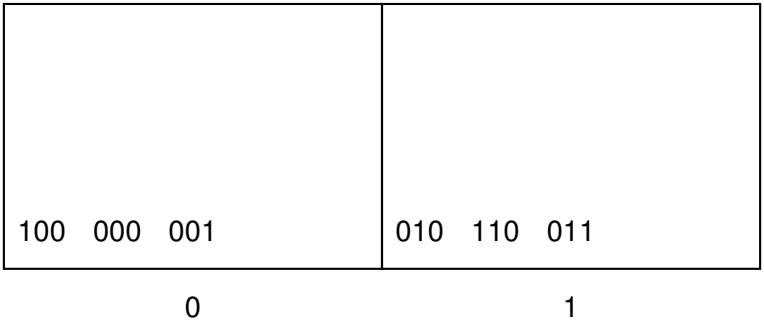
100	010	000	110	011	001	111	101	011	101

Radixsort — Funktionsweise



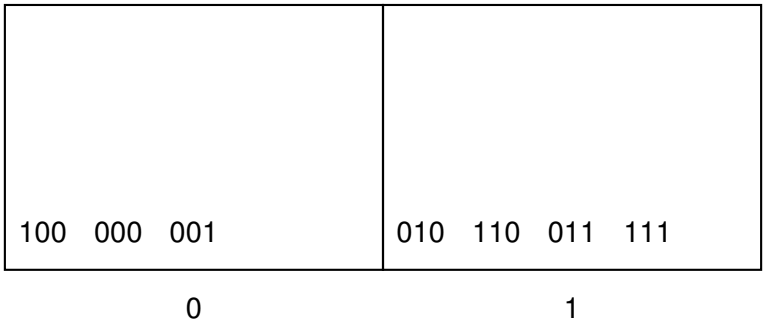
100	010	000	110	011	001	111	101	011	101

Radixsort — Funktionsweise



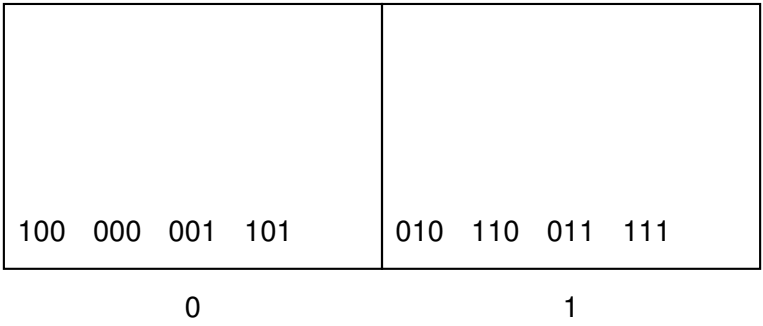
100	010	000	110	011	001	111	101	011	101

Radixsort — Funktionsweise



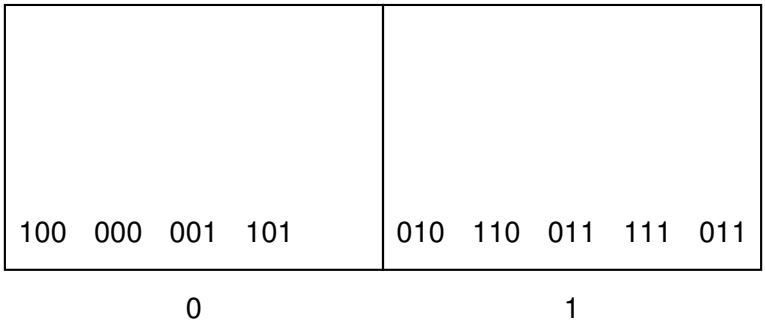
100	010	000	110	011	001	111	101	011	101

Radixsort — Funktionsweise



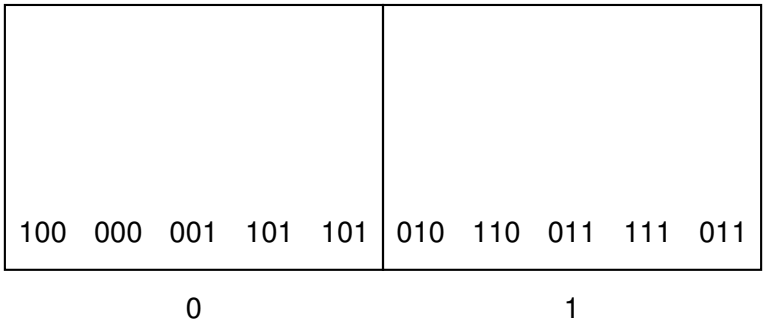
100	010	000	110	011	001	111	101	011	101

Radixsort — Funktionsweise



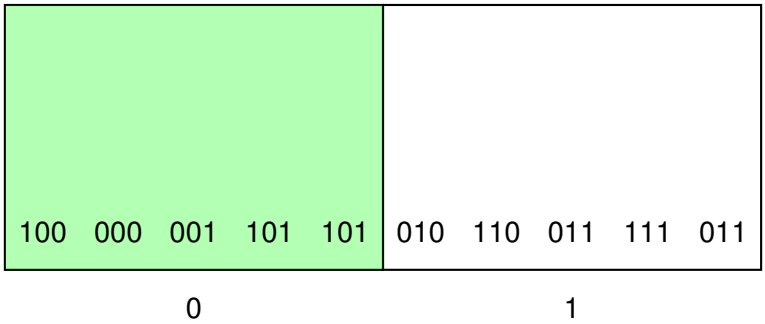
100	010	000	110	011	001	111	101	011	101

Radixsort — Funktionsweise



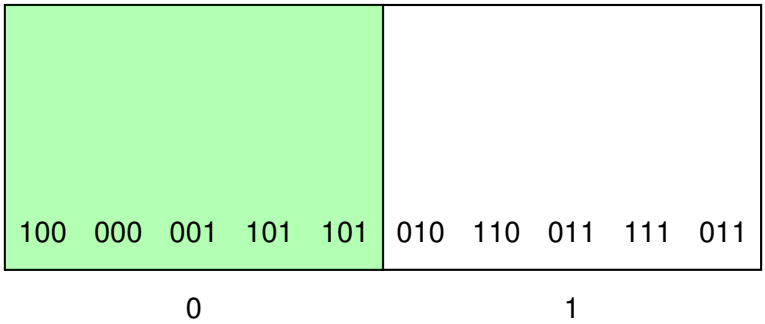
100	010	000	110	011	001	111	101	011	101

Radixsort — Funktionsweise



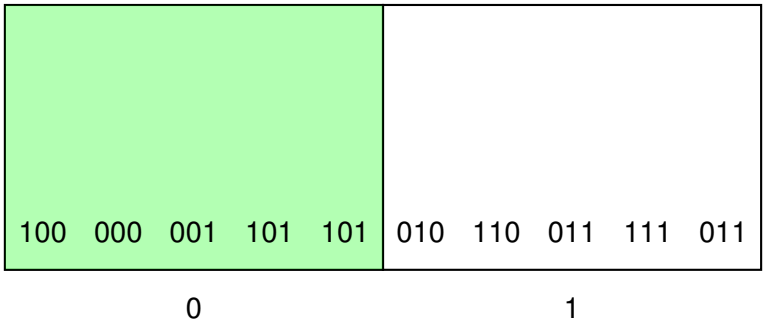
100	010	000	110	011	001	111	101	011	101

Radixsort — Funktionsweise



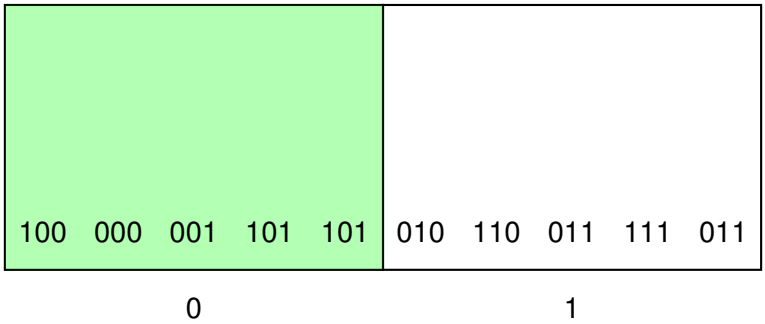
100	010	000	110	011	001	111	101	011	101

Radixsort — Funktionsweise



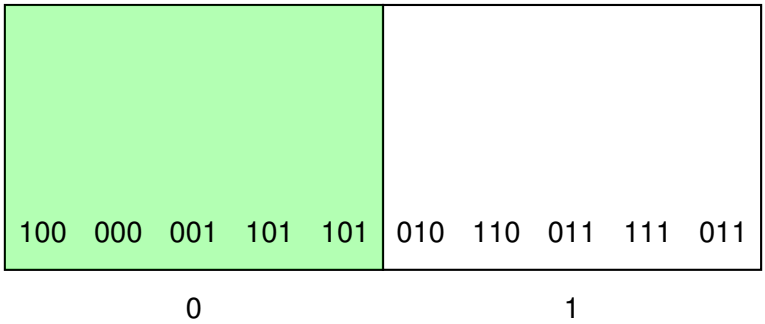
100	000	000	110	011	001	111	101	011	101

Radixsort — Funktionsweise



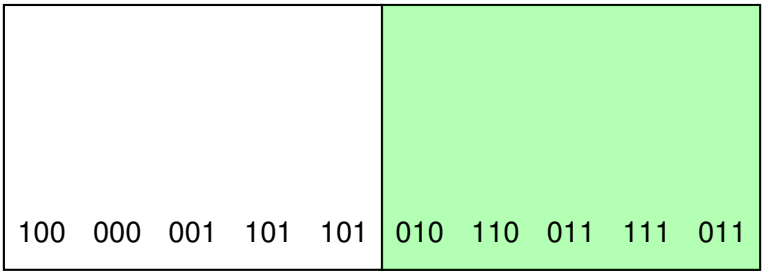
100	000	001	110	011	001	111	101	011	101

Radixsort — Funktionsweise



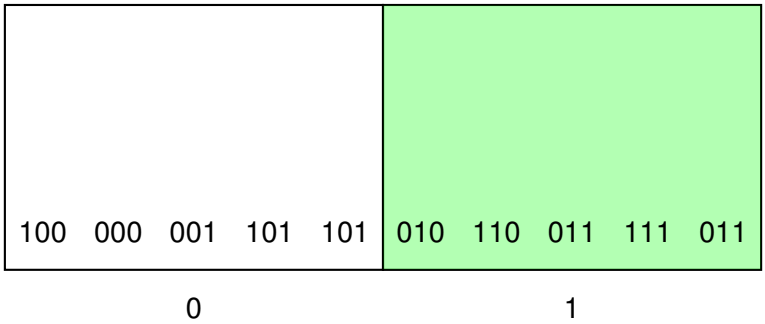
100	000	001	101	011	001	111	101	011	101

Radixsort — Funktionsweise



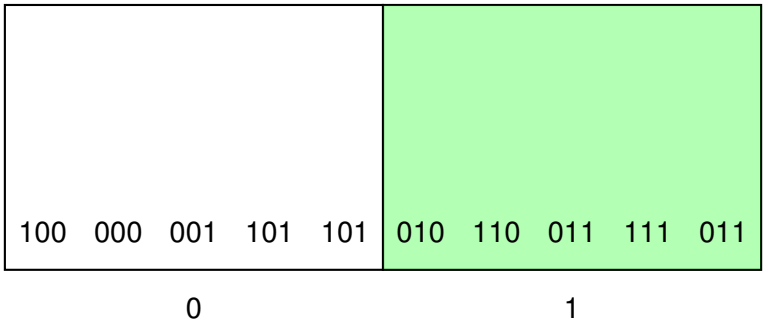
100	000	001	101	101	001	111	101	011	101

Radixsort — Funktionsweise



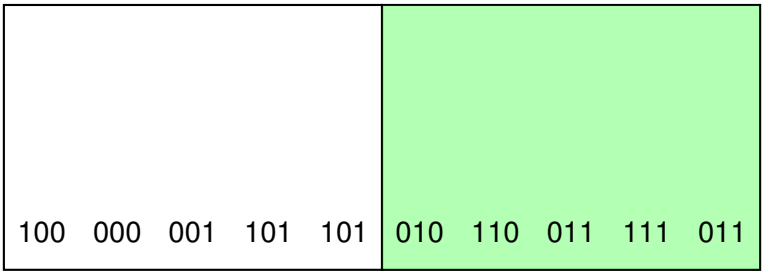
100	000	001	101	101	010	111	101	011	101

Radixsort — Funktionsweise



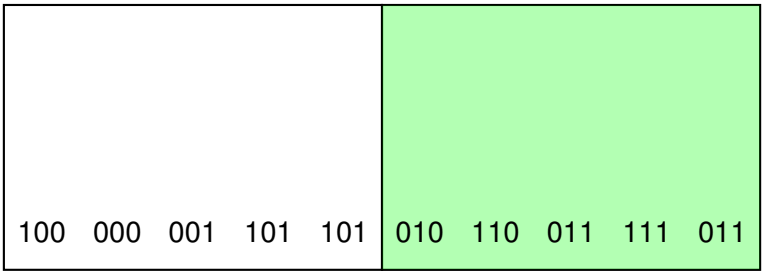
100	000	001	101	101	010	110	101	011	101

Radixsort — Funktionsweise



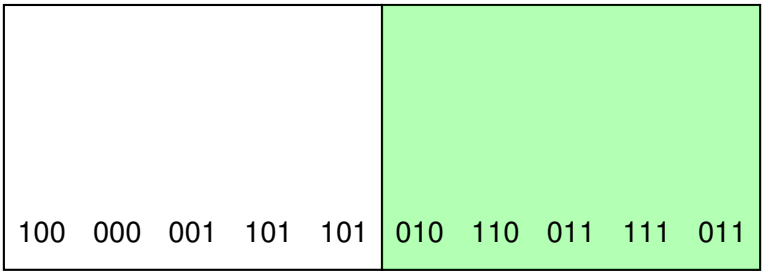
100	000	001	101	101	010	110	011	011	101

Radixsort — Funktionsweise



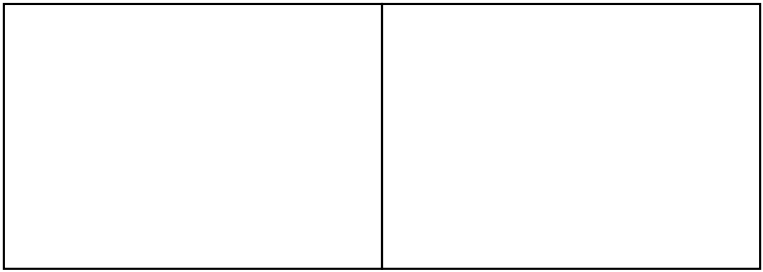
100	000	001	101	101	010	110	011	111	101

Radixsort — Funktionsweise



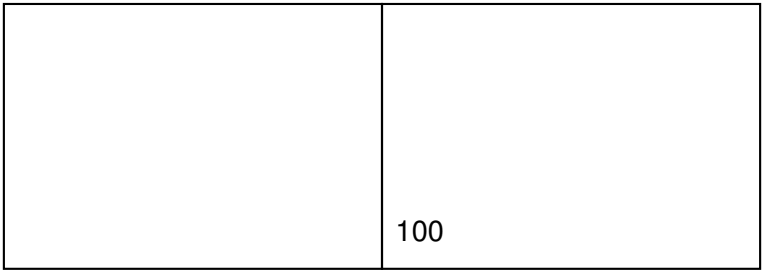
100	000	001	101	101	010	110	011	111	011

Radixsort — Funktionsweise



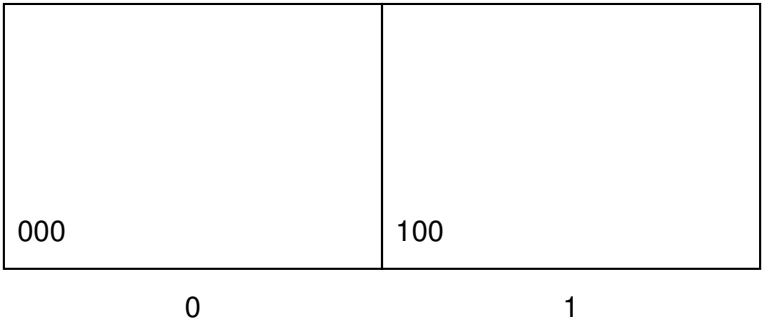
100	000	001	101	101	010	110	011	111	011

Radixsort — Funktionsweise



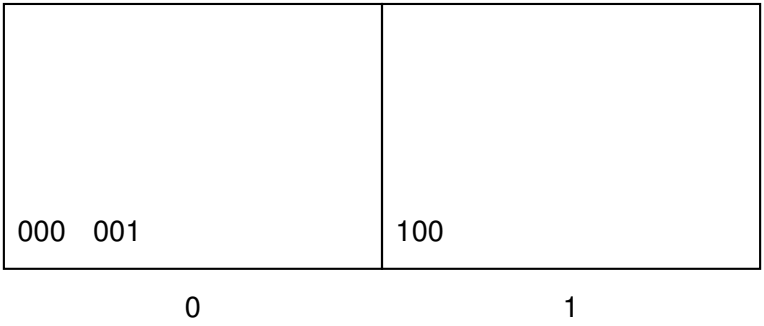
100	000	001	101	101	010	110	011	111	011

Radixsort — Funktionsweise



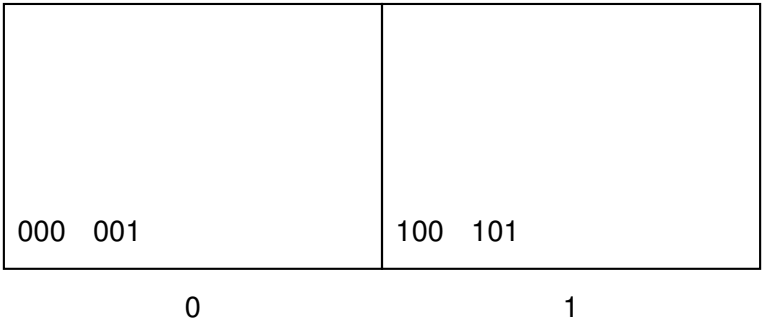
100	000	001	101	101	010	110	011	111	011

Radixsort — Funktionsweise



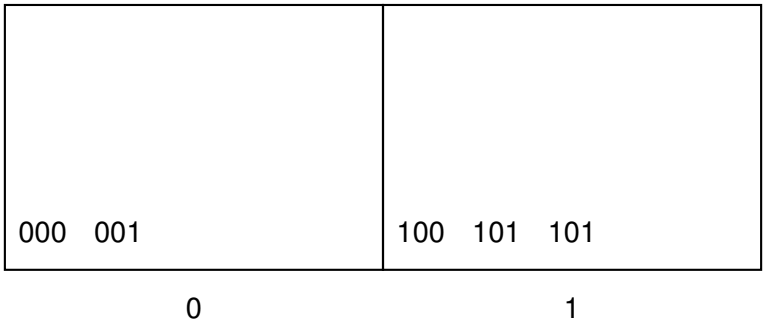
100	000	001	101	101	010	110	011	111	011

Radixsort — Funktionsweise



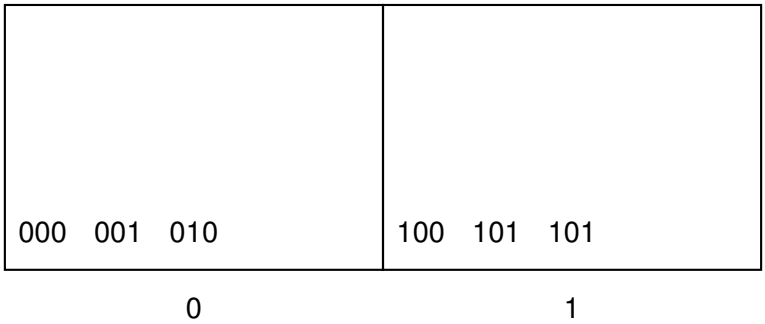
100	000	001	101	101	010	110	011	111	011

Radixsort — Funktionsweise



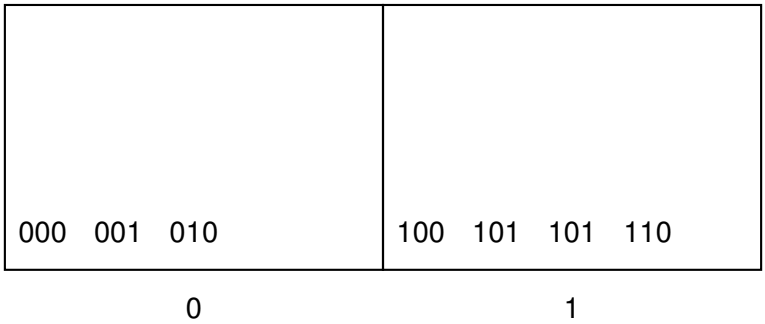
100	000	001	101	101	010	110	011	111	011

Radixsort — Funktionsweise



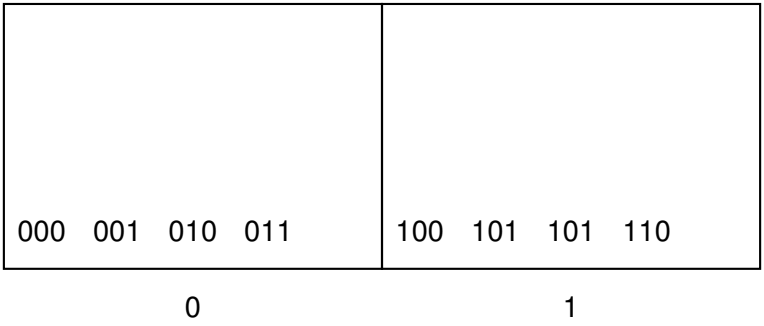
100	000	001	101	101	010	110	011	111	011

Radixsort — Funktionsweise



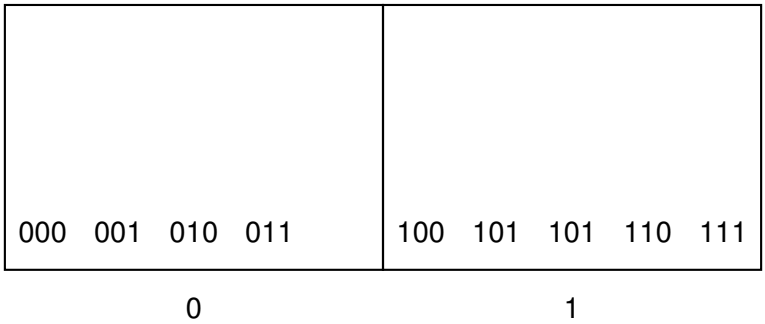
100	000	001	101	101	010	110	011	111	011

Radixsort — Funktionsweise



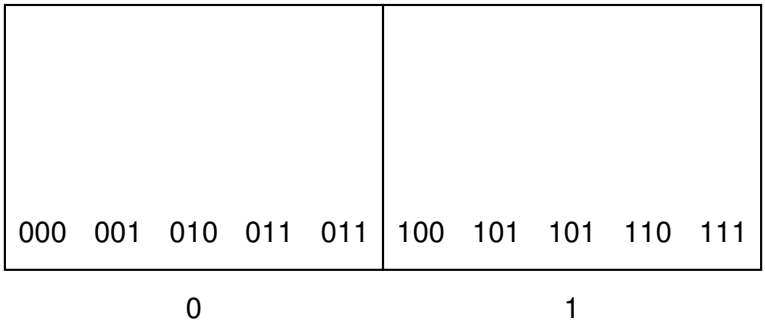
100	000	001	101	101	010	110	011	111	011

Radixsort — Funktionsweise



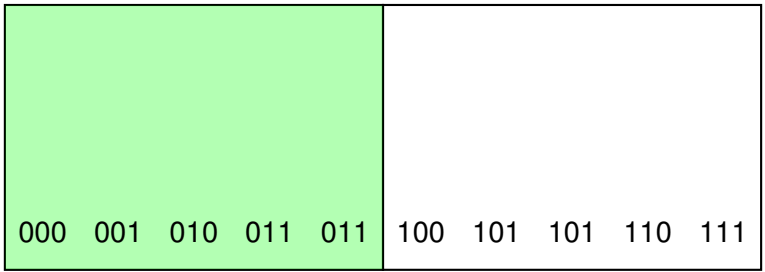
100	000	001	101	101	010	110	011	111	011

Radixsort — Funktionsweise



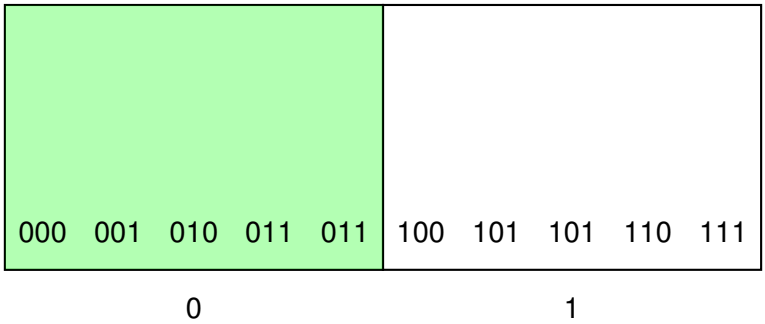
100	000	001	101	101	010	110	011	111	011

Radixsort — Funktionsweise



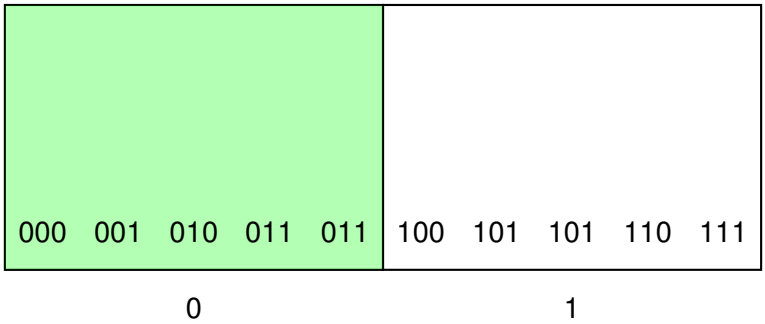
100	000	001	101	101	010	110	011	111	011

Radixsort — Funktionsweise



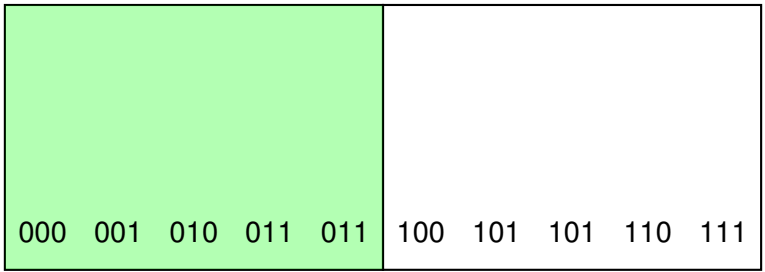
000	000	001	101	101	010	110	011	111	011

Radixsort — Funktionsweise



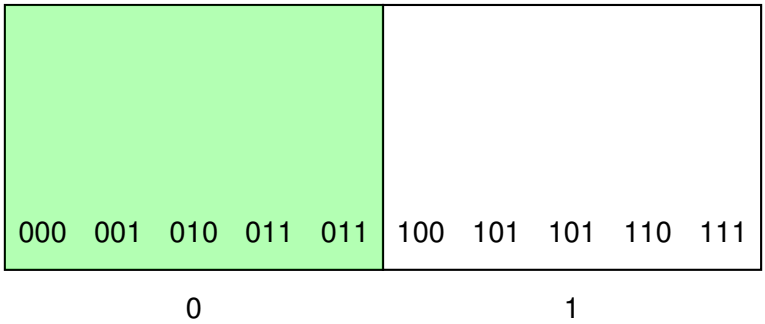
000	001	001	101	101	010	110	011	111	011

Radixsort — Funktionsweise



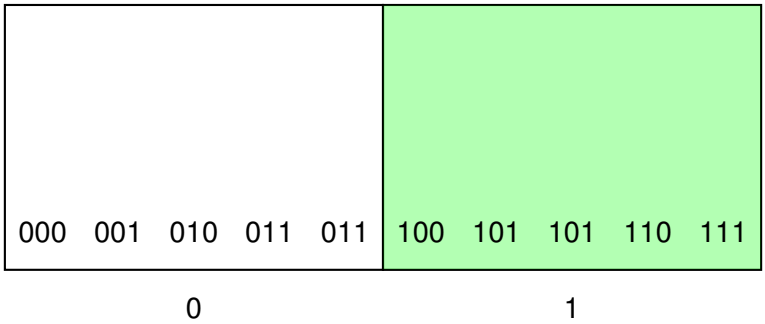
000	001	010	101	101	010	110	011	111	011

Radixsort — Funktionsweise



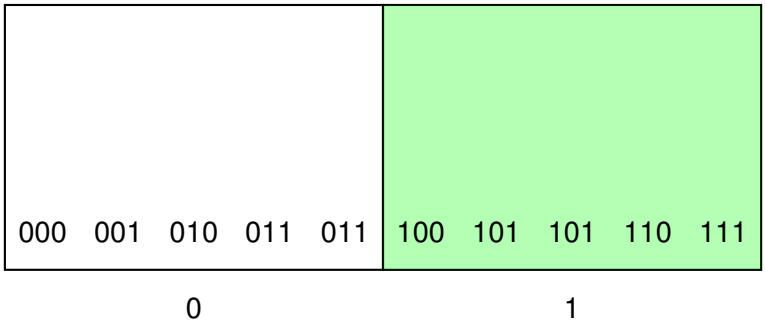
000	001	010	011	101	010	110	011	111	011

Radixsort — Funktionsweise



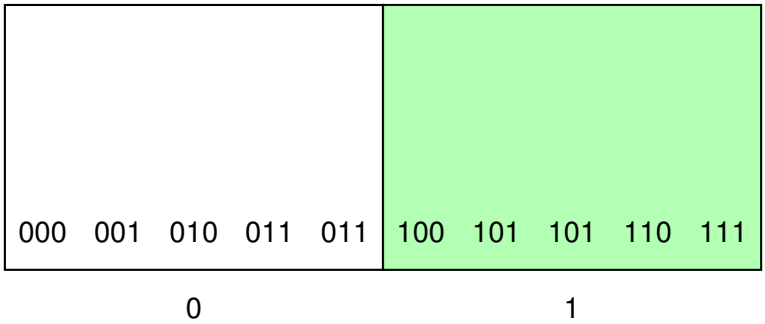
000	001	010	011	011	010	110	011	111	011

Radixsort — Funktionsweise



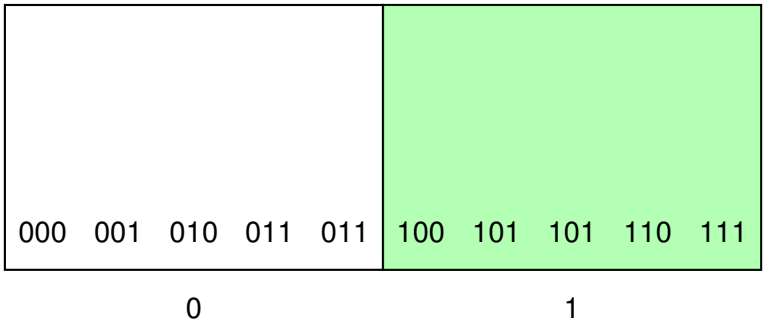
000	001	010	011	011	100	110	011	111	011

Radixsort — Funktionsweise



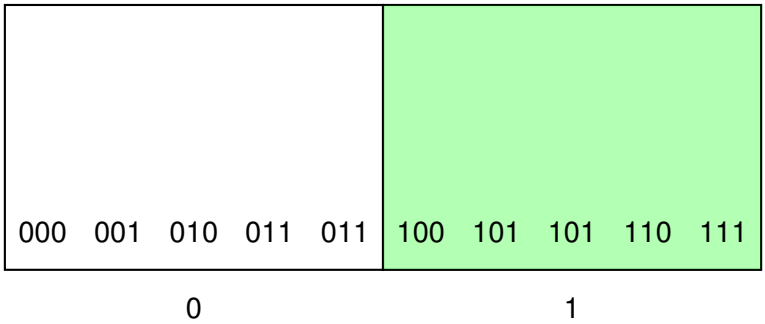
000	001	010	011	011	100	101	011	111	011

Radixsort — Funktionsweise



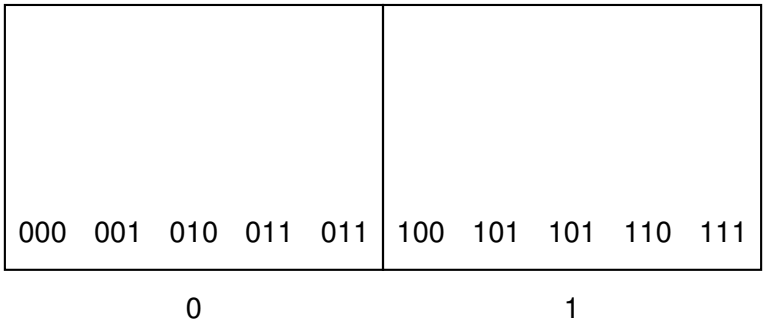
000	001	010	011	011	100	101	101	111	011

Radixsort — Funktionsweise



000	001	010	011	011	100	101	101	110	011

Radixsort — Funktionsweise



000	001	010	011	011	100	101	101	110	111

Radixsort — Code (1)

```
int[] radixSort(int[] array) {
    int[][] buckets = int[2];
    for(int digit = 1; digit < 2**31 && digit > 0; digit = digit * 2)
        {
        for(int pos = 0; pos < array.length; pos++) {
            int value = (array[pos] / digit) % 2;
            buckets[value].add(array[pos]);
        }
        arrayIndex = 0;
        for(int i = 0; i < buckets[0].length; i++) {
            array[arrayIndex++] = buckets[0][i];
        }
        buckets[0].clear();
```

Radixsort — Code (2)

```
        for(int i = 0; i < buckets[1].length; i++) {
            array[arrayIndex++] = buckets[1][i];
        }
        buckets[1].clear();
    }
    return array;
}
```

Radixsort — Steckbrief

Laufzeitkomplexität (best case)	$\mathcal{O}(n)$
Laufzeitkomplexität (avarage case)	$\mathcal{O}(n)$
Laufzeitkomplexität (worst case)	$\mathcal{O}(n)$
Speicherkomplexität	$\mathcal{O}(n)$
Stabilität	Stabil
Vergleichsbasiertes Verfahren	Nein

Quellen

- Vorlesungsunterlagen “Algorithmen und Datenstrukturen”