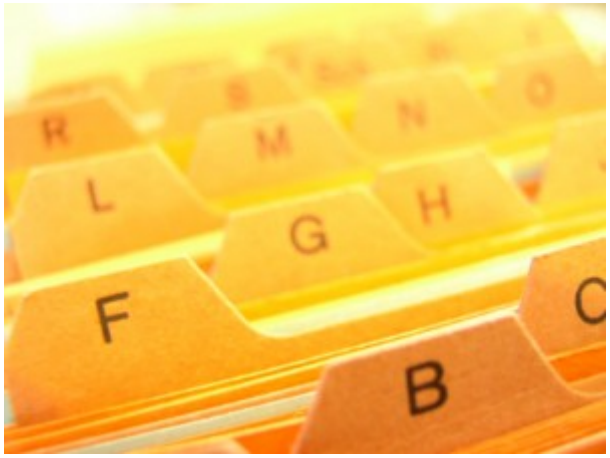
Sortieralgorithmen

Gliederung

1. Einleitung
2. Grundlagen
3. Sortieralgorithmen
4. Evaluation
5. Fazit

1. Einleitung



Quelle: http://office-lernen.com/excel-tabelle-sortieren/

- Ordnen von großen Dateiverwaltungssystemen
- Schnelleres Finden von gesuchten Objekten
- Weiterentwicklung der Algorithmen mit immer verkürzter Laufzeit
- *Frage:* Welche Unterschiede herrschen zwischen den Sortieralgorithmen?

2. Grundlagen

O-Notation:

- Berechnung einer Laufzeit anhand der gegebenen Eingabelänge n
- Definiert ein ungefähres Wachstumsverhalten des Algorithmus als mathematische Funktion

2. Grundlagen

O-Notation:

- Analyse des Quellcodes und Durchführung für drei Fälle:

- ***Worst-Case***: Sucht nach der maximalen Laufzeit des Algorithmus mit oberer Schranke

- ***Best-Case***: Darstellung der minimalen Laufzeit mit unterer Schranke

- ***Average-Case***: Darstellung der

2. Grundlagen

O-Notation:

- Vorteil: unabhängig von Hardware und Betriebssystem
- Nachteil: grobe Einschätzung einer möglichen Laufzeit

2. Grundlagen

In-Place:

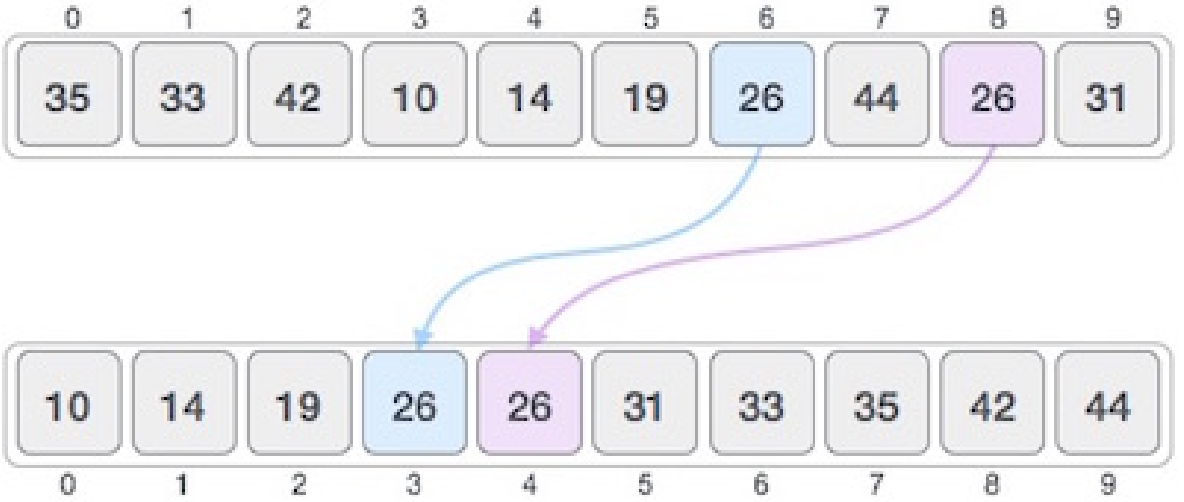
- Bekanntgebung, ob weiterer Hilfsspeicher für den Tauschvorgang benötigt wird
- Bezeichnung von Sortieralgorithmen als In-Place, wenn sie auf dem Container arbeiten und nur einen zusätzlichen Zwischenspeicher benötigen

Quelle: https://www.icpdr.org/main/publications/danube-countries-put-pieces-place-river-basin-management

2. Grundlagen

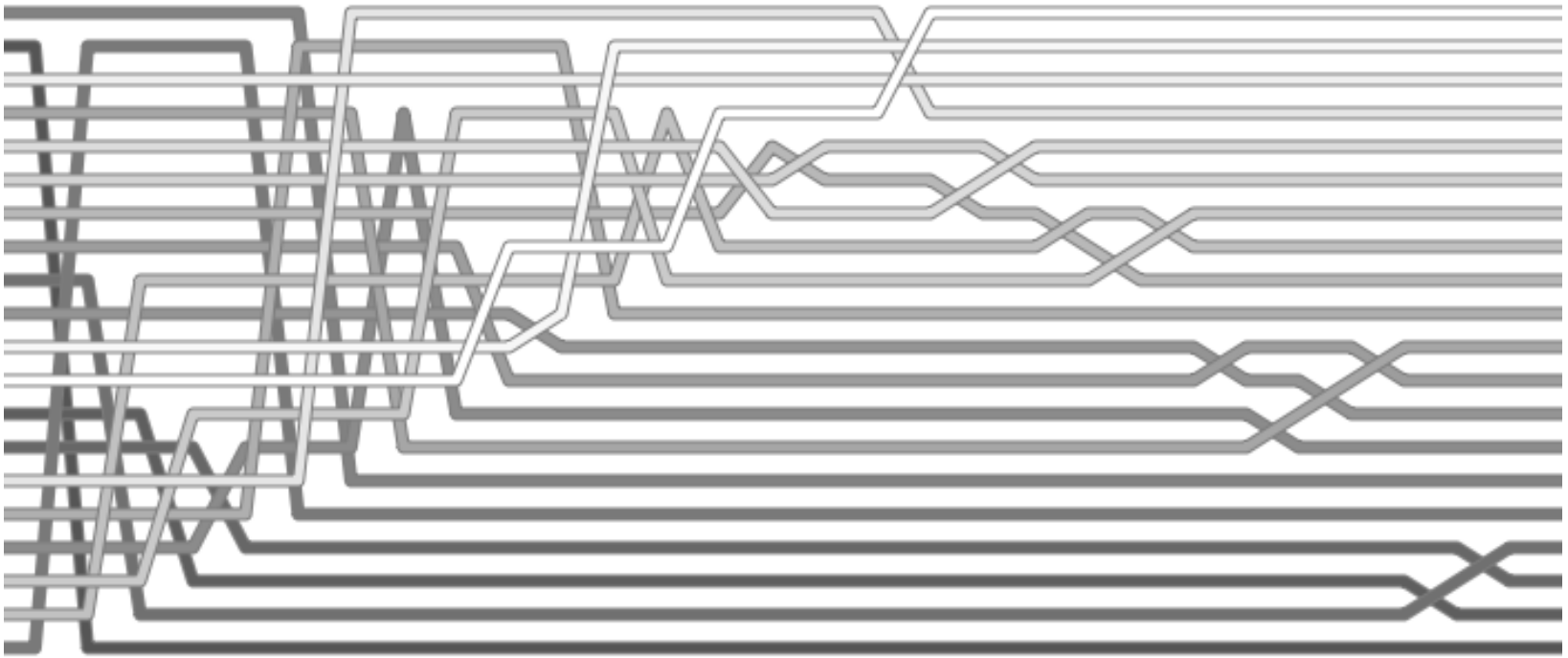
Stabilität:

Beibehaltung der Reihenfolge von äquivalenten Elementen



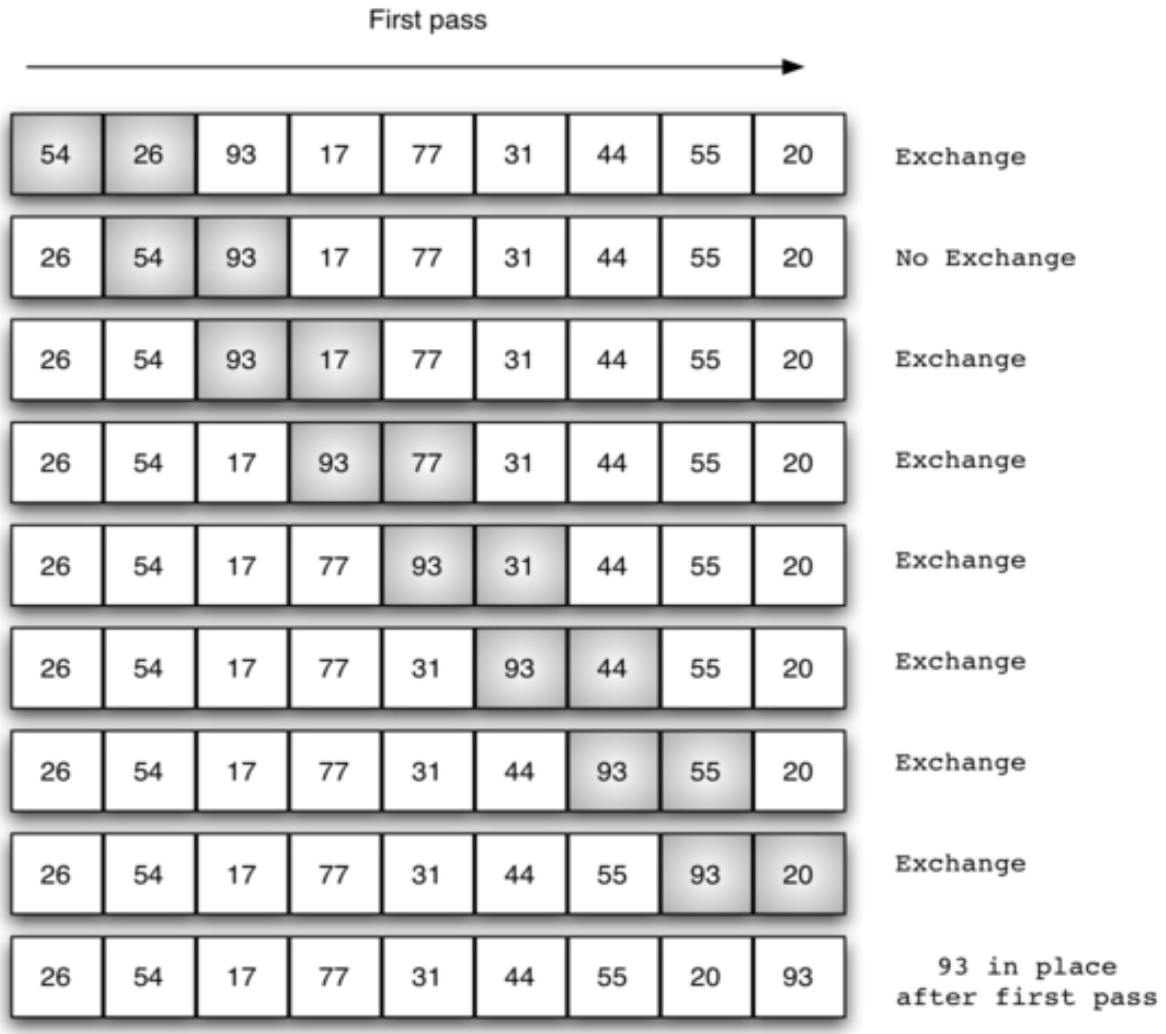
Quelle: https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm

3. Sortieralgorithmen



Quelle: http://www.thelowlyprogrammer.com/2010/04/introducing-marriage-sort.html

3. Sortieralgorithmen: **BubbleSort**



Quelle: http://interactivepython.org/runestone/static/pythonds/SortSearch/Thel

3. Sortieralgorithmen: **BubbleSort**

Best-Case

- O(n)
- bereits sortierte Liste
- geht die Liste nur einmal durch

Worst-Case

- $O(n^2)$
- verkehrt herum sortierte Liste
- geht beide Schleifen ganz durch

Quelle: http://www.algolist.net/Algorithms/Sorting/Bubble_sort

3. Sortieralgorithmen: **BubbleSort**

Stabilität & In-Place:

- stabiler Algorithmus
- Nur benachbarte Elemente werden getauscht
- In-Place
- arbeitet mit Hilfsspeicher O(1)

3. Sortieralgorithmen: **InsertionSort**

Quelle: http://interactivepython.org/runestone/static/pythonds/SortSearch/The

3. Sortieralgorithmen: **InsertionSort**

Best-Case

- O(n)
- bereits sortierte Liste
- keine Verschiebungen

Worst-Case

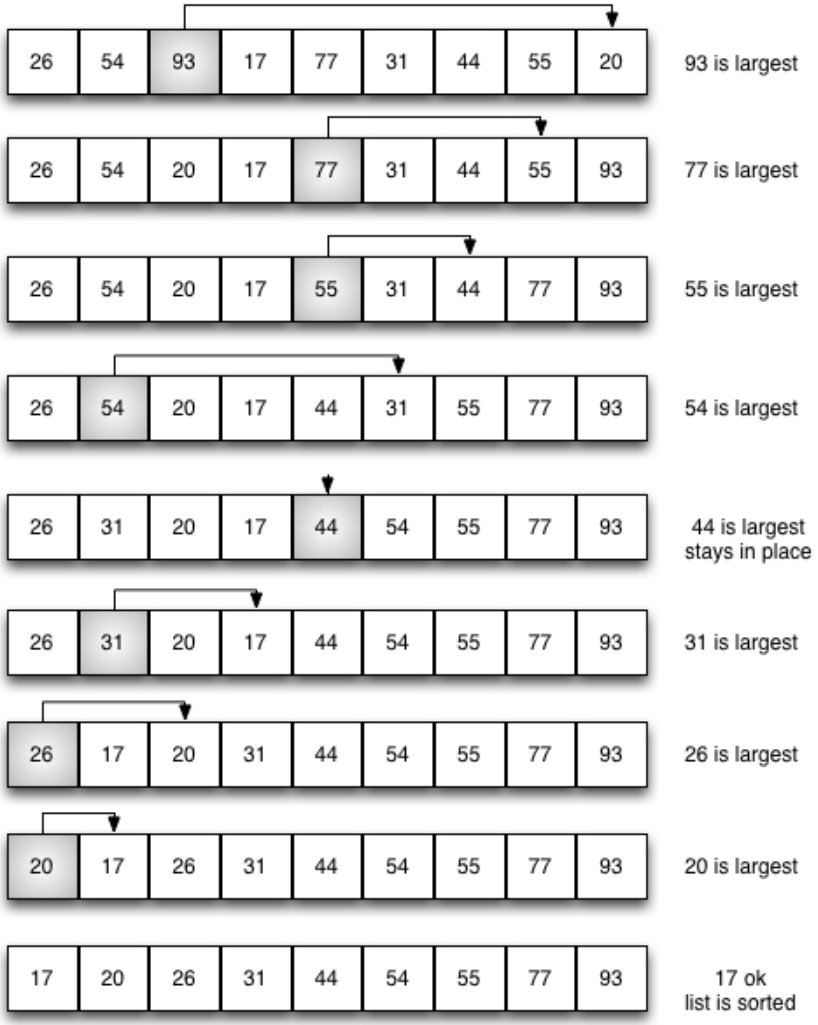
- $O(n^2)$
- verkehrt herum sortierte Liste
- geht beide Schleifen ganz durch

3. Sortieralgorithmen: **InsertionSort**

Stabilität & In-Place:

- Stabilität ist gesorgt, da nur benachbarte Elemente getauscht werden
- In-Place, denn der InsertionSort benötigt nur einen zusätzlichen Speicher, um den Tauschvorgang durchzuführen
- Hilfsspeicher O(1)

3. Sortieralgorithmen: **SelectionSort**



Quelle: http://interactivepython.org/runestone/static/pythonds/SortSearch/TheS

3. Sortieralgorithmen: **SelectionSort**

Best-Case

- $O(n^2)$
- bereits sortierte Liste
- trotzdem wird der Container komplett durchgelaufen

Worst-Case

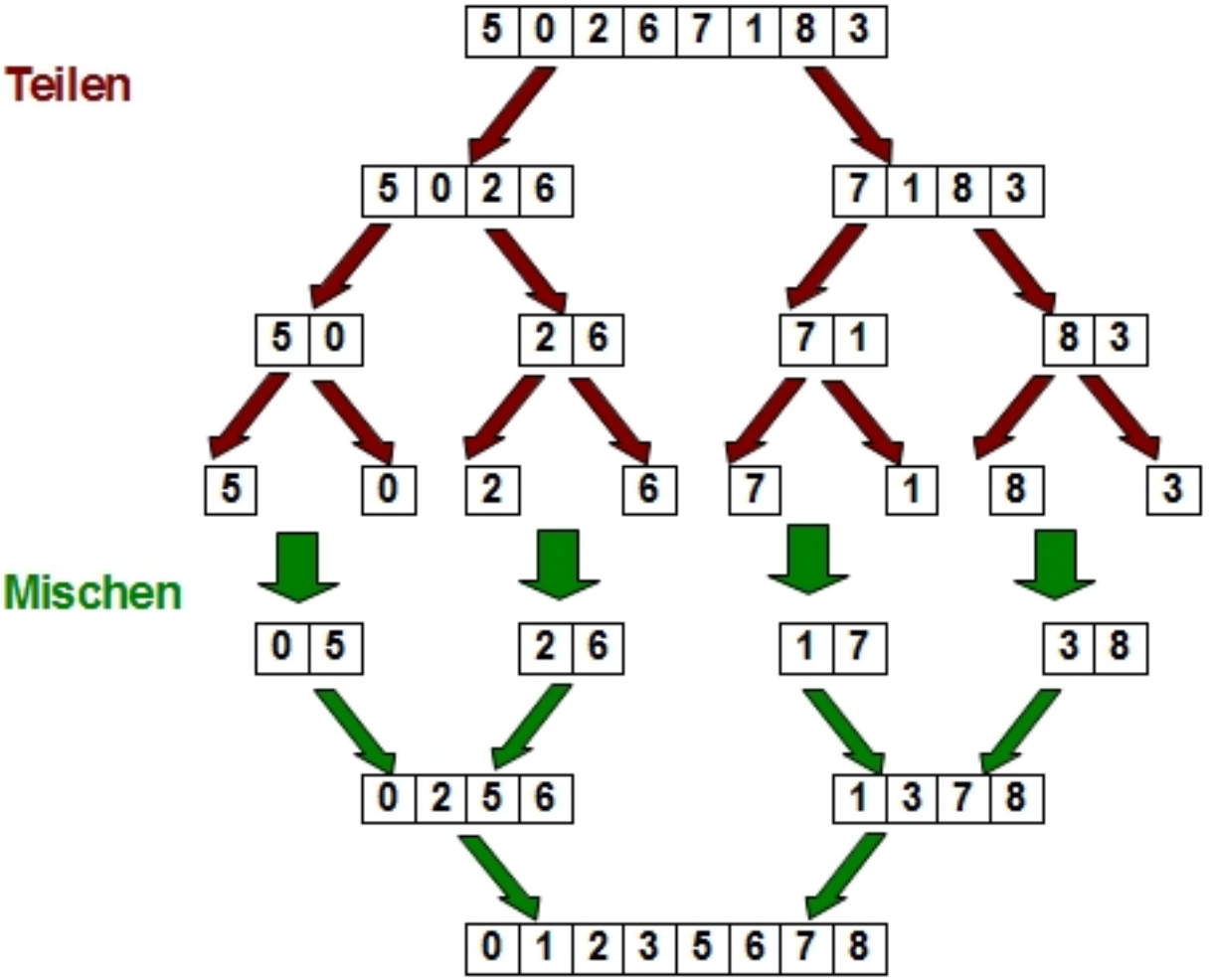
- $O(n^2)$
- ähnlich wie beim Best-Case werden beide Schleifen ganz durchlaufen, nur hier mit dem Tauschvorgang

3. Sortieralgorithmen: **SelectionSort**

Stabilität & In-Place:

- nicht stabil, weil beim Tauschen die Plätze mit dem vordersten Element vertauscht werden
- In-Place
- arbeitet auf dem gegebenen Container mit einem zusätzlichem Tauschspeicher
- benötigt Hilfsspeicher von O(1)

3. Sortieralgorithmen: **MergeSort**

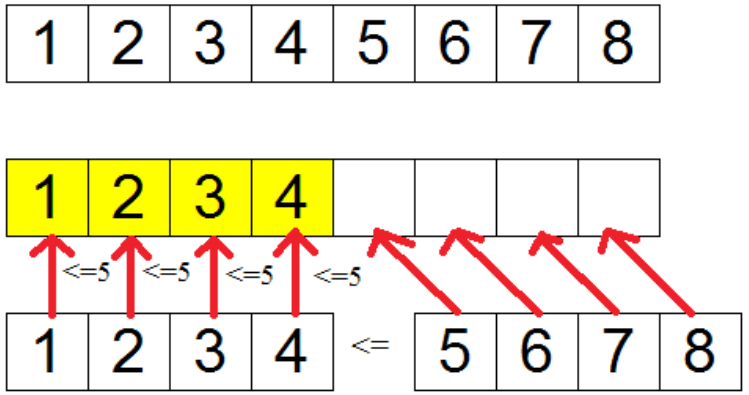


Quelle: http://www.theoretische-informatik.com/mergesort.php

3. Sortieralgorithmen: **MergeSort**

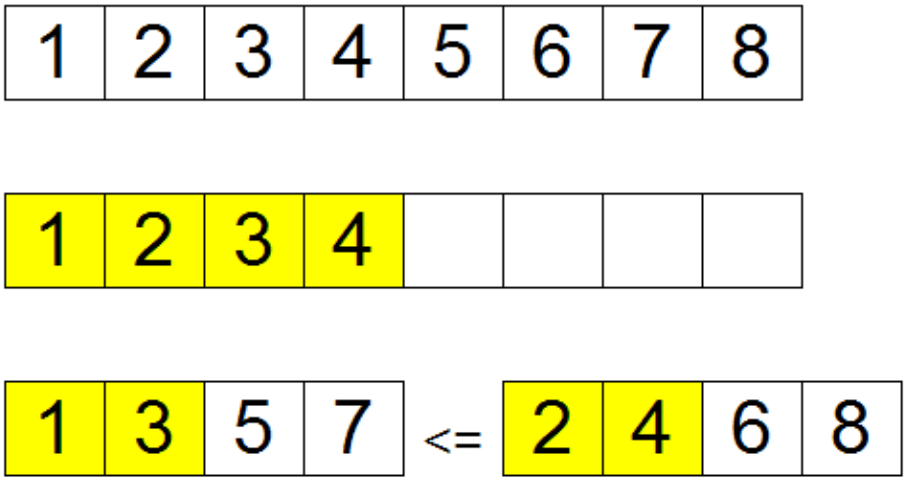
Best-Case

- O(n log n)
- Wenig Vergleiche beim Mergen
- z.B. Sortierte Liste

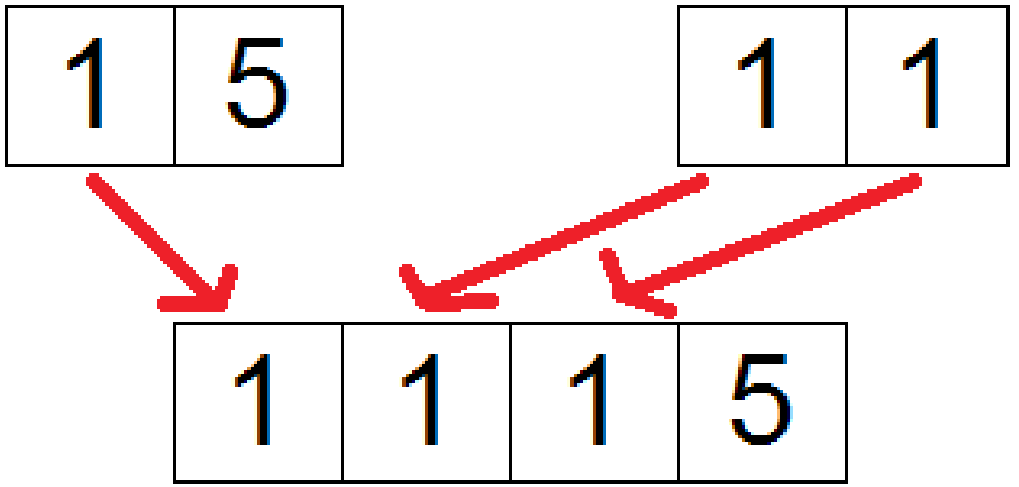


Worst-Case

- O(n log n)
- Maximale vergleiche beim Mergen

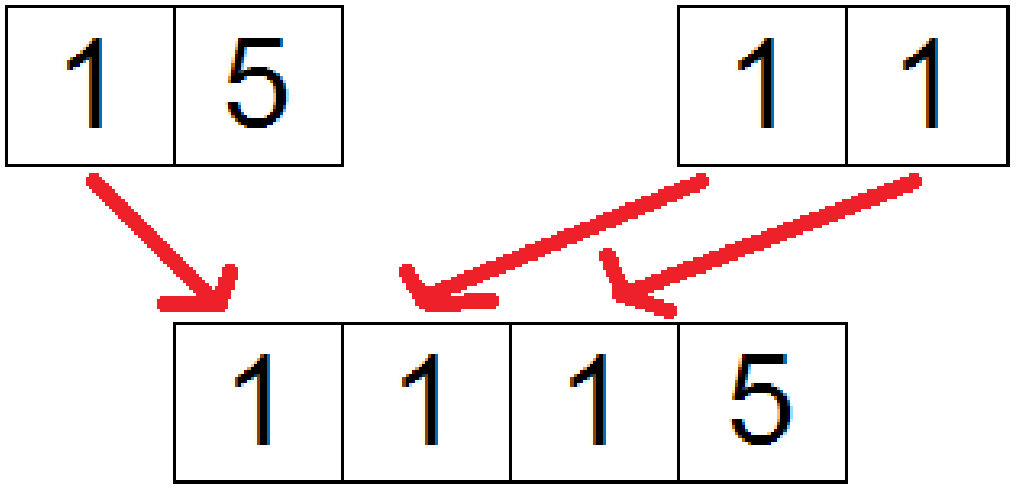


3. Sortieralgorithmen: **MergeSort**



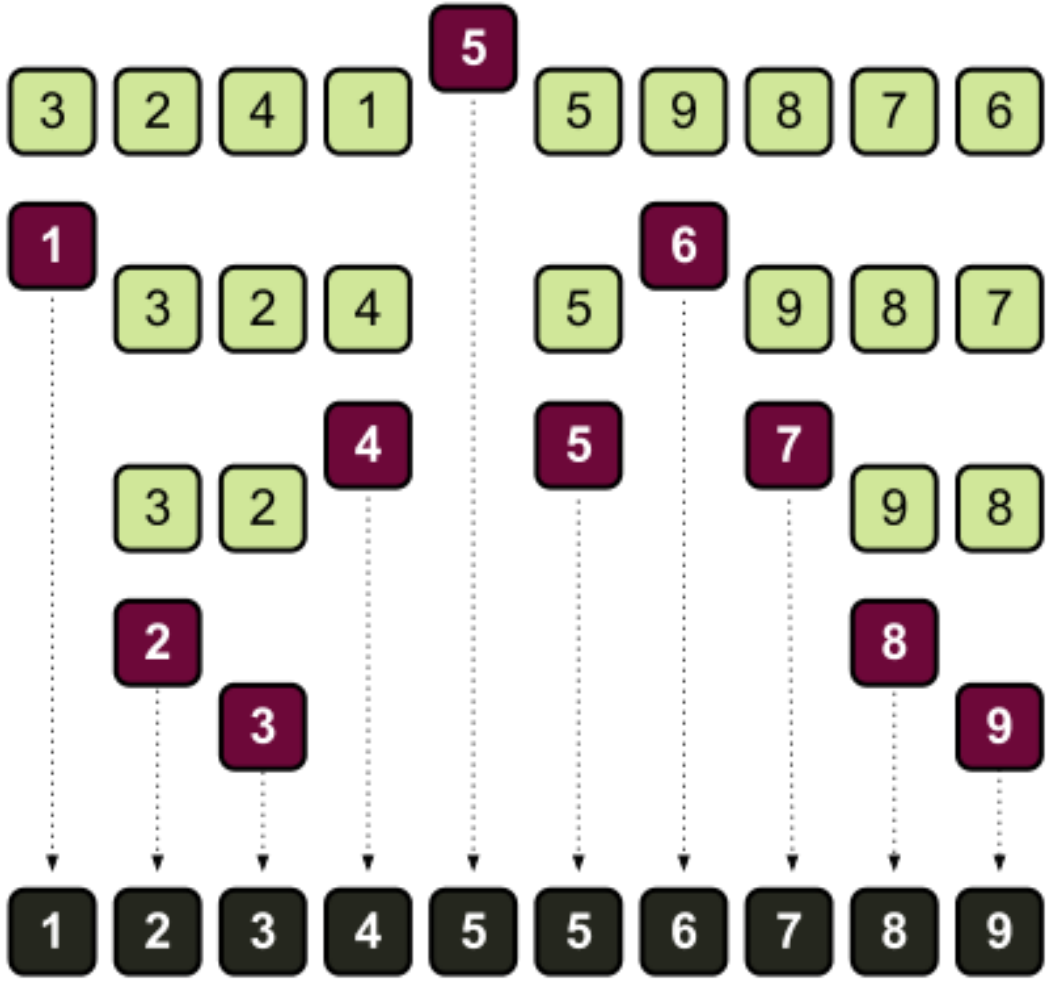
- Stabil: Ja
- In-Place: Nein
- Benötigt zusätzlich O(n) Hilfsspeicher

3. Sortieralgorithmen: **MergeSort**



- Stabil: Ja
- In-Place: Nein
- Benötigt zusätzlich O(n) Hilfsspeicher

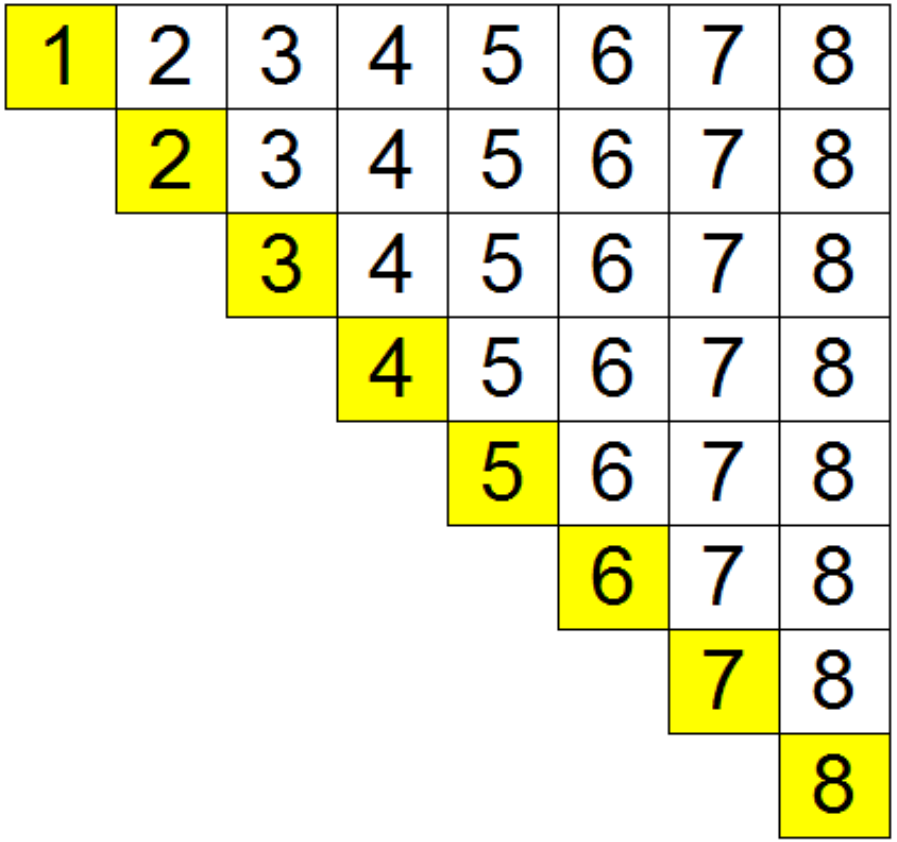
3. Sortieralgorithmen: **QuickSort**



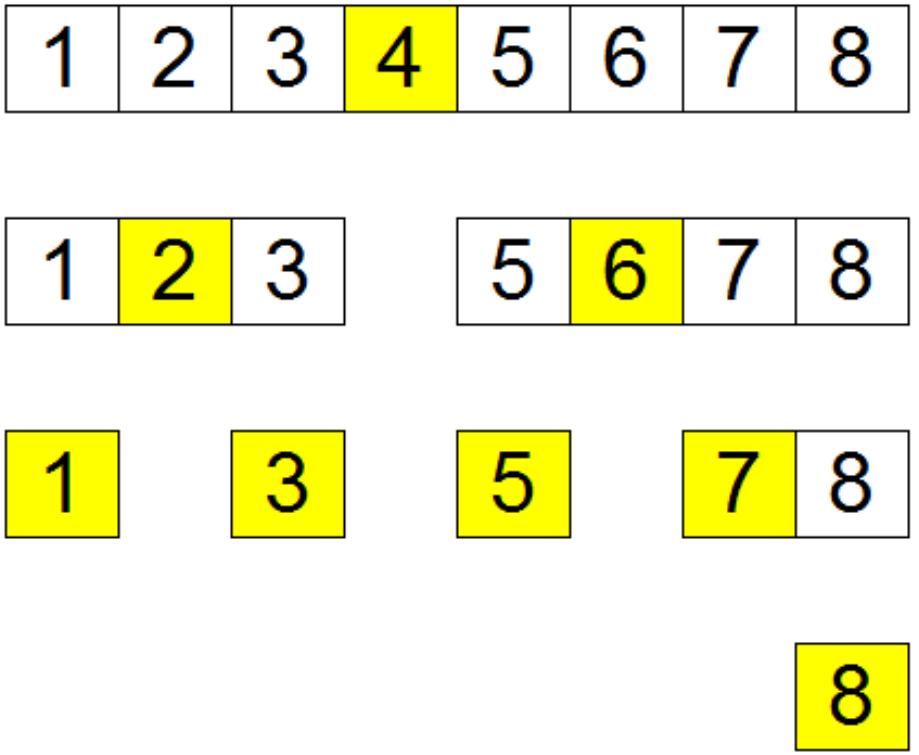
Quelle: http://www.stoimen.com/blog/2012/03/13/computer-algorithms-quicksort/

3. Sortieralgorithmen: **QuickSort**

Worst-Case O(n²)



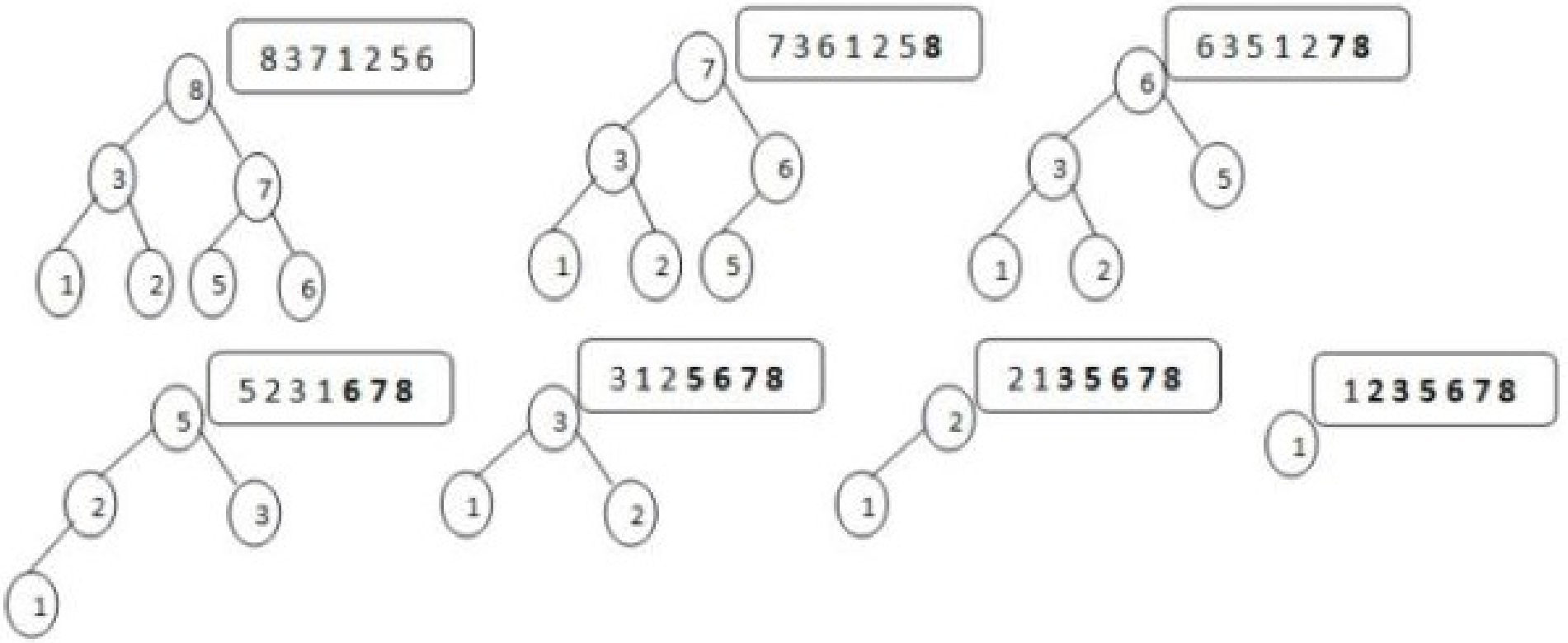
Best-Case O(n log n)



3. Sortieralgorithmen: **QuickSort**

- Stabil: Nein
- In-Place
- Hilfsspeicher im Worst-Case O(n)

3. Sortieralgorithmen: **HeapSort**



Quelle: http://www.c-sharpcorner.com/UploadFile/fd0172/heap-sort-in-java/

3. Sortieralgorithmen: **HeapSort**

Best-Case

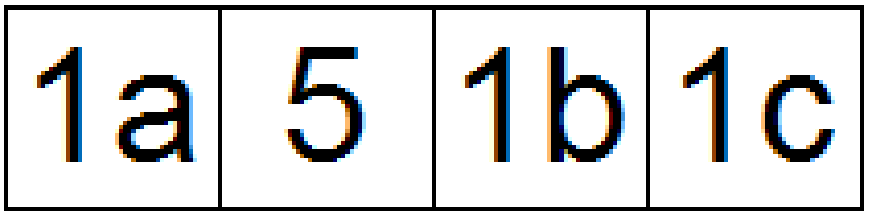
- O(n log n)
- Heapify: wenig Vertauschungen
- Liste mit vielen gleichen Werten

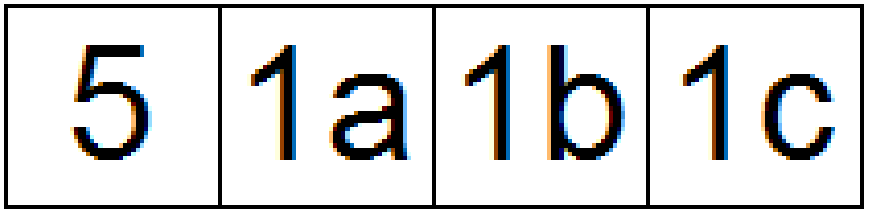
Worst-Case

- O(n log n)
- Heapify: maximale Vertauschungen
- z.B. nächst großes Element ist im Baum unten

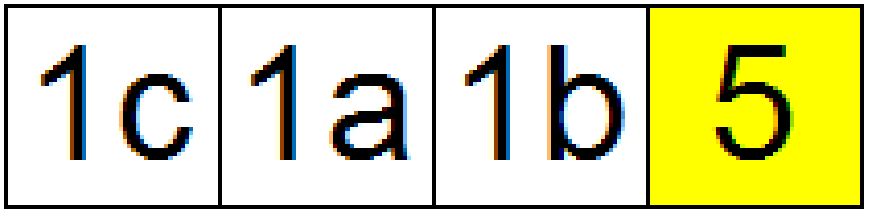
3. Sortieralgorithmen: **HeapSort**

- Stabil: Nein
- Wurzel mit letzter Stelle vertauschen





- In-Place: Ja
- Hilfsspeicher O(1)



Evaluation

Zeit in ms	BubbleSort	SelectionSort	InsertionSort	QuickSort	MergeSort	Heapsort
Zufall(100k)	32093,6	14644	12765,4	15,8	46,4	31,4
aufsteigend(100k)	<1	14640,6	<1	6,4	31,2	34
absteigend(100k)	32427,8	15290,6	30072,2	9,2	31,2	40,8
aufsteigend, kleinste am Ende(100k)	13956,4	14553	<1	<1	34,4	31
gleiche werte(100k)	<1	14687	<1	6,4	31,2	<1

Zeit in ms	BubbleSort	SelectionSort	InsertionSort	QuickSort
absteigend(10k)	465,6	137,4	259,2	218,8

Fazit

- **Quadratische**
- Bei kleinen Werten
- Zu hohe Laufzeit

QuickSort

- Schneller als MS
- Worst-Case $O(n^2)$

- **MergeSort**
- Stabil
- Hoher Speicherplatzbedarf

- **HeapSort**
- Langsamer als MS
- Konstante Ergebnisse
- Hilfsspeicher von O(1)