# Sortieralgorithmen

## KLASSIFIZIERUNGEN:

| Intern | Extern |
|---|---|
| Vergleichsbasiert | Nicht-Vergleichsbasiert |
| Stabil<br>Reihenfolge von gleichen Elementen bleibt unverändert | Instabil |
| In-Situ *(in-place)*<br>Überschreibt Eingabe- mit Ausgabedaten<br>→ Konstanter Speicherverbrauch | Ex-Situ *(out-of-place)*<br>→ Speicherverbrauch steigt mit Eingabeanzahl |

## SELECTIONSORT

1. Lösche Maximum aus der Eingabeliste
2. Füge Maximum ans Ende der Ergebnisliste an
3. Wiederhole bis Eingabeliste leer ist

Laufzeitkomplexität: $O(n^2)$

Stabil, In-Situ

**Beispiel**

Sortieren Sie folgende Elemente absteigend mittels SelectionSort:
13, 7, 24, 3, 19, 12, 11.

| $l_s$ \| $l$ | | | | | | |
|---|---|---|---|---|---|---|
| 13 | 7 | 24 | 3 | 19 | 12 | 11 |
| 24 \| 13 | 7 | 3 | 19 | 12 | 11 | |
| 24 | 19 \| 13 | 7 | 3 | 12 | 11 | |
| 24 | 19 | 13 \| 7 | 3 | 12 | 11 | |
| 24 | 19 | 13 | 12 \| 7 | 3 | 11 | |
| 24 | 19 | 13 | 12 | 11 \| 7 | 3 | |
| 24 | 19 | 13 | 12 | 11 | 7 \| 3 | |
| 24 | 19 | 13 | 12 | 11 | 7 | 3 |

## INSERTIONSORT

1. Lösche erstes Element aus der Eingabeliste
2. Füge sortiert in Ergebnisliste ein
3. Wiederhole bis Eingabeliste leer ist

Laufzeitkomplexität:
Best-Case: $O(n)$ ; Worst-Case: $O(n^2)$

Stabil, In-Situ

**Beispiel**

Sortieren Sie folgende Elemente aufsteigend mittels InsertionSort:
13, 7, 24, 3, 19, 12, 11.

| $l_s$ \| $l$ | | | | | | |
|---|---|---|---|---|---|---|
| 13 | 7 | 24 | 3 | 19 | 12 | 11 |
| 13 \| 7 | 24 | 3 | 19 | 12 | 11 | |
| 7 | 13 \| 24 | 3 | 19 | 12 | 11 | |
| 7 | 13 | 24 \| 3 | 19 | 12 | 11 | |
| 3 | 7 | 13 | 24 \| 19 | 12 | 11 | |
| 3 | 7 | 13 | 19 | 24 \| 12 | 11 | |
| 3 | 7 | 12 | 13 | 19 | 24 \| 11 | |
| 3 | 7 | 11 | 12 | 13 | 19 | 24 |

## BUBBLESORT

1. Vorne anfangen
2. Zwei nicht sortierte Element vertauschen
3. Ende erreicht, wenn in einem Durchlauf nichts vertauscht wurde

Laufzeitkomplexität: Best-Case: $O(n)$ ; Worst-Case: $O(n^2)$

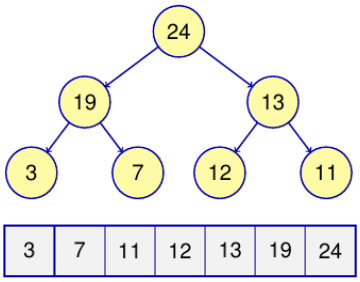
Stabil, In-Situ

**Beispiel**

Sortieren Sie folgende Elemente absteigend mittels BubbleSort:
13, 7, 24, 3, 19, 12, 11.

| $l$ | | | | | | |
|---|---|---|---|---|---|---|
| 13 | 7 | 24 | 3 | 19 | 12 | 11 |
| 13 | 24 | 7 | 19 | 12 | 11 | 3 |
| 24 | 13 | 19 | 12 | 11 | 7 | 3 |
| 24 | 19 | 13 | 12 | 11 | 7 | 3 |
| 24 | 19 | 13 | 12 | 11 | 7 | 3 |

## HeapSort

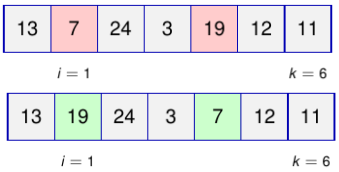



1. Alle Elemente in Min/Max-Halde einfügen
2. Wurzel entnehmen und in Ergebnisliste einfügen
3. Wiederhole bis Halde leer ist

Laufzeitkomplexität: $O(n \cdot \log(n))$

Instabil, In-Situ *möglich*

### 1.1 HeapSort in-situ:



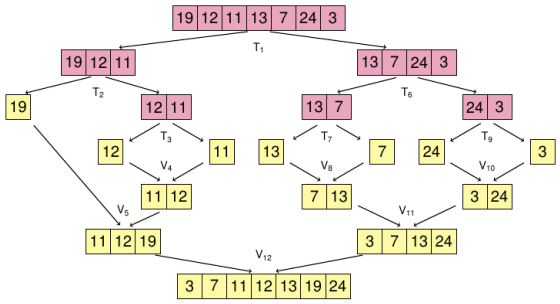

Unsortierte Eingabeliste kann *in situ* in eine Min/Max-Halde umgewandelt werden:

1. Von hinten nach vorne für jeden Eintrag die Halden-Eigenschaft überprüfen:

   $a[i] \geq a[2i+1]$ $a[i] \geq a[2i+2]$ ← Index *i* geht von *a.length-1* bis *0*

2. Falls ein falscher Eintrag (*a[i]*) gefunden wurde, mit entsprechendem Eintrag (*a[2i+1]* / *a[2i+2]*) vertauschen („versickern“)

## MergeSort (*divide → merge*)

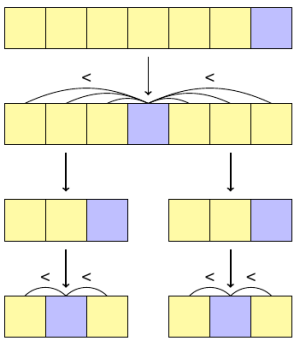



1. Falls Eingabeliste mehr als 1 Element enthält:
   1. Aufteilen in zwei kleinere Listen
   2. Solange rekursiv Aufteilen bis nur noch „1er-Listen“ übrig sind
2. Listen sortiert zusammenfügen
   (durch Sortierung muss nicht jedes Element mit der kompletten anderen Liste verglichen werden, sondern nur mit den nachfolgenden Elem. des vorherigen Eintrags, im Bsp. *12* muss nur noch mit *[13,24]* vergl. werden)

Laufzeitkomplexität: $O(n \cdot \log(n))$

Stabil, *je nach Merge-Verfahren*, Ex-Situ

## QuickSort



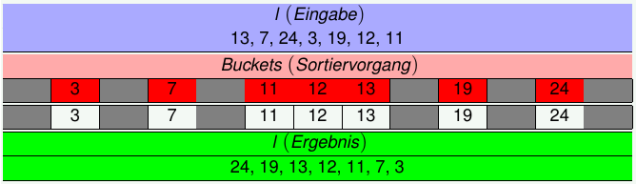

1. Falls Eingabeliste mehr als 1 Element enthält:
   1. Pivot-Element auswählen
   2. Kleinere Element vor / größere Element hinter das Pivot-Element tauschen
   3. Solange rekursiv Aufteilen bis nur noch „1er-Listen“ übrig sind
2. Listen hintereinander zusammenfügen (sind automatisch sortiert)

| | | | | | | |
|---|---|---|---|---|---|---|
| 13 | 7 | 24 | 3 | 19 | 12 | 11 |
| 7 | 3 | **11** | 13 | 19 | 12 | 24 |
| **3** | 7 | 11 | 13 | 19 | 12 | 24 |
| 3 | 7 | 11 | 13 | 19 | 12 | **24** |
| 3 | 7 | 11 | **12** | 19 | 13 | 24 |
| 3 | 7 | 11 | 12 | **13** | 19 | 24 |
| 3 | 7 | 11 | 12 | 13 | 19 | 24 |

Laufzeitkomplexität: Best-Case: $O(n \cdot \log(n))$ Worst-Case: $O(n^2)$

Instabil, Ex-Situ *(wg. Rekursion)*

## BucketSort




Elemente müssen eine bekannte Ordnung haben
(z. B. bei Zahlen *A* > *B)*

1. Elemente in entsprechende Buckets einfügen
2. Elemente der Reihe nach den Buckets entnehmen

Es werden nie 2 Werte miteinander verglichen → **Nicht-Vergleichsbasiert**

Laufzeitkomplexität: $O(n+m)$

Stabil, Ex-Situ

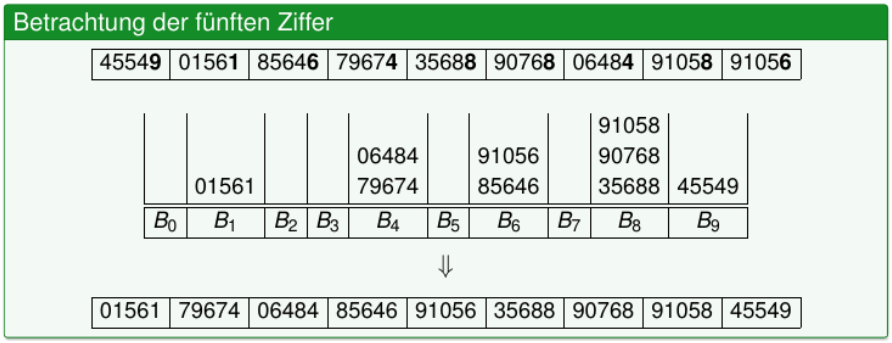
## RadixSort

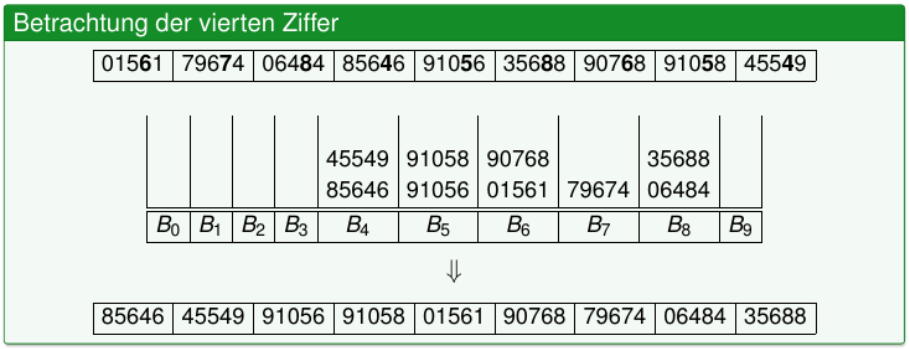
Elemente sind aus einem festen Alphabet zusammengesetzt mit bekannter Ordnung (s. o.)

1. Die einzelnen Zeichen der Elemente werden von hinten nach vorne betrachtet:
    1. Element entsprechend dem Zeichen in passenden Bucket einfügen
    2. Elemente der Reihe nach aus den Buckets entnehmen (LIFO)
2. Wiederhole solange bis alle Zeichen betrachtet wurden

Laufzeitkomplexität: $O(n \cdot l)$ (*l* ist Länge des längsten Elements)

Stabil, Ex-Situ, **Nicht-Vergleichsbasiert**




*usw. bis zur ersten Ziffer*