

# Algorithmen und Datenstrukturen

## A3. Sortieren I: Selection- und Insertionsort

Marcel Lüthi and Gabriele Röger

Universität Basel

24. Februar 2022

# Algorithmen und Datenstrukturen

## 24. Februar 2022 — A3. Sortieren I: Selection- und Insertionsort

A3.1 Sortieralgorithmen

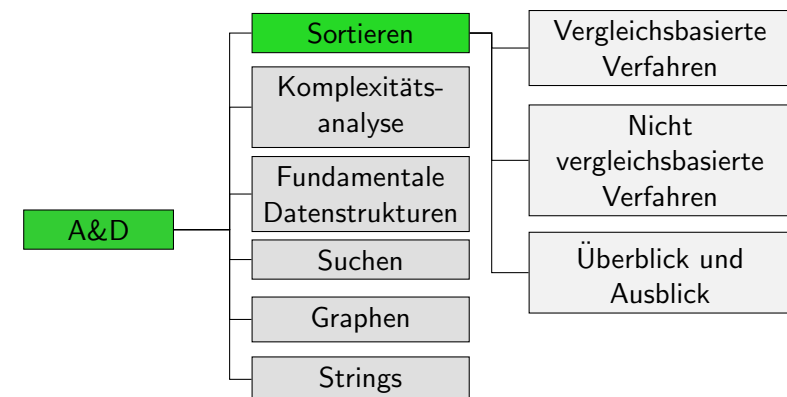
A3.2 Selectionsort

A3.3 Insertionsort

A3.4 Zusammenfassung

## A3.1 Sortieralgorithmen

## Inhalt dieser Veranstaltung



## Relevanz

Sortieren von Daten wichtig für viele Anwendungen, z.B.

- ▶ **sortierte Darstellung** (z.B. auf Webseite)
  - ▶ Produkte sortiert nach Preis, Kundenbewertung, etc.
  - ▶ Kontobewegungen sortiert nach Buchungsdatum
- ▶ **Vorverarbeitung** für viele effiziente **Suchalgorithmen**
  - ▶ Wie schnell können Sie eine Nummer im Telefonbuch nachschlagen? Und wenn die Einträge nicht sortiert wären?
- ▶ **Vorverarbeitung** für viele **andere Verfahren**
  - ▶ z.B. Kruskals Algorithmus zur Berechnung minimaler Spannbäume von ungerichteten Graphen

Fachzeitschrift „Computing in Science & Engineering“ nennt Quicksort-Sortieralgorithmus als einen der 10 wichtigsten Algorithmen des 20. Jahrhunderts.

## Aufgabenstellung

### Aufgabenstellung Sortieralgorithmen

#### Eingabe

- ▶ Sequenz von  $n$  Elementen  $e_1, \dots, e_n$
- ▶ Jedes Element  $e_i$  hat Schlüssel  $k_i = \text{key}(e_i)$
- ▶ Ordnungsrelation  $\leq$  auf den Schlüsseln
  - reflexiv:  $k \leq k$
  - transitiv:  $k \leq k'$  und  $k' \leq k'' \Rightarrow k \leq k''$
  - antisymmetrisch:  $k \leq k'$  und  $k' \leq k \Rightarrow k = k'$

#### Resultat

- ▶ Sequenz der Eingabeelemente gemäss Ordnungsrelation ihrer Schlüssel sortiert

**Notation:** auch  $e \leq e'$  für  $\text{key}(e) \leq \text{key}(e')$

## Aufgabenstellung: Beispiele

### Beispiel

**Eingabe:**  $\langle 3, 6, 2, 3, 1 \rangle$ ,  $\text{key}(e) = e$ ,  $\leq$  auf natürlichen Zahlen

**Ausgabe:**  $\langle 1, 2, 3, 3, 6 \rangle$

### Beispiel

**Eingabe:** Liste aller Studierenden der Uni Basel,

$\text{key}(e) = \langle \text{Wohnort von } e \rangle$ , lexikographische Ordnung

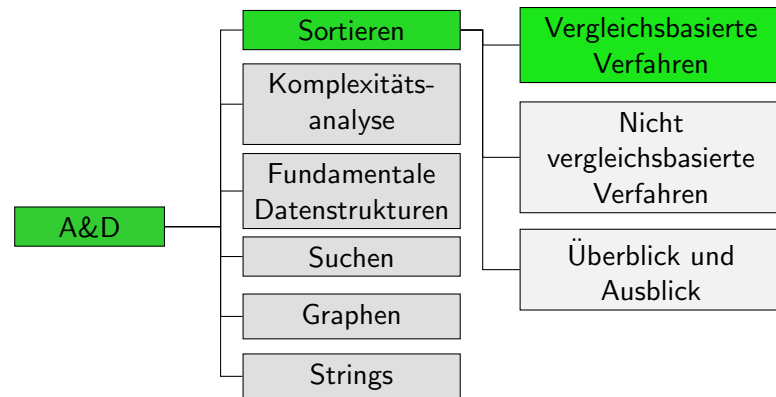
**Ausgabe:** Liste aller Studierenden, nach Wohnort sortiert

Bis auf weiteres: ganze Zahlen,  $\text{key}(e) = e$  und „kleiner gleich“  
Später (und Übung): Umgang mit komplexen Objekten

## Interessante Eigenschaften von Sortieralgorithmen

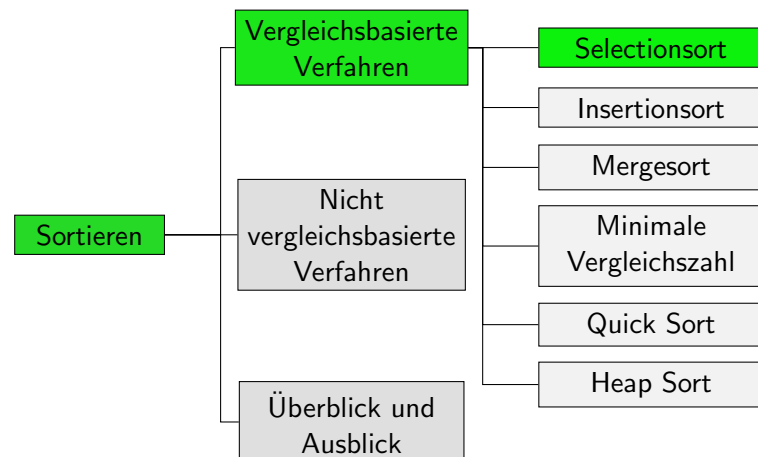
- ▶ **Zeitbedarf:** Wieviele Schlüsselvergleiche und Element-Vertauschungen werden durchgeführt?
- ▶ **adaptiv:** Verfahren ist schneller, wenn Eingabe bereits (teilweise) vorsortiert ist.
- ▶ **Platzbedarf:** Wieviel Speicherplatz wird zusätzlich zum Eingabearray verwendet (explizit oder im call stack)?
- ▶ **in-place:** Zusätzlich verbrauchter Platz ist konstant (nicht abhängig von der Eingabegrösse).
- ▶ **stabil:** Reihenfolge von Elementen mit gleichem Schlüssel wird nicht verändert.
- ▶ **vergleichsbasiert:** Verfahren verwendet nur Vergleich von Schlüsselpaaren und Tausch zweier Elemente.

## Inhalt dieser Veranstaltung

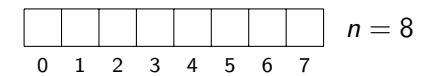


## A3.2 Selectionsort

## Sortierverfahren



## Selectionsort: Informell



- ▶ Finde kleinstes Element an Positionen  $0, \dots, n - 1$  und tausche es an Position 0
- ▶ Finde kleinstes Element an Positionen  $1, \dots, n - 1$  und tausche es an Position 1
- ▶ ...
- ▶ Finde kleinstes Element an Positionen  $n - 2, \dots, n - 1$  und tausche es an Position  $n - 2$

## Selectionsort: Algorithmus

```

1 def selection_sort(array):
2     n = len(array)
3     for i in range(n - 1): # i = 0, ..., n-2
4         # find index of minimum element at positions i, ..., n-1
5         min_index = i
6         for j in range(i + 1, n): # j = i+1, ..., n-1
7             if array[j] < array[min_index]:
8                 min_index = j
9         # swap element at position i with minimum element
10        array[i], array[min_index] = array[min_index], array[i]

```

## Selectionsort: Beispiel

i	min_ind.	0	1	2	3	4	5	6	7
		3	7	2	9	7	1	4	5
0	5	3	7	2	9	7	1	4	5
1	2	1	7	2	9	7	3	4	5
2	5	1	2	7	9	7	3	4	5
3	6	1	2	3	9	7	7	4	5
4	7	1	2	3	4	7	7	9	5
5	5	1	2	3	4	5	7	9	7
6	7	1	2	3	4	5	7	9	7
		1	2	3	4	5	7	7	9

Minimum wird in dunklen Einträgen gesucht.

Roter Eintrag ist gefundenes Minimum.

Graue Einträge sind in richtiger Reihenfolge.

## Selectionsort: Korrektheit

- ▶ **Invariante:** Eigenschaft, die während der gesamten Algorithmenlaufzeit gilt.
- ▶ **Invariante 1:** Zum Ende jedes Durchlaufs der äusseren Schleife sind die Elemente an den Positionen  $\leq i$  sortiert.
- ▶ **Invariante 2:** Zum Ende jedes Durchlaufs der äusseren Schleife ist keines der Elemente an den Positionen  $\leq i$  grösser als ein Element an einer Position  $> i$ .
- ▶ Korrektheit der Invarianten per (gemeinsamer) Induktion
- ▶ Nach letztem Schleifendurchlauf sind alle Elemente bis auf das letzte in korrekter Reihenfolge und das letzte ist nicht kleiner als das vorletzte.  
→ gesamte Eingabe sortiert

## Selectionsort: Eigenschaften

- ▶ **in-place:** zusätzlicher Speicherbedarf nicht abhängig von Eingabegrösse
- ▶ **Zeitbedarf:** hängt nur von Grösse der Eingabe ab (nicht adaptiv für teilsortierte Eingaben)  
*genauere Analyse: nächste Woche*
- ▶ **nicht stabil:** beim Tausch kann das Element an Position  $i$  hinter ein gleiches Element springen, was später nicht mehr "repariert" wird.

## Jupyter-Notebook

**Selection Sort**

Hier ist nochmal der Selection-Sort-Algorithmus aus der Vorlesung:

```
In [1]: 1 def selection_sort(array):
2         n = len(array)
3         for i in range(n - 1):
4             # print(array)
5             min_index = i
6             for j in range(i + 1, n):
7                 if array[j] <= array[min_index]:
8                     min_index = j
9             # print("Minimales Element an Pos.", i, "-", len(array) - 1,
10            # "ist", array[min_index])
11            # print("Tausche es mit", array[i], "an Pos.", i)
12            array[i], array[min_index] = array[min_index], array[i]
```

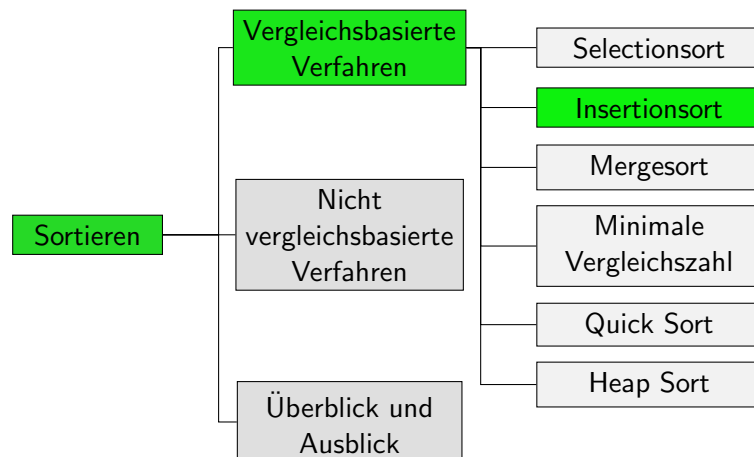
Hier ein Beispielaufwurf:

```
In [2]: 1 test_array = [7,3,5,9,3]
2         selection_sort(test_array)
3         print(test_array)
[3, 3, 5, 7, 9]
```

Jupyter-Notebook: selection\_sort.ipynb

## A3.3 Insertionsort

## Sortierverfahren



## Insertionsort: Informell



- ▶ Ähnlich zum Sortieren von Spielkarten auf der Hand
- ▶ Elemente werden nacheinander in bereits sortierten Bereich am Sequenzanfang einsortiert.
- ▶ Grössere Elemente werden entsprechend nach hinten verschoben.

## Insertionsort: Beispiel

i	0	1	2	3	4	5	6	7
	3	7	2	9	7	1	4	5
1	3	7	2	9	7	1	4	5
2	2	3	7	9	7	1	4	5
3	2	3	7	9	7	1	4	5
4	2	3	7	7	9	1	4	5
5	1	2	3	7	7	9	4	5
6	1	2	3	4	7	7	9	5
7	1	2	3	4	5	7	7	9

← Graue Einträge  
wurden nicht bewegt.

↗ Roter Eintrag  
wurde einsortiert.

↖ Schwarze Einträge  
wurden um eins  
nach rechts verschoben.

## Insertionsort: Algorithmus

```

1 def insertion_sort(array):
2     n = len(array)
3     for i in range(1, n): # i = 1, ..., n - 1
4         # move array[i] to the left until it is
5         # at the correct position.
6         for j in range(i, 0, -1): # j = i, ..., 1
7             if array[j] < array[j-1]:
8                 # not yet at final position.
9                 # swap array[j] and array[j-1]
10                array[j], array[j-1] = array[j-1], array[j]
11            else:
12                break # continue with next i

```

## Insertionsort: Algorithmus (etwas schneller)

Vorherige Version: meiste Zuweisungen an `array[j-1]` unnötig.

```

1 def insertion_sort(array):
2     for i in range(1, len(array)):
3         val = array[i]
4         j = i
5         while j > 0 and array[j - 1] > val:
6             array[j] = array[j - 1]
7             j -= 1
8         array[j] = val

```

Laufzeitanalyse (später): kein fundamentaler Unterschied  
trotzdem: zu bevorzugen, wenn direkte Zuweisung möglich

## Insertionsort: Korrektheit

- ▶ **Invariante 1:** Zu Beginn jedes Durchlaufs der äusseren Schleife sind die Elemente an den Positionen  $< i$  sortiert.
- ▶ **Invariante 2:** Sei  $val$  der Wert an Position  $i$  vor Beginn der inneren Schleife. Zu Beginn jedes Durchlaufs der inneren Schleife sind die Elemente an den Positionen  $j$  bis  $i$  grösser oder gleich  $val$ .
- ▶ Korrektheit der Invarianten per Induktion
- ▶ Die innere Schleife verändert die Reihenfolge der an eine höhere Position verschobenen Elemente nicht und das nach unten verschobene Element wird korrekt einsortiert.
- ▶ Nach letztem Schleifendurchlauf sind alle Elemente sortiert.

## Insertionsort: Eigenschaften

- ▶ **in place**: zusätzlicher Speicherbedarf nicht abhängig von Eingabegrösse
  - ▶ **Zeitbedarf**: adaptiv für teilsortierte Eingaben
    - ▶ Bei bereits sortierter Eingabe bricht innere Schleife direkt ab.
    - ▶ Bei umgekehrt sortierter Eingabe wird jedes Element schrittweise bis ganz vorne verschoben.
- genauere Analyse: nächste Woche
- ▶ **stabil**: Element wird nur so lange nach vorne verschoben, solange es mit echt grösserem Element getauscht wird.  
→ kann nicht Reihenfolge mit gleichem Element tauschen.

## A3.4 Zusammenfassung

## Zusammenfassung

- ▶ **Selectionsort** und **Insertionsort** sind zwei einfache Sortierverfahren.
- ▶ **Selectionsort** baut die sortierte Sequenz von vorne auf, indem es sukzessive ein minimales Element aus dem noch unsortierten Bereich an das Ende des sortierten Bereichs tauscht.
- ▶ **Insertionsort** betrachtet die Elemente von vorne nach hinten und sortiert sie in den bereits sortierten Bereich am Sequenzanfang ein.