Algorithmen und Datenstrukturen
Kapitel 1: Grundlagen

Prof. Dr. Wolfgang Mühlbauer
Fakultät für Informatik
wolfgang.muehlbauer@th-rosenheim.de

Wintersemester 2019/2020

Wichtiges vorab

- *Bei Algorithmen und Datenstrukturen geht es genauso wenig um Programmiersprachen wie in der Astronomie um Teleskope."*
  - *(abgewandelt von E. Dijkstra, 1930-2002)*

Aus [1]

- Wissen im Bereich Informatik altert nicht!
  - Im Gegensatz zu Programmiersprachen.

- Der steigenden Daten- und Rechenkomplexität muss man mit effizienten Algorithmen begegnen.
  - Neue, schnellere Hardware alleine genügt nicht.

Inhalt

- **Beispiel 1: Pledge-Algorithmus**
- Beispiel 2: Sortieren mit Insertionsort
- Analyse von Algorithmen
- Asymptotisches Wachstum
- Zusammenfassung

Algorithmus

- **Definition "Algorithmus"**:
  - ***Eindeutige***, ***ausführbare*** Folge von Anweisungen ***endlicher*** Länge zur Lösung eines Problems.

- *"Mathematisch"*: Berechnungsvorschrift, die
  - zu einer Menge an *Eingabewerten*
  - eine Menge an *Ausgabewerten* liefert.

- Beispiel:
  - Zahlen sollen aufsteigend sortiert werden.
  - Formelle Spezifikation
    - *Eingabe*: Sequenz von *n* Zahlen $\langle a_1, a_2, \ldots, a_n \rangle$
    - *Ausgabe*: Permutation (=Umsortierung) der Eingangssequenz $\langle a_1', a_2', \ldots, a_n' \rangle$, so dass

$$\langle a_1' \leq a_2' \leq \cdots \leq a_n' \rangle$$

Beispiele für Algorithmen: Brainstorming

- Sortieren
- ggT, kkV: Euklidischer Algorithmus
- RSA, Verschlüsselung
- Levenstheim-Distanz, Ähnlichkeit von Wörtern
- Huffman-Code
- Shuffle Algorithmus: Fischer-Yates-Algorithmus
- A*-Algorithmus
- …

Kriterien zur Beurteilung von Algorithmen

- ***Terminiert***
  - Algorithmus liefert Ergebnis in *endlicher* Zeit.

- ***Korrekt***
  - Algorithmus liefert den korrekten, erwarteten Wert.
  - Hinweis: Korrektheit wird in Vorlesung selten bewiesen.

- ***Effizienz bzgl.***
  - ***Laufzeit:*** Wieviel Zeit benötigt Algorithmus zur Berechnung?
  - ***Speicherplatz***: Wie viel Speicher benötigt Algorithmus?
    - *In-Place:* Algorithmus benötigt ***keinen*** zusätzlichen Speicher.

- ***Deterministisch***
  - Wiederholt man Algorithmus mit *gleichen* Eingabewerten→ *Gleiches* Ergebnis!

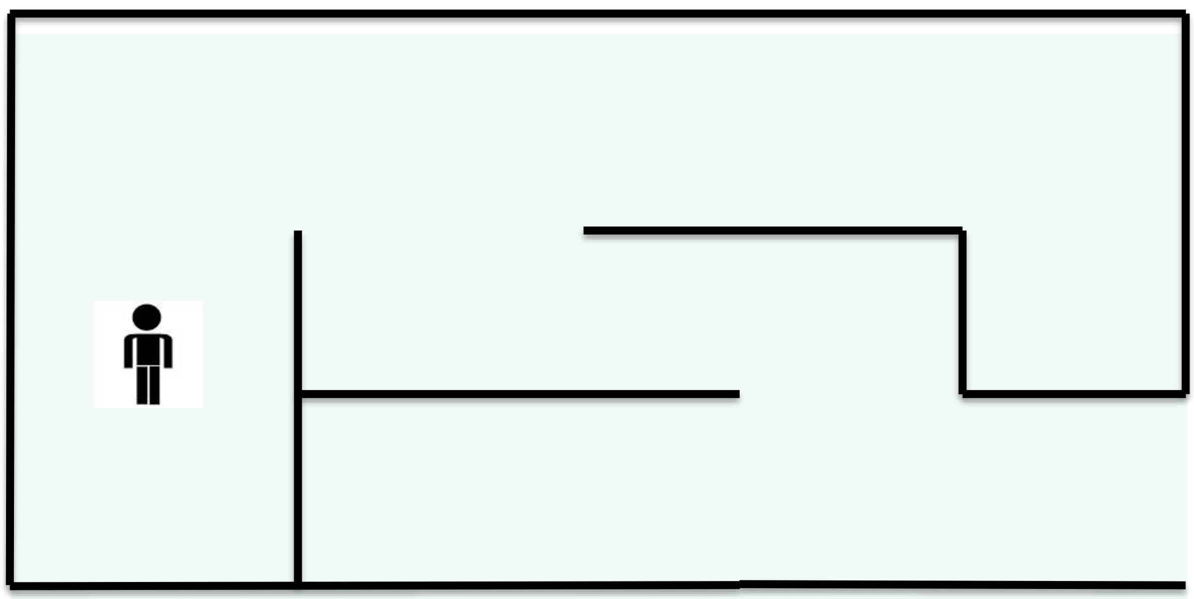
- ***Parallel***
  - Algorithmus führt (einen Teil der) Berechnungen parallel / mehrfädig aus.

Inhalt

- Algorithmus: Definition und Eigenschaften
- **Beispiel 1: Pledge-Algorithmus**
- Beispiel 2: Sortieren mit Insertion Sort
- Analyse von Algorithmen
- Asymptotisches Wachstum

Wie entkommt man dem Labyrinth?

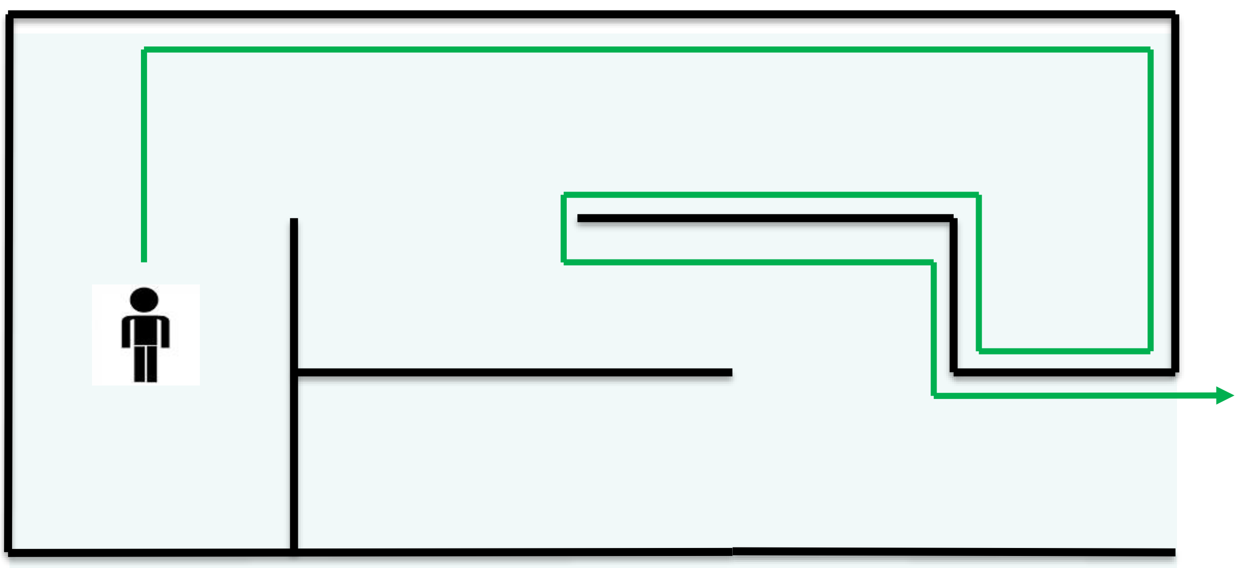
- **Problem**
  - Es ist dunkel. → Man kann Wände nur mit Händen ertasten.
  - Keine Markierungsmöglichkeit, z.B. Kreide → Man kann sich Rückweg nicht merken.
- **Annahmen**:
  - Nur rechtwinklige Ecken.
  - Man kann auch im Dunkeln geradeaus gehen (der "Nase lang")
- **Anwendung:** Roboter ohne GPS muss sich orientieren



Wie entkommt man dem Labyrinth?

Version 1.0:

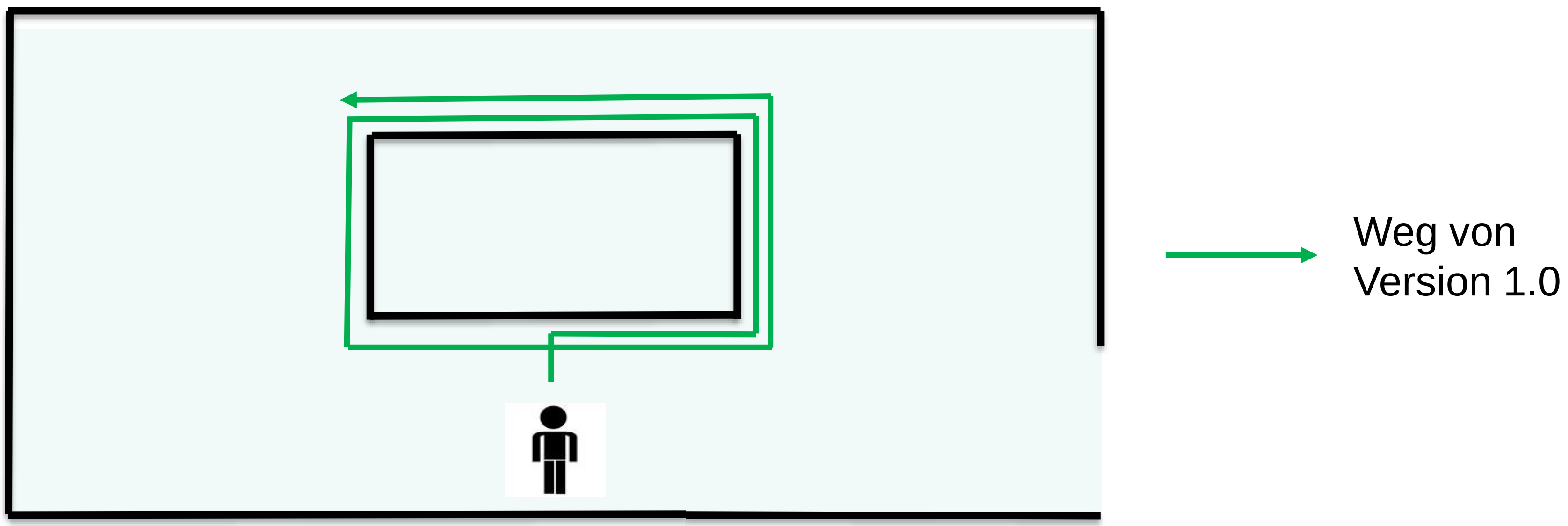
```
1.  Geradeaus gehen bis man auf Wand trifft ("der Nase lang").
2.  Biege nach rechts ab, um 90° im Uhrzeigersinn drehen.
3.  Von nun an mit linker Hand an Wand entlang gehen bis zum
    Ausgang.
```



- Funktioniert dieser "Algorithmus" immer?

Ist Version 1.0 korrekt?

- Nein, Version 1.0 funktioniert hier leider nicht.
  - Man kreist unendlich oft um die Säule.

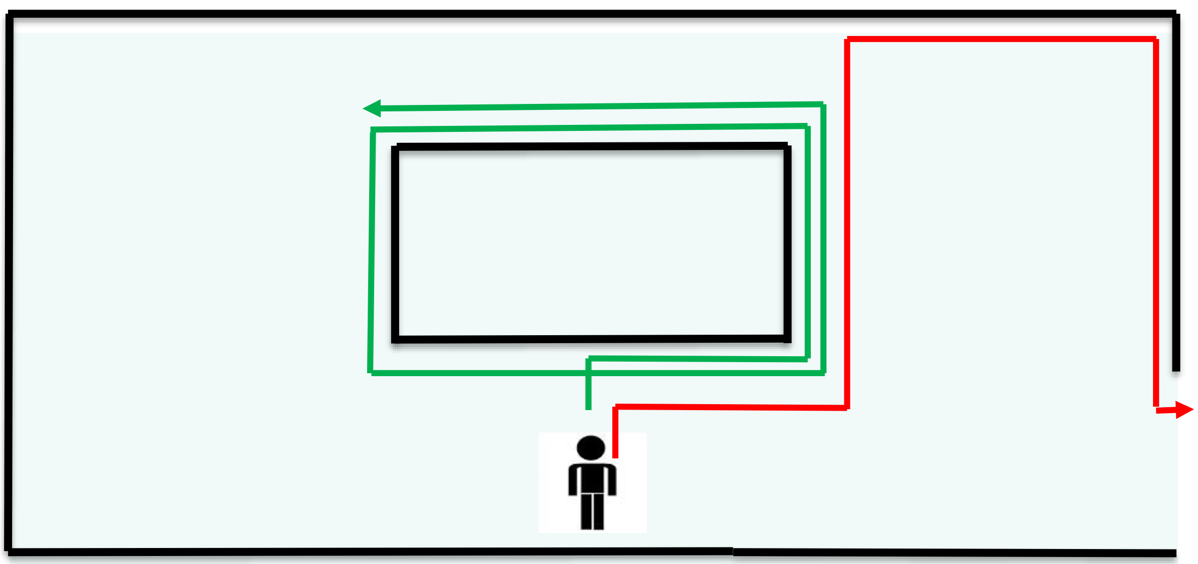


- Wie könnte man den Algorithmus modifizieren, damit er auch bei einer Säule funktioniert?

Wie entkommt man dem Labyrinth?

<u>Version 2.0 (Pseudocode):</u>

```
repeat
   Geradeaus gehen bis man auf Wand trifft ("der Nase lang")
   Biege nach rechts ab, um 90° im Uhrzeigersinn drehen.
   Wand nun nur so lange folgen, bis man wieder in "alte"
   Richtung läuft → dann wieder "der Nase lang"
until Ausgang gefunden
```



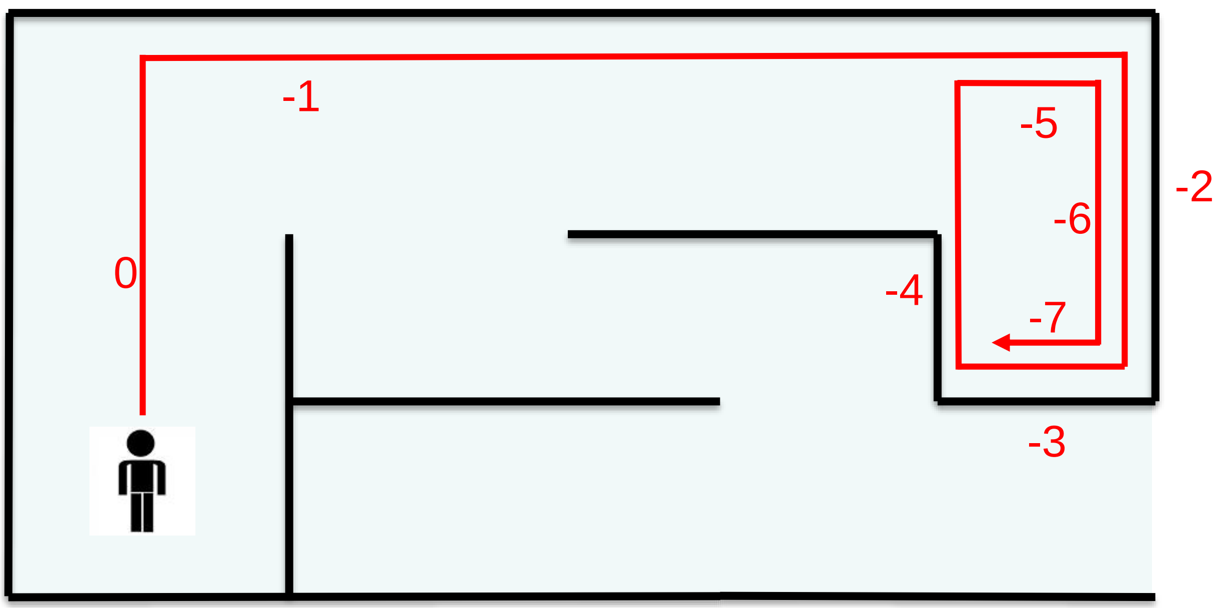
Lösung:
Man merkt sich die Richtung!

Version 1.0

Version 2.0

Ist Version korrekt?

- Version 2.0 funktioniert nun aber beim 1. Beispiel nicht!
  - Was passiert hier?

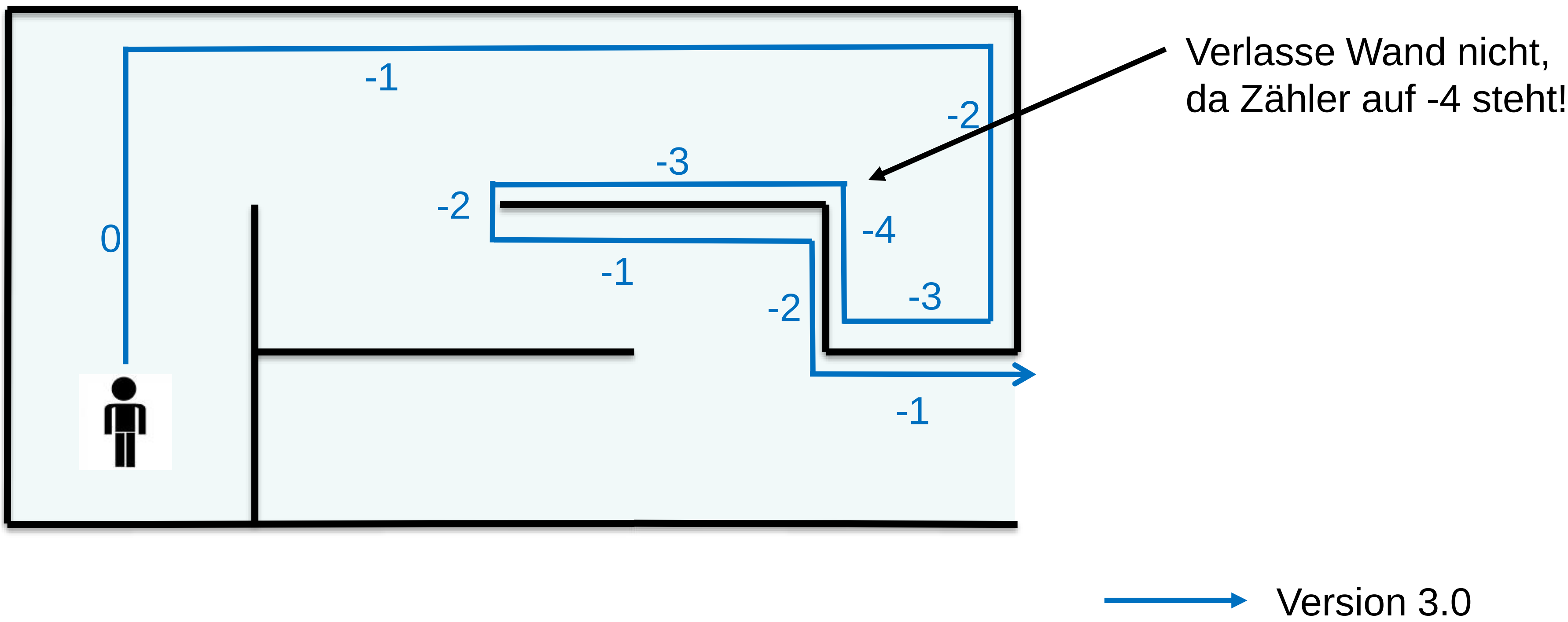


Lösung: Umdrehungszähler
- -1 bei Abbiegen nach rechts
- +1 bei Abbiegen nach links
- Nur bei 0 die Wand verlassen

→ Version 2.0

Ist Version korrekt?

- Version 3.0
  - Verlasse Wand nur, wenn der Umdrehungszähler auf 0 steht.



Version 3.0

<u>Algorithmus von Pledge (Version 3.0)</u>

```
setze Umdrehungszähler auf 0
repeat
   repeat
      gehe geradeaus
   until Wand erreicht
   Biege nach rechts ab, dekrementiere Umdrehungszähler
   repeat
      folge dem Hindernis mit einer Hand
      dabei: je nach Drehrichtung Umdrehungszähler anpassen
   until Umdrehungszähler = 0
until Ausgang erreicht
```

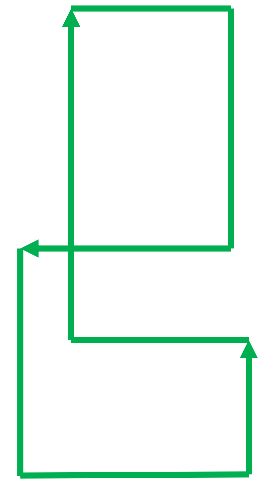
- Bei Umdrehungszähler 0 und *nur genau dann* muss das Hindernis verlassen werden.
  - Ansonsten bleibt man am Hindernis
  - Auch wenn Umdrehungszähler ein Vielfaches von 4 ist wie z.B. -4, 8!

Exkurs: Korrektheit des Pledge Algorithmus

- Korrektheitsbeweise werden in der Vorlesung oft weggelassen
  - Häufig schwieriger als Laufzeitanalyse!

- Pledge Algorithmus ist **korrekt**, Details siehe [2].

- Beweis: Grundidee
  - Annahme: Man käme mit Pledge-Algorithmus nicht heraus.
  - Dann: Teil des Weges wird immer wieder durchlaufen (nur endlich viele *Punkt*e, wo Änderung der Bewegungsrichtung möglich)
  - Dieser Teil des Weges bildet zwingend ***Zyklus.*** Warum?
  - Dann zeigt man: Zyklus kann sich nicht selbst kreuzen.
  - Zuletzt zeigt man: So ein Zyklus muss im Uhrzeigersinn immer wieder durchlaufen werden. Das ist gleichzeitig aber nur möglich, wenn es keinen Weg nach draußen gibt.



Zyklus mit Kreuzung

- Mehr Lösungsalgorithmen für Irrgärten:
  - https://de.wikipedia.org/wiki/L%C3%B6sungsalgorithmen_f%C3%BCr_Irrg%C3%A4rten

Publikums-Joker

Welche der folgenden Aussagen ist ***korrekt?***

A. Der Pledge-Algorithmus hat keinerlei Voraussetzungen.

B. Der Pledge-Algorithmus erfordert das Mitführen von Kreide.

C. Der Pledge-Algorithmus erfordert ein gewisses Gedächtnis.

D. Der Pledge-Algorithmus erlaubt es, den kürzesten Ausweg aus einem Irrgarten zu finden.

Inhalt

- Algorithmus: Definition und Eigenschaften
- Beispiel 1: Pledge-Algorithmus
- **Beispiel 2: Sortieren mit Insertion Sort**
- Abschätzung der Laufzeit
- Asymptotische Laufzeit

InsertionSort

- Effizienter Algorithmus fürs Sortieren einer Menge an Elementen.

gemeinfrei

- **Idee**
  - Ähnliches Vorgehen wie beim Sortieren von Spielkarten.
  - Beginne mit leerer linker Hand.
  - Wiederhole
    - Nimm (mit rechter Hand) nächste Karte vom Stapel.
    - Füge diese in die korrekte Position der linken Hand ein.
      - Ermittlung der korrekten Position: Vergleiche gezogene Karte mit jeder Karte, die bereits in der Hand ist.
    - Um Platz für das Einfügen zu schaffen, müssen höhere Karten nach rechts verschoben werden.

InsertionSort: Code

```
INSERTION-SORT(A)                                        // A: Input Array
    for j = 1 to A.length-1
        key = A[j]
        // insert A[j] into already sorted sequence
        i = j - 1
        while i ≥ 0 and A[i] > key
            A[i+1] = A[i]     // shift to the right
            i = i - 1
        A[i+1] = key
```

Quellcode: InsertionSort.java

- ***Invariante:***
  - Nach jedem Durchlauf der for-Schleife: Linker Teil des Arrays bis (*j*-1) ist bereits sortiert.
- Aktuelles Element (*key*) wird in den bereits sortierten linken Teil eingefügt.
  - Vergleiche *key* mit Elementen im linken Teil (von rechts nach links), bis passende Position gefunden.
  - Verschiebe Elemente im linken, sortierten Teil um 1 nach rechts, um Platz für *key* zu machen.
- **Pseudocode**
  - Aussagekräftige, knappe Beschreibung des Algorithmus!
  - Ignoriere Details von Programmiersprachen (z.B. Variablendeklaration)

Publikums-Joker

Welche Aussage ist ***korrekt*** bzgl. des *InsertionSort* Codes?

A. *InsertionSort* ist bei gleicher Eingabegröße immer gleich schnell.

B. Falls *InsertionSort* ein *bereits sortiertes Array* sortieren soll, nimmt der Algorithmus dennoch Vertauschungen vor.

C. Falls *InsertionSort* ein Array mit *lauter "gleich großen"* Einträgen sortieren soll, nimmt der Algorithmus dennoch Vertauschungen vor.

D. Das *InsertionSort*-Prinzip kann auch auf das lexikographische Sortieren von Wörtern angewendet werden.

Inhalt

- Algorithmus: Definition und Eigenschaften
- Beispiel 1: Pledge-Algorithmus
- Beispiel 2: Sortieren mit Insertion Sort
- **Analyse von Algorithmen**
- Asymptotisches Wachstum

Analyse von Algorithmen

- Meist Analyse der *Laufzeit* oder des *Speicherverbrauchs*.

- ***Laufzeitanalyse*** abhängig von
  - Größe der Eingabe
    - Beispiel: Anzahl zu sortierender Elemente
  - Eigenschaften der Eingabeinstanz
    - Beispiel: Es dauert länger ein absteigend sortiertes Array aufsteigend zu sortieren als ein Array, das schon fast korrekt sortiert ist (*Worst Case*)

- ***Rechnermodell*** notwendig, um die Laufzeit eines Algorithmus unabhängig von der eingesetzten Hardware zu beurteilen.
  - Hier: *Random Access Machine Modell* (dt. "Registermaschine")
  - Laufzeiten auf verschiedenen Computern unterscheiden sich eigentlich nur um einen konstanten Faktor.

Rechnermodell: Random Access Machine (RAM)

- ***Ziel:*** Analyse in Abhängigkeit der ***Eingabegröße***
  - *Meist*: Zahl der Eingabewerte, z.B. Sortieren von *n* Integer.
  - *Manchmal*: Anzahl an Bits, z.B. bei Multiplikation von 2 großen Zahlen

- ***Annahmen***
  - Instruktionen werden *sequentiell* abgearbeitet.
  - Typische PC-Instruktionen benötigen alle *ähnlich viel* (konstante) Zeit.
    - Addieren, Dividieren, Schiebeoperationen, etc.
  - Datentypen: *Integer*- und *Gleitkomma*
  - Eingabewerte nicht zu groß
    - Jeder Eingabewert passt in ein Register.
    - Sonst Aufwand, um große Werte auf mehrere Speicherorte zu verteilen.

Best, Worst und Average Case

- **Best Case**
  - Oft leicht zu bestimmen
  - Überlegen, welche Eingabe (= *Probleminstanz*) den Best Case darstellt.

- **Average Case**
  - Nicht leicht zu handhaben, für die Praxis jedoch relevant
  - Hier muss oft mit *Erwartungswerten* und *Wahrscheinlichkeiten* von Eingaben gerechnet werden.

- **Worst Case**
  - Meist leicht zu bestimmen.
  - Überlegen, welche Eingabe (= Probleminstanz) den Worst Case darstellt.
  - Fokus auf Worst Case, da
    - garantierte obere Schranke für Laufzeit → Landau-Notation, siehe später!
    - in Praxis: Average Case oft nicht "viel besser" als Worst Case.

Analyse von InsertionSort

- ***Annahme: RAM-Modell***
  - Jede Operation kostet gleich viel Zeit und hat Kosten 1

- ***Parameter***
  - ***n:*** Eingabegröße = Anzahl der zu sortierenden Elemente im Array A
  - c_i: Kosten für die Ausführung der **i.** Zeile
  - t_j**:** Gibt an, wie oft die `while`-Schleife in Zeile 5 für $j = 1,2,\ldots,n-1$ geprüft wird.
    - Unterschiede für Worst, Best und Average Case.

Kosten

$c_1 = n$

$c_2 = n - 1$

$c_4 = n - 1$

$c_5 = \sum_{j=1}^{n-1} t_j$

$c_6 = \sum_{j=1}^{n-1} (t_j - 1)$

$c_7 = \sum_{j=1}^{n-1} (t_j - 1)$

$c_8 = n - 1$

```
INSERTION-SORT(A)    // A: Input Array mit n Elementen
1    for j = 1 to A.length-1
2       key = A[j]
3       // insert A[j] into already sorted sequence
4       i = j - 1
5       while i ≥ 0 and A[i] > key
6          A[i+1] = A[i]
7          i = i - 1
8       A[i+1] = key
```

Laufzeit T(n) in Abhängigkeit der Eingabegröße *n*

- $T(n) = n + (n-1) + (n-1) + \sum_{j=1}^{n-1} t_j + \sum_{j=1}^{n-1}(t_j - 1) + \sum_{j=1}^{n-1}(t_j - 1) + (n-1)$
  - Laufzeit abhängig von t_j
- **Best Case**
  - Array ist bereits sortiert, alle t_j sind 1.
  - $T(n) = an + b$, mit *a* und *b* Konstanten
  - Die Laufzeit wächst **linear** mit der Eingabe *n*
- **Worst Case**
  - Array ist absteigend sortiert.
  - `key` muss in jeder Iteration mit j Elementen vergleichen werden + letzter Vergleich für Schleifenabbruch → $t_j = j$+1
  - $T(n) = an^2 + bn + c$ mit *a* und *b* Konstanten
  - Die Laufzeit wächst **quadratisch** mit der Eingabe *n*

- Fokus auf die Größenordnung (quadratisch, linear) genügt in der Regel!

Inhalt

- Algorithmus: Definition und Eigenschaften
- Beispiel 1: Pledge-Algorithmus
- Beispiel 2: Sortieren mit Insertion Sort
- Analyse von Algorithmen
- **Asymptotisches Wachstum**

Motivation

- Exakte Laufzeitanalysen mühsam
  - siehe Analyse InsertionSort.
- Relevant für Praxis:
  - Wachstum der Funktionen für ***sehr große*** Eingaben (=***asymptotisches*** Verhalten).
  - Grobe Aussagen, z.B.:
    - "Verdoppelung der Eingabegröße → Vervierfachung der Laufzeit."
  - Fokus auf "**Größenordnung**"
    - Keine niedrigwertigen Termine und Konstanten.
    - $T(n) = an^2 + bn + c \approx n^2$

Landau-Symbole:

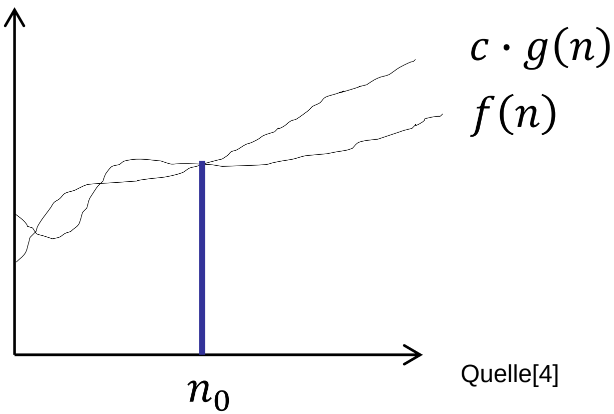
Beschreiben das Wachstum / Schranken für sehr große Eingabewerte! .

Bsp: $f(n) \in O(g(n))$ bedeutet, dass *f(n)* höchstens so schnell wächst (≤) wie *g(n)*.

- $O \approx \leq$
- $\Omega \approx \geq$
- $\Theta \approx =$
- $o \approx <$
- $\omega \approx >$

O-Notation

Definition: $\mathbf{f(n)} \in \boldsymbol{O(g(n))}$ → Es existieren positive **Konstanten** c und n_0,
so dass $0 \leq f(n) \leq c \cdot g(n)$ für alle n $\geq n_0$



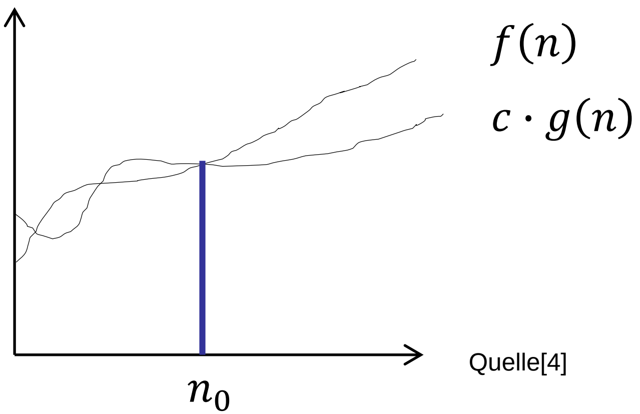
$c \cdot g(n)$ ist ab n_0 immer größer als $f(n)$

- $O(g(n))$ beschreibt eine ***Menge*** von Funktionen
  - Oft wird (unsauber) geschrieben: $f(n) = O(g(n))$
- *g(n)* ist ***asymptotische, obere Schranke*** für *f(n)*
  - *f(n)* wächst höchstens so schnell wie g(n)

- ***Übung:*** Gilt, dass $2n^2 \in O(n^3)$ bzw. $n^3 = O(n^2)$?
- Folgende Funktionen sind auch in $O(n^2)$: z.B. $n^2 + n$ oder $n^{1,99}$

Ω-Notation

Definition: $\mathbf{f}(\mathbf{n}) \in \boldsymbol{\Omega}(\boldsymbol{g}(\boldsymbol{n})) \rightarrow f(n)$: Es existieren positive Konstanten c und n_0, so dass $f(n) \geq c \cdot g(n)$ für alle $\mathrm{n} \geq n_0$



$c \cdot g(n)$ ist ab n_0 immer kleiner als $f(n)$

- $\Omega(g(n))$ beschreibt eine ***Menge*** von Funktionen
  - Oft wird (unsauber) geschrieben: $f(n) = \Omega(g(n))$
- *g(n)* ist eine ***asymptotische, untere Schranke*** für *f(n)*
  - *f(n)* wächst mindestens so schnell wie g(n).

Publikums-Joker

Welche Aussage ist ***richtig?***

A. $\sqrt{n} \in \Omega(\log_2 n)$

B. $\sqrt{n} \in O(\log_2 n)$

Funktionsplotter: https://www.mathe-fa.de/

Weitere Notationen

- **Θ-Notation**

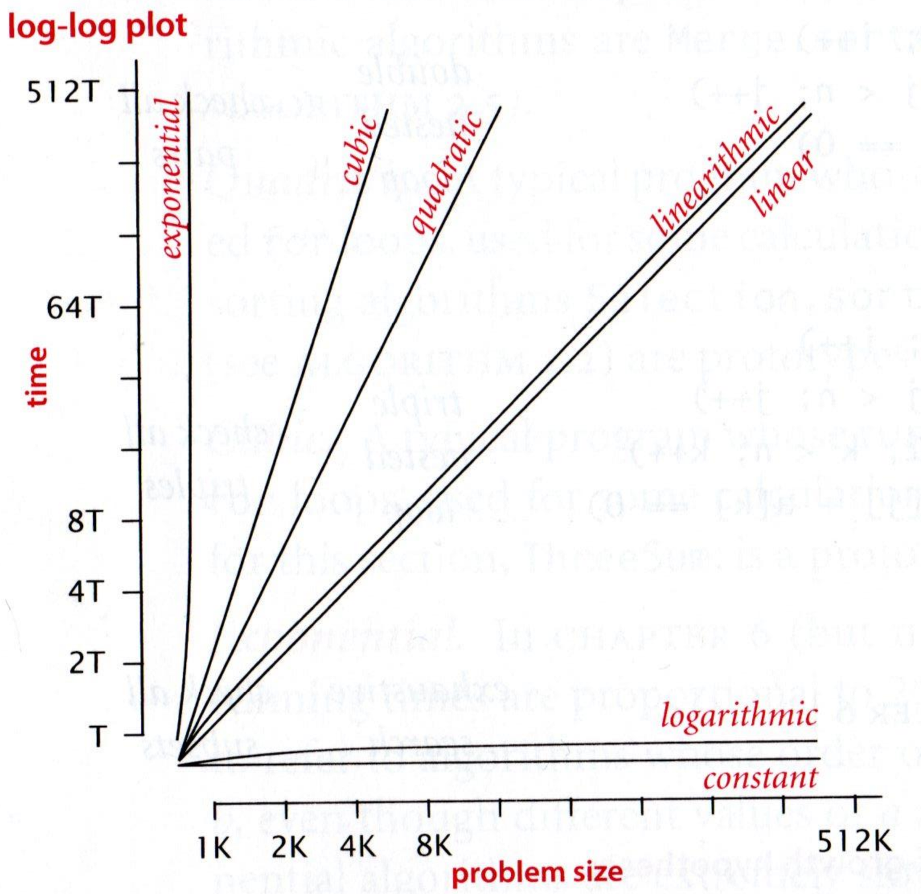
  Definition: $\mathbf{f(n) = \Theta(g(n))} \rightarrow f(n)$: Es existieren positive **Konstanten** c_1, c_2 und n_0, so dass $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ für alle $n \geq n_0$

  - Funktionen f(n) für die gilt: $f(n) \epsilon\ O(n)$ **und** $f(n) \epsilon\ \Omega(n)$.
  - Funktionen mit "gleichem asymptotischen Verhalten".
  - Beispiel: $\frac{n^2}{2} - n \ \in \Theta(n^2)$ (ohne Beweis)

- Für Vorlesung weniger relevant:
  - ***o*-Notation**
    - Identisch wie O-Notation, aber "<" statt "≤"
  - ***ω*-Notation**
    - Identisch wie Ω-Notation, aber ">" statt "≥"

Typische Größenordnungen

- Konstante Funktionen: **O(1)**
- Logarithmische Funktionen: **O(log n)**
  - Die Basis des Logarithmus spielt keine Rolle
- Lineare Funktionen: **O(n)**
- "Linearrithmic": **O(n log n)**
- Quadratische Funktionen: **O(n^2)**
- Polynomielle Funktionen: **O(n^k)**
- Exponentielle Funktionen: **O(2^n)**



Typische Größenordnungen, doppelt-logarithmische Darstellung
[5]

Inhalt

- Algorithmus: Definition und Eigenschaften
- Beispiel 1: Pledge-Algorithmus
- Beispiel 2: Sortieren mit Insertion Sort
- Analyse von Algorithmen
- Asymptotisches Wachstum

Quellenverzeichnis

[1] http://www.asterix.com/asterix-de-a-a-z/les-personnages/perso/g32b.gif (abgerufen am 22.09.16)

[2] Vöcking et al. *Taschenbuch der Algorithmen*, Springer Verlag, 2008 (eBook in der Bibliothek), Kapitel 8

[3] Geometry Lab, Universität Bonn, http://www.geometrylab.de/, (abgerufen am 23.09.16)

[4] Cormen, Leiserson, Rivest, Stein. Introduction to Algorithms, 3rd Edition, MIT Press, 2009

[5] Sedgewick, Wayne. Algorithms, 4th Edition, Addison-Wesley, 2011