Algorithmen und Datenstrukturen

TECHNISCHE UNIVERSITÄT DARMSTADT

vi visual inference SYSTEMS

Stefan Roth, SS 2025

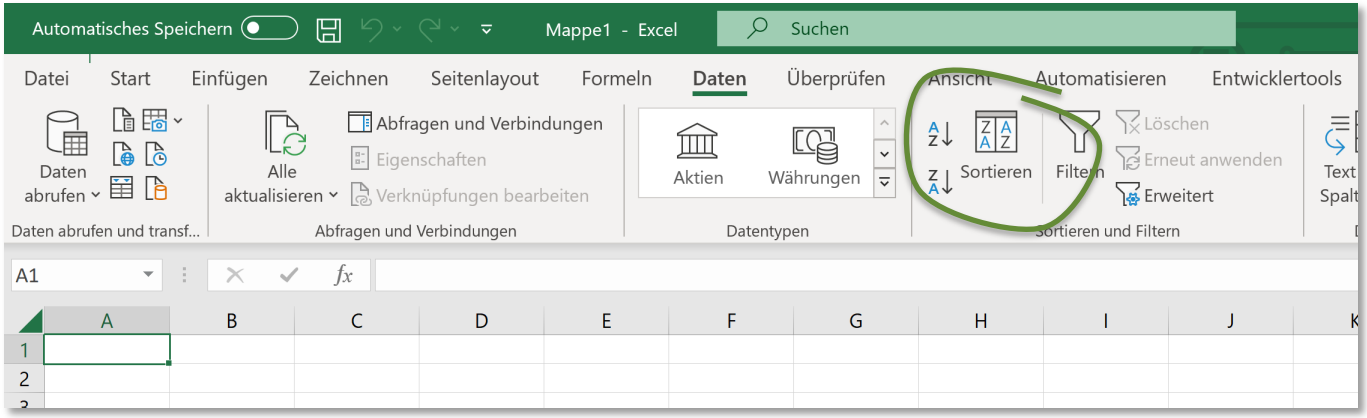
02

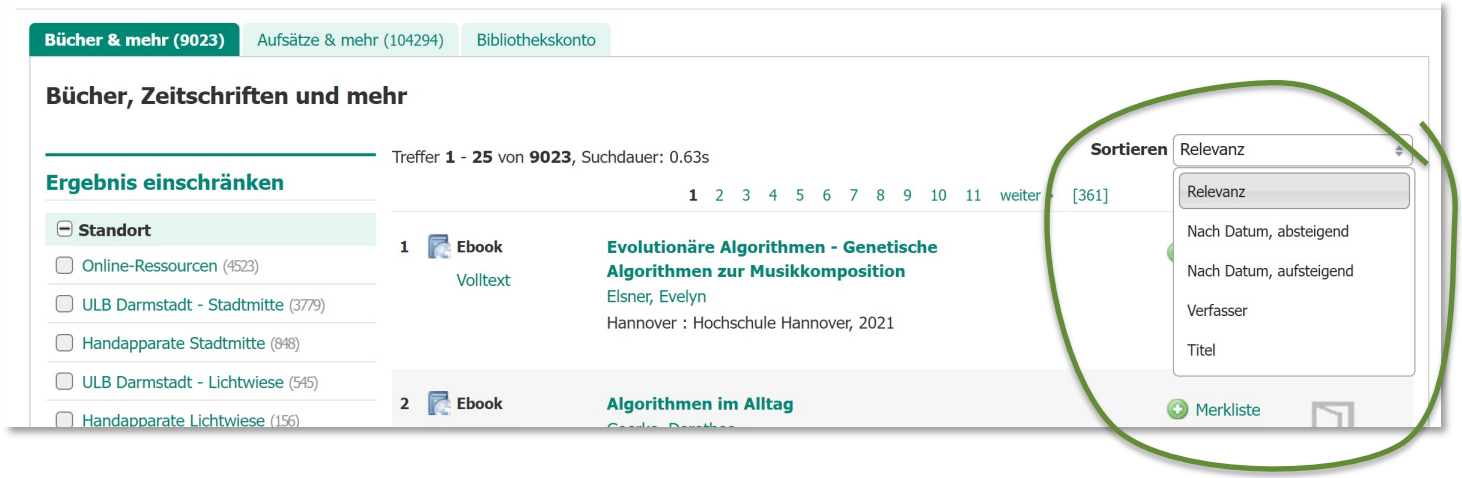
Sortieren

Folien beruhen auf der Veranstaltung von Prof. Marc Fischlin und Christian Janson aus dem SS 2024

Das Sortierproblem

Sortierproblem in der Praxis





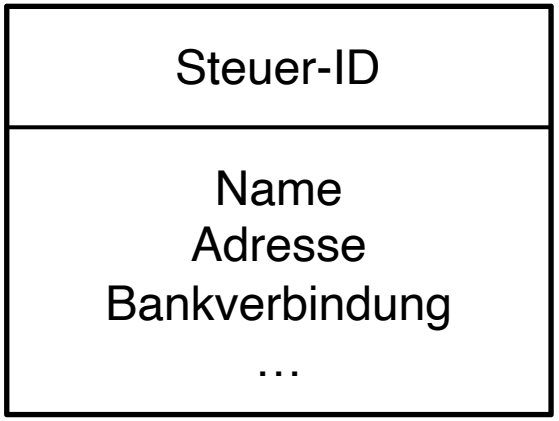
Sortierproblem

Gegeben: Folge von Objekten
Gesucht: Sortierung der Objekte
(gemäß ausgewiesenen Schlüsselwerts)

Wert, nach dem Objekt sortiert wird

Objekt

Beispiel



Schlüsselproblem

Schlüssel(werte) müssen nicht eindeutig sein,

aber müssen „sortierbar“ sein

Wert, nach dem Objekt sortiert wird

Objekt

Annahme im Folgenden:

Es gibt eine totale Ordnung $\leq$ auf der Menge M aller möglichen Schlüsselwerte

Totale Ordnung

Beispiel: lexikographische Ordnung auf Strings

Sei M eine nicht leere Menge und $\leq \subseteq M \times M$ eine binäre Relation auf M

Die Relation $\leq$ auf M ist genau dann eine totale Ordnung, wenn gilt:

Reflexivität: $\forall x \in M: \; x \leq x$

Transitivität: $\forall x, y, z \in M: \; x \leq y \; \wedge \; y \leq z \; \Rightarrow x \leq z$

Antisymmetrie: $\forall x, y \in M: \; x \leq y \; \wedge \; y \leq x \; \Rightarrow x = y$

Totalität: $\forall x, y \in M: \; x \leq y \; \vee \; y \leq x$

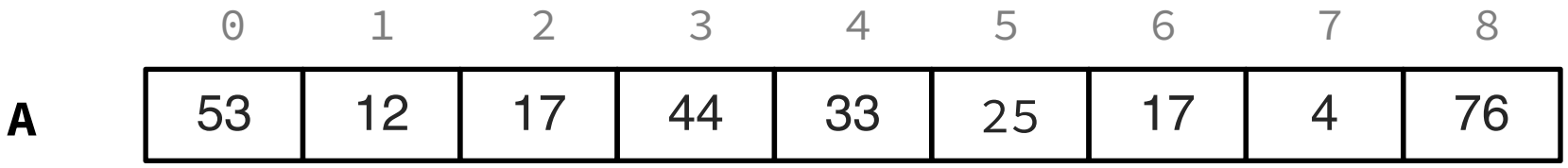
(ohne Totalität: partielle Ordnung)

Bemerkung: Totale Ordnung $\leq$ impliziert Relation $>$ durch $x > y :\Leftrightarrow x \not\leq y$

Darstellung in diesem Abschnitt

Wir betrachten nur Schlüssel(werte) ohne Satellitendaten,
in Beispielen meistens durch Zahlen gegeben

Eingabe der Objekte bzw. Schlüsselwerte in Form eines Arrays **`A`**:



Lese-/Schreibzugriff per Index in konstanter Zeit:

`y=A[2]` weist **`y`** den Wert 17 zu **`A[4]=99`** überschreibt 33 mit 99

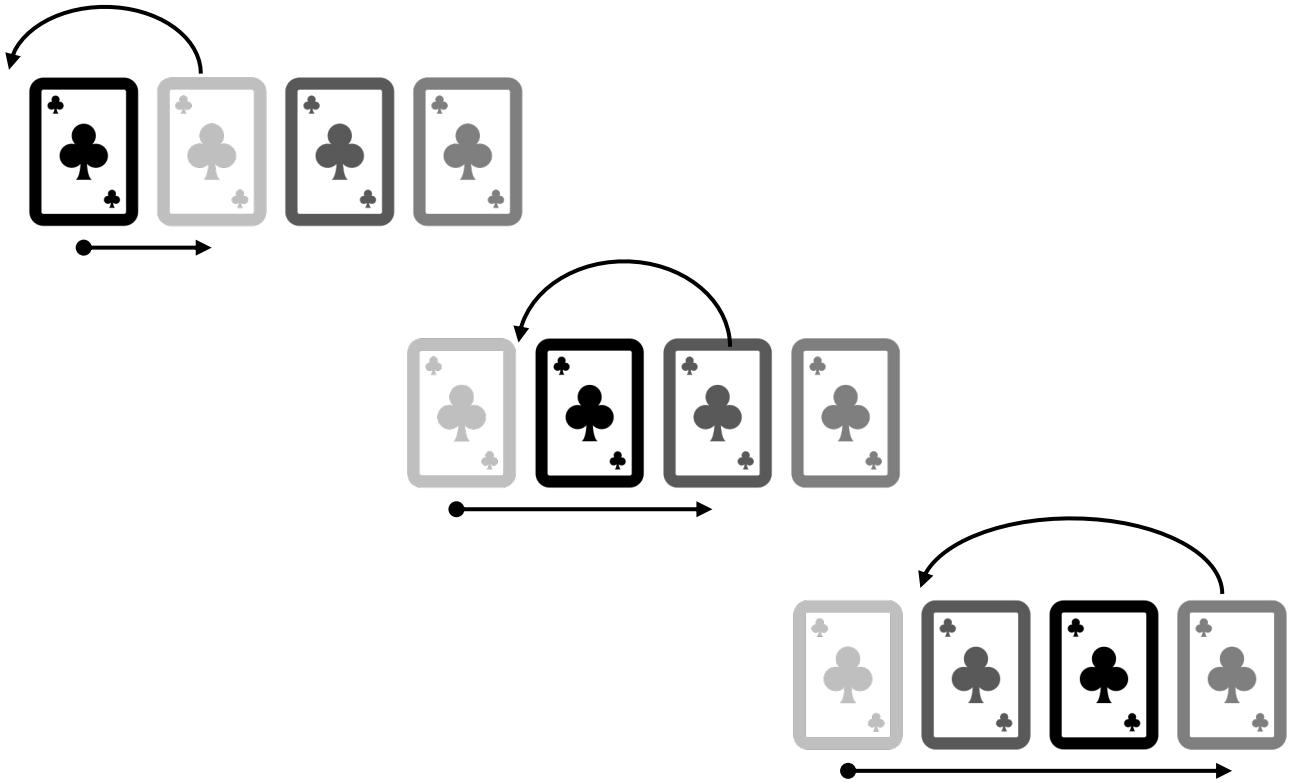
`A.length` beschreibt (fixe) Länge des Arrays, im Beispiel 9

`A[i…j]` beschreibt Teil-Array von Index **`i`** bis **`j`**, **`0<=i<=j<=A.length-1`**

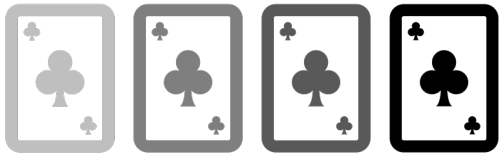
Insertion Sort

Idee: Karten umsortieren

(hell nach dunkel)



Gehe von links nach rechts durch
und sortiere aktuelle Karte richtig nach links ein



Algorithmus: Insertion Sort

Durch `!(A[j]=<key)` wohldefiniert

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
     // insert A[i] in pre-sorted sequence A[0...i-1]
2    key=A[i];
3    j=i-1; // search for insertion point backwards
4    WHILE j>=0 AND A[j]>key DO
5        A[j+1]=A[j]; // move elements to right
6        j=j-1;
7    A[j+1]=key;
```

Wir beginnen mit **`i=1`**, aber erstes Element ist **`A[0]`**

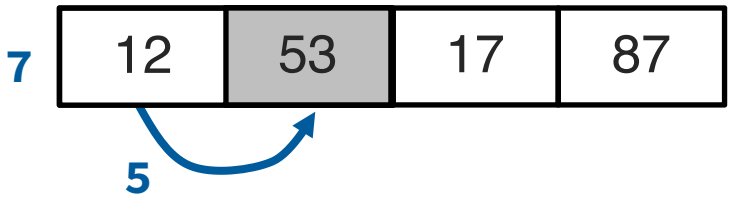
Short Circuit Evaluation (wie in Java):
Wenn erste AND-Bedingung **`false`**, wird zweite Bedingung nicht mehr ausgewertet

Beispiel: Insertion Sort (I)

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
     // insert A[i] in pre-sorted sequence A[0...i-1]
2    key=A[i];
3    j=i-1; // search for insertion point backwards
4    WHILE j>=0 AND A[j]>key DO
5        A[j+1]=A[j]; // move elements to right
6        j=j-1;
7    A[j+1]=key;
```

Beispiel: **FOR-Schleife i=1** **2 key=12** **3 j=0**

WHILE-Schleife j=0 **A[j]=53 > key=12**



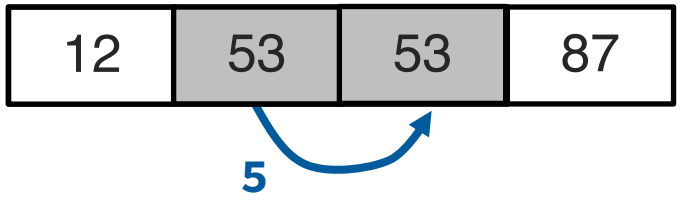
1. Fall:
linker Rand erreicht

Beispiel: Insertion Sort (II)

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
     // insert A[i] in pre-sorted sequence A[0...i-1]
2    key=A[i];
3    j=i-1; // search for insertion point backwards
4    WHILE j>=0 AND A[j]>key DO
5        A[j+1]=A[j]; // move elements to right
6        j=j-1;
7    A[j+1]=key;
```

Beispiel: **FOR-Schleife i=2** **2 key=17** **3 j=1**

WHILE-Schleife j=1 **A[j]=53 > key=17**



6 j=0

2. Fall:
Einfügeposition erreicht

Beispiel: Insertion Sort (III)

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
     // insert A[i] in pre-sorted sequence A[0...i-1]
2    key=A[i];
3    j=i-1; // search for insertion point backwards
4    WHILE j>=0 AND A[j]>key DO
5        A[j+1]=A[j]; // move elements to right
6        j=j-1;
7    A[j+1]=key;
```

Beispiel: **FOR-Schleife i=2** **2 key=17** **3 j=1**

WHILE-Schleife j=0 **A[j]=12 < key=17**

7

6 j=0

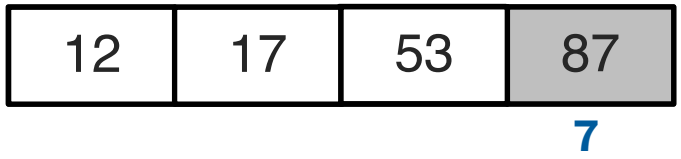
2. Fall:
Einfügeposition erreicht

Beispiel: Insertion Sort (IV)

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
     // insert A[i] in pre-sorted sequence A[0...i-1]
2    key=A[i];
3    j=i-1; // search for insertion point backwards
4    WHILE j>=0 AND A[j]>key DO
5        A[j+1]=A[j]; // move elements to right
6        j=j-1;
7    A[j+1]=key;
```

Beispiel: **FOR-Schleife i=3** **2 key=87** **3 j=2**

WHILE-Schleife j=2 **A[j]=53 < key=87**



3. Fall:
Element bleibt

Terminierung

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
     // insert A[i] in pre-sorted sequence A[0...i-1]
2    key=A[i];
3    j=i-1; // search for insertion point backwards
4    WHILE j>=0 AND A[j]>key DO
5        A[j+1]=A[j]; // move elements to right
6        j=j-1;
7    A[j+1]=key;
```

Jede Ausführung der **`WHILE`**-Schleife erniedrigt **`j<n`** in jeder Iteration um 1 und bricht ab, wenn **`j<0`** → terminiert also immer

`FOR`-Schleife wird nur endlich oft durchlaufen

Korrektheit (I)

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
     // insert A[i] in pre-sorted sequence A[0...i-1]
2    key=A[i];
3    j=i-1; // search for insertion point backwards
4    WHILE j>=0 AND A[j]>key DO
5        A[j+1]=A[j]; // move elements to right
6        j=j-1;
7    A[j+1]=key;
```

Schleifeninvariante (der **FOR**-Schleife):

Bei jedem Eintritt für Zählerwert **i** (und nach letzter Ausführung) entsprechen die aktuellen Einträge in **A[0],…,A[i-1]** den sortierten ursprünglichen Eingabewerten **a[0],…,a[i-1]**. Ferner gilt **A[i]=a[i],…, A[n-1]=a[n-1]**.

Beweis per vollständiger Induktion

Klassische **mathematische Beweismethode** für Aussagen $\mathrm{A}(n)$, die für **alle natürliche Zahlen** $n \in \mathbb{N}$ gelten, die größer oder gleich einem **bestimmten Startwert** n_0 sind. n_0 ist häufig 0 oder 1.

Behauptung:	Aussage $\mathrm{A}(n)$ gilt $\forall n \in \mathbb{N}, n \geq n_0 \in \mathbb{N}$.
Beweisstruktur:	
Induktionsanfang:	Zeige, dass Aussage $\mathrm{A}(n_0)$ für Startwert n_0 gilt.
Induktionsschritt:	Zeige, dass wenn $\mathrm{A}(n)$ für ein beliebiges $n \in \mathbb{N}, n \geq n_0$ gilt *(Induktionsannahme)*, auch $\mathrm{A}(n+1)$ gelten muss *(Induktionsbehauptung)*.
Induktionsschluss:*	Aus Induktionsanfang und Induktionsschritt folgt die Behauptung.

* Wird der Vereinfachung halber oft weggelassen

Beweis per vollständiger Induktion – Beispiel I

Gauß'sche Summenformel

Behauptung: Aussage $A(n)$: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ gilt $\forall n \in \mathbb{N}$.

Beweis:

Induktionsanfang: $n = 1$:

$$\sum_{i=1}^{n} i = \sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2} = \frac{n(n+1)}{2}$$

Induktionsschritt: $n \to n+1$:

Induktionsannahme

$$\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + (n+1) = \frac{n(n+1)}{2} + (n+1) = \frac{n(n+1) + 2(n+1)}{2}$$

$$= \frac{(n+2)(n+1)}{2} = \frac{(n+1)\big((n+1)+1\big)}{2}$$

■

Beweis per vollständiger Induktion – Beispiel II

Bernoulli'sche Ungleichung

Behauptung: Aussage $A(n)$: $(1+x)^n \geq 1+nx$
gilt $\forall n \in \mathbb{N}_0$ und $\forall x \in \mathbb{R}, x > -1$.

Beweis:

Induktionsanfang: $n = 0$: $(1+x)^n = (1+x)^0 = 1 = 1 + 0x = 1 + nx$

Induktionsschritt: $n \to n+1$:

Induktionsannahme

$$(1+x)^{(n+1)} = (1+x)^n \cdot (1+x) \geq (1+nx) \cdot (1+x) = 1 + nx + x + nx^2$$

$$= 1 + (n+1)x + nx^2 \geq 1 + (n+1)x \quad \blacksquare$$

≥ 0

Korrektheit (II)

Beweis per Induktion:

Induktionsbasis **`i=1`**: Beim ersten Eintritt ist **`A[0]=a[0]`** und „sortiert“.
Ferner gilt noch **`A[1]=a[1],…,A[n-1]=a[n-1]`**.

Bei jedem Eintritt für Zählerwert **`i`** (und nach letzter Ausführung) entsprechen die aktuellen Einträge in **`A[0],…,A[i-1]`** den sortierten ursprünglichen Eingabewerten **`a[0],…,a[i-1]`**.
Ferner gilt **`A[i]=a[i],…, A[n-1]=a[n-1]`**.

Korrektheit (III)

Beweis per Induktion: Induktionsschritt von **`i-1`** auf **`i`**:

Vor der (**`i-1`**)–ten Ausführung galt Schleifeninvariante nach Voraussetzung. Inbesondere war **`A[0],…,A[i-2]`** sortierte Version von **`a[0],…,a[i-2]`** und **`A[i-1]=a[i-1],…,A[n-1]=a[n-1]`**

Durch die **`WHILE`**-Schleife wurde **`A[i-1]=a[i-1]`** nach links einsortiert und größere Elemente von **`A[0],…,A[i-2]`** um jeweils eine Position nach rechts verschoben. Elemente **`A[i],…,A[n-1]`** wurden nicht geändert

Also gilt Invariante auch für **`i`**

> Bei jedem Eintritt für Zählerwert **`i`** (und nach letzter Ausführung) entsprechen die aktuellen Einträge in **`A[0],…,A[i-1]`** den sortierten ursprünglichen Eingabewerten **`a[0],…,a[i-1]`**. Ferner gilt **`A[i]=a[i],…, A[n-1]=a[n-1]`**.

Korrektheit (IV)

Aus Schleifeninvariante folgt für letzte Ausführung
(also quasi vor gedanklichem Eintritt der Schleife für **`i=n`**):

`A[0],…,A[n-1]` ist sortierte Version von **`a[0],…,a[n-1]`**

und somit am Ende das Array sortiert

Bei jedem Eintritt für Zählerwert **`i`** (und nach letzter Ausführung) entsprechen die aktuellen Einträge in **`A[0],…,A[i-1]`** den sortierten ursprünglichen Eingabewerten **`a[0],…,a[i-1]`**.
Ferner gilt **`A[i]=a[i],…, A[n-1]=a[n-1]`**.

Wozu brauchen Sie (intuitiv) beim Sortieren die vier Eigenschaften einer totalen Ordnung?

Überlegen Sie sich, dass Insertion Sort **stabil** ist, d.h. die Reihenfolge von Objekten mit gleichem Schlüssel bleibt erhalten.

Laufzeitanalysen: $\mathcal{O}$-Notation

Laufzeitanalyse

Wieviel Schritte macht Algorithmus
in Abhängigkeit von der Eingabekomplexität?

meistens: schlechtester Fall über
alle Eingaben gleicher Komplexität

(Worst-Case-)Laufzeit
$T(n)$ = max { Anzahl Schritte für x }

Maximum über alle Eingaben
x der Komplexität n

fasst alle Eingaben
ähnlicher Komplexität zusammen

Beispiel: n Anzahl zu
sortierender Zahlen

(man könnte auch zusätzlich
Größe der Zahlen betrachten;
wird aber meist von Anzahl dominiert)

Laufzeitanalyse Insertion Sort (I)

n Anzahl zu sortierender Elemente

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
     // insert A[i] in pre-sorted sequence A[0..i-1]
2    key=A[i];
3    j=i-1; // search for insertion point backwards
4    WHILE j>=0 AND A[j]>key DO
5        A[j+1]=A[j]; // move elements to right
6        j=j-1;
7    A[j+1]=key;
```

Analysiere, wie oft jede Zeile **maximal** ausgeführt wird

Jede Zeile i hat Aufwand ci

Laufzeitanalyse Insertion Sort (II)

n Anzahl zu sortierender Elemente

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
     // insert A[i] in pre-sorted sequence A
2    key=A[i];
3    j=i-1; // search for insertion point ba
4    WHILE j>=0 AND A[j]>key DO
5        A[j+1]=A[j]; // move elements to r
6        j=j-1;
7    A[j+1]=key;
```

Zeile	Aufwand	Anzahl
1	c1	n
2	c2	$n-1$
3	c3	$n-1$
4	c4	**Z5**$+n-1$
5	c5	$n(n-1)/2$
6	c6	$n(n-1)/2$
7	c7	$n-1$

Zeilen **4**, **5** und **6** im schlimmsten Fall bis **j=-1** also jeweils **i**-mal. Insgesamt:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

(Zeile **4** jeweils einmal mehr bis Abbruch)

Analysiere, wie oft jede Zeile **maximal** ausgeführt wird

Jede Zeile i hat Aufwand ci

Laufzeitanalyse Insertion Sort (III)

n Anzahl zu sortierender Elemente

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
     // insert A[i] in pre-sorted sequence A
2    key=A[i];
3    j=i-1; // search for insertion point ba
4    WHILE j>=0 AND A[j]>key DO
5        A[j+1]=A[j]; // move elements to r
6        j=j-1;
7    A[j+1]=key;
```

Zeile	Aufwand	Anzahl
1	c1	n
2	c2	$n-1$
3	c3	$n-1$
4	c4	**Z5**$+n-1$
5	c5	$n(n-1)/2$
6	c6	$n(n-1)/2$
7	c7	$n-1$

maximale Gesamtlaufzeit Insertion-Sort:

$$T(n) = c1 \cdot n + (c2 + c3 + c4 + c7) \cdot (n-1) + (c4 + c5 + c6) \cdot \frac{n(n-1)}{2}$$

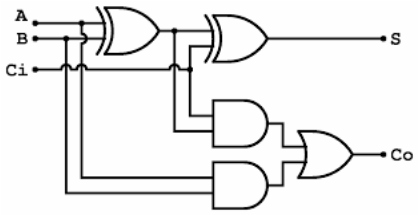
Analysiere, wie oft jede Zeile **maximal** ausgeführt wird

Jede Zeile i hat Aufwand ci

Kosten für individuelle Schritte?

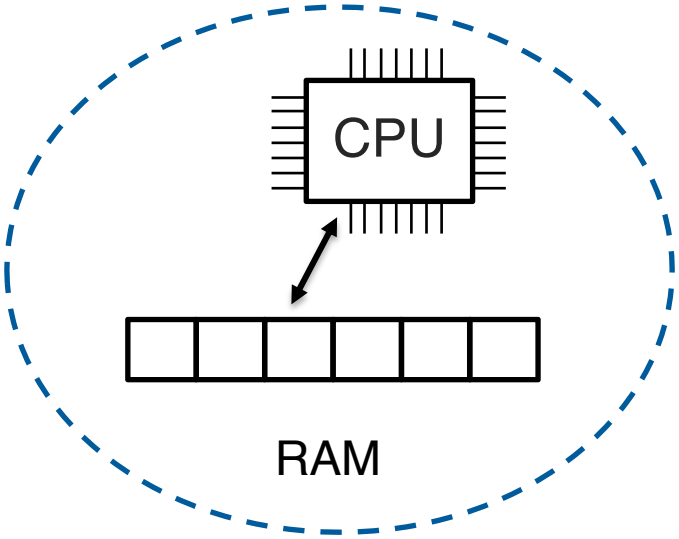
Wie teuer ist z.B. Zuweisung **`A[j+1]=A[j]`** in Zeile **`5`**, also was ist $c5$?

Hängt stark von Berechnungsmodell ab
(in dem Pseudocode-Algorithmus umgesetzt wird)…

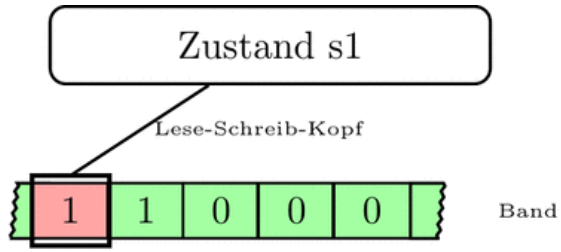


Quelle: ke.tu-darmstadt.de

Schaltkreis



RAM



Quelle: de.wikipedia.org/wiki/Turingmaschine

Turingmaschine

nehmen üblicherweise an, dass elementare Operationen (Zuweisung, Vergleich,…) in einem Schritt möglich → $c5 = 1$

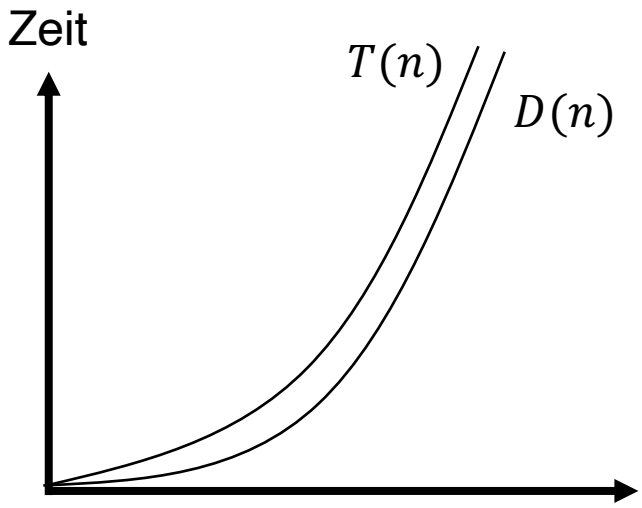
Asymptotische Vereinfachung (I)

Gesamtlaufzeit Insertion-Sort (mit $ci = 1$):

$$T(n) = n + 4 \cdot (n-1) + 3 \cdot \frac{n(n-1)}{2}$$

Zum Vergleich (nur dominanter Term) :

$$D(n) = 3 \cdot \frac{n(n-1)}{2}$$



n	$T(n)$	$D(n)$	**relativer Fehler** $(T(n) - D(n))/T(n)$
100	15.346	14.850	3,2321 %
1.000	1.503.496	1.498.500	0,3323 %
10.000	150.034.996	149.985.000	0,0333 %
100.000	15.000.349.996	14.999.850.000	0,0033 %
1.000.000	150.000.034.999.996	1.499.998.500.000	0,0003 %

Asymptotische Vereinfachung (II)

Weiter Vereinfachung (nur abhängiger Term) :

$$A(n) = \frac{n(n-1)}{2}$$

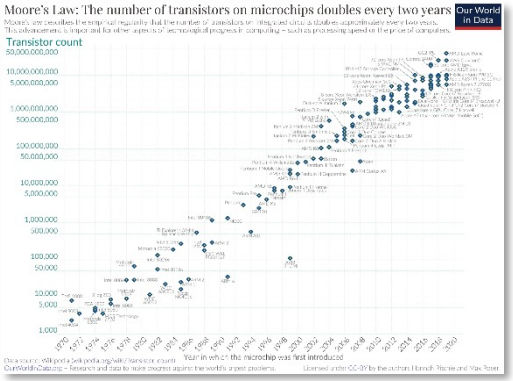
Zum Vergleich (nur dominanter Term) :

$$D(n) = 3 \cdot \frac{n(n-1)}{2}$$

Konstante hängt stark vom Berechnungsmodell ab

Konstante ändert sich „schnell“ durch Fortschritte in Rechenleistung

Beispiel Moore‘s Law (bis ca. 2000):
Verdoppelung der Transistoren etwa alle 1,5 Jahre



Quelle: **de.wikipedia.org/wiki/Mooresches_Gesetz**

Θ-Notation / Landau-Symbole

Paul Bachmann
Edmund Landau
ca. 1900

Funktionen $f, g\colon \mathbb{N} \longrightarrow \mathbb{R}_{>0}$

Eingabekomplexität — Laufzeit

$$\Theta(g) = \{f : \exists\, c_1, c_2 \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Funktion f

Positive Konstanten

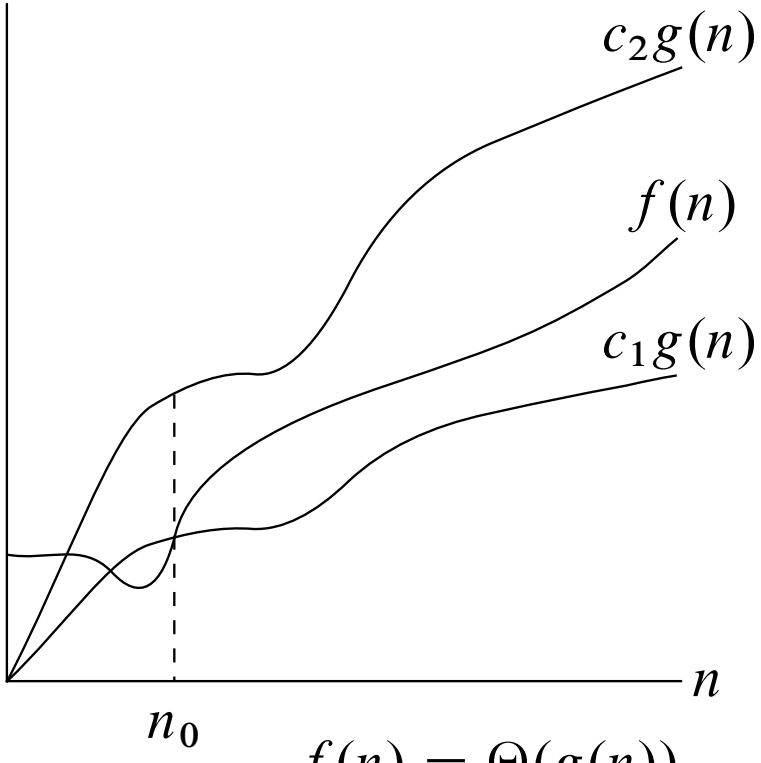
Für alle n größer gleich n_0

$f(n)$ wird von $c_1 g(n)$ und $c_2 g(n)$ für hinreichend große n eingeschlossen

Schreibweise: $f \in \Theta(g)$, manchmal auch $f = \Theta(g)$

Veranschaulichung Θ-Notation

Quelle: Corman et al. Introduction to Algorithms



$f(n) = \Theta(g(n))$

$n \geq n_0$:

$c_1 g(n) \leq f(n)$

$c_2 g(n) \geq f(n)$

$g(n)$ ist eine asymptotisch scharfe Schranke von $f(n)$

Θ-Notation beschränkt eine Funktion asymptotisch von oben und unten

Beispiel: Laufzeit Insertion Sort in Θ-Notation (I)

$$T(n) = n + 4 \cdot (n-1) + 3 \cdot \frac{n(n-1)}{2} \quad = \Theta(n^2)$$

Für untere Schranke wähle $c_1 = \frac{3}{2}$ und $n_0 = 2$:

$$\begin{aligned} T(n) \geq \quad & 5 \cdot (n-1) \quad + \frac{3}{2} \cdot n^2 \quad - \frac{3}{2} \cdot n \\ \geq \quad & \frac{7}{2} \cdot n - 5 \quad + \frac{3}{2} \cdot n^2 \\ \geq \quad & \frac{3}{2} \cdot n^2 \qquad \text{für } n \geq 2 \end{aligned}$$

$$\Theta(g) = \{f : \exists\, c_1, c_2 \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Beispiel: Laufzeit Insertion Sort in Θ-Notation (II)

$$T(n) = n + 4 \cdot (n-1) + 3 \cdot \frac{n(n-1)}{2} \quad = \Theta(n^2)$$

für $c_1 = \frac{3}{2}$, $c_2 = 7, n_0 = 2$

Für obere Schranke wähle $c_2 = 7$ und das bereits fixierte $n_0 = 2$:

$$\begin{aligned} T(n) &\leq n + \quad 4 \cdot n \quad + 2 \cdot n(n-1) \\ &\leq \quad 5 \cdot n \quad + 2 \cdot n^2 \\ &\leq \quad 5 \cdot n^2 \quad + 2 \cdot n^2 \\ &\leq \quad 7 \cdot n^2 \end{aligned}$$

$$\Theta(g) = \{f : \exists\, c_1, c_2 \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

$\mathcal{O}$-Notation

Obere asymptotische Schranke

$$\mathcal{O}(g) = \{f : \exists\, c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$$

Funktion f

Positive Konstanten

Für alle n größer gleich n_0

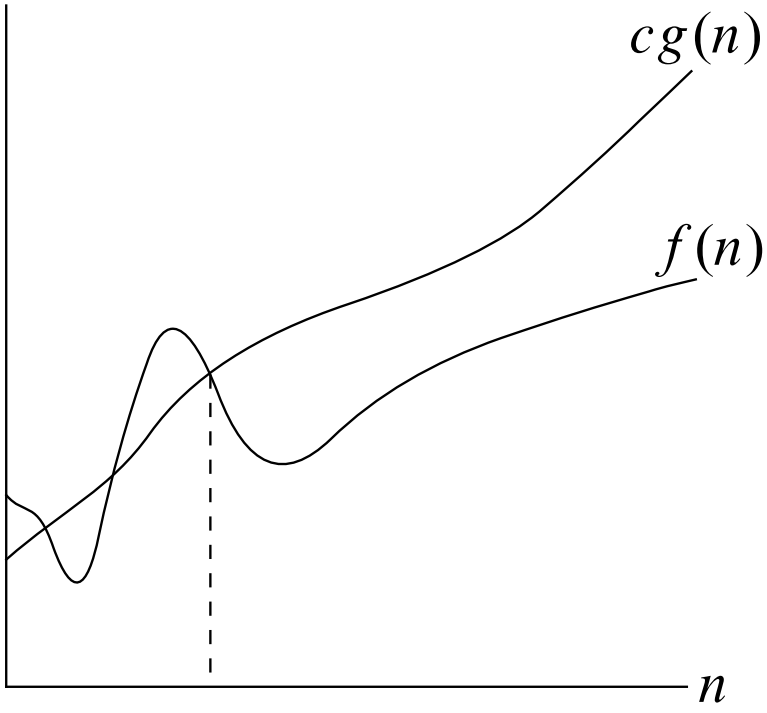
$f(n)$ wird von $cg(n)$ für hinreichend große n von oben beschränkt

Sprechweise: f wächst höchstens so schnell wie g

Schreibweise: $f = \mathcal{O}(g)$ oder auch $f \epsilon\, \mathcal{O}(g)$

Veranschaulichung der $\mathcal{O}$-Notation

Quelle: Corman et al. Introduction to Algorithms



$f(n) = O(g(n))$

$n \geq n_0$:

$0 \leq f(n) \leq cg(n)$

Beachte: $\Theta(g(n)) \subseteq \mathcal{O}(g(n))$ und somit $f(n) = \Theta(g) \Rightarrow f(n) = \mathcal{O}(g)$

$\mathcal{O}$-Notation: Rechenregeln

Konstanten: $f(n) = a$ mit $a \in \mathbb{R}_{>0}$ konstant. Dann $f(n) = \mathcal{O}(1)$

Skalare Multiplikation: $f = \mathcal{O}(g)$ und $a \in \mathbb{R}_{>0}$. Dann $a \cdot f = \mathcal{O}(g)$

Addition: $f_1 = \mathcal{O}(g_1)$ und $f_2 = \mathcal{O}(g_2)$. Dann $f_1 + f_2 = \mathcal{O}(\max\{g_1, g_2\})$

punktweise

Multiplikation: $f_1 = \mathcal{O}(g_1)$ und $f_2 = \mathcal{O}(g_2)$. Dann $f_1 \cdot f_2 = \mathcal{O}(g_1 \cdot g_2)$

Ω-Notation

Untere asymptotische Schranke

$$\Omega(g) = \{f : \exists\, c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$$

Funktion f

Positive Konstanten

Für alle n größer gleich n_0

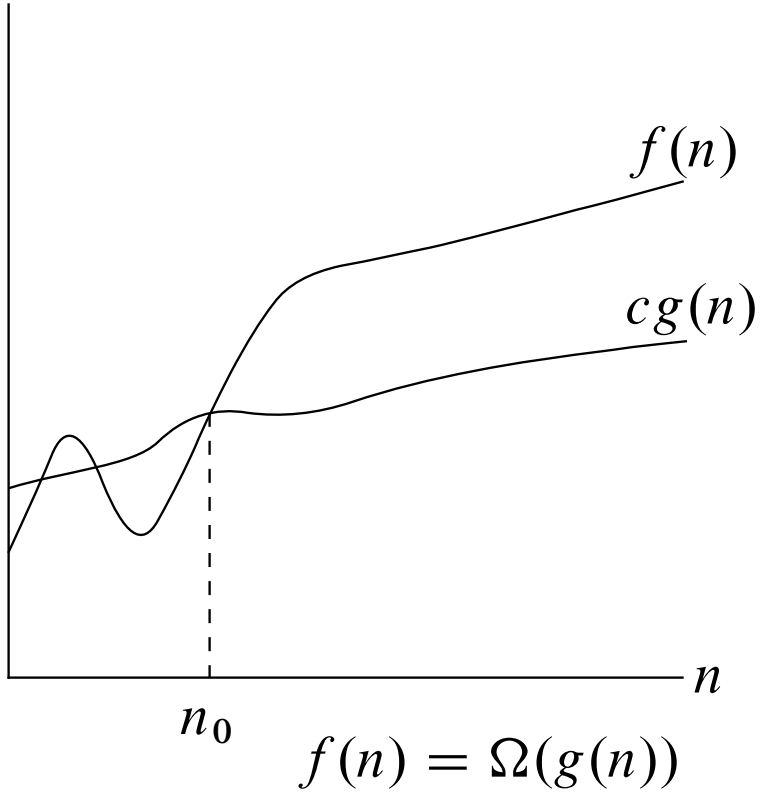
$f(n)$ wird von $cg(n)$ für hinreichend große n von unten beschränkt

Sprechweise: f wächst mindestens so schnell wie g

Schreibweise: $f = \Omega(g)$ oder auch $f \in \Omega(g)$

Veranschaulichung der Ω-Notation

Quelle: Corman et al. Introduction to Algorithms



$f(n) = \Omega(g(n))$

$n \geq n_0$:

$0 \leq cg(n) \leq f(n)$

Beachte: $\Theta(g(n)) \subseteq \Omega(g(n))$ und somit $f(n) = \Theta(g) \Longrightarrow f(n) = \Omega(g)$

Überlegen Sie sich die Rechenregeln für Ω analog zu denen für die $\mathcal{O}$-Notation.

Überlegen Sie sich, dass gilt:
$\mathcal{O}(n) \subseteq \mathcal{O}(n^2) \subseteq \mathcal{O}(n^3) \subseteq \mathcal{O}(n^4) \subseteq \mathcal{O}(n^5) \subseteq \ldots$
und
$\Omega(n) \supseteq \Omega(n^2) \supseteq \Omega(n^3) \supseteq \Omega(n^4) \supseteq \Omega(n^5) \supseteq \ldots$
und dass die Inklusionen jeweils strikt sind

Zusammenhang $\mathcal{O}, \Omega$ und Θ

Für beliebige $f(n), g(n): \mathbb{N} \longrightarrow \mathbb{R}_{>0}$ gilt:

$$f(n) = \Theta(g(n))$$
genau dann, wenn
$f(n) = \mathcal{O}(g(n))$ und $f(n) = \Omega(g(n))$

Beachte: $\Omega(g), \mathcal{O}(g)$ sind nur untere bzw. obere Schranken:

Beispiel: $f(n) = \Theta(n^3)$, also auch:

$$f(n) = \mathcal{O}(n^5), \text{ da } \Theta(n^3) \subseteq \mathcal{O}(n^3) \subseteq \mathcal{O}(n^5)$$

$$f(n) = \Omega(n), \text{ da } \Theta(n^3) \subseteq \Omega(n^3) \subseteq \Omega(n)$$

Anwendung $\mathcal{O}$-Notation (I)

$f = \mathcal{O}(g)$ übliche Schreibweise, sollte aber gelesen werden als $f \in \mathcal{O}(g)$

Allgemein besser immer als Mengen auffassen
und von links nach rechts lesen (mit $\in, \subseteq$):

$$5 \cdot n^2 + n^4 = \mathcal{O}(n^2) + n^4 = \mathcal{O}(n^4) = \mathcal{O}(n^5)$$

$\in$ $\subseteq$ $\subseteq$

als Menge

$$\{f(n)\} + n^4 = \{\, f(n) + n^4 \,\}$$

nicht: $\mathcal{O}(n^4) = \mathcal{O}(n^5)$, und damit auch $\mathcal{O}(n^5) = \mathcal{O}(n^4)$

wird mit Mengenschreibweise klarer: $A \subseteq B$ bedeutet allgemein nicht auch $B \subseteq A$

Anwendung $\mathcal{O}$-Notation (II)

Ungleichungen mit $\leq$ sollten nur mit $\mathcal{O}$ verwendet werden,
Ungleichungen mit $\geq$ sollten nur mit Ω verwendet werden.

$$5 \cdot n^2 + n^4 \leq 6 \cdot n^4 = \mathcal{O}(n^4)$$

es gibt c, n_0 und Funktion $f(n)$ mit $6 \cdot n^4 \leq c \cdot f(n)$

obere Schranke vs. untere Schranke

nicht: $5 \cdot n^2 + n^4 \leq 6 \cdot n^4 = \Omega(n^4)$

$\mathcal{O}, \Omega$ und Θ bei Insertion Sort (I)

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
     // insert A[i] in pre-sorted sequence A[0..i-1]
2    key=A[i];
3    j=i-1; // search for insertion point backwards
4    WHILE j>=0 AND A[j]>key DO
5        A[j+1]=A[j]; // move elements to right
6        j=j-1;
7    A[j+1]=key;
```

Algorithmus macht maximal $T(n)$ viele Schritte und $T(n) = \Theta(n^2)$

Also Laufzeit Insertion Sort $= \Theta(n^2)$?

korrekte Anwendung: Laufzeit $\leq T(n) = \mathcal{O}(n^2)$

$\mathcal{O}, \Omega$ und Θ bei Insertion Sort (II)

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
     // insert A[i] in pre-sorted sequence A[0..i-1]
2    key=A[i];
3    j=i-1; // search for insertion point backwards
4    WHILE j>=0 AND A[j]>key DO
5        A[j+1]=A[j]; // move elements to right
6        j=j-1;
7    A[j+1]=key;
```

zur Erinnerung: (Worst-Case-)Laufzeit $T(n)$ = max { Anzahl Schritte für x }

Für untere Schranke muss man „nur“
eine schlechte bzw. die schlechteste Eingabe x finden

Dann gilt $T(n) \geq$ Anzahl Schritte für schlechtes x

$\mathcal{O}, \Omega$ und Θ bei Insertion Sort (III)

Insertion Sort hat quadratische Laufzeit $\Theta(n^2)$

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
     // insert A[i] in pre-sorted sequence A[0..i-1]
2    key=A[i];
3    j=i-1; // search for insertion point backwards
4    WHILE j>=0 AND A[j]>key DO
5        A[j+1]=A[j]; // move elements to right
6        j=j-1;
7    A[j+1]=key;
```

„Schlechte“ Eingabe:

A

n	$n-1$	$n-2$		…	…		2	1

Jede **WHILE**-Schleifenausführung für $i = 1, \ldots, n-1$ macht jeweils i Iterationen

Insgesamt macht Algorithmus für **A** also $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Omega(n^2)$ Schritte

„Gute“ Eingaben für Insertion Sort?

```
insertionSort(A)
1  FOR i=1 TO A.length-1 DO
     // insert A[i] in pre-sorted sequence A[0..i-1]
2    key=A[i];
3    j=i-1; // search for insertion point backwards
4    WHILE j>=0 AND A[j]>key DO
5        A[j+1]=A[j]; // move elements to right
6        j=j-1;
7    A[j+1]=key;
```

„gute“ Eingaben bereits vorsortiert, extremes Beispiel:

A

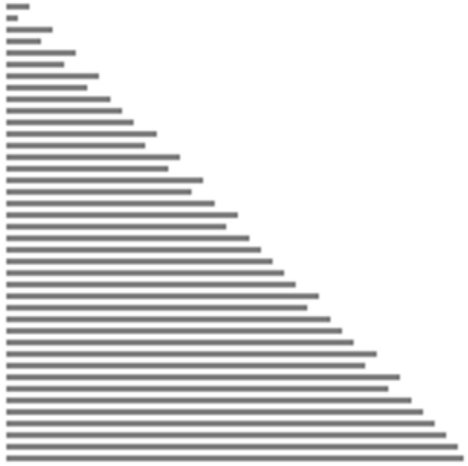
1	2	3		…	…		$n-2$	$n-1$

`WHILE`-Schleife wird für dieses **`A`** nie ausgeführt

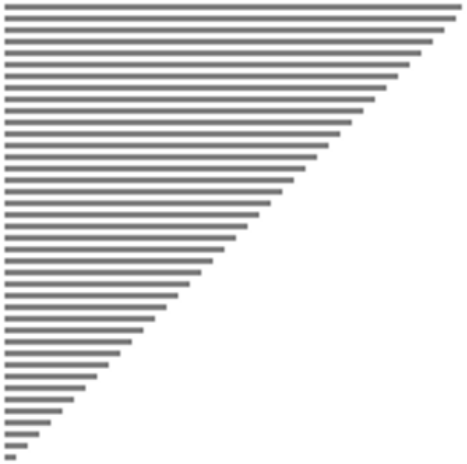
Insgesamt macht Algorithmus für dieses **`A`** also $\Theta(n)$ Schritte

Laufzeit Insertion Sort (I)

„guter“ Fall
(fast vorsortiertes Array)



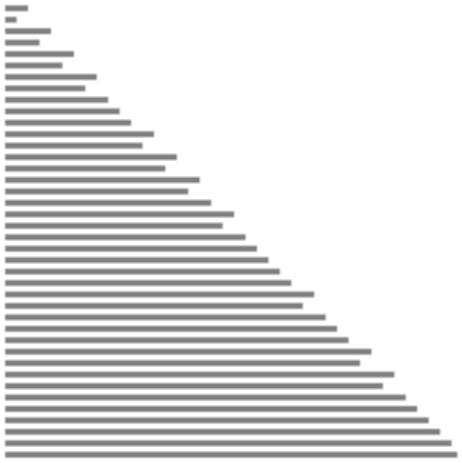
„schlechter“ Fall
(invertiertes, vorsortiertes Array)



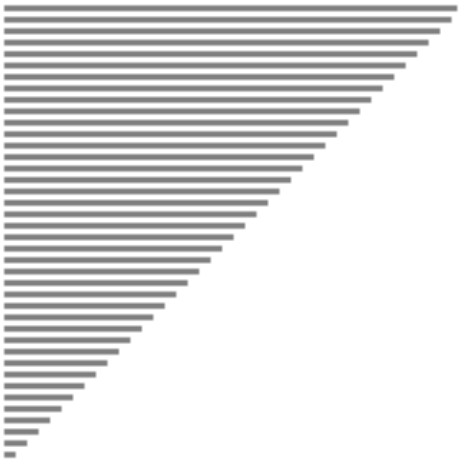
Quelle: **https://web.archive.org/web/20150302064244/http://www.sorting-algorithms.com/insertion-sort**

Laufzeit Insertion Sort (II)

„guter“ Fall
(fast vorsortiertes Array)



„schlechter“ Fall
(invertiertes, vorsortiertes Array)



Quelle: **https://web.archive.org/web/20150302064244/http://www.sorting-algorithms.com/insertion-sort**

Wort-Case-Laufzeiten:
auch wenn für manche Eingaben schneller, gilt:

Insertion Sort hat quadratische Laufzeit $\Theta(n^2)$

Komplexitätsklassen

n ist die Länge der Eingabe (z.B. Arraylänge, Länge des Strings)

Klasse	Bezeichnung	Beispiel
$\Theta(1)$	Konstant	Einzeloperation
$\Theta(\log n)$	Logarithmisch	Binäre Suche
$\Theta(n)$	Linear	Sequentielle Suche
$\Theta(n \log n)$	Quasilinear	Sortieren eines Arrays
$\Theta(n^2)$	Quadratisch	Matrixaddition
$\Theta(n^3)$	Kubisch	(naive) Matrixmultiplikation*
$\Theta(n^k)$	Polynomiell	
$\Theta(\mathrm{k}^n)$	Exponentiell	Travelling-Salesperson†
$\Theta(n!)$	Faktoriell	Permutationen

* Strassen-Algorithmus $\mathcal{O}(n^{2.8074})$

† $\Theta(n^2 2^n)$ wenn der Algorithmus geschickt implementiert ist

o-Notation und ω-Notation

nicht asymptotisch scharfe Schranken

$$o(g) = \{f : \underbrace{\forall\, c \in \mathbb{R}_{>0}}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$$

Gilt für **alle** Konstanten $c > 0$,
in $\mathcal{O}$-Notation für eine Konstante $c > 0$

Beispiel: $2n = o(n^2)$ und $2n^2 \neq o(n^2)$

$$\omega(g) = \{f : \forall c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$$

Beispiel: ${n^2}/{2} = \omega(n)$ und ${n^2}/{2} \neq \omega(n^2)$

Was macht der folgende Sortier-Algorithmus Bubble-Sort?

```
bubbleSort(A)
1  FOR i=A.length-1 DOWNTO 0 DO
2     FOR j=0 TO i-1 DO
3        IF A[j]>A[j+1] THEN SWAP(A[j],A[j+1]);
            //temp=A[j+1]; A[j+1]=A[j]; A[j]=temp;
```

Welche Laufzeit hat der Algorithmus?

Wie verhält er sich im Vergleich zu Insertion Sort?