

# Berechenbarkeits- und Komplexitätstheorie

## Lerneinheit 3: Komplexitätsklassen

Prof. Dr. Christoph Karg

Studiengang Informatik  
Hochschule Aalen



Wintersemester 2015/2016



2.12.2015

## Einleitung

Diese Lerneinheit beschäftigt sich mit grundsätzlichen Fragestellungen der **Komplexitätstheorie**.

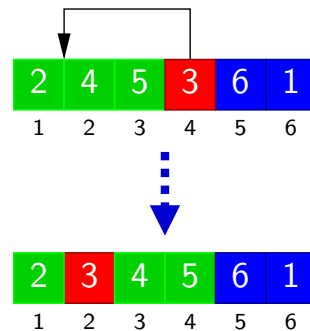
- Welche Arten von Ressourcen gibt es?
- Wie mißt man den Ressourcenverbrauch eines Algorithmus?
- Was versteht man unter einer Komplexitätsklasse?

# Insertion Sort

**Beispiel:** Insertion Sort ist ein Algorithmus zur Lösung des Sortierproblems.

**Idee:** Konstruktion der sortierten Folge durch Einfügen eines Elements in eine bereits sortierte Teilfolge

**Beispiel:**



## Insertion Sort Algorithmus

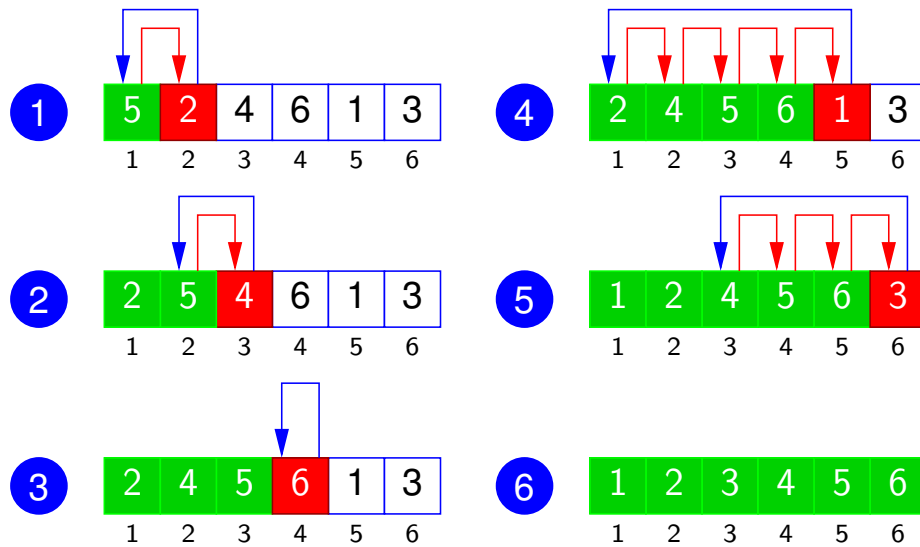
**INSERTIONSORT( $A$ )**

**Input:** Array  $A$

**Output:** Array  $A$  in sortierter Form

```
1 for  $j := 2$  to  $length(A)$  do
2    $key := A[j]$ 
3    $i := j - 1$ 
4   while ( $i > 0$  and  $A[i] > key$ ) do
5      $A[i + 1] := A[i]$ 
6      $i := i - 1$ 
7    $A[i + 1] := key$ 
```

# Arbeitsweise von Insertion Sort



# Analyse von Algorithmen

Die Analyse besteht aus zwei Teilen:

- **Korrektheit:** Zu zeigen ist, dass der Algorithmus für alle Instanzen des Problems eine korrekte Lösung berechnet
- **Komplexität:** Hier interessiert der Ressourcenverbrauch des Algorithmus. Wichtigste Ressource ist die Laufzeit, da sie über die Praxistauglichkeit des Algorithmus entscheidet

# Korrektheit von Insertion Sort

## Schleifeninvariante:

Zu Beginn jeder Iteration der for-Schleife in Zeile 1–7 enthält das Teilarray  $A[1..j - 1]$  die ursprünglichen Elemente von  $A[1..j - 1]$  in sortierter Form

Zu beweisen:

- **Initialisierung:** Die Invariante gilt vor dem ersten Schleifendurchlauf
- **Aufrechterhaltung:** Ist die Invarinate vor einer Iteration der Schleife wahr, dann ist sie auch nach der iteration wahr
- **Beendigung:** Wenn die Schleife endet, dann liefert die Invariante einen wichtigen Beitrag zur Korrektheit des Algorithmus

# Korrektheitsbeweis: Initialisierung

Die Initialisierung ist der Zeitpunkt direkt vor dem ersten Ausführung der Schleife.

- Vor dem ersten Durchlauf wird die Variable  $j$  auf 2 gesetzt.
- Wegen  $A[1..j - 1] = A[1]$  enthält  $A[1..j - 1]$  exakt ein Element, nämlich das (usprüngliche) Element in  $A[1]$ .
- Da  $A[1]$  nur ein Element enthält, ist das Array bereits sortiert.

Somit: Initialisierung ✓

## Korrektheitsbeweis: Aufrechterhaltung

**Annahme:**  $A[1..j-1]$  ist eine sortierte Folge der ursprünglichen Elemente in  $A[1..j-1]$  und  $2 \leq j < n$ .

Betrachte den Schleifendurchlauf für  $j+1$ :

- Es wird das Element in  $A[j]$  bearbeitet.
- Am Ende des Durchlaufs enthält  $A[1..j]$  die ursprünglichen Elemente von  $A[1..j]$ .
- Um  $A[j]$  einzufügen, wird in der while-Schleife ein  $i$  gesucht mit der Eigenschaft  $A[i] \leq A[j] < A[i+1]$ .
- Der Wert in  $A[j]$  wird „zwischen“  $A[i]$  und  $A[i+1]$  eingefügt, in dem die Werte in  $A[i+1..j]$  entsprechend um eine Position nach rechts verschoben werden. Somit ist das Array  $A[1..j]$  sortiert.

Also: Aufrechterhaltung ✓

## Korrektheitsbeweis: Beendigung

Die Schleife endet, wenn  $j = n + 1$ .

Da die Invariante aufrecht erhalten bleibt, gilt:

$A[1..n]$  enthält die ursprünglichen Elemente in sortierter Reihenfolge.

Somit ist das komplette Array sortiert.

**Fazit:** Der Algorithmus INSERTIONSORT liefert das geforderte Ergebnis. Er ist also korrekt.

# Analyse der Laufzeit eines Algorithmus

- Die Laufzeit eines Algorithmus ist ein wichtiges Kriterium für dessen Praxistauglichkeit
- Die Laufzeit ist die Anzahl der Elementaroperationen, die der Algorithmus zur Verarbeitung einer gegebenen Problemistanz ausführt
- Eine Elementaroperation ist eine maschinen- und programmiersprachenunabhängige Maßeinheit
- Die Laufzeit wird in Abhängigkeit der Größe der zu verarbeitenden Problemistanz gemessen
- Die Eingabegröße ist eine naheliegende Kennzahl z.B. Anzahl der Elemente in einem Feld, Anzahl Knoten in einem Graphen

## Analyse von Insertion Sort

Zeile	Aufwand
<b>for</b> $j := 2$ <b>to</b> $\text{length}(A)$ <b>do</b>	$n$
$\text{key} := A[j]$	$n - 1$
$i := j - 1$	$n - 1$
<b>while</b> $(i > 0 \text{ and } A[i] > \text{key})$ <b>do</b>	$\sum_{j=2}^n t_j + 1$
$A[i + 1] := A[i]$	$\sum_{j=2}^n t_j$
$i := i - 1$	$\sum_{j=2}^n t_j$
$A[i + 1] := \text{key}$	$n - 1$

$t_j$  = Anzahl der Durchläufe der while-Schleife im Durchgang  $j$

Gesamt:

$$T(n) = 4n - 3 + \sum_{j=2}^n (3t_j + 1) \text{ Elementaroperationen}$$

# Analyse von Insertion Sort (Forts.)

**Beobachtung:** Laufzeit ist abhängig von der Anzahl der Durchläufe der while Schleife

Drei Abschätzungen:

- **Bester Fall (Best Case):** Wieviele Operationen werden mindestens ausgeführt?
- **Schlimmster Fall (Worst Case):** Wieviele Operationen werden höchstens ausgeführt?
- **Durchschnittsfall (Average Case):** Wieviele Operationen werden im Mittel ausgeführt?

Alle Abschätzungen erfolgen in Abhängigkeit der Eingabegröße.

## Bester Fall

Ist  $A$  bereits sortiert, dann wird die while-Schleife nie durchlaufen.

Also ist  $t_j = 0$  für alle  $j$ . Die Laufzeit ist

$$T_{BC}(n) = 4n - 3 + \sum_{j=2}^n 1 = 5n - 4$$

Die Laufzeit ist **linear**, da  $T_{BC}(n) = an + b$ , wobei  $a, b$  konstant.

## Schlimmster Fall

Ist  $A$  in umgekehrter Reihenfolge sortiert, d.h.

$$A[1] \geq A[2] \geq \dots \geq A[n]$$

dann wird die while-Schleife immer solange durchlaufen bis  $i = 0$  gilt.

Also ist  $t_j = j$  für alle  $j$ . Die Laufzeit ist

$$T_{WC}(n) = 4n - 3 + \sum_{j=2}^n (3j + 1)$$

## Schlimmster Fall (Forts.)

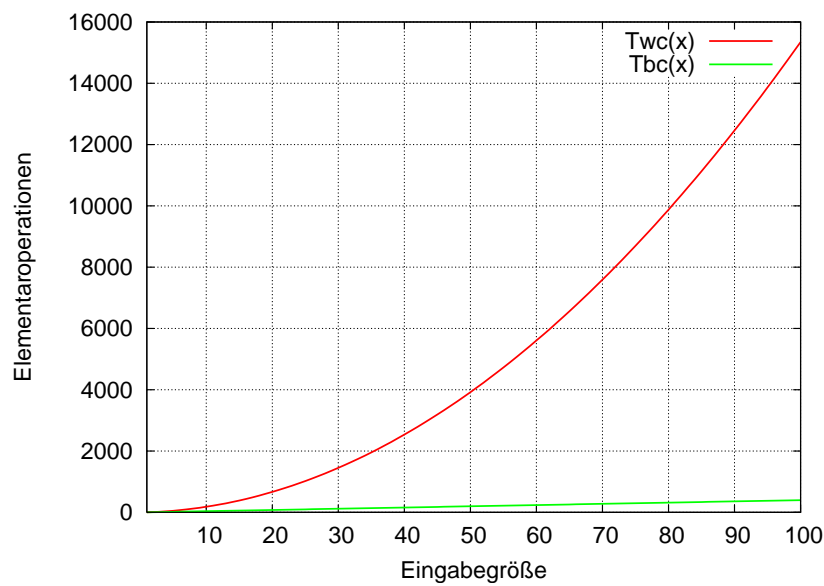
Umformen:

$$\begin{aligned} T_{WC}(n) &= 4n - 3 + n - 1 + 3 \sum_{j=2}^n j \\ &= 5n - 4 + 3 \left( \frac{n(n+1)}{2} - 1 \right) \\ &= \frac{3}{2}n^2 + \frac{13}{2}n - 7 \end{aligned}$$

Die Laufzeit ist **quadratisch**, da  $T_{WC}(n) = an^2 + bn + c$  wobei  $a, b, c$  konstant sind.



## Zwischenbilanz



**Fakt:** Die tatsächliche Laufzeit für eine Eingabe der Größe  $n$  liegt irgendwo zwischen besten und schlimmsten Fall.

## Zwischenbilanz (Forts.)

**Beispiel:** Laufzeit von Insertion Sort unter der Annahme, dass die Ausführung einer Elementaroperation 0.001 Sekunden dauert.

$n$	$T_{BC}(n)$	$Z_{min}(n)$ [sec]	$T_{WC}(n)$	$Z_{max}(n)$ [sec]
50	99	0.099	3912	3.921
100	199	0.199	15346	15.346
500	999	0.999	37674	37.674
1000	1999	1.999	1503496	1503.496
5000	9999	9.999	37517496	37517.496

**Frage:** Tendiert die Laufzeit von Insertion Sort zu linear oder zu quadratisch?

↪ Analyse des Durchschnittsfalls

# Durchschnittsfall

## Annahmen:

- Die Elemente des Arrays  $A$  sind paarweise verschieden, d.h.,  $A[i] \neq A[j]$  für alle  $1 \leq i < j \leq n$ .
- Jede Permutation der Elemente des Arrays ist gleichwahrscheinlich. Eine konkrete Belegung des Arrays tritt also mit Wahrscheinlichkeit  $\frac{1}{n!}$  auf.
- Idee hinter der Verteilung: „Ziehen der Elemente ohne Zurücklegen“

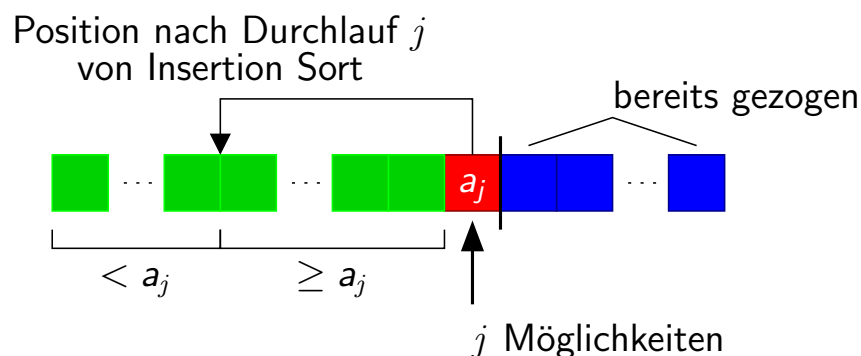
**Aufgabe:** Berechnung der zu erwartenden Durchläufe der while-Schleife in Abhängigkeit von  $j$

# Durchschnittsfall (Forts.)

**Situation** nach  $n - j$  gezogenen Elementen: es sind noch  $j$  Elemente zu verteilen.

**Beobachtung:** Die Anzahl der Durchläufe der while-Schleife in Durchgang  $j$  hängt ab von der Ordnung von  $a_j$  bezüglich der restlichen  $j - 1$  Elemente.

**Begründung:**



## Durchschnittsfall (Forts.)

Zufallsvariable  $X_j$ : Anzahl der Durchläufe der while-Schleife in Durchgang  $j$

Da das  $j$ -te Element unter Gleichverteilung gezogen wird, gilt

$$\text{Prob} [X_j = k] = \frac{1}{j}$$

für alle  $k = 0, \dots, j - 1$

Somit:

$$\text{Exp} [X_j] = \sum_{k=0}^{j-1} k \cdot \text{Prob} [X_j = k] = \sum_{k=0}^{j-1} \frac{k}{j} = \frac{j-1}{2}$$

## Durchschnittsfall (Forts.)

Es gilt: Der Erwartungswert der Anzahl der Durchläufe der while-Schleife in Durchgang  $j$  ist

$$t_j = \frac{j-1}{2}$$

Somit:

$$T_{AC}(n) = 4n - 3 + \sum_{j=2}^n \left( \frac{3(j-1)}{2} + 1 \right)$$

## Durchschnittsfall (Forts.)

Umformen:

$$\begin{aligned}T_{AC}(n) &= 4n - 3 + n - 1 + \frac{3}{2} \sum_{j=2}^n (j - 1) \\&= 5n - 4 + \frac{3}{2} \sum_{j=1}^{n-1} j \\&= 5n - 4 + \frac{3}{2} \cdot \frac{(n-1)n}{2} \\&= \frac{3}{4}n^2 + \frac{17}{4}n - 4\end{aligned}$$

Somit: Im Mittel ist die Laufzeit von Insertion Sort quadratisch.

## Ergebnis

Laufzeitverhalten von Insertion Sort:

- **Bester Fall:** linear

$$T_{BC}(n) = 5n - 4$$

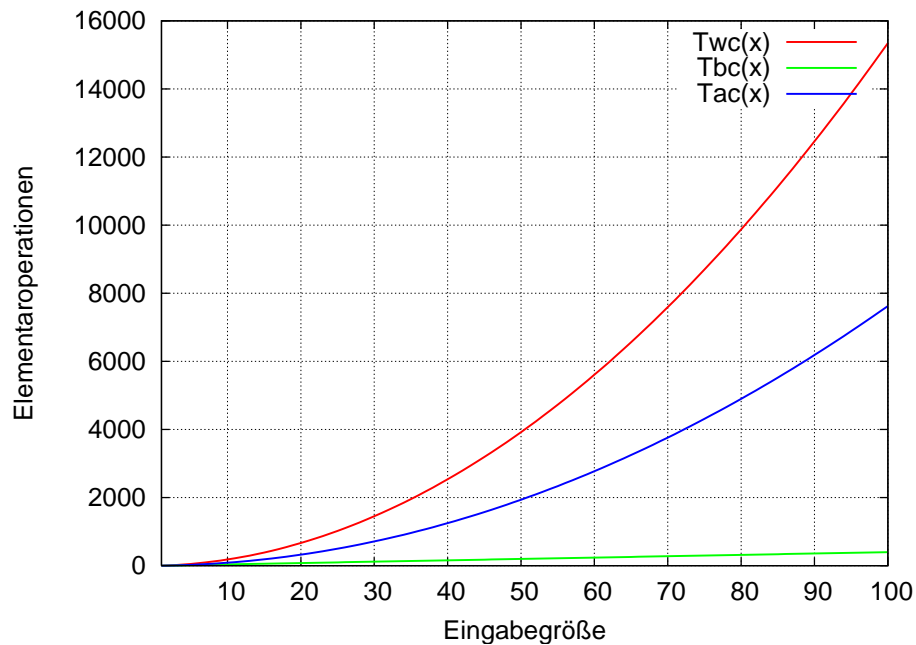
- **Schlimmster Fall:** quadratisch

$$T_{WC}(n) = \frac{3}{2}n^2 + \frac{13}{2}n - 7$$

- **Durchschnittsfall:** quadratisch

$$T_{AC}(n) = \frac{3}{4}n^2 + \frac{5}{4}n - 1$$

## Ergebnis in grafischer Form



## Worst Case Komplexität

Die Performance eines Algorithmus wird in der Regel anhand dessen Worst Case Laufzeit beurteilt.

Gründe:

- Der Worst Case ist eine obere Schranke für die Laufzeit des Algorithmus auf alle Eingaben
- Bei vielen Algorithmen tritt das Worst Case Verhalten bei den meisten Eingaben auf
- Oft das Laufzeitverhalten im Average Case (fast) identisch mit dem Verhalten im Worst Case
- Die Worst Case Analyse ist oft einfacher durchzuführen als die Average Case Analyse

# Asymptotische Notation

**Ziel:** Entwicklung eines Komplexitätsmaßes zum Vergleich von Algorithmen

**Anforderung:** Komplexitätsmaß muss unabhängig sein von:

- Programmiersprache
- Compiler
- Prozessortyp

**Idee:** Vereinfache die Komplexitätsfunktion durch

- Weglassen von Termen niedrigerer Ordnung
- Ignorieren von multiplikativen Konstanten

## Asymptotische Notation (Forts.)

**Beispiel:** Betrachte die Laufzeit von Insertion Sort

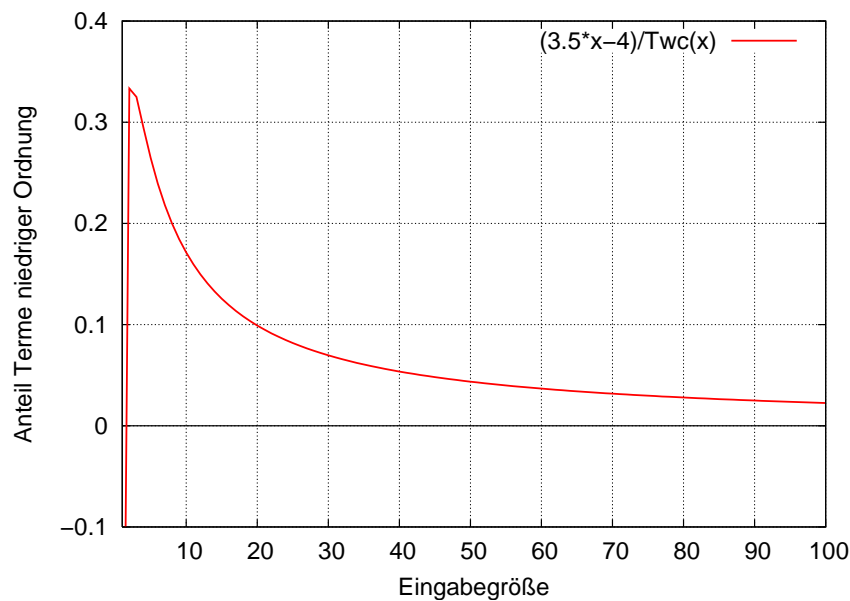
$$T_{WC}(n) = \frac{3}{2}n^2 + \frac{7}{2}n - 4$$

- Der Beitrag der Terme niedrigerer Ordnung  $\frac{7}{2}n - 4$  zu  $T_{WC}(n)$  geht für  $n \rightarrow \infty$  gegen 0 (siehe nächste Folie).
- Der Faktor  $\frac{3}{2}$  verändert nicht das quadratische Verhalten der von  $T_{WC}(n)$ .
- Die Laufzeit von Insertion Sort nimmt (im schlimmsten Fall) quadratisch mit der Eingabegröße zu.

Ergebnis: Insertion Sort hat eine **asymptotische obere Schranke** von  $n^2$ .

# Asymptotische Notation (Forts.)

Anteil von  $\frac{7}{2}n - 4$  an  $T_{WC}(n)$ :



## O-Notation

Sei  $f$  eine Abbildung von  $\mathbb{N}$  nach  $\mathbb{R}^+$ .

$O(f)$  ist die Menge aller Funktionen  $g$  mit folgender Eigenschaft:

Es gibt zwei Konstanten  $c > 0$  und  $n_0 > 0$  so dass für alle  $n \geq n_0$  gilt:  $0 \leq g(n) \leq c \cdot f(n)$ .

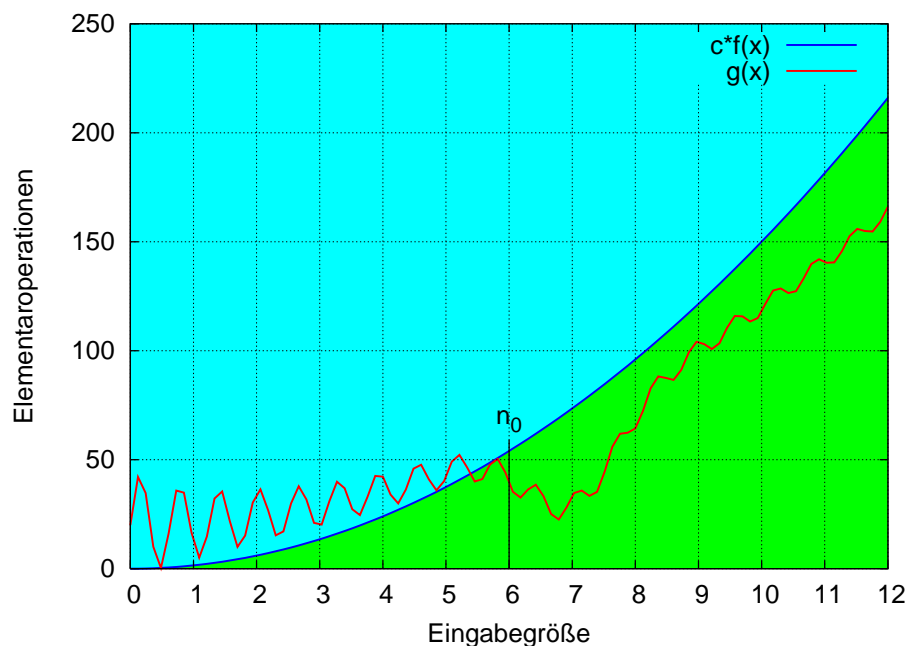
Formal:

$$O(f) = \{g \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq g(n) \leq c \cdot f(n)\}$$

Ist  $g \in O(f)$ , dann nennt man  $f$  **asymptotische obere Schranke** von  $g$ .

**Beachte:** Man schreibt oft  $g(n) = O(f(n))$  anstatt  $g(n) \in O(f(n))$ .

## O-Notation (Forts.)



## O-Notation: Beispiele

**Beispiel:**  $n^2 + n + 2 \in O(n^2)$ :

**Lösung 1:** Abschätzen. Es gilt:

$$n^2 + n + 2 \leq n^2 + n^2 + 2n^2 = 4n^2$$

Somit:  $c = 5$ ,  $n_0 = 1$

**Lösung 2:** Nullstellen berechnen.

$$\begin{aligned} c \cdot n^2 &\geq n^2 + n + 2 \\ (c - 1)n^2 - n - 2 &\geq 0 \end{aligned}$$

Mitternachtsformel:

$$n_0 = \frac{1 + \sqrt{1 + 4(c - 1)2}}{2(c - 1)}$$



## O-Notation: Beispiele (Forts.)

Ergebnis: Gleichung lösbar für alle  $c > 1$

$\rightsquigarrow$  es gibt unendlich viele  $\langle c, n_0 \rangle$  Paare. Hier eine Auswahl:

$c$	$n_0$
1.1	12
1.001	1002
1.0001	10002

**Bemerkung:** In vielen Fällen ist das Lösen der Gleichung deutlich aufwändiger als das Abschätzen.

## O-Notation: Beispiele (Forts.)

**Beispiel:**  $n \in O(n^2)$ : Es gilt  $n \leq n^2$  für alle  $n$ . Wähle  $c = 1$ ,  $n_0 = 1$ .

**Beispiel:**  $n^3 \notin O(n^2)$ : Angenommen, doch! Dann existieren Konstanten  $c > 0$  und  $n_0 > 0$  so dass für alle  $n \geq n_0$

$$n^3 \leq c \cdot n^2$$

gilt.

Umformen:  $n \leq c \rightsquigarrow$  Widerspruch für alle  $n \geq c + 1$

## O-Notation: Beispiele (Forts.)

**Beispiel:**  $n^3 \in O(2^n)$ : Auch hier gibt es unendlich viele  $\langle c, n_0 \rangle$  Paare.

Begründung: Für alle  $c > 0$  gilt:

$$\lim_{n \rightarrow \infty} \frac{n^3}{c \cdot 2^n} = 0$$

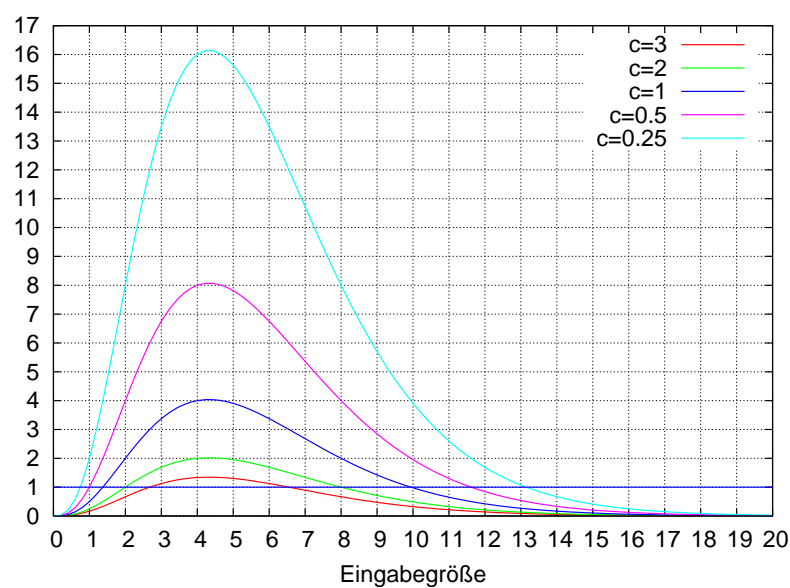
Für jedes noch so kleine  $c > 0$  gibt es ein  $n_0$  so dass

$$c \cdot 2^n \geq n^3$$

für alle  $n \geq n_0$ .

## O-Notation: Beispiele (Forts.)

Funktionsplots für  $\frac{n^3}{c \cdot 2^n}$ :



# $\Omega$ -Notation

Sei  $f$  eine Abbildung von  $\mathbb{N}$  nach  $\mathbb{R}^+$ .

$\Omega(f)$  ist die Menge aller Funktionen  $g$  mit folgender Eigenschaft:

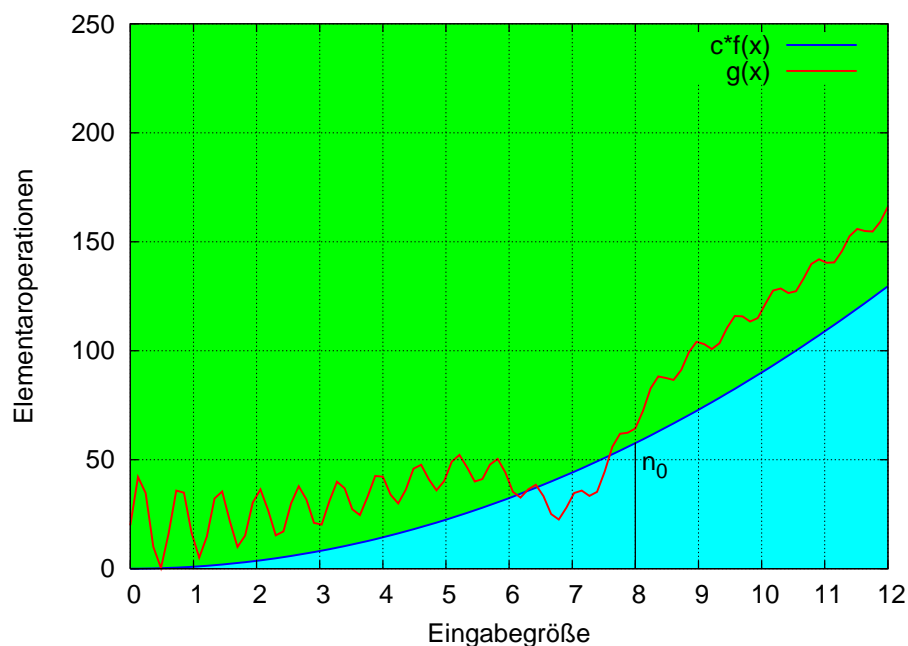
Es gibt zwei Konstanten  $c > 0$  und  $n_0 > 0$  so dass für alle  $n \geq n_0$  gilt:  $0 \leq c \cdot f(n) \leq g(n)$ .

Formal:

$$\Omega(f) = \{g \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq c \cdot f(n) \leq g(n)\}$$

Ist  $g \in \Omega(f)$ , dann nennt man  $f$  **asymptotische untere Schranke** von  $g$ .

## $\Omega$ -Notation (Forts.)



## $\Omega$ -Notation: Beispiele

**Beispiel:**  $n^2 + n + 2 = \Omega(n^2)$ , da  $n^2 + n + 2 \geq n^2$  für alle  $n \geq 1$ .

**Beispiel:**  $n^2 \in \Omega(n \log_2 n)$

Es gilt:  $2^n = 1 + \sum_{i=0}^{n-1} 2^i \geq n$ .

Somit:  $n \geq \log_2 n$ .

Abschätzen:  $n^2 = n \cdot n \geq n \log_2 n$

**Beispiel:**  $n^3 \notin \Omega(2^n)$ : Angenommen, doch. Dann gibt es Konstanten  $c > 0$  und  $n_0 > 0$ , so dass  $n^3 \geq c \cdot 2^n$  für alle  $n \geq n_0$ .

Dies steht jedoch im Widerspruch zu der Tatsache, dass

$$\lim_{n \rightarrow \infty} \frac{c \cdot 2^n}{n^3} \rightarrow \infty$$

## $\Theta$ -Notation

Sei  $f$  eine Abbildung von  $\mathbb{N}$  nach  $\mathbb{R}^+$ .

$\Theta(f)$  ist die Menge aller Funktionen  $g$  mit folgender Eigenschaft:

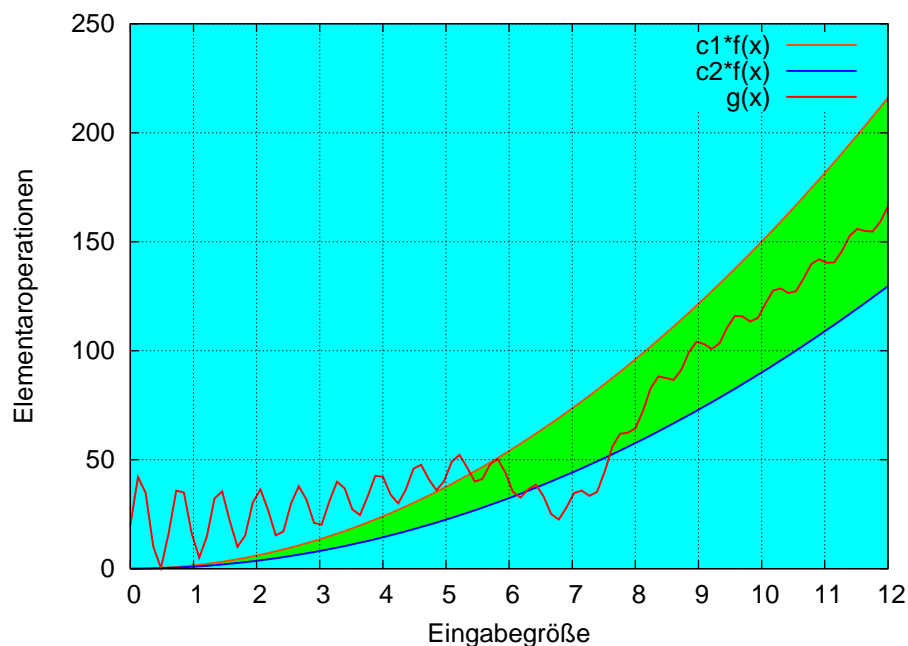
Es gibt Konstanten  $c_1 > 0$ ,  $c_2 > 0$  und  $n_0 > 0$  so dass für alle  $n \geq n_0$  gilt:  
 $0 \leq c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$ .

Formal:

$$\Theta(f) = \left\{ g \mid \begin{array}{l} \exists c_1, c_2 > 0 \exists n_0 > 0 \forall n \geq n_0 : \\ 0 \leq c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n) \end{array} \right\}$$

Ist  $g \in \Theta(f)$ , dann nennt man  $f$  **asymptotische dichte Schranke** von  $g$ .

## $\Theta$ -Notation (Forts.)



## $\Theta$ -Notation (Forts.)

**Satz:** Für beliebige Funktionen  $g(n)$  und  $f(n)$  gilt:

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

**Beweis.** " $\Rightarrow$ ":  $\checkmark$

" $\Leftarrow$ ":

- Da  $f(n) = \Omega(g(n))$ , gibt es eine Konstante  $c_1 > 0$  so dass  $0 \leq c_1 \cdot g(n) \leq f(n)$  für alle  $n \geq n_0$ .
- Da  $f(n) = O(g(n))$ , gibt es eine Konstante  $c_2 > 0$  so dass  $0 \leq f(n) \leq c_2 \cdot g(n)$  für alle  $n \geq n_0$ .

Somit:  $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

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

# Relationen

## Reflexivität

- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$
- $f(n) = \Theta(f(n))$

## Transitivität

- Falls  $f(n) = O(g(n))$  und  $g(n) = O(h(n))$ , dann  $f(n) = O(h(n))$ .
- Falls  $f(n) = \Omega(g(n))$  und  $g(n) = \Omega(h(n))$ , dann  $f(n) = \Omega(h(n))$ .
- Falls  $f(n) = \Theta(g(n))$  und  $g(n) = \Theta(h(n))$ , dann  $f(n) = \Theta(h(n))$ .

# Relationen (Forts.)

## Symmetrie

- $f(n) = \Theta(g(n))$  genau dann, wenn  $g(n) \in \Theta(f(n))$

## Transponierte Symmetrie

- $f(n) = O(g(n))$  genau dann, wenn  $g(n) = \Omega(f(n))$

## Analogie zu reellen Zahlen:

$$f(n) = O(g(n)) \rightsquigarrow a \leq b$$

$$f(n) = \Omega(g(n)) \rightsquigarrow a \geq b$$

$$f(n) = \Theta(g(n)) \rightsquigarrow a = b$$

# Laufzeitschranken

Die folgende Tabelle enthält gängige Laufzeitschranken und deren informelle Bezeichnung:

$O(1)$	konstant
$O(\log n)$	logarithmisch
$O(n)$	linear
$O(n \log n)$	quasi-linear
$O(n^2)$	quadratisch
$O(n^3)$	kubisch
$O(2^n)$	exponentiell

**Ziel** bei der Komplexitätsanalyse ist es, eine möglichst dichte obere Schranke anzugeben.

## Wann ist ein Algorithmus effizient?

Zu Beantwortung dieser Frage betrachten wir folgende Tabelle:

Algorithmus	Komplexität	DS/h	DS/h (Faktor 1000)
$A_1$	$O(n)$	$n_1$	$1000n_1$
$A_2$	$O(n^2)$	$n_2$	$\sqrt{1000} \cdot n_2 = 31.6n_2$
$A_3$	$O(n^3)$	$n_3$	$\sqrt[3]{1000} \cdot n_3 = 10n_3$
$A_4$	$O(2^n)$	$n_4$	$n_4 + \log_2 1000 = n_4 + 10$

- DS/h steht für die maximale Eingabegröße, die der Algorithmus in einer Stunde bewältigen kann.
- Faktor 1000 steht für die Verbesserung bei Einsatz eines 1000-mal schnelleren Computers.

Die obige Tabelle liefert folgende Erkenntnisse:

- Bei Algorithmen mit Komplexität  $O(n^k)$  erzielt man eine Verbesserung um einen Faktor, der von der ursprünglichen Eingabegröße  $x$  und  $k$  abhängt. Dieser Faktor ist gleich  $\sqrt[k]{x}$
- Bei dem Algorithmus mit Komplexität  $O(2^n)$  ist die erzielte Verbesserung additiv. Die Eingabegröße erhöht sich um  $\log_2 x$

## Definition von Effizienz

### Definition:

Ein Algorithmus  $A$  ist effizient, falls seine Zeitkomplexität  $T_A(n)$  Polynomialzeit-beschränkt ist, d.h., dass  $T_A(n) \in O(n^k)$  für eine Konstante  $k > 0$

Bemerkungen:

- Diese Definition ist **robust**. Werden effiziente Algorithmen kombiniert, dann ist das Ergebnis wiederum ein effizienter Algorithmus
- Diese Definition ist eine mathematische **Idealisierung**. Ein Algorithmus mit Laufzeit  $O(n^{1000})$  ist sicherlich nicht mehr praktikabel



# Komplexitätstheorie

**Aufgabe:** Einteilung der entscheidbaren Probleme anhand des Ressourcenverbrauchs, der bei ihrer Verarbeitung anfällt

**Ressourcen:**

- Rechenzeit
- Speicherplatz
- Kommunikationsaufwand
- ...

Der Verbrauch wird in Abhängigkeit der Eingabe bzw. deren Länge gemessen

**Annahme:** Die Eingaben werden über einem Alphabet  $\Sigma$  kodiert

# Komplexitätsklassen

**Definition.** Eine **Komplexitätsklasse**  $\mathcal{C}$  über dem Alphabet  $\Sigma$  ist eine Menge von Sprachen über  $\Sigma$ , d.h.,  $\mathcal{C} \subseteq P(\Sigma^*)$

**Unterscheidung** zwischen

- Determinismus und
- Nichtdeterminismus

sowie zwischen

- Rechenzeit und
- Speicherplatz

**Ressourcenfunktion:** Funktion der Bauart

- $\mathbb{N} \mapsto \mathbb{N}$  oder
- $\Sigma^* \mapsto \mathbb{N}$

# Zeitkonstruierbare Funktionen

**Definition.** Eine Abbildung  $t : \mathbb{N} \mapsto \mathbb{N}$ , wobei  $t(n) \geq kn \log_2(n)$  für eine Konstante  $k > 0$ .  $t$  nennt man **zeitkonstruierbar**, falls es eine deterministische Turing Maschine  $M$  gibt, die auf Eingabe  $1^n$  in Laufzeit  $O(t(n))$  den Wert  $t(n)$  in Binärdarstellung berechnet

**Fakt:** Alle gängigen Funktionen  $t(n)$  mit  $t(n) \geq n \log_2 n$  sind zeitkonstruierbar, z.B.  $n \log_2 n$ ,  $n\sqrt{n}$ ,  $n^2$  und  $2^n$

**Anwendung:** Einsatz als Count-Down Zähler in Algorithmen  
 $\rightsquigarrow$  Beschränkung der Rechenzeit auf eine vorgegebene Anzahl von Rechenschritten

## Beispiel: $n^2$ ist zeitkonstruierbar

**Beispiel:** Algorithmus zur Berechnung von  $t(n) = n^2$

**TIMESQUARE**( $1^n$ )

**Input:** Natürliche Zahl  $n \geq 1$  in Unärkodierung

**Output:** Wert von  $t(n)$  in Binärkodierung

```
1  $s := 0$ ;  
2 Scanne die Eingabe von links nach rechts und  
   erhöhe für jede gelesene 1 den Zähler  $s$  um 1  
3  $t := 0$   
4 for  $i := 1$  to  $s$  do  
5    $t := t + s$   
6 return  $t$ 
```

## Beispiel: $n^2$ ist zeitkonstruierbar (Forts.)

**Korrektheit:** Da  $s = n$ , gilt:

$$t = \sum_{i=1}^n n = n^2$$

**Laufzeit:**

- Scannen der Eingabe und  $s$  inkrementieren:  $O(n \cdot \log_2 n)$
- Berechnen von  $t$ :  $O(n \log_2 n)$

Gesamt:  $O(n \cdot \log_2 n)$

## Beispiel: $n^2$ ist zeitkonstruierbar (Forts.)

**Anwendung:** Stoppuhr bei der Simulation von Turing Maschinen

$\text{SIMULATE}(\langle M, x \rangle)$

**Input:** Turing Maschine  $M$  mit zugehöriger Eingabe  $x$

**Output:** true, falls  $M$  das Wort  $x$  in  $\leq |x|^2$  Schritten akzeptiert, false, sonst.

```
1 timer := TIMESQUARE( $1^{|x|}$ )
2  $C$  := Startkonfiguration von  $M$  auf Eingabe  $x$ 
3 while ( $timer > 0$ ) do
4   if ( $C$  ist eine akzeptierende Konfiguration) then
5     return true
6    $C$  := Folgekonfiguration von  $C$ 
7    $timer := timer - 1$ 
8 return false
```

**Definition.** Sei  $M$  eine deterministische Turing Maschine, die auf allen Eingaben hält. Die **Laufzeit** (**Zeitkomplexität**) von  $M$  ist eine Funktion  $f : \mathbb{N} \mapsto \mathbb{N}$  mit der Eigenschaft, dass  $M$  bei allen Eingaben der Länge  $n$  höchstens  $O(f(n))$  Rechenschritte durchläuft.

Man unterscheidet:

- Zeitkomplexität eines Algorithmus
- Zeitkomplexität eines Entscheidungsproblems

**Definition.** Sei  $t : \mathbb{N} \mapsto \mathbb{N}$  eine beliebige zeitkonstruierbare Abbildung. Die Menge  $\text{DTIME}(t(n))$  enthält alle Sprachen, die durch eine deterministische Einband Turing Maschine mit Zeitkomplexität  $t(n)$  entscheidbar sind.

## Einband vs. Mehrband Turing Maschinen

**Satz.** Sei  $L$  eine Sprache, die durch eine deterministische  $k$ -Band Turing Maschine  $M$  akzeptiert wird. Angenommen,  $M$  verarbeitet jede Eingabe der Länge  $n$  in  $t(n)$  Rechenschritten. Dann ist  $L \in \text{DTIME}(t(n)^2)$ .

*Beweis.* Betrachte die deterministische Einband Turing Maschine  $M'$ , die  $M$  simuliert, indem sie den Inhalt der  $k$  Arbeitsbänder hintereinander auf ihr Arbeitsband schreibt

**Beobachtung:** Da  $M$  höchstens  $t(n)$  Rechenschritte ausführt, kann sie pro Arbeitsband höchstens  $t(n)$  Zellen beschreiben  
 $\rightsquigarrow M'$  belegt höchstens  $O(k \cdot t(n))$  Speicherzellen

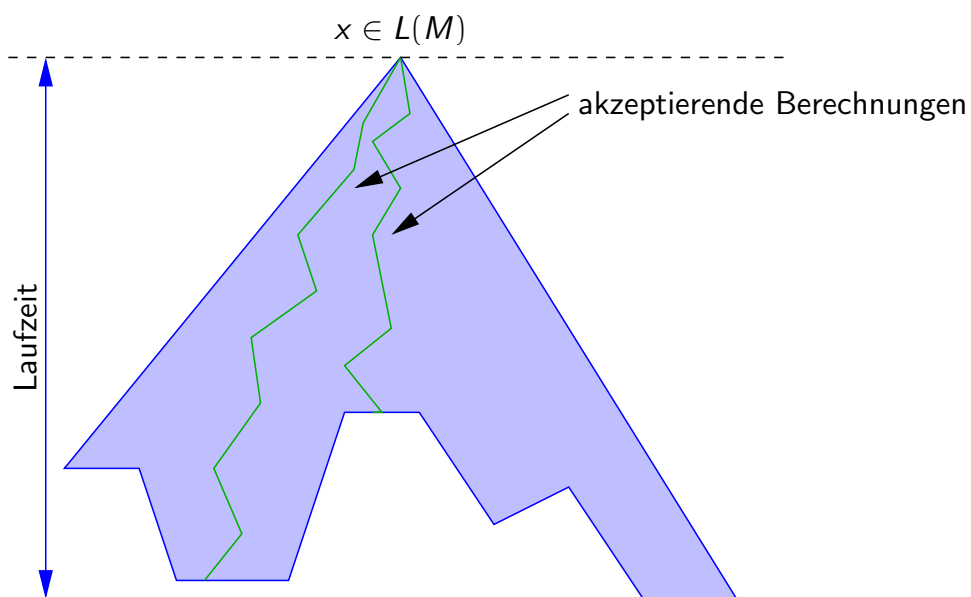
# Einband vs. Mehrband Turing Maschinen (Forts.)

Abschätzung des Aufwands:

- Simulation eines Rechenschritts von  $M$ :
    - ▷ Scannen des Bandes, um die Buchstaben der  $k$  S/L-Köpfe zu ermitteln:  $O(k \cdot t(n))$
    - ▷ Ausführen des Rechenschritts:  $O(k \cdot t(n))$
    - ▷ Vergrößern der Arbeitsbänder:  $O(k^2 \cdot t(n))$
- Gesamt:  $O(k^2 + 2k \cdot t(n))$
- Laufzeit insgesamt:  $O(t(n) \cdot (k^2 + 2k \cdot t(n))) = O(t(n)^2)$

Fazit:  $L \in \text{DTIME}(t(n)^2)$

## Laufzeit einer nichtdet. Turing Maschine



# Nichtdeterministische Zeitkomplexität

**Laufzeit** von  $M$  für Eingaben der Länge  $n$ :

Maximale Länge eines Berechnungspfads bei einer beliebigen Eingabe der Länge  $n$

**Definition.** Sei  $t : \mathbb{N} \mapsto \mathbb{N}$  eine beliebige zeitkonstruierbare Abbildung. Die Menge  $\text{NTIME}(t(n))$  enthält alle Sprachen, die durch eine nichtdeterministische Einband Turing Maschine mit Laufzeit  $O(t(n))$  entscheidbar sind.

**Satz.** Sei  $L$  eine Sprache, die durch eine nichtdeterministische  $k$ -Band Turing Maschine  $M$  akzeptiert wird. Angenommen,  $M$  verarbeitet jede Eingabe der Länge  $n$  in  $t(n)$  Rechenschritten. Dann ist  $L \in \text{NTIME}(t(n)^2)$ .

## Determinismus vs. Nichtdeterminismus

**Satz.**  $\text{NTIME}(t(n)) \subseteq \text{DTIME}(2^{t(n)})$ .

*Beweis.* Sei  $M = (Z, \Gamma, \Sigma, \vdash, \delta, z_s, z_{acc}, z_{rej})$  eine nichtdeterministische Turing Maschine mit einer Laufzeit von  $t(n)$ .

Der Einfachheit halber sei angenommen, dass

$$\|\delta(z, a)\| \leq 2$$

für alle  $z \in Z$  und  $a \in \Gamma$

Angenommen,  $M$  benötigt zur Verarbeitung von  $x$   $t(|x|)$  Rechenschritte

## Determinismus vs. Nichtdeterminismus (Forts.)

Dann ist der Berechnungsbaum von  $M$  auf Eingabe  $x$  schlimmstenfalls ein vollständiger Binärbaum der Tiefe  $t(|x|)$

Die Anzahl der Knoten in einem vollständigen Binärbaum der Tiefe  $t(|x|)$  ist

$$\sum_{i=0}^{t(|x|)} 2^i = 2^{t(|x|)+1} - 1$$

Durch eine Breitensuche in diesem Baum kann eine akzeptierende Endkonfiguration gesucht werden.

Die Laufzeit des Algorithmus ist  $O(2^{t(|x|)})$

## Platzkonstruierbare Funktionen

**Definition.** Eine Abbildung  $s : \mathbb{N} \mapsto \mathbb{N}$ , wobei  $s(n) \geq k \log_2(n)$  für eine Konstante  $k > 0$ .  $s$  nennt man **platzkonstruierbar**, falls es eine deterministische Turing Maschine  $M$  gibt, die auf Eingabe  $1^n$  den Wert  $s(n)$  in Binärdarstellung mit einem Speicherplatzverbrauch von  $O(s(n))$  berechnet

**Fakt:** Alle gängigen Funktionen  $s(n)$  mit  $s(n) \geq \log_2 n$  sind platzkonstruierbar, z.B.  $n \log_2 n$ ,  $n\sqrt{n}$ ,  $n^2$  und  $2^n$

**Anwendung:** Einsatz als Speicherplatzbeschränkung

# Platzkomplexität

**Definition.** Sei  $M$  eine deterministische Turing Maschine, die auf allen Eingaben hält. Der **Speicherplatzverbrauch** (**Platzkomplexität**) von  $M$  ist eine Funktion  $f : \mathbb{N} \mapsto \mathbb{N}$  mit der Eigenschaft, dass  $M$  bei allen Eingaben der Länge  $n$  höchstens  $f(n)$  Bandzellen beschreibt.

Man unterscheidet:

- Platzkomplexität eines Algorithmus
- Platzkomplexität eines Entscheidungsproblems

**Definition.** Sei  $s : \mathbb{N} \mapsto \mathbb{N}$  eine beliebige platzkonstruierbare Abbildung. Die Menge  $\text{DSPACE}(s(n))$  enthält alle Sprachen, die durch eine deterministische Einband Turing Maschine mit Platzkomplexität  $s(n)$  entscheidbar sind.

# Mehrband Turing Maschinen

**Satz.** Sei  $M$  eine  $k$ -Band Turing Maschine, die auf allen Eingaben der Länge  $n$  auf jedem Band höchstens  $s(n)$  viele Speicherzellen benutzt. Dann ist  $L(M) \in \text{DSPACE}(s(n))$ .

*Beweis.* Betrachte die Simulation  $M$  durch eine Einband Turing Maschine.

- Die Simulation verbraucht höchstens  $k \cdot s(n) = O(s(n))$  Speicherzellen
- Die Laufzeit wird nicht berücksichtigt

**Fazit:**  $L(M) \in \text{DSPACE}(s(n))$



## Platz- vs. Zeitkomplexität

**Satz.**  $\text{DSPACE}(s(n)) \subseteq \text{DTIME}(2^{s(n)})$

*Beweis.* Sei  $M = (Z, \Gamma, \Sigma, \sqsubset, \delta, z_s, z_{acc}, z_{rej})$  eine deterministische Einband Turing Maschine, die  $s(n)$  platzbeschränkt ist. Sei  $q = \|Z\|$  und  $g = \|\Gamma\|$

Anzahl möglicher Konfigurationen bei Eingaben der Länge  $n$ :

$$q \cdot s(n) \cdot g^{O(s(n))}$$

Da  $M$  auf allen Eingaben stoppt, darf sich während der Verarbeitung von  $x$  keine Konfiguration wiederholen. Somit ist die Anzahl aller Konfigurationen eine obere Schranke für die Laufzeit von  $M$

**Somit:**  $L(M) \in \text{DTIME}(2^{s(n)})$

## Nichtdeterministische Platzkomplexität

**Speicherplatzverbrauch** von  $M$  für Eingaben der Länge  $n$ :

Maximale Anzahl gelesener Speicherzellen über alle Berechnungspfade von  $M$  auf eine beliebige Eingabe der Länge  $n$

**Definition.** Sei  $s : \mathbb{N} \mapsto \mathbb{N}$  eine beliebige platzkonstruierbare Abbildung. Die Menge  $\text{NSPACE}(s(n))$  enthält alle Sprachen, die durch eine nichtdeterministische Einband Turing Maschine entscheidbar sind, die auf allen Eingaben stoppt und einen Speicherplatzverbrauch von  $O(s(n))$  hat.

**Satz.** Sei  $L$  eine Sprache, die durch eine nichtdeterministische  $k$ -Band Turing Maschine  $M$  akzeptiert wird. Angenommen,  $M$  benutzt bei jeder Eingabe der Länge  $n$   $O(s(n))$  Speicherzellen. Dann ist  $L \in \text{NSPACE}(s(n))$ .

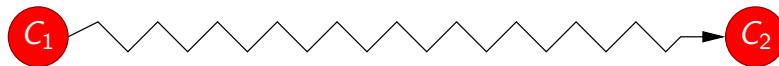
# Der Satz von Savitch

**Satz.** (Savitch) Für jede platzkonstruierbare Funktion  $f : \mathbb{N} \mapsto \mathbb{R}_0^+$  mit  $f(n) \geq n$  gilt:

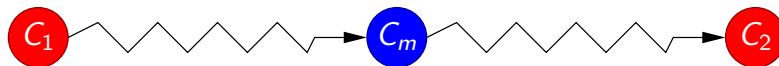
$$\text{NSPACE}(f(n)) \subseteq \text{DSPACE}(f(n)^2)$$

**Beweisidee:** Test von Erreichbarkeit von Konfigurationen

**Frage:** Ist  $C_1$  in  $\leq t$  Schritten in  $C_2$  überführbar?



**Antwort:** Ja, falls es eine Konfiguration  $C_m$  gibt, so dass  $C_1$  in  $\leq \frac{t}{2}$  Schritten in  $C_m$  und  $C_m$  in  $\leq \frac{t}{2}$  Schritten in  $C_2$  überführbar ist



# Der Satz von Savitch (Forts.)

**Beweis.** Sei  $f(n)$  eine platzkonstruierbare Funktion. Sei  $M = (Z, \Gamma, \Sigma, \sqsubset, \delta, z_s, z_{acc}, z_{rej})$  eine nichtdeterministische Turing Maschine mit einer Platzkomplexität von  $f(n)$ .

Beachte:

- Da  $f$  platzkonstruierbar ist, gibt es einen Algorithmus, der alle Konfigurationen von  $M$  der Länge  $f(n)$  auflistet
- Es gibt höchstens  $2^{O(f(n))}$  solcher Konfigurationen
- Der Einfachheit halber sei angenommen, dass  $M$  das Eingabeband komplett löscht, bevor sie eine Eingabe akzeptiert. Folglich gibt es eine einzige akzeptierende Konfiguration  $C_a$

## Der Satz von Savitch (Forts.)

$\text{CANYIELD}(C_1, C_2, t, n)$

**Input:** Konfigurationen  $C_1, C_2$ , Zahlen  $t \geq 1, n \geq 1$

**Output:** true, falls  $C_1$  in  $\leq t$  Schritten in  $C_2$  überführbar ist, false sonst.

```
1 if  $t = 1$  then
2   if  $(C_1 = C_2)$  or  $(C_1 \rightarrow_M C_2)$  then
3     return true
4   else
5     return false
6 else
7   for jede Konfiguration  $C_m$  von  $M$  der Länge  $f(n)$  do
8     if  $(\text{CANYIELD}(C_1, C_m, \frac{t}{2}, n) = \text{true})$  and
         $(\text{CANYIELD}(C_m, C_2, \frac{t}{2}, n) = \text{true})$  then
9       return true
10  return false
```

## Der Satz von Savitch (Forts.)

**Korrektheit:** Für alle  $x \in \Sigma^*$  gilt:

$$M \text{ akzeptiert } x \iff \text{CANYIELD}(C_x, C_a, 2^{df(n)}, n) = \text{true}$$

wobei  $C_x$  die Startkonfiguration von  $M$  auf Eingabe  $x$  ist

**Speicherplatzverbrauch:**

- Es gibt  $2^{df(n)}$  viele unterschiedliche Konfigurationen von  $M$  der Länge  $f(n)$ , wobei  $d$  konstant
- Die Anzahl der Rekursionen von  $\text{CANYIELD}(C_x, C_a, 2^{df(n)}, n)$  ist  $\log_2 2^{df(n)} = O(f(n))$
- Zur Ausführung der rekursiven Aufrufe kommt ein Stack zum Einsatz. Pro Aufruf werden die Übergabeparameter sowie die Schleifenvariable  $C_m$  abgespeichert

Gesamt:  $O(f(n)) \cdot O(4 \cdot f(n)) = O(f(n)^2)$  Speicherzellen

## o-Notation

**Definition.** Seien  $f$  eine Abbildung von  $\mathbb{N}$  nach  $\mathbb{R}^+$ .  $o(f)$  ist die Menge aller Funktionen  $g$  mit folgender Eigenschaft:

Für alle Konstanten  $c > 0$  gibt es ein  $n_0 > 0$  so dass für alle  $n \geq n_0$  gilt:  $f(n) < c \cdot g(n)$ .

Formal:

$$o(f) = \{g \mid \forall c > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq g(n) \leq c \cdot f(n)\}$$

Für alle  $g \in o(f)$  gilt:

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0.$$

## Platz Hierarchie Satz

**Satz.** Für jede platzkonstruierbare Funktion  $f$  gibt es eine Sprache  $L$ , die in Platzkomplexität  $O(f(n))$  entscheidbar ist, aber nicht in Platzkomplexität  $o(f(n))$ .

## Beispiel: Platz Hierarchie Satz

**Beispiel.** Betrachte  $f(n) = n^2$  und  $g(n) = n \log_2 n$ .

Es gilt:

$$\lim_{n \rightarrow \infty} \frac{n \log_2 n}{n^2} = \lim_{n \rightarrow \infty} \frac{\log_2 n}{n} = \lim_{n \rightarrow \infty} \frac{1}{n \ln 2} = 0$$

Also ist  $g(n) \in o(f(n))$ .

Mit dem Platz Hierarchie Satz folgt:

$$\text{DSPACE}(n \log_2 n) \subsetneq \text{DSPACE}(n^2)$$

## Beweis Platz Hierarchie Satz

*Beweis.*

**Idee:** Konstruiere eine Sprache  $L$  mit folgenden Eigenschaften:

- $L$  ist in Platzkomplexität  $O(f(n))$  entscheidbar
- $L$  ist nicht in Platzkomplexität  $o(f(n))$  entscheidbar

**Beweistechnik:** Diagonalisierung

## Beweis Platz Hierarchie Satz (Forts.)

Betrachte folgenden Algorithmus:

DIAGF( $w$ )

**Input:** Wort  $w \in \{0, 1\}^*$

**Output:** true oder false

- 1  $n := |w|$
- 2 **if**  $w \neq \langle M \rangle 10^k$  für eine TM  $M$  und eine Zahl  $k > 0$  **then**
- 3     **return** false
- 4 *Markiere einen Bandbereich von  $f(n)$  Zellen. Wird im weiteren Verlauf dieser Bereich überschritten, dann breche ab und gib false zurück*

## Beweis Platz Hierarchie Satz (Forts.)

- 5 *Simuliere  $M$  auf Eingabe  $w$  und zähle dabei die Anzahl Rechenschritte. Falls diese Zahl  $2^{f(n)}$  übersteigt, dann gib false zurück*
- 6 **if** ( $M$  akzeptiert  $w$ ) **then**
- 7     **return** false
- 8 **else**
- 9     **return** true

## Beweis Platz Hierarchie Satz (Forts.)

Sei  $D$  eine Turing Maschine, die den Algorithmus  $\text{DIAGG}(w)$  berechnet. Definiere  $L = L(D)$ .

**Bemerkung:**  $D$  stoppt auf allen Eingaben, d.h., die von  $D$  akzeptierte Sprache ist entscheidbar

**Platzkomplexität:** Die Berechnung wird abgebrochen, wenn mehr als die markierten  $f(n)$  Bandzellen benutzt werden  
 $\leadsto$  Speicherplatzverbrauch von  $D$ :  $O(f(n))$

**Annahme:** Es gibt eine Turing Maschine  $M$ , die  $L$  akzeptiert und eine Platzkomplexität von  $g(n) = o(f(n))$  hat

## Beweis Platz Hierarchie Satz (Forts.)

**Simulation** von  $M$  durch die Turing Maschine  $D$ :

- Das Arbeitsalphabet von  $D$  ist fest gewählt und verschieden vom Arbeitsalphabet von  $M$
- Zur Simulation muss  $D$  eine Bandzelle von  $M$  auf  $d$  Bandzellen verteilen.
- Der Wert von  $d$  hängt von der Größe der Arbeitsalphabets von  $M$  ab
- Zur Simulation von  $M$  auf eine Eingabe der Länge  $n$  benötigt  $D$  also  $d \cdot g(n)$  Speicherzellen

## Beweis Platz Hierarchie Satz (Forts.)

Da  $g(n) = o(f(n))$ , gibt es für  $d$  ein  $n_0$ , so dass  
 $d \cdot g(n) < f(n)$  für alle  $n \geq n_0$

Für die Eingabe  $w = \langle M \rangle 10^{n_0}$  markiert  $D$  genug Bandzellen,  
um  $M$  erfolgreich zu simulieren

Fallunterscheidung:

- $M$  akzeptiert  $w \rightsquigarrow D$  verwirft  $w$ . Widerspruch!
- $M$  verwirft  $w \rightsquigarrow D$  akzeptiert  $w$ . Widerspruch!

Somit: Es gibt keine Turing Maschine, die  $L$  akzeptiert und  
Platzkomplexität  $o(f(n))$  hat

## Folgerung

**Korollar.** Seien  $f_1$  und  $f_2$  Abbildungen von  $\mathbb{N}$  nach  $\mathbb{N}$ , wobei  
 $f_1(n) = o(f_2(n))$  und  $f_2$  platzkonstruierbar ist. Dann gilt:

$$\text{DSPACE}(f_1(n)) \subsetneq \text{DSPACE}(f_2(n))$$

**Korollar.** Für alle  $0 \leq k_1 < k_2$  gilt:

$$\text{DSPACE}(n^{k_1}) \subsetneq \text{DSPACE}(n^{k_2})$$



## Zeit Hierarchie Satz

**Satz.** Für jede zeitkonstruierbare Abbildung  $t : \mathbb{N} \mapsto \mathbb{N}$  gibt es eine Sprache  $L$ , die in Zeitkomplexität  $O(t(n))$  entscheidbar ist, aber nicht in Zeitkomplexität  $o(t(n)/\log_2 t(n))$ .

**Korollar.** Für zwei beliebige Funktion  $t_1, t_2 : \mathbb{N} \mapsto \mathbb{N}$ , wobei  $t_1(n) = o(t_2(n)/\log_2 t_2(n))$  und  $t_2(n)$  zeitkonstruierbar ist, gilt:

$$\text{DTIME}(t_1(n)) \subsetneq \text{DTIME}(t_2(n))$$

**Korollar.** Für alle  $1 = k_1 < k_2$  gilt:

$$\text{DTIME}(n^{k_1}) \subsetneq \text{DTIME}(n^{k_2})$$

## Beispiel Zeit Hierarchie Satz

**Beispiel.** Betrachte  $g(n) = n^{1.9}$  und  $f(n) = n^2$

Es gilt:

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{n^{1.9}}{\frac{n^2}{\log_2 n}} &= \lim_{n \rightarrow \infty} \frac{n^{1.9} \log_2 n}{n^2} \\ &= \lim_{n \rightarrow \infty} \frac{\log_2 n}{n^{0.1}} \\ &= \lim_{n \rightarrow \infty} \frac{1}{0.1 \cdot n^{0.1} \ln 2} \rightarrow 0 \end{aligned}$$

Mit dem Zeit Hierarchie Satz folgt:

$$\text{DTIME}(n^{1.9}) \subsetneq \text{DTIME}(n^2)$$

# Wichtige Komplexitätsklassen

$$L = \text{DSpace}(\log_2 n)$$

$$NL = \text{NSpace}(\log_2 n)$$

$$P = \bigcup_{k=1}^{\infty} \text{DTIME}(n^k)$$

$$NP = \bigcup_{k=1}^{\infty} \text{NTIME}(n^k)$$

# Wichtige Komplexitätsklassen (Forts.)

$$\text{EXP} = \bigcup_{k=1}^{\infty} \text{DTIME}(2^{n^k})$$

$$\text{NEXP} = \bigcup_{k=1}^{\infty} \text{NTIME}(2^{n^k})$$

$$\text{PSPACE} = \bigcup_{k=1}^{\infty} \text{DSpace}(n^k)$$

$$\text{NPSPACE} = \bigcup_{k=1}^{\infty} \text{NSpace}(n^k)$$

# Beziehungen zwischen Komplexitätsklassen

**Satz.**  $NL \subseteq P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXP$

*Beweis.* Anwendung der bereits bewiesenen Sätze

$NL \subseteq P$ : ✓

$P \subseteq NP$ : ✓

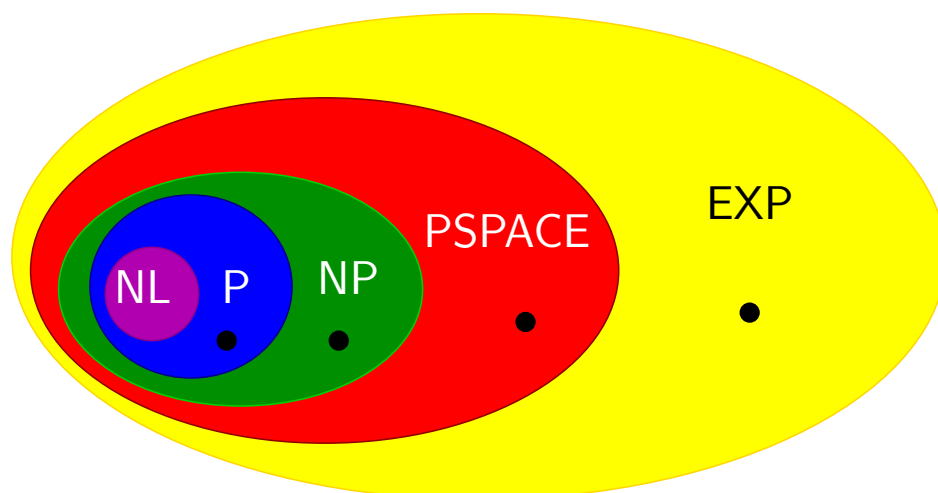
$PSPACE = NPSPACE$ : Satz von Savitch

$NP \subseteq PSPACE$ : ✓

$NPSPACE \subseteq EXP$ : ✓

## Eine offene Vermutung

**Vermutung:**  $NL \subsetneq P \subsetneq NP \subsetneq PSPACE = NPSPACE \subsetneq EXP$



Diese Vermutung wurde bisher weder bewiesen noch widerlegt

- Ziel der Komplexitätstheorie ist die Einteilung von entscheidbaren Sprachen anhand des zu ihrer Berechnung notwendigen Ressourcenverbrauchs
- Die wichtigsten Ressourcen sind Rechenzeit und Speicherplatz
- Es gibt keine obere Schranke für den maximalen Ressourcenverbrauch
- Eine bisher nicht bewiesene Vermutung ist:

$$NL \subsetneq P \subsetneq NP \subsetneq PSPACE \subsetneq EXP$$