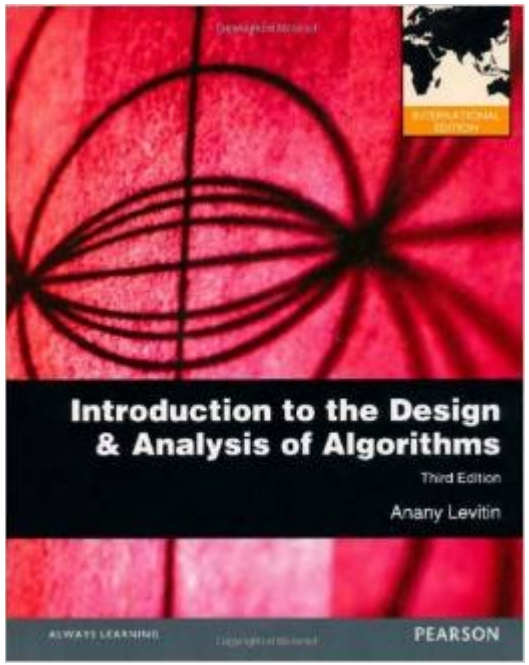
Algorithmen & Datenstrukturen

Effizienzanalyse von Algorithmen

Literaturangaben

Diese Lerneinheit basiert größtenteils auf dem Buch „The Design and Analysis of Algorithms“ von Anany Levitin.



In dieser Einheit behandelte Kapitel:
2.1 Analysis Framework
2.2 Asymptotic Notations and the Basic Efficiency Classes
2.3 Mathematical Analysis of Nonrecursive Algorithms
2.4 Mathematical Analysis of Recursive Algorithms

Analyse von Algorithmen

Zentrale Aspekte

- Korrektheit
- Laufzeiteffizienz
- Speicherplatzeffizienz
- Optimalität

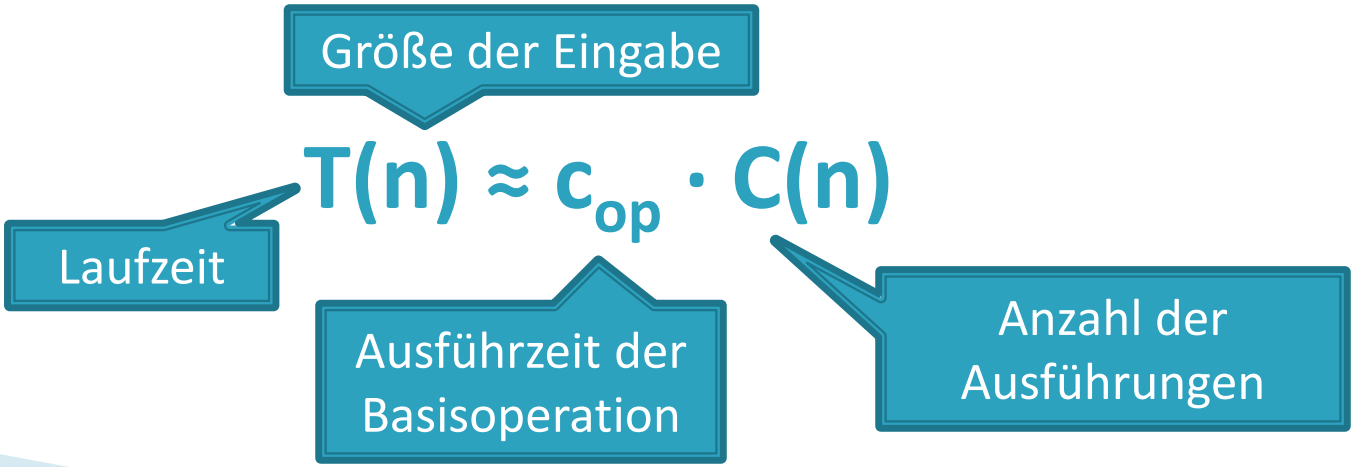
Ansätze

- Theoretische Analyse
- Empirische Analyse

Theoretische Analyse der Laufzeiteffizienz

Wie oft wird die Basisoperation des Algorithmus ausgeführt?

- **Basisoperation**: Operation, die am stärksten zur Laufzeit des Algorithmus beiträgt
- Anzahl der Ausführungen abhängig von der **Größe der Eingabe**



Größe der Eingabe und grundlegende Operationen – Beispiele

Problem	Maß für die Eingabegröße	Basisoperation
Suche nach einem Schlüssel in einer Liste mit n Elementen	Anzahl der Elemente der Liste	Schlüsselvergleich
Multiplikation zweier Matrizen	Matrizendimension oder Gesamtzahl der Zahlenelemente	Multiplikation zweier Zahlen
Überprüfung einer Ganzzahl n auf Primalität	Größe von n = Anzahl der Ziffern (in binärer Darstellung)	Division
Typisches Graphen-Problem	Anzahl der Knoten und/oder Kanten	Knoten besuchen oder Kante durchlaufen

Empirische Analyse der Laufzeiteffizienz

- Auswahl von bestimmten (typischen) Beispieleingaben verschiedener Größen
- Messe/zähle
  - Verstrichene Zeiteinheiten* (z. B. Millisekunden) oder
  - Anzahl der Ausführungen der Basisoperationen
- Analyse der empirisch ermittelten Daten

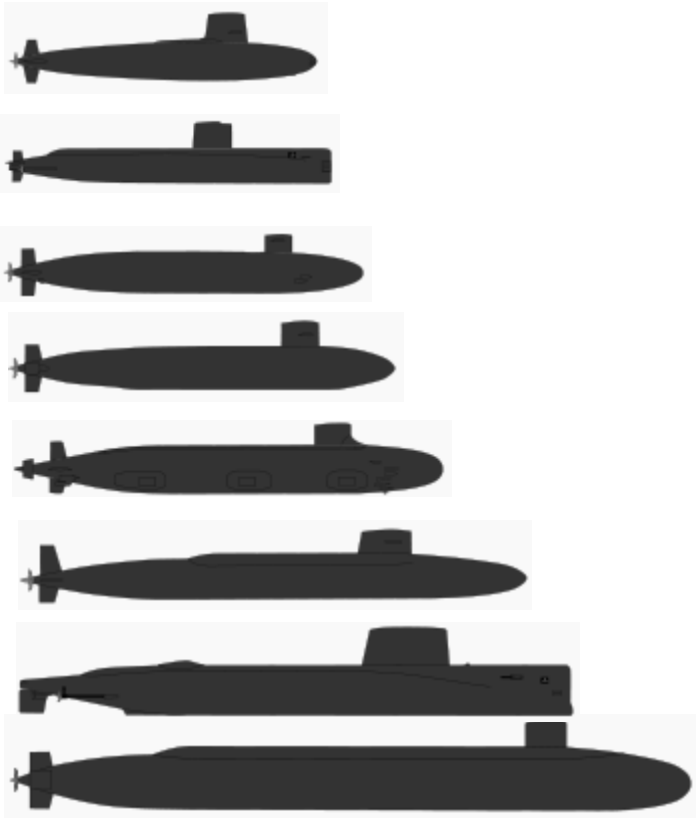
*) Welche weiteren Faktoren beeinflussen die Laufzeit eines Programms?

Größenordnung des Wachstums

- **Wichtigste Frage**: Wie stark wächst die Laufzeit/der Speicherbedarf mit zunehmender Größe der Eingabe ($n \to \infty$)?
- **Von geringerer Bedeutung**
  - Konstante Faktoren
  - Verhalten bei geringen Eingabegrößen
- **Typische Fragen**
  - Was gewinnt man durch Einsatz eines doppelt so schnellen Computers?
  - Wie viel länger dauert es, das Problem bei doppelter Eingabegröße zu lösen?

Häufig auftretende Wachstumsfunktionen

$O(1)$	konstant
$O(\log n)$	logarithmisch
$O(n)$	linear
$O(n \cdot \log n)$	überlogarithmisch
$O(n^2)$	quadratisch
$O(n^3)$	kubisch
$O(k^n)$	exponentiell
$O(n!)$	faktoriell

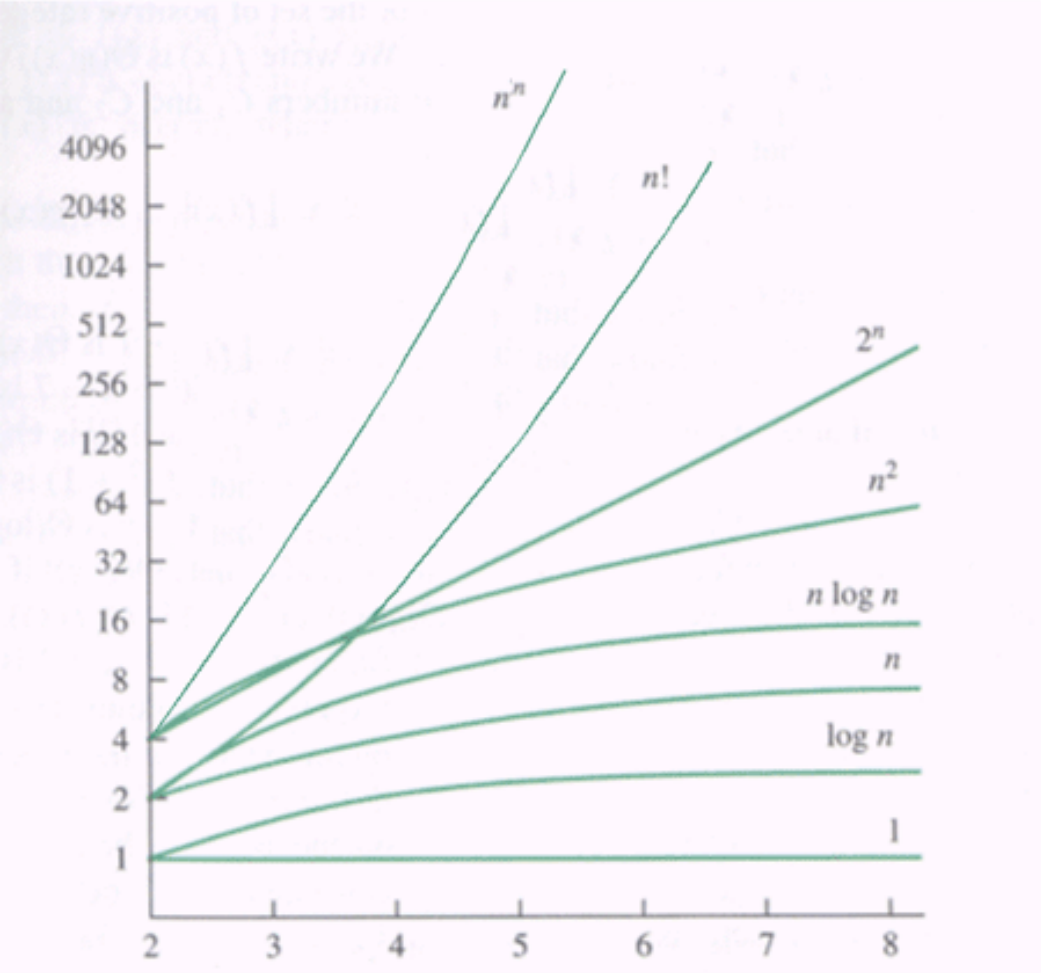


Sortierung in aufsteigender Reihenfolge

Werte wichtiger Wachstumsfunktionen für $n \to \infty$

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10^1	3,3	10^1	$3{,}3 \cdot 10^1$	10^2	10^3	10^3	$3{,}6 \cdot 10^6$
10^2	6,6	10^2	$6{,}6 \cdot 10^2$	10^4	10^6	$1{,}3 \cdot 10^{30}$	$9{,}3 \cdot 10^{157}$
10^3	10	10^3	$1{,}0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1{,}3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1{,}7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2{,}0 \cdot 10^7$	10^{12}	10^{18}		

Typische Wachstumsfunktionen im Vergleich



Bester Fall, mittlerer Fall, schlechtester Fall

Bei manchen Algorithmen hängt die Effizienz von der Beschaffenheit der Eingabe ab:

- **Schlechtester Fall (worst case)**
  - $C_{worst}(n)$: Maximum bei Eingaben der Größe n
- **Bester Fall (best case)**
  - $C_{best}(n)$: Minimum bei Eingaben der Größe n
- **Mittlerer Fall (average case)**
  - $C_{avg}(n)$: „Mittel“ bei Eingaben der Größe n
  - Annahmen über die Beschaffenheit und Wahrscheinlichkeit von „typischen“ Eingaben notwendig
  - NICHT Durchschnitt von bestem und schlechtestem Fall

Beispiel: Sequenzielle Suche

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//          or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

- Schlechtester Fall?
- Bester Fall?
- Mittlerer Fall?

Anzahl der grundlegenden Operationen – Formelarten

- Genaue Formel, z. B.:
  $C(n) = n(n-1)/2$
- Formel mit Angabe der Wachstumsfunktion und genauer multiplikativer Konstante, z. B.:
  $C(n) \approx 0{,}5 \cdot n^2$
- Formel mit Angabe der Wachstumsfunktion mit unbekannter multiplikativer Konstante, z. B.:
  $C(n) \approx c \cdot n^2$

Zusammenfassung der Analysemethodik

- Laufzeit- und Speicherplatzeffizienz wird als Funktion der Eingabegröße des Algorithmus ausgedrückt
  - **Laufzeiteffizienz**: Anzahl der Ausführungen der Basisoperation des Algorithmus
  - **Speicherplatzeffizienz**: Anzahl zusätzlich benötigter Speicherplätze
- Die Effizienz einiger Algorithmen kann bei Eingaben gleicher Größe erheblich schwanken
- Fokus: Größenordnung der Wachstumsfunktion des Algorithmus mit größer werdender Eingabegröße

Asymptotische Wachstumsanalyse

Ansatz, um konstante Faktoren und Besonderheiten bei kleinen Eingabegrößen ausblenden zu können

Groß-O

- **O(g(n))**: Klasse der Funktionen f(n), die **nicht schneller** als g(n) wachsen

Groß-Theta

- **Θ(g(n))**: Klasse der Funktionen f(n), die **gleich schnell** wie g(n) wachsen

Groß-Omega

- **Ω(g(n))**: Klasse der Funktionen f(n), die **mindestens so schnell** wie g(n) wachsen

Definitionen von Groß-O, Groß-Θ und Groß-Ω

Asymptotische obere Schranke

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

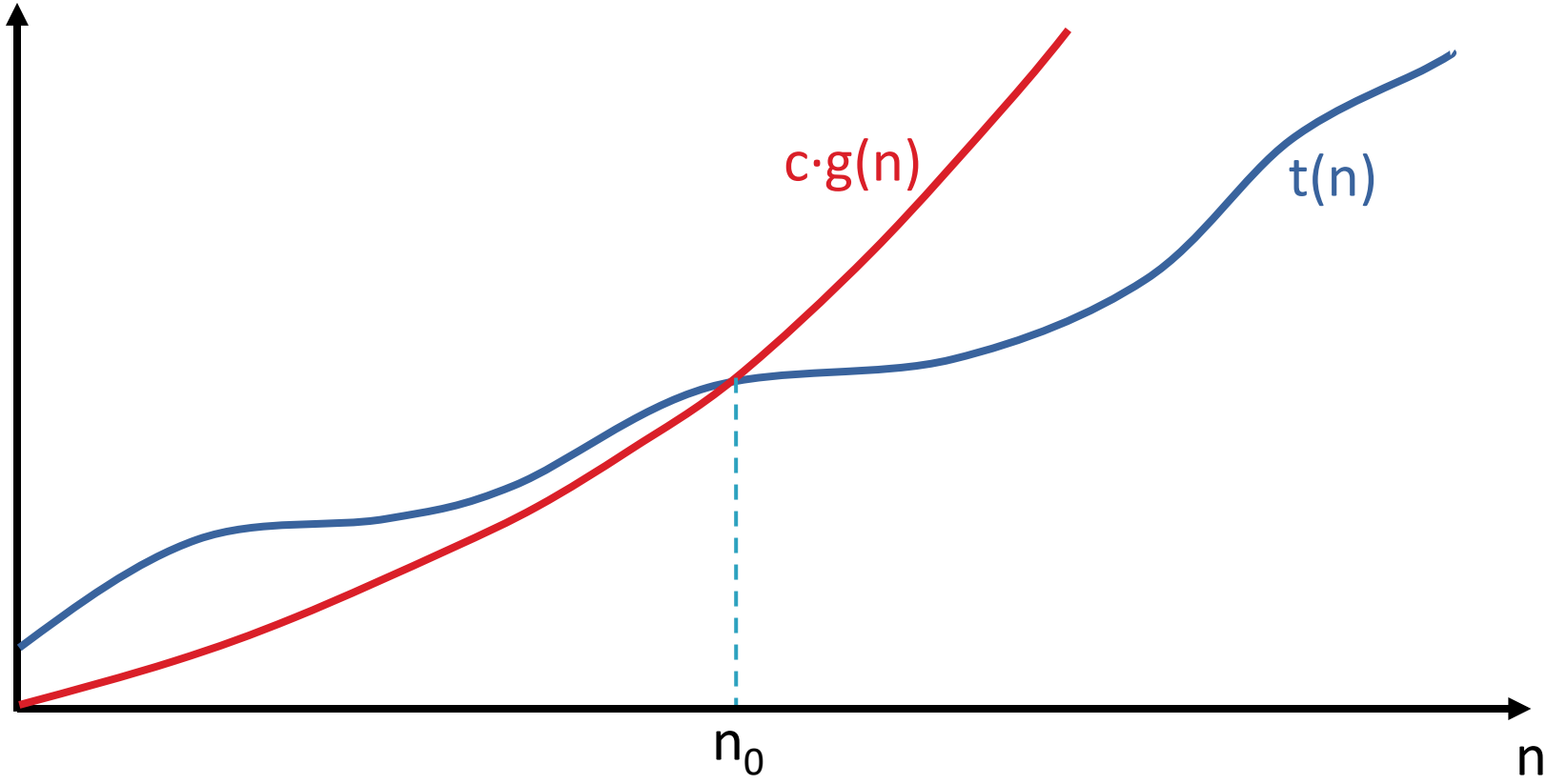
Asymptotische untere Schranke

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

Asymptotische scharfe/enge Schranke

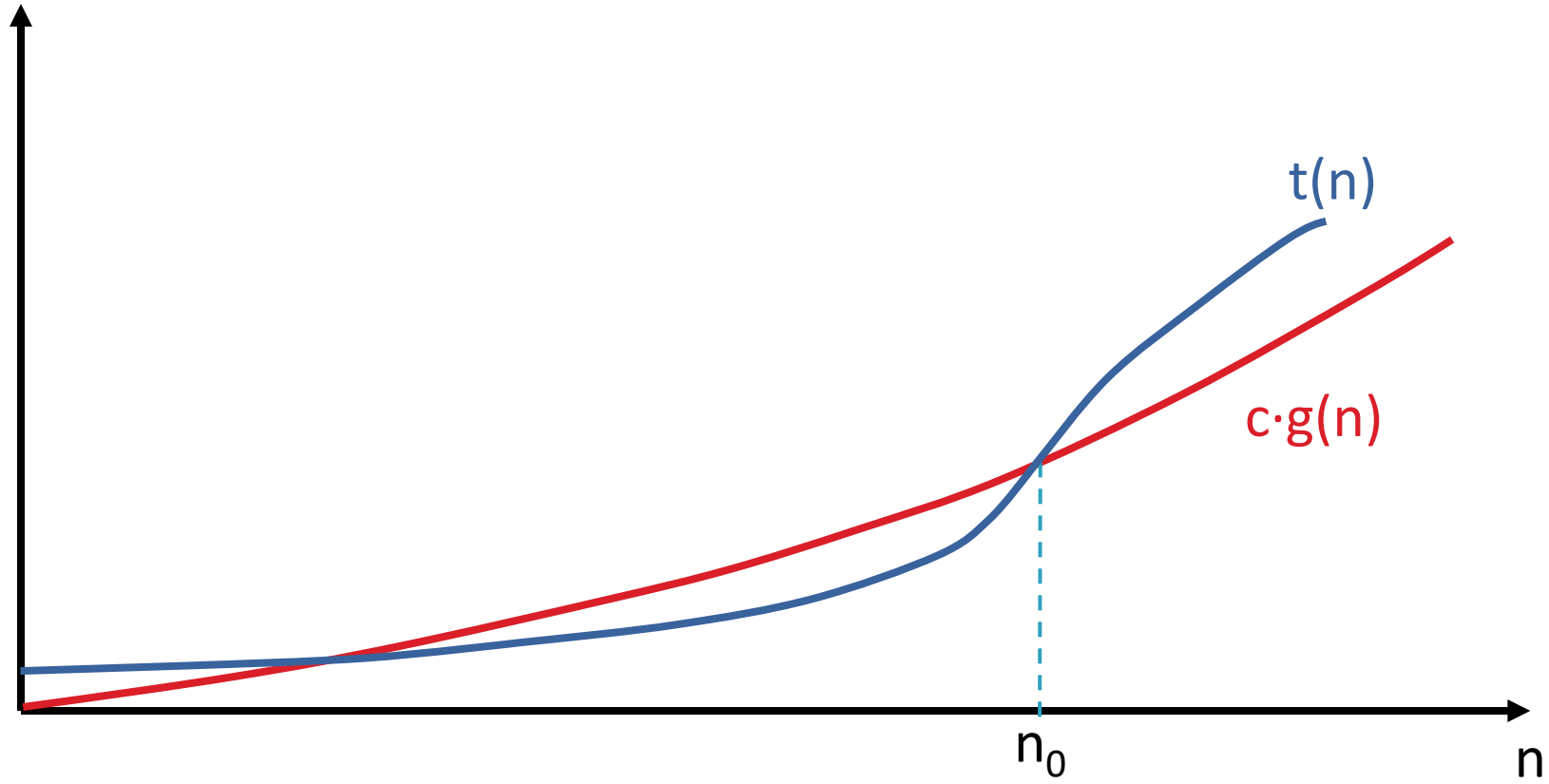
$$\Theta(g(n)) \stackrel{\text{def}}{=} \{ f \mid \exists c_1 > 0, c_2 > 0, n_0 > 0: \forall n \geq n_0: c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \}$$

Groß-O-Notation



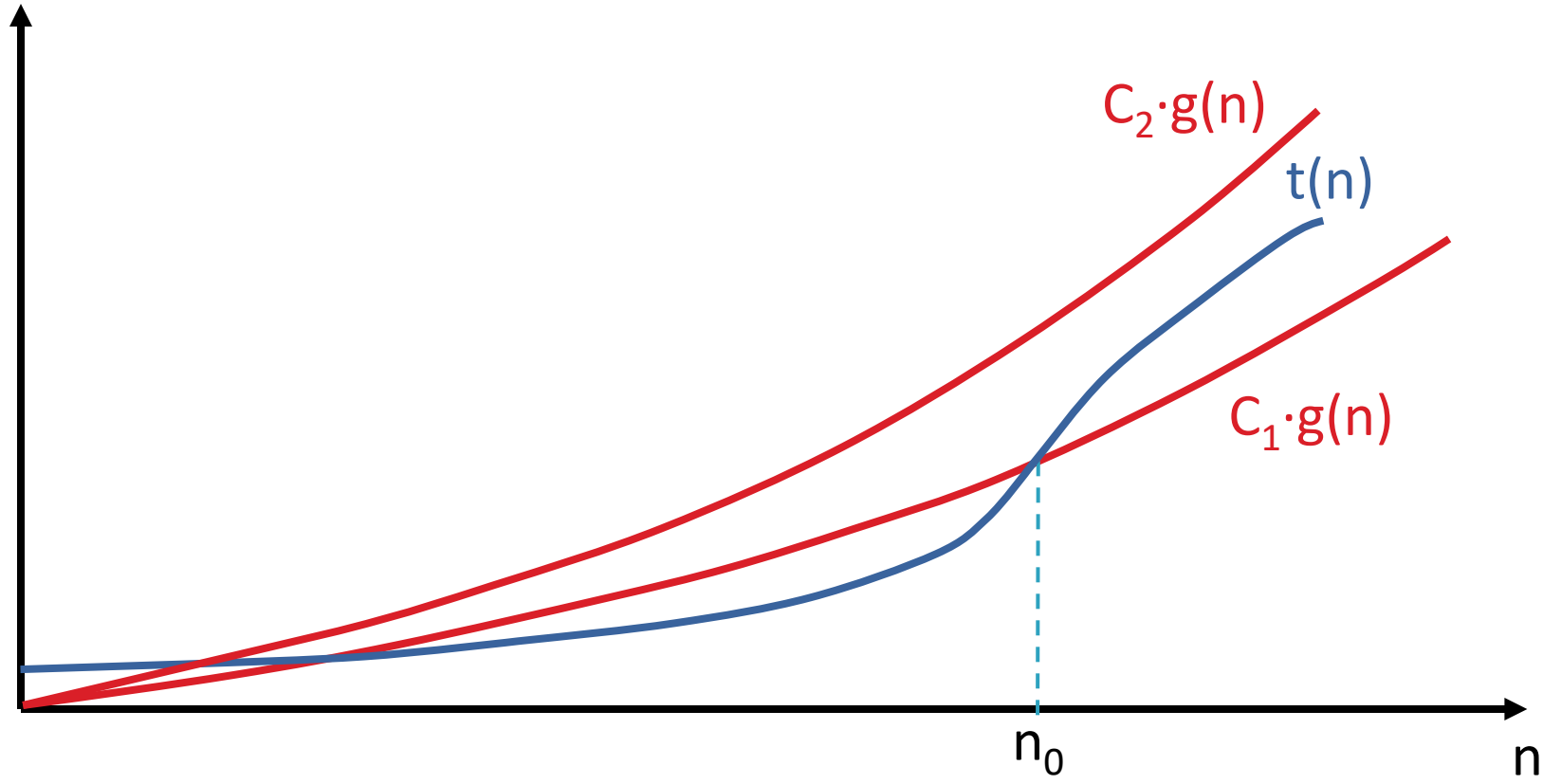
$t(n) \in O(g(n))$

Groß-Ω-Notation



$t(n) \in \Omega(g(n))$

Groß-Θ-Notation



$$t(n) \in \Theta(g(n))$$

Laufzeiteffizienz nicht-rekursiver Algorithmen

Allgemeines Vorgehen bei der Analyse

- Identifiziere **Parameter n**, der die Größe der Eingabe widerspiegelt
- Ermittle die **Basisoperation** des Algorithmus
- Bestimme den **besten**, **schlechtesten** und **mittleren Fall** für eine Eingabe der Größe n
- Erstelle **Summenformel** für die Anzahl der Ausführungen der Basisoperation
- Vereinfache die Summenformel mit Hilfe von **Formeln** und **Regeln**

Beispiel: Finde Maximum in Array

ALGORITHM *MaxElement*$(A[0..n-1])$

```
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

Beispiel: Alle Elemente einzigartig?

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//          and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

Laufzeiteffizienz rekursiver Algorithmen

Allgemeines Vorgehen bei der Analyse

- Identifiziere **Parameter n**, der die Größe der Eingabe widerspiegelt
- Ermittle die **Basisoperation** des Algorithmus
- Bestimme den **besten**, **schlechtesten** und **mittleren Fall** für eine Eingabe der Größe n
- Stelle eine **Rekursionsgleichung** mit zugehöriger **Abbruchbedingung** auf, die widerspielt, wie oft die Basisoperation ausgeführt wird.
- Löse die Rekursionsgleichung (oder bestimme zumindest die Größenordnung ihres Wachstums) durch **rückwärtiges Einsetzen**.

Beispiel: Rekursive Berechnung von n!

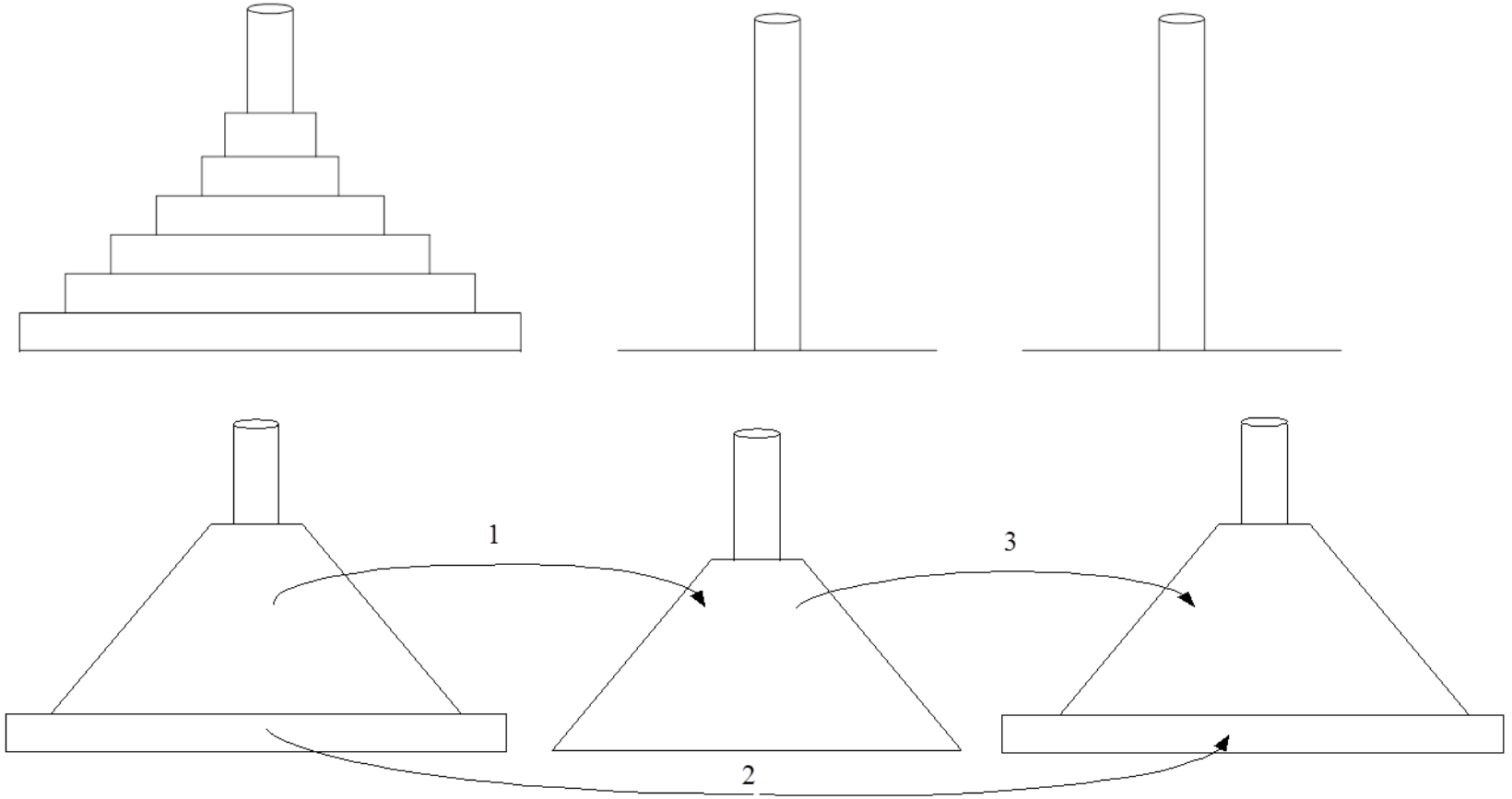
- **Definition:**
  $n! = 1 * 2 * \ldots * (n-1) * n$ für $n \geq 1$
  $0! = 1$
- **Rekursive Definition:**
  $F(n) = F(n-1) * n$ für $n \geq 1$
  $F(0) = 1$ für $n = 0$

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

- Größenparameter n?
- Basisoperation?
- Bester/schlechtester/ mittlerer Fall?
- Rekursionsgleichung?
- Lösung der Rekursionsgleichung?

Beispiel: Türme von Hanoi



Türme von Hanoi: Algorithmus

Algorithm *hanoi(n, p1, p2, p3)*

// Solves the problem of the Towers of Hanoi

// Input: number of disks n, starting, target, and auxiliary pole

// Output: All disks in the correct order on target pole

if n = 1

„Move disk from pole p1 to pole p2“ // Trivial case

else

hanoi(n-1, p1, p3, p2) // move n-1 disks from start to auxiliary pole

„Move disk from pole p1 to pole p2“ // move base disk

hanoi(n-1, p3, p2, p1) // move n-1 disk from auxiliary to target pole

Türme von Hanoi: Baum der Funktionsaufrufe