

Programmierung 1

Vorlesung 17

Livestream beginnt um 14:15 Uhr

Laufzeit
rekursiver Prozeduren

Programmierung 1

Wohlfundierte Induktion (Noethersche Induktion)

- ▶ Sei $\forall x \in X : A(x)$ eine **allquantifizierte Aussage** über eine Menge X .
- ▶ Eine **Induktionsrelation** $>$ ist eine **terminierende Relation** auf X .
Um die allquantifizierte Aussage zu beweisen, zeigen wir den **Induktionsschritt**:
für jedes Argument x folgt aus der Tatsache,
dass für jedes kleinere Argument y $A(y)$ gilt, dass $A(x)$ gilt.
- ▶ **Wohlfundierte Induktion:**

$$(\forall x \in X (\forall y \in X : x > y \Rightarrow A(y)) \Rightarrow A(x)) \Rightarrow \forall x \in X : A(x)$$

Induktionsschritt

Breite vs. Tiefe

Proposition 10.7 (Breite versus Tiefe) $\forall t \in \mathcal{B}: bt = 2^{dt}$.

Beweis Durch strukturelle Induktion über $t \in \mathcal{B}$. Wir unterscheiden zwei Fälle.

Sei $t = []$. Dann $bt = 1 = 2^{dt}$ gemäß der Definition von b und d .

Sei $t = [t_1, t_2]$. Dann gilt:

$$bt = bt_1 + bt_2$$

Definition b

$$= 2^{dt_1} + 2^{dt_2}$$

Induktion für t_1 und t_2

$$= 2 \cdot 2^{dt_1}$$

t balanciert, also $dt_1 = dt_2$

$$= 2^{1+dt_1}$$

$$= 2^{1+\max\{dt_1, dt_2\}}$$

t balanciert, also $dt_1 = dt_2$

$$= 2^{dt}$$

Definition d

Größe vs. Tiefe

Proposition 10.8 (Größe versus Tiefe) $\forall t \in \mathcal{B}: st = 2^{dt+1} - 1$.

Beweis Durch strukturelle Induktion über $t \in \mathcal{B}$. Wir unterscheiden zwei Fälle.

Sei $t = []$. Dann $st = 1 = 2^{dt+1} - 1$ gemäß der Definition von b und d .

Sei $t = [t_1, t_2]$. Dann gilt:

$$st = 1 + st_1 + st_2$$

Definition s

$$= 1 + 2^{dt_1+1} - 1 + 2^{dt_2+1} - 1$$

Induktion für t_1 und t_2

$$= 2 \cdot 2^{dt_1+1} - 1$$

t balanciert, also $dt_1 = dt_2$

$$= 2^{1+dt_1+1} - 1$$

$$= 2^{1+\max\{dt_1, dt_2\}+1} - 1$$

t balanciert, also $dt_1 = dt_2$

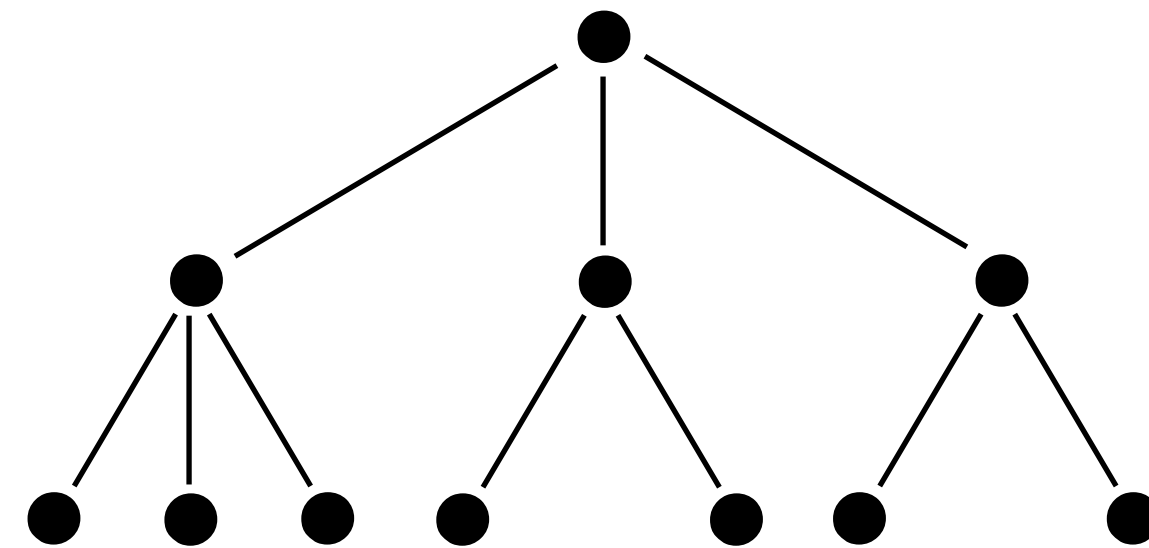
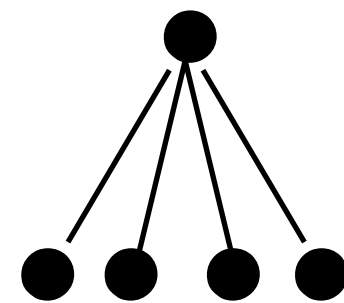
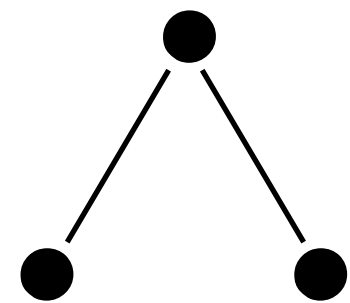
$$= 2^{dt+1} - 1$$

Definition d

Bäume mit mindestens zwei Nachfolgern

Wir betrachten Bäume $\mathcal{M} \subseteq \mathcal{T}$, bei denen **jeder innere Knoten mindestens zwei Nachfolger** hat:

1. $[] \in \mathcal{M}$.
2. Wenn $n \geq 2$ und $t_1, \dots, t_n \in \mathcal{M}$, dann $[t_1, \dots, t_n] \in \mathcal{M}$.



Bäume mit mindestens zwei Nachfolgern

$$b: \mathcal{T} \rightarrow \mathbb{N}_+$$

$$b[t_1, \dots, t_n] = \text{if } n = 0 \text{ then } 1 \text{ else } b t_1 + \dots + b t_n$$

$$s: \mathcal{T} \rightarrow \mathbb{N}_+$$

$$s[t_1, \dots, t_n] = \text{if } n = 0 \text{ then } 1 \text{ else } 1 + s t_1 + \dots + s t_n$$

Proposition 10.9 (Breite versus Größe) $\forall t \in \mathcal{M}: 2 \cdot b t > s t$.

Beweis Durch strukturelle Induktion über $t \in \mathcal{M}$. Wir unterscheiden zwei Fälle.

Sei $t = []$. Dann $2 \cdot b t = 2 > 1 = s t$ gemäß der Definition von b und s .

Sei $t = [t_1, \dots, t_n]$ mit $n \geq 2$. Dann gilt:

$$2 \cdot b t = 2(b t_1 + \dots + b t_n)$$

Definition b

$$= 2 \cdot b t_1 + \dots + 2 \cdot b t_n$$

$$\geq (s t_1 + 1) + \dots + (s t_n + 1)$$

Induktion für t_1, \dots, t_n

$$> s t_1 + \dots + s t_n + 1$$

$n \geq 2$

$$= s t$$

Definition s

Sekundäre Listenrekursion

$$s : \mathcal{T} \rightarrow \mathbb{N}_+$$

$$s[t_1, \dots, t_n] = \text{if } n = 0 \text{ then } 1 \text{ else } 1 + s t_1 + \dots + s t_n$$

- ▶ Neben der **primären Baumrekursion** verwenden die Prozeduren eine **sekundäre Listenrekursion** die durch "... " formuliert ist.
- ▶ In den **Anwendungsgleichungen** ist die sekundäre Listenrekursion nicht mehr sichtbar:

$$s[t_1, t_2] = 1 + s t_1 + s t_2$$

$$b[t_1, t_2, t_3] = b t_1 + b t_2 + b t_3$$

$$d[t_1, t_2] = 1 + \max\{d t_1, d t_2\}$$

- ▶ **Rekursionsfunktion:** $\lambda [t_1, \dots, t_n] \in \mathcal{T} . \langle t_1, \dots, t_n \rangle$
- ▶ **Terminierungsfunktion:** $\lambda t \in \mathcal{T} . t.$

Binäre Charakterisierung von Bäumen

$$\mathcal{T} := \mathcal{L}(\mathcal{T})$$

- ▶ Ein reiner **Baum** ist die **Liste** seiner **Unterbäume**.
- ▶ Daraus folgt: ein **Baum** ist
 - ▶ entweder die **leere Liste**
 - ▶ oder ein **Paar** $t::t'$ von **Bäumen**,
wobei t' die Liste der restlichen Unterbäume ist
- ▶ Wir können die Größe von Bäumen mit einer **binärrekursiven** Prozedur **ohne Sekundärrekursion** berechnen:

$$size : \mathcal{T} \rightarrow \mathbb{N}_+$$

$$size\ nil = 1$$

$$size(t :: t') = size\ t + size\ t'$$

Kapitel 11

Laufzeit

rekursiver Prozeduren

Laufzeitunterschied

```
fun rev nil      = nil  
  | rev (x::xr) = rev xr @ [x]
```

```
fun rev' xs = foldl op:: nil xs
```

- ▶ *rev'* hat eine sehr viel kürzere Laufzeit als *rev*.
- ▶ **Wir werden zeigen:**
 - ▶ *rev* hat **quadratische Komplexität**
 - ▶ *rev'* hat **lineare Komplexität**

Laufzeit

$$@ : \mathcal{L}(X) \times \mathcal{L}(X) \rightarrow \mathcal{L}(X)$$

$$nil@ys = ys$$

$$(x::xr)@ys = x::(xr@ys)$$

- Wovon hängt die Laufzeit der Prozedur ab?

`["l", "a", "n", "g", "s", "a", "m"] @ nil`

`["s", "c", "h", "n", "e", "l", "l"] @ nil`

`["f", "i", "x"] @ nil`

`["s", "c", "h", "n", "e", "l", "l"] @ ["e", "r"]`

- Die **Laufzeitfunktion einer Prozedur** gibt die Laufzeit abhängig von der **Größe der Argumente** an.

Konkatenation von Listen

$$@ : \mathcal{L}(X) \times \mathcal{L}(X) \rightarrow \mathcal{L}(X)$$

$$nil@ys = ys$$

$$(x::xr)@ys = x::(xr@ys)$$


► Rekursionsbaum:

$$([1,2,3], ys) \rightarrow ([2,3], ys) \rightarrow ([3], ys) \rightarrow ([], ys)$$

► Größenfunktion: $\lambda (xs, ys). |xs|.$

► Laufzeitfunktion: $\lambda n. n + 1.$

Laufzeit

- ▶ **Vorläufige Definition** (wird später verfeinert):
Die **Laufzeit einer Prozedur** für ein Argument x ist die **Größe des Rekursionsbaums** für x .
- ▶ Eine **Größenfunktion** für eine terminierende Prozedur $p: X \rightarrow Y$ ist eine **natürliche Terminierungsfunktion** $s \in X \rightarrow \mathbb{N}$ für p , die die folgende Bedingung erfüllt:
 $\forall n \in \mathbb{N} \exists k \in \mathbb{N} \forall x \in X:$
wenn $s\ x = n$, dann ist die Laufzeit von p für x kleiner als k .



Die Prozedur p sei wie folgt gegeben.

$$p: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$p(0, k) = 0$$

$$p(n, k) = p(n-1, 0) + \dots + p(n-1, k) \text{ für } k > 0.$$

Die Funktion $\lambda (n, k) \in \mathbb{N} \times \mathbb{N}. n$ ist...


A: eine natürliche
Terminierungsfunktion ✓

B: eine Größenfunktion ✗

C: beides ✗

D: weder noch ✗

Laufzeit

- ▶ **Vorläufige Definition** (wird später verfeinert):
Die **Laufzeit einer Prozedur** für ein Argument x ist die **Größe des Rekursionsbaums** für x .
- ▶ Eine **Größenfunktion** für eine terminierende Prozedur $p: X \rightarrow Y$ ist eine **natürliche Terminierungsfunktion** $s \in X \rightarrow \mathbb{N}$ für p , die die folgende Bedingung erfüllt:
 $\forall n \in \mathbb{N} \exists k \in \mathbb{N} \forall x \in X$:
wenn $s x = n$, dann ist die Laufzeit von p für x kleiner als k .
- ▶ Die Zahl $s x$ ist die **Größe** von x .
- ▶ Die **Laufzeitfunktion** von p **gemäß** s ist die Funktion $r \in \mathbb{N} \rightarrow \mathbb{N}_+$, die für jedes $n \in \mathbb{N}$ die **maximale Laufzeit** liefert, die p für Argumente der **Größe** n benötigt.
Wir vereinbaren: $r 0 = 1$, falls es keine Argumente der Größe 0 gibt;
 $r n = r(n-1)$, falls $n > 0$ und es keine Argumente der Größe n gibt.

Faltung von Listen

$$\textit{foldl} : (X \times Y \rightarrow Y) \times Y \times \mathcal{L}(X) \rightarrow Y$$

$$\textit{foldl}(f, s, \textit{nil}) = s$$

$$\textit{foldl}(f, s, x::xr) = \textit{foldl}(f, f(x, s), xr)$$

► Größenfunktion:

$$\lambda (f, s, xs). |xs|.$$

► Laufzeitfunktion:

$$\lambda n. n + 1.$$

Elementtest für Listen

$member: \mathbb{Z} \times \mathcal{L}(\mathbb{Z}) \rightarrow \mathbb{B}$

$member(x, nil) = 0$

$member(x, y::yr) = \text{if } x = y \text{ then } 1 \text{ else } member(x, yr)$

- ▶ **Größenfunktion:** $\lambda (x, xs). |xs|$.
- ▶ Laufzeit kann für eine Argument der Größe n jeden Wert zwischen 1 und $n+1$ annehmen.
- ▶ Wir sagen, dass die Laufzeit einer Prozedur **uniform** ist, wenn für jede Größe gilt, dass die Prozedur für **alle Argumente** dieser Größe die **gleiche Laufzeit** hat.
- ▶ Wenn die Laufzeit einer Prozedur **nicht uniform** ist, liegt der Laufzeitfunktion eine **worst-case Annahme** zu Grunde: $r\ n$ ist die **maximale** Laufzeit für Argumente der Größe n .
- ▶ **Laufzeitfunktion:** $\lambda n. n + 1$.



Die Laufzeit der Prozedur *fac* gemäß der Größenfunktion $\lambda n \in \mathbb{N}. n$ ist...

$fac: \mathbb{N} \rightarrow \mathbb{N}$

$fac\ n = \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot fac(n - 1)$

A: uniform:
 $\lambda n \in \mathbb{N}. n+1$ ✓

B: nicht uniform:
 $\lambda n \in \mathbb{N}. n+1$ ✗

C: uniform:
 $\lambda n \in \mathbb{N}. fac(n)$ ✗

D: nicht uniform:
 $\lambda n \in \mathbb{N}. fac(n)$ ✗

Fakultät

$$fac : \mathbb{N} \rightarrow \mathbb{N}$$

$$fac\ n = \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot fac(n - 1)$$

► Größenfunktion: $\lambda n. n$

► Laufzeitfunktion: $\lambda n. n + 1$ (uniform)

Rekursive Darstellung der Laufzeitfunktion:

$$r : \mathbb{N} \rightarrow \mathbb{N}_+$$

$$r0 = 1$$

$$r\ n = 1 + r(n - 1) \quad \text{für } n > 0$$

Balancierte Binärbäume

$$ntree : \mathbb{N} \rightarrow \mathcal{T}$$

$$ntree\ 0 = nil$$

$$ntree\ n = [ntree(n-1), ntree(n-1)] \quad \text{für } n > 0$$

► Größenfunktion: $\lambda n. n$

► Laufzeitfunktion: $\lambda n \in \mathbb{N}. 2^{n+1} - 1$ (uniform)

Rekursive Darstellung der Laufzeitfunktion:

$$r : \mathbb{N} \rightarrow \mathbb{N}_+$$

$$r\ 0 = 1$$

$$r\ n = 1 + r(n-1) + r(n-1) \quad \text{für } n > 0$$

Laufzeit konkret

Anzahl Prozeduraufrufe (PA)	Ausführungszeit bei 10^9 PA pro Sekunde	
	in Sekunden	etwa
10^4	10^{-5}	10 Mikrosekunden
10^6	10^{-3}	1 Millisekunde
10^9	10^0	1 Sekunde
10^{11}	10^2	2 Minuten
10^{13}	10^4	3 Stunden
10^{14}	10^5	1 Tag
10^{15}	10^6	2 Wochen
10^{16}	10^7	4 Monate
10^{17}	10^8	3 Jahre
10^{19}	10^{10}	3 Jahrhunderte
10^{20}	10^{11}	3 Jahrtausende
10^{21}	10^{12}	ewig

Laufzeit konkret

Größe	Laufzeitfunktion			
n	linear n	quadratisch n^2	kubisch n^3	exponentiell 2^n
	Ausführungszeit bei 10^9 Prozeduraufrufen pro Sekunde			
10^3	10^{-6} Sekunden	10^{-3} Sekunden	1 Sekunde	ewig
10^4	10^{-5} Sekunden	10^{-1} Sekunden	20 Minuten	ewig
10^5	10^{-4} Sekunden	10 Sekunden	10 Tage	ewig
10^6	10^{-3} Sekunden	20 Minuten	30 Jahre	ewig
10^7	10^{-2} Sekunden	1 Tag	ewig	ewig

Laufzeiten und Komplexitäten

Um die Laufzeit einer Prozedur beurteilen zu können, genügt es, die **Komplexität** ihrer Laufzeitfunktion zu kennen:

$\lambda n. n$ lineare Komplexität
 $\lambda n. n^2$ quadratische Komplexität
 $\lambda n. n^3$ kubische Komplexität
 $\lambda n. 2^n$ exponentielle Komplexität

Prozedur	Größenfunktion	Laufzeitfunktion	Komplexität
@	$\lambda (xs, ys). xs $	$\lambda n. n + 1$	$O(n)$
<i>foldl</i>	$\lambda (f, s, xs). xs $	$\lambda n. n + 1$	$O(n)$
<i>member</i>	$\lambda (x, xs). xs $	$\lambda n. n + 1$	$O(n)$
<i>ntree</i>	$\lambda n. n$	$\lambda n. 2^{n+1} - 1$	$O(2^n)$

Komplexitätsklassen

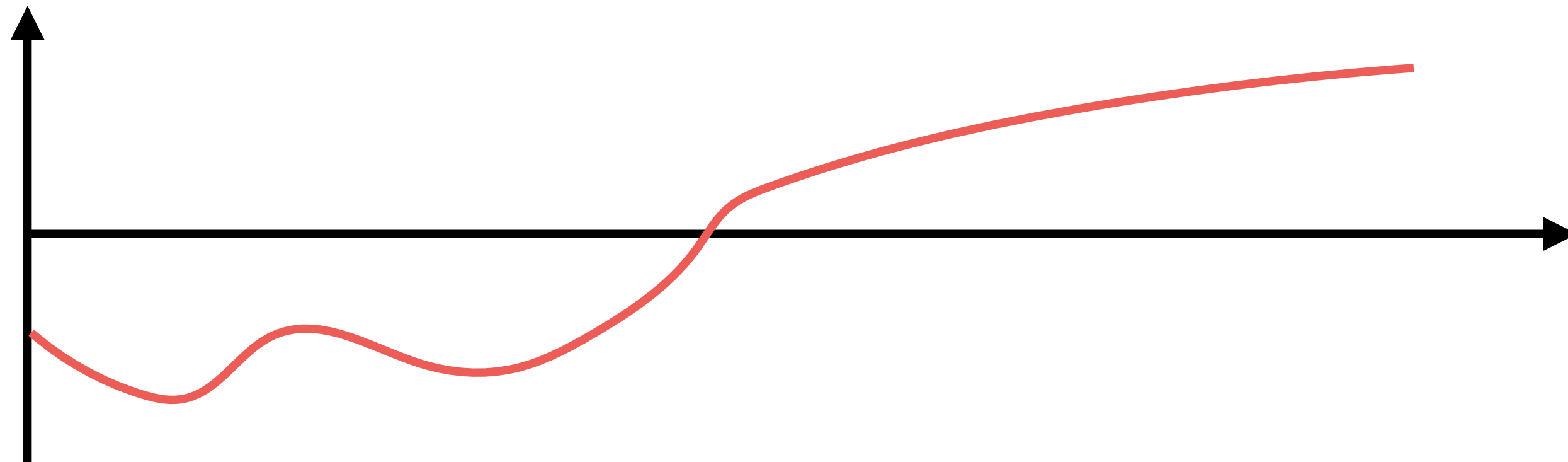
- ▶ **Laufzeit** ist nur von der **Größenordnung** her interessant.
- ▶ **Wichtige Komplexitätsklassen**

$O(1)$	konstante Komplexität
$O(\log n)$	logarithmische Komplexität
$O(n)$	lineare Komplexität
$O(n \cdot \log n)$	linear-logarithmische Komplexität
$O(n^2)$	quadratische Komplexität
$O(n^3)$	kubische Komplexität
$O(b^n)$	exponentielle Komplexität ($b > 1$)

O-Notation

- ▶ **O-Funktionen** sind Funktionen des Typs $\mathbb{N} \rightarrow \mathbb{R}$ die **fast überall** (= überall bis auf endlich viele Ausnahmen) **nicht negativ** sind.

$$OF := \{f \in \mathbb{N} \rightarrow \mathbb{R} \mid \exists n_0 \in \mathbb{N} \forall n \geq n_0: f n \geq 0\}$$



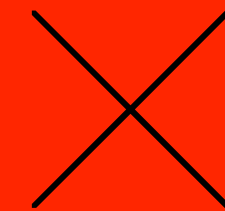


Welche der folgenden Funktionen
sind O-Funktionen?

A: $\lambda x \in \mathbb{N}. x-50$



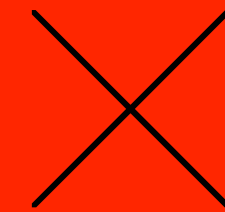
B: $\lambda x \in \mathbb{N}. 2x-x^2$



C: $\lambda x \in \mathbb{N}. x^2-2x$



D: $\lambda x \in \mathbb{N}. \sin(x)$

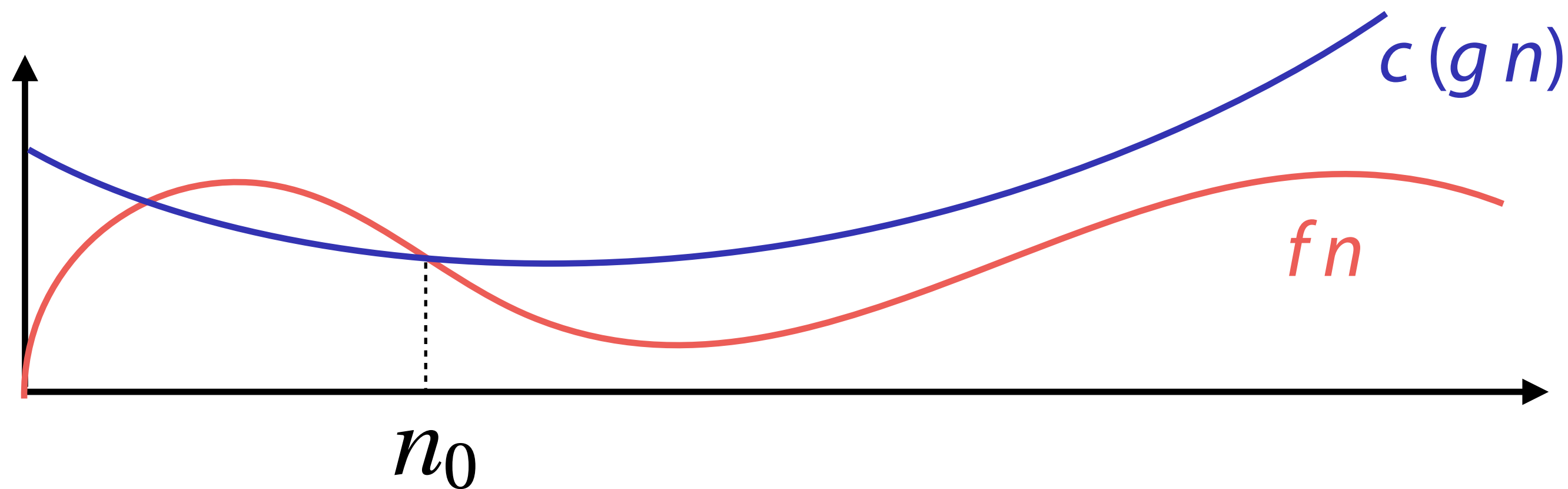


O-Notation

- ▶ Eine O-Funktion f **wird dominiert von** einer O-Funktion g , wenn es einen Faktor c gibt, so dass **fast überall** $f n \leq c(g n)$ gilt.



$$f \leq g \quad :\Longleftrightarrow \quad \exists n_0 \in \mathbb{N} \exists c \in \mathbb{N} \forall n \geq n_0: f n \leq c(g n)$$



Beispiele: $\lambda n \in \mathbb{N}. 270 \leq \lambda n \in \mathbb{N}. 1$
 $\lambda n \in \mathbb{N}. 11n + 273 \leq \lambda n \in \mathbb{N}. n$



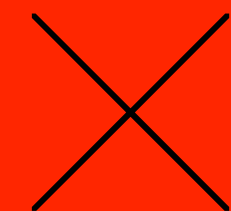
Dominiert die O-Funktion $\lambda n \in \mathbb{N}. n^3$

die O-Funktion $\lambda n \in \mathbb{N}. n^3 + n^2$?

A: Ja



B: Nein



Eigenschaften der Dominanz

Proposition 11.1 Für alle $f, g, h \in OF$ gilt:

1. $f \preceq f$ (Reflexivität von \preceq)
2. $f \preceq g \wedge g \preceq h \implies f \preceq h$ (Transitivität von \preceq)

► **Beweis zu 1:** Wähle $n_0=0$ und $c=1$.

► **Beweis zu 2:**

- Sei $n_{0,1}$ und c_1 so dass $\forall n \geq n_{0,1} \ f n \preceq c_1 (g n)$.
- Sei $n_{0,2}$ und c_2 so dass $\forall n \geq n_{0,2} \ g n \preceq c_2 (h n)$.
- Wähle $n_0 = \max \{n_{0,1}, n_{0,2}\}$ und $c = c_1 \cdot c_2$.
- Für alle $n \geq n_0$ gilt: $f n \preceq c_1 (g n) \preceq c_1 (c_2 (h n)) = c_1 \cdot c_2 (h n) = c (h n)$.

$$f \preceq g \quad :\Longleftrightarrow \quad \exists n_0 \in \mathbb{N} \ \exists c \in \mathbb{N} \ \forall n \geq n_0: \ f n \preceq c(g n)$$

Eigenschaften der Dominanz

- ▶ **Antisymmetrie** nicht zwingend!
- ▶ Dominanzrelation kann von unnötigen Details abstrahieren.
- ▶ **Beispiel:**

$$\lambda n \in \mathbb{N}. n^3 \preceq \lambda n \in \mathbb{N}. 33n^3 + 22n^2 + 11 \preceq \lambda n \in \mathbb{N}. n^3$$

$$f \preceq g \quad :\Longleftrightarrow \quad \exists n_0 \in \mathbb{N} \ \exists c \in \mathbb{N} \ \forall n \geq n_0: \ f n \leq c(g n)$$

Komplexität einer O-Funktion

- ▶ **Die Komplexität einer O-Funktion** f ist die Menge aller O-Funktionen, die höchstens so komplex wie f sind.

$$O(f) := \{g \in OF \mid g \preceq f\}$$

- ▶ Aufgrund der **Inklusionsordnung** liefert dies eine **Ordnung für Komplexitäten**.

$$O(\lambda n. 1) = O(\lambda n. 133) \subset O(\lambda n. n) = O(\lambda n. 7n - 26) \subset O(\lambda n. n^2)$$

- ▶ **Konvention:** Lambda-Präfix wird typischerweise weggelassen.

$$O(\lambda n \in \mathbb{N}. f n) \longrightarrow O(f n)$$

$$O(\lambda n \in \mathbb{N}. n^2) \longrightarrow O(n^2)$$

Komplexitätshierarchie

$$O(n) \subsetneq O(n^2)$$

Beweis:

▶ $\lambda n. n \leq \lambda n. n^2$ (einfach)

▶ $\lambda n. n \not\leq \lambda n. n^2$

Beweis durch Widerspruch:

Annahme: $\lambda n. n^2 \leq \lambda n. n$, also

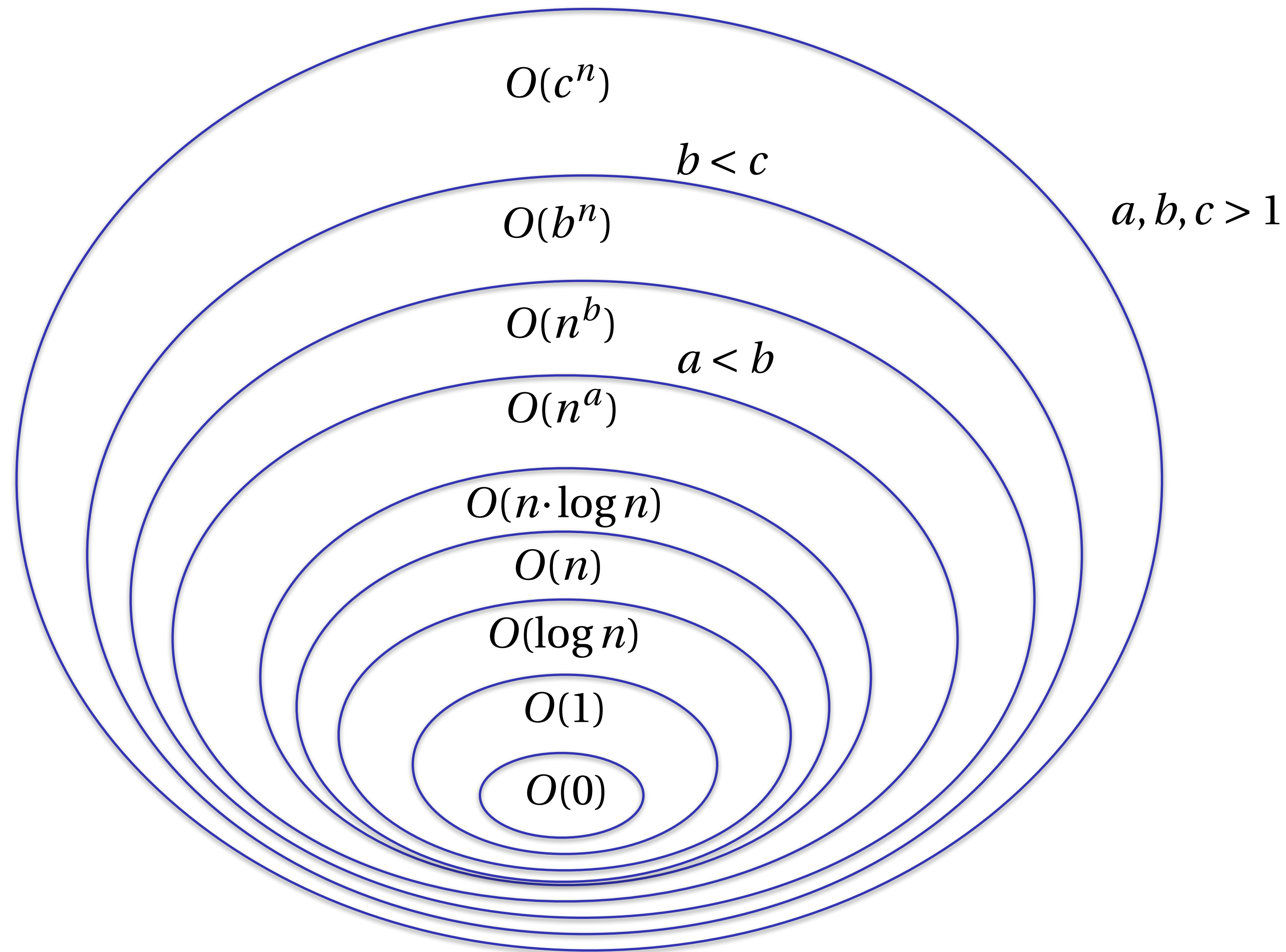
$\exists n_0 \in \mathbb{N} \exists c \in \mathbb{N}$ so dass $\forall n \geq n_0 : n^2 \leq c n$

Dies impliziert dass $\exists n_0 \in \mathbb{N} \exists c \in \mathbb{N} \forall n \geq n_0 : n \leq c$.

Widerspruch.

$$f \leq g \quad :\Longleftrightarrow \quad \exists n_0 \in \mathbb{N} \exists c \in \mathbb{N} \forall n \geq n_0 : f n \leq c(g n)$$

Komplexitätshierarchie (Proposition 11.3)



$\log n := \text{if } n = 0 \text{ then } 0 \text{ else } \log_2 n$

www.prog1.saarland