Technische Universität Braunschweig



Algorithmen und Datenstrukturen – Übung #2

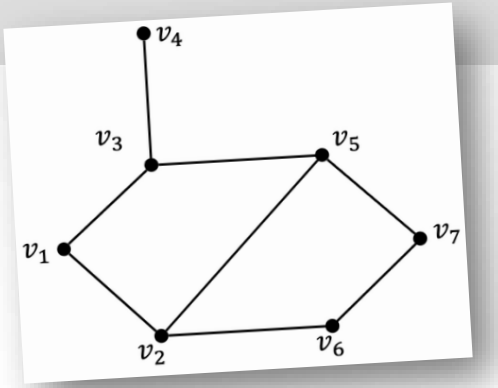
BFS, DFS und Wachstum von Funktionen

Ramin Kosfeld & Chek-Manh Loi

26.11.2026

Unser Programm

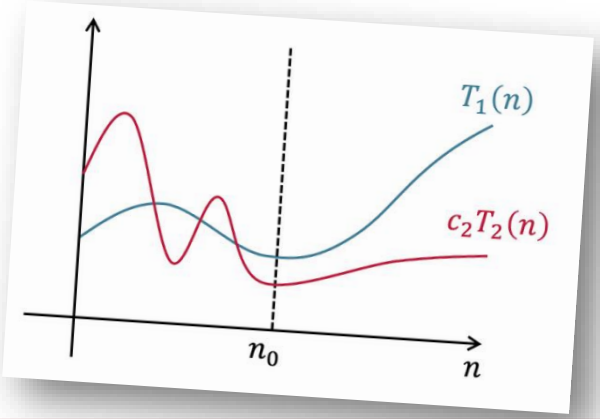
- Wiederholung Breiten- und Tiefensuche
- Laufzeiten und O-Notation
- Codierungsgrößen



Dezimal	Unär	Binär	Oktal	Hexadezimal
27	𝍸 𝍸 𝍸 𝍸 𝍸 II	11011	33	1B

b-adische Notation

- Dezimal (10-adisch)
- Binär (2-adisch)
- Oktal (8-adisch)
- Hexadezimal (16-adisch)
- Allgemein: $a_m a_{m-1} \dots a_1 a_0, a_{-1} a_{-2} \dots a_{-\infty}$ wobei $n = \sum_{i=-\infty}^{m} a_i b^i$ und $0 \leq a < b$



Zeit für ein Quiz!



Ordnet die folgenden Dinge in zwei Gruppen.

Tiefensuche

(BFS)

first-in-last-out



Stapel / Stack

Breitensuche

(DFS)

Warteschlange / Queue

first-in-first-out

Zeit für ein Quiz!

Ordnet die folgenden Dinge in zwei Gruppen.



Tiefensuche (DFS)

Stapel / Stack

first-in-last-out

Breitensuche (BFS)

Warteschlange / Queue

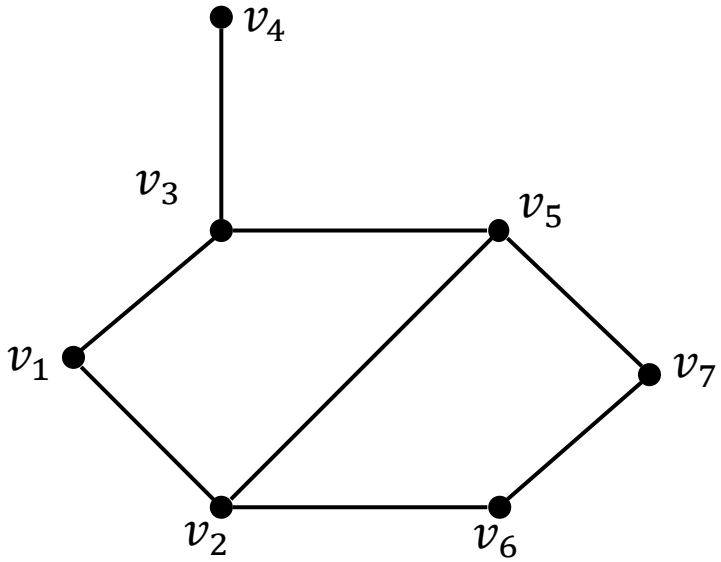
first-in-first-out



Suche in Graphen

Breitensuche

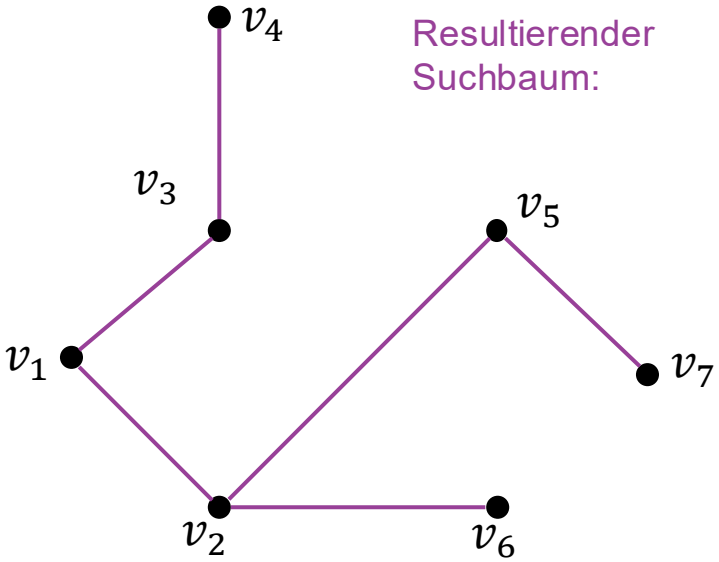
Benutze Warteschlange
Prinzip: First-in-first-out



← out	← in
v_1	
v_1, v_2	
v_1, v_2, v_3	
v_2, v_3	
v_2, v_3, v_5	
v_2, v_3, v_5, v_6	
v_3, v_5, v_6	
v_3, v_5, v_6, v_4	
v_5, v_6, v_4	
v_5, v_6, v_4, v_7	
v_6, v_4, v_7	
v_4, v_7	
v_7	
$\emptyset$	

Breitensuche

Benutze Warteschlange
Prinzip: First-in-first-out

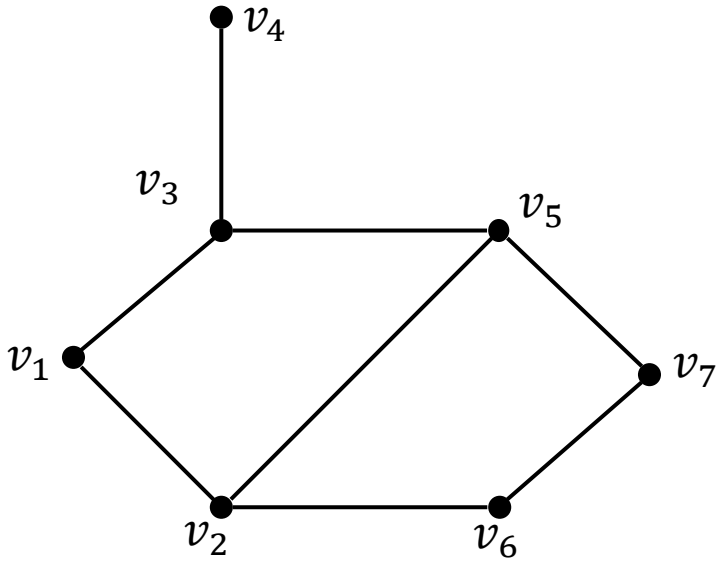


Resultierender Suchbaum:

← out	← in
v_1	
v_1, v_2	
v_1, v_2, v_3	
v_2, v_3	
v_2, v_3, v_5	
v_2, v_3, v_5, v_6	
v_3, v_5, v_6	
v_3, v_5, v_6, v_4	
v_5, v_6, v_4	
v_5, v_6, v_4, v_7	
v_6, v_4, v_7	
v_4, v_7	
v_7	
$\emptyset$	

Tiefensuche

Benutze Stapel
Prinzip: Last-in-first-out



v_1 ← in
→ out
v_1, v_2
v_1, v_2, v_5
v_1, v_2, v_5, v_3
v_1, v_2, v_5, v_3, v_4
v_1, v_2, v_5, v_3
v_1, v_2, v_5
v_1, v_2, v_5, v_7
v_1, v_2, v_5, v_7, v_6
v_1, v_2, v_5, v_7
v_1, v_2, v_5
v_1, v_2
v_1
$\emptyset$

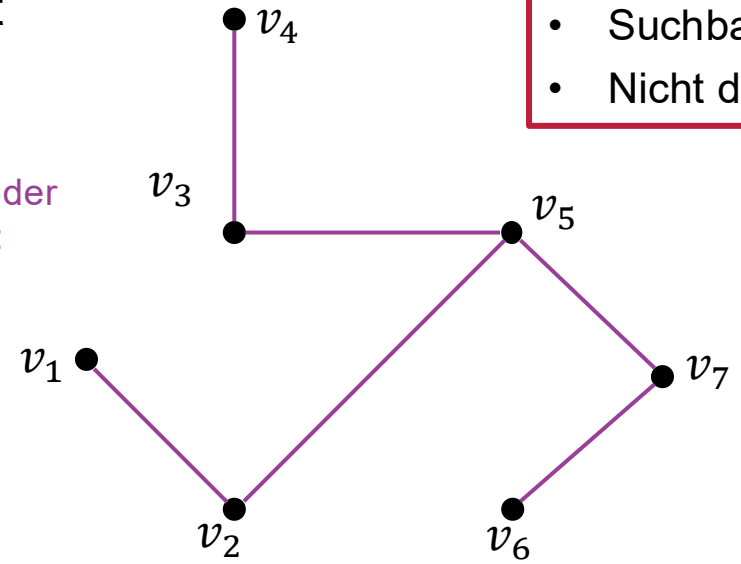
Tiefensuche

Benutze Stapel
Prinzip: Last-in-first-out

Beliebte Fehler

- Leere Menge am Ende vergessen
- Nicht jede Änderung angegeben
- Suchbaum nicht angegeben
- Nicht den kleinsten Index beachtet

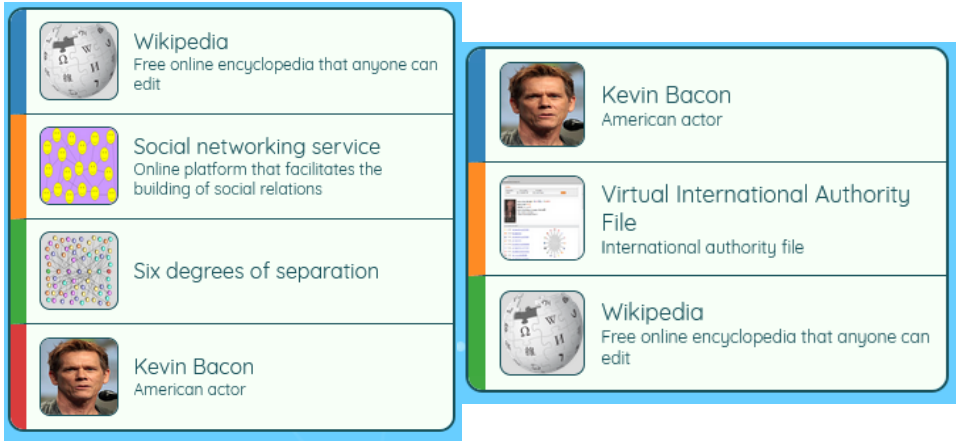
Resultierender Suchbaum:



v_1, v_2, v_5, v_3, v_4
v_1, v_2, v_5, v_3
v_1, v_2, v_5
v_1, v_2, v_5, v_7
v_1, v_2, v_5, v_7, v_6
v_1, v_2, v_5, v_7
v_1, v_2, v_5
v_1, v_2
v_1
$\emptyset$

Six Degrees of Kevin Bacon



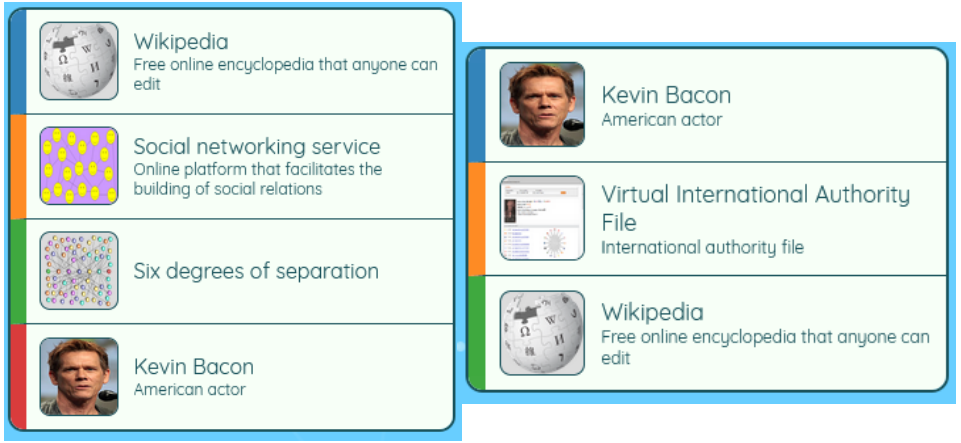


Finde kürzesten Weg von einem Schauspieler über Filme zu Kevin Bacon

Finde kürzesten Weg von einer Seite über enthaltene Links zu einer anderen

Six Degrees of Kevin Bacon vs. Six Degrees of Wikipedia





Finde kürzesten Weg von einem Schauspieler über Filme zu Kevin Bacon

Ungerichteter Graph

Hin- und Rückweg sind gleich lang

Finde kürzesten Weg von einer Seite über enthaltene Links zu einer anderen

Gerichteter Graph

Hin- und Rückweg können unterschiedlich lang sein

Zeit für ein Labyrinth

Average BFS Fan



Weakest DFS Enjoyer



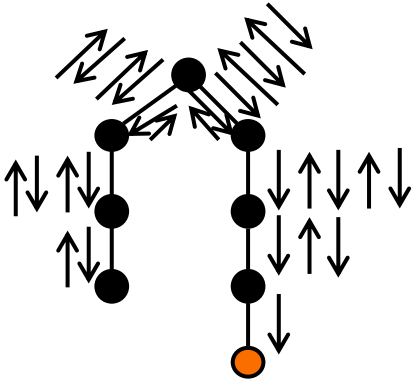
Zeit für ein Labyrinth

Wie oft muss man *alleine* durch die Gänge des Labyrinths laufen, wenn man…

1. BFS
2. oder DFS nutzt?

BFS - Worst-Case

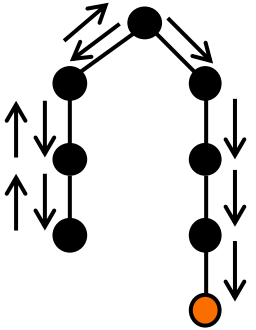
$\Omega(n^2)$ mal über Kanten laufen!



DFS schafft das schneller!

Man kann zeigen:

- Jede Kante wird maximal zwei Mal benutzt.
- Man benötigt maximal $2n - 1$ Schritte
- Es gibt Bäume, bei denen $2n - 1$ Schritte benötigt werden



Wachstum von Funktionen

Motivation

Wir haben einen coolen Algorithmus entworfen.

Wir wollen unseren Pseudocode einordnen bezüglich …

- Laufzeit
- Speicher

in Abhängigkeit von der Größe

Wie geben wir den Sp

$s(n) = 51G + 42n + \lfloor n/4$

… die Formel gilt für die Implementierung in C++17, di… Formel für die Implementierun… in Java ist …

G … auf 32 Bit Systemen 4, auf 64 Bit Geräten 8, …



Wie geben wir Laufzeiten an?

Zahl der Rechenschritte? …nen etc…

Zeit in Sekunden?

…l: …2

Welche Programmiersprache? C#? JavaScript? Python? Rust? …

Mit Wunderchip ABC kann man 4 Additionen auf einmal machen!

…1,2x so schnell!

Leute, *wollen* wir das überhaupt implementieren? Das ist nen theoretischer Algorithmus und der würde überhaupt nie in den Speicher passen …

Wir arbeiten mit *Pseudocode*

- Die genaue Laufzeit und Speicherbedarf sind von der Implementierung und dem genauen Setup abhängig
- Gerade, wenn wir nur mit Pseudocode arbeiten, ist es vielleicht gar nicht sinnvoll, eine genaue Funktion anzugeben

→ Wir wollen nur eine grobe Abschätzung,
eine Reduzierung auf das Wesentliche
(die für alle Implementierungen zutrifft)

→ **„Wie (gut) skaliert das?“**

… wo wir hinwollen (wie Leute tatsächlich über Laufzeiten sprechen):

Mit Informatikern beim Mittagessen

… und das kann man dann in Linearzeit verarbeiten …

Ne, das ist leider exponentiell …

… man kann zeigen, dass man *mindestens* quadratische Laufzeit dafür braucht

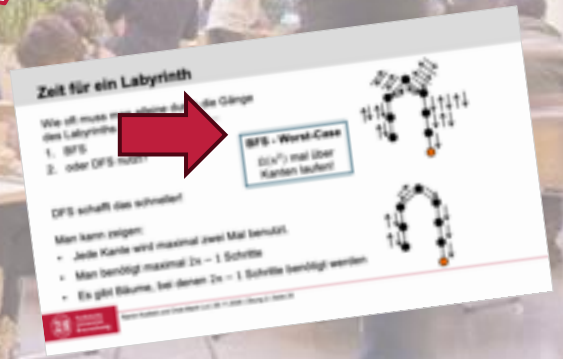
mit nem Trick kriegst du das tatsächlich in $O(1)$ hin …

… das läuft dann immer in $\Theta(n^2)$

Immerhin noch $O(n^4)$ Speicherbedarf …

Damit hast du im Durchschnitt konstante Zugriffszeit!

$O(m \cdot n^2)$ Speicher schafft man schon mit nem trivialen Verfahren



*das ist natürlich nur Mittagstischgerede und auch noch nicht komplett präzise, gerade ob wir neben $\mathcal{O}$ auch in Θ sind, ist hier oft nicht erkennbar

Die Idee

Wir wollen Funktionen einordnen …

Unabhängig von
konstanten Faktoren
– die kennen wir *eh nicht*

Betrachte die Entwicklung
auf lange Sicht
– für *immer größere* Eingaben

→ Ordne so das Wachstum der Funktion in eine Klasse ein

Die Idee

Betrachte Laufzeit / Speicherverbrauch etc. als Funktion $T: \mathbb{N} \to \mathbb{R}^+$

→ Die Funktion ist abhängig von der *Größe der Eingabe* n.

Wir vergleichen jetzt das Wachstum der beiden Funktionen $T_1(n)$ und $T_2(n)$.

$\mathcal{O}$-Notation

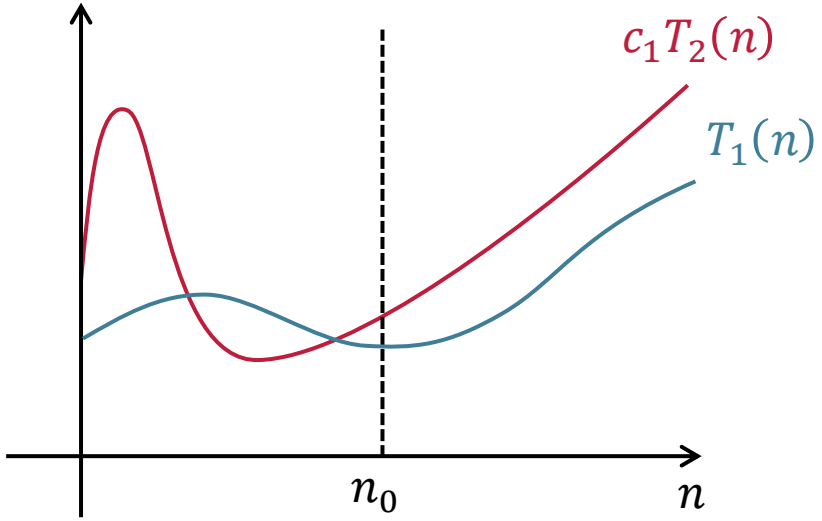
Definition:

Es gibt Konstanten $n_0 \in \mathbb{N}$ und $c_1 \in \mathbb{R}^+$,
sodass für alle $n \geq n_0$ gilt:

$$0 \leq T_1(n) \leq c_1 T_2(n)$$

$$\Leftrightarrow$$

$$T_1(n) \in \mathcal{O}\big(T_2(n)\big)$$



„T_1 wächst (asymptotisch) höchstens so schnell wie T_2“

Wie zeigen wir Zugehörigkeit?

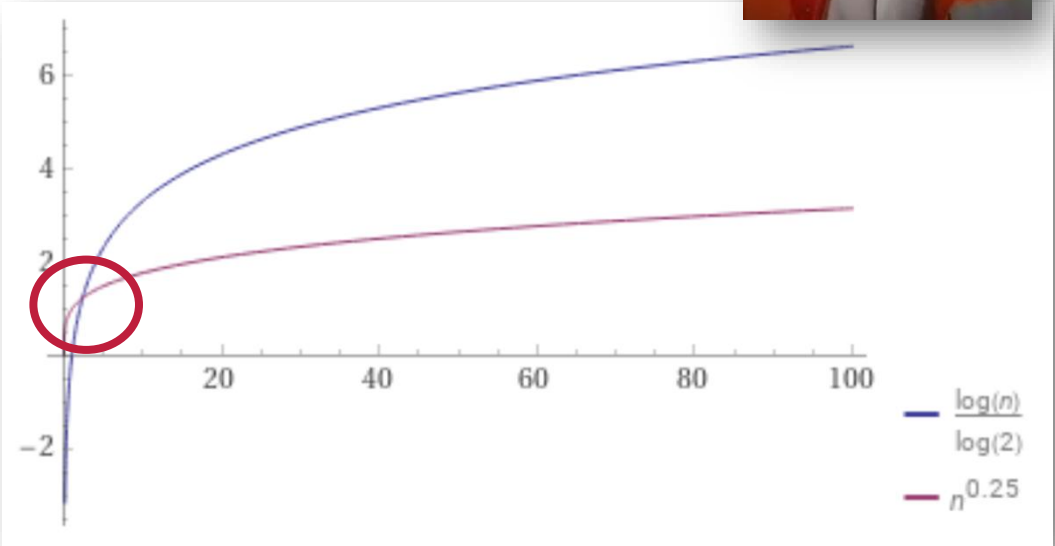


Ist $n^{0.25} \in \mathcal{O}(\log_2 n)$?

Ja! Abbildung rechts zeigt das:
Wähle $c = 1, n_0 \geq 5$

(Sieht man ja)

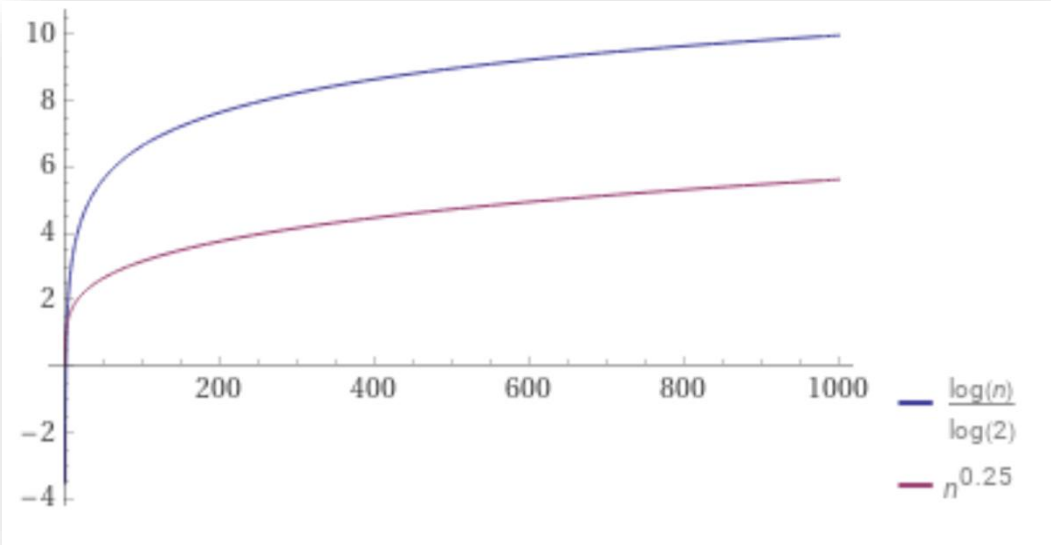




Wie zeigen wir Zugehörigkeit?

Ist $n^{0.25} \in \mathcal{O}(\log_2 n)$?

Ja! Abbildung rechts zeigt das:
Wähle $c = 1, n_0 \geq 5$



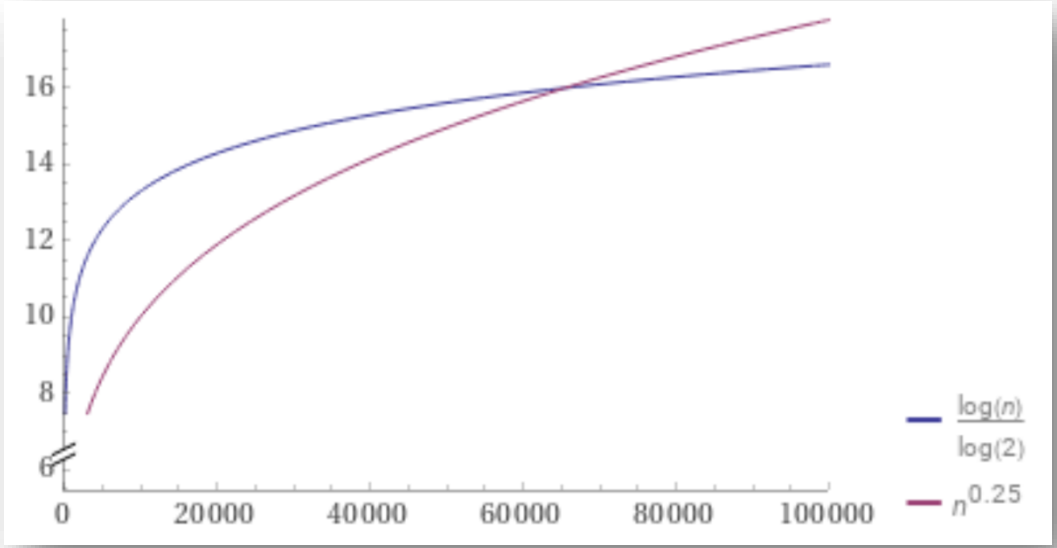
Wie zeigen wir Zugehörigkeit?

Ist $n^{0.25} \in \mathcal{O}(\log_2 n)$?

~~Ja! Abbildung rechts zeigt das:~~
~~Wähle $c = 1, n_0 \geq 5$~~

Auch wenn es für kleine n gut aussieht, kann es für große n anders sein!

Letztendlich muss eine *Allaussage* bewiesen werden:
„Für alle $n \geq n_0$“



Eine bessere Möglichkeit:

Zeige $4n^2 + 12n - 15 \in \mathcal{O}(n^2)$ ⟶ Bestimme n_0 und c_1, sodass für alle $n \geq n_0$ gilt: $0 \leq 4n^2 + 12n - 15 \leq c_1 \cdot n^2$

Suche nach c_1

$4n^2 + 12n - 15 \leq$

Also: $4n^2 + 12n - 15 \leq 16n^2$ für $n \geq 1$.

Die Aussagen oben gelten ab $n_0 = 1$.
Also haben wir mit $c_1 = 16$ und $n_0 = 1$ Werte für die Konstanten gefunden, für die die Definition gilt.

Eine bessere Möglichkeit:

Zeige $4n^2 + 12n - 15 \in \mathcal{O}(n^2)$ ⟶ Bestimme n_0 und c_1, sodass für alle $n \geq n_0$ gilt: $0 \leq 4n^2 + 12n - 15 \leq c_2 \cdot n^2$

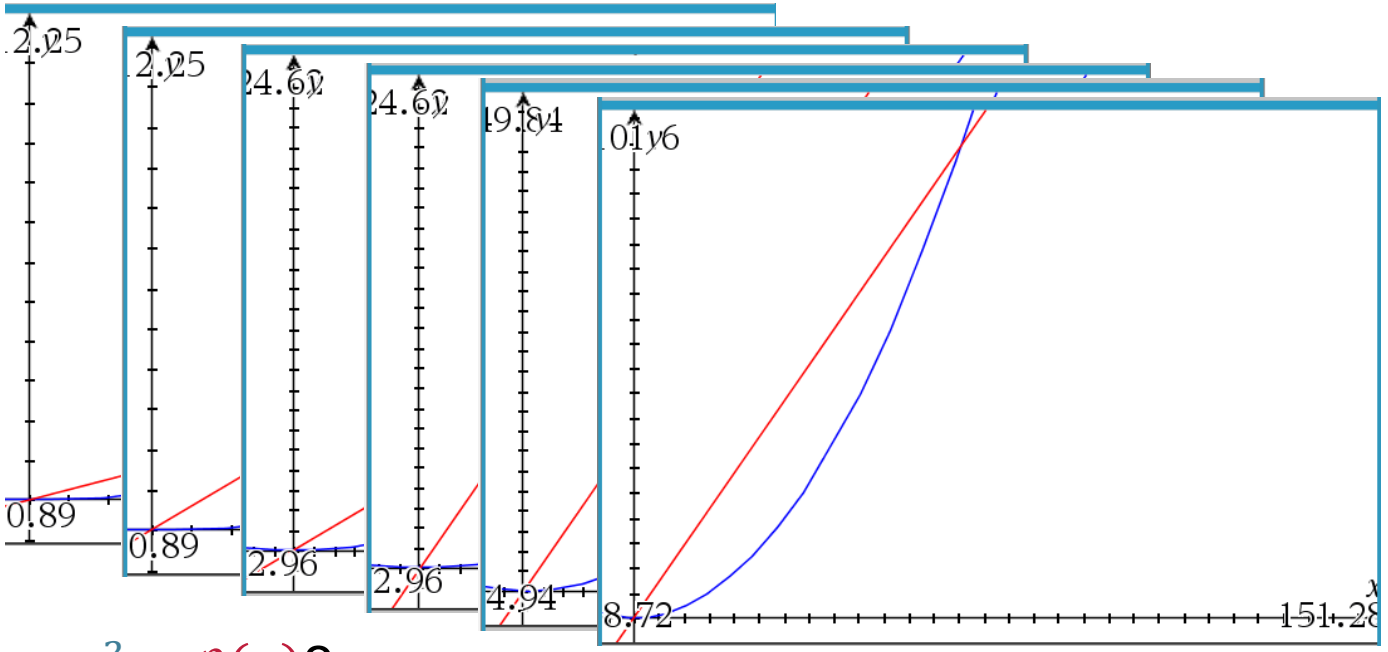
Suche nach c_1

$4n^2 + 12n - 15 \leq 4n^2 + 12n \leq 4n^2 + 12n^2 = 16n^2$ für $n \geq 1$.

Also: $4n^2 + 12n - 15 \leq 16n^2$ für $n \geq 1$.

Die Aussagen oben gelten ab $n_0 = 1$.
Also haben wir mit $c_1 = 16$ und $n_0 = 1$ Werte für die Konstanten gefunden, für die die Definition gilt.
Damit ist $4n^2 + 12n - 15 \in \mathcal{O}(n^2)$.

Kann man sich das mit dem c_1 nicht immer zurechtbiegen?



$n^2 \in \mathcal{O}(n)$?

… wirkt irgendwie nicht so.

(Nur zu Anschauungszwecken. Das ist natürlich, wie eben schon angemerkt, kein Beweis. Aber das verschafft vielleicht ein Gefühl, dass ein konstanter Faktor nicht so mächtig ist, wie er wirkt.)

Ω-Notation

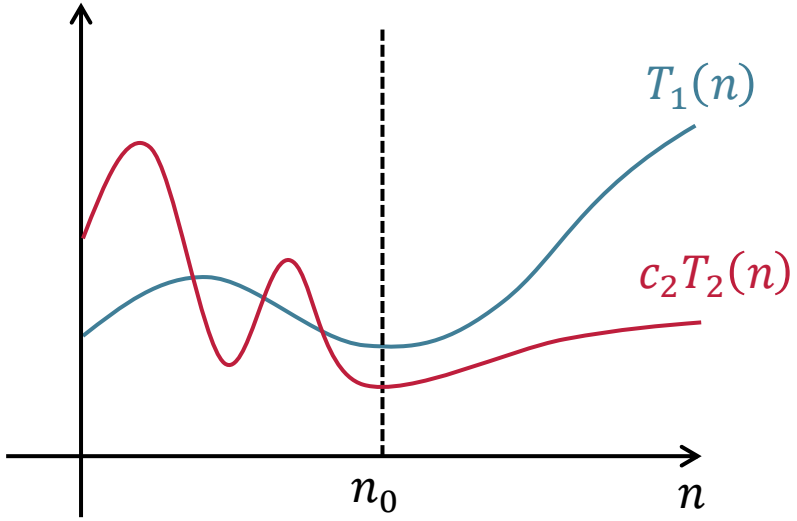
Definition:

Es gibt Konstanten $n_0 \in \mathbb{N}$ und $c_2 \in \mathbb{R}^+$,
sodass für alle $n \geq n_0$ gilt:

$$T_1(n) \geq c_2 T_2(n) \geq 0$$

$$\Leftrightarrow$$

$$T_1(n) \in \Omega\big(T_2(n)\big)$$



„T_1 wächst (asymptotisch) mindestens so schnell wie T_2“

Θ-Notation

Definition:

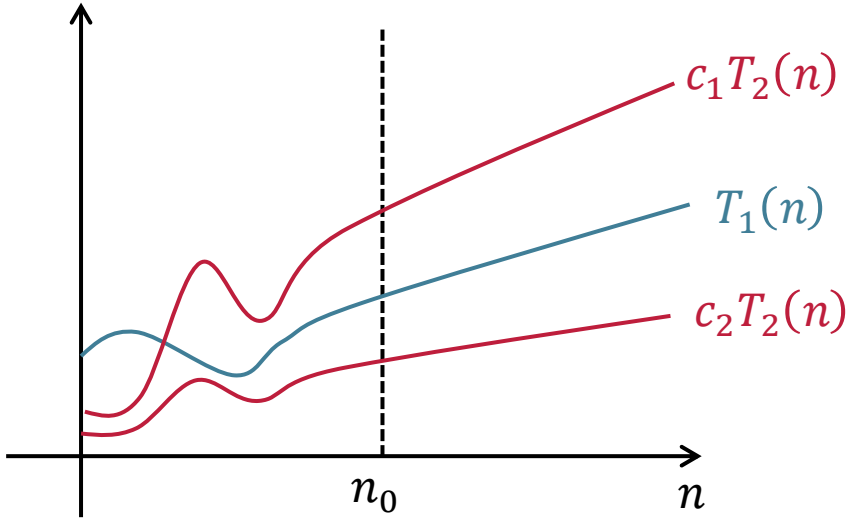
Es gibt Konstanten $n_0 \in \mathbb{N}$ und

$c_1, c_2 \in \mathbb{R}^+$, sodass für alle $n \geq n_0$ gilt:

$$0 \leq c_2 T_2(n) \leq T_1(n) \leq c_1 T_2(n)$$

$$\Leftrightarrow$$

$$T_1(n) \in \Theta(T_2(n))$$



„T_1 wächst (asymptotisch) genau so schnell wie T_2“

Zusammengefasst

Achtung:
$\mathcal{O}$, Ω und Θ sind *Mengen* (von Funktionen)!

Wir haben folgende drei Definitionen gesehen:

$\mathcal{O}$-Notation:

Es gibt Konstanten $n_0 \in \mathbb{N}$ und $c_1 \in \mathbb{R}^+$, sodass für alle $n \geq n_0$ gilt:

$$0 \leq T_1(n) \leq c_1 T_2(n) \Leftrightarrow T_1(n) \in \mathcal{O}\big(T_2(n)\big)$$

Ω-Notation:

Es gibt Konstanten $n_0 \in \mathbb{N}$ und $c_2 \in \mathbb{R}^+$, sodass für alle $n \geq n_0$ gilt:

$$T_1(n) \geq c_2 T_2(n) \geq 0 \Leftrightarrow T_1(n) \in \Omega\big(T_2(n)\big)$$

Θ-Notation:

Es gibt Konstanten $n_0 \in \mathbb{N}$ und $c_1, c_2 \in \mathbb{R}^+$, sodass für alle $n \geq n_0$ gilt:

$$0 \leq c_2 T_2(n) \leq T_1(n) \leq c_1 T_2(n) \Leftrightarrow T_1(n) \in \Theta\big(T_2(n)\big)$$

→ „Landau-Symbole“

Rechenregeln

Ganz grundsätzlich gelten diese Beobachtungen:

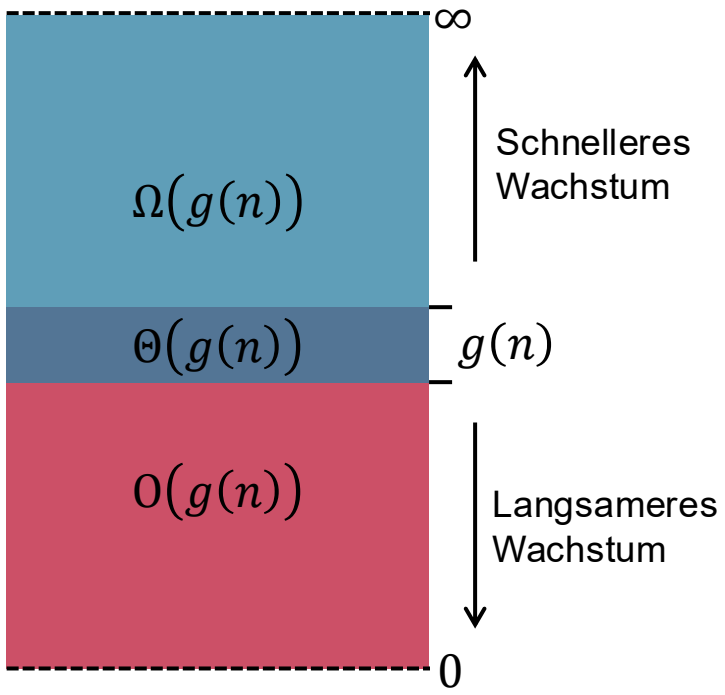
$$f \in \mathcal{O}(g) \Leftrightarrow c \cdot f \in \mathcal{O}(g) \text{ für } c \in \mathbb{R}^+$$

$$f \in \mathcal{O}(g) \Leftrightarrow g \in \Omega(f)$$

$$f \in \Theta(g) \Leftrightarrow f \in \mathcal{O}(g) \wedge f \in \Omega(g)$$

$$f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$$

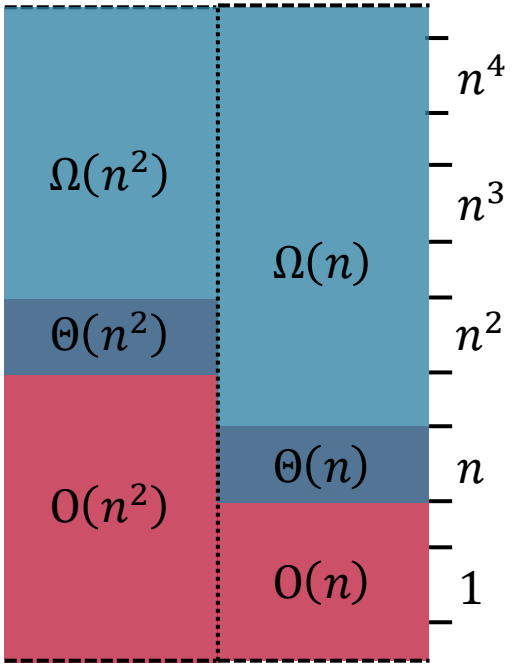
Relationen zwischen Klassen - Eine grafische Darstellung



Wir können sogar Klassen miteinander vergleichen.
Beispiel:

$$\Theta(n^2) \subsetneq \Omega(n)$$

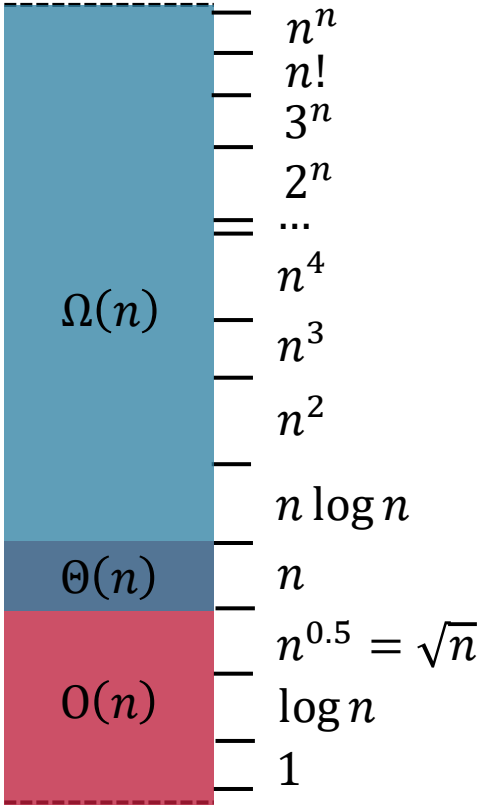
Zum Merken:
Wächst die Funktion schneller, wächst der $\mathcal{O}$-Bereich und der Ω-Bereich schrumpft.



Relationen zwischen Klassen – Eine Tabelle

Bedingung	**Klasse** / **Klasse**	$O(g(n))$	$\Theta(g(n))$	$\Omega(g(n))$
$f(n) \in o(g(n))$ $(o(g(n)) := O(g(n)) \setminus \Theta(g(n)))$ "Klein-o-Notation"	$O(f(n))$	$\subsetneq$	X	X
	$\Theta(f(n))$	$\subsetneq$	X	X
	$\Omega(f(n))$	X	$\supsetneq$	$\supsetneq$
$f(n) \in \Theta(g(n))$	$O(f(n))$	=	$\supsetneq$	X
	$\Theta(f(n))$	$\subsetneq$	=	$\subsetneq$
	$\Omega(f(n))$	X	$\supsetneq$	=
$f(n) \in \omega(g(n))$ $(\omega(g(n)) := \Omega(g(n)) \setminus \Theta(g(n)))$ "Klein-ω-Notation"	$O(f(n))$	$\supsetneq$	$\supsetneq$	X
	$\Theta(f(n))$	X	X	$\subsetneq$
	$\Omega(f(n))$	X	X	$\subsetneq$

Relationen zwischen Klassen – Die wichtigsten Klassen



Beispiele

Beispiel 1

Zeige $4n^2 + 12n - 15 \in \Theta(n^2)$ ⟶ Bestimme n_0, c_1, c_2, sodass für alle $n \geq n_0$ gilt: $0 \leq c_2 \cdot n^2 \leq 4n^2 + 12n - 15 \leq c_1 \cdot n^2$

Suche nach c_1

$4n^2 + 12n - 15 \leq 4n^2 + 12n \leq 4n^2 + 12n^2 = 16n^2$ für $n \geq 1$.

Suche nach c_2

$4n^2 + 12n - 15$

Beispiel 1

Zeige $4n^2 + 12n - 15 \in \Theta(n^2)$ ⟶ Bestimme n_0, c_1, c_2, sodass für alle $n \geq n_0$ gilt: $0 \leq c_2 \cdot n^2 \leq 4n^2 + 12n - 15 \leq c_1 \cdot n^2$

Suche nach c_1

$4n^2 + 12n - 15 \leq 4n^2 + 12n \leq 4n^2 + 12n^2 = 16n^2$ für $n \geq 1$.

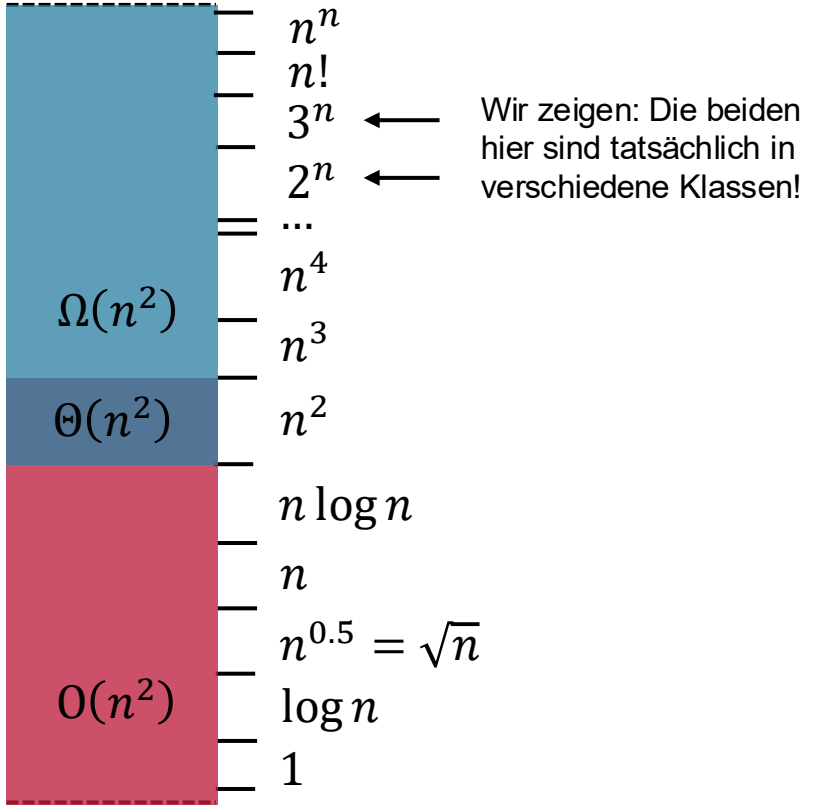
Suche nach c_2

$4n^2 + 12n - 15 \geq 4n^2 - 15 \overset{n \geq 4}{\geq} 4n^2 - n^2 = 3n^2$ für $n \geq 4$.

Beide Ungleichungen gelten ab $n_0 = 4$. Also $c_2 = 3,\ c_1 = 16$ und $n_0 = 4$.

Beispiel 2

Zeige oder widerlege: $2^n \in \Theta(3^n)$



Beispiel 2

Zeige oder widerlege: $2^n \in \Theta(3^n)$

Beispiel 2

Zeige oder widerlege: $2^n \in \Theta(3^n)$

Zunächst: $2^n \in \mathcal{O}(3^n)$, denn $2^n \leq (2+1)^n = 3^n$.

Aber: $2^n \notin \Omega(3^n)$!

Ansonsten gäbe es eine Konstante c_1 mit

$$2^n \geq c_1 \cdot 3^n, \text{ also } \frac{2^n}{3^n} \geq c_1$$

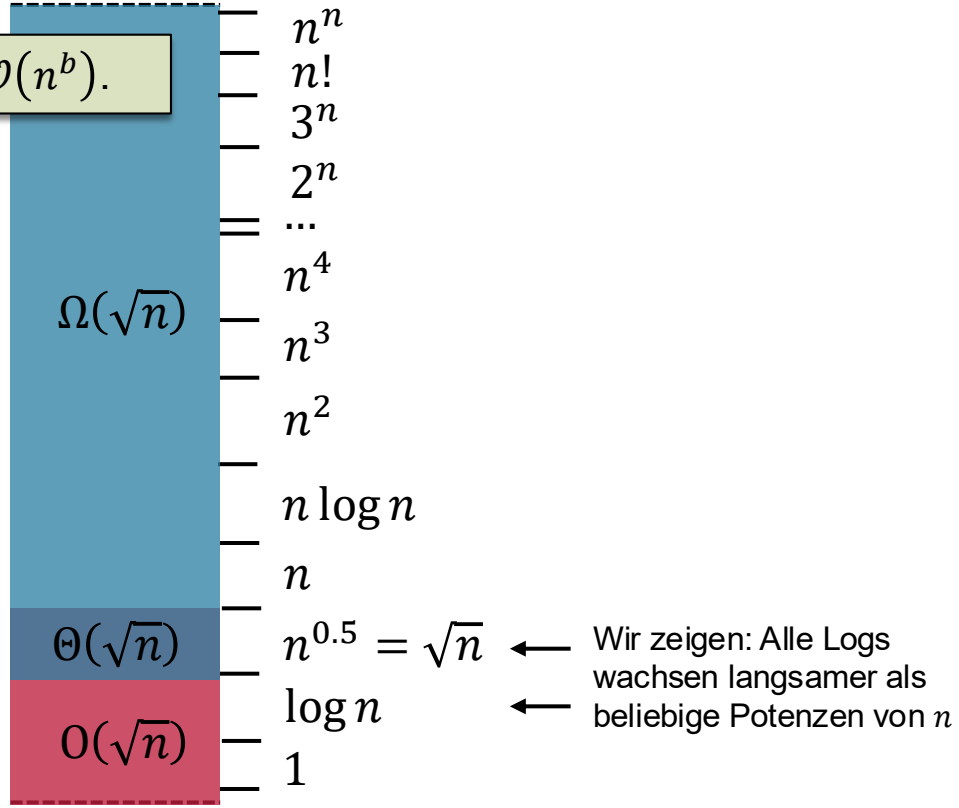
Aber: $\frac{2^n}{3^n} = \left(\frac{2}{3}\right)^n$ und $\lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0$, d.h. dieses c_1 kann nicht existieren!

⇒ Aussage widerlegt.

Beispiel 3

Satz (2): Seien $a, b \in \mathbb{R}^+$. Dann gilt $\log_2^a n \in \mathcal{O}(n^b)$.

$$\log_2^a n = (\log_2 n)^a$$



Beispiel 3

$\log_2 n \in \mathcal{O}(n)$

Satz (1): Für jedes $c > 0$ gibt es ein n_0, sodass für alle $n \geq n_0$ gilt: $\log_2 n \leq c \cdot n$

Beweisidee: Zeige, dass $\lim_{n \to \infty} \frac{\log_2 n}{n} = 0$, d.h. für wachsendes n kommen wir beliebig nah an 0 heran. D.h. wir können n_0 so wählen, dass $\frac{\log_2 n}{n} \leq c$ für alle $n \geq n_0$ gilt.

Beispiel 3

Satz (1): Für jedes $c > 0$ gibt es ein n_0, sodass für alle $n \geq n_0$ gilt: $\log_2 n \leq c \cdot n$

Satz (2): Seien $a, b \in \mathbb{R}^+$. Dann gilt $\log_2^a n \in \mathcal{O}(n^b)$.

$$\log_2^a n = (\log_2 n)^a$$

Beweis: Wir setzen $c = 1$. Wir zeigen, dass ab einem n_0 gilt:

$$\begin{aligned} & \log_2^a n \leq n^b \\ \Leftrightarrow \quad & \log_2 \log_2^a n \leq \log_2 n^b \\ \Leftrightarrow \quad & a \cdot \log_2 \log_2 n \leq b \cdot \log_2 n \\ \overset{m := \log_2 n}{\Leftrightarrow} \quad & \log_2 m \leq \frac{b}{a} m \end{aligned}$$

Da $a, b \in \mathbb{R}^+$ ist auch $\frac{b}{a} \in \mathbb{R}^+$. Nach Satz (1) oben ist die letzte Ungleichung ab einem n_0 wahr.

Beispiel 3

Satz (1): Für jedes $c > 0$ gibt es ein n_0, sodass für alle $n \geq n_0$ gilt: $\log_2 n \leq c \cdot n$

Satz (2): Seien $a, b \in \mathbb{R}^+$. Dann gilt $\log_2^a n \in \mathcal{O}(n^b)$.

$$\log_2^a n = (\log_2 n)^a$$

Beweis: Wir setzen $c = 1$. Wir zeigen, dass ab einem n_0 gilt:

$$\begin{aligned} & \log_2^a n \leq n^b \\ \Leftrightarrow \quad & \log_2 \log_2^a n \leq \log_2 n^b \\ \Leftrightarrow \quad & a \cdot \log_2 \log_2 n \leq b \cdot \log_2 n \\ \overset{m := \log_2 n}{\Leftrightarrow} \quad & \log_2 m \leq \frac{b}{a} m \end{aligned}$$

Da $a, b \in \mathbb{R}^+$ ist auch $\frac{b}{a} \in \mathbb{R}^+$. Nach Satz (1) oben ist die letzte Ungleichung ab einem n_0 wahr. Da wir Äquivalenzumformungen benutzt haben, können wir die Lösungskette von unten nach oben gehen.

Codierungsgrößen

Codierungsgröße

Wie viel Speicher brauchen wir, um eine ganze Zahl bis zu einer bestimmten Größe darzustellen?

Dezimal
27

b-adische Notation

- Dezimal (10-adisch)
- Binär (2-adisch)
- Oktal (8-adisch)
- Hexadezimal (16-adisch)

Codierungsgröße - Zahlen

Wie viel Speicher brauchen wir, um eine ganze Zahl bis zu einer bestimmten Größe darzustellen?

Dezimal	Binär	Oktal	Hexadezimal
27	11011	33	1B

b-adische Notation

- Dezimal (10-adisch)
- Binär (2-adisch)
- Oktal (8-adisch)
- Hexadezimal (16-adisch)
- Allgemein: $a_m a_{m-1} \dots a_1 a_0, a_{-1} a_{-2} \dots a_{-\infty}$ wobei $n = \sum_{i=-\infty}^{m} a_i b^i$ und $0 \leq a < b$

Codierungsgröße - Zahlen

Wie viel Speicher brauchen wir, um eine ganze Zahl bis zu einer bestimmten Größe darzustellen?

Dezimal	Binär	Oktal	Hexadezimal
27	11011		

TL;DR: Codierungsgrößen
Ganze Zahlen belegen *logarithmisch* viel Speicher
(zur maximalen Zahlengröße)

b-adische I

- Dezim
- Binär (
- Oktal (
- Hexadezimal (16-adisch)
- Allgemein: $a_m a_{m-1} \dots a_1 a_0, a_{-1} a_{-2} \dots a_{-\infty}$ wobei $n = \sum_{i=-\infty}^{m} a_i b^i$ und $0 \leq a < b$

Codierungsgröße - Beispiele

- Zeichenketten/Strings S.

Beispiel ASCII-Zeichen

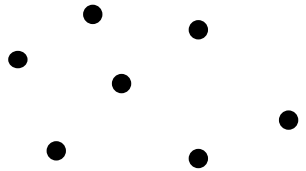
7 bits pro Symbol → 128 mögliche Zeichen

Code	…0	…1	…2	…3	…4	…5	…6	…7	…8	…9	…A	…B	…C	…D	…E	…F
0…	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1…	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2…	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3…	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4…	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5…	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6…	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7…	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Codierungsgröße - Beispiele

- Zeichenketten/Strings S.
  $|S|$ Anzahl Zeichen, N mögliche Zeichen.
  $$\approx |S| \cdot \log N$$

- Punktmenge P in d Dimensionen
  mit N als die größtmögliche ganzzahliger Koordinate:
  $\approx d \cdot |P| \cdot \log N,$



- $n \times m$-Matrizen von ganzen Zahlen
  mit dem größtmöglichen Wert N:
  $\approx n \cdot m \cdot \log(N),$

$$\begin{pmatrix} 5 & 12 & 1 \\ 6 & 22 & 5 \\ 0 & 42 & 21 \end{pmatrix}$$

Codierungsgröße - Graphen

Wie kann man Graphen speichern?

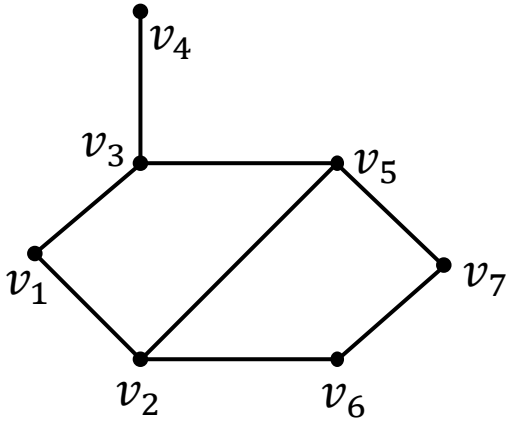
Adjazenzmatrix

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

$|V|^2$ bits

Adjazenzliste

$v_1: v_2, v_3$
$v_2: v_1, v_5, v_6$
$v_3: v_1, v_4, v_5$
$v_4: v_3$
$v_5: v_2, v_3, v_7$
$v_6: v_2, v_7$
$v_7: v_5, v_6$



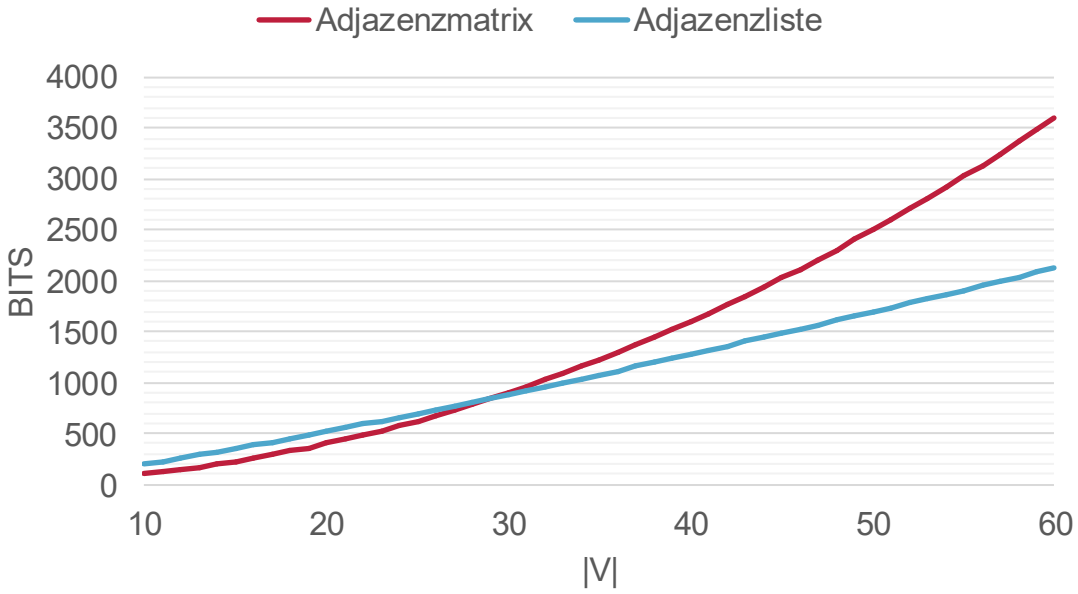
$\approx (|V| + 2|E|) \cdot \log|V|$ bits

Codierungsgröße - Graphen

Adjazenzmatrix: $|V|^2$ bits

Adjazenzliste: $(|V| + 2|E|) \cdot \log|V|$ bits

Annahme: Der Graph besitzt 3-Mal so viele Kanten wie Knoten.

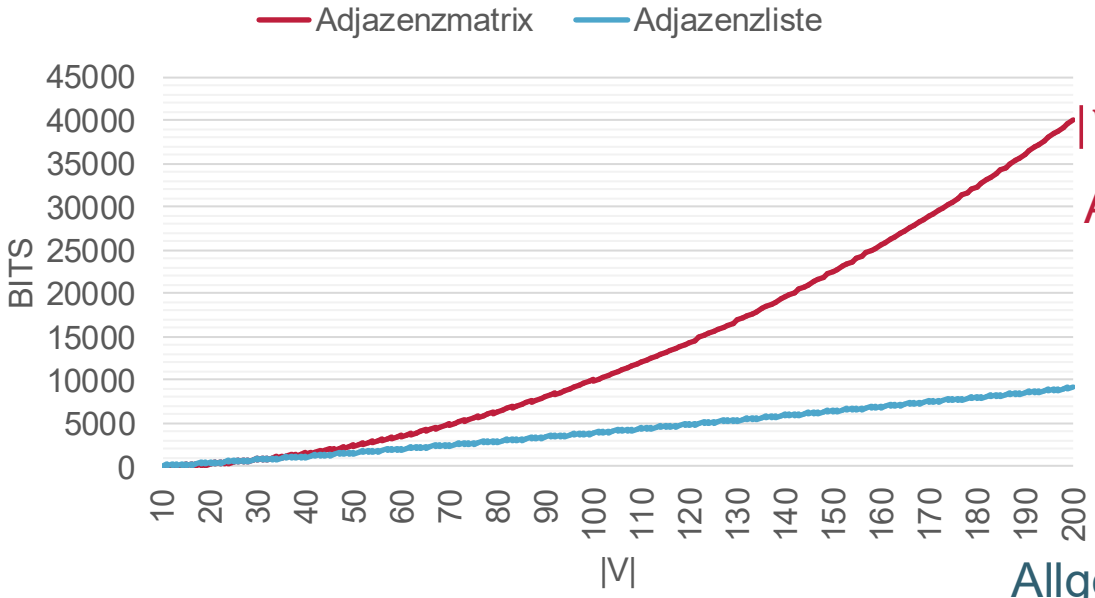


Codierungsgröße - Graphen

Adjazenzmatrix: $|V|^2$ bits

Adjazenzliste: $(|V| + 2|E|) \cdot \log|V|$ bits

Annahme: Der Graph besitzt 3-Mal so viele Kanten wie Knoten.



$|V|^2$ bits $\Theta(|V|^2)$

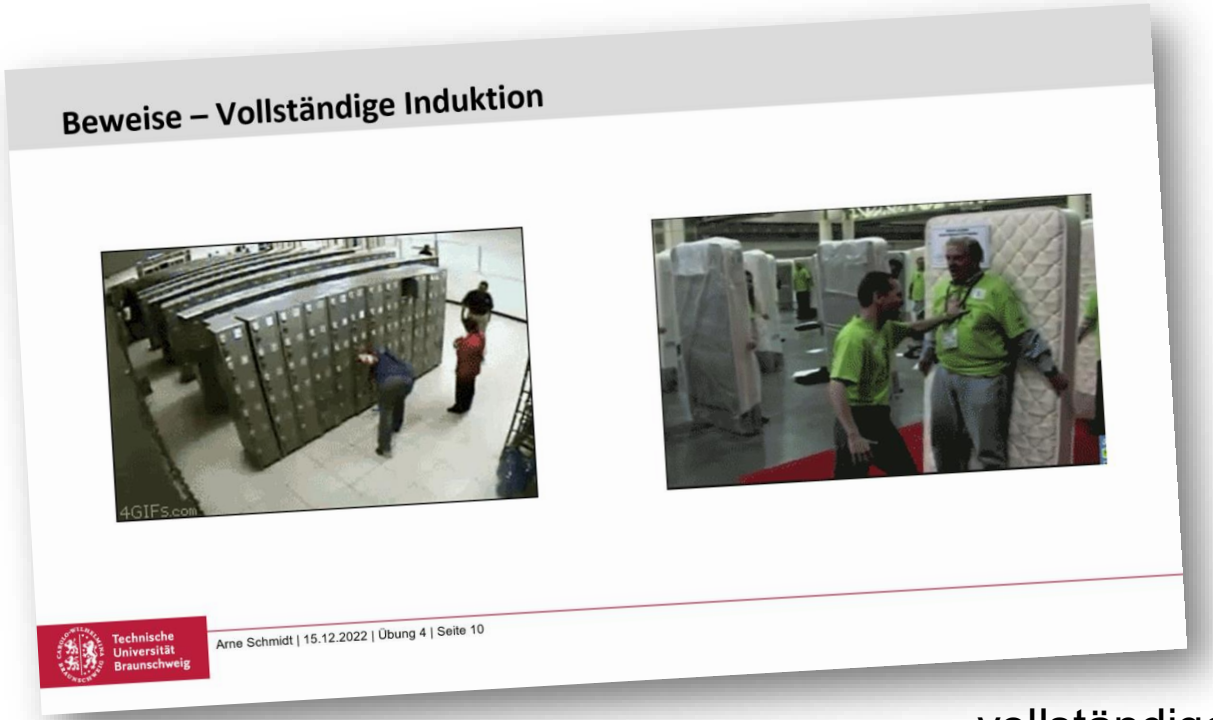
Allgemein : $\Theta(n^2)$

$7|V| \cdot \log_2|V|$ bits

$\Theta(|V| \log|V|)$

Allgemein: $\Theta(n \log m)$

… nächstes Mal:



… vollständige Induktion!