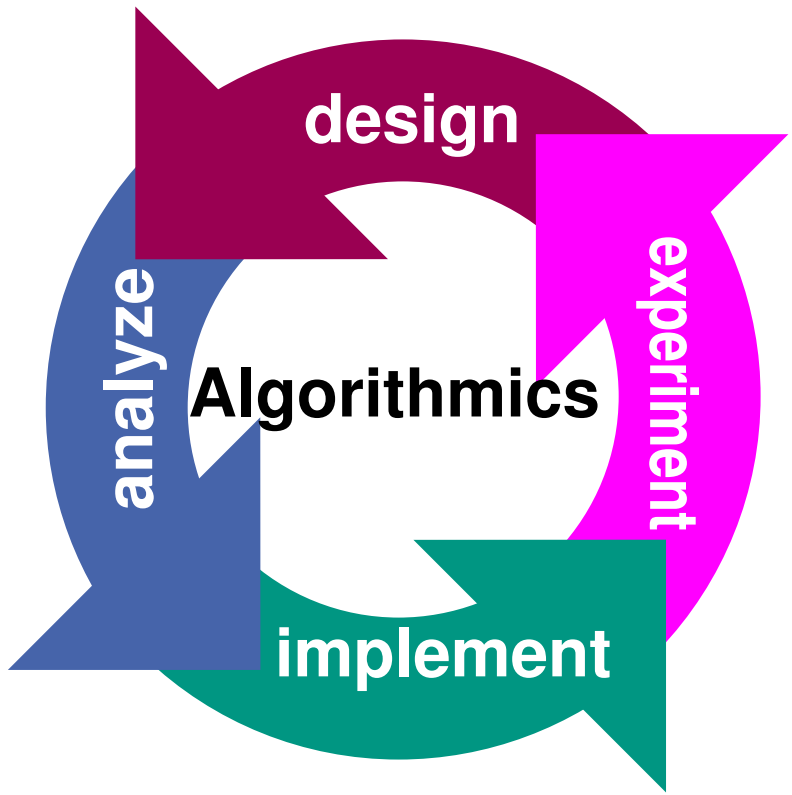
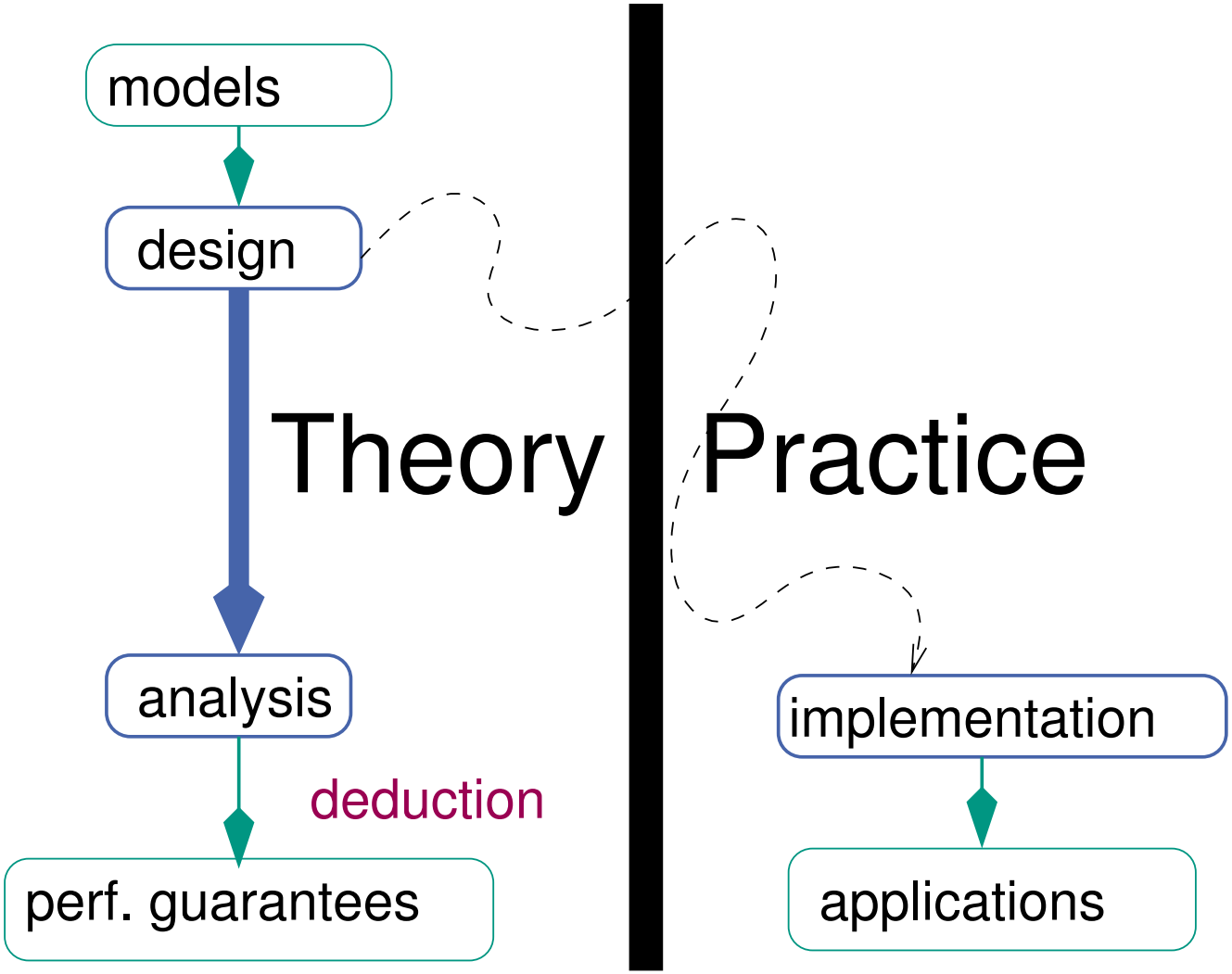
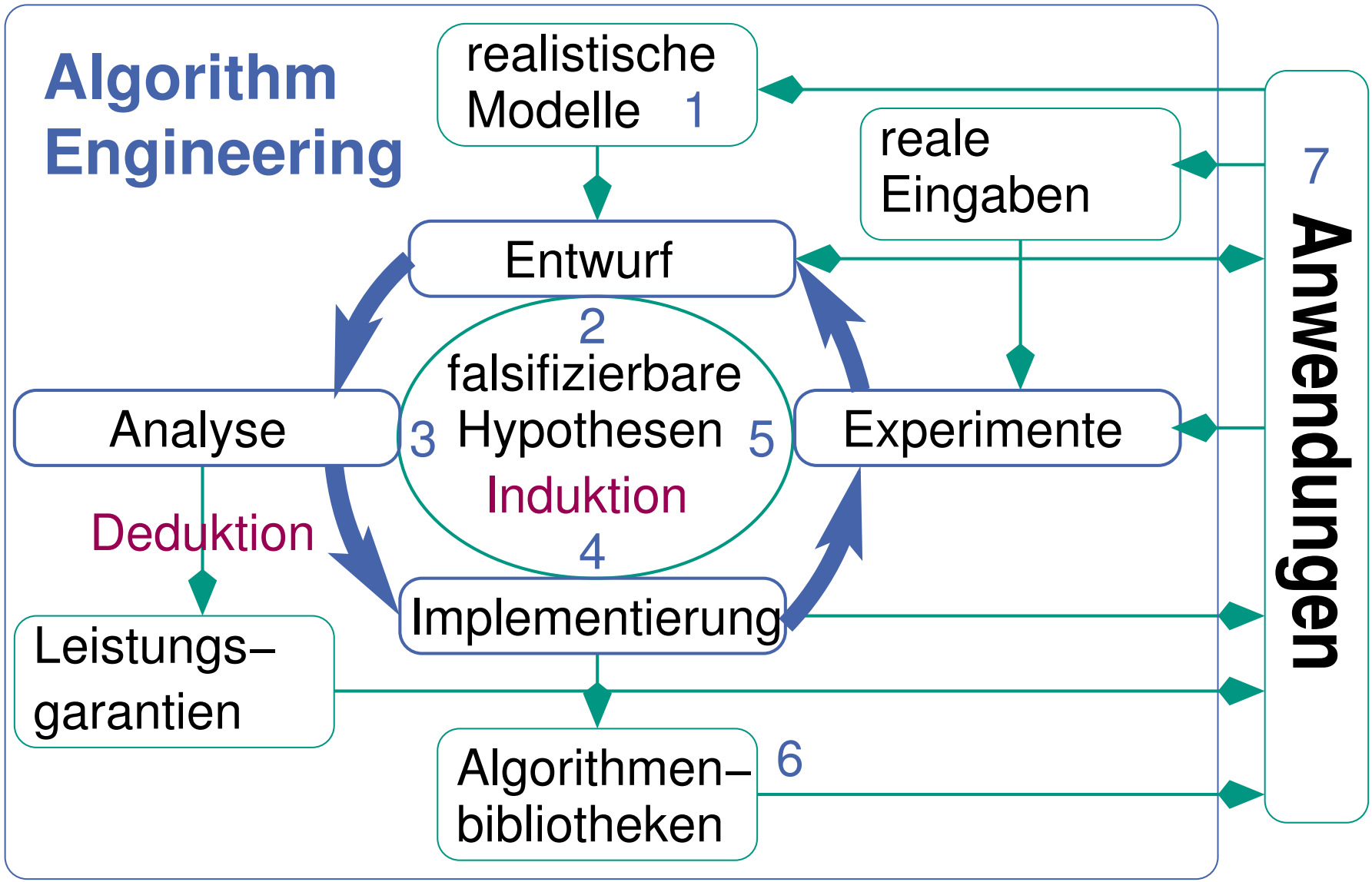
Algorithm Engineering – was hat das mit der Praxis zu tun?′



Algorithmentheorie (Karikatur)

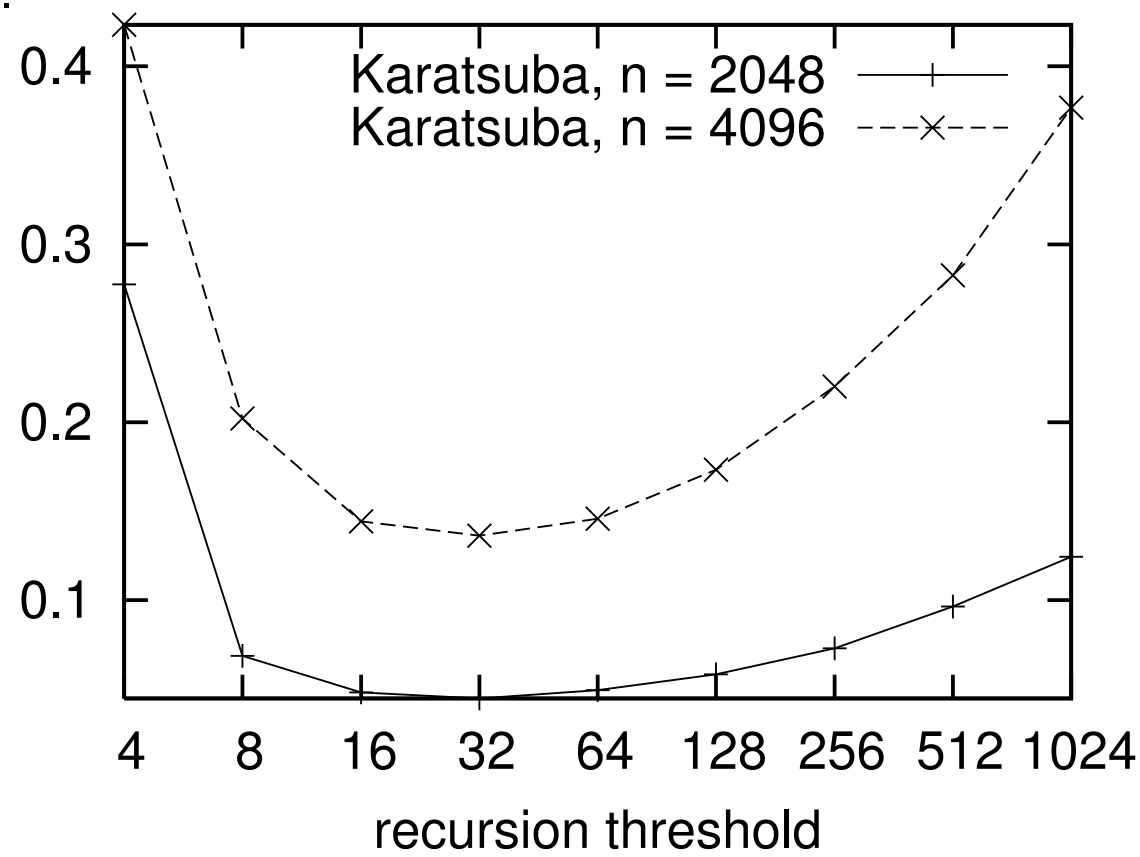


Algorithmik als Algorithm Engineering



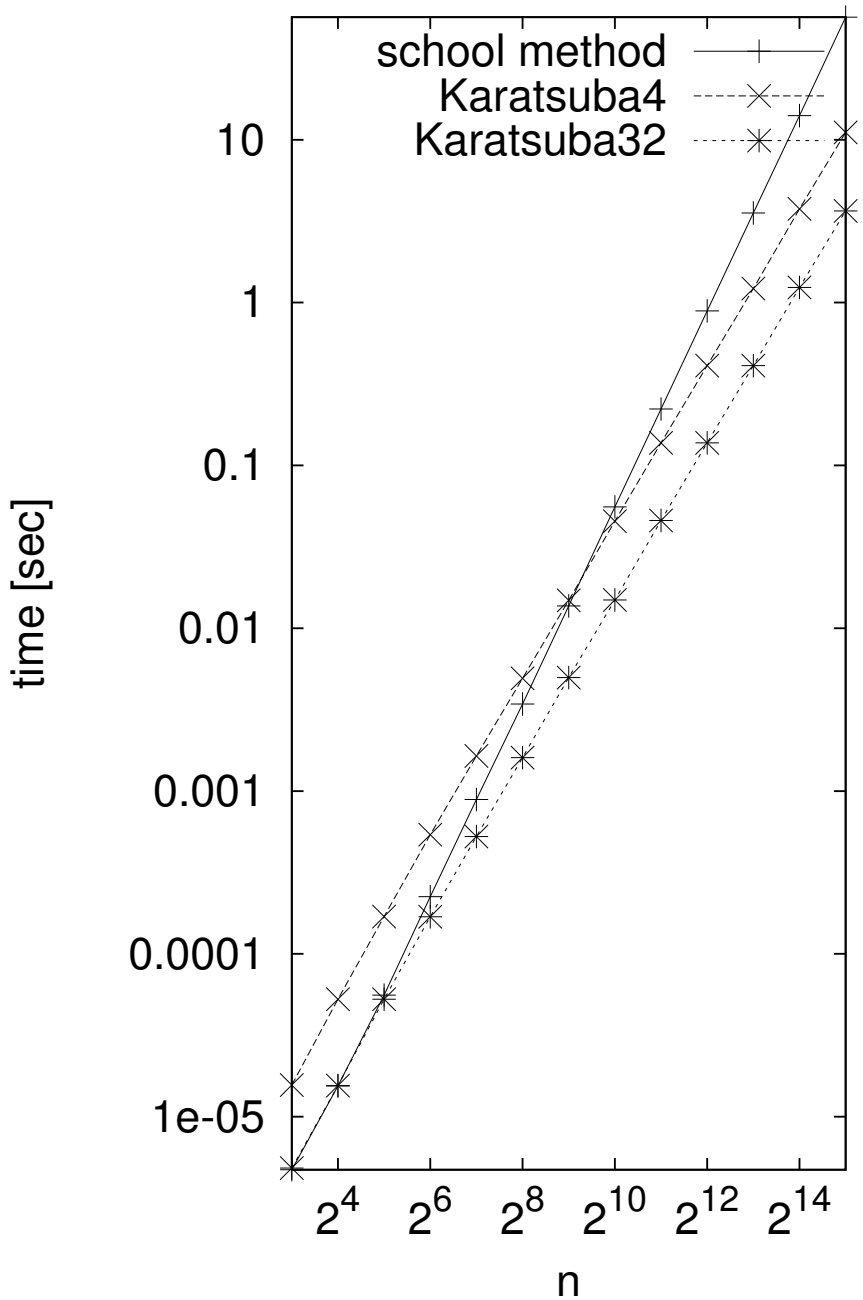
Zurück zur Langzahlmultiplikation

- Zifferngröße $\leftrightarrow$ Hardware-Fähigkeiten
  z. B. 32 Bit
- Schulmultiplikation für kleine Eingaben
- Assembler, SIMD,...



Skalierung

- Asymptotik
  setzt sich durch
- Konstante Faktoren oft
  Implementierungsdetail

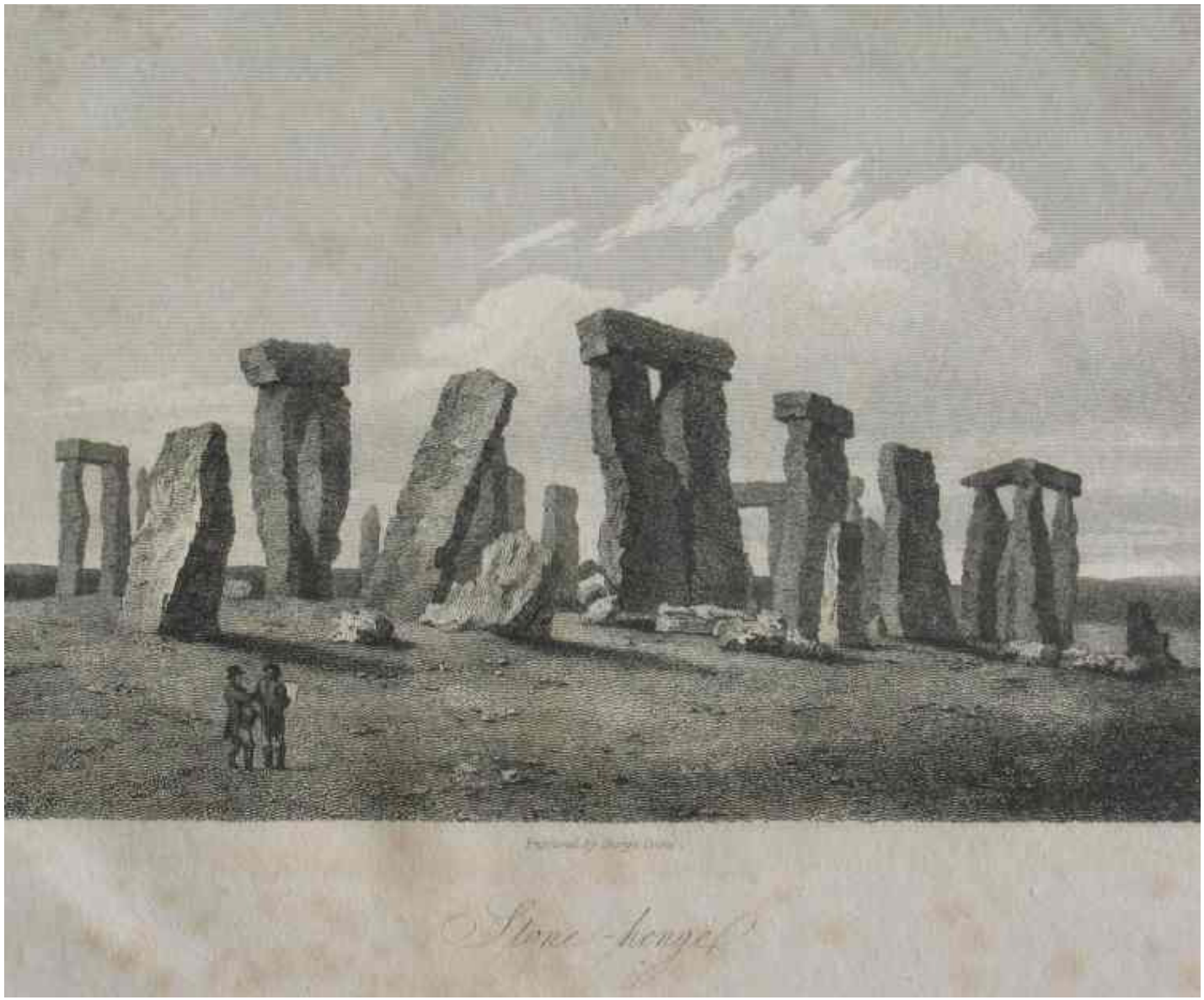


Blick über den Tellerrand

- Bessere Potenzen durch Aufspalten in mehr Teile
- Schnelle Fourier Transformation
  $\rightsquigarrow$ $O(n)$ Multiplikationen von $O(\log n)$-Bit Zahlen
- [Schönhage-Strassen 1971]: Bitkomplexität $O(n \log n \log\log n)$
- [Fürer 2007]: Bitkomplexität $2^{O(\log^* n)} n \log n$
- Praxis: Karatsuba-Multiplikation ist nützlich für Zahlenlängen aus der Kryptographie
- GnuPG, OpenSSL verwenden Karatsuba (ab best. Bitlänge)

Iterierter Logarithmus: $\log^* n = \begin{cases} 0 & \text{falls } n \leq 1 \\ 1 + \log^* \log n & \text{sonst} \end{cases}$

Einführendes



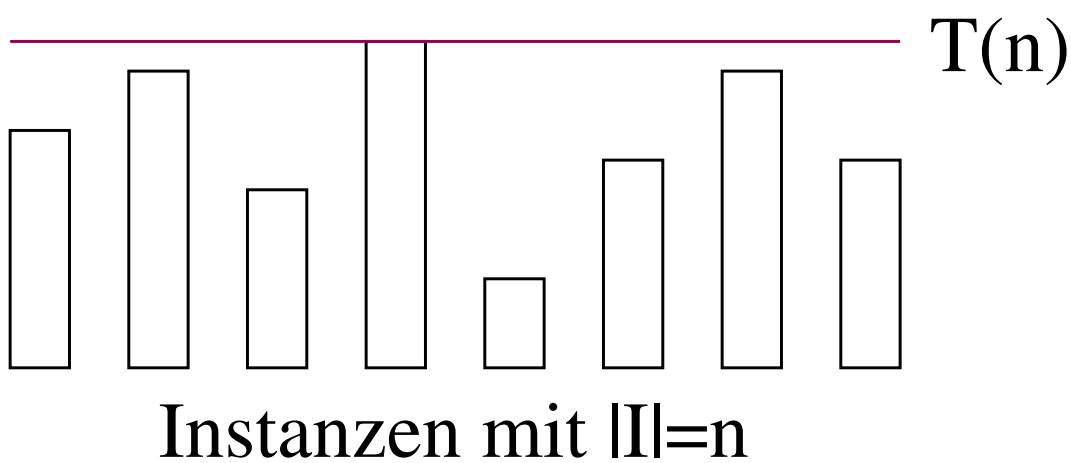
Überblick

- Algorithmenanalyse
- Maschinenmodell
- Pseudocode
- Codeannotationen
- Mehr Algorithmenanalyse
- Graphen

(Asymptotische) Algorithmenanalyse

Gegeben:

Ein Programm
Gesucht: Laufzeit $T(I)$ (# Takte), eigentlich für alle Eingaben I (!)
(oder auch Speicherverbrauch, Energieverbrauch,...)
Erste Vereinfachung: Worst case: $T(n) = \max_{|I|=n} T(I)$
(Später mehr:
average case, best case, die Rolle des Zufalls, mehr Parameter)



Zweite Vereinfachung: Asymptotik

$$\mathrm{O}(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

„höchstens“

$$\Omega(f(n)) = \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

„mindestens“

$$\Theta(f(n)) = \mathrm{O}(f(n)) \cap \Omega(f(n))$$

„genau“

$$o(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

„weniger“

$$\omega(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

„mehr“

O-Kalkül Rechenregeln

Schludrigkeit: implizite Mengenklammern.
Lese '$f(n) = E$' als '$\{f(n)\} \subseteq E$'

$$cf(n) \in \Theta(f(n)) \text{ für jede positive Konstante } c$$

$$\sum_{i=0}^{k} a_i n^i \in O(n^k)$$

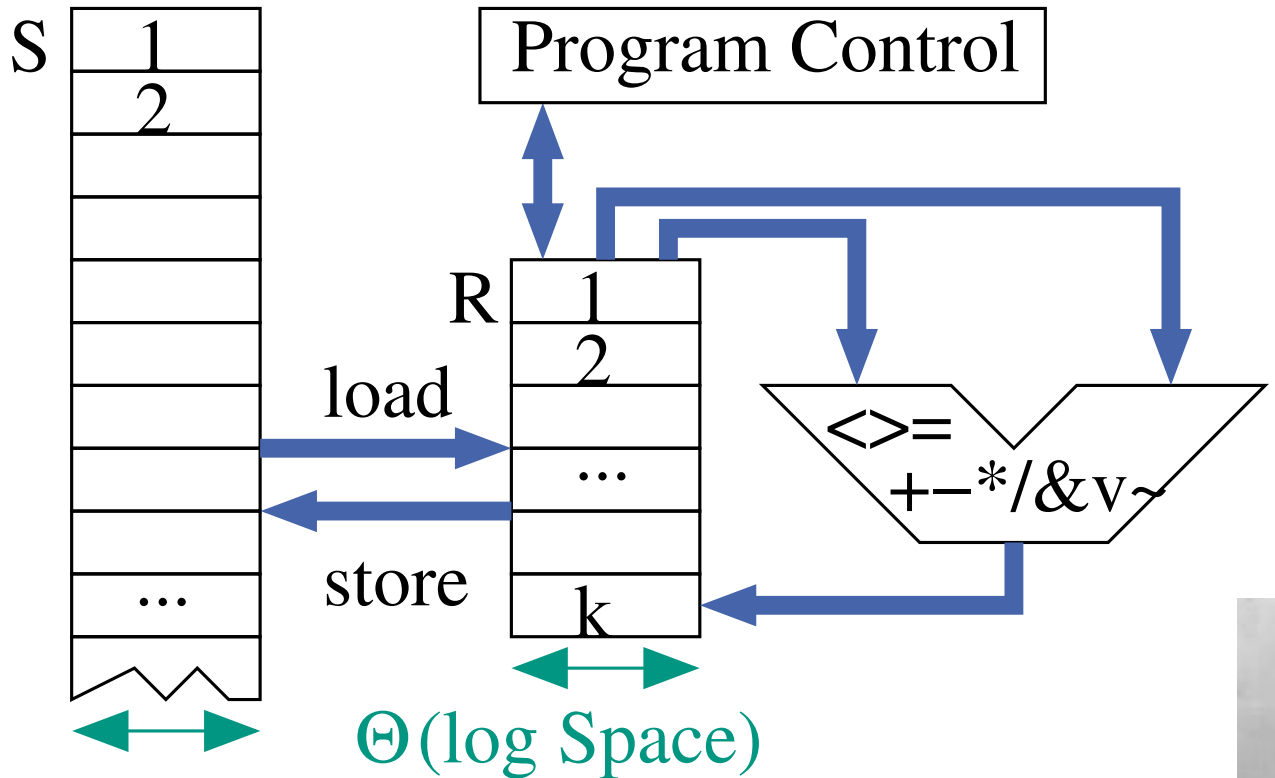
$$f(n) + g(n) \in \Omega(f(n)),$$

$$f(n) + g(n) \in \mathrm{O}(f(n)) \text{ falls } g(n) = \mathrm{O}(f(n)),$$

$$\mathrm{O}(f(n)) \cdot \mathrm{O}(g(n)) = \mathrm{O}(f(n) \cdot g(n)).$$

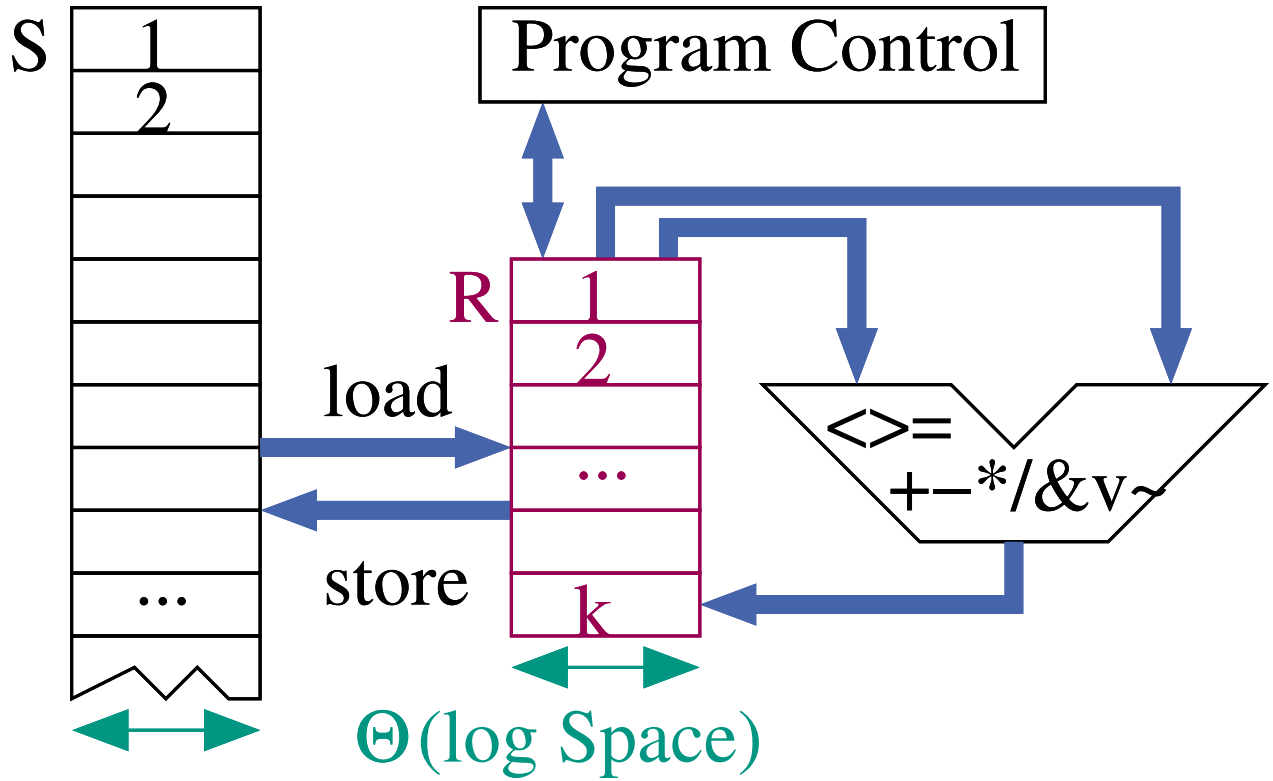
u. s. w.

Maschinenmodell:
RAM (Random Access Machine)



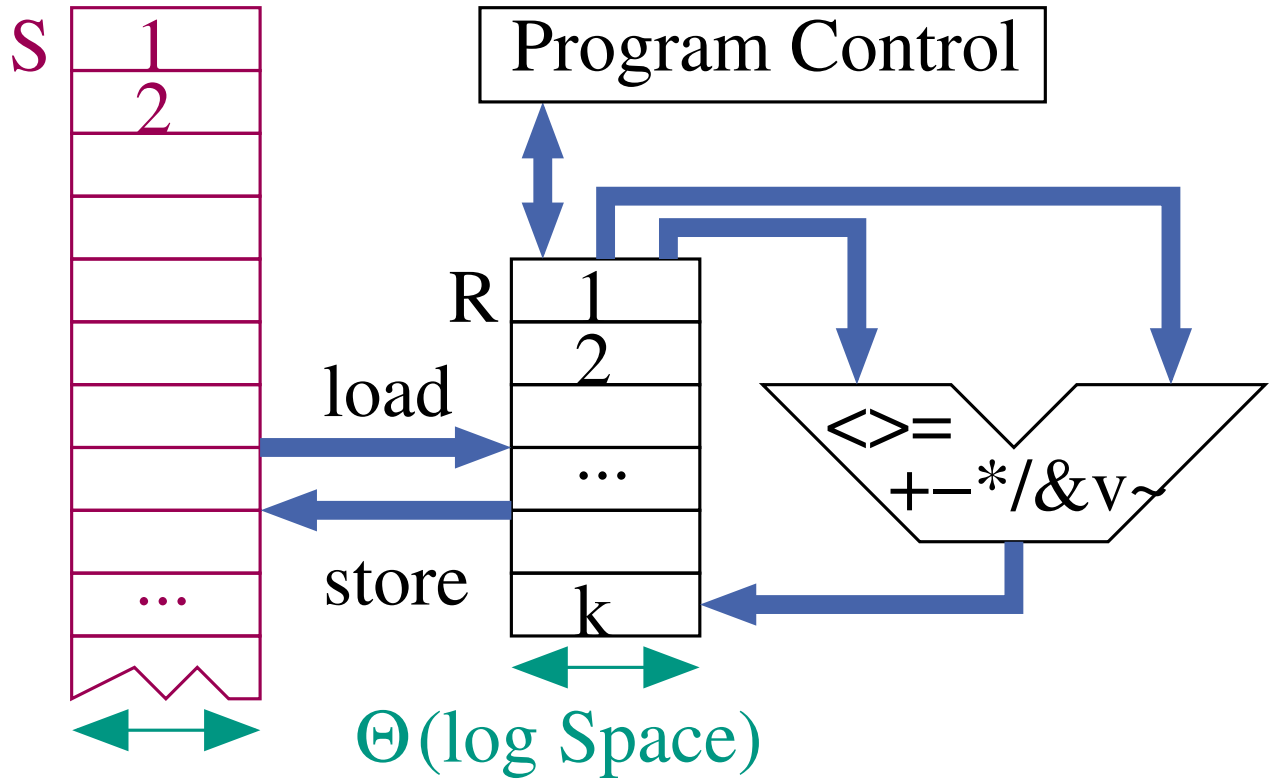
Moderne (RISC) Adaption des
von Neumann-Modells [von Neumann 1945]

Register



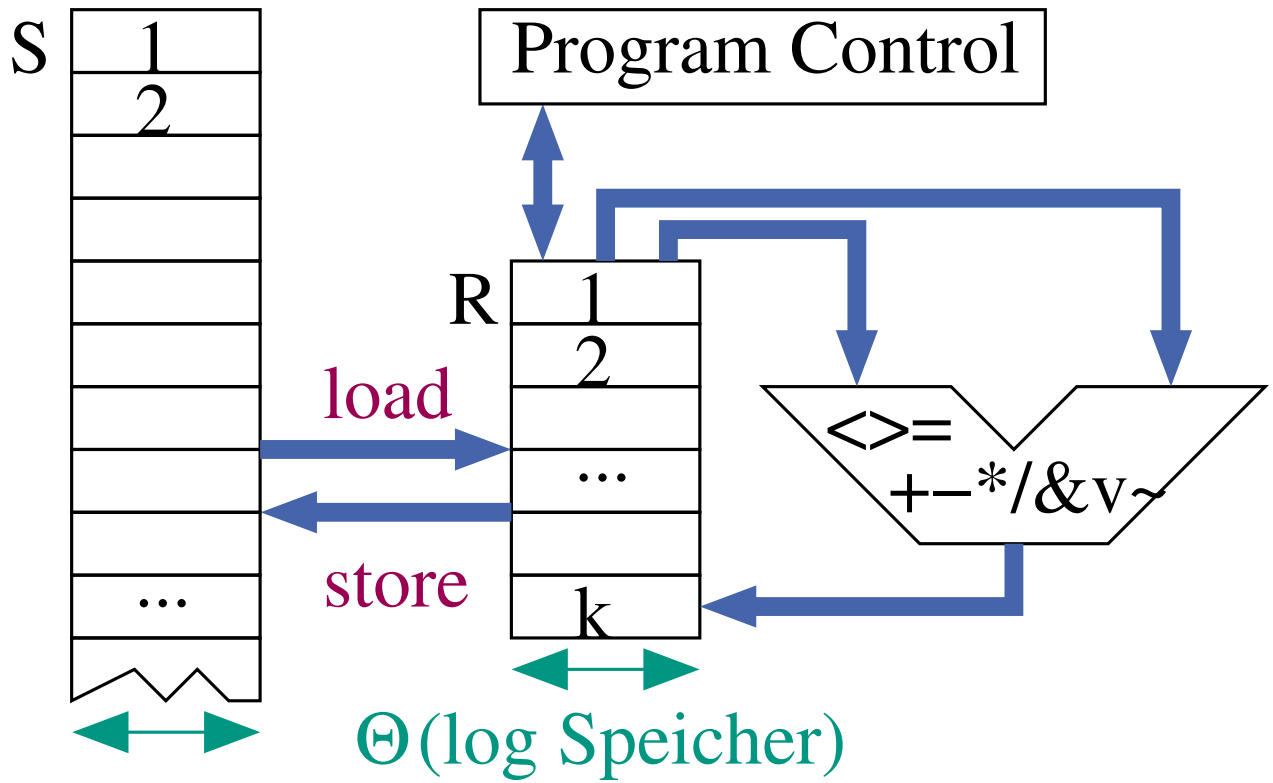
k (irgendeine Konstante) Speicher
$R_1, \ldots, R_k$ für
(kleine) ganze Zahlen

Hauptspeicher



Unbegrenzter Vorrat an Speicherzellen
$S[1], S[2] \ldots$ für
(kleine) ganze Zahlen

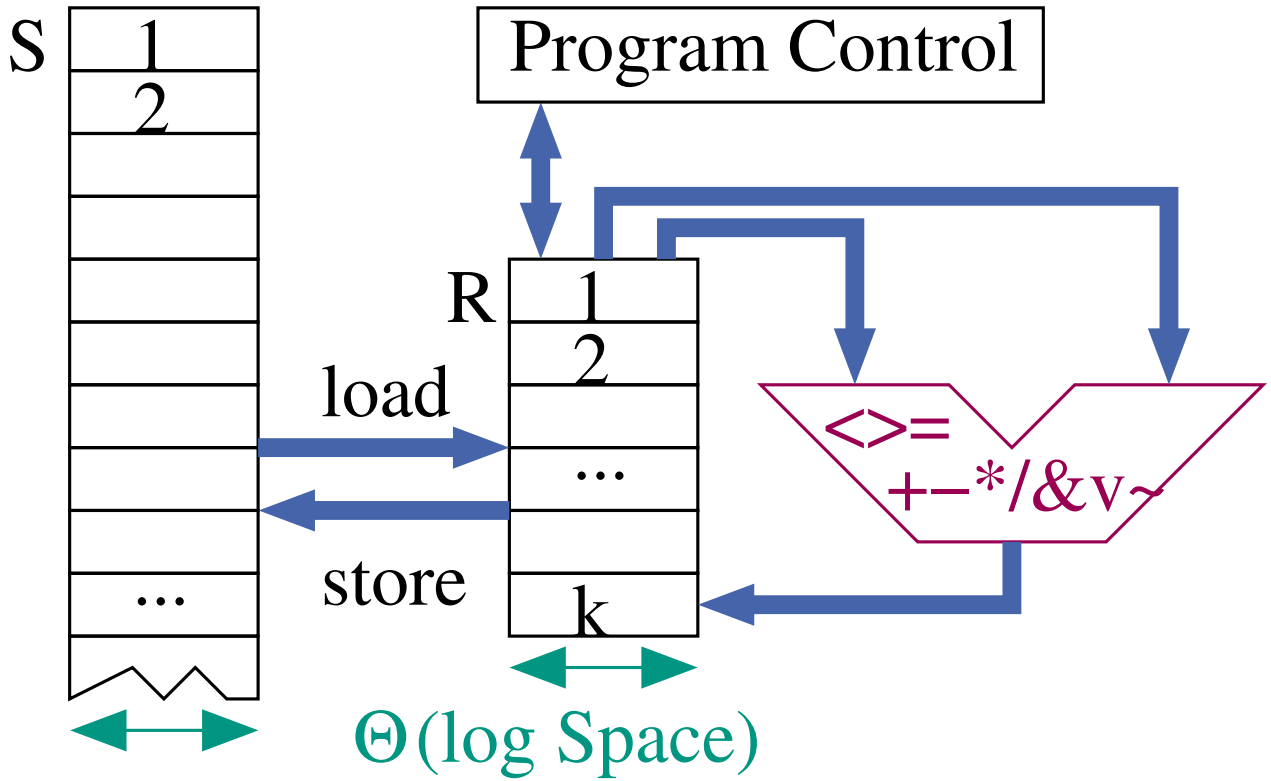
Speicherzugriff



$R_i := S[R_j]$ lädt Inhalt von Speicherzelle $S[R_j]$ in Register R_i.

$S[R_j] := R_i$ speichert Register R_i in Speicherzelle $S[R_j]$.

Rechnen

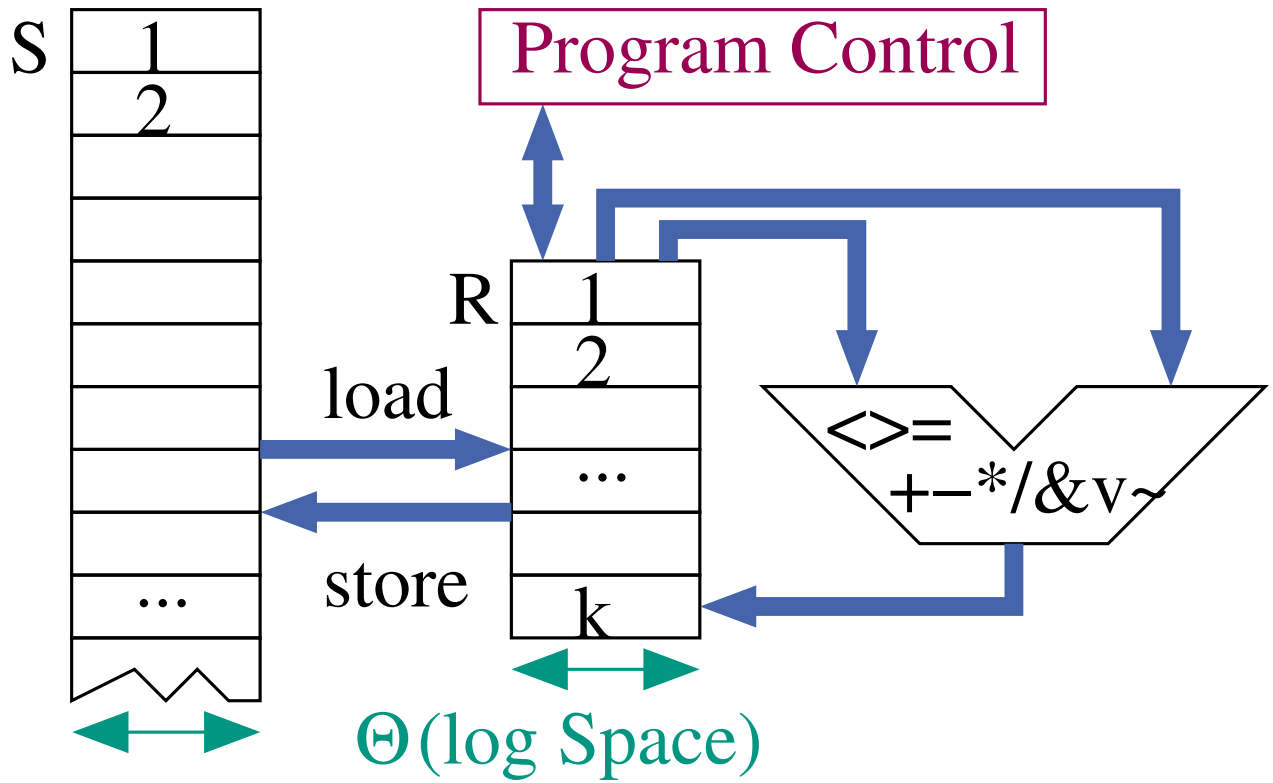


$R_i := R_j \odot R_\ell$ Registerarithmetik.

'$\odot$' ist Platzhalter für eine Vielzahl von Operationen

Arithmetik, Vergleich, Logik

Bedingte Sprünge



JZ j, R_i Setze Programmausführung an Stelle j fort falls $R_i = 0$

„Kleine“ ganze Zahlen?

Alternativen:

Konstant viele Bits (64?): theoretisch unbefriedigend, weil nur endlich viel Speicher adressierbar $\rightsquigarrow$ endlicher Automat

Beliebige Genauigkeit: viel zu optimistisch für vernünftige Komplexitätstheorie. Beispiel: n-maliges Quadrieren führt zu einer Zahl mit $\approx 2^n$ Bits.
OK für Berechenbarkeit

Genug um alle benutzten Speicherstellen zu adressieren: bester Kompromiss.

Algorithmenanalyse im RAM-Modell

Zeit: Ausgeführte Befehle zählen,
d. h. Annahme 1 Takt pro Befehl.
Nur durch späteres $O(\cdot)$ gerechtfertigt!
Ignoriert Cache, Pipeline, Parallelismus. . .

Platz: Etwas unklar:

- letzte belegte Speicherzelle?
- Anzahl benutzter Speicherzellen?
- Abhängigkeit von Speicherverwaltungsalgorithmen?

Hier: Es kommt eigentlich nie drauf an.

Mehr Maschinenmodell

Cache: schneller Zwischenspeicher

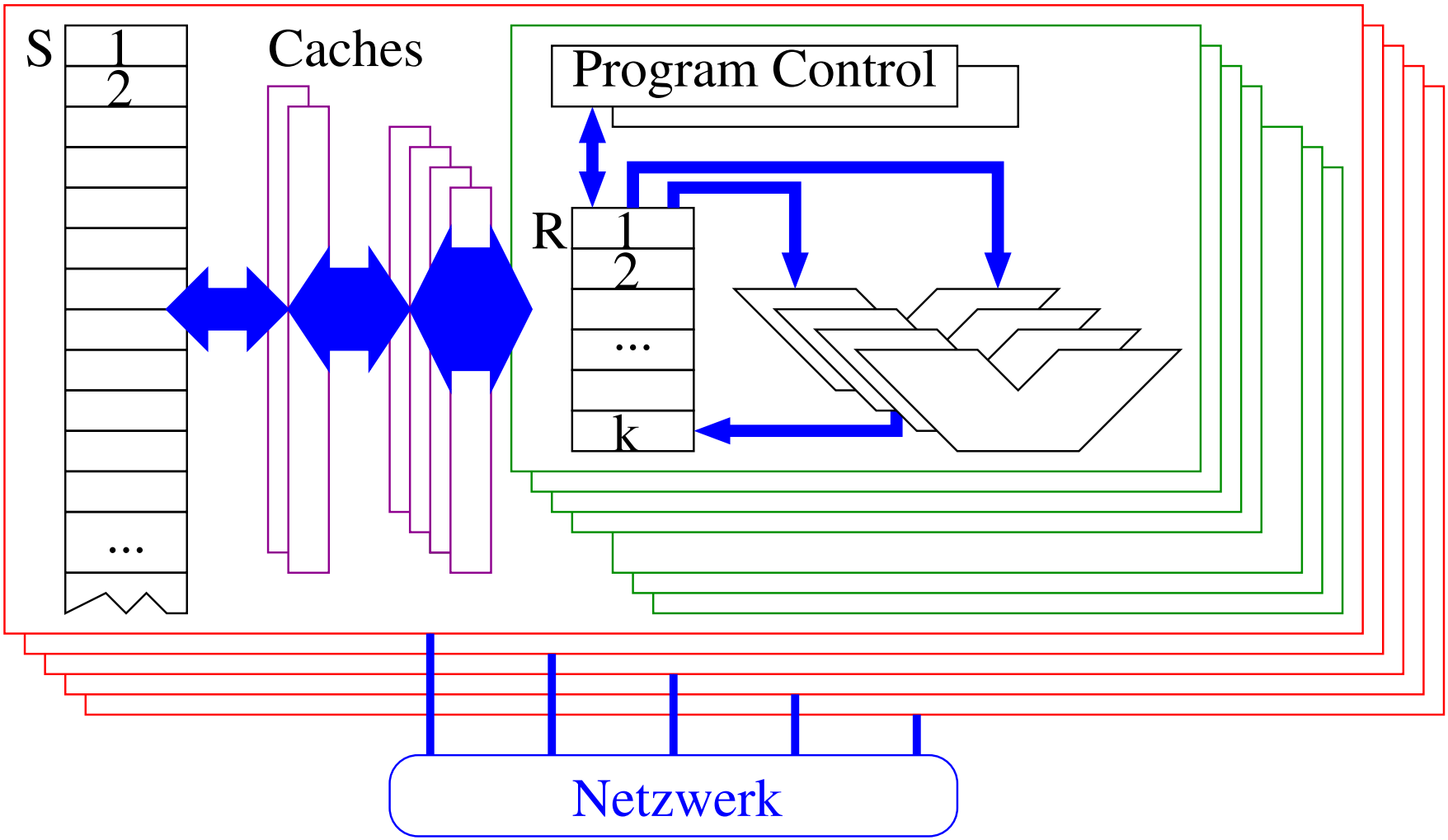
- begrenzte Größe
  $\rightsquigarrow$ kürzlich/häufig zugegriffene Daten sind eher im Cache
- blockweiser Zugriff
  $\rightsquigarrow$ Zugriff auf konsekutive Speicherbereiche sind schnell

Parallelverarbeitung: Mehrere Prozessoren
$\rightsquigarrow$ unabhängige Aufgaben identifizieren

...

mehr in TI, Algorithmen II, Programmierparadigmen,...

Mehr Maschinenmodell



Pseudocode

just in time

Beispiel:

Class Complex(x, y : Number) **of** Number
    Number $r := x$
    Number $i := y$
    Function abs : Number **return** $\sqrt{r^2 + i^2}$
    Function add(c' : Complex) : Complex
        return Complex($r + c'.r, i + c'.i$)

Design by Contract / Schleifeninvarianten

assert: Aussage über Zustand der Programmausführung

Vorbedingung: Bedingung für korrektes Funktionieren einer Prozedur

Nachbedingung: Leistungsgarantie einer Prozedur, falls Vorbedingung erfüllt

Invariante: Aussage, die an „vielen“ Stellen im Programm gilt

Schleifeninvariante: gilt vor / nach jeder Ausführung des Schleifenkörpers

Datenstrukturinvariante: gilt vor / nach jedem Aufruf einer Operation auf abstraktem Datentyp

Hier: Invarianten als zentrales Werkzeug für Algorithmenentwurf und Korrektheitsbeweis.

Beispiel (Ein anderes als im Buch)

Function power($a : \mathbb{R}$; $n_0 : \mathbb{N}$) : $\mathbb{R}$

$p{=}a : \mathbb{R}$; $r{=}1 : \mathbb{R}$; $n{=}n_0 : \mathbb{N}$

while $n > 0$ **do**

if n is odd **then** $n{-}{-}$; $r{:=}\, r \cdot p$

else $(n, p){:=}\, (n/2, p \cdot p)$

return r

Beispiel (Ein anderes als im Buch)

Function power($a : \mathbb{R};\ n_0 : \mathbb{N}) : \mathbb{R}$

  assert $n_0 \geq 0$ and $\neg(a = 0 \wedge n_0 = 0)$ // Vorbedingung

  $p{=}a : \mathbb{R};\quad r{=}1 : \mathbb{R};\quad n{=}n_0 : \mathbb{N}$ // $p^n r = a^{n_0}$

  while $n > 0$ **do**

    invariant $p^n r = a^{n_0}$ // Schleifeninvariante **(*)**

    if n is odd **then** $n{-}{-}$; $r{:=}\, r \cdot p$

    else $(n, p){:=}\, (n/2, p \cdot p)$

  assert $r = a^{n_0}$ // **(*)**$\wedge n = 0 \longrightarrow$Nachbedingung

  return r

Rechenbeispiel: 2^5

$p{=}a = 2 : \mathbb{R}; \quad r{=}1 : \mathbb{R}; \quad n{=}n_0 = 5 : \mathbb{N}$ // $2^5 \cdot 1 = 2^5$

while $n > 0$ **do**

  if n is odd **then** $n{-}{-}$; $r{:=}\, r \cdot p$

  else $(n, p){:=}\, (n/2, p \cdot p)$

Iteration	p	r	n	$p^n r$
0	2	1	5	32
1	2	2	4	32
2	4	2	2	32
3	16	2	1	32
4	32	32	0	**32**

Beispiel

Function power($a : \mathbb{R};\ n_0 : \mathbb{N}) : \mathbb{R}$

assert $n_0 \geq 0$ and $\neg(a = 0 \wedge n_0 = 0)$ // Vorbedingung

$p{=}a : \mathbb{R};\quad r{=}1 : \mathbb{R};\quad n{=}n_0 : \mathbb{N}$ // $p^n r = a^{n_0}$

while $n > 0$ **do**

invariant $p^n r = a^{n_0}$ // Schleifeninvariante **(*)**

if n is odd **then** $n{-}{-}$; $r{:=}\, r \cdot p$

else $(n, p){:=}\, (n/2, p \cdot p)$

assert $r = a^{n_0}$ // **(*)**$\wedge n = 0 \longrightarrow$Nachbedingung

return r

Fall n ungerade: Invariante erhalten wegen $p^n r = p^{\overbrace{n-1}^{\text{neues } n}} \underbrace{pr}_{\text{neues } r}$

Beispiel

Function power($a : \mathbb{R};\ n_0 : \mathbb{N}) : \mathbb{R}$

assert $n_0 \geq 0$ and $\neg(a = 0 \wedge n_0 = 0)$ // Vorbedingung

$p{=}a : \mathbb{R};\quad r{=}1 : \mathbb{R};\quad n{=}n_0 : \mathbb{N}$ // $p^n r = a^{n_0}$

while $n > 0$ **do**

invariant $p^n r = a^{n_0}$ // Schleifeninvariante **(*)**

if n is odd **then** $n{-}{-}$; $r{:=}\, r \cdot p$

else $(n, p){:=}\, (n/2, p \cdot p)$

assert $r = a^{n_0}$ // **(*)** $\wedge n = 0 \longrightarrow$ Nachbedingung

return r

Fall n gerade: Invariante erhalten wegen $p^n = \underbrace{(p \cdot p)}_{\text{neues } p}^{\overbrace{n/2}^{\text{neues } n}}$

Programmanalyse

Die fundamentalistische Sicht: Ausgeführte RAM-Befehle zählen

Pseudo-Code $\overset{\text{einfache Übersetzungsregeln}}{\longrightarrow}$ Maschinenbefehle

Idee: $\mathrm{O}(\cdot)$-Notation vereinfacht die direkte Analyse des Pseudocodes.

- $T(I; I') = T(I) + T(I')$.
- $T(\textbf{if } C \textbf{ then } I \textbf{ else } I') \in \mathrm{O}(T(C) + \max(T(I), T(I')))$.
- $T(\textbf{repeat } I \textbf{ until } C) \in \mathrm{O}(\sum_i T(i\text{-te Iteration}))$

Rekursion $\rightsquigarrow$ Rekurrenzrelationen

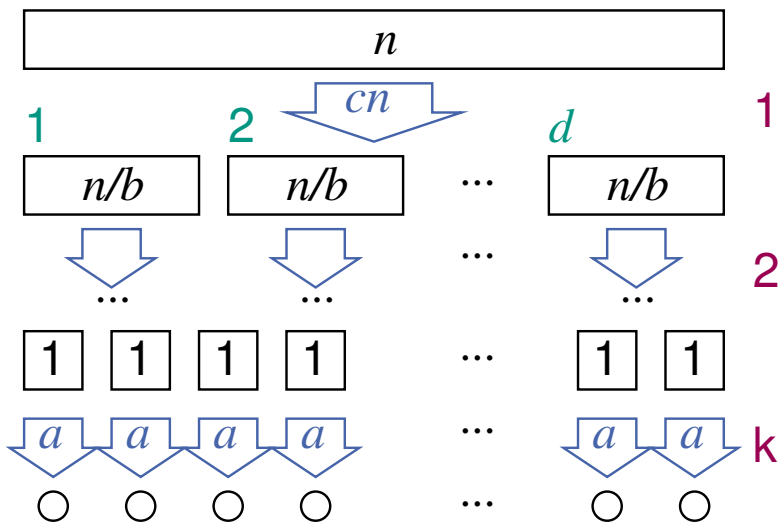
Schleifenanalyse $\rightsquigarrow$ Summen ausrechnen

Das lernen Sie in Mathe
Beispiel: Schulmultiplikation

Eine Rekurrenz für Teile und Herrsche

Für positive Konstanten a, b, c, d, sei $n = b^k$ für ein $k \in \mathbb{N}$.

$$r(n) = \begin{cases} a & \text{falls } n = 1 \text{ Basisfall} \\ cn + dr(n/b) & \text{falls } n > 1 \text{ teile und herrsche.} \end{cases}$$



Master Theorem (Einfache Form)

Für positive Konstanten a, b, c, d, sei $n = b^k$ für ein $k \in \mathbb{N}$.

$$r(n) = \begin{cases} a & \text{falls } n = 1 \text{ Basisfall} \\ cn + dr(n/b) & \text{falls } n > 1 \text{ teile und herrsche.} \end{cases}$$

Es gilt

$$r(n) = \begin{cases} \Theta(n) & \text{falls } d < b \\ \Theta(n \log n) & \text{falls } d = b \\ \Theta\left(n^{\log_b d}\right) & \text{falls } d > b. \end{cases}$$

Beweisskizze

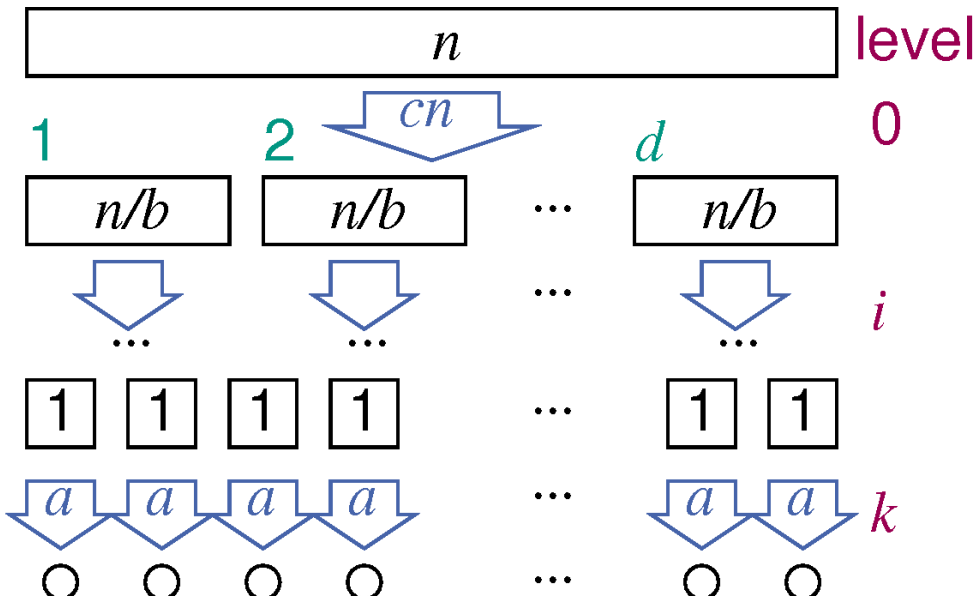
Auf Ebene i haben wir d^i Probleme @ $n/b^i = b^{k-i}$

cost

cn

$$d^i \cdot c \cdot \frac{n}{b^i} = cn\left(\frac{d}{b}\right)^i$$

ad^k

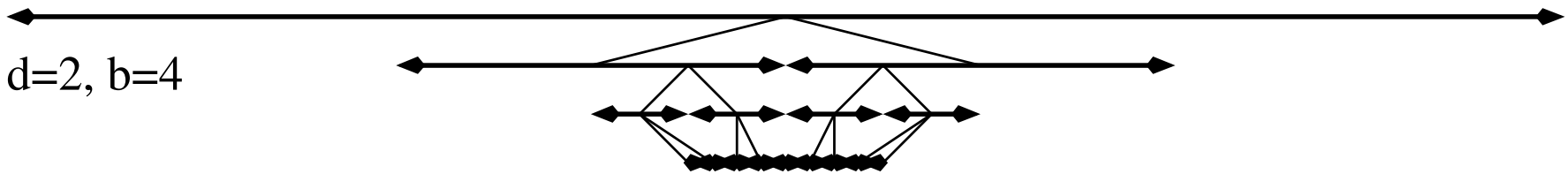


Beweisskizze Fall $d < b$

geometrisch schrumpfende Reihe

$\longrightarrow$ erste Rekursionsebene kostet konstanten Teil der Arbeit

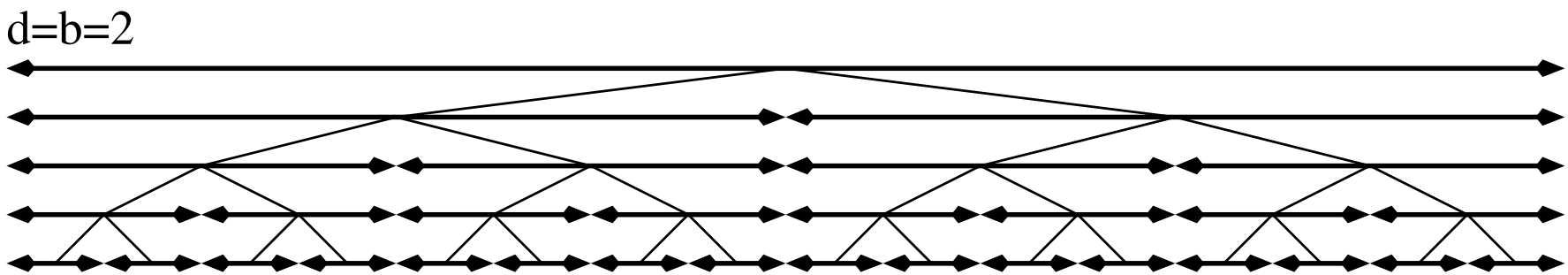
$$r(n) = \underbrace{a \cdot d^k}_{o(n)} + cn \cdot \underbrace{\sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i}_{O(1)} \in \Theta(n)$$



Beweisskizze Fall $d = b$

gleich viel Arbeit auf allen $k = \log_b(n)$ Ebenen.

$$r(n) = an + cn \log_b n \in \Theta(n \log n)$$



Beweisskizze Fall $d > b$

geometrisch wachsende Reihe

$\longrightarrow$ letzte Rekursionsebene kostet konstanten Teil der Arbeit

$$r(n) = ad^k + cn \cdot \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i \in \Theta\left(n^{\log_b d}\right)$$

beachte: $d^k = 2^{k \log d} = 2^{k \frac{\log b}{\log b} \log d} = b^{k \frac{\log d}{\log b}} = b^{k \log_b d} = n^{\log_b d}$