Algorithmentechnik

Peter Sanders und Rob van Stee

Übungen:

Veit Batz und Johannes Singler

Institut für theoretische Informatik, Algorithmik II

Web:

`http://algo2.iti.uni-karlsruhe.de/algotech.php`

Organisatorisches

Vorlesungen:

Di 15:45–17:15

Do 15:45–17:15 14-täglich

Saalübung:

Do 15:45–17:15 14-täglich (erstmals 8.11.)

Übungsblätter: 14-täglich

Auf dem Web bis Do, 8 Tage vor Übung, spätestens 15:45.

Keine Abgabe

Organisatorisches

Sprechstunde:

- Peter Sanders, Dienstag 14–15 Uhr, Raum 217

Klausuren:

Mi 27.2.2008, Beginn 14:00 Uhr

Do 10.4.2008, Beginn 9:00 Uhr

Materialien

- Folien, Übungsblätter
- Buch:

  K. Mehlhorn, P. Sanders

  Algorithms and Data Structures — The Basic Toolbox

  Springer 2008
- Materialien aus der Vorlesung von Frau Wagner

  `http://i11www.iti.uni-karlsruhe.de/teaching/WS_0607/algotech/`

Inhaltsübersicht (provisorisch)

1. Einführung
2. Folgen, Felder, Listen, (amort. Analyse und mehr)
3. Universal Hashing (Theorie wird realistisch)
4. Sortieren I (Tips, Tricks, Theorie: rand. Alg., untere Schranken,. . .)
5. Sortieren II: große Datenmengen
6. Addressierbare Prioritätslisten (endlich alle Operationen)
7. Suchbäume: mehr Operationen
8. Graphrepräsentation (wie es wirklich geht)
9. Graphtraversierung und starke Zusammenhangskomponenten (Invarianten, Zertifikate)
10. Kürzeste Wege I: radix heaps, Linearzeit im Mittel

11. Kürzeste Wege II: negative Kreise, Potentialmethode
12. Minimale Spannbäume
13. Rucksackproblem (greedy, dyn. Programm., Approx.-Alg.)
14. Scheduling (Onlinealgorithmen, mehr Approx.)
15. Lineare Programmierung
16. Lokale Suche und evol. Algorithmen
17. Maximale Flüsse I: Worum es geht
18. Maximale Flüsse II: highest label preflow push
19. Stringalgorithmen: sorting, indexing,. . .
20. Geometrische Algorithmen

Programmierprojekt

Informaticup 2007 `http://www.informaticup.de/`

Aufgabe 3: Entfernen subsumierter Datenbankeinträge. (3.2 ist der wichtigste Teil ?)

Deadline: 15.1.2008

Danach: interner Wettbewerb

Letze Vorlesungswoche: Vorstellung der besten Ergebnisse

Subsumption kurz gefasst

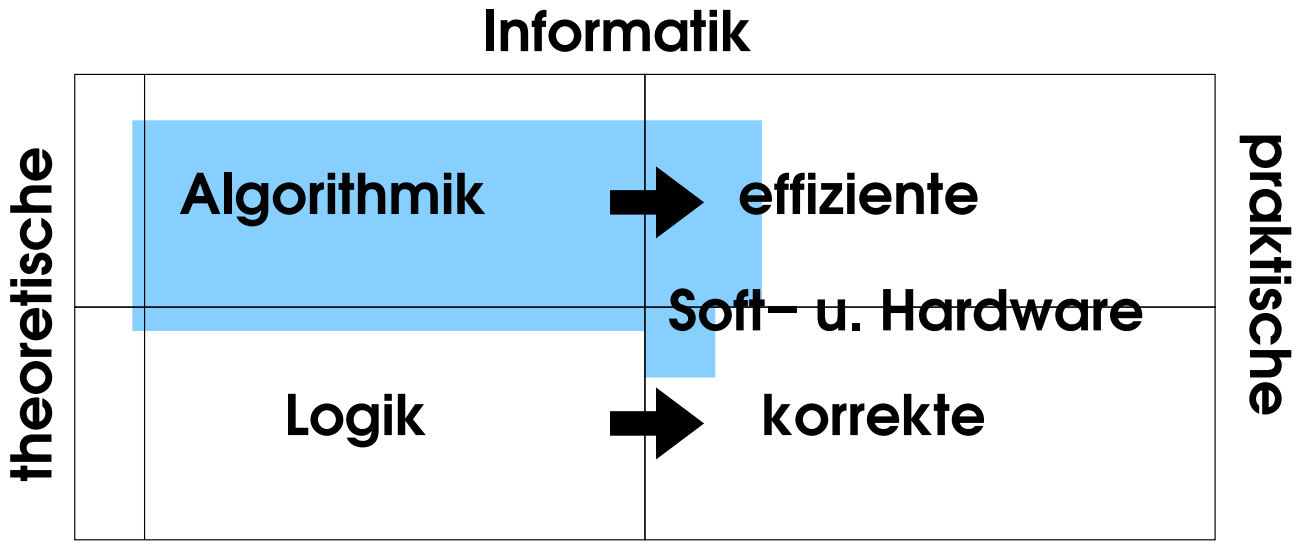
Gegeben eine Menge $R \subseteq (\mathbb{N} \cup \{\perp\})^k$.

Bestimme $\left\{ x \in R : \exists y \in R : x \neq y \wedge \bigwedge_{i=1}^{k} x_i = y_i \vee x_i = \perp \right\}$

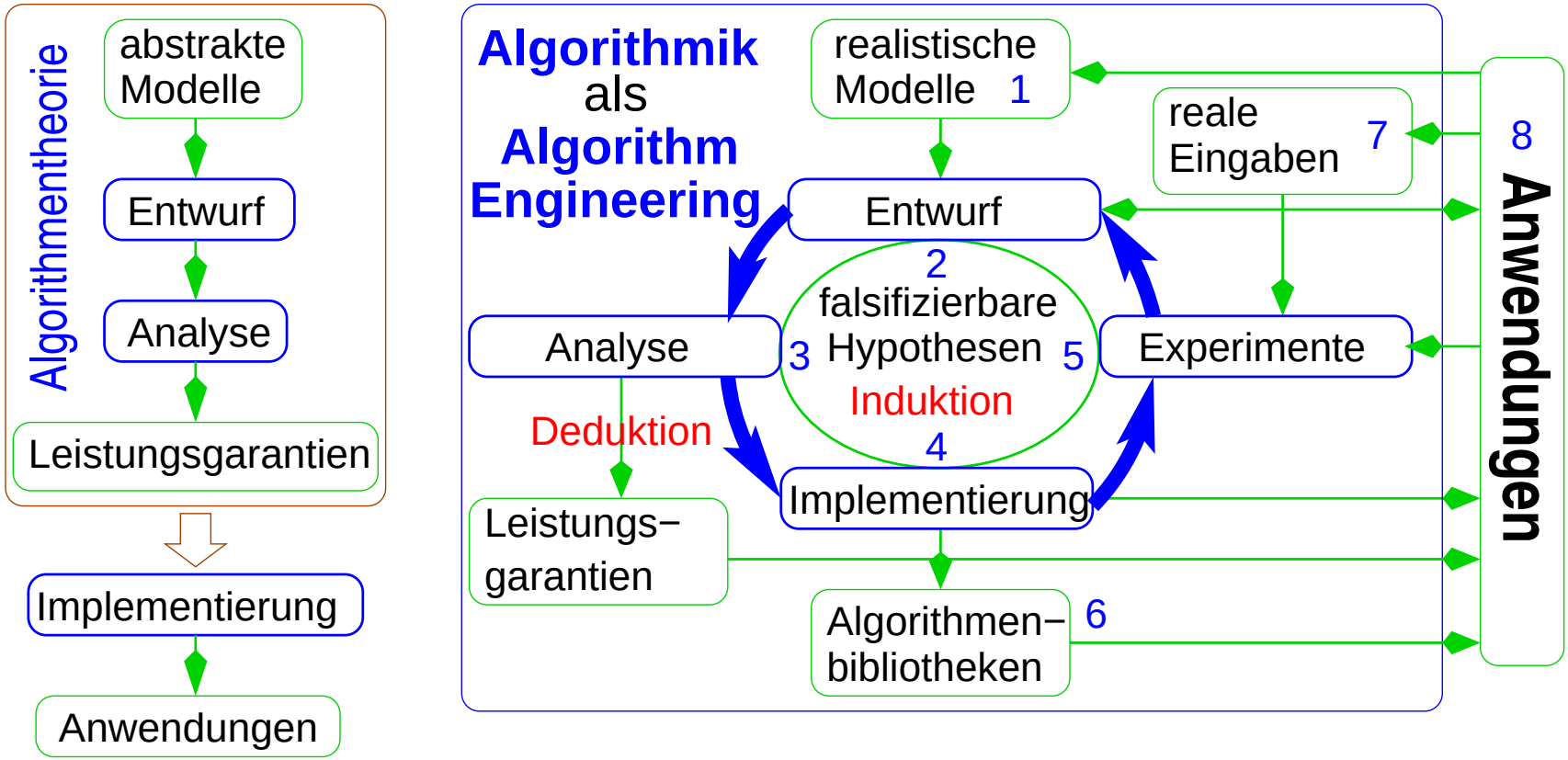
Algorithmik

Kerngebiet der (theoretischen) Informatik

mit direktem Anwendungsbezug



Algorithmentechnik $\stackrel{?}{\approx}$ Algorithm Engineering



Trotzdem machen wir hier $2/3$ Algorithmentheorie

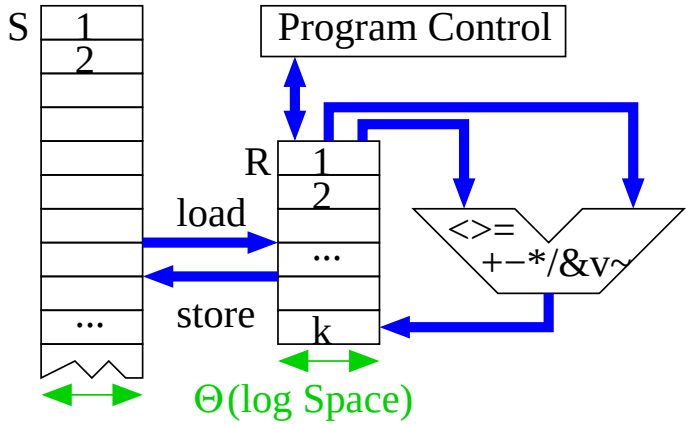
– der praktikable Teil

2 Einführendes

RAM/von Neumann Modell

Analyse: zähle Maschinenbefehle — load, store, Arithmetik, Branch,. . .

- Einfach
- Sehr erfolgreich



Algorithmenanalyse:

- Zyklen zählen: $T(I)$, für gegebene Probleminstanz I.
- Worst case in Abhängigkeit von Problemgröße:
  $T(n) = \max_{|I|=n} T(I)$
- Average case: $T_{\text{avg}}(n) = \frac{\sum_{|I|=n} T(I)}{|\{I : |I| = n\}|}$
  Beispiel: Quicksort hat average case Ausführungszeit $\mathrm{O}(n \log n)$
- Probabilistische (randomisierte) Algorithmen:
  $T(n)$ (worst case) ist eine Zufallsvariable.
  Wir interessieren uns z.B. für deren Erwartungswert (später mehr).
  Nicht mit average case verwechseln.
  Beispiel: Quicksort mit *zufälliger* Pivotwahl hat erwarteten worst case Aufwand $\mathbb{E}[T(n)] = \mathrm{O}(n \log n)$

Algorithmenanalyse: Noch mehr Konventionen

- $\mathrm{O}(\cdot)$ plättet lästige Konstanten
- Sekundärziel: Speicherplatz
- Die Ausführungszeit kann von mehreren Parametern abhängen:

  Beispiel: Eine effiziente Variante von Dijkstra's Algorithmus für kürzeste Wege benötigt Zeit $\mathrm{O}(m + n \log n)$ wenn n die Anzahl Knoten und m die Anzahl Kanten ist.

  (Es muss immer klar sein, welche Parameter was bedeuten.)

Mehr Asymptotik

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

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

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

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

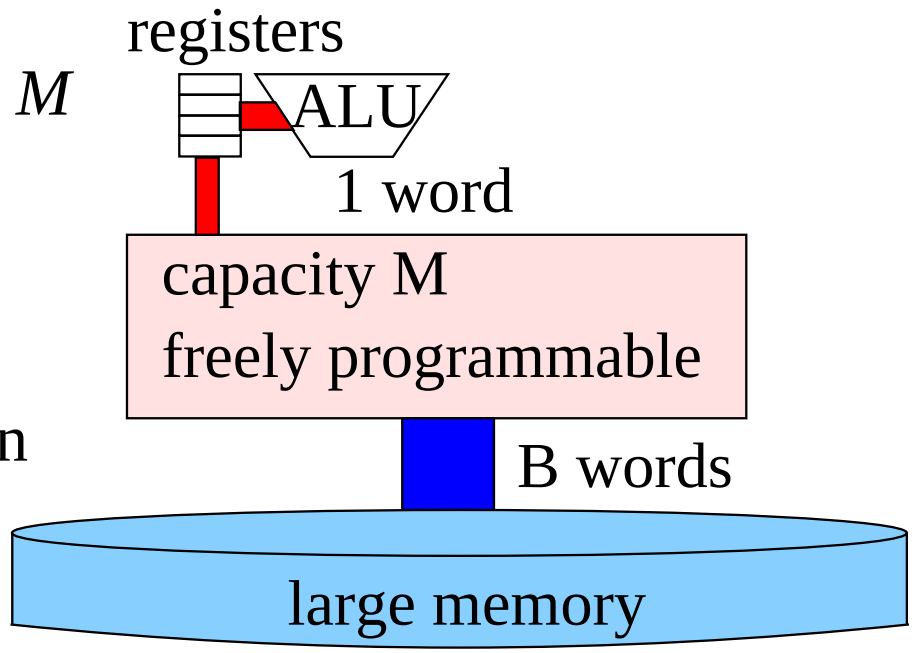
$$\omega(f(n)) = \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$$

Das Sekundärspeichermodell

M: Schneller Speicher der Größe M

B: Blockgröße

Analyse: Blockzugriffe zählen



Pseudocode

- Pascal-ähnlich
- Einrückung statt begin-end-Schachteln
- freigiebiger Gebrauch mathematischer Notation
- $-\infty$ / ∞ statt kleiner/großer Werte
- $\perp$ unterscheidbar von „echten“ Werten
- rudimentär objekt-orientiert und generisch

Beispiel

Function quickSort(s : Sequence **of** Element) : Sequence **of** Element

if $|s| \leq 1$ **then return** s // base case

pick $p \in s$ uniformly at random // pivot key

$a := \langle e \in s : e < p \rangle$

$b := \langle e \in s : e = p \rangle$

$c := \langle e \in s : e > p \rangle$

return quickSort$(a) \circ \langle b \rangle \circ$ quickSort(c)

Notation für Folgen (sequences) analog Mengennotation.

Repräsentiert durch Felder, verkettete Listen, Zeichenketten. . .

Beispiel

Class Complex$(x, y$: Element$)$ **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

invariant: 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

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=$a : \mathbb{R}$; r=$1 : \mathbb{R}$; 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

Programmanalyse

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

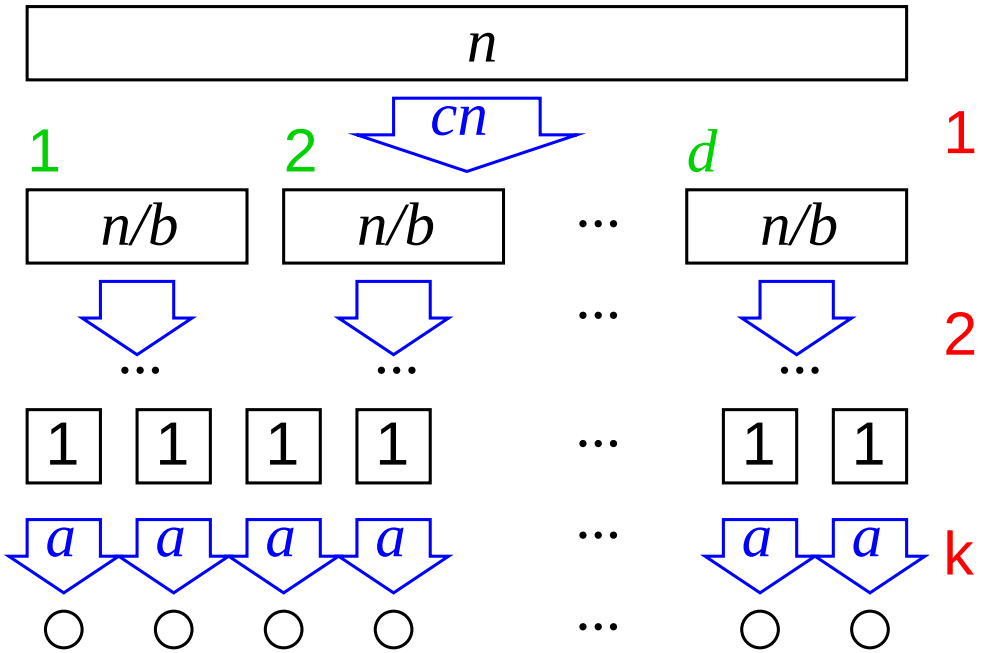
- $T(I;I') = T(I) + T(I')$.
- $T(\textbf{if } C \textbf{ then } I \textbf{ else } I') = \mathrm{O}(T(C) + \max(T(I), T(I')))$.
- $T(\textbf{repeat } I \textbf{ until } C) = \mathrm{O}(\sum_i T(i\text{-te Iteration}))$

Rekursion $\rightsquigarrow$ Rekurrenzrelationen

Eine Rekurrenz für Teilen und Herrschen

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 $k \in \mathbb{N}$.

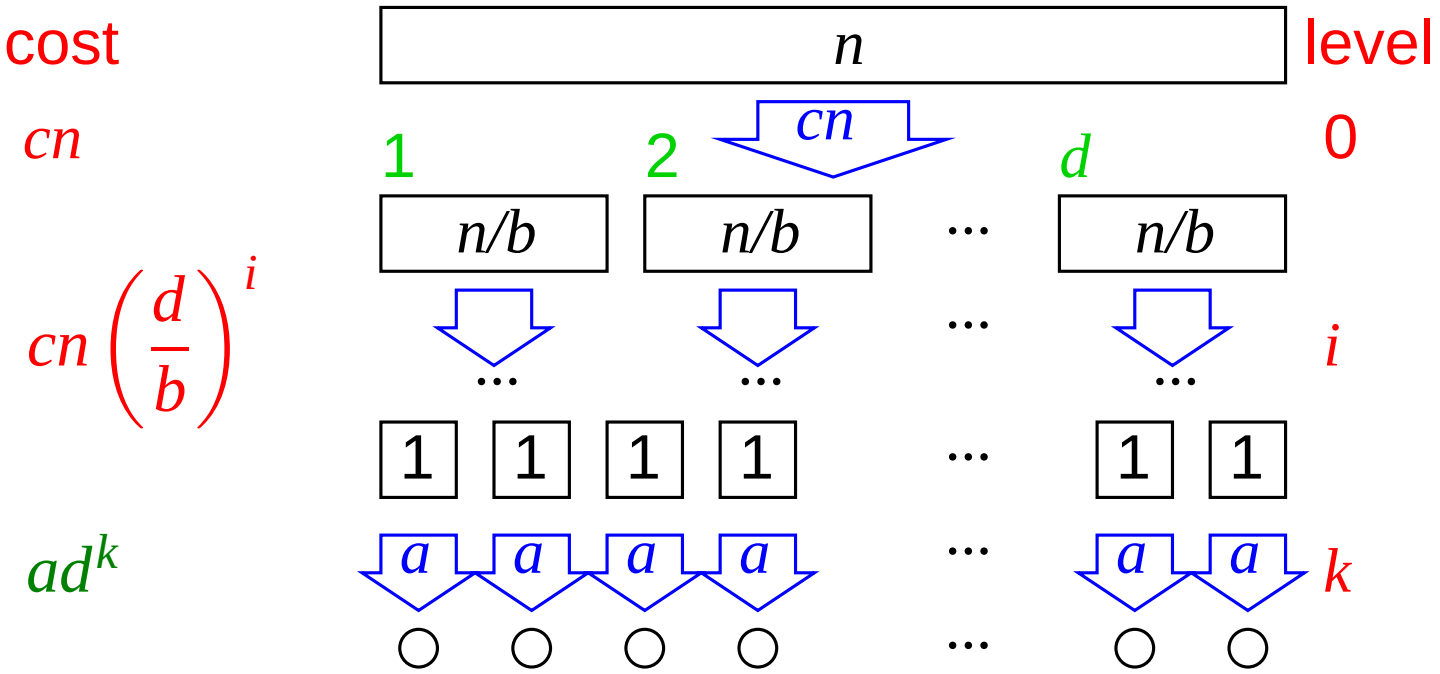
$$r(n) = \begin{cases} a & \text{falls } n = 1 \text{ Basisfall} \\ cn + dr(n/b) & \text{falls } n > 1 \text{ teile und herrsche.} \end{cases}$$

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 Problem @ $n/b^i = b^{k-i}$



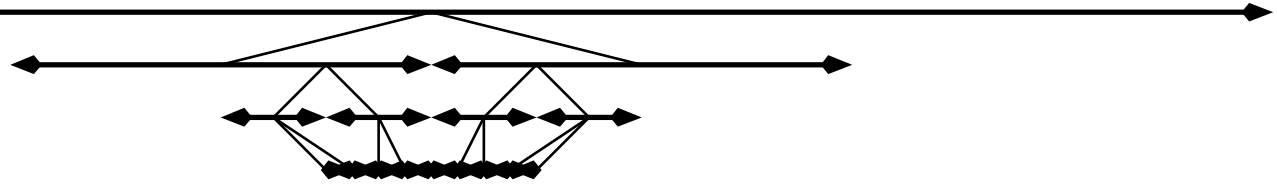
Beweisskizze Fall $d < b$

geometrisch schrumpfende Reihe

$\longrightarrow$ erste Rekursionsebene kostet konstanten Teil der Arbeit

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

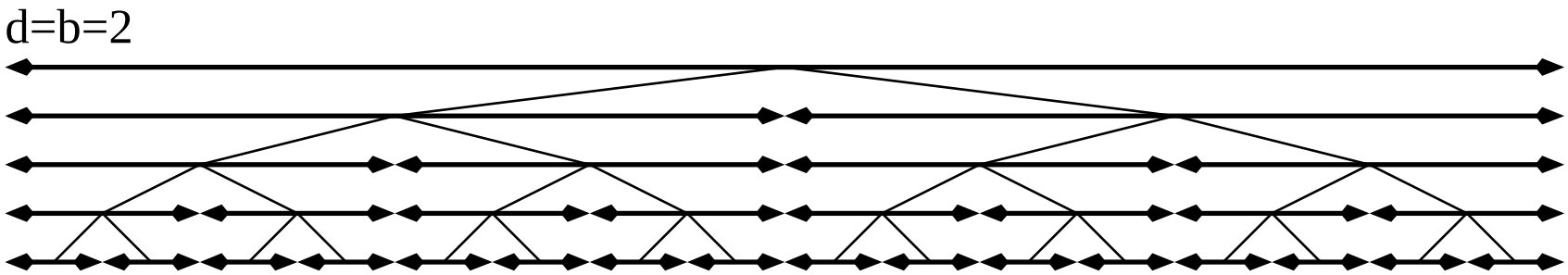
d=2, b=4



Beweisskizze Fall $d = b$

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

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



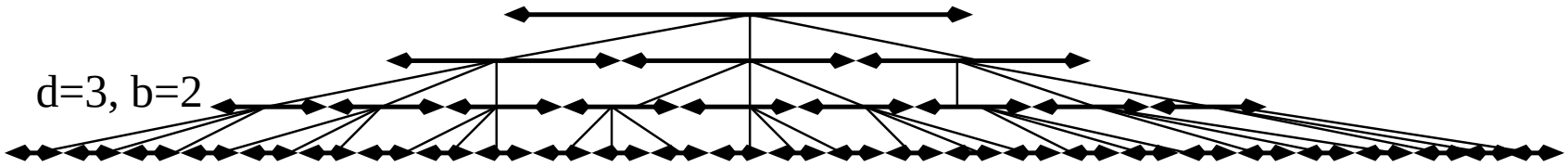
Beweisskizze Fall $d > b$

geometrisch wachsende Reihe

$\longrightarrow$ letzte Rekursionsebene kostet konstanten Teil der Arbeit

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

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



Beispiel: Langzahl-Multiplikation

Schreibe Zahlen als Ziffernfolgen $a = (a_{n-1} \ldots a_0)$, $a_i \in 0..B-1$.

Wir zählen

Ziffernmultiplikationen $p := a_i \cdot b_j$ und

Volladditionen $(c', s) := a_i + b_j + c_k$.

Addition

c=0 : Digit // Variable for the carry digit

for $i := 0$ **to** $n-1$ **do** $(c, s_i) := a_i + b_i + c$

$s_n = c$

Satz: Addition von n-Ziffern-Zahlen braucht n Ziffern-Additionen.

Ein erster Teile-und-Herrsche Algorithmus

Function recMult(a,b)

assert a und b haben $n = 2k$ Ziffern, n ist Zweierpotenz

if $n = 1$ **then return** $a \cdot b$

Schreibe a als $a_1 \cdot B^k + a_0$

Schreibe b als $b_1 \cdot B^k + b_0$

return

$$\text{recMult}(a_1,b_1) \cdot B^{2k} +$$
$$(\text{recMult}(a_0,b_1) + \text{recMult}(a_1,b_0))B^k$$
$$+\text{recMult}(a_0,b_0)$$

Analyse

Function recMult(a,b) // $T(n)$ Ops

assert a und b haben $n = 2k$ Ziffern, n ist Zweierpotenz

if $n = 1$ **then return** $a \cdot b$ // 1 Op

Schreibe a als $a_1 \cdot B^k + a_0$ // 0 Ops

Schreibe b als $b_1 \cdot B^k + b_0$ // 0 Ops

return

recMult$(a_1,b_1) \cdot B^{2k} +$ // $T(n/2) + 2n$ Ops

$($recMult$(a_0,b_1) +$ recMult$(a_1,b_0))B^k$ // $2T(n/2) + 2n$ Ops

$+$recMult(a_0,b_0) // $T(n/2) + 2n$ Ops

Also $T(n) \leq 4T(n/2) + 6n$

Übung: Wo kann man hier $\approx 2n$ Ops sparen ?

Analyse

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ 4 \cdot T(\lceil n/2 \rceil) + 6 \cdot n & \text{if } n \geq 2. \end{cases}$$

$\longrightarrow$ (Master-Theorem)

$$T(n) = \Theta\left(n^{\log_2 4}\right) = \Theta\left(n^2\right)$$

Karatsuba-Multiplikation[Karatsuba-Ofman 1962]

Beobachtung: $(a_1 + a_0)(b_1 + b_0) = a_1 b_1 + a_0 b_0 + a_1 b_0 + a_0 b_1$

Function recMult(a, b)

assert a und b haben $n = 2k$ Ziffern, n ist Zweierpotenz

if $n = 1$ **then return** $a \cdot b$

Schreibe a als $a_1 \cdot B^k + a_0$

Schreibe b als $b_1 \cdot B^k + b_0$

c_{11}:= recMult(a_1, b_1)

c_{00}:= recMult(a_0, b_0)

return

$$c_{11} \cdot B^{2k} +$$
$$(\text{recMult}((a_1 + a_0), (b_1 + b_0)) - c_{11} - c_{00})B^k$$
$$+ c_{00}$$

Analyse

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ 3 \cdot T(\lceil n/2 \rceil) + 10 \cdot n & \text{if } n \geq 2. \end{cases}$$

$\longrightarrow$ (Master-Theorem)

$$T(n) = \Theta\left(n^{\log_2 3}\right) \approx \Theta\left(n^{1.58}\right)$$

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

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