ADS: Algorithmen und Datenstrukturen

Akuter Denk-Stau

Prof. Peter F. Stadler & Dr. Christian Höner zu Siederdissen

Bioinformatik/IZBI
Institut für Informatik
& Interdisziplinäres Zentrum für Bioinformatik
Universität Leipzig

12. Oktober 2015

[Letzte Aktualisierung: 20/10/2015, 00:11]

Ankündigungen im Netz

Anmeldung zu Übungen, Übungsaufgaben, Vorlesungsfolien, Termin- und Raumänderungen, ...

http://www.bioinf.uni-leipzig.de

→ Teaching → Current classes → Algorithmen und Datenstrukturen 1

`www.bioinf.uni-leipzig.de/currentClasses/class210.html`

Übungsaufgaben

- Abzugeben sind Lösungen zu sechs Aufgabenblättern.
- Termine

Aufgabenblatt	1	2	3	4	5+6
Ausgabe	26.10.	9.11.	23.11.	07.12.	04.01.
Abgabe	2.11.	16.11.	30.11.	14.12.	11.01.

- Lösungen sind **spätestens direkt vor** Beginn der Vorlesung im Hörsaal abzugeben.
- Lösungen werden bewertet und in der auf den Abgabetermin folgenden Übungsstunde zurückgegeben.

Übungstermine

Vorläufige Termine der Übungsgruppen

Gruppe	Tag	Uhrzeit	
01	Mo	11:15–12:45	
02	Mo	11:15–12:45	
03	Mo	17:15–18:45	
04	Mo	17:15–18:45	
05	Di	9:15 –10:45	
06	Di	9:15 –10:45	
07	Di	11:15–12:45	(englisch)
08	Di	11:15–12:45	
09	Mi	13:15–14:45	
10	Fr	7:30 – 9:00	
11	Fr	9:15 –10:45	
12	Fr	9:15 –10:45	

Übungen — Anmeldung

Anmeldung für die Übungen

Für die Teilnahme an den Übungen (und damit die Anerkennung von Übungsleistungen!) ist eine Anmeldung für eine Übungsgruppe **unbedingt** erforderlich!

Freischaltung der Online-Anmeldung:

Montag, 12.10., direkt nach der Vorlesung

—

Montag, 19.10., vor der Vorlesung

`http://www.bioinf.uni-leipzig.de` → Teaching → Current classes → Algorithmen und Datenstrukturen 1

Klausur

Termin (voraussichtlich): 01.02.2016, 15:15–16:45 Uhr

Zulassungsvoraussetzungen

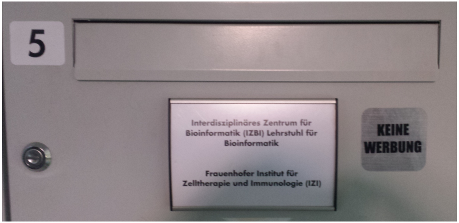
- Modul- UND Prüfungsanmeldung in `almaweb` bis 18.10.2015
- Erreichen von 50% der Punkte in den Übungsaufgaben
- Fähigkeit, abgegebene Lösungen an der Tafel zu erläutern
- $\rightarrow$ wird in den Übungen überprüft
- $\rightarrow$ abgeben allein reicht nicht
- wer absolut nicht kann: frühzeitig Alternative für Überprüfung mit Übungsleiter abklären

Übungsmodalitäten

- Team-Arbeit bei den Übungsblättern gestattet
- jeder muß alle abgegeben Lösungen erklären können
- jeder muß eine eigene Abgabe haben
- keine Kopien von Ausarbeitungen
- absolut keine Möglichkeit per email abzugeben
- Lösungen links oben tackern; kein: kleben, vernähen, Büroklammern, schweissen, sonstwas
- Name und Matrikelnummer auf jedes Blatt

Übungsmodalitäten II

- wenn Sie direkt vor der Vorlesung nicht abgeben können:
- geben Sie die Abgabe einem einem Ihrer Kommilitonen mit zur Abgabe direkt vor der Vorlesung
- oder werfen Sie die Abgabe bis spätestens 09 Uhr (Neun Uhr Morgens) am Abgabetag in den Briefkasten Nr. 5 der BioInformatik in der Härtelstraße 16–18.



`http://www.bioinf.uni-leipzig.de`

$\rightarrow$ Teaching $\rightarrow$ Current classes
$\rightarrow$ Algorithmen & Datenstrukturen 1

(`http://www.bioinf.uni-leipzig.de/teaching/currentClasses/class210.html`)

Inhalt

Wozu das Ganze?

- Algorithmen stehen im Mittelpunkt der Informatik
- Entwurfsziele bei Entwicklung von Algorithmen:
  1. Korrektheit Cool! Wir werden Beweise führen
  2. Terminierung Kommt später im Studium
  3. Effizient Doof kann's (fast) jeder
- Wahl der Datenstrukturen ist für Effizienz entscheidend
- Schwerpunkt der Vorlesung: Entwurf von effizienten Algorithmen und Datenstrukturen, sowie die nachfolgende Analyse ihres Verhaltens

Wozu das Ganze?

- Funktional gleichwertige Algorithmen weisen oft erhebliche Unterschiede in der Effizienz (Komplexität) auf.
- Bei der Verarbeitung soll effektiv mit den Daten umgegangen werden.
- Die Effizienz hängt bei großen Datenmengen ab von
  - internen Darstellung der Daten
  - dem verwendeten Algorithmus
- Zusammenhang dieser Aspekte?

Beispiel Telefon-CD

Anforderungen

- 700 MB Speicherplatz
- 40 Millionen Telefone $\times$ 35 Zeichen (Name Ort Nummer)
  1.4 GB ASCII-Text
- **passt nicht**
- Wir brauchen eine Komprimierung der Daten, und eine Möglichkeit schnell zu suchen (Index!).
- (Technische Nebenbemerkung: ausserdem ist der Speicherzugriff auf der CD sehr langsam)

Design-Überlegungen

- Verwende Datenstruktur, die die vollständige Verwaltung der Einträge erlaubt: Suchen, Einfügen und Löschen.

– ODER –

- Weil sowieso nur das Suchen erlaubt ist:
  verwende Datenstruktur, die zwar extrem schnelles Suchen in komprimierten Daten erlaubt, aber möglicherweise kein Einfügen.

Also: Mitdenken hilft

Effizienz: Zeit und Speicher

- Die Abarbeitung von Programmen (Software) beansprucht zwei Ressourcen:
  Zeit und Hardware (insbesondere: **Speicher**).
- **FRAGE:** Wie steigt dieser Ressourcenverbrauch bei größeren Problemen (d.h. mehr Eingabedaten)?
- Es kann sein, dass Probleme ab einer gewissen Größe praktisch unlösbar sind, weil
  1. Ihre Abarbeitung zu lange dauern würde (z.B. länger als ein Informatikstudium) oder
  2. Das Programm mehr Speicher braucht, als zur Verfügung steht.

Wichtig ist auch der Unterschied zwischen **internem (RAM) und externem Speicher**, da z.B. der Zugriff auf eine Festplatten ca. 100.000 mal langsamer ist als ein RAM-Zugriff.

Ressourcenbedarf von Algorithmen

- **Wesentliche Ressourcen**:
  - Rechenzeitbedarf
  - Speicherplatzbedarf
- **Programmlaufzeit abhängig von**
  - Eingabe für das Programm
  - Qualität des vom Compiler generierten Codes
  - Leistungsfähigkeit der Maschine, die das Programm ausgeführt
  - "Kompliziertheit" des Algorithmus, den das Programm implementiert
- **Maß für "Algorithmus-Kompliziertheit": Komplexität**
  - Rechenzeitbedarf $\longrightarrow$ Zeitkomplexität
  - Speicherplatzbedarf $\longrightarrow$ Platzkomplexität

Komplexität von Algorithmen

- **Komplexität**: Maß für Kosten eines Algorithmus
- **Bedeutung**: K. hängt nur vom Algorithmus ab; unabhängig von übrigen Faktoren, die Rechenzeit und Hardwarebedarf beeinflussen
- **Bestimmung der Komplexität**
  - Messungen auf einer bestimmten Maschine
  - Aufwandsbestimmungen für idealisierten Modellrechner (Bsp.: Random-Access-Maschine oder RAM)
  - *Abstraktes* Komplexitätsmaß zur *asymptotischen* Kostenabschätzung in Abhängigkeit von Problemgöße/Eingabegröße n

abstrakt $\equiv$ unabhängig von konkreter Maschine (konstanter Faktor)
asymptotisch $\equiv$ muss nur für große Probleme gelten

Best Case, Average Case, Worst Case

- Komplexität hängt ab
  - von **Eingabegröße**; z.B. Anzahl der Datensätze, über die gesucht werden soll
  - **ausserdem**: von weiteren **Eigenschaften der Eingabe**; z.B. Reihenfolge der Datensätze (unsortiert – grob vorsortiert – sortiert)
- Meist Abschätzung der *worst case* Komplexität, d.h. unter Annahme der für den Algorithmus ungünstigsten Eingabe der jeweiligen Größe (z.B. bestimmte Reihenfolge)
- Analyse der *average case* Komplexität oft viel schwerer; *best case* weniger interessant.

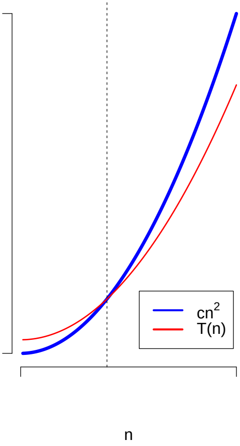
Obere Abschätzung asymptotischer Kosten

- Meist Abschätzung oberer Schranken: **Groß-Oh-Notation**

Beispiel: Die asymptotische Komplexität $T(n)$ eines Algorithmus ist durch die Funktion $f(n) = n^2$ nach oben beschränkt, wenn es Konstanten n_0 und $c > 0$ gibt, so daß für alle Werte von $n > n_0$ gilt:

$$T(n) \leq c \cdot n^2$$

man sagt "$T(n)$ ist in $O(n^2)$", in Zeichen $T(n) \in O(n^2)$ oder einfach $T(n) = O(n^2)$ (oder im Informatikerslang: "der Algo hat quadratische Komplexität").



Allgemeine Definition

Jetzt wird's ernst

Wir definieren eine Menge $O(f)$ aller Funktionen der Grössenordnung f, d.h. aller Funktionen die durch f nach oben beschränkt sind.

Definition: Die Klasse $O(f)$ der Funktionen von der Grössenordnung f ist

$$O(f) = \{g | \exists c > 0 \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq cf(n)\}$$

Beispiele für Groß-Oh Definition

$$\boxed{O(f) = \{g | \exists c > 0 \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq cf(n)\}}$$

Beispiel. $T(n) = 6n^4 + 3n^2 - 7n + 42 \log n + \sin\cos(2n)$

Behauptung: $T(n) \in O(n^4)$

Beweis: Für $n \geq 1$ gilt: $n^4 \geq n^3 \geq n^2 \geq n \geq logn$ und $n^4 \geq 1 \geq \sin(\text{irgendwas})$. Also ist $(6 + 3 + 7 + 42 + 1)n^4$ auf jeden Fall grösser als $T(n)$.
Damit folgt die Behauptung aus $\forall n \geq 1 : T(n) < 59f(n)$. $\square$

Alles klar?

Dann: Warum gilt
$g(n) \in O(n) \Rightarrow g(n) \in O(n \log n) \Rightarrow g(n) \in O(n^2)$**?**

Untere Schranken

So wie $T(n) \in O(f)$ eine obere Schranke ausdrückt, beschreibt $T(n) \in \Omega(f)$ eine untere Schranke.

$g \in \Omega(f)$ bedeutet, dass g (asymptotisch und in der Grössenordnung) mindestens so stark wächst wie f.

Definition:

$$\Omega(f) = \{h | \exists c > 0 \exists n_0 > 0 : \forall n > n_0 : h(n) \geq cf(n)\}$$

Anmerkung: alternativ kann man definieren

$$\Omega(f) = \{h | \exists c > 0 : \exists \text{ unendlich viele } n : h(n) \geq cf(n)\}$$

Exakte Schranke (d.h., nach oben *und* unten)

Gilt sowohl $g \in O(f)$ als auch $g \in \Omega(f)$ schreibt man $g \in \Theta(f)$.

Also:
$g \in \Theta(f)$ bedeutet: die Funktion g verläuft fuer hinreichend grosse n im Bereich $[c_1 f, c_2 f]$ mit geeigneten Konstanten c_1 und c_2.

[Formale Definition zur Übung analog zu $O(f)$ und $\Omega(f)$]

Satz: Ist $T(n)$ ein Polynom vom Grad p, dann ist $T(n) \in \Theta(n^p)$
Wachtumsordnung = höchste Potenz

Polynom vom Grad p

Satz: Ist $T(n)$ ein Polynom vom Grad p, dann ist $T(n) \in \Theta(n^p)$
Wachtumsordnung = höchste Potenz

1. $T(n) = c_1 n^p + c_2 n^(p-1) + \cdots + c_{p+1} n^0$
2. $T(n) \in O(n^p)$
3. $T(n) \in \Omega(n^p)$

$\Rightarrow T(n) \in \Theta(n^p)$

Wichtige Wachstumsfunktionen

$O(1)$ konstante Kosten

$O(logn)$ logarithmisches Wachstum

$O(n)$ lineares Wachstum

$O(n \log n)$ $n \log n$-Wachstum

$O(n^2)$ quadratisches Wachstum

$O(n^3)$ kubisches Wachstum

$O(2^n)$ exponentielles Wachstum

$O(n!)$ Wachstum der Fakultät
$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n$

HAUSÜBUNG: Rechnen Sie das mal aus für $n = 10, 100, 1000, 10000, 100000, 1000000$. **Bei welcher Platzkomplexität passt das jeweils noch auf ihren Laptop (z.B. unter Annahme eines konstanten Faktors von 4 Bytes)?**

Problemgröße bei vorgegebener Zeit

Annahme $n = 1$ in 1ms ...

$\log_2 n$	2^{1000}	2^{60000}	$2^{3600000}$
n	1000	60000	3600000
$n \log_2 n$	140	4893	20000
n^2	31	244	1897
n^3	10	39	153
2^n	9	15	21

Rechengeschwindigkeit

Welche Problemgröße kann bei verschiedenen Kostenfunktionen mit Rechnern verschiedener Geschwindigkeit bearbeitet werden?

Zeitbedarf $T(n)$ bei gegebener Rechengeschwinding, $T_k(n)$ bei k-facher Geschwindigkeit

Bei gleicher Rechenzeit: $T(n) = KT_k(n)$

Alternativ:
Rechner #2 löst Problem der Größe n_k
$T(n) = T_k(n_k) = kT(n_k)$
Lösung: $n_k = T^{-1}(k(T(n)))$

Beispiele: (1) $T(n) = n^m$, (2) $T(n) = 2^n$

Komplexitätsklassen

- linear-beschränkt, $T \in O(n)$
- polynomial-beschränkt, $T \in O(n^k)$
- exponentiell-beschränkt, $T \in O(\alpha^n)$

Exponentiell-zeitbeschränkte Algorithmen sind im Allgemeinen, d.h. für große n, nicht nutzbar. (Vergleiche das gerade betrachtete Beispiel $T(n) = 2^n$)

Probleme, für die kein polynomial-zeitbeschränkter Algorithmus existiert, heissen deshalb *intractable* ($\approx$ "unlösbar", hart).

(umgekehrt: polynomial-beschränkt $\equiv$ effizient / effizient lösbar)

Rechenregeln für die Zeitkomplexität I:

Mit folgenden Regeln können sie die Komplexität eines Algorithmus (nach oben) abschätzen:

- **Elementare Operationen**: $O(1)$
  z.B. Zuweisungen, 32/64bit-Arithmetik, Ein-/Ausgabe, etc.
- **Fallunterscheidungen** $O(1)$
- **Schleife** Produkt aus Anzahl der Schleifendurchläufe mit Kosten der teuersten Schleifenausführung
- **Summenregel**: Das Programm P führe die Programmteile P_1 und P_2 hintereinander aus. Die Komplexitäten von P_1 und P_2 seien $T_1(n) \in O(f(n))$ und $T_2(n) \in O(g(n))$. Die Komplexität von P wird berechnet nach der Summenregel

$$T_1(n) + T_2(n) \in O\big(\max\{f(n), g(n)\}\big).$$

Beispiel:

```
x=0;                                  /* P1 */
for (i=1; i<=n; i++) { x=x+i; }       /* P2 */
```

Rechenregeln für die Zeitkomplexität II:

- **Produktregel**: Das Programm P entstehe durch Einsetzen von P_1 in der innerensten Schleife von P_2. Die Komplexitäten von P_1 und P_2 seien $T_1(n) \in O(f(n))$ und $T_2(n) \in O(g(n))$. Die Komplexität von P wird berechnet nach der Produktregel

$$T_1(n) \cdot T_2(n) \in O(f(n)g(n)).$$

Beispiel:

```
for (i=1; i<=n; i++) {
  for (j=1; j<=n; j++) {
    x=x+i*j;
  }
}
```

Rechenregeln für die Zeitkomplexität III:

- **Rekursive Prozeduraufrufe**: Produkt aus Anzahl der rekursiven Aufrufe mit Kosten der teuersten Prozedurausführung.
- **Beispiele:**
  1. ```
     int sum(n) {
       if (n<=1) return 1;
       return n+sum(n-1);
     }
     ```
  2. ```
     int sumsum(n) {
       if (n==0) return 1;
       return sum(n)+sumsum(n-1);
     }
     ```

Schätzen sie die Komplexitäten von $\mathtt{sum}(n)$ und $\mathtt{sumsum}(n)$ mit Hilfe der Rechenregeln ab.

Maximale Teilsumme

Tag	1	2	3	4	5	6	7	8	9	10
Δ	+5	-6	+4	+2	-5	+7	-2	-7	+3	+5

```
int maxSubSum( int[] a) {
   int maxSum = 0;
   for( i=0; i<a.length; i++)
      for( j=i; j<a.length; j++) {
        int thisSum =0;
        for (int k = i; k<=j; k++)
           thisSum += a[k];
        if(thisSum>maxSum) maxSum=thisSum;
      }
   return maxSum;
}
```

Maximale Teilsumme: Analyse

- n =`a.length`
- Innerste Schleife `for (k=i; k<=j; k++)`
  $j - i + 1$ mal durchlaufen fuer jedes i, j.
- Mittlere Schleife `for (j=i; j<n; j++)`
  jeweils $j - i + 1$ Aktionen
  $\rightarrow 1 + 2 + 3 + \ldots n - i = (n - i)(n - i + 1)/2$ Aktionen
- äußere Schleife `for (i=0; i<n; i++)`
  aufsummieren ueber den Aufwand des Durchlaufes für jedes i
- Beispiel $n = 32$ 5984 Additionen

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i}^{j} 1 = \sum_{j=i}^{n-1} \sum_{j=i}^{n-1} (j - i + 1) = \sum_{j=i}^{n-1} \sum_{l=1}^{n-i} l \\ &= \sum_{i=0}^{n-1} (n - i)(n - i + 1)/2 = \sum_{k=1}^{n} k(k+2)/2 = n^3/6 + n^2/2 + n/3 \end{aligned}$$

$\sum_{k=1}^{n} k^2 = n^3/3 + n^2/2 + n/6$ (wie sieht man das?)

Beweis . . .

$$\sum_{k=1}^{n} k^2 \quad = \quad n^3/3 + n^2/2 + n/6 \quad = \quad 1/6(n)(n+1)(2n+1)$$

(i) $n = 1$: $\sum_{k=1}^{1} k^2 = 1 = 1/3 + 1/2 + 1/6$

(ii) $n - 1 \to n$:

1. $\sum_{k=1}^{n-1} + n^2 = 1/6(n-1)(n)(2n-1) + 6n^2/6$
2. $= 1/6(2n^3 - n^2 - 2n^2 + n) + 6n^2/6$
3. $= n^3/3 + n^2/2 + n/6$

“Versteckte” Zeitkomplexität

Beispiel: Fibonacci Zahlen $\boxed{0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots}$

Definition $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ für $n > 1$.

1.
```
int fibE (int n) {
  if (n <= 0) return 0;
  else if (n ==1) return 1;
  else return fibE(n-2) + fibE(n-1)
}
```
`fibE(n)` berechnet F_n, aber **exponentieller Aufwand**!

2.
```
int fibL (int n, int f1, int f2) {
  if (n <=0) return 0;
  else if (n==1) return f1+f2;
  else return fibL(n-1,f2+f1,f1)
}
```
`fibL(n,1,0)` berechnet F_n mit **linearem Aufwand**

z.B. `fibL(4,1,0)=fibL(3,1,1)=fibL(2,2,1)=fibL(1,3,2)=5`

"Versteckte" Platzkomplexität am Beispiel $n!$

```
int fakL(int n) {
   if(n<=1) return 1;
   else return n*fakL(n-1);
}
```

`fakL(n)` $\longrightarrow$ $n!$ mit linearem Speicheraufwand $\Omega(n)$

```
int fakK(int n,int a) {
   if(n<=1) return a;
   else return fakK(n-1,n*a);
}
```

`fakK(n,1)` $\longrightarrow$ $n!$ mit konstantem Speicheraufwand $\Omega(1)$

Bessere Abschätzung: Wie waechst der Aufwand bei der Multiplikation grosser Zahlen?

Was ist der Speicheraufwand für n und für $n!$?

Multiplikation für Fortgeschrittene

Wie in der Schule:

```
5432*1995
5432
48888
 48888
  27160
-------
10836840
```

$\Longrightarrow O(\ell^2)$ für ℓ-stellige Zahlen.

Besser:

$(100A + B) \times (100C + D) = 10000AC + 100(AD + BC) + BD$

Nach $AD + BC = (A + B) \times (C + D) - AC - BD$, nur noch 3 Multiplikationen von Zahlen der halben Länge:

$$T(n) = 3T(n/2),$$

wobei der Aufwand für die Addition vernachlässigt wird.
Lösung: $T(n) = n^{\log_2 3} \approx n^{1.585} \ll n^2$

Für *große* Zahlen geht's also intelligenter als in der Schule!

Noch ein paar Beispiele

- Exponentielles Wachstum:
  $T(n+1) = aT(n)$, daher ...
  $T(n) = aT(n-1) = a^2 T(n-2) = a^k T(n-k)$
  $= a^n T(0) = 2^{\log_2 an}$
  "Skalengröße" $k = \log_2 a$

- Fibonacci-Zahlen: *Binet*sche Formel

$$F_n = (A^n - B^n)/\sqrt{5}$$

  mit $A = (1+\sqrt{5})/2$ und $B = (1-\sqrt{5})/2$.
  Umschreiben: $F_n = (1/\sqrt{5})A^n[1-(B/A)^n]$
  Für große n: $(B/A)^n = (-1)^n[(\sqrt{5}-1)/(\sqrt{5}+1)]^n \to 0$
  Daher $F_n < cA^n$ für hinreichend große n, d.h.
  $F_n \in O(A^n)$. Genaugenommen sogar $F_n \in \Theta(A^n)$

Das Mastertheorem

- Allgemeines Theorem zur Lösung von Funktionalgleichungen (Rekursionsgleichungen) der Form

$$T(n) = aT\left(\frac{n}{b}\right) + g(n),\ a \geq 1, b > 1$$

Warum können wir i.d.R. $T(1) = 1$ annehmen?

- Funktionalgleichung beschreibt *algorithmische Strategie* "Divide-and-Conquer":
  - Zerlege Gesamtproblem in b gleich grosse Teilprobleme.
  - Für Gesamtproblem löse a solche Teilprobleme.
  - Für Zerlegung und Kombination der Teillösungen entstehen jeweils Overhead-Kosten $g(n)$.

Das Mastertheorem — Polynomialer Overhead

- Es gibt mehrere Lösungen je nach Verhalten von $g(n)$.
- Sei jetzt $\underline{g(n) \text{ polynomial}}$, d.h. $g(n) = \Theta(n^k)$:

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k),\ a \geq 1, b > 1$$

- Dann unterscheide 3 Fälle:

$$T(n) = \begin{cases} \Theta(n^k) & \text{falls } a < b^k \\ \Theta(n^k \log n) & \text{falls } a = b^k \\ \Theta(n^{\log_b(a)}) & \text{falls } a > b^k \end{cases}$$

Das Mastertheorem — Beispiele

Sei wieder $g(n)$ polynomial, $g(n) = \Theta(n^k)$.

Setze $k = 2$ und $b = 3$ und betrachte Beispiele für $a \left\{ \begin{smallmatrix} < \\ = \\ > \end{smallmatrix} \right\} n^k$.

$$a = 8 \quad T(n) = 8T\left(\tfrac{n}{3}\right) + \Theta(n^2) \quad \Rightarrow \quad T(n) = \Theta(n^2)$$

$$a = 9 \quad T(n) = 9T\left(\tfrac{n}{3}\right) + \Theta(n^2) \quad \Rightarrow \quad T(n) = \Theta(n^2 \log_2 n)$$

$$a = 10 \quad T(n) = 10T\left(\tfrac{n}{3}\right) + \Theta(n^2) \quad \Rightarrow \quad T(n) = \Theta(n^{\log_3 10})$$

$$\boxed{\text{Zur Erinnerung: } T(n) = \begin{cases} \Theta(n^k) & \text{falls } a < b^k \\ \Theta(n^k \log n) & \text{falls } a = b^k \\ \Theta(n^{\log_b(a)}) & \text{falls } a > b^k \end{cases}}$$

Das Mastertheorem — allgemein

Setze $u := log_b(a)$. (Idee: die Fälle scheiden sich an n^u)

Im Fall ohne Overhead-Kosten, d.h. $g(n) = 0$, gilt $T(n) = \Theta(n^u)$.

(vgl. Analyse des verbesserten Multiplikationsverfahrens / Übung).

Allgemeine Lösung

- $T(n) = \Theta(n^u)$, falls $g(n) = O(n^{u-\epsilon})$ für ein $\epsilon > 0$
- $T(n) = \Theta(n^u \log n)$, falls $g(n) = \Theta(n^u)$
- $T(n) = \Theta(g(n))$, falls $g(n) = \Omega(n^{u+\epsilon})$ und $ag(\frac{n}{b}) \leq cg(n)$ für ein $\epsilon > 0$

Zusammenfassung

- Komplexität / Effizienz
  = wesentliche Eigenschaft von Algorithmen
- meist asymptotische Worst-Case-Abschätzung in Bezug auf Problemgröße n
  - Unabhängigkeit von konkreten Umgebungsparametern (Hardware, Betriebsystem, ...)
  - asymptotisch “schlechte” Verfahren können bei kleiner Problemgröße ausreichen
- wichtige Klassen: $O(1)$, $O(\log n)$, $O(n)$, $O(nlogn)$, $O(n^2)$, $\ldots$, $O(2^n)$
- zu gegebener Problemstellung gibt es oft Algorithmen mit stark unterschiedlicher Komplexität
  - unterschiedliche Lösungsstrategien
  - Raum/Zeit-“Tradeoff”: Zwischenspeichern von Ergebnissen statt mehrfacher Berechnung
- Abschätzung der Komplexität aus Programmfragmenten nach Rechenregeln