

ADS: Algorithmen und Datenstrukturen

Akuter Denk-Stau

Peter F. Stadler & Konstantin Klemm

Bioinformatics Group, Dept. of Computer Science & Interdisciplinary Center for
Bioinformatics, **University of Leipzig**

29. Oktober 2009

Ankündigungen im Netz

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

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

→ Teaching → Current classes → Algorithmen und Datenstrukturen

Übungsaufgaben

- Abzugeben sind Lösungen zu fünf Aufgabenblättern.
- Abgabetermine: 28.10., 11.11., 25.11., 09.12., 06.01.
- Lösungen sind direkt *vor* Beginn der Vorlesung im Hörsaal abzugeben.
- Lösungen werden bewertet und in der Übungsstunde zurückgegeben.

Übungstermine

Vorläufige Termine der Übungsgruppen

- Dienstag 17:15 – 18:45 Uhr
- Mittwoch 09:15 – 10:45 Uhr
- Mittwoch 13:15 – 14:45 Uhr
- Donnerstag 13:15 – 14:15 Uhr
- Donnerstag 15:15 – 16:45 Uhr

Eventuell zusätzlich

- Montag 11:15-12:45
- Freitag Nachmittag

Übungen — Anmeldung

Anmeldung für die Übungen

Donnerstag (15.10.), 12:00 Uhr — Dienstag (20.10.), 12:00 Uhr

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

→ Teaching → Current classes → Algorithmen und Datenstrukturen

Klausur

Termin (voraussichtlich): 03.02.2010, 17:15 – 18:15 Uhr

Zulassungsvoraussetzungen

- Erreichen von 50% der Punkte in den Übungsaufgaben
- Anwesenheit in der Übung
- Fähigkeit, abgegebene Lösungen an der Tafel zu erläutern

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

→ Teaching → Current classes → Algorithmen und Datenstrukturen

Inhalt

- 1 Einführung: Typen von Algorithmen, Komplexität von Algorithmen
- 2 Einfache Suchverfahren in Listen
- 3 Verkettete Listen, Stacks und Schlangen
- 4 Sortierverfahren
 - Elementare Verfahren
 - Shell-Sort, Heap-Sort, Quick-Sort
 - Externe Sortierverfahren
- 5 Allgemeine Bäume und Binärbäume
 - Orientierte und geordnete Bäume
 - Binärbäume (Darstellung, Traversierung)
- 6 Binäre Suchbäume
- 7 Mehrwegbäume

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 (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

- 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 (mittels eines Index) zu suchen.
- (Technische Nebenbemerkung: ausserdem ist der Speicherzugriff auf der CD sehr langsam)

Beispiel Telefon-CD

- Design-Überlegungen:
Wir könnten eine Datenstruktur verwenden, die die vollständige Verwaltung der Einträge erlaubt: Suchen, Einfügen und Löschen.
- Weil sowieso nur das Suchen erlaubt ist, können wir vielleicht eine Datenstruktur verwenden, die zwar extrem schnelles Suchen in einer komprimierten Datei erlaubt, aber möglicherweise kein Einfügen.
- **Also:** Mitdenken kann helfen

Effizienz: Zeit und Speicher

- Die Abarbeitung von Programmen (Software) beansprucht 2 Ressourcen:
Zeit und Hardware (wichtig: 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 RAM und externem Speicher, da der Zugriff auf eine Festplatten ca. 100.000 mal langsamer ist als ein RAM-Zugriff. Deshalb werden manche Algorithmen bei Überschreiten des RAM so langsam, dass sie praktisch nutzlos sind.

Komplexität von Algorithmen

- **Wesentliche Maße:**
 - Rechenzeitbedarf (Zeitkomplexität)
 - Speicherplatzbedarf (Speicherplatzkomplexität)
- **Programmlaufzeit von zahlreichen Faktoren abhängig**
 - Eingabe für das Programm
 - Qualität des vom Compiler generierten Codes und des gebundenen Objektprogramms
 - Leistungsfähigkeit der Maschineninstruktionen, mit deren Hilfe das Programm ausgeführt wird
 - Zeitkomplexität des Algorithmus, der durch das ausgeführte Programm verkörpert wird
- **Bestimmung der Komplexität**
 - Messungen auf einer bestimmten Maschine
 - Aufwandsbestimmungen für idealisierten Modellrechner (Bsp.: Random-Access-Maschine oder RAM)
 - Abstraktes Komplexitätsmaß zur asymptotischen Kostenschätzung in Abhängigkeit zur Problemgröße (Eingabegröße) n

Asymptotische Kostenmaße

Festlegung der Größenordnung der Komplexität in Abhängigkeit der Eingabegröße:

Best Case, Worst Case, Average Case

Meist Abschätzung oberer Schranken (Worst Case):

Gross-Oh-Notation

Zeitkomplexität $T(n)$ eines Algorithmus ist von der Größenordnung n , wenn es Konstanten n_0 und $c > 0$ gibt, so daß für alle Werte von $n > n_0$ gilt:

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

man sagt " $T(n)$ ist in $O(n)$ ", in Zeichen $T(n) \in O(n)$ oder einfach $T(n) = O(n)$.

Allgemeine Definition

Jetzt wird's ernst

Die Klasse $O(f)$ der Funktionen von der Grössenordnung f is

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

also eigentlich ganz einfach

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

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

Beweis: Für $n \geq n$ gilt: $n^4 \geq n^3 \geq n^2 \geq n \geq \log n$ und $n^4 \geq 1 \geq \sin(\text{irgendwas})$. Also ist

$(6 + 3 + 7 + 42 + 1)n^4$ jedenfalls grösser als $T(n)$.

Damit hätten wir die geforderten konstanten gefunden:

$n_0 = 1$ und $c = 59$.

Alles klar?

$g(n) \in O(n)$ impliziert $g(n) \in O(n \log n)$ impliziert $g(n) \in O(n^2)$
(warum?)

Untere Schranken

$g \in \Omega(f)$ meint, dass g mindestens so stark wächst wie f (untere Schranke).

Definition:

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

Bemerkung: alternativ kann man definieren:

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

Exakte Schranke

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 für hinreichend große n im Bereich $[c_1f, c_2f]$ mit geeigneten Konstanten c_1 und c_2 .

Ist $T(n)$ ein Polynom vom Grad p dann ist $T(n) \in \Theta(n^p)$

Wachstumsordnung = höchste Potenz

BEWEIS → TAFEL

Wichtige Wachstumsfunktionen

$O(1)$ konstante Kosten

$O(\log n)$ 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/Faktorielle

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$$

HAUSÜBUNG und zum NACHDENKEN: Rechnen Sie das für $n = 10, 100, 1000, 10000, 100000, 1000000$ mal aus. Was passt noch auf ihren Laptop?

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 Rechengeschwindigkeit, $T_K(n)$ bei k -facher Geschwindigkeit

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

Alternativ:

Rechner #2 löst Problem der Größe n_K

$$T(n) = T_K(n_K) = KT(n_K)$$

$$\text{Lösung: } n_k = T^{-1}(K(T(n)))$$

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

Zeitkomplexitätsklassen

- linear-zeitbeschränkt, $T \in O(n)$
- polynomial-zeitbeschränkt, $T \in O(n^K)$
- exponentiell-zeitbeschränkt, $T \in O(\alpha^n)$

Exponentiell-zeitbeschränkte Algorithmen im Allgemeinen (große n) nicht nutzbar. Probleme, für die kein polynomial-zeitbeschränkter Algorithmus existiert, gelten als unlösbar (intractable).

Berechnung der Zeitkomplexität I:

- Elementare Operationen (Zuweisungen, Ein/Ausgabe) $O(1)$
- **Summenregel:**
 T_1 und T_2 seien die Laufzeiten zweier Programmfragmente P_1 und P_2 ; es gelte $T_1(n) \in O(f(n))$ und $T_2(n) \in O(g(n))$.

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

- **Produktregel:** für geschachtelte Schleifenausführung von P_1 und P_2 :

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

- **Fallunterscheidungen** $O(1)$
- **Schleife** Produkt aus Anzahl der Schleifendurchläufe mit Kosten der teuersten Schleifenausführung
- **rekursive Prozeduraufrufe:** Produkt aus Anzahl der rekursiven Aufrufe mit Kosten der teuersten Prozedurausführung

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

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

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

Rekursion vs. Iteration I

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

```
Int fibRekursiv (int n) {
    if (n <=0) return 0;
    else if (n ==1) return 1;
    else return fibRekursiv (n-2) +
        fibRekursiv (n-1)
}
```

Exponentieller Aufwand!

```
Int fibIterativ (int n) {
    if (n <=0) return 0;
    else {
        int aktuelle=1, vorherige=0, temp=1,
        for (int i=1; i<n; i++) {
            temp = aktuelle;
            aktuelle += vorherige;
            vorherige = temp;}
        return aktuelle;
    }
}
```

Linearer Aufwand

Rekursion vs. iteration: $n!$

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

Speicheraufwand $\Omega(n)$

```
int fakIterativ(int n) {  
    int fak=1;  
    for (int i=2;i<=n;i++) fak *= i;  
    return fak;  
}
```

Speicheraufwand $\Omega(1)$

Bessere Abschätzung: Wie wächst der Aufwand mit der bei der Multiplikation großer Zahlen?

Einfach zum Nachdenken:

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

Multiplikation für Fortgeschrittene

Wie in der Schule:

```

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

```

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

Besser:

$$\begin{aligned}
 (100A + B) \times (100C + D) &= 10000AC + 100(AD + BC) + BD \\
 &= 10000AC + 100((A + B)(C + D) - AC - BD) + BD
 \end{aligned}$$

nur 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 ...

$$T(n) = aT(n-1) = a^2T(n-2) = a^kT(n-k) = a^nT(0) = 2^{\log_2 an}$$

“Skalengröße” $k = \log_2 a$

- Fibonacci Zahlen:

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

$A = (1 + \sqrt{5})/2$ und $B = (1 - \sqrt{5})/2$.

umschreiben: $F_n = (1/\sqrt{5})A^n[1 - (B/A)^n]$

$(B/A)^n = (-1)^n[(\sqrt{5} - 1)/(\sqrt{5} + 1)]^n \rightarrow 0$ für große n

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

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

- Funktionalgleichung beschreibt *algorithmische Strategie*: Zerlege Problem der Größe n in b Teilprobleme.
- Lösung des Gesamtproblems koste das a -fache der Lösung eines Teilproblems
- zusätzlich entstehen einmalige overhead Kosten $g(n)$

Das Mastertheorem — Polynomial

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

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

- Dann ist

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

Das Mastertheorem — Beispiele

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

Betrachte den Fall $k = 2$ und $b = 3$:

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

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

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

Das Mastertheorem — allgemein

Setze $u = \log_b(a)$.

Zur Erinnerung: Falls $g(n) = 0$ dann ist $T(n) = \Theta(n^u)$, vgl.

Analyse des verbesserten Multiplikationsverfahrens. Dieses Gesetz markiert den Fall ohne Overhead-Kosten.

Allgemeine Lösung

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

Zusammenfassung

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