Algorithmen und Datenstrukturen

SoSe 2008 in Trier

Henning Fernau
Universität Trier
fernau@uni-trier.de

Algorithmen und Datenstrukturen

Gesamtübersicht

- Organisatorisches / Einführung
- Grundlagen: RAM, $\mathcal{O}$-Notation, Rekursion, Datenstrukturen
- Sortieren
- Wörterbücher und Mengen
- Graphen und Graphalgorithmen

Organisatorisches

Vorlesungen MI 8-10 im HS 11/12; FR 12-14 im HS 13

Übungsbetrieb Die Übungen werden wöchentlich besprochen.
BEGINN: in der zweiten Semesterwoche DO 16-18, HS 12

Dozentensprechstunde DO, 13-14 in meinem Büro H 410 (4. Stock)

Mitarbeitersprechstunde Daniel Schmitt, MO 14-16, H429

Tutorensprechstunde
Sebastian Schlecht, MO 12.45 - 13.45, H407
Alexander Woskobojnik, FR 14 - 15, H447

Teilnahme an der Abschlussklausur

Termin Abschlussklausur: Letzte Semesterwoche oder erste Woche in der vorlesungsfreien Zeit
GENAUER: MI, 9.7., vermutlich nachmittags

Nachklausur zu Begin des nächsten Semesters.

Genaueres wird noch bekanntgegeben.

Sollten Sie an einer der Klausuren nicht teilnehmen können, so legen Sie bitte ein ärztliches Attest vor; andernfalls wird die Klausur mit 0 Punkten bewertet.

Aufwandanalyse

meistens bei uns: *Zeitaufwand*
manchmal auch von Interesse:
Speicherverbrauch
Anzahl Prozessoren
Energieverbrauch
...

Aufwandanalyse Was wollen wir genau abschätzen ?

Meist: Abschätzung des *schlimmsten Falles* (worst case analysis)

Das bedeutet: Versuche, den Aufwand, den Eingaben der Länge n schlimmstenfalls verursachen können, "möglichst genau" durch eine Funktion $f(n)$ anzugeben bzw. nach oben abzuschätzen.

Für die Praxis oft eigentlich interessanter, aber weit schwieriger zu besorgen: Abschätzung des *mittleren Falles* (average case analysis)

Hierzu nötig: Kenntnisse über die Häufigkeiten, dann interpretiert als Wahrscheinlichkeiten, von Vorkommen in der Eingabe der Länge n, also der *Eingabeverteilung*.

Warnung: Annahme von Gleichverteilung ist oft verkehrt.

Selten praktisch, aber für untere Schranken theoretisch wertvoll: Abschätzung des *besten Falles*

Ein Beispiel aus VL1 Lineare Suche nach Wert `x` im Array `A[0,n-1]`:

```
i = 0;
while ((i < n) && (A[i] != x))
   i++;
```

Wieviele “Schritte” führt der Algorithmus im schlimmsten Falle aus ?
Eingangs erfolgt eine Zuweisung.
Jedesmal, wenn die Schleife betreten werden soll, werden zwei Tests durchgeführt und die (Booleschen) Ergebnisse per Konjunktion verknüpft.
Mit dem nachfolgenden Inkrement (im Schleifenrumpf) werden “im negativen Fall” (`x` wird nicht gefunden) je Schleifeniteration vier “Operationen” durchgeführt.
Im schlimmsten Fall wird `x` nicht gefunden, es werden also `n` (vergebliche) Schleifendurchläufe durchgeführt sowie noch ein abschließender negativer Test der ersten Bedingung `(i < n)`.
Insgesamt sind das also `4n+2` Operationen.

Aufgabe: Geht es “besser” ?

Ein Beispiel aus VL1 Lineare Suche nach Wert `x` im Array `A[0,n-1]`:
Wir nehmen jetzt an, es gäbe noch das Element `A[n]` in der Deklaration des Feldes, dieses wird aber nur als Hilfselement genutzt:

```
i = 0; A[n] = x;
while (A[i] != x)
   i++;
```

Eine Analyse wie im vorigen Fall liefert:
Im schlimmsten Fall wird `x` nicht gefunden, es werden also `n` (vergebliche) Schleifendurchläufe durchgeführt und somit insgesamt `2n+3` Operationen ausgeführt.
Beim Verlassen der Schleife kann man wieder an der Laufvariablen erkennen, ob das Element im "eigentlichen Feld" `A[0,n-1]` enthalten ist oder nicht.
Wir haben bei dieser Version übrigens auch keine Probleme mehr mit der Semantik des logischen UND !
In der Praxis sind solche Verbesserungen um den Faktor 2 interessant !

Vom Vernachlässigen der Konstanten oder
Wie vergleicht man Laufzeiten verschiedener Algorithmen ?

Absolute Laufzeiten in Millisekunden sind stark abhängig von Hardware, Betriebssystem, Programmiersprache, Compiler, gewählten Beispielen, ...
Es ist oft nicht klar, was als Schritt gezählt werden soll: Bei bestimmten Aufgaben (wie z.B. der Suche) werden oft nur die Vergleiche gezählt (s.u.).

Lässt sich die Laufzeit von Algorithmus A_i $(i = 1, 2)$ zur Lösung desselben Problems für Eingaben der Größe n abschätzen durch $T_i(n)$, und ist $T_i(n) = c_i n^i$, $c_i \geq 1$, so ist bei genügend großen Eingaben A_1 irgendwann besser als A_2, unabhängig von den Konstanten c_i.
$\rightsquigarrow$ Versuchen wir, Wachstum von Funktionen unabhängig von multiplikativen und additiven Konstanten auszudrücken.

$\mathcal{O}$-Notation

Definition (Paul Bachmann 1894, später von Edmund Landau popularisiert):
Es seien $f, g \in \mathbb{R}^{\mathbb{N}}$. $g(n) = \mathcal{O}(f(n))$ für $n \in \mathbb{N}$, falls $\exists M, n_0 \in \mathbb{N}$, so dass gilt:
$\forall n \geq n_0 : |g(n)| \leq |Mf(n)|$

Hinweis: Ungenauer (asymmetrischer) Gebrauch von "$=$"; genauer ist $\mathcal{O}(f)$ als Menge von Funktionen aufzufassen, dann müsste man $g \in \mathcal{O}(f)$ schreiben. Diese Sicht nehmen wir oft ein.

D.E. Knuth liest $\mathcal{O}$ (wie Bachmann) als "Groß-Omikron"; weitere Schreibweisen:
$g \in \Omega(f)$ gdw. $f \in \mathcal{O}(g)$ (definiert wiederum Funktionenmenge)
$g \in \Theta(f)$ gdw. $g \in \mathcal{O}(f) \cap \Omega(f)$.

Einige Beispiele

$\mathcal{O}(1)$:	konstanter Aufwand, unabhängig von n
$\mathcal{O}(n)$:	linearer Aufwand (z.B. Einlesen von n Zahlen)
$\mathcal{O}(n \ln n)$:	Aufwand guter Sortierverfahren (z.B. Quicksort)
$\mathcal{O}(n^2)$:	quadratischer Aufwand
$\mathcal{O}(n^k)$:	polynomialer Aufwand (bei festem k)
$\mathcal{O}(2^n)$:	exponentieller Aufwand
$\mathcal{O}(n!)$:	Bestimmung aller Permutationen von n Elementen

$H_m = \sum_{i=1}^{m} \frac{1}{i}$ *m-te harmonische Zahl*

Möglicherweise bekannt aus Analysis: Die Folge H_m konvergiert nicht gegen eine konstante Zahl, sondern sie wächst über alle Maßen.
Frage: Wie "schnell" wächst H_m mit m ?
Antwort: "Etwa so wie der Logarithmus zur Basis zwei."

Satz: $\forall n(1 + \frac{n}{2} \leq H_{2^n} \leq 1 + n)$, also $H_m \in \Theta(\log(m))$ (Basis egal).

Beweis: IA: klar für $n = 0$, denn $H_1 = 1$.

IS (für erste Ungleichung): Angenommen, die Aussage gilt für n. Dann rechnen wir:

$$
\begin{aligned}
H_{2^{n+1}} &= \sum_{i=1}^{2^{n+1}} \frac{1}{i} \\
&= \sum_{i=1}^{2^n} \frac{1}{i} + \sum_{i=2^n+1}^{2^{n+1}} \frac{1}{i} \\
&= H_{2^n} + \sum_{i=2^n+1}^{2^{n+1}} \frac{1}{i} \\
&\geq 1 + \frac{n}{2} + 2^n \cdot \frac{1}{2^{n+1}} \qquad \text{IV und einfache Absch.} \\
&= 1 + \frac{n}{2} + \frac{1}{2} \\
&= 1 + \frac{n+1}{2}
\end{aligned}
$$

Entsprechend sieht man für die zweite Ungleichung:

$$\begin{aligned} H_{2^{n+1}} &= \cdots = H_{2^n} + \sum_{i=2^n+1}^{2^{n+1}} \frac{1}{i} \\ &\leq 1 + n + 2^n \cdot \frac{1}{2^n} \qquad \text{IV und einfache Absch.} \\ &= 1 + n + 1 \\ &= 1 + (n+1) \end{aligned}$$

Nach dem Prinzip der mathematischen Induktion folgt die Behauptung.

Konkreter Vergleich

Annahme: 1 Schritt dauert 1 μs = 0.000001 s

$n =$	10	20	30	40	50	60
n	10 μs	20 μs	30 μs	40 μs	50 μs	60 μs
n^2	100 μs	400 μs	900 μs	1.6 ms	2.5 ms	3.6 ms
n^3	1 ms	8 ms	27 ms	64 ms	125 ms	216 ms
2^n	1 ms	1 s	18 min	13 Tage	36 J	366 Jh
3^n	59 ms	58 min	6.5 J	3855 Jh	10^8 Jh	10^{13} Jh
$n!$	3.62 s	771 Jh	10^{16} Jh	10^{32} Jh	10^{49} Jh	10^{66} Jh

Anwendung in der Algorithmik: Laufzeitanalyse

Verschiedene Analysen sind von Interesse:

- bester Fall (best case)
- mittlerer Fall (average case)
- schlimmster Fall (worst case)

Für den eigentlich meist interessantesten mittleren Fall wären auch noch Annahmen über die zu erwartende Eingabeverteilung sinnvoll.
Meistens beschränkt man sich bei der Analyse auf den schlimmsten Fall.

Anwendung in der Algorithmik: Laufzeitanalyse von for-Schleifen (Beispiele)

Minimumsuche in einem Array der Länge n

```
min = a[0];
for (i = 1; i < n; i++)
       if(a[i] < min) min = a[i];
```

n "Schritte" für n Daten $\rightsquigarrow$ Laufzeit $\Theta(n)$

```
for (i = 0; i < k; i++)
   for (j = 0; j < k; j++)
      brett[i][j] = 0;
```

k^2 Schritte für k^2 Daten

$\rightsquigarrow$ linearer Algorithmus

```
for (i = 0, i < k; i++)
   for (j = 0; j < k; j++)
       if (a[i] == a[j]) treffer = true;
```

k^2 Schritte für k Daten

$\rightsquigarrow$ quadratischer Algorithmus

Anwendung in der Algorithmik: Laufzeitanalyse von while-Schleifen (Beispiele)

Lineare Suche im Array

```
i = 0;
while (i < n) && (a[i] != x)
   i++;
```

Laufzeit:	bestenfalls	1 Schritt	$\Rightarrow$	$\Theta(1)$
	schlimmstenfalls	n Schritte	$\Rightarrow$	$\Theta(n)$
	im Mittel	$\frac{n}{2}$ Schritte	$\Rightarrow$	$\Theta(n)$

Annahme für den mittleren Fall: Es liegt (gleichwahrscheinliche) Permutation der Zahlen von 1 bis n vor. Dann ist die mittlere Anzahl

$$= \frac{1}{n} \sum_{i=1}^{n} i \approx \frac{n}{2}$$

Erinnerung: **Geometrische Verteilung**

Wir werfen wiederum wiederholt mit einer Münze, die mit Wahrscheinlichkeit p "Kopf" zeigt.

Wie oft muss man werfen, bis das erst Mal "Kopf" erscheint ?

X: ZV, die die Anzahl der nötigen Würfe beschreibt.
Definitionsbereich von X: Menge der endlichen Folgen von Münzwürfen.
$X(e)$ ist dann der Index der ersten Stelle, die "Kopf" ist.
Beispiel: $X((\text{Zahl}, \text{Zahl}, \text{Zahl}, \text{Kopf}, \text{Zahl}, \text{Kopf})) = 4$.
Wertebereich von X: Menge der positiven ganzen Zahlen.

$P[X = k] = (1-p)^{k-1}p$
$P[X = \cdot]$ ist Wahrscheinlichkeitsdichte wegen geometrischer Reihe:
$\sum_{k=1}^{\infty} P[X = k] = \sum_{k=1}^{\infty}(1-p)^{k-1}p = p \cdot \frac{1}{1-(1-p)} = 1.$

$$E[X] = \sum_{k=1}^{\infty} k \cdot (1-p)^{k-1}p = \frac{p}{1-p}\sum_{k=0}^{\infty} k(1-p)^k = \frac{p}{1-p} \cdot \frac{1-p}{p^2} = \frac{1}{p}$$

Anwendung in der Algorithmik: Laufzeitanalyse von while-Schleifen (Beispiele)

Lineare Suche in 0/1-Array (also $x, a[i] \in \{0, 1\}$)

```
i = 0;
while (i < n) && (a[i] != x)
   i++;
```

Bester und schlimmster Fall wie bisher.
Mittlerer Fall bei Annahme einer "Erfolgswahrscheinlichkeit" pro Stelle von 1/2 und der weiteren (in der Regel fälschlichen) Annahme der Unabhängigkeit aufeinanderfolgender "Experimente"
$\leadsto$ Erwartungswert der geometrischen Verteilung ist 2 (unabhängig von n)
$\leadsto \Theta(1)$ im mittlereren Fall
Gilt analog für beliebige "endliche Alphabetgrößen"

o, ω-Notation: Angabe vernachlässigbarer Terme

Definition (Paul Bachmann 1894, später von Edmund Landau popularisiert):
Es seien $f, g \in \mathbb{R}^{\mathbb{N}}$. $g(n) = o(f(n))$ für $n \in \mathbb{N}$, falls $\forall c > 0 \exists n_0 \in \mathbb{N}$, so dass gilt:
$\forall n \geq n_0 : |g(n)| < |cf(n)|$

D.E. Knuth liest o als "Klein-Omikron"; weitere Schreibweisen:
$g \in \omega(f)$ gdw. $f \in o(g)$ (definiert wiederum Funktionenmenge)
Frage: Wie lautet die "entsprechende Aussage" für ω zu folgendem Satz ?

Satz:

$$g \in o(f) \iff \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

Satz:

$$g \in o(f) \iff \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

Beweis: Wir betrachten vereinfachend den Fall, dass f und g nur nichtnegative Werte annehmen. Sei $g \in o(f)$. Wäre $\lim_{n\to\infty} \frac{g(n)}{f(n)} = d > 0$, dann gölte:

$$\forall\epsilon\exists n_0\forall n \geq n_0 : \frac{g(n)}{f(n)} \in (d-\epsilon, d+\epsilon).$$

Mit $\epsilon = c = d/2$ gilt also:

$$\forall n \neq n_0 : g(n) > cf(n),$$

im Widerspruch zur Def. von $o(f)$.

Umgekehrt folgt aus $g \notin o(f)$ die Existenz eines $c > 0$, sodass für manche beliebig groß wählbare n gilt: $g(n) \geq cf(n)$. Daher kann der fragliche Grenzwert nicht verschwinden.

Frage: Wieso existiert der Grenzwert aus dem ersten Beweisteil überhaupt ?

Weitere Eigenschaften

Satz: (Transitivität) Für $\mathrm{op} \in \{\mathrm{o}, \mathcal{O}, \omega, \Omega, \Theta\}$ gilt:
Mit $\mathrm{f} \in \mathrm{op}(\mathrm{g})$ und $\mathrm{g} \in \mathrm{op}(\mathrm{h})$ folgt: $\mathrm{f} \in \mathrm{op}(\mathrm{h})$.

Satz: (Reflexivität) Für $\mathrm{op} \in \{\mathcal{O}, \Omega, \Theta\}$ gilt: $\mathrm{f} \in \mathrm{op}(\mathrm{f})$.

Satz: (Symmetrie) $\mathrm{f} = \Theta(\mathrm{g})$ gdw. $\mathrm{g} = \Theta(\mathrm{f})$.

Weitere Schreibweisen

Das “Gleichheitszeichen” wird mit den “Landau-Symbolen” streng von links nach rechts gedeutet.
So meint man mit:

$$t(n) = s(n) + \Theta(g(n))$$

Es gibt eine Funktion $f \in \Theta(g)$, sodass $t(n) = s(n) + f(n)$.
Ferner bedeutet:

$$t(n) + \Theta(g_1(n)) = s(n) + \Theta(g_2(n)) :$$

Zu jeder Funktion $f_1 \in \Theta(g_1)$ findet sich eine Funktion $f_2 \in \Theta(g_2)$, sodass $t(n) + f_1(n) = s(n) + f_2(n)$ gilt.

Amortisierte Analyse

Hinweis: Mehr Beweise dazu im Skript von Rudolf Fleischer.

Oft wird der schlechteste Fall eines komplizierteren Algorithmus abgeschätzt durch die schlechtesten Fälle der Teilschritte; dies vermeidet die *Aggregat-Methode*, eine Möglichkeit der *amortisierten Analyse*.

Beispiel: Wir wollen zählen (abschätzen), wie viele Bitwechsel beim Inkrementieren eines k-Bit-Zählers von 0 bis $(2^k - 1)$ entstehen. Eine einfache Analyse, ausgehend vom schlimmsten Fall eines einzelnen Inkrements, liefert $\mathcal{O}(k \cdot 2^k)$.
Geht es besser ?

Betrachten wir die Anzahl der Bitwechsel bei einem Zähler mit 3 Bit:

Zähler	0000	0001	0010	0011	0100	0101	0110	0111	1000
Anzahl Bitwechsel	0	1	2	1	3	1	2	1	4

Beobachte: Das niedrigste Bit ändert sich bei jeder Inkrementation, das nächst höhere bei jeder zweiten, das wiederum nächst höhere bei jeder vierten usw. Damit ergibt sich bei n Inkrementationen folgende Summe von Bitwechseln:

$$n + \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{2^2} \right\rfloor + \left\lfloor \frac{n}{2^3} \right\rfloor + \cdots + \left\lfloor \frac{n}{2^k} \right\rfloor \leq n \sum_{i=0}^{k} \frac{1}{2^i}$$

Diese Summe können wir nach oben abschätzen:

$$n \sum_{i=0}^{k} \frac{1}{2^i} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} \leq 2n$$

Ein einzelnes Inkrement kostet daher amortisiert 2 Bitwechsel, nicht etwa k wie zunächst abgeschätzt.