UNIVERSITÄT KONSTANZ
LEHRSTUHL FÜR PRAKTISCHE INFORMATIK
Prof. Dr. D. Wagner / Annegret Liebers

September 1998

Grundlagen: Algorithmen und ihre Laufzeit

Als ergänzende Literatur ist das Buch [CLR94] empfehlenswert.

1 Der Algorithmusbegriff

Zu diesem Abschnitt siehe insbesondere [CLR94, Kapitel 1].

1.1 Erstes Beispiel: das „Element-von"-Problem

Input: Eine Menge M von n Zahlen $a_1, \ldots, a_n$ und eine zusätzliche Zahl q.

Output: Aussage „ $q \in M$" oder Aussage „ $q \notin M$".

Algorithmus anschaulich: Vergleiche jedes Element a_i mit q. Wenn mindestens einmal Gleichheit auftritt, gib aus „ $q \in M$", ansonsten gib aus „ $q \notin M$".

Algorithmus formalisierter (*Pseudocode* — so, daß man ihn z.B. in einer gängigen Programmiersprache implementieren kann):

1. Setze die Boole'sche Variable $b :=$ false.
   (Wir werden b jeweils so setzen, daß man am Inhalt von b ablesen kann, ob q bereits gefunden wurde oder nicht.)
2. $i := 1$
3. Solange $i \leq n$ und solange $b =$ false:
4.   Falls $a_i = q$:
5. $b :=$ true
6.   Gib aus „ $q \in M$".
7.   ansonsten:
8. $i := i + 1$
9. Falls $i = n + 1$:
10.   Gib aus „ $q \notin M$".

1.2 Zweites Beispiel: modifiziertes „Element-von"-Problem

Input: Eine sortierte Folge F von $n \geq 1$ Zahlen $a_1 \leq a_2 \leq \cdots \leq a_n$ und eine zusätzliche Zahl q.

Output: Aussage „ $q \in F$" oder Aussage „ $q \notin F$".

Algorithmus anschaulich: Vergleiche q mit dem mittleren Element der Folge. Ist q größer als dieses Element, kann q höchstens noch in der „rechten" Hälfte der Folge gefunden werden, betrachte in diesem Fall also von nun an nur noch die rechte Hälfte der Folge. Ist q kleiner oder gleich als das mittlere Element, betrachte analog nur noch die linke Hälfte der Folge. Iteriere. (Diese Vorgehensweise heißt *binäre Suche*.)

Beobachtung: Die zusätzliche Eigenschaft, daß die gegebenen n Zahlen sortiert sind, erlaubt ein geschickteres Vorgehen: Im Algorithmus zum ersten Beispiel wurde im ungünstigsten Fall jedes Element der Menge M mit q verglichen, insbesondere dann, wenn q nicht in M vorkam. Nun vergleichen wir nur noch einen Bruchteil der Elemente der Folge F mit q, auch dann, wenn q nicht in F vorkommt.

Algorithmus in Pseudocode:

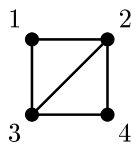
```
l := 1
r := n
Solange r ≠ l:
    k := ⌊(r+l)/2⌋
    Falls q > a_k
        l := k+1
    ansonsten
        r := k
Falls q = a_r (Hier könnte genauso gut stehen: Falls q = a_l)
    gib aus „q ∈ F“
ansonsten
    gib aus „q ∉ F“
```

1.3 Drittes Beispiel: Knotenfärbung überprüfen

Input: Ein (einfacher) Graph mit n Knoten und m Kanten, gegeben durch seine *Adjazenzlisten*, und zu jedem Knoten eine Farbe f_v. D.h. gegeben ist für jeden Knoten v die Farbe f_v und die Folge derjenigen Knoten, die zu v adjazent sind[1].

Output: Aussage „Färbung“ oder Liste aller Kanten, deren Endknoten die gleiche Farbe haben.

Einschub: Beispiel für Adjazenzlisten:



Graph G

1 :	2	3	
2 :	3	1	4
3 :	1	4	2
4 :	2	3	

Adjazenzlisten zu G

Anmerkung: Wir nehmen an, daß die Länge einer jeden Adjazenzliste nicht von vornherein bekannt ist. Wir können uns eine Liste so vorstellen, daß hinter dem letzten Listenelement noch ein künstliches Listenelement kommt, das NIL heißt, und an dem man erkennt, daß man am Ende der Liste angekommen ist.

Algorithmus anschaulich: Überprüfe für jeden Knoten i, ob alle zu ihm adjazenten Knoten anders gefärbt sind als er selbst.

Algorithmus in Pseudocode:

```
Setze die Boole'sche Variable b := true.
für i := 1 bis n:
    f_akt := f_i
    j := erstes Element der Adjazenzliste von i
    Solange j ≠ NIL:
        Falls f_akt = f_j:
            b := false
            Gib aus „{i, j}“.
        j := nächstes Element in der Liste
Falls b = true:
    Gib aus „Färbung“
```

[1]Zu Graphen und ihrer Darstellung, z.B. durch Adjazenzlisten, siehe auch [CLR94, Kapitel 23] sowie das Handout *Grundlagen: Begriffe zu Graphen*.

Bemerkung: Da jede Kante $\{u, v\}$ zweimal in den Adjazenzlisten auftaucht (einmal in der Adjazenzliste von u und einmal in der Adjazenzliste von v), wird jede Kante, deren Endknoten dieselbe Farbe haben, auch zweimal von obigem Algorithmus „gefunden“.

2 Laufzeit eines Algorithmus

Sei A ein Algorithmus zu einem bestimmten Problem Π. Gesucht ist eine Angabe für den „Aufwand“ des Algorithmus (die Anzahl der durchgeführten Schritte, die „Laufzeit“ des Algorithmus)[2]. Präziser ausgedrückt suchen wir eine Funktion

$$T_A : \text{ Menge aller möglichen Eingaben } \rightarrow I\!N,$$

so daß $T_A(I)$ den Rechenzeitverbrauch des Algorithmus A für die Eingabe I (den Input I) angibt. Hilfsweise begnügen wir uns mit einer Funktion

$$T_A : I\!N \rightarrow I\!N$$

für die gilt: Algorithmus A braucht für eine Eingabe mit Kenngröße n höchstens $T_A(n)$ Schritte, und zwar insbesondere

- unabhängig davon, ob man den Algorithmus z.B. in Fortran, Pascal oder C implementiert, und
- unabhängig von einem konkreten Rechner, auf dem die Implementation läuft.

Wir interessieren uns besonders für Algorithmen, für die $T(n)$ (bzw. $T(|I|)$) ein Polynom (oder durch ein Polynom nach oben beschränkt) ist.[3] Dabei ist der Grad des Polynoms wichtig, während seine Koeffizienten nur sekundär interessieren.

Wir müssen zunächst klären,

a) was mit „Kenngröße“ gemeint ist

b) was als „Schritt“ eines Algorithmus gilt.

zu a) Eine Kenngröße soll ein sinnvolles Maß für die „Größe“ einer Eingabe sein. Beim „Element-von“-Problem bietet sich dafür die Anzahl n der Zahlen a_i an. Beim Knotenfärbungsproblem ist die Größe der Eingabe durch die Anzahl n der Knoten und die Anzahl m der Kanten festgelegt, wir wählen daher als Kenngröße $n + m$.

zu b) Eine Lese- oder Schreibanweisung (in obigen Algorithmen also z.B. „Gib aus ’Färbung’“), eine arithmetische Operation (z.B. „$k := \lfloor \frac{r+l}{2} \rfloor$“), ein Vergleich (z.B. „Solange $i \leq n$“), oder eine Sprunganweisung (in den Beispielen jeweils am Ende der Schleifen implizit verwendet) betrachten wir jeweils als einen Schritt. Für jeden solchen Einzelschritt veranschlagen wir einen Zeitaufwand von einer Zeiteinheit (und nennen diese Festlegung *Einheitskostenmaß*).

Diese Festlegung ist absichtlich unpräzise. Denn wenn wir „$k := \lfloor \frac{r+l}{2} \rfloor$“ z.B. als vier Schritte (drei arithmetische Operationen plus eine Variablenzuweisung) zählen anstatt als einen einzigen, dann erhöht das die Laufzeit des betreffenden Algorithmus zwar um den konstanten Faktor vier, aber es erhöht nicht die *Größenordnung* der Laufzeit. Den Begriff der Größenordnung (auch *Wachstumsrate* genannt) einer Laufzeit werden wir im Abschnitt 3 formalisieren.

[2] Vergleiche wieder [CLR94, Kapitel 1].

[3] Der Index A wird oft weggelassen, wenn aus dem Zusammenhang klar ist, welcher Algorithmus gemeint ist.

2.1 Laufzeit zum ersten Beispiel:

Wir geben im folgenden obere Schranken für die Anzahlen von Einzelschritten an, die die einzelnen Zeilen im oben angegebenen Pseudocode benötigen:

1. und 2.	zusammen zwei Schritte
3. bis 8.	Schleife mit maximal n Durchläufen, d.h. maximal $n+1$ mal die beiden Vergleiche für Zeile 3, und wenn die Bedingung in Zeile 3 nicht erfüllt ist (nach dem letzten Durchlauf) eine implizite Sprunganweisung nach Zeile 9. Pro Schleifendurchlauf maximal 4 Schritte (einschließlich der impliziten Sprunganweisung am Ende der Schleife).
9. und 10.	zusammen maximal 2 Schritte
insgesamt:	$\leq 2 \cdot (n+1) + 4 \cdot n + 5$ Schritte

Die Laufzeit ist also im wesentlichen proportional zur Kenngröße n. Daher sagen wir, die Laufzeit ist *linear*.

2.2 Laufzeit zum zweiten Beispiel:

1. und 2.	zusammen zwei Schritte
3. bis 8.	Schleife mit maximal $\lceil \log_2 n \rceil + 1$ Durchläufen (da das Intervall $[l, r]$ bei jedem Durchlauf im wesentlichen halbiert wird), d.h. maximal $\lceil \log_2 n \rceil + 2$ mal der Vergleich in Zeile 3, und einmal eine implizite Sprunganweisung nach Zeile 9 (nach dem letzten Schleifendurchlauf). Pro Durchlauf maximal 5 Schritte.
9. bis 12.	zusammen maximal 4 Schritte
insgesamt:	$\leq 1 \cdot (\lceil log_2 n \rceil + 2) + 5 \cdot (\lceil log_2 n \rceil + 1) + 7$ Schritte (*logarithmische Laufzeit*)

2.3 Laufzeit zum dritten Beispiel:

1., 10. und 11.		zusammen maximal 3 Schritte
2. bis 9.		Schleife mit n Durchläufen
	3. und 4.	zusammen zwei Schritte pro Durchlauf
	5. bis 9.	Schleife mit maximal $n-1$ Durchläufen (jeder Knoten kann maximal zu $n-1$ anderen Knoten adjazent sein), innerhalb dieser inneren Schleife maximal 4 Schritte

Die Gesamtanzahl von Einzelschritten beträgt also höchstens $c_1 \cdot n \cdot (n-1) + c_2 \cdot n + c_3$, wobei c_i positive Konstanten sind, deren genaue Werte uns nicht interessieren. Diese obere Laufzeitschranke ist im wesentlichen proportional zu n^2. Aber bei genauerer Betrachtung stellen wir fest, daß die innere Schleife 5. bis 9. insgesamt für jede Kante des Graphen höchstens zweimal durchlaufen wird. Wenn der Graph nicht viele Kanten hat, sind das deutlich weniger als n^2 Durchläufe. Eine erneute Analyse ergibt:

1., 10. und 11.	zusammen maximal 3 Schritte
2. bis 9.	(ohne die Schritte in der inneren Schleife 5. bis 9.) Schleife mit n Durchläufen und maximal 4 Schritten pro Durchlauf

5. bis 9.	gezählt über alle Durchläufe der Schleife 2. bis 9.: maximal $2 \cdot m$ Durchläufe mit maximal 5 Schritten pro Durchlauf (Diese Art der Argumentation heißt *amortisierte Analyse*.)

Insgesamt erhalten wir nun höchstens $c_4 \cdot n + c_5 \cdot m + c_6 \leq c_7 \cdot (n + m) + c_6$ Schritte. Diese obere Laufzeitschranke ist im wesentlichen proportional zur gewählten Kenngröße $n + m$. Daher bezeichnen wir die Laufzeit auch hier wieder als linear.

Wir interessieren uns wie gesagt besonders für Algorithmen, für die (im Einheitskostenmaß) $T(n)$ (bzw. $T(|I|)$) ein Polynom (oder durch ein Polynom nach oben beschränkt) ist. Ein Algorithmus heißt *polynomial*, wenn es ein Polynom p gibt, so daß für die Laufzeitfunktion T des Algorithmus gilt:

$$T(|I|) \leq p(|I|) \qquad \text{für alle Eingaben } I$$

Dabei ist wie schon gesagt nur der Grad des Polynoms wichtig. Wenn das Polynom Grad 1 hat, heißt der Algorithmus *linear*. Wir sprechen in diesem Zusammenhang auch von einem *effizienten* Algorithmus, wenn der Algorithmus polynomial ist mit einem Polynom von kleinem Grad, am besten 1. [4]

3 Wachstumsraten von Funktionen

Zu diesem Abschnitt siehe auch [CLR94, Kapitel 2] sowie [Man89, Abschnitt 3.2].

3.1 Definition:

Zur Einteilung von Wachstumsraten von Funktionen werden folgende Mengen von Funktionen definiert:

$$\begin{aligned}
\mathcal{O}(g(n)) &= \{f(n) : \exists c > 0, \exists n_0 : f(n) \leq c \cdot g(n) \quad \forall n \geq n_0\} \\
\Omega(g(n)) &= \{f(n) : \exists c > 0, \exists n_0 : f(n) \geq c \cdot g(n) \quad \forall n \geq n_0\} \\
\Theta(g(n)) &= \{f(n) : \exists c_1 > 0, c_2 > 0, \exists n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n \geq n_0\} \\
o(g(n)) &= \{f(n) : f(n) \in \mathcal{O}(g(n)) \text{ und } g(n) \notin \mathcal{O}(f(n))\}
\end{aligned}$$

Aus diesen Definitionen folgt:

$$f(n) \in \Theta(g(n)) \qquad \text{g.d.w} \qquad f(n) \in \mathcal{O}(g(n)) \text{ und } f(n) \in \Omega(g(n))$$

Wenn $f(n) \in \mathcal{O}(g(n))$, dann heißt $g(n)$ *asymptotisch obere Schranke* für $f(n)$.

Wenn $f(n) \in \Omega(g(n))$, dann heißt $g(n)$ *asymptotisch untere Schranke* für $f(n)$.

Wenn $f(n) \in \Theta(g(n))$, dann heißt $g(n)$ *asymptotisch scharfe Schranke* für $f(n)$.

3.2 Beispiele:

a) $3 \cdot n^2 \in \mathcal{O}(n^3)$ (nimm $c = 1, n_0 = 3$), d.h. $n^3 \in \Omega(3 \cdot n^2)$.

b) $\frac{3}{8}n \in \mathcal{O}(n)$ (nimm $c = \frac{3}{8}$, $n_0 = 1$)

c) $n \in \mathcal{O}(\frac{3}{8}n)$ (nimm $c = \frac{8}{3}, n_0 = 1$), damit folgt dann insgesamt: $\frac{3}{8}n \in \Theta(n)$.

[4] Der Begriff „effizienter Algorithmus“ ist notwendigerweise schwammig und wird in der Literatur uneinheitlich verwendet.

3.3 Satz:

Für alle $k > 0$ und $i > 0$ gilt:	$n^k \in o(n^{k+i})$
Für alle $k > 0$ und $a > 1$ gilt:	$n^k \in o(a^n)$
Für alle $a > 1$ und $b > 1$ mit $a < b$ gilt:	$a^n \in o(b^n)$
Für alle $b_1 > 1$ und $b_2 > 1$ gilt:	$\log_{b_1} n \in \Theta(\log_{b_2} n)$

(Dementsprechend lassen wir die Basis des Logarithmus i.a. weg, wenn wir Laufzeiten von Algorithmen angeben.)

Für alle $b > 1$, $k > 0$ und $l > 0$ gilt:	$\log_b(n^k) \in \Theta(\log_b(n^l))$
Für alle $b > 1$, $k > 0$ und $l > k$ gilt:	$(\log_b n)^k \in o((\log_b n)^l)$
Für alle $b > 1$, $k > 0$ gilt:	$(\log_b n)^k \in o(n)$

Für alle $c > 0$, $a > 1$ und für alle monoton wachsenden Funktionen $f(n)$ gilt:

$$(f(n))^c \in o(a^{f(n)})$$

3.4 Asymptotische Schranken für die Laufzeiten der Beispielalgorithmen:

erstes Beispiel:	$T(n) \in \Theta(n)$
zweites Beispiel:	$T(n) \in \Theta(\log n)$
drittes Beispiel:	$T(n) \in \Theta(n + m)$

4 „Arten“ von Laufzeit

Wir betrachten i.a. (und auch bei der Analyse obiger Beispielalgorithmen) die „worst case“-Laufzeit: $T^{wc}(n)$ beschreibt die Laufzeit, die für die „ungünstigste“ Eingabe mit Kenngröße n benötigt wird.

Ebenso interessant, aber meist schwieriger zu analysieren, ist die „durchschnittliche“ Laufzeit eines Algorithmus, die „average-case“-Laufzeit:

$$T^{ave}(n) = \frac{\text{Summe der Laufzeiten des Algorithmus über alle Eingaben mit Kenngröße } n}{\text{Anzahl der möglichen Eingaben mit Kenngröße } n}$$

Literatur

[CLR94] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw–Hill, 1994. ISBN 0-262-03141-8, 0-07-013143-0, 0-262-53091-0. Unibib KN: kid 112/c67c, lbs 830/c67.

[Man89] Udi Manber. *Introduction to algorithms: a creative approach*. Addison–Wesley Publishing Company, 1989. Unibib KN: kid 112/m16, lbs 840/m16.