Vorlesung Informatik 1 (Wintersemester 2020/2021)

Kapitel 13: Effizienz von Algorithmen

Martin Frieb
Johannes Metzger

Universität Augsburg
Fakultät für Angewandte Informatik

27. Januar 2021

Das Problem des Handlungsreisenden

Manche Probleme sind theoretisch durch Algorithmen lösbar, praktisch insbesondere für große Instanzen aber nicht

Beispiel 13.1 (Das Problem des Handlungsreisenden)

Eingabe: n Städte mit ihren wechselseitigen Entfernungen ($n > 2$)
Ausgabe: Die kürzeste Rundreise durch diese Städte

Algorithmus

Durchlaufe systematisch in einer Schleife alle möglichen Rundreisen und speichere die kürzeste.

Das ist **keine effiziente Lösung**:
Für n Städte gibt es $n!$ mögliche Rundreisen. Schon für $n > 30$ braucht der Rechner sehr lange (mehrere Tage), um alle Rundreisen zu durchlaufen.

Suche in sortierter Folge

Probleme können von verschiedenen Algorithmen gelöst werden, die unterschiedlich viel Zeit benötigen

Beispiel 13.2 (Suche Einfügeposition in einer sortierten Folge)

Eingabe: $a_1, \ldots, a_n$ mit $\forall i \in \{1, 2..., n-1\}(a_i \leq a_{i+1})$ und ein Suchwert a
Ausgabe: Die größte Zahl $k \in \{1, \ldots, n+1\}$ mit $\forall i \in \{1, 2, ..., k-1\}(a > a_i)$.

Algorithmus

- Sequentieller Suchalgorithmus:
  a muss im schlechtesten Fall mit allen n Folgenelementen verglichen werden (1000 Vergleiche für $n = 1000$)
- Binärer Suchalgorithmus:
  a muss im schlechtesten Fall nur mit $log(n)$ Folgenelementen verglichen werden (($log(1000) \approx 10$ Vergleiche für $n = 1000$))

Der binäre Suchalgorithmus ist **effizienter** als der sequentielle Suchalgorithmus

Fazit

Für die Praxis spielt der **Zeitbedarf** eines Algorithmus eine wichtige Rolle

In dieser Vorlesung:
Berechnung und Abschätzung des (maximalen) Zeitbedarfs eines konkret gegebenen Algorithmus für kleine und große Probleminstanzen

Ausblick:

- Neben dem Zeitbedarf betrachtet man auch den **Speicherbedarf**
- Entwicklung von Algorithmen, die ein vorgegebenes Problem mit möglichst geringem Zeit- und Speicherplatzverbrauch lösen

(siehe *Informatik 3*, *Einführung in die Theoretische Informatik* und Veranstaltungen der Lehrstühle Prof. Hagerup und Prof. Mömke)

Einflussfaktoren auf den Zeitbedarf eines Algorithmus

Der gesamte Zeitbedarf für die Lösung einer Probleminstanz entspricht der Summe der Zeitbedarfe der auszuführenden (Rechen-)Operationen. Er hängt damit ab von:

- **der Anzahl der auszuführenden Operationen**:
  diese Zahl hängt in der Regel von der **Problemgröße** (siehe unten) und der **Abstraktionsebene der Operationen** (Maschinen-, Programmiersprachen-, Algorithmusebene) ab
- **dem Zeitbedarf für die Ausführung der einzelnen Operationen**:
  dieser Zeitbedarf hängt von der eingesetzten **Rechenanlage** und der benutzten **Programmiersprache** ab

Problemgröße

In dieser Vorlesung betrachten wir als **Problemgröße**

- entweder die **Anzahl der Eingabedaten**
  Beispiel: Bei der sequentiellen Suche müssen um so mehr Vergleiche durchgeführt werden, je mehr Elemente die Folge hat.
- oder die **Größe eines Eingabewerts**
  Beispiel: Bei der Berechnung der Fakultät von n müssen umso mehr Schleifendurchläufe durchgeführt werden, je größer n ist.

Berechnung des Zeitbedarfs eines Algorithmus

Zeitbedarf

Bei der Berechnung des **Zeitbedarfs** abstrahieren wir von der eingesetzten Programmiersprache und Rechenanlage und nehmen an, dass jede Operation **den gleichen Zeitbedarf** hat (Rechtfertigung für dieses Vorgehen am Ende des Kapitels).

Unter diesen Voraussetzungen können wir den Zeitbedarf berechnen, indem wir den Algorithmus in einer **formalen** programmiersprachenunabhängigen Darstellung betrachten (Pseudocode, Struktogramm, Programmablaufplan) und **die Anzahl der auszuführenden Operationen in Abhängigkeit von der Problemgröße zählen**.

Ausblick weiterführende Vorlesungen

In den Vorlesungen *Informatik 3* und *Einführung in die Theoretische Informatik* betrachtet man auch abstrakte Rechnermodelle (Random-Access-Maschine (RAM), Turing-Maschine, ...) anstatt programmiersprachenunabhängigen Darstellungen von Algorithmen für Effizienzbetrachtungen.

Was ist eine ELOP?

Wir betrachten ab jetzt Algorithmen auf der Abstraktionebene einer **formalen programmiersprachenunabhängigen Darstellung** und die auf dieser Abstraktionsebene auszuführenden Operationen als **Elementaroperationen** (**ELOPs**). Wir zählen die Anzahl der auszuführenden ELOPs zur Bearbeitung einer Probleminstanz.

Das sind ELOPs

- Wertzuweisungen ($\leftarrow$)
- Arithmetische Operationen ($+, -, /, *, \div, \mathrm{mod}$)
- Vergleiche von Werten ($\leq, \geq, <, >, =, \neq$)
- Einfache mathematische Operationen ($|\cdot|$)
- Logische Operationen ($\neg, \wedge, \vee$)

Das sind **keine** ELOPs

Aufwendige mathematische Operationen / Überprüfungen (z.B Matrizenmultiplikation, Elementbeziehung, Fakultätsfunktion, Mengenoperationen), **natürlichsprachliche Anweisungen**

Den Zeitbedarf berechnen: ELOPs zählen

Definition 13.3 (Zeitbedarf (formal))

Der **Zeitbedarf** $t(A, e)$ eines Algorithmus A angewendet auf eine Eingabe e ist die Anzahl der ausgeführten ELOPs ohne Berücksichtigung von Eingabe und Ausgabe.

Zur Berechnung des Zeitbedarfs muss insbesondere die **Anzahl der Schleifendurchläufe** bestimmt werden

Beispiel 13.4 (Summe einer Zahlenfolge berechnen)

```
Eingabe : x_1, ..., x_n ∈ ℤ, n ∈ ℕ
1  s ← 0;
2  i ← 1;
3  solange (i ≤ n) tue
4  |   s ← s + x_i;
5  |   i ← i + 1;
   Ausgabe : s
```

Problemgröße: n
Anzahl ELOPs hängt nur ab von n, aber nicht von Größe der Zahlen $x_1, \ldots, x_n$

Anzahl der Schleifendurchläufe: n
Ist gleich für jede Wahl von $x_1, \ldots, x_n$
Schleifenbedingung wird $n+1$-mal überprüft

Den Zeitbedarf berechnen: ELOPs zählen

Definition 13.3 (Zeitbedarf (formal))

Der **Zeitbedarf** $t(A, e)$ eines Algorithmus A angewendet auf eine Eingabe e ist die Anzahl der ausgeführten ELOPs ohne Berücksichtigung von Eingabe und Ausgabe.

Zur Berechnung des Zeitbedarfs muss insbesondere die **Anzahl der Schleifendurchläufe** bestimmt werden

Beispiel 13.4 (Summe einer Zahlenfolge berechnen)

Eingabe : $x_1, \ldots, x_n \in \mathbb{Z}, n \in \mathbb{N}$

1 $s \leftarrow 0;$
2 $i \leftarrow 1;$
3 **solange** $(i \leq n)$ **tue**
4 $\quad s \leftarrow s + x_i;$
5 $\quad i \leftarrow i + 1;$

Ausgabe : s

ELOPs:
1 (Wertzuweisung)
1 (Wertzuweisung)
$1 \cdot (n+1)$ (Vergleich)
$2 \cdot n$ (Addition + Wertzuweisung)
$2 \cdot n$ (Addition + Wertzuweisung)
Ergebnis: $5 \cdot n + 3$

Den Zeitbedarf berechnen: ELOPs zählen

Beispiel 13.5 (Sequentielle Suche in einer sortierten Folge)

Eingabe : $x_1, \ldots, x_n \in \mathbb{Z}$, $\forall i \in \{1, 2, ..., n-1\}(x_i \leq x_{i+1}), n \in \mathbb{N}, x \in \mathbb{Z}$

```
1 i ← 1;
2 solange (i ≤ n) tue
3     wenn x ≤ x_i dann
          Ausgabe : i
4     i ← i + 1;
Ausgabe : i
```

ELOPs:

1

Minimal: 1, Maximal: $1 \cdot (n+1)$

Minimal: 1, Maximal: $1 \cdot n$

Minimal: 0, Maximal: $2 \cdot n$

Minimal: 3**, Maximal:** $4 \cdot n + 2$

Problemgröße: n

Anzahl der Schleifendurchläufe: hängt ab vom Wert von x

- Für $x \leq x_1$: Schleife wird im ersten Durchlauf abgebrochen (minimale Anzahl)
- Für $x > x_n$: Schleife wird n-mal durchlaufen (maximale Anzahl)

Die Anzahl der ELOPs muss nicht für jede Eingabe e **der Größe** n **gleich sein!**

Den Zeitbedarf berechnen: ELOPs zählen

Beispiel 13.6 (Matrixoperation)

		ELOPs:
	Eingabe : $(m_{i,j})_{1\leq i,j\leq n} \in (\mathbb{Z}^n)^n, n \in \mathbb{N}$	
1	$i \leftarrow 1;$	1
2	$s \leftarrow 0;$	1
3	**solange** $(i \leq n)$ **tue**	$1 \cdot (n+1)$
4	$\quad j \leftarrow 1;$	$1 \cdot n$
5	$\quad$ **solange** $(j < i)$ **tue**	$1 \cdot (1+2+\ldots+n)$
6	$\quad\quad s \leftarrow s + m_{i,j};$	$2 \cdot (1+2+\ldots+n-1)$
7	$\quad\quad j \leftarrow j+1;$	$2 \cdot (1+2+\ldots+n-1)$
8	$\quad i \leftarrow i+1;$	$2 \cdot n$
	Ausgabe : s	**Ergebnis:** $2.5 \cdot n^2 + 2.5 \cdot n + 3$

Anzahl der Durchläufe der inneren Schleife: abhängig von äußerer Schleife

- Im k-ten Durchlauf der **äußeren** Schleife wird die **innere** Schleife $k-1$-mal durchlaufen.
- Formel zu Berechnung des Ergebnisses: $\sum_{k=1}^{n} k = \frac{n\cdot(n+1)}{2}$.

Was ist Zeitkomplexität?

Definition 13.7 (Zeitkomplexität)

Es bezeichne $|e|$ die Größe einer Eingabe e und A einen Algorithmus.

- Die **Best-Case-Komplexität** ist die Funktion $T_A^{min} : \mathbb{N} \to \mathbb{N}$ mit

$$T_A^{min}(n) := min\{t(A, e) \mid |e| = n\}$$

  Anwendung: Gibt eine **untere Schranke** für den Zeitbedarf an.

- Die **Worst-Case-Komplexität** ist die Funktion $T_A^{max} : \mathbb{N} \to \mathbb{N}$ mit

$$T_A^{max}(n) := max\{t(A, e) \mid |e| = n\}$$

  Anwendung: Gibt eine **obere Schranke** für den Zeitbedarf an.

Beispiele für Zeitkomplexität

Eingabe : $x_1, \ldots, x_n \in \mathbb{Z}, n \in \mathbb{N}$

1 $s \leftarrow 0$;
2 $i \leftarrow 1$;
3 **solange** $(i \leq n)$ **tue**
4 $\quad s \leftarrow s + x_i$;
5 $\quad i \leftarrow i + 1$;

Ausgabe : s

ELOPs:
1 (Wertzuweisung)
1 (Wertzuweisung)
$1 \cdot (n+1)$ (Vergleich)
$2 \cdot n$ (Addition + Wertzuweisung)
$2 \cdot n$ (Addition + Wertzuweisung)
$T_A^{min}(n) = T_A^{max}(n) = 5 \cdot n + 3$

Eingabe : $x_1, \ldots, x_n \in \mathbb{Z}, \forall i \in \{1, 2, ..., n-1\}(x_i \leq x_{i+1}), n \in \mathbb{N}, x \in \mathbb{Z}$

1 $i \leftarrow 1$;
2 **solange** $(i \leq n)$ **tue**
3 $\quad$ **wenn** $x \leq x_i$ **dann**
$\quad\quad$ **Ausgabe :** i
4 $\quad i \leftarrow i + 1$;

Ausgabe : i

ELOPs:

1
Minimal: 1, Maximal: $1 \cdot (n+1)$
Minimal: 1, Maximal: $1 \cdot n$

Minimal: 0, Maximal: $2 \cdot n$
$T_A^{min}(n) = 3,\ T_A^{max}(n) = 4 \cdot n + 2$

Größenordnungen für Zeitkomplexität

In Theorie und Praxis interessiert man sich

- **nicht für die genaue Anzahl** der ELOPs,
- sondern für die **Größenordnung** des **Wachstums der Anzahl der ELOPs mit Erhöhung der Problemgröße**

Beschreibung dieses Wachstums durch die sog. **O-Notation** (Sprechweise: **“Groß-Oh-Notation”**)

Definition 13.8 (O-Notation)

Seien P die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{R}^+$, $g, h \in P$, $n, n_0 \in \mathbb{N}$ und $c \in \mathbb{R}^+$. Wir definieren die Menge **“Groß-Oh von g”** $O(g)$:

$$O(g) := \{h \mid \exists n_0(\exists c(\forall n > n_0(h(n) \leq c \cdot g(n))))\}$$

Interpretation und Sprechweisen

Eine Funktion $h \in O(g)$ wächst für große n höchstens so stark wie g. Man sagt

- h ist aus Groß-Oh von g
- h hat höchstens die Ordnung g

Konstante Zeitkomplexität $O(1)$

Wir unterscheiden die sog. **Komplexitätsklassen** $O(1)$, $O(log(n))$, $O(n)$, $O(n^2)$. $O(2^n)$

Definition 13.9 (Konstante Zeitkomplexität)

Ein Algorithmus A hat **konstante Zeitkomplexität**, falls $T_A^{max} \in O(1)$.
(d.h. der Zeitbedarf von A **ist unabhängig von** der Problemgröße)

Beispiel 13.10

- $h \in O(1)$ für $h(n) := 1000$: Es gilt $\forall n > 0 (h(n) \leq 1000 \cdot 1)$ $(n_0 = 1, c = 1000)$
- $h \notin O(1)$ für $h(n) := n$: Es gilt $\forall n > n_0 (h(n) \leq c \cdot 1) \Leftrightarrow \forall n > n_0 (n \leq c)$ (nicht erfüllbar).

Beispiele 13.11 (Algorithmen mit konstanter Zeitkomplexität)

- Einfügen eines neuen Werts in ein (statisches oder dynamisches) unsortiertes Feld
- Prüfung ob eine Zeichenkette leer ist

Logarithmische Zeitkomplexität $O(log(n))$

Definition 13.12 (Logarithmische Zeitkomplexität)

Ein Algorithmus A hat **logarithmische Zeitkomplexität**, falls $T_A^{max} \in O(log(n)) \setminus O(1)$.
(d.h. der Zeitbedarf von A **steigt um** 1 **bei Verdopplung der Problemgröße**, denn $log(2 \cdot n) = log(2) + log(n) = 1 + log(n)$)

Beispiel 13.13

- $h \in O(log(n))$ für $h(n) := 6 \cdot log(n) + 1000$: Es gilt $h(n) \leq 7 \cdot log(n) \Leftrightarrow 1000 \leq log(n) \Leftrightarrow 2^{1000} \leq n$ $(n_0 = 2^{1000}, c = 7)$
- $h \notin O(log(n))$ für $h(n) := n$: Es gilt $\forall n > n_0 (h(n) \leq c \cdot log(n)) \Leftrightarrow \forall n > n_0 (\frac{n}{log(n)} \leq c)$ (nicht erfüllbar: siehe Anhang).

Beispiele 13.14 (Algorithmen mit logarithmischer Zeitkomplexität)

- Binäre Suche in einer sortierten Folge

Lineare Zeitkomplexität $O(n)$

Definition 13.15 (Lineare Zeitkomplexität)

Ein Algorithmus A hat **lineare Zeitkomplexität**, falls
$T_A^{max} \in O(n) \setminus O(log(n))$.
(d.h. der Zeitbedarf von A **wächst proportional zur Problemgröße**)

Beispiel 13.16

- $h \in O(n)$ für $h(n) := 3 \cdot n + 100$: Es gilt $h(n) \leq 4 \cdot n \Leftrightarrow 100 \leq n$
  $(n_0 = 100, c = 4)$
- $h \notin O(n)$ für $h(n) := n^2$: Es gilt
  $\forall n > n_0 (h(n) \leq c \cdot n) \Leftrightarrow \forall n > n_0 (n \leq c)$ (nicht erfüllbar).

Beispiele 13.17 (Algorithmen mit linearer Zeitkomplexität)

- Sequentielle Suche in einer sortierten Folge
- Umkehrung einer Zeichenkette

Quadratische Zeitkomplexität $O(n^2)$

Definition 13.18 (Quadratische Zeitkomplexität)

Ein Algorithmus A hat **quadratische Zeitkomplexität**, falls $T_A^{max} \in O(n^2) \setminus O(n)$. (d.h. der Zeitbedarf von A **vervierfacht sich bei Verdopplung der Problemgröße**, denn $(2 \cdot n)^2 = 4 \cdot n^2$)

Beispiel 13.19

- $h \in O(n^2)$ für $h(n) := n^2 + 10 \cdot n - 2$: Es gilt $h(n) \leq 2 \cdot n^2 \Leftrightarrow 10 \cdot n - 2 \leq n^2 \Leftarrow 10 \cdot n \leq n^2 \Leftrightarrow 10 \leq n$ $(n_0 = 10, c = 2)$
- $h \notin O(n^2)$ für $h(n) := 2^n$: Es gilt $\forall n > n_0 (h(n) \leq c \cdot n^2) \Leftrightarrow \forall n > n_0 (\frac{2^n}{n^2} \leq c)$ (nicht erfüllbar: siehe Anhang).

Beispiele 13.20 (Algorithmen mit quadratischer Zeitkomplexität)

- Summieren aller Einträge einer quadratischen Matrix (Problemgröße = Anzahl der Zeilen)

Exponentielle Zeitkomplexität $O(2^n)$

Definition 13.21 (Exponentielle Zeitkomplexität)

Ein Algorithmus A hat **exponentielle Zeitkomplexität**, falls $T_A^{max} \in O(2^n) \setminus \bigcup_{k \in \mathbb{N}} O(n^k)$.
(d.h. der Zeitbedarf von A **quadriert sich bei Verdopplung der Problemgröße**, denn $2^{2 \cdot n} = 2^n \cdot 2^n = (2^n)^2$)

Beispiel 13.22

- $h \in O(2^n)$ für $h(n) := 2^n + n^2$: Es gilt $h(n) \leq 2 \cdot 2^n \Leftrightarrow n^2 \leq 2^n \Leftrightarrow 1 \leq \frac{2^n}{n^2}$ (erfüllt für $n \geq 4$, also $n_0 = 4, c = 2$))
- $h \notin O(2^n)$ für $h(n) := 2^{2 \cdot n}$: Es gilt $\forall n > n_0 (h(n) \leq c \cdot 2^n) \Leftrightarrow \forall n > n_0 (2^n \leq c)$ (nicht erfüllbar).

Beispiele 13.23 (Algorithmen mit exponentieller Zeitkomplexität)

- Viele Optimierungsalgorithmen (z.B. zur Lösung des Problems des Handlungsreisenden)

Bewertung der O-Notation

- Die O-Notation beschreibt das Wachstum von Funktionen **für große** n
- Für **große Problemausprägungen** sind Aussagen mit dieser Notation meist relativ genau zutreffend, da konstante Summanden und Multiplikatoren in den Hintergrund treten
- Für **kleine Problemausprägungen** kommt konstanten Multiplikatoren große Bedeutung zu; Die O-Notation ist in diesen Fällen oft sehr unpräzise
- Algorithmen mit **quadratischer Zeitkomplexität** sind in der Praxis für große n i.d.R. gerade noch brauchbar
- Algorithmen mit **exponentieller Zeitkomplexität** sind allenfalls für kleine n geeignet

Die für ELOPs gewählte Abstraktionsebene hat **keinen** Einfluss auf die Komplexitätsklasse eines Algorithmus

- Jede ELOP eines Algorithmus in formaler programmiersprachenunabhängiger Darstellung muss auf Rechnerebene durch viele Rechenschritte (Takte) implementiert werden
- **Aber**: Die Anzahl dieser Rechenschritte hängt nicht von der Problemgröße ab, ist also im Sinne der O-Notation **konstant**
- **Folgerung**: Auf dem Abstraktionsniveau des Rechners gehören die Algorithmen zur selben Komplexitätsklasse wie auf der von uns betrachteten ELOP-Ebene

Die Exponentialfunktion

Exponentialfunktion zur Basis 2: $g(n) := 2^n$

- $2^n \cdot 2^m = 2^{n+m}$ (Rechenregel)
- $2^{log(n)} = n$ (Umkehrfunktion)

Asymptotisches Verhalten für große n (Mathematischer Satz)

Für jedes $c > 0$ und $k > 0$ gibt es ein $n > 0$ mit $\frac{2^n}{n^k} > c$
(2^n wächst schneller als jedes n^k)

Der Logarithmus

Wir betrachten hier ausschließlich den Logarithmus zur Basis 2.

Logarithmusfunktion zur Basis 2: $log(n)$

- $log(2^n) = n$ (Umkehrfunktion)
- $log(n \cdot m) = log(n) + log(m)$ (Rechenregel)
- $log(2) = log(2^1) = 1$
- $log(1) = log(2^0) = 0$
- $log(0.5) = log(2^{-1}) = -1$

Asymptotisches Verhalten für große n (Mathematischer Satz)

Für jedes $c > 0$ und $k > 0$ gibt es ein $n > 0$ mit $\frac{n^k}{log(n)} > c$

(Jedes n^k wächst schneller als $log(n)$)