Kapitel 10

Komplexität von Algorithmen

Ziele:

- Zeit- und Speicherplatzbedarf einer Anweisung berechnen können
- Zeit- und Speicherplatzbedarf einer Methode berechenen können
- Unterschiede der Komplexität vom schlechtesten, besten und mittleren Fall kennen
- *O*-Notation kennenlernen
- „praktische“ Berechenbarkeit eines Algorithmus einschätzen können

Ein Algorithmus ist umso effizienter, je geringer der Aufwand zu seiner Abarbeitung ist. Wir unterscheiden zwei Arten von Komplexität: Zeitkomplexität und Speicherplatzkomplexität.

10.1 Zeit- und Speicherplatzbedarf

Der Aufwand für die Abarbeitung eines Algorithmus hängt ab von

- Art, Anzahl und Zusammensetzung der verwendeten Datentypen und algorithmischen Konzepte
- Art der Realisierung der Datentypen und algorithmischen Konzepte auf einer bestimmten Rechenanlage (z.B. Verwaltungsaufwand bei Rekursionen) sowie von konkreten Maschineneigenschaften (z.B. Art und Ausführungsgeschwindigkeit der Maschinenoperationen).

Wir nehmen idealisierend an:

Speicherplatzbedarf S für die Deklaration von	Summe des Speicherplatzbedarfs der Attribute
Zahl-, Char-, Boole'schen Werten	1 Maschinenwort
Feldern vom Typ `type`	Länge des Feldes * Bedarf für 1 Wert vom Typ `type`
z.B. einstufigen **`int`**-Feldern	Länge des Feldes viele Maschinenworte
zweistufigen Feldern **`int`**`[n][m]`	$(n * m)$ viele Maschinenworte
Referenzvariablen (Objektreferenzen) und Objekte	1 Maschinenwort

Der Speicherplatzbedarf einer Folge stms von Anweisungen ist die Summe der Kosten der in stms auftretenden Deklarationen.

Zeitbedarf T für die Deklaration von	Zeitbedarf
Konstante `k`	konstant c_{fetch}
lokale Variable `x`	konstant c_{fetch}
Zuweisung `x = exp;`	c_{store} + Zeit von `exp`
Grundoperationen:	
`+`	konstant c`+`
`*`	konstant c`*`
`-1`	konstant c`-1`
`<`	konstant c`<`
`exp`$_1$`+ exp`$_2$	Zeit von `exp`$_1$ + c`+` + Zeit von `exp`$_2$
`new` `C(t)`	Zeit von `t` + c_{store} + c_{call} + $T_{\texttt{C}}$(R-Wert von `t`)
S_1 S_2	Zeit von S_1 + Zeit von S_2
`o.att`	Zeit von `o` + $c_{\texttt{.}}$
`if` `(B)` S_1 **`else`** S_2	Zeit von `B` + Zeit von S_1, falls `b == `**`true`**, Zeit von `B` + Zeit von S_2, falls `b == `**`false`**
`while` `(B) S`	Anzahl der Durchläufe der Schleife * (Zeit von B + Zeit von S) + Zeit von B
`{ S }`	Zeit von S

Sei die Funktions-/Prozedurdeklaration `type f ([VAR] type x) {stms}` gegeben. Der *Zeitbedarf eines Aufrufs* `f(a)` hängt ab von den Kosten der Parameterübergabe c_p, bestehend aus

- den Kosten zur Berechnung von `a` und
- den Kosten der Übergabe (des Wertes) von `a` (die Kosten des „organisatorischen" Bedarfs zur Berechnung der Rücksprungadresse etc. werden vernachlässigt), sowie

- der Anzahl der Berechnungsschritte des Rumpfes

und ist definiert als

$$T_{f(a)} = c_{call} + T_a + c_{store} + T_f(a)$$

wobei $T_f(a)$ die Anzahl der Berechnungsschritte des Rumpfes (in Abhängigkeit vom R-Wert von a) angibt.

Der *Speicherplatzbedarf eines Aufrufs* `f(a)` hängt ab von

- den Kosten S_a des aktuellen Parameters,
- den organisatorischen Kosten der Parameterübergabe s_{call} und
- den Kosten $S_f(a)$ für die lokalen Deklarationen.

Der Speicherplatzbedarf von `f(a)` ist definiert als

$$S_{f(a)} = s_{call} + S_a + S_f(a)$$

Beispiele:

1. $T_{\texttt{int result = 1;}} = c_{store} + c_{fetch}$
   $S_{\texttt{int result = 1;}} = 1$
2. Sei $stm \equiv$ `result = x * result; x = x -1;`
   $T_{stm} = 2c_{fetch} + c_* + c_{store} + c_{fetch} + c_{-1} + c_{store}$
   $S_{stm} = 0$
3. Sei $stm \equiv$ `Point p = new Point(0, 1);`
   $T_{stm} = \underbrace{c_{store}}_{\text{Zuweisung an p}} + c_{call} + \underbrace{2c_{fetch}}_{\text{Kosten zur Berechnung von 0,1}} + \underbrace{2c_{store}}_{\text{Übergabe der akt. Parameter}} + \underbrace{2 \cdot (c_{fetch} + c_{store})}_{T_{\texttt{Point}}(0,1)}$
   $S_{stm} = \underset{\uparrow}{1} + \underset{\uparrow}{2}$
   p Halde

Die folgenden Beispiele sind Beispiele für den Zeit- und Speicherplatzbedarf des Rumpfs einer Methode.

1. Iterative Fakultät:

```
int fak1(int x)
{
    int result = 1;
    while (x > 0) {
        result = x * result;
        x = x - 1;
    }
    return result;
}
```

Die Kosten berechnen sich wie folgt:

$$
\begin{aligned}
T_{fak1}(x) &= c_{store} + c_{fetch} && \text{// Kosten der Vorbesetzung}\\
&\quad + x * (c_1 + c_2 + c_3) + c_1 && \text{// Kosten der Schleife}\\
&\quad + c_{return} && \text{// Kosten von } \texttt{return}\\
S_{fak1}(x) &= 1 && \text{// Kosten der lokalen Variablen}
\end{aligned}
$$

wobei $c_1 = 2c_{fetch} + c_2 + c_*$, $c_2 = 2c_{fetch} + c_{store} + c_>$, $c_3 = c_{fetch} + c_{store} + c_{-1}$.

2. Binäre Suche:

```
class SearchArray
{
    char[] arr;

    boolean binSuch(char e)
    {
        int left = 0;
        int right = this.arr.length - 1;
        int mid;
        boolean found = false;

        while ((left <= right) & !found) {
            mid = (left + right)/2;
            if (e < this.arr[mid]) right = mid - 1;
            else if (e > this.arr[mid]) left = mid + 1;
            else found = true;
        }
        return found;
    }
}
```

Kosten für die binäre Suche:

$$
\begin{aligned}
T_{\texttt{binSuch}}(e) \leq\ & 2(c_{fetch} + c_{store}) + c_{-1} + 2c_{\texttt{.}} && \text{// Vorbesetzung}\\
&+ c_{fetch} + c_{store} && \text{// Vorbesetzung}\\
&+ c_{return} && \text{// Kosten von } \texttt{return}\\
&+ \lceil \log_2 \texttt{a.length} \rceil * \\
&((3c_{fetch} + c_{\leq} + c_{!}) && \text{// Bedingung}\\
&+ c_{store} + 3c_{fetch} + c_{/} + c_{+} && \text{// mid = ...}\\
&+ 2 \cdot (3c_{fetch} + c_{[]} + \max\{c_<, c_>\}) && \text{// 2mal Bedingung}\\
&+ c_{fetch} + c_{store} + \max\{c_<, c_{==}\}) && \text{// Ja-, Nein-Zweig}\\
&+ (3c_{fetch} + c_{\leq} + c_{!} + c_{\&})
\end{aligned}
$$

wobei $\lceil \log_2 x \rceil$ die kleinste ganze Zahl bezeichnet, die größer oder gleich $\log_2 x$ ist.

$$S_{\texttt{binSuch}}(e) = 4 \qquad \text{// Kosten der lokalen Variablen}$$

3. Rekursive Fakultät:

```
1 public static int fac(int n)
2 {
3     if (n == 0)
4         return 1;
5     else
6         return n * fac(n - 1);
7 }
```

Die Kosten für die Laufzeit berechnen sich wie folgt:

	Zeit	
Anweisung	`n == 0`	`n > 0`
(3)	$2c_{fetch} + c_{<}$	$2c_{fetch} + c_{<}$
(4)	$c_{fetch} + c_{return}$	—
(6)	—	$3c_{fetch} + c_{-} + c_{store} + c_{*} +$ $+c_{call} + c_{return} + T_{\texttt{fac}}(n-1)$

Also

$$T_{\texttt{fac}}(n) = \begin{cases} t_1 & \text{falls } \texttt{n == 0} \\ T_{\texttt{fac}}(n-1) + t_2 & \text{falls } \texttt{n > 0} \end{cases}$$

mit $t_1 = 3c_{fetch} + c_{<} + c_{return}$ und $t_2 = 3c_{fetch} + c_{-} + c_{store} + c_{*} + c_{call} + c_{return}$
Die Lösung dieser Gleichung erhält man durch wiederholte Substitution:

$$\begin{aligned} T_{\texttt{fac}}(n) &= T_{\texttt{fac}}(n-1) + t_2 \\ &= T_{\texttt{fac}}(n-2) + 2 \cdot t_2 \\ &\vdots \\ &= T_{\texttt{fac}}(0) + n \cdot t_2 \end{aligned}$$

d.h.

$$T_{\texttt{fac}}(n) = t_1 + n \cdot t_2$$

Im Vergleich dazu erhält man für die iterative Berechnung

$$T_{\texttt{fak1}}(n) = t_1' + n \cdot t_2'$$

mit $t_1' = 3c_{fetch} + c_{store} + c_{<} + c_{return}$ und $t_2' = 5c_{fetch} + 2c_{store} + c_{<} + c_{-1}$. D.h. die Laufzeiten unterscheiden sich nicht prinzipiell (siehe später).

Die Kosten für den Speicherplatzbedarf berechnen sich folgendermaßen:

Speicher	
`n == 0`	`n > 0`
$1 + s_{call}$	$1 + s_{call} + S_{\texttt{fac}}(n-1)$

Wieder ergibt sich

$$S_{\texttt{fac}}(n) = \begin{cases} S_1 & \text{falls } \texttt{n == 0} \\ S_{\texttt{fac}}(n-1) + S_2 & \text{falls } \texttt{n > 0} \end{cases}$$

und damit

$$S_{\texttt{fac}(n)} = S_1 + n \cdot S_2$$

Während der Speicherplatzbedarf von `fac1` gleich $2 + S_{call}$, also konstant, ist, hängt der Speicherplatzbedarf der rekursiven Funktion von der Größe von n ab und wächst linear mit n.

Um den Zeit- und Speicherplatzbedarf verschiedener Algorithmen vergleichen zu können, abstrahiert man von der speziellen Eingabe und gibt die Kosten *in Abhängigkeit von der Größe der Eingabe* an. Man unterscheidet die Komplexität *im schlechtesten, mittleren und besten Fall* (engl. worst case, average case, best case):

$$\begin{aligned} T_f^w(n) &= \max\{T_f(a) \mid \text{Größe von } a = n\} \\ T_f^{av}(n) &= \text{Durchschnitt von } \{T_f(a) \mid \text{Größe von } a = n\} \\ T_f^b(n) &= \min\{T_f(a) \mid \text{Größe von } a = n\} \end{aligned}$$

$$\begin{aligned} S_f^w(n) &= \max\{S_f(a) \mid \text{Größe von } a = n\} \\ S_f^{av}(n) &= \text{Durchschnitt von } \{S_f(a) \mid \text{Größe von } a = n\} \\ S_f^b(n) &= \min\{S_f(a) \mid \text{Größe von } a = n\} \end{aligned}$$

Beispiel:

$$T^w_{\texttt{binSuch}}(n) = \max\{T_{\texttt{binSuch}}(e, a) \mid \text{Größe von } a = (n-1)\}$$

da die Größe von $e = 1$. Für `fac`, `fak1` stimmen der beste, schlechteste und mittlere Fall jeweils überein.

$$\begin{aligned} &T_{\texttt{such}}(\textbf{this}, \texttt{e}) \leq t_1 + n \cdot t_2 \quad \text{mit } n = \textbf{this}\texttt{.arr.length} \\ &S_{\texttt{such}}(\textbf{this}, \texttt{e}) = 2 \end{aligned}$$

Beispiel 10.1 Suche im (ungeordneten) Feld

```
class SearchArray
{
    int[] arr;
    ...

    boolean such(int e)
    {
        int k = this.arr.length;
        int i = 0;

        while (i < k & !(arr[i] == e))
            i = i + 1;

        return (i < k);
    }
}
```

wobei

$$t_1 = \underbrace{c_{store} + 2c_{.} + c_{fetch}}_{\texttt{int k = ...}} + \underbrace{c_{store} + c_{fetch}}_{\texttt{int i = 0}} + \\ + \underbrace{2c_{fetch} + c_{<} + c_{return}}_{\texttt{return}} + \underbrace{2c_{fetch} + c_{<} + c_{!} + 3c_{fetch} + c_{.} + c_{[]}}_{\text{Bedingung} := c_b}$$

und

$$t_2 = c_b + c_{store} + c_{fetch} + c_{+1}$$

Als Größe von (**this**, e) betrachten wir die Längen von **this**.arr.

$$\begin{aligned} T^{w}_{\mathsf{such}}(n) &= \max\{T_{\mathsf{such}}(\textbf{this}, \mathsf{e}) \mid \text{Größe von } (\textbf{this}, \mathsf{e}) = n\} \\ &= t_1 + n \cdot t_2 \\ T^{av}_{\mathsf{such}}(n) &= \text{Durchschnitt von } \{T_{\mathsf{such}}(\textbf{this}, \mathsf{e}) \mid \text{Größe von } (\textbf{this}, \mathsf{e}) = n\} \\ &= t_1 + \frac{\sum_{i=1}^{n} i}{n} \cdot t_2 = t_1 + \frac{n+1}{2} \cdot t_2 \end{aligned}$$

wenn man das arithmetische Mittel als Durchschnitt wählt, i.a. hängt dies von der Verteilung der Elemente ab.

$$\begin{aligned} T^{b}_{\mathsf{such}}(n) &= \min\{T_{\mathsf{such}}(\textbf{this}, \mathsf{e}) \mid \text{Größe von } (\textbf{this}, \mathsf{e}) = n\} \\ &= t_1 + t_2 \end{aligned}$$

d.h. bei such stimmen die Komplexitäten des besten und schlechtesten Falls nicht überein.

Im Folgenden bezeichnen wir die Komplexität im schlechtesten Fall meist mit $T_f(n)$ und $S_f(n)$ anstelle von $T_f^w(n)$ und $S_f^w(n)$.

10.2 Größenordnung der Komplexität: Die O-Notation

Die Komplexität $T(n)$ bzw. $S(n)$ wird häufig nur bis auf konstante Faktoren untersucht, da der gleiche Algorithmus, auf unterschiedlichen Rechnern oder unterschiedlichen Programmiersprachen kodiert, oder von unterschiedlichen Implementierungen der gleichen Sprache verarbeitet, unterschiedlich hohen Aufwand verursacht. Dazu ordnen wir den Aufwandsfunktionen $T(n)$ und $S(n)$ bestimmte Funktionenklassen ihrer „Größenordnung“ zu.

Wir definieren für eine Funktion $f : \mathbb{N} \longrightarrow \mathbb{N}$ (bzw. $f : \mathbb{N} \longrightarrow \mathbb{R}$):

$$O(f(n)) = \{g(n) \mid \exists c > 0, n_0 \in \mathbb{N} : 0 \leq g(n) \leq c \cdot f(n) \text{ für alle } n \geq n_0\}$$

$h(n) \in O(f(n))$ bedeutet also, daß h höchstens so schnell wächst wie f.

Satz. Sei $k > 0$ eine Konstante, $h \in O(f(n))$, g eine beliebige Funktion, die für $n \geq n_0$ nichtnegativ ist. Dann gilt:

1. $k \cdot h(n), k + h(n), k - h(n) \in O(f(n))$
2. $(h\ op\ g)(n) \in O((f\ op\ g)(n))$ für $op \in \{+, -, *, /\}$
3. O ist transitiv: für alle f, g, h gilt: Wenn $f(n) \in O(g(n))$ und $g(n) \in O(h(n))$, so ist $f(n) \in O(h(n))$.

Man nennt f

konstant	falls $f \in O(1)$
logarithmisch	falls $f \in O(\log_2 n)$
linear	falls $f \in O(n)$
quadratisch	falls $f \in O(n^2)$
polynomial	falls $f \in O(n^k)$ für ein $k \geq 0$
exponentiell	falls $f \in O(k^n)$ für ein $k \geq 2$

Beispiele:

$T_{\texttt{fak1}}(n) \in O(n)$	linear,	$S_{\texttt{fak1}}(n) \in O(1)$	konstant
$T_{\texttt{binSuch}}(n) \in O(\log_2 n)$	logarithmisch,	$S_{\texttt{binSuch}}(n) \in O(1)$	konstant
$T_{\texttt{fac}}(n) \in O(n)$	linear,	$S_{\texttt{fac}}(n) \in O(n)$	linear

Außerdem gilt: Die Hintereinanderausführung zweier Anweisungen mit linearer Zeitkomplexität ist linear. Die Zeitkomplexität zweier verschachtelter Schleifen mit linearer Terminierung ist quadratisch.

Eine exponentielle Zeitkomplexität hat folgendes Problem des „Handelsreisenden“ (engl. Traveling Salesman): Gegeben sei ein Graph mit n Städten und den jeweiligen Entfernungen sowie eine Entfernung B. Gibt es eine Tour der Länge $\leq B$ durch alle Städte, sodaß jede Stadt nur einmal besucht wird?

Algorithmen mit *polynomialer* Komplexität nennt man *praktisch berechenbar*, während *exponentielle* Algorithmen *nicht* (mehr) praktisch berechenbar sind. (Grund: bei Vergrößerung der Eingabe um 1 verdoppelt sich der Aufwand.)

Bemerkung: Für das Traveling-Salesman-Problem gibt es einen nichtdeterministisch-polynomialen Algorithmus („man darf ide richtige Lösung raten“). Die Klasse der nichtdeterministisch-polynomialen Algorithmen (bzgl. der Zeitkomplexität) nennt man NP, die Klasse der polynomialen Algorithmen (bzgl. der Zeitkomplexität) nennt man P. Die bekannteste ungelöste Frage der theoretischen Informatik ist: „P = NP?“. Das Traveling-Salesman-Problem ist NP-vollständig, d.h. falls es einen polynomialen Algorithmus zu seiner Lösung gibt, so hat jeder nichtdeterministisch-polynomiale Algorithmus eine polynomiale Lösung.

10.3 Zusammenfassung

1. Der Zeitbedarf einer Anweisung berechnet sich aus der Anzahl der Berechnungsschritte, der Speicherplatzbedarf aus dem Bedarf an lokalen Variablen und (neuen) Objekten.

2. Die Zeit- und Speicherplatzkomplexität eines Algorithmus hängt von der Größe der Eingabe ab. Im schlechtesten Fall bildet man das Maximum der Berechnungsschritte für Eingaben gleicher Größe. Analog nimmt man im mittleren Fall den Durchschnitt.

3. Speicher- und Zeitkomplexität werden nach ihrer Größenordnung, der O-Notation, klassifiziert. Diese hängt im wesentlichen von der Anzahl der nötigen Schleifendurchläufe ab. Geschachtelte Schleifen erhöhen die Komplexität, im Gegensatz zu hintereindander ausgeführten Schleifen.

4. Polynomial berechenbare Algorithmen heißen auch praktisch berechenbar (obwohl schon die Komplexität n^3 häufig Probleme bereitet). Exponentielle Algorithmen sind nicht praktisch berechenbar.

5. Bis heute offen ist die Frage P=NP: ob nichtdeterministisch polynomiale Algorithmen auch polynomial sind. (Beispiel: Handelsreisender)