Komplexität von Algorithmen

Martin Wirsing

in Zusammenarbeit mit
Matthias Hölzl, Piotr Kosiuczenko, Dirk Pattinson

10/04/03

Ziele

- Zeit- und Speicherplatzbedarf einer Anweisung berechnen können
- Zeit- und Speicherplatzbedarf einer Methode berechnen können
- Unterschiede der Komplexität vom schlechtesten, besten und mittleren Fall kennen
- *O*-Notation wiederholen
- „Praktische" Berechenbarkeit eines Algorithmus einschätzen können

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 Rekursion) sowie von konkreten Maschineneigenschaften (z.B. Art und Ausführungsgeschwindigkeit der Maschinenoperationen).

Zeit- und Speicherplatzbedarf

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

Zeit- und Speicherplatzbedarf

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:	
`+`	c_+
`*`	c_*
`-1`	c_{-1}
`<`	$c_<$
exp_1 + exp_2	Zeit von exp_1+ c_+ + Zeit von exp_2
`new C(t)`	c_{call} + Zeit von `t` + T_C + c_{store}

Zeit- und Speicherplatzbedarf

Zeitbedarf T für die Deklaration von	Zeitbedarf
S_1S_2	Zeit *von* S_1 + *Zeit von* S_2
o.a	Zeit von o_{att} + *c (c = Zeit für Zugriff auf Halde)*
if (B) S_1 **else** S_2	*Zeit von if* + *Zeit von B* + *Zeit von* S_1, falls `B == true`, *Zeit von if* + *Zeit von B* + Zeit von S_2 , falls `B == false`
while (B) { S }	Anzahl der Durchläufe * (Zeit von B + Zeit von S) + Zeit von B

Zeit- und Speicherplatzbedarf

Sei die Funktions- / Methoden `type m (type x) {stms}` gegeben.

Der *Zeitbedarf eines Aufrufs m (a)* hängt ab von den Kosten der Parameterübergabe c_p, bestehen aus

- den Kosten der Berechnung von a T_a und
- den Kosten der Übergabe (des Wertes) von a c_{store}

  (die Kosten des „organisatorischen" Bedarfs zur Berechnung der Rücksprungadresse etc. werden vernachlässigt), sowie
- der Anzahl der Berechnungsschritte des Rumpfes $T_{stms[x=a]}$

ist definiert als $T_m(a) = c_{call} + T_a + T_{stms[x=a]} + c_{store}$

wobei $T_{stms[x=a]}$ die Anzahl der Berechnungsschritte des Rumpfes (in Abhängigkeit vom R-Wert von a) angibt.

Zeit- und Speicherplatzbedarf

Der Speicherplatzbedarf eines Aufrufs $m(a)$ hängt ab von

- den Kosten S_a des aktuellen Parameters,
- den organisatorischen Kosten der Parameterübergabe call S_{call} und
- den Kosten $S_{stms[x=a]}$ für die lokalen Deklarationen.

Der Speicherbedarf von $m(a)$ ist definiert als

$$S_m(a) = S_{call} + S_a + S_{stms[x=a]}$$

Bemerkung: In aktuellen Java-Implementierungen wird der zur Berechnung von a benötigte Speicherplatz gelöscht, bevor der Rumpf ausgewertet wird; d.h man verwendet das Maximum anstelle der Summe:

$$S_m(a) = \max(S_a\ ,\ S_{call} + S_{stms[x=a]})\ .$$

Zeit- und Speicherplatzbedarf

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);`

   $T_{stm} = \underbrace{c_{store}}_{\text{Zuweisung an p}} + \underbrace{2c_{fetch}}_{\text{Kosten zur Berechnung von 0,1}} + \underbrace{2c_{store}}_{\text{Übergabe der akt. Parameter}} + c_{call} + 2\cdot(c_{fetch} + c_{store})$ (mit $T_{Point}(0,1)$)

   $S_{stm} = 1 + 2$ (p, Halde)

Beispiele für den Zeit- und Speicherplatzbedarf des Rumpfs einer Methode

Beispiel 1: Iterative Fakultät (Zeitbedarf)

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

```
int fak1(int x)                    T_fak1(x) | =
{ int result = 1;                  c_fetch + c_store                 // Kosten der
                                                                     // Vorbesetzung
   while (x > 0) {                 (c_1
      result = x * result;          + c_2                            // Kosten der Schleife
      x = x - 1;                    + c_3) * x +                     // x – Anzahl der Dl.
      }                             + c_1
   return result;                   + c_fetch + c_return             // Kosten von return
}
```


Beispiele für den Zeit- und Speicherplatzbedarf des Rumpfs einer Methode

Beispiel 1: Iterative Fakultät (**Speicherplatzbedarf**)

$S_{fak1}(x) = s_{call} + \underbrace{1}_{x} + \underbrace{1}_{result} + \underbrace{1}_{return}$ // Kosten der lokalen // Variablen

Beispiele für den Zeit- und Speicherplatzbedarf des Rumpfs einer Methode

Beispiel 2: Binäre Suche

```
class SearchArray                                   T_binSuch(e)≤
{  char[] arr;                                      2(c_fetch + c_store) + c_-1 + c_fetch + 2c_att
   boolean binSuch(char e)                                                  // Vorbesetzung
   {   int left = 0;
       int right = this.arr.length - 1;
       int mid;
       boolean found = false;                       + c_fetch + c_store     // Vorbesetzung
                                                    + ⌈log2a.length⌉ *
       while ((left <= right) & !found) {           ((3c_fetch + c_< + c_&) //Bedingung
         mid = (left + right) / 2;                  + c_store + c_fetch? + c_/ + c_* // mid=...
         if (e < this.arr[mid]) right = mid - 1;    +2·(3c_fetch + 2c_att + max{c_<, c_>})
         else if (e > this.arr[mid]) left = mid + 1; //2mal Bedingung
         else found = true;                         + c_fetch + c_store + max{c_+1, c_-1})
                                                                            //Ja-, Nein-Zweig
       }
       return found;                                + c_return + c_fetch    // Kosten von return
   }
}
                                                    + (3c_fetch + c_< + c_1 + c_&)
                                                    S_binSuch(e) = 4// Kosten der lokalen Variablen
```

Beispiele für den Zeit- und Speicherplatzbedarf des Rumpfs einer Methode

Beispiel 3: Rekursive Fakultät

Anweisung	Zeit $T_{fac}(n)$ `n == 0`	`n > 0`
`1 public static int fac(int n)`		
`2 {`		
`3 if (n == 0)`	$c_{ife} + 2c_{fetch}$	$c_{ife} + 2c_{fetch} + c_{==}$
`4 return 1;`	$c_{fetch} + c_{return}$	----
`5 else`		
`6 return n * fac(n - 1);`	----	$2c_{fetch} + c_{-} + c_{store} + c_{*} + c_{call} + c_{return} + T_{fac}(n-1)$
`7 }`		

Beispiele für den Zeit- und Speicherplatzbedarf des Rumpfs einer Methode

Beispiel 3: Fortsetzung

Also

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

mit $t_1 = c_{ife} + 2c_{fetch} + c_{<} + c_{return}$ und $t_2 = c_{ife} + 2c_{fetch} + c_{-} + c_{store} + c_{*} + c_{call} + c_{return}$

Beispiele für den Zeit- und Speicherplatzbedarf des Rumpfs einer Methode

Beispiel 3: Fortsetzung

Die Lösung dieser Gleichung erhält man durch wiederholte Substitution:

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

d.h.

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

Im Vergleich dazu erhält man für die iterative Berechnung ???

$$T_{fak1}(n) = t'_1 + n \cdot t'_2$$

mit $t'_1 = c_{ife} + 3c_{fetch} + c_{store} + c_{<} + c_{return}$ und $t'_2 = c_{ife} + 5c_{fetch} + 2c_{store} + c_{*} + c_{-1} + c_{>}$

D.h. Die Laufzeiten unterscheiden sich nicht prinzipiell (siehe später).

Beispiele für den Zeit- und Speicherplatzbedarf des Rumpfs einer Methode

Beispiel 3: Rekursive Fakultät

Anweisung	Speicher `n == 0`	`n > 0`
`1 public static int fac(int n)`	$1 + 1 + s_{call}$	$1 + 1 + s_{call} + s_{fac}(n-1)$
`2 {`		
`3 if (n == 0)`		
`4 return 1;`		
`5 else`		
`6 return n * fac(n - 1);`		
`7 }`		

Beispiele für den Zeit- und Speicherplatzbedarf des Rumpfs einer Methode

Wieder ergibt sich

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

Folgerung: $S_{rec}(n) = S_1 + n \cdot S_2$

Während der Speicherplatzbedarf von `fac1` gleich $3 + S_{call}$, also konstant, ist, hängt der Speicherplatzbedarf der rekursiven Funktion von der Größe von *n* ab und wächst linear mit *n*.

Beispiel: Suche im (ungeordneten) Feld

```
class SearchArray
{  int[] arr;
   ...
   boolean such(int e)                          c_call + T_e +
 {    int k = this.arr.length;                  c_store + 2c_att + c_fetch
      int i = 0                                 + c_store + c_fetch

      while (i < k & !(arr[i] == e))            + k*(2c_fetch + c_< + 2c_fetch + 2c_att + c_== + c_& +c_!

        i = i + 1;                              + c_store + c_fetch + c_+1 )

      return (i < k);                           + 2c_fetch + c_< + c_return}
 }
```


Komplexitätsarten

- 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)

Kiomplexität im schlechtesten, besten und mittleren Fall

Zeitkomplexität im

schlechtesten Fall	$T^w_m(n) = max\{T_m(a) \mid \text{Größe von } a = n\}$
mittleren Fall	$T^{av}_m(n) = \text{Durchschnitt von}\{T_m(a) \mid \text{Größe von } a = n\}$
besten Fall	$T^b_m(n) = min\{T_m(a) \mid \text{Größe von } a = n\}$

Speicherkomplexität im

schlechtesten Fall	$S^w_m(n) = max\{S_m(a) \mid \text{Größe von } a = n\}$
mittleren Fall	$S^{av}_m(n) = \text{Durchschnitt von}\{S_m(a) \mid \text{Größe von } a = n\}$
besten Fall	$S^b_m(n) = min\{S_m(a) \mid \text{Größe von } a = n\}$

Beispiele für den Zeit- und Speicherplatzbedarf des Rumpfs einer Methode

Beispiel:

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

da die Größe von $e = 1$.

- Für `fac, fak1` stimmen der beste, schlechteste und mittlere Fall jeweils überein.

Beispiel:

- $T^w_{such}(\textbf{this}, e) \leq t_1 + n \cdot t_2$,
- $S_{such}(\textbf{this}, e) =$ konstant

Beispiele für den Zeit- und Speicherplatzbedarf des Rumpfs einer Methode

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

  wenn man das arithmetische Mittel als Durchschnitt wählt, wobei Größe von (`this, e`) = Länge von `this.arr`
- $T^b_{such}(n) = min\{T_{such}(\textbf{this}, e) \mid \text{Größe von } (\textbf{this}, e) = n\}$
  $= t_1 + t_2$

Größenordnung der Komplexität: Die O-Notation

Die Komplexität $T(n)$ bzw. $S(n)$ wird häufig nur bis auf konstante Faktoren untersucht. Dazu ordnen wir den Aufwandsfunktionen $T(n)$ und $S(n)$ bestimmte Funktionsklassen ihrer „Größenordnung" zu.

Sei $f : \mathbb{N} \to \mathbb{N}$ eine Funktion. Wir definieren die Klasse:

$O(f(n)) =$

$\{g(n) \mid$ es gibt $c > 0$, $n_0 : 0 \leq g(n) \leq c \cdot f(n)$ für alle $n \geq n_0\}$

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

Größenordnung der Komplexität: Die O-Notation

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

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

Größenordnung der Komplexität: Die O-Notation

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 $k \geq 2$

Größenordnung der Komplexität: Die O-Notation

Beispiele:

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

Außerdem gilt:

- Die Hintereinanderausführung zweier Anweisungen mit linearer Zeitkomplexität ist linear.
- Die Zeitkomplexität zweier verschachtelter Schleifen mit linearer Terminierung ist quadratisch.

Exponentielle und polynomiale Zeitkomplexität

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, so daß jede Stadt einmal besucht wird?

Bemerkung:

Für das Traveling-Salesman-Problem gibt es einen nichtdeterministisch-polynominalen Algorithmus („man darf die richtige Lösung raten“).

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.

Exponentielle und polynomiale Zeitkomplexität

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.)

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?“.

Zusammenfassung

- Der Zeitbedarf einer Anweisung berechnet sich aus der Anzahl der Berechnungsschritte, der Speicherplatzbedarf aus dem Bedarf an lokalen Variablen und (neuen) Objekten.
- 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.
- 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 hintereinander ausgeführten Schleifen.
- Polynomial berechenbare Algorithmen heißen auch praktisch berechenbar (obwohl schon die Komplexität n^3 häufig Probleme bereitet). Exponentielle Algorithmen sind nicht praktisch berechenbar.
- Bis heute offen ist die Frage P=NP: ob nichtdeterministisch polynomiale Algorithmen auch polynomial sind. (Beispiel: Handelsreisender)