Komplexität von Algorithmen

Martin Wirsing

in Zusammenarbeit mit
Michael Barth, Philipp Meier und Gefei Zhang

01/05

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 kennenlernen
- „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

Der Einfachheit halber zählen wir

- beim **Zeitbedarf**
  - die Anzahl der Anweisungen, **oder**
  - die Anzahl der (zeitintensiven) Operationen

  z.B. bei der Berechnung des Minimums einer Reihung die Anzahl der notwendigen Vergleiche.

- beim **Speicherplatzbedarf** die maximale Anzahl der während der Berechnung benötigten Speicherplätze für (neue) lokale Variablen und (neuen) Objekte,
  z.B. bei der Berechnung des Minimums einer Reihung
  die lokalen Variablen für die Parameterübergabe, für die Zwischenspeicherung des Minimums und für die Laufvariable der Schleife.

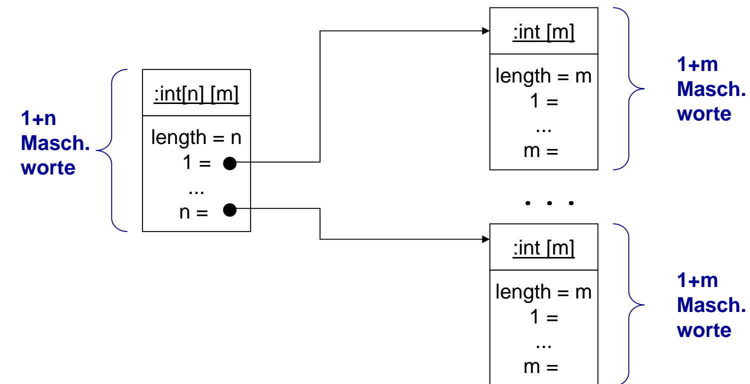
Speicherplatzbedarf

Konstrukt	Speicherplatzbedarf S
Jedes Element vom **Grundtyp oder Objekttyp** (Objektreferenz) belegt im **Keller**	1 Maschinenwort
Also	
Deklaration einer lokalen Variable	1 Maschinenwort (im Keller)
Formaler Parameter	1 Maschinenwort (im Keller)
Erzeugung eines Objekts	Summe des Speicherplatzbedarfs der Attribute
Erzeugung einer Reihung vom Typ `type`	1+ (Länge des Feldes * Bedarf für 1 Wert vom Typ `type`)
z.B.	
▪. einstufige **`int`**-Reihung	1 + Länge der Reihung viele Maschinenworte
▪ zweistufige Reihung **`int`** `[n][m]`	`1 + n + ((1+m) * n)` viele Maschinenworte

Für die Speicherung der Länge des Feldes

Speicherplatzbedarf: 2-dim. Reihung

Der Speicherplatzbedarf einer zwei-stufigen Reihung vom Typ **`int`** `[n][m]` beträgt `1 + n + ((1+m) * n)` Maschinenworte:



Speicherplatzbedarf

Beispiele:

1. `Point p = new Point (0,1);`

   Speicherplatzbedarf : `1 Maschinenwort` für `p`
   `2 Maschinenworte` für **`new`** `Point(0,1)`

2. **Speicherplatzbedarf :**

```
int x = 5;                          1
int[] myArray = new int [6*x];   1 + (1+30)
int laenge = myArray.length;        1
myArray[0]=1;                       0
for (int k=1; k<laenge; k++)        1
    myArray[k] = 2*myArray[k-1];  0
```

Also **Speicherplatzbedarf :** `35 Maschinenworte`

Zeitbedarf

Konstrukt	Zeitbedarf
`x = exp;`	`1` (falls Anweisungen gezählt werden und `exp` kein Methodenaufruf)
S_1 S_2	Zeit von S_1 + Zeit von S_2
`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 }`	Zeit von `B` + Anzahl der Durchläufe * (Zeit von `B` + Zeit von `S`)
`for``(`**`int`** `k=A; B; k++) { S }`	Zeit von`(`**`int`** `k=A;)` + Zeit von `B` + Anzahl der Durchläufe * (Zeit von `B` + Zeit von `S` + Zeit von `k++`)

Bem.: Zur Vereinfachung wird der Zeitbedarf von Konstruktoren vernachlässigt.

Zeitbedarf

Beispiele:

Zeitbedarf :

1. `Point p = new Point (0,1);` — 1

2.
```
int x = 5;                                  1
int[] myArray = new int [6*x];              1
int laenge = myArray.length;                1
myArray[0]=1;                               1
for (int k=1; k<laenge; k++)             1 + 1 + 29*3
    {   myArray[k] = 2*myArray[k-1];
    }
```
Also **Zeitbedarf :** 93 Zeiteinheiten (bzw. Berechnungsschritte)

Zeitbedarf von Methodenaufrufen

Sei die Methode `type m(type1 x) {stms}` der Klasse `C` gegeben.

- Der **Zeitbedarf** *eines Aufrufs o.m (a)* berechnet sich aus der Summe
  - der Kosten der Parameterübergabe,
    d.h. den Kosten der Berechnung von `this=o; x=a;`
    und organisatorischen Kosten (die wir außer Acht lassen)
  - und dem Zeitbedarf für die Ausführung des Rumpfes
- Der **Speicherplatzbedarf** *eines Aufrufs o.m (a)* berechnet sich aus der Summe
  - der Kosten der Parameterübergabe,
    d.h. dem Speicherplatzbedarf von `C this = o; type1 x = a;`
    und organisatorischen Kosten (die wir außer Acht lassen)
  - und dem Speicherplatzbedarf für die Ausführung des Rumpfes

Zeit- und Speicherplatzbedarf von Methodenaufrufen

Beispiel:
```
class C
{   int[]     m (int x)
    {   int[] myArray = new int [6*x];
        int laenge = myArray.length; myArray[0]=1;
        for (int k=1; k<laenge; k++)
              myArray[k] = 2*myArray[k-1];
        return myArray;
    }
    public static void main(String[] 
    {   C c = new C();
        c.m(5);
    }
}
```
Kosten der Parameterübergabe

Zeitbedarf von `c.m(5);` : 2+ 5 + (6*5-1)*3 +1 = 95 Berechnungsschritte
Speicherplatzbedarf von `c.m(5);` : 2 + (1+(1+30)) + 1+1 = 36 Speicherplätze

Zeit- und Speicherplatzbedarf von Methoden

Sei die Methode `type m() {stms}` gegeben.

- Der Zeit- und Speicherplatzbedarf einer Methode ist

  abhängig vom aufrufenden Objekt (und den aktuellen Parametern – sofern vorhanden).

Bem. : Dies gilt auch für Konstruktoren, bei denen wir aber zur Vereinfachung den Zeitbedarf vernachlässigen.

Zeit- und Speicherplatzbedarf von Methoden

Beispiel:

```
class C
{   int[] m (int x)
    {   int[] myArray = new int [6*x];
        int laenge = myArray.length; myArray[0]=1;
        for (int k=1; k<laenge; k++)
              myArray[k] = 2*myArray[k-1];
    }
    ...
}
```

Zeitbedarf von `o.m(x);`: 2 + 5 + (6*x-1)*3 +1= 5 +18x Berechnungsschritte
Speicherplatzbedarf von `o.m(x);`: 2 + (1+(1+6*x)) + 1+1 = 6 +6x Speicherplätze

hängt von x und o ab

Beispiel: Suche im (ungeordneten) Feld

```
class SA                                                  Zeitbedarf      Speicherplatzbedarf
{  int[] arr;                             (Zeitberechnung nach Anzahl der Vergleiche)
   public SA(int[]a){ arr = a; }
   boolean such(int e)
   {   int k = this.arr.length;                             0                1
       int i = 0;                                           0                1
       while (i < k && !(arr[i] == e))i++;                  ≤ 1 + k*2        0
       return (i < k);}                                     1                1
   public static main (String[] args)
   {   int a1 = {3,5,7,9}, a2 = {2,4,6,8}, a3 = {7,3,9,5};
       SA o1 = new SA(a1); SA o2 = new SA(a2); SA o3 = new SA(a3); }
}
```

Zeitbedarf für

`o1.such(3);`: 3 `o2.such(3);`:10 `o3.such(3);`: 5

Speicherplatzbedarf für

`o1.such(3);`: 2+3 `o2.such(3);`:2+3 `o3.such(3);`: 2+3

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.
- Der Algorithmus zum Suchen eines Elements in einer Reihung liefert für Reihungen gleicher Länge unterschiedliche Kosten bei der Zeit.
- Um solche Unterschiede abschätzen zu können, unterscheidet man die

  Komplexität im schlechtesten, mittleren und besten Fall

  (engl. worst case, average case, best case complexity).

Komplexität im schlechtesten, besten und mittleren Fall

Sei die Methode `type m() {stms}` gegeben und bezeichne
$T_m(o)$ den Zeitbedarf von `o.m();` und **$S_m(o)$** den Speicherplatzbedarf von `o.m();` .

Zeitkomplexität im

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

Speicherkomplexität im

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

Beispiele für den Zeit- und Speicherplatzbedarf des Rumpfs einer Methode

Suchen in einer Reihung:

Wir wählen als Größe der aktuellen Parameter o, e die Größe der Reihung von o.

- $T^{w}_{such}(n) = 2 + 2n$,
- $T^{av}_{such}(n) = \text{Durchschnitt von } \{T_{such}(o, e) \mid \text{Größe von } (\mathbf{o}, e) = n\}$

$$= 2 + \frac{\sum^{n}_{i=1} 2i}{n} = 2 + \frac{2(n+1)n}{2\,n} = 2 + (n+1)$$

wenn man das arithmetische Mittel als Durchschnitt wählt.

- $T^{b}_{such}(n) = \min\{T_{such}(\mathbf{o}, e) \mid \text{Größe von } (\mathbf{o}, e) = n\} = 2 + 1$

Beispiele für den Zeit- und Speicherplatzbedarf einer Methode

Beispiel: Binäre Suche in geordneter Reihung (Zeitberechnung nach **Anzahl der Vergleiche**)

```
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;
   }
}
```

$T_{binSuch}(this, e) \le$

- Deklarationen (left, right, mid, found): 0 // Vorbesetzung
- while: $1 + \lceil \log_2 \text{arr.length} \rceil$ * (1 //Bedingung
- Rumpf: + 2 //2mal Bedingung)
- return: 0 // Kosten von `return`

$= 0 + 1 + 3 \lceil \log_2 \text{arr.length} \rceil$

$S_{binSuch}(this, e) = 2+4$ // Kosten der Parameterübergabe und der lokalen Variablen

Beispiele für den Zeit- und Speicherplatzbedarf des Rumpfs einer Methode

Binäre Suche Fortsetzung:

- $T^{w}_{binSuch}(n) = \max\{T_{binSuch}(this, e) \mid \text{Größe von } this.arr = n)\}$

  $= 1 + 3 \lceil \log_2 (n) \rceil$

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 $f : N \to N$ eine Funktion. Wir definieren die Klasse $\mathbf{O(f(n))}$ der Funktionen, die höchstens so schnell wachsen wie f(n):

$O(f(n)) =$

$\{g(n) \mid \text{es gibt } c>0, n_o: \ 0 \le g(n) \le c*f(n) \text{ für alle } n \ge n_o\}$

$h(n) \in O(f(n))$ bedeutet also, daß h höchstens so schnell wächst wie f (modulo linearer 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 $n \geq n_o$ 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:

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

Außerdem gilt:

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

Exponentielle und polynomiale Zeitkomplexität

Für folgendes Problem des „Handelsreisenden“ (engl. Traveling Salesman) sind nur Algorithmen mit exponentieller Zeitkomplexität bekannt:

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**.
Das bekannteste ungelöste Problem der theoretischen Informatik ist:die Frage **„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)