Einführung in Algorithmen

Theorie (L)

Version vom 19. Februar 2021

Inhaltsverzeichnis

1 Der Algorithmusbegriff

Eine erste Umschreibung

Robert Sedgewick und Kevin Wayne umschreiben in ihrem Buch über Algorithmen und Datenstrukturen[1] auf Seite 20 den Algorithmusbegriff wie folgt:

> „Ein Computerprogramm zu schreiben bedeutet im Allgemeinen nichts anderes, als ein Verfahren zu implementieren, das zuvor dafür entwickelt wurde, ein bestimmtes Problem zu lösen. Dieses Verfahren ist meistens unabhängig von der eingesetzten Programmiersprache – mit grosser Wahrscheinlichkeit dürfte es für viele Computer und viele Programmiersprachen geeignet sein. Und es ist das Verfahren und nicht das Computerprogramm, welches die Schritte vorgibt, mit denen wir das Problem lösen können.
>
> Lösungsverfahren, die
>
> - endlich,
> - deterministisch und
> - effektiv
>
> sind und als Computerprogramme implementiert werden können, werden in der Informatik als *Algorithmen* bezeichnet. . . . “

Historisches zum Algorithmusbegriff

Der Begriff Algorithmus ist eng mit dem Namen des Gelehrten

Abu Abdallah Muhammed ibn Musa al-Hwarizimi al-Magusi (Al-Hwarizimi)

verbunden, der um etwa 800 n. Chr. im „Haus der Weisheit“ in Bagdad tätig war. Das Haus der Weisheit war eine Art Akademie, wo Gelehrte aus verschiedenen Kulturen unter anderem wissenschaftliche Werke der Antike (Platon, Aristoteles, Euklid, . . .) vom Griechischen ins Arabische übersetzten.

Von Al-Hwarizimi stammen mehrere Mathematikbücher, in denen das aus Indien stammende dezimale Stellenwertsystem sowie das Rechnen damit beschrieben wird.

Originale von Al-Hwarizimi sind nich überliefert. Dafür lateinsche Übersetzungen, in denen das Werk Buch des Algorithmus (oder auch Buch des Algorismus) genannt wird. Im Laufe der Zeit wurde der Name des Autors immer mehr zu einer Bezeichnung für das von ihm beschriebene Rechnen mit den arabischen (bzw. indischen) Ziffern. So wandelte sich der Begriff zur Bezeichnung für ein automatisierbares Rechenverfahren.

[1] R. Sedgewick, K. Wayne: (2014) *Algorithmen.* 4. Auflage. Pearson.

2 Der Algorithmus von Euklid

Einleitung

Der grösste gemeinsame Teiler (ggT) von zwei natürlichen Zahlen a und b ist die grösste natürliche Zahl, die sowohl a als auch b teilt.

(a) $\text{ggT}(21, 15) = 3$

(b) $\text{ggT}(14, 0) = 14$

(c) $\text{ggT}(-6, -8) = 2$

Definitionen

Man sagt, die ganze Zahl d teilt die ganze Zahl a, wenn a ohne Rest durch d teilbar ist, d. h. wenn $\text{mod}(a, d) = 0$. In diesem Fall schreibt man kurz $d \mid a$ [*sprich:* „d teilt a“]. Andernfalls ist a nicht (ohne Rest) durch d teilbar und man schreibt $d \nmid a$ [*sprich:* „d teilt a nicht“].

Lösungsidee 1

Man könnte beide Zahlen $a \geq b$ in einer Schleife jeweils durch die absteigende Folge b, $b-1$, $b-2$, …, 3, 2, 1 dividieren. Die erste dieser Zahlen, die *a und* b teilt, muss der grösste gemeinsame Teiler sein.

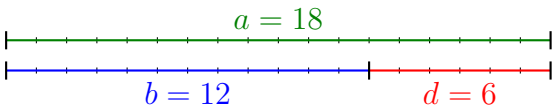
Beispiel: $\text{ggT}(12, 8)$

$8 \mid 12$	und	$8 \mid 8$	falsch
$7 \mid 12$	und	$7 \mid 8$	falsch
$6 \mid 12$	und	$6 \mid 8$	falsch
$5 \mid 12$	und	$5 \mid 8$	falsch
$4 \mid 12$	und	$4 \mid 8$	wahr

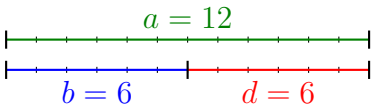
Also ist 4 der grösste gemeinsame Teiler von 12 und 8.

Lösungsidee 2

Wenn $t = \text{ggT}(18, 12) \quad \Leftrightarrow \quad t = \text{ggT}(12, 6)$



$t = \text{ggT}(12, 6) \quad \Leftrightarrow \quad t = \text{ggT}(6, 6)$



$t = \text{ggT}(6, 6)$

Idee: Euklid von Alexandria (griechischer Mathematiker, ca. 3. Jh. v. Chr.)

Die Berechnungsvorschrift (Version 1)

Wie kann diese Anforderung von einer Rechenmaschine gelöst werden?

Duch die folgende Sequenz von Anweisungen:

(1) Input: natürliche Zahlen a und b ein.

(2) Falls $a = b$: Wir sind fertig und geben $\text{ggT}(a, b) = a$ aus.

(3) Falls $a < b$: Vertausche die Werte von a und b.

(4) Berechne $(a - b) \to a$ und gehe zu (2).

Eine solche Formulierung wird Algorithmus genannt.

Ein *Algorithmus* ist eine in der Beschreibung und Ausführung endliche und eindeutige Vorschrift zur Lösung einer Klasse von Problemen.

Beispiel 1

$\text{ggT}(21, 15)$

(1) $a = 21$, $b = 15$

(2) $a = b$? *falsch*

(3) $a < b$? *falsch*

(4) $a - b = 6 \to a$

$a = 6$, $b = 15$

(2) $a = b$? *falsch*

(3) $6 < 15$ *wahr* $\Rightarrow (a, b) = (15, 6)$

(4) $15 - 6 = 9 \to a$

$a = 9$, $b = 6$

(2) $a = b$? *falsch*

(3) $a < b$? *falsch*

(4) $a - b = 3 \to a$

$a = 3$, $b = 6$

(2) $a = b$? *falsch*

(3) $a < b$? *wahr* $\Rightarrow (a, b) = (6, 3)$

(4) $6 - 3 = 3 \to a$

$a = 3$, $b = 3$

(2) $3 = 3$? *wahr*

Ende

$\text{ggT}(21, 15) = 3$

Beispiel 2 (Spezialfall 1)

$\text{ggT}(14, 0)$

(1) $a = 14,\ b = 0$
(2) $14 = 0$? *falsch*
(3) $14 < 0$? *falsch*
(4) $14 - 0 = 14 \rightarrow a$
$[a = 14,\ b = 0]$

gleiche Situation wie im letzten Schritt $\Rightarrow$ Endlosschleife

Lösung

$\text{ggT}(7, 0) = 7$
$\text{ggT}(0, 125) = 125$
$\text{ggT}(0, 0) =$ nicht definiert (warum?)

Beispiel 3 (Spezialfall 2)

$\text{ggT}(-6, -8)$

(1) $a = -6,\ b = -8$
(2) $-6 = -8$? *falsch*
(3) $-6 < -8$? *falsch*
(4) $-6 - (-8) = 2 \rightarrow a$
$[a = 2,\ b = -8]$

(2) $2 = -8$? *falsch*
(3) $2 < -8$? *falsch*
(4) $2 - (-8) = 10 \rightarrow a$
$[a = 10,\ b = -8]$

a wird immer grösser und b bleibt konstant -8 $\Rightarrow$ Endlosschleife

Lösung

Berechne den Algorithmus mit den Beträgen (Absolutwerten) von a und b.

Die Berechnungsvorschrift (Version 2)

(1) Input: natürliche Zahlen a und b ein.

(1.1) $\text{abs}(a) \to a$ und $\text{abs}(b) \to b$

(1.2) Falls $a = 0$ und $b = 0$: Fehlermeldung

(1.3) Falls $a = 0$: Gib b aus

(1.4) Falls $b = 0$: Gib a aus

(2) Falls $a = b$: Wir sind fertig und geben $\text{ggT}(a, b) = a$ aus.

(3) Falls $a < b$: Vertausche die Werte von a und b.

(4) Berechne $(a - b) \to a$ und gehe zu (2).

Aufgabe

Implementiere den Algorithmus von Euklid (Version 2) als Python-Funktion `euklid(a,b)`, die $\text{ggT}(a, b)$ oder `None` zurückgibt.

Beispiel 4

Überlege, wie der Algorithmus mit der Aufgabe $\text{ggT}(1000, 1)$ verfährt.

Das Programm rechnet aufwändig:

$\text{ggT}(999, 1)$

$\text{ggT}(998, 1)$

...

$\text{ggT}(1, 1)$

Lösung: Modulo-Operator (`%`) statt fortgesetztes Subtrahieren

`a % b == 0` bedeutet, dass `a` restlos durch `b` teilbar ist.

Die Berechnungsvorschrift (Version 3)

(1) Input: natürliche Zahlen a und b ein.

(1.1) $\text{abs}(a) \to a$ und $\text{abs}(b) \to b$

(1.2) Falls $a = 0$ und $b = 0$: Fehlermeldung

(1.3) Falls $a = 0$: Gib b aus

(2) Falls $b = 0$: Wir sind fertig und geben $\text{ggT}(a, b) = a$ aus.

(3) Berechne $(b, a \,\%\, b) \to (a, b)$ und gehe zu (2).

Aufgabe 2

Implementiere den Algorithmus von Euklid (Version 3) als Python-Funktion `euklid2(a,b)`.

3 Laufzeitanalyse

Hat man einen Algorithmus implementiert, möchte man gerne wissen, wie gross der Zeit- und der Speicherbedarf für eine bestimmte Eingabe ist.

Da dies jedoch in den meisten Fällen sehr aufwändig oder gar unmöglich ist, begnügt man sich damit, den Zeit- oder den Speicheraufwand in Abhängigkeit der Anzahl der Eingabegrössen n auszdrücken.

Die Beschreibung der Laufzeit als Funktion $f(n)$ soll ...

- abhängig von der Anzahl n der Eingabdaten sein,
- unabhängig von der Programmiersprache oder der Hardware sein,
- den ungünstigsten Fall („worst case“) darstellen.

4 Die $\mathcal{O}$-Notation

Die $\mathcal{O}$-Notation (engl.: *Big-Oh-Notation*) ist ein Hilfsmittel zur mathematischen Beschreibung der Laufzeit eines Algorithmus.

Definition

$\mathcal{O}(f(n))$ ist die Menge aller Funktionen $g(n)$, für die es eine Konstante c und eine Schranke N gibt, so dass $g(n) \leq c \cdot f(n)$ für alle $n \geq N$.

oder etwas umgangssprachlicher:

$\mathcal{O}(f(n))$ ist die Menge aller Funktionen $g(n)$, die ab einem bestimmten n nicht schneller wachsen als die Funktion $c \cdot f(n)$, wobei c eine frei wählbare Kontante ist.

Beispiel 1

$g(n) = 10n$

$g(n) = 10n \leq 10 \cdot n \quad$ (für alle $n \geq 1$)

$\Rightarrow \quad g(n) \in \mathcal{O}(n)$

Beispiel 2

$g(n) = 3n + 2$

$g(n) = 3n + 2 \leq 3n + 2n = 5n \quad$ (für alle $n \geq 1$)

$\Rightarrow \quad g(n) \in \mathcal{O}(n)$

Beispiel 3

$g(n) = 2n^2$

$g(n) = 2n^2 \leq 2 \cdot n^2 \quad$ (für alle $n \geq 1$)

$\Rightarrow \quad g(n) \in \mathcal{O}(n^2)$

Beispiel 4

$g(n) = n^2 + 2n$

$g(n) = n^2 + 2n \leq n^2 + 2n^2 = 3n^2 \quad$ (für alle $n \geq 1$)

$\Rightarrow \quad g(n) \in \mathcal{O}(n^2)$

Beispiel 5

$g(n) = 5n^4 + 4n^3 + 3n^2 + 2n + 1$

$g(n) \leq 5n^4 + 4n^4 + 3n^4 + 2n^4 + n^4 = 15n^4 \quad$ (für alle $n \geq 1$)

$\Rightarrow \quad g(n) \in \mathcal{O}(n^4)$

Beispiel 6

$g(n) = |\sin(n)|$

$g(n) = |\sin(n)| \leq 1 = 1 \cdot 1 \quad$ (für alle $n \geq 1$)

$\Rightarrow \quad g(n) \in \mathcal{O}(1)$

Beispiel 7

$g(n) = 2^{n+1}$

$g(n) = 2^{n+1} = 2^n \cdot 2^1 = 2 \cdot 2^n \quad$ (für alle $n \geq 1$)

$\Rightarrow \quad g(n) \in \mathcal{O}(2^n)$

Rechenregeln

Ist $g_1(n) \in \mathcal{O}\big(f_1(n)\big)$ und $g_2(n) \in \mathcal{O}\big(f_2(n)\big)$, so gilt:

- $g_1(n) + g_2(n) \in \mathcal{O}\big(\max(f_1(n), f_2(n))\big)$
- $g_1(n) \cdot g_2(n) \in \mathcal{O}\big(f_1(n) \cdot f_2(n)\big)$

5 Wichtige Klassen von Laufzeitkomplexitäten

Konstante Laufzeit

$\mathcal{O}(1)$

- Wert in einer Liste lesen/schreiben
- Zwei Zahlen multiplizieren

Logarithmische Laufzeit

$\mathcal{O}(\log n)$

- Suche in einer sortierten Liste

Lineare Laufzeit

$\mathcal{O}(n)$

- Suche in einer unsortierten Liste

Log-Lineare Laufzeit

$\mathcal{O}(n \cdot \log n)$

- fortgeschrittene Sortieralgorithmen

Quadratische Laufzeit

$\mathcal{O}(n^2)$

- „naive“ Sortieralgorithmen

Kubische Laufzeit

$\mathcal{O}(n^3)$

- Matrizenmultiplikation

Polynomielle Laufzeit

$\mathcal{O}(n^p)$

- Simplex-Algorithmus (lineare Optimierung)

Exponentielle Laufzeit

$\mathcal{O}(2^n)$

- Rucksackproblem

Faktorielle Laufzeit

$\mathcal{O}(n!)$

- Problem des Handlungsreisenden

Bemerkung:

Algorithmen mit $\mathcal{O}(n^4)$ und höher benötigen bei einer Verdoppelung der Inputgrösse bereits die 16-fache Laufzeit, was problematisch ist. Algorithmen mit $\mathcal{O}(2^n)$ oder höher sind praktisch unbrauchbar.

Beispiel 8

Welche Laufzeitkomplexität hat das Python-Programmfragment?

```
1  s = 1
2  for i in range(0, n):
3      for j in range(0, n):
4          for k in range(0, n):
5              s = i + j + k
```

Zeile	Kosten	Anzahl
1	c_1	1
2	c_2	n
3	c_3	n^2
4	c_4	n^3
5	c_5	n^3

$T(n) = c_1 + c_2 n + c_3 n^2 + c_4 n^3 \in \mathcal{O}(n^3)$