2

Aufwandsabschätzung von Algorithmen

Kauft man sich ein technisches Gerät (z. B. einen Anrufbeantworter), so schlägt man zunächst die beiliegende Bedienungsanleitung auf und hält sich dann – so gut man kann – an die Anweisungen, die darin zur Inbetriebnahme des Geräts beschrieben sind. Ist die Anleitung gut, so muss man sie nur einmal durchlesen und alle Schritte nur einmal ausführen. Danach funktioniert das Gerät: Der Anrufbeantworter steht anrufaufnahmebereit da.[2] Eine gute Anleitung ist ein Algorithmus.

2.1 Algorithmen

Algorithmus

Ein *Algorithmus*[3] ist also ein schrittweises Verfahren zum Lösen eines Problems, das „mechanisch“ ausgeführt werden kann, zum Beispiel auch durch einen Computer. Die einzelnen Schritte sind verständlich und nachvollziehbar. In der Informatik werden Probleme, die mittels Computer zu lösen sind, durch die Art der erlaubten Eingaben und die Art der geforderten Ausgaben spezifiziert. Der Entwurf von Algorithmen ist oft eher eine Kunst als eine Wissenschaft.[4]

Algorithmen für Computer werden in Programmiersprachen formuliert, und die so erstellten Programme werden mit einem Übersetzungsprogramm (englisch: *compiler*) in für Computer verständliche Elementaroperationen (Maschinenbefehle) umgewandelt. Ein abstraktes, formales Algorithmenmodell, das wegen seiner Einfach-

[2] Ist Ihnen das schon einmal passiert?

[3] Das Wort *Algorithmus* ist eine latinisierte Abwandlung des Namens von Muhammed al-Chwarizmi (etwa 783-850), dessen arabisches Lehrbuch über das Rechnen mit „indischen Ziffern“ (um 825) in der mittelalterlichen lateinischen Übersetzung mit den Worten Dixit Algorizmi („Algorizmi hat gesagt“) begann und das (indisch-)arabische Zahlensystem in der westlichen Welt einführte. Beispielsweise legte dieses Werk die Grundlage für das Buch von Adam Ries über die „Rechenung auff der linihen und federn in zal / maß und gewicht“ (1522).

[4] Die beeindruckendsten Werke über die Kunst der Programmierung sind die Bücher der Serie „The Art of Computer Programming (TAOCP)“ von Donald Knuth, siehe [Knu97, Knu98a, Knu98b].

F. Gurski et al., *Exakte Algorithmen für schwere Graphenprobleme*, eXamen.press,
DOI 10.1007/978-3-642-04500-4_2,

heit besonders in der Berechenbarkeits- und Komplexitätstheorie üblich ist, ist das der *Turingmaschine*, siehe z. B. [Rot08]. Hier verzichten wir jedoch auf diesen Formalismus und beschreiben Algorithmen stattdessen informal und gut verständlich.

Turingmaschine

Eine Anleitung befolgt man, um am Ende ein funktionierendes Gerät vor sich zu haben. Einen Algorithmus führt man aus, um ein Ergebnis zu erhalten, d. h., der Algorithmus hält (*terminiert*) und gibt eine Lösung für die jeweilige Probleminstanz aus, sofern eine existiert. Gibt es keine Lösung der gegebenen Probleminstanz, so hält ein Algorithmus für das betrachtete Problem ebenfalls, verwirft die Eingabe jedoch, d. h., die Ausgabe liefert die Information, dass diese Eingabe keine Lösung hat.

Allerdings gibt es nicht für alle mathematisch beschreibbaren Probleme Algorithmen, die sie lösen. Denn selbstverständlich müssen Algorithmen nicht immer – nicht bei jeder Eingabe – halten. Das Problem, für einen gegebenen Algorithmus A und eine gegebene Eingabe x zu entscheiden, ob A bei x hält, ist als das *Halteproblem* bekannt, welches die interessante Eigenschaft hat, dass es beweisbar nicht algorithmisch lösbar ist. Gegenstand dieses Buches sind jedoch ausnahmslos solche Probleme, die sich algorithmisch lösen lassen.

Halteproblem

deterministischer Algorithmus

Definition 2.1 (Algorithmus). *Ein* (deterministischer) Algorithmus *ist ein endlich beschreibbares, schrittweises Verfahren zur Lösung eines Problems. Die Schritte müssen eindeutig (ohne Wahlmöglichkeiten) beschrieben und ausführbar sein. Ein Algorithmus* endet *(oder* terminiert*), wenn er nach endlich vielen Schritten hält und ein Ergebnis liefert.*

Neben deterministischen Algorithmen gibt es noch weitere Typen von Algorithmen, zum Beispiel

nichtdeterministischer Algorithmus

- *nichtdeterministische Algorithmen* (bei denen es in jedem Rechenschritt Wahlmöglichkeiten geben kann; siehe Abschnitt 5.1.3),

randomisierter Algorithmus

- *randomisierte Algorithmen* (die von Zufallsentscheidungen Gebrauch und daher Fehler machen können, dafür aber oft effizienter sind; siehe z. B. die Abschnitte 7.1.3, 7.1.4 und 7.4.2)
- usw.

Wir werden uns jedoch vorwiegend mit deterministischen Algorithmen beschäftigen.

Beispiel 2.2 (Algorithmus für Zweifärbbarkeit). Das Problem der k-Färbbarkeit für Graphen wurde bereits in der Einleitung erwähnt und wird in Kapitel 3 formal beschrieben (siehe Definition 3.17 und Abschnitt 3.4.2). Es ist jedoch nicht nötig, die formale Definition zu kennen, um den folgenden Algorithmus zu verstehen, der für einen gegebenen Graphen testet, ob er zweifärbbar ist. Unter „Graph“ kann man sich vorläufig einfach eine Struktur wie in Abb. 1.1 vorstellen: eine endliche Menge so genannter Knoten (die in dieser Abbildung mit den Namen von Schülern beschriftet sind), von denen manche durch eine so genannte Kante miteinander verbunden sein können (solche Knoten heißen Nachbarn), andere möglicherweise nicht.

Die Frage, die unser Algorithmus beantworten soll, ist: Können die Knoten des Eingabegraphen mit höchstens zwei Farben so gefärbt werden, dass keine zwei benachbarten Knoten dieselbe Farbe erhalten?

Für einen gegebenen Graphen geht der Algorithmus nun folgendermaßen vor:

1. Zu Beginn seien alle Knoten des Graphen ungefärbt.
2. Wähle einen beliebigen Knoten und färbe ihn rot und alle seine Nachbarn blau.
3. Färbe für alle bereits roten Knoten jeden seiner noch ungefärbten Nachbarn blau.
4. Färbe für alle bereits blauen Knoten jeden seiner noch ungefärbten Nachbarn rot.
5. Wiederhole den 3. und den 4. Schritt, bis sämtliche Knoten des Graphen entweder rot oder blau sind.
6. Gibt es nun zwei rote Nachbarknoten oder zwei blaue Nachbarknoten, so ist der Graph nicht zweifärbbar; anderenfalls ist er es.

Dieser einfache Algorithmus wird später noch gebraucht werden.

2.2 Komplexitätsfunktionen

Ein wichtiges Maß für die Güte von Algorithmen ist die zur Ausführung benötigte Rechenzeit. Diese wird in der Anzahl der elementaren Schritte gemessen, die auszuführen sind, um das Problem mit dem betrachteten Algorithmus zu lösen. Elementare Schritte sind zum Beispiel Zuweisungen, arithmetische Operationen, Vergleiche oder Ein- und Ausgabeoperationen.

Neben der Rechenzeit ist auch der benötigte Speicherplatz von Bedeutung. Für die Abschätzung des Platzbedarfs wird die Anzahl der benötigten elementaren Speichereinheiten herangezogen. Elementare Speichereinheiten sind zum Beispiel Speicherplätze für Buchstaben, Zahlen oder für Knoten und Kanten in Graphen.

Die Rechenzeit und der Platzbedarf von Algorithmen werden immer in Abhängigkeit von der Größe der Eingabe gemessen. Die Größe der Eingabe muss daher vor der Aufwandsabschätzung genau spezifiziert sein. Diese Spezifikation ist nicht immer trivial und sollte sich der Fragestellung anpassen. Sind zum Beispiel n Zahlen $a_1, \ldots, a_n$ zu sortieren, so wird in der Regel n als die Größe der Eingabe verwendet. Der Platz für das Speichern einer nichtnegativen ganzen Zahl a benötigt jedoch genau genommen mindestens $\lceil \log_2(a) \rceil$ Bits, ist also logarithmisch in dem Zahlenwert a. Der Vergleich von zwei zu sortierenden Zahlen wird für gewöhnlich als eine elementare Operation angesehen. Auch hier ist genau betrachtet der Zeitaufwand des Vergleichs abhängig von der Größe der Zahlen. Diese Aspekte sollten bei der Bewertung der Güte von Algorithmen (in diesem Fall von Sortieralgorithmen) jedoch keine entscheidende Rolle spielen. Deshalb wird beim Sortieren im Allgemeinen die Anzahl der durchgeführten Vergleiche in Abhängigkeit von der Anzahl der zu sortierenden Zahlen gemessen.

Die verschiedenen Möglichkeiten der Darstellung von Graphen, die offenbar einen Einfluss auf die Laufzeitanalyse von Graphalgorithmen hat, werden ausführlich in Abschnitt 3.1 erörtert. Die Eingabegröße bei Graphenproblemen und -algorithmen ist üblicherweise die Anzahl der Knoten und der Kanten, manchmal als Summe und manchmal separat betrachtet, und manchmal gibt man die Laufzeit eines Graphalgorithmus auch nur als Funktion der Anzahl der Knoten des Graphen an.

$\mathbb{N}$ Mit dem Symbol $\mathbb{N}$ bezeichnen wir die Menge der nichtnegativen ganzen (also der natürlichen) Zahlen, d. h., $\mathbb{N} = \{0,1,2,\ldots\}$. Weiter sei $\mathbb{N}^+ = \mathbb{N} - \{0\} = \{1,2,3,\ldots\}$ die Menge der positiven natürlichen Zahlen. Mit $\mathbb{R}$ bezeichnen wir die Menge der reellen Zahlen, mit $\mathbb{R}_{\geq 0}$ die der nichtnegativen reellen Zahlen und mit $\mathbb{R}_{>0} = \mathbb{R}_{\geq 0} - \{0\}$ die der positiven reellen Zahlen.

$\mathbb{N}^+$
$\mathbb{R}$
$\mathbb{R}_{\geq 0}$
$\mathbb{R}_{>0}$

Zum Messen der Güte von Algorithmen betrachten wir Funktionen der Art

$$f : \mathbb{N} \to \mathbb{N}.$$

Diese Funktionen bilden also natürliche Zahlen auf natürliche Zahlen ab, wie zum Beispiel $f(n) = n^2$ oder $g(n) = n^3 - 2n + 4$. Den Buchstaben n benutzen wir als Standardargument für die Größe einer Eingabe. So ergibt sich etwa für den Algorithmus aus Beispiel 2.2 in Abhängigkeit von der Anzahl n der Knoten des Graphen eine Laufzeit von $c \cdot n^2$ für eine geeignete Konstante c, wie man sich leicht überlegen kann:

Übung 2.3. Analysieren Sie den Algorithmus in Beispiel 2.2 und zeigen Sie, dass er für eine geeignete Konstante c in der Zeit $c \cdot n^2$ läuft, wobei n die Anzahl der Knoten des gegebenen Graphen ist. (Diese Abschätzung der Laufzeit wird in Übung 2.9 noch verbessert.)

Diese Abhängigkeit der Laufzeit eines Algorithmus von der Größe der Eingabe (in einer geeigneten Codierung[5]) beschreibt die Funktion[6] $f : \mathbb{N} \to \mathbb{N}$ mit

$$f(n) = c \cdot n^2.$$

Wir sagen dann: „Der Algorithmus läuft in quadratischer Zeit“ und ignorieren die Konstante c (siehe Definition 2.8 in Abschnitt 2.3). Gemeint ist damit die Laufzeit des Algorithmus *im schlimmsten Fall* (englisch: *worst case*), das heißt, dies ist eine obere Schranke für die Laufzeit, die für alle denkbaren Eingabegraphen Gültigkeit hat.

Die Beschränkung auf nichtnegative ganze Zahlen liegt darin begründet, dass bei der Analyse (Abzählen der Programmschritte oder Speicherplätze) ausschließlich ganzzahlige nichtnegative Einheiten gezählt werden. Ein Programm kann nun

[5] Wir nehmen stets an, dass eine Eingabe x in vernünftiger, natürlicher Weise über einem Alphabet wie $\{0,1\}$ codiert ist und $n = |x|$ die Länge des Eingabewortes $x \in \{0,1\}^*$ in dieser Codierung bezeichnet; $\{0,1\}^*$ sei dabei die Menge der Wörter über dem Alphabet $\{0,1\}$. Da man jedoch die durch die verwendete Codierung verursachte Laufzeitänderung in der Regel vernachlässigen kann, nimmt man der Einfachheit halber oft an, dass n der „typische“ Parameter der Eingabe ist. Ist die Eingabe zum Beispiel ein Graph, so ist n die Anzahl seiner Knoten; ist sie eine boolesche Formel, so ist n die Anzahl der darin vorkommenden Variablen oder Literale; usw.

Codierung
$\{0,1\}^*$

[6] Vereinfacht schreiben wir für eine solche Funktion f auch $f(n)$ oder $c \cdot n^2$, obwohl diese Bezeichnung eigentlich den Funktionswert von f für das Argument n darstellt. In unserer Schreibweise sind zum Beispiel $n^2 + 2 \cdot n + 5$ und $3 \cdot 2^n$ zwei Funktionen. Im ersten Beispiel ist es die quadratische Funktion f mit $f(n) = n^2 + 2 \cdot n + 5$ und im zweiten Beispiel die exponentielle Funktion g mit $g(n) = 3 \cdot 2^n$.

mal nicht eine Schleife 3.7 Mal durchlaufen, 7.2 Vergleiche durchführen oder 66.19 Speicherplätze benutzen.

Definition 2.4 (Maximum, Minimum, Supremum und Infimum). *Sei X eine nicht leere Menge von reellen Zahlen.*

1. *Eine Zahl a ist eine* obere Schranke *für X, falls $x \leq a$ für alle $x \in X$ gilt.*
2. *Eine Zahl a ist eine* untere Schranke *für X, falls $x \geq a$ für alle $x \in X$ gilt.*
3. *Eine obere Schranke a für X ist eine* kleinste *obere Schranke für X, auch* Supremum *genannt und mit* $\sup(X)$ *bezeichnet, falls $a \leq b$ für jede obere Schranke b für X gilt.* — Supremum
4. *Eine untere Schranke a für X ist eine* größte *untere Schranke für X, auch* Infimum *genannt und mit* $\inf(X)$ *bezeichnet, falls $a \geq b$ für jede untere Schranke b für X gilt.* — Infimum
5. *Das* Maximum der Menge X, *bezeichnet mit* $\max(X)$, *ist eine Zahl a aus der Menge X, die eine obere Schranke für X ist.* — Maximum
6. *Das* Minimum der Menge X, *bezeichnet mit* $\min(X)$, *ist eine Zahl a aus der Menge X, die eine untere Schranke für X ist.* — Minimum

Nicht jede Menge muss ein Supremum, Infimum, Maximum oder Minimum besitzen.

Beispiel 2.5. Die Menge

$$A = \{n^2 - 100n \mid n \in \mathbb{R}\}$$

besitzt offensichtlich kein Maximum und kein Supremum, da es für jede Zahl x eine Zahl n gibt mit $n^2 - 100n > x$. Das Minimum und Infimum der Menge A ist -2500.

Die Menge

$$B = \{1 - \frac{1}{n} \mid n \in \mathbb{R}, n > 0\}$$

besitzt kein Maximum, kein Minumum und kein Infimum. Das Supremum der Menge B ist jedoch 1.

Definition 2.6 (Zeitkomplexitäten). *Seien W_n die* Menge aller Eingaben der Größe $n \in \mathbb{N}$ *und $A_T(w)$ die* Anzahl der elementaren Schritte von Algorithmus A für eine Eingabe w. *Wir definieren:* — Zeitkomplexität

1. *die* Worst-case-Zeitkomplexität (die Zeitkomplexität im schlechtesten Fall) von Algorithmus A für Eingaben der Größe n *als* — Worst-case-Zeitkomplexität

$$T_A^{WC}(n) \;=\; \sup\{A_T(w) \mid w \in W_n\};$$

2. *die* Best-case-Zeitkomplexität (die Zeitkomplexität im besten Fall) von Algorithmus A für Eingaben der Größe n *als* — Best-case-Zeitkomplexität

$$T_A^{BC}(n) \;=\; \inf\{A_T(w) \mid w \in W_n\}.$$

$T_A^{\mathrm{WC}}(n)$ ist die beste obere Schranke und $T_A^{\mathrm{BC}}(n)$ ist die beste untere Schranke für die Anzahl der Schritte, die Algorithmus A ausführt, um Eingaben der Größe n zu bearbeiten.

Komplexitätsfunktionen lassen sich in gleicher Weise für das Komplexitätsmaß Speicherplatz (kurz Platz; manchmal auch als *Raum* bezeichnet) definieren.

Platzkomplexität

Definition 2.7 (Platzkomplexitäten). *Seien W_n die* Menge aller Eingaben der Größe $n \in \mathbb{N}$ *und $A_S(w)$ die* Anzahl der elementaren Speicherplätze von Algorithmus A für eine Eingabe w. *Wir definieren:*

Worst-case-Platzkomplexität

1. *die* Worst-case-Platzkomplexität von Algorithmus A für Eingaben der Größe n *als*
$$S_A^{WC}(n) \;=\; \sup\{A_S(w) \mid w \in W_n\};$$

Best-case-Platzkomplexität

2. *die* Best-case-Platzkomplexität von Algorithmus A für Eingaben der Größe n *als*
$$S_A^{BC}(n) \;=\; \inf\{A_S(w) \mid w \in W_n\}.$$

2.3 Asymptotische Wachstumsfunktionen

Komplexitätsfunktionen sind meist nur sehr schwer exakt zu bestimmen. Die Berechnung der genauen Schrittanzahl ist aber oft nicht notwendig, da eigentlich nur das asymptotische Laufzeitverhalten von Interesse ist, d. h., nur die Größenordnung des Wachstums einer Komplexitätsfunktion spielt eine Rolle. Bei dieser Messmethode werden endlich viele Ausnahmen und konstante Faktoren (und damit natürlich auch konstante additive Terme) in den Komplexitätsfunktionen nicht berücksichtigt, sondern es wird lediglich das Grenzverhalten der Funktion für hinreichend große Eingaben betrachtet.

Definition 2.8 ($\mathcal{O}$-Notation). *Seien $f : \mathbb{N} \to \mathbb{N}$ und $g : \mathbb{N} \to \mathbb{N}$ zwei Funktionen. Wir sagen, f* wächst asymptotisch höchstens so stark wie *g, falls es eine reelle Zahl $c > 0$ und eine Zahl $x_0 \in \mathbb{N}$ gibt, sodass für alle $x \in \mathbb{N}$ mit $x \geq x_0$ gilt:*

$$f(x) \leq c \cdot g(x).$$

$\mathcal{O}(g)$

Die Menge der Funktionen f, die asymptotisch höchstens so stark wachsen wie g, bezeichnen wir mit $\mathcal{O}(g)$.

Zum Beispiel wächst die Funktion $2 \cdot n + 10315$ höchstens so stark wie die Funktion $n^2 + 2 \cdot n - 15$, also gilt $2 \cdot n + 10315 \in \mathcal{O}(n^2 + 2 \cdot n - 15)$. Umgekehrt ist dies jedoch nicht der Fall, $n^2 + 2 \cdot n - 15 \notin \mathcal{O}(2 \cdot n + 10315)$.

Obwohl der Begriff „wächst asymptotisch höchstens so stark wie“ in Definition 2.8 ausschließlich für Funktionen auf $\mathbb{N}$ definiert ist, benutzen wir die Schreibweise[7] $f \in \mathcal{O}(g)$ und $f \notin \mathcal{O}(g)$ auch für Funktionen f, die nicht ausschließlich auf nichtnegative ganze Zahlen abbilden, wie zum Beispiel die Funktionen $\sqrt{n}$ oder $\log_2(n)$.

[7] In der nordamerikanischen Literatur wird oft die Schreibweise $f = \mathcal{O}(g)$ für $f \in \mathcal{O}(g)$ verwendet.

In diesen Fällen sind immer die entsprechenden nichtnegativen, ganzzahligen Versionen gemeint, wie zum Beispiel $\lceil\sqrt{n}\rceil$ bzw. $\lceil\log_2(n)\rceil$, wobei $\lceil x\rceil$ die kleinste ganze Zahl bezeichnet, die nicht kleiner als x ist. Im Allgemeinen beziehen wir uns dann also auf die Funktion $\hat{f}:\mathbb{N}\to\mathbb{N}$ mit $\lceil x\rceil$

$$\hat{f}(n)=\max\{\lceil f(\max\{\lceil n\rceil,0\})\rceil,0\}.$$

In der Klasse $\mathcal{O}(n)$ befinden sich zum Beispiel die Funktionen

$$17\cdot n+18,\quad \frac{n}{2},\quad 4\sqrt{n},\quad \log_2(n)^2,\quad 2\cdot\log_2(n)\quad\text{und}\quad 48,$$

wobei 48 die Funktion $f:\mathbb{N}\to\mathbb{N}$ mit $f(b)=48$ ist. Nicht in der Klasse $\mathcal{O}(n)$ sind zum Beispiel die Funktionen

$$n\cdot\log_2(n),\quad n\cdot\sqrt{n},\quad n^2,\quad n^2\cdot\log_2(n),\quad n^3\quad\text{und}\quad 2^n.$$

Oft besteht die Eingabe aus verschiedenen Komponenten, die einen unterschiedlich starken Einfluss auf die Laufzeit haben können. Angenommen, wir suchen ein Muster M mit m Zeichen in einem Text T mit t Zeichen. Ein naiver Algorithmus würde zuerst die m Zeichen des Musters mit den ersten m Zeichen im Text vergleichen. Falls ein Zeichen im Muster nicht mit dem entsprechenden Zeichen im Text übereinstimmt, würde der Algorithmus die m Zeichen im Muster mit den m Zeichen im Text auf den Positionen $2,\ldots,m+1$ vergleichen, und so weiter. Das Muster wird also über den Text gelegt und schrittweise nach hinten verschoben. Die Anzahl der Vergleiche ist dabei insgesamt maximal $(t-(m-1))\cdot m$, da das Muster auf den letzten $m-1$ Positionen im Text nicht mehr angelegt werden muss. Bei einer Eingabegröße von $n=t+m$ hat der naive Algorithmus eine Worst-case-Laufzeit aus $\mathcal{O}(n^2)$. Diese obere Schranke wird zum Beispiel für $t=3m-1$ erreicht. Die genaue Laufzeit kann jedoch wesentlich präziser mit der Funktion $f:\mathbb{N}^2\to\mathbb{N}$ beschrieben werden, die durch $f(n,m)=\max\{(n-(m-1))\cdot m,0\}$ definiert ist.

Die Definition der Menge $\mathcal{O}(g)$ kann einfach angepasst werden, um mehrstellige Funktionen mit der $\mathcal{O}$-Notation abschätzen zu können. In unserem Beispiel wäre für eine zweistellige Funktion g die Klasse $\mathcal{O}(g)$ so definiert: $\mathcal{O}(g(n,m))$

$$\mathcal{O}(g)=\left\{f:\mathbb{N}^2\to\mathbb{N}\;\middle|\;\begin{array}{l}(\exists c\in\mathbb{R}_{>0})\ (\exists x_0\in\mathbb{N})\ (\forall x,y\in\mathbb{N})\\ [x,y\geq x_0\Longrightarrow f(x,y)\leq c\cdot g(x,y)]\end{array}\right\}.$$

Die Worst-case-Laufzeit des naiven Algorithmus für das Suchen eines Musters mit m Zeichen in einem Text mit n Zeichen ist dann aus $\mathcal{O}(n\cdot m)$. Ebenso hängen die Laufzeiten von Graphalgorithmen oft von der Anzahl n der Knoten und der Anzahl m der Kanten ab.

Übung 2.9. (a) In Übung 2.3 war zu zeigen, dass der Algorithmus für das Zweifärbbarkeitsproblem in Beispiel 2.2 eine in der Knotenzahl des gegebenen Graphen quadratische Laufzeit hat. Zeigen Sie nun, dass dieses Problem bezüglich der Anzahl n der Knoten und der Anzahl m der Kanten des gegebenen Graphen sogar in Linearzeit lösbar ist, also in der Zeit $\mathcal{O}(n+m)$.
Hinweis: Modifizieren Sie dabei den Algorithmus aus Beispiel 2.2 so, dass jeder Nachbar eines jeden gefärbten Knotens nur einmal betrachtet werden muss.

(b) Vergleichen Sie die Komplexitätsschranke $\mathcal{O}(n^2)$ aus Übung 2.3 mit der Komplexitätsschranke $\mathcal{O}(n+m)$ aus Teil (a). Ist die Linearzeit $\mathcal{O}(n+m)$ für alle Graphen eine echte Verbesserung gegenüber der quadratischen Zeit $\mathcal{O}(n^2)$?

Bei Exponentialzeit-Algorithmen ist es üblich, nicht nur konstante Faktoren wie bei der $\mathcal{O}$-Notation, sondern sogar polynomielle Faktoren zu vernachlässigen, da diese asymptotisch gegenüber einer exponentiellen Schranke kaum ins Gewicht fallen. Für die entsprechende Funktionenklasse ist die Bezeichnung $\tilde{\mathcal{O}}(g)$ gebräuchlich, wobei g eine gegebene exponentielle Funktion ist. Beispielsweise gilt $\tilde{\mathcal{O}}(n^2 \cdot 2^n) = \tilde{\mathcal{O}}(2^n)$.

$\tilde{\mathcal{O}}(g)$

Übung 2.10. Geben Sie eine formale Definition der Klasse $\tilde{\mathcal{O}}(g)$ an, wobei $g : \mathbb{N} \to \mathbb{N}$ eine gegebene exponentielle Funktion ist.

Die Menge $\Omega(g)$ der Funktionen, die asymptotisch mindestens bzw. genau so stark wachsen wie eine Funktion $g : \mathbb{N} \to \mathbb{N}$, ist definiert durch

$\Omega(g)$

$$\Omega(g) \;=\; \{f : \mathbb{N} \to \mathbb{N} \mid g \in \mathcal{O}(f)\},$$

und die Menge $\Theta(g)$ der Funktionen, die asymptotisch genau so stark wachsen wie eine Funktion $g : \mathbb{N} \to \mathbb{N}$, ist definiert durch

$\Theta(g)$

$$\Theta(g) \;=\; \mathcal{O}(g) \cap \Omega(g).$$

Übung 2.11. Stellen Sie für die folgenden Paare von Funktionen jeweils fest, ob $f \in O(g)$, $f \in \Omega(g)$ oder $f \in \Theta(g)$ gilt. Analog zur Definition von $\lceil x \rceil$ bezeichnet dabei $\lfloor x \rfloor$ die größte ganze Zahl, die x nicht überschreitet.

$\lfloor x \rfloor$

(a) $f(n) = \lfloor \sqrt{n} \rfloor$ und $g(n) = 100 \cdot n$.
(b) $f(n) = \lfloor \log_{10}(n) \rfloor$ und $g(n) = \lfloor \log(n) \rfloor$.
(c) $f(n) = \lfloor \sqrt[4]{n} \rfloor$ und $g(n) = \lfloor \sqrt{n} \rfloor$.
(d) $f(n) = n^3$ und $g(n) = \lfloor n \cdot \log(n) \rfloor$.
(e) $f(n) = 155 \cdot n^2 + 24 \cdot n + 13$ und $g(n) = n^2$.
(f) $f(n) = \lfloor n \cdot \log(n) \rfloor + \lfloor \sqrt{n} \rfloor$ und $g(n) = \lfloor n \cdot \log(\log(\log(n))) \rfloor$.

Wie leicht zu zeigen ist, gelten die folgenden Eigenschaften.

Behauptung 2.12. *1. Sind $f_1 \in \mathcal{O}(g_1)$ und $f_2 \in \mathcal{O}(g_2)$, so sind $f_1 + f_2 \in \mathcal{O}(g_1 + g_2)$ und $f_1 \cdot f_2 \in \mathcal{O}(g_1 \cdot g_2)$, wobei $f_1 + f_2$ die Funktion $f' : \mathbb{N} \to \mathbb{N}$ mit $f'(n) = f_1(n) + f_2(n)$ und $f_1 \cdot f_2$ die Funktion $f'' : \mathbb{N} \to \mathbb{N}$ mit $f''(n) = f_1(n) \cdot f_2(n)$ ist.*
2. Sind $f \in \mathcal{O}(g)$ und $g \in \mathcal{O}(h)$, so ist $f \in \mathcal{O}(h)$.
3. Sind $f \in \Omega(g)$ und $g \in \Omega(h)$, so ist $f \in \Omega(h)$.
4. Ist $f \in \Theta(g)$, so ist $g \in \Theta(f)$.
5. Sind $f_1 \in \mathcal{O}(g)$ und $f_2 \in \mathcal{O}(g)$, so ist $f_1 + f_2 \in \mathcal{O}(g)$.
6. Ist $f \in \mathcal{O}(g)$, so ist $f + g \in \Theta(g)$.

Übung 2.13. Beweisen Sie Behauptung 2.12.

Neben den Klassen $\mathcal{O}(g)$ und $\Omega(g)$ sind auch die Klassen $o(g)$ und $\omega(g)$ gebräuchlich. Die Klasse $o(g)$ ist für eine gegebene Funktion $g : \mathbb{N} \to \mathbb{N}$ wie folgt definiert: $o(g)$, $\omega(g)$

$$o(g) = \left\{ f : \mathbb{N} \to \mathbb{N} \;\middle|\; \begin{array}{l} (\forall c \in \mathbb{R}_{>0})\ (\exists x_0 \in \mathbb{N})\ (\forall x \in \mathbb{N}) \\ [x \geq x_0 \Longrightarrow f(x) < c \cdot g(x)] \end{array} \right\}$$

und enthält intuitiv genau die Funktionen, die asymptotisch weniger stark als g wachsen: Egal, wie klein die reelle positive Konstante c ist (und das wird durch den Allquantor davor ausgedrückt), wächst g im Vergleich zu f so stark, dass $c \cdot g(x)$ für hinreichend großes x größer als $f(x)$ ist. Ähnlich ist $\omega(g)$ die Klasse der Funktionen, die asymptotisch stärker wachsen als g, d. h., eine Funktion $f : \mathbb{N} \to \mathbb{N}$ ist genau dann in $\omega(g)$, wenn $g \in o(f)$.

Übung 2.14. (a) Geben Sie ein Gegenbeispiel an, das zeigt, dass die Gleichheit

$$o(g) = \mathcal{O}(g) - \Theta(g) \tag{2.1}$$

im Allgemeinen nicht gilt, d. h., definieren Sie Funktionen $f : \mathbb{N} \to \mathbb{N}$ und $g : \mathbb{N} \to \mathbb{N}$ mit $f \in \mathcal{O}(g) - \Theta(g)$, aber $f \notin o(g)$.

Hinweis: Definieren Sie f für gerade n anders als für ungerade n.

(b) Für eine gegebene Funktion $g : \mathbb{N} \to \mathbb{N}$ enthält die Klasse $\Omega_\infty(g)$ genau die Funktionen $f : \mathbb{N} \to \mathbb{N}$, für die es eine reelle Zahl $c > 0$ gibt, sodass für unendlich viele $x \in \mathbb{N}$ gilt:

$$f(x) > c \cdot g(x).$$

Zeigen Sie:

$$o(g) = \mathcal{O}(g) - \Omega_\infty(g). \tag{2.2}$$

2.4 Einige wichtige Klassen von Funktionen

In den vorherigen Abschnitten wurden Komplexitätsfunktionen eingeführt und das asymptotische Wachstumsverhalten von Funktionen beschrieben. Nun stellen wir noch einige Klassen von Funktionen vor, die bei der Abschätzung der Laufzeiten von Algorithmen eine besonders wichtige Rolle spielen. Nicht alle Funktionen, die man sich ausdenken könnte, sind bei der Analyse von Algorithmen relevant. Die folgenden Aufwandsklassen kommen jedoch häufiger vor. Sie sind hier aufsteigend bezüglich ihrer Inklusionen aufgeführt:

Klasse	Aufwand
$\mathcal{O}(1)$	konstant
$\mathcal{O}(\log_2(n))$	logarithmisch
$\mathcal{O}(n)$	linear
$\mathcal{O}(n \cdot \log_2(n))$	
$\mathcal{O}(n^2)$	quadratisch
$\mathcal{O}(n^3)$	kubisch
$\mathcal{O}(2^n)$	exponentiell

Im Folgenden gehen wir auf einige dieser Klassen noch etwas genauer ein.

2.4.1 Konstante Funktionen

Konstante Funktionen $f : \mathbb{N} \to \mathbb{N}$ liefern immer (also unabhängig von der Eingabegröße) den gleichen Funktionswert, d. h., $f(n) = f(n')$ gilt für alle $n, n' \in \mathbb{N}$. Wir verwenden den Begriff der „konstanten Funktion“ jedoch für alle Funktionen $f : \mathbb{N} \to \mathbb{N}$, bei denen sämtliche Funktionswerte $f(n)$ durch eine Konstante c beschränkt sind. Die Klasse der konstanten Funktionen wird in der Regel mit $\Theta(1)$ bezeichnet, obwohl natürlich $\Theta(49)$ und $\Theta(1)$ die gleiche Klasse definieren. Da wir ausschließlich Funktionen der Art $f : \mathbb{N} \to \mathbb{N}$ betrachten, gilt sogar $\Theta(1) = \mathcal{O}(1)$.

Algorithmen mit einer Zeitschranke aus $\mathcal{O}(1)$, die also nicht von der Größe der Eingabe abhängt, lösen in der Regel keine interessanten Probleme.

2.4.2 Logarithmische Funktionen

Logarithmen sind sehr schwach wachsende Funktionen. Die Anzahl der Ziffern der Dezimaldarstellung einer Zahl $m \in \mathbb{N}^+$ ist $\lfloor \log_{10}(m) \rfloor + 1$, also z. B. 5 für $m = 12345$ und 7 für $m = 3425160$.

Für eine positive reelle Zahl $b \neq 1$ (die die *Basis* genannt wird) und eine positive reelle Zahl n ist $\log_b(n)$ definiert als die reelle Zahl x mit $b^x = n$. Logarithmen mit verschiedenen Basen lassen sich leicht ineinander umrechnen. Für alle positiven reellen Zahlen a und b mit $a \neq 1$ und $b \neq 1$ gilt

$$\log_a(n) = \frac{\log_b(n)}{\log_b(a)}.$$

Für feste Basen a und b ist $\log_b(a)$ ein Faktor, der unabhängig von n ist. Deshalb ist

$$\Theta(\log_a(n)) = \Theta(\log_b(n)).$$

Weiterhin gilt $\log_b(n^k) = k \cdot \log_b(n)$ und somit

$$\Theta(\log_b(n^k)) = \Theta(\log_b(n)).$$

Logarithmen wachsen weniger stark als Polynome. Genauer gesagt gilt $\log_b(n) \in \mathcal{O}(n^k)$ und sogar $\log_b(n) \in o(n^k)$ für jedes $k > 0$.

Da für das asymptotische Wachstum die Basis des Logarithmus also keine Rolle spielt, schreiben wir bei allen Laufzeitbetrachtungen vereinfacht gern $\log(n)$ für $\log_2(n)$.

Weitere grundlegende Rechenregeln für Basen $b > 1$ und positive reelle Zahlen x und y sind:

$$\begin{aligned} \log_b(x \cdot y) &= \log_b(x) + \log_b(y) \quad \text{und} \\ \log_b(\tfrac{x}{y}) &= \log_b(x) - \log_b(y). \end{aligned}$$

Logarithmische Komplexitätsfunktionen sind hauptsächlich für den Speicherplatzbedarf bei der Lösung von Problemen von Belang. Logarithmische Zeit ist so knapp bemessen, dass in ihr nicht einmal die ganze Eingabe gelesen werden kann

(außer man verwendet geeignete „parallele“ Berechnungsmodelle[8]). Logarithmen spielen aber als Faktoren bei der Abschätzung des Zeitaufwands von Problemen durchaus eine wichtige Rolle. Beispielsweise arbeiten die besten Sortierverfahren (wie *Quicksort*, *Mergesort* usw.) in der Zeit $\mathcal{O}(n \cdot \log(n))$, und unter bestimmten Voraussetzungen kann man zeigen, dass man nicht schneller sortieren kann: Sortieren hat die Komplexität $\Theta(n \cdot \log(n))$.

Ein klassisches Graphenproblem, das nichtdeterministisch mit logarithmischem Platzbedarf gelöst werden kann, ist das Grapherreichbarkeitsproblem: Gegeben ein gerichteter Graph G mit zwei ausgezeichneten Knoten, s und t, gibt es einen Weg von s zu t in G? Auch wenn diese graphentheoretischen Begriffe erst in Kapitel 3 formal definiert werden, ist das Problem sicherlich verständlich. Informal kann ein Algorithmus, der es mit logarithmischem Platzbedarf löst, so beschrieben werden: Bei Eingabe von (G,s,t) wird „nichtdeterministisch“ (siehe Kapitel 5) ein gerichteter Weg geraten, der von s ausgeht. Dabei genügt es, sich immer nur den aktuellen Knoten zu merken, dessen Nachfolger geraten wird, und da man die Knotenindizes binär speichern kann, ist es klar, dass man mit logarithmischem Platz auskommt. Dieser nichtdeterministische Rateprozess geschieht in einer systematischen Art und Weise (zum Beispiel mittels Breitensuche, siehe Abschnitt 3.3). Viele der geratenen Wege werden nicht zum Ziel führen. Aber wenn es überhaupt einen erfolgreichen Weg von s zu t gibt, so wird er auch gefunden.

2.4.3 Polynome

Polynome $p : \mathbb{N} \to \mathbb{N}$ mit ganzzahligen Koeffizienten c_i, $0 \leq i \leq d$, sind Funktionen der Gestalt

$$p(n) = c_d \cdot n^d + c_{d-1} \cdot n^{d-1} + \cdots + c_1 \cdot n + c_0. \tag{2.3}$$

(Als Laufzeiten sind natürlich keine negativen Werte von $p(n)$ sinnvoll.) Den größten Einfluss in einer polynomiellen Laufzeit der Form (2.3) hat der Term $c_d \cdot n^d$ und damit der Grad d des Polynoms. Da wir bei der Analyse von Algorithmen an der asymptotischen Laufzeit interessiert sind und gemäß Abschnitt 2.3 konstante Faktoren vernachlässigen (d. h., für das Polynom p in (2.3) gilt $p \in \mathcal{O}(n^d)$), spielt eigentlich nur der Grad d eine Rolle.

Polynomialzeit-Algorithmen fasst man als „effizient“ auf, wohingegen Exponentialzeit-Algorithmen als „ineffizient“ gelten (siehe [Rot08, Dogma 3.7]), denn Polynome wachsen weniger stark als exponentielle Funktionen: Es gilt zum Beispiel $n^k \in \mathcal{O}(2^n)$ und sogar $n^k \in o(2^n)$ für jedes $k > 0$. Natürlich ist Effizienz ein dehnbarer Begriff, denn einen Polynomialzeit-Algorithmus, dessen Laufzeit ein Polynom vom Grad 153 ist, kann man unmöglich als effizient bezeichnen. Unter den natürlichen Problemen, die sich in Polynomialzeit lösen lassen, wird man jedoch so gut wie nie polynomielle Laufzeiten dieser Größenordnung finden; die meisten Polynomialzeit-Algorithmen für natürliche Probleme haben kubische, quadratische oder sogar lineare Laufzeiten, die man tatsächlich effizient nennen kann. Ein Beispiel für einen

[8] Zum Beispiel eine alternierende Turingmaschine mit Indexband, siehe [Rot08, Kapitel 5].

Algorithmus, der in quadratischer Zeit[9] läuft, ist der Algorithmus von Dijkstra, mit dem kürzeste Wege in gerichteten Graphen gefunden werden können. Ein anderes Beispiel für ein in quadratischer Zeit lösbares Problem haben wir in Beispiel 2.2 kennen gelernt: das Zweifärbbarkeitsproblem (siehe Übung 2.3, in der eine quadratische Laufzeit bezüglich der Knotenzahl im Eingabegraphen zu zeigen ist, und die auf Linearzeit bezüglich der Knoten- und Kantenzahl im gegebenen Graphen verbesserte Analyse in Übung 2.9).

2.4.4 Exponentielle Funktionen

Für reelle Zahlen x, a und b gilt:

$$\begin{aligned} x^a \cdot x^b &= x^{a+b}; \\ \frac{x^a}{x^b} &= x^{a-b}; \\ (x^a)^b &= x^{a \cdot b}, \text{ und im Allgemeinen ist dies } \neq x^{(a^b)}. \end{aligned}$$

Für reelle Zahlen $a > 1$ und $b > 1$ gilt:

$$\mathcal{O}(a^n) \subsetneq \mathcal{O}(a^{b \cdot n}), \tag{2.4}$$

das heißt, es gilt $\mathcal{O}(a^n) \subseteq \mathcal{O}(a^{b \cdot n})$, aber nicht $\mathcal{O}(a^n) = \mathcal{O}(a^{b \cdot n})$. Dies folgt aus

$$a^{b \cdot n} = a^{(b-1) \cdot n + n} = a^{(b-1) \cdot n} \cdot a^n.$$

Der Faktor $a^{(b-1) \cdot n}$ überschreitet für hinreichend großes n jede beliebige Konstante c, nämlich für $n > \frac{\log_a(c)}{b-1}$.

Weiterhin gilt für reelle Zahlen a mit $1 \leq a < b$:

$$\mathcal{O}(a^n) \subsetneq \mathcal{O}(b^n). \tag{2.5}$$

Da $b = a^{\log_a(b)}$ (und somit $b^n = a^{\log_a(b) \cdot n}$) gilt und da $\log_a(b) > 1$ aus $1 \leq a < b$ folgt, ergibt sich die Ungleichheit (2.5) aus der Ungleichheit (2.4).

In der Einleitung wurde das Problem der k-Färbbarkeit für Graphen an einem Beispiel vorgestellt. Für dieses Problem ist für $k \geq 3$ kein Algorithmus bekannt, der besser als in Exponentialzeit arbeitet. Die meisten der in diesem Buch vorgestellten Probleme haben im Allgemeinen (d. h., sofern nicht geeignete Einschränkungen betrachtet werden) ebenfalls exponentielle Zeitkomplexität.

2.5 Literaturhinweise

Die $\mathcal{O}$-, Ω- und Θ-Notationen wurden von Landau [Lan09] und Bachmann [Bac94] eingeführt.

[9] Genauer gesagt hat der Algorithmus von Dijkstra die Komplexität $\Theta(n^2)$, wobei n die Zahl der Knoten des Eingabegraphen ist. Verwendet man bei der Implementierung geeignete Datenstrukturen, so erhält man eine Komplexität von $\Theta(m + n \cdot \log(n))$, wobei m die Zahl der Kanten des Eingabegraphen ist. Für „dünne“ Graphen (also Graphen mit wenigen Kanten) ist diese Laufzeit besser als $\Theta(n^2)$.