A. Schulz, J. Rollin

Modul 63916

Effiziente Algorithmen

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Kurseinheit 1

Einführung in die Theorie der Algorithmen

Algorithmen spielen in unserem Leben eine immer wichtigere Rolle, sei es beim Handel an der Börse, der Verschlüsselung im Netzwerkprotokoll, bei der Tourenplanung im Logistikunternehmen, den vorgeschlagenen Freundschaften im sozialen Netzwerk, beim Computerspiel oder der Zuordnung von Studienplätzen und Bewerberinnen und Bewerbern. Diese Liste könnte man fast endlos fortsetzen. Im öffentlichen Diskurs haben Algorithmen oft eine negative Konnotation, denn sie stehen oft dafür, dass Entscheidungen statt von Menschen durch Maschinen und deren formale Regeln dazu, getroffen werden (die zudem häufig nicht transparent sind). An dieser Stelle begegnet man häufig einer Fehlauffassung davon, was Algorithmen eigentlich sind. Algorithmen haben streng genommen gar nichts mit Maschinen zu tun. Es handelt sich einfach um formalisierte Regeln, wie man ein Problem lösen kann. Häufig wird die Metapher des „Kochrezepts“ hierfür gebraucht. Es stimmt natürlich, dass Algorithmen häufig als Computerprogramme implementiert werden. Denn hier sind sie notwendig, um der Maschine das Lösen von Problemen zu vermitteln. Aber ein Algorithmus kommt beispielsweise auch für das Sortieren eines Spielkartendecks von Hand oder bei Montageanleitungen von Möbelstücken zum Einsatz.

In diesem Kurs wollen wir uns nicht an diesen ethischen Fragestellungen abarbeiten. Wir wollen Ihnen vielmehr die theoretischen Aspekte der interessanten Welt der effizienten Algorithmen näherbringen.

Algorithmen kommen in allen Teilgebieten in der Informatik vor. Manchmal sind sie selbst Gegenstand der Forschung, manchmal steht ihre Anwendung im Vordergrund. Je nachdem aus welcher Richtung man sich den Algorithmen nähert, geht es dabei um unterschiedliche Fragestellungen. In diesem Kurs konzentrieren wir uns auf die *Theorie* der effizienten Algorithmen. Wir blenden zum größten Teil Fragen zur Implementierung aus. Auch abstrahieren wir von der ausführenden Maschine und werden die Algorithmen so beschreiben, dass sie in einem relativ allgemeingültigen formalen Modell ausführbar sind.

In dieser ersten Kurseinheit wollen wir einige Grundlagen besprechen, die wir für den weiteren Verlauf des Kurses benötigen. In erster Linie geht es hierbei darum, sich über gewisse formale Grundprinzipien zu verständigen. So muss man zum Beispiel klären, welche Operationen ein Algorithmus einsetzen darf oder wie man die Laufzeit eines Algorithmus misst. Wir werden in dieser Kurseinheit aber auch einige Werkzeuge vorstellen, die im

späteren Verlauf des Kurses zum Einsatz kommen werden.

Nicht alle im Kurs vorgestellten Algorithmen sind die bestmöglichen für ihre Probleme. Sie werden aber alle den einen oder anderen Kniff benutzen, um eine Lösung effizient zu bestimmen. Es gibt zudem einige Algorithmen, die zwar im theoretischen Modell sehr performant sind, in der Praxis aber weniger sinnvoll sind. Meist liegt es daran, dass die Vorteile erst bei sehr sehr großen Instanzen zum Tragen kommen. Trotzdem hat es einen gewissen Reiz, zu verstehen, wie schnell man Algorithmen für bestimmte Probleme im theoretischen Modell machen kann.

1.1 Der Algorithmusbegriff

Ganz allgemein formuliert ist ein Algorithmus eine Folge von Instruktionen, die ein bestimmtes Problem lösen. Natürlich kann eine solche Instruktion nicht lauten: Schritt 1: Das Problem wird gelöst. In diesem Fall ist es unklar, wie die Instruktion ausgeführt werden soll. Deshalb stellen wir folgende Forderungen an unsere Instruktionen. Eine Instruktion in einem Algorithmus muss

- eindeutig,
- elementar und
- präzise sein.

Auch diese Begriffe sind natürlich dehnbar. Deshalb werden wir uns im weiteren Verlauf dieser Einheit auf ein formales System einigen, in welchem wir Instruktionen umsetzen können, die zweifellos diese Eigenschaften besitzen.

Bevor wir mit dem formalen Modell fortfahren, wollen wir ganz kurz auf die historischen Ursprünge von Algorithmen eingehen. Bereits im antiken Griechenland beschäftigte man sich intensiv mit Algorithmen. Der nach Euklid benannte Algorithmus eignet sich beispielsweise dazu, den größten gemeinsamen Teiler zweier Zahlen zu bestimmen. Auch einfache Primzahltests wie das Sieb des Eratosthenes entstanden in der hellenistischen Epoche. Neben diesen zahlentheoretischen Fragestellungen gab es auch Untersuchungen zu geometrischen Problemen, wie zum Beispiel Konstruktionsanweisungen zur Zeichnung von regulären Polygonen mit Zirkel und Lineal.

Der Begriff *Algorithmus* hat nicht, wie man vielleicht vermuten würde, seinen Ursprung im Griechischen. Er geht vielmehr auf den persischen Forscher Mūsā al-Khwārizmī zurück, der im 9. Jahrhundert lebte. Al-Khwārizmī (der auch ein Werk verfasste, welches Namensgeber des Wortes *Algebra* war) hatte eine Abhandlung über das Aufschreiben und Manipulieren von Zahlen in der Dezimaldarstellung (inklusive der Null) geschrieben.

In der Mathematik spielten algorithmische Fragestellungen lange Zeit eine untergeordnete Rolle. Sie traten zwar häufig implizit auf (zum Beispiel entdeckte Carl-Friedrich Gauß mit der diskreten Fourier-Transformation einen der wichtigsten Algorithmen überhaupt), waren aber selbst nicht Gegenstand der Forschung. Das lag auch zum Teil daran, dass es der Mathematik lange Zeit an einer formalen Basis fehlte. Als man sich um 1900 mehr und mehr mit den Grundlagen der Mathematischen Logik beschäftigte, rückten

algorithmische Fragestellungen vermehrt in den Fokus. Im Jahr 1900 formulierte der Mathematiker David Hilbert 23 programmatische Probleme, um die Mathematik gezielt weiterzuentwickeln. Das zehnte dieser Probleme fragte nach einem Algorithmus zur Lösung von diophantischen Gleichungen. Dieses Problem war schwierig zu lösen, auch deshalb, weil es kein formales System zur Beschreibung von Algorithmen gab. Als man nach (vollständigen und widerspruchsfreien) formalen Systemen für die Grundlagen der Mathematik suchte, erkannte Kurt Gödel, dass es ein solches System nicht geben kann. Hilberts 10. Problem wurde 70 Jahre später von Yuri Matiyasevich gelöst, indem er bewies, dass es keinen Algorithmus für das Lösen von diophantischen Gleichungen geben kann. Wie man also sieht, gibt es nicht für alle Probleme einen Algorithmus.

In den 1970er-Jahre beschäftigte man sich mehr und mehr mit der Frage, für welche Probleme es *effiziente* Algorithmen gibt. Man hatte beobachten können, dass es eine große Klasse von Problemen gibt, die sich nur schwer lösen lassen. Dies führte zur Theorie der NP-Vollständigkeit.

1.2 Ein formales Modell für Algorithmen

Das klassische Modell für die Formalisierung von *Berechnung* ist die Turingmaschine. Der Vorteil der Turingmaschine ist, dass sie über einen sehr eingeschränkten Befehlssatz verfügt, der auf das Notwendigste reduziert ist. Dadurch ist es sehr bequem, mit diesem Modell zu arbeiten, wenn man über Berechnungen Aussagen treffen will. Es ist aber sehr unhandlich, in diesem Modell konkret etwas zu programmieren. Hinzu kommt, dass die Turingmaschine sich stark von der Funktionsweise eines normalen Computers unterscheidet. In erster Linie ist hier anzumerken, dass die Turingmaschine einen sequenziellen Speicher benutzt, wohingegen Computer im Allgemeinen *Random-Access-Speicher* besitzen. Die Übersetzung der Turingmaschine in andere Berechnungsmodelle ist zwar möglich, aber in der Regel mit einem rechnerischen Mehraufwand verbunden. Dieser ist in der Regel polynomiell beschränkt, weshalb dies für Fragen der NP-Vollständigkeit keinen Unterschied macht. Für die Analyse von Algorithmen ist ein polynomieller Mehraufwand aber durchaus von Relevanz, weshalb wir hier ein anderes Modell benutzen werden, das sich näher an der (maschinellen) Realität orientiert.

1.2.1 Word-RAM

Eine Alternative zum Modell der Turingmaschine ist die *Registermaschine* (engl. *random access machine*, auch kurz RAM). Wie schon der Name andeutet, ist hier insbesondere der Zugriff auf den Speicher gegenüber der Turingmaschine anders organisiert. In der Literatur finden sich viele verschiedene Varianten des Modells der Registermaschine. Unterschiede gibt es im verwendeten Befehlssatz, der Größe der Speicherzellen und in der Art und Weise, wie Laufzeit und Speicherplatzverbrauch gemessen werden. Alle Varianten des Registermaschinenmodells sind gleichmächtig zum Modell der Turingmaschine. Es kann aber Unterschiede bei den Komplexitätsmaßen geben. Wir stellen mit der **Word-RAM** eine Version der Registermaschine vor, die sich eng an der Architektur von modernen Computern orientiert und sich deshalb auch gut für die Analyse von Algorithmen eignet.

Eine Word-RAM nutzt als Speicher eine Menge von Registern. In einem Register können wir eine natürliche Zahl in Binärdarstellung eintragen, die maximal w Bits benutzt. Man sagt in diesem Zusammenhang, dass das **Universum** des Modells die Menge $\{0, 1, \ldots, u = 2^w - 1\}$ ist, also alle Zahlen, die wir potenziell in einem Register speichern können. Die Zahl w, genannt **Wortgröße**, ist somit ein Parameter des Modells. Rechnen wir mit anderen Objekten als mit natürlichen Zahlen, müssen wir diese entsprechend kodieren. Bei einer n-elementigen Zeichenkette würden wir beispielsweise für jedes Zeichen eine Zahl wählen und die Zeichenkette dann in n Registern ablegen.

Die Register sind durchnummeriert. Wir bezeichnen sie mit $R0, R1, R2, R3, \ldots, Ru$. Das Register $R0$ ist ein spezielles Register, welches wir **Akkumulator** nennen. Zu Beginn der Berechnung steht die Eingabe in den Registern $R1, R2, \ldots, Rn$. Alle anderen Register enthalten die Zahl 0. Beachten Sie, dass die Eingabegröße n die Anzahl der Eingabewörter, und nicht deren Länge in Binärdarstellung, benennt. Wir werden voraussetzen, dass $u \geq n$, oder genauer gesagt, dass $w \geq \log n$.[1] Das heißt, wir erwarten eine Mindestwortgröße relativ zur Eingabe. Diese Annahme mag zunächst etwas irritieren, da die Eingabe des Algorithmus keinen Einfluss auf das Modell haben sollte. Es macht jedoch sehr viel Sinn, diese Schranke zu fordern, da wir ansonsten die einzelnen Teile der Eingabe nicht problemlos mit unserem Universum adressieren können.

Das eigentliche Berechnen funktioniert in der Word-RAM durch eine Ausführung von Befehlen, welche die Register modifizieren. Die Word-RAM verfügt über ein Programm, welches eine Sequenz dieser Befehle ist, und über einen Befehlszähler, welcher eine natürliche Zahl ist. Der Befehlszähler verweist auf den aktuellen Befehl. Nach jedem Befehl wird der Befehlszähler um eins erhöht. Das Programm kann nicht durch die Word-RAM verändert werden. Folgende Befehle kann eine Word-RAM ausführen:

- **Laden und Speichern:** Der Befehl LOAD x speichert den Wert von Rx in $R0$, der Befehl STORE x speichert den Wert von $R0$ in Rx.
- **Laden und Speichern mit indirekter Adressierung:** Der Befehl LOAD $[x]$ speichert den Wert von Ri in $R0$, wobei i der Wert ist, der in Rx steht. Analog dazu speichert der Befehl STORE $[x]$ den Wert von $R0$ in Ri (wiederum ist i der Wert von Rx).
- **Zuweisung von Konstanten:** Der Befehl $Ri = c$ speichert die Zahl c in Ri ab.
- **Arithmetische Operationen:** Der Befehl ADD x addiert zu der in $R0$ gespeicherten Zahl den Wert, der in Rx gespeichert ist. Analog dazu sind Befehle für andere elementare arithmetische Operationen wie Multiplikation, (ganzzahlige) Division und Modulo definiert.
- **Bitmanipulation:** Der Befehl AND x speichert in $R0$ das bitweise Und der Zahlen in $R0$ und Rx in Binärdarstellung. Andere Bitoperationen wie bitweises Oder, bitweises Negieren, Bitshifts etc. können ebenfalls mit entsprechenden Befehlen ausgeführt werden.
- **Steuerung des Programmflusses:** Der Befehl JZ i setzt den Befehlszähler auf den Wert i, falls $R0$ den Wert 0 enthält. Zusätzlich gibt es den Befehl HALT, welcher die Berechnung stoppt.

[1] Wenn die Basis beim Logarithmus nicht angegeben ist, verstehen wir den Logarithmus zur Basis 2.

Das Ergebnis einer Berechnung (oder der Verweis darauf) befindet sich am Ende im Register $R0$.

Das Word-RAM-Modell eignet sich gut zur Analyse von Algorithmen. Beim Sortieren von natürlichen Zahlen würde man zum Beispiel im Turingmaschinenmodell als Eingabelänge die Länge der kompletten Darstellung der Eingabe benutzen. Bei der Analyse interessiert uns hingegen eher die Anzahl der zu sortierenden Zahlen. Wir setzen aber hier voraus, dass die Zahlen von der Größe sind, dass wir sie ohne Umwege im Prozessor des Computers verarbeiten können. Diese Sichtweise gibt das Word-RAM-Modell sehr gut wieder. Beachten Sie, dass man durch die Nutzung der Bitoperationen auch eine gewisse Parallelität realisieren kann. So kann man in nur einem Schritt zwei $w/2 - 1$ Bit große Zahlen addieren. Deshalb gelten vergleichsbasierte untere Schranken in diesem Modell nicht. In der Tat gibt es Algorithmen zum Sortieren von natürlichen Zahlen, die auf einer Word-RAM schneller als $\Theta(n \log n)$ arbeiten. Bei der Analyse von Algorithmen in diesem Modell kann die Wortgröße w in der Laufzeit auftreten. In den Beispielen dieses Kurses nutzen wir aber die bereitgestellte Parallelität nicht aus und erhalten damit auch nur Laufzeiten in Bezug zu n.

Insbesondere, wenn wir mit numerischen Werten arbeiten, ist es wichtig, die Details unseres Modells zu verstehen. Durch (unerlaubtes) Ausnutzen der Parallelität gibt es hier das Potenzial für unerlaubte „Tricks“. Wir werden entsprechend unserer Modellannahmen in Zukunft immer davon ausgehen, dass sich die numerischen Werte der Eingabe mit $O(\log n)$ Bits darstellen lassen. Nur Zahlen mit $O(w)$ Bits können wir in konstanter Zeit verarbeiten. Das bedeutet implizit, dass wir arithmetische oder logische Operationen auf den Eingabewerten in $O(1)$ Zeit durchführen können.

Für den Fokus dieses Kurses müssen wir hier nicht auf weitere Details des Word-RAM-Modells eingehen. Es sei allerdings darauf hingewiesen, dass man auch noch strengere Varianten der Word-RAM kennt. So verbietet man beispielsweise bei der AC0-RAM Multiplikation (und alle Operationen, die sich nicht als ein Schaltwerk mit konstanter Tiefe darstellen lassen).

1.2.2 Pseudocode

Wenn wir Algorithmen angeben, werden wir dazu nicht zum Word-RAM-Programm greifen. Der Grund dafür ist, dass Word-RAM-Programme ähnlich zu Assemblercode sind und dadurch schwer lesbar. Viele Operationen in einem Word-RAM-Programm sind technischer Natur und für die Beschreibung des Kerns des Algorithmus unnötig. Aus diesem Grunde werden wir Algorithmen in *Pseudocode* angeben. Damit ist gemeint, dass wir eine imaginäre Programmiersprache benutzen, die sich nicht fest an formale Regeln hält. Pseudocode dient dazu, einen Algorithmus möglichst kompakt und eindeutig zu beschreiben. Seine Aufgabe ist es nicht, ausführbaren Programmcode zu liefern. Auch werden wir technische Details ausblenden (zum Beispiel werden wir nie prüfen, ob Eingaben syntaktisch korrekt sind).

Auch wenn ein strenges formales Gerüst für den Pseudocode fehlt, wollen wir uns dennoch auf einige sinnvolle Annahmen einigen.

- Variablen werden nicht deklariert und sind untypisiert. Sie werden kursiv geschrieben.
- Es werden nur die Schlüsselwörter(-kombinationen) **for .. to .. do**, **foreach .. do**, **while .. do**, **repeat .. until**, **if .. then .. else** und **return** benutzt. Ihre Interpretation

sollte selbsterklärend sein. Programmblöcke werden durch Einrücken gekennzeichnet.

- Zuweisungen werden mit einem Pfeil markiert, zum Beispiel $x \leftarrow x + 1$.
- Mathematische Ausdrücke, insbesondere auch Mengenschreibweise, dürfen verwendet werden.
- Auf die Elemente von Listen oder Arrays können wir über ihren Index in eckigen Klammern zugreifen, zum Beispiel $L[2] \leftarrow 3$. Wenn nichts anderes erwähnt wird, hat das erste Element den Index 1.
- Komplexere Aufgaben werden verbal in deutscher Sprache angegeben. Zum Beispiel „Lösche das kleinste Element aus L“.
- Nutzen wir Unterprogramme, geben wir diese als Prozeduren in Großschreibung (Kapitälchen) an, zum Beispiel HILFSFUNKTION(x, y, z).
- Nutzen wir komplexere Datenstrukturen, bezeichnen wir deren Operationen in englischer Sprache und geben sie in kursiver Schreibweise an, zum Beispiel $B.search(q)$ für die Suchoperationen eines Suchbaumes B.
- Kommentare geben wir rechtsbündig an und stellen diesen ein ▷ voran.
- Gelegentlich nutzen wir auch andere Notationen, die dann aber selbsterklärend sind.

Der folgende einfache Algorithmus löst das Problem, für eine Liste von Zahlen zu prüfen, ob ein Element doppelt vorhanden ist.

Algorithmus 1 Prüfe, ob eine Liste L doppelte Einträge enthält

```
1: Sortiere L
2: for i ← 1 to |L| − 1 do                          ▷ i ist aktuell zu prüfende Stelle
3:     if L[i] = L[i + 1] then return „L enthält Duplikate“
4: return „keine Duplikate“
```

Ein etwas komplexeres Beispiel ist die Beschreibung von Mergesort, angegeben als Algorithmus 2. Hier ist die Verwendung von rekursiven Aufrufen zu sehen.

Beim Mergesort-Beispiel benutzten wir $X[2..]$ für die Liste X ohne erstes Element. Solche Notationen sollten sich immer aus dem Kontext erklären, auch wenn wir sie nicht formal besprochen haben. Da der Pseudocode im weiteren Verlauf des Kurses immer nur eine Zusammenfassung von zuvor erklärten Ideen ist, ist zudem stets klar, was wir meinen.

1.3 Laufzeit

Neben der Angabe eines Algorithmus und dem Beweis seiner Korrektheit, interessiert uns immer auch seine Laufzeit. Wir wollen deshalb an dieser Stelle besprechen, was die Laufzeit eines Algorithmus ist, wie wir diese messen und wie wir sie abschätzen können.

Algorithmus 2 Sortiere Liste L mit Mergesort(L)

```
1:  procedure MERGE(X, Y)                          ▷ Übergebene Listen sind vorsortiert
2:      if X = ∅ then return Y
3:      if Y = ∅ then return X
4:      if X[1] ≤ Y[1] then
5:          Z ← MERGE(X[2..], Y)
6:          return X[1] ∘ Z                        ▷ ∘ verknüpft Listen
7:      if Y[1] < X[1] then
8:          Z ← MERGE(X, Y[2..])
9:          return Y[1] ∘ Z
10: procedure MERGESORT(L)
11:     if |L| ≤ 1 then return L                   ▷ Rekursionsanker
12:     L1 ← erste Hälfte von L
13:     L2 ← zweite Hälfte von L
14:     L1 ← MERGESORT(L1)
15:     L2 ← MERGESORT(L2)
16:     return MERGE(L1, L2)
```

Offensichtlich hängt die Laufzeit eines Algorithmus von seiner Eingabe $I \in \mathcal{I}$ ab, wobei $\mathcal{I}$ die Menge aller Eingabeinstanzen bezeichnet. Für eine feste Eingabe I bezeichnen wir als Laufzeit eines Algorithmus die Anzahl der Operationen $t(I)$, die im dazugehörigen Word-RAM-Programm ausgeführt werden. Da wir die Algorithmen nicht direkt als Word-RAM-Programm angeben (sondern in Pseudocode), ist eine genaue Bestimmung der Laufzeit nicht möglich. Wir geben uns deshalb damit zufrieden, die Laufzeit abzuschätzen.

Die Laufzeit für *eine* Eingabe hilft uns aber bei der Analyse eines Algorithmus nur wenig weiter, denn wir wollen ja das generelle Laufzeitverhalten beschreiben. Würden wir die Laufzeiten für alle Eingaben kennen, könnten wir dies aber nur schlecht in kompakter Form angeben. Deshalb misst man die Laufzeit von I in der Regel in Abhängigkeit zu einem Parameter der Eingabe $s(I)$, wobei s eine Funktion ist, die jeder Eingabeinstanz eine natürliche Zahl zuordnet. Formal entspricht der Wert $s(I)$ der Anzahl der Register, die man benötigt, um die Instanz I einem Word-RAM-Programm zu übergeben. Meist ergibt sich der Wert direkt aus der Struktur der Eingabeinstanz, ohne dass wir jedes Mal auf die technischen Aspekte bei der Kodierung von I als Eingabe für ein Word-RAM-Programm eingehen. Zum Beispiel werden wir die Laufzeit von Graphenalgorithmen in Abhängigkeit zur Anzahl der Knoten und Kanten des Graphen messen. Mit steigendem $s(I)$ sollten die Probleme prinzipiell „schwieriger“ werden und damit auch eine größere Laufzeit bei den Algorithmen erfordern. In der Praxis können wir ein solches Verhalten aber nicht voraussetzen. Das Sortieren einer langen vorsortierten Liste von Zahlen ist gegebenenfalls einfacher (zum Beispiel mit Bubblesort) als das Sortieren einer kürzeren Liste. Deshalb unterscheidet man drei verschiedene Formen der Laufzeitanalyse. Im **Best-Case** interessieren wir uns für die schnellste Laufzeit unter allen Eingaben I, bei denen $s(I)$ den gleichen Wert liefert. Wir definieren hier als Laufzeit die Funktion

$$T_{\text{best}}(n) = \min_{I \in \mathcal{I}} \{t(I) \mid s(I) = n\}.$$

Im **Average-Case** hingegen beziehen wir uns auf die durchschnittliche Laufzeit. Das heißt,

hier betrachten wir die Funktion

$$T_{\text{avg}}(n) = \frac{1}{|\{I \in \mathcal{I} \colon s(I) = n\}|} \sum_{I \in \mathcal{I} \colon s(I)=n} t(I).$$

Als eine dritte Möglichkeit existiert zudem die **Worst-Case-Laufzeit**, welche sich mit

$$T_{\text{worst}}(n) = \max_{I \in \mathcal{I}} \{t(I) \mid s(I) = n\}$$

ergibt. Für die Analyse von Algorithmen nutzen wir in der Regel (und in diesem Kurs ausschließlich) die Worst-Case-Laufzeit und schreiben statt $T_{\text{worst}}(n)$ nur $T(n)$. Die best-case Analyse ist deshalb wenig sinnvoll, da es für die meisten Probleme triviale Instanzen gibt, die man leicht lösen kann. Wie effizient der Algorithmus dann aber im Allgemeinen laufen wird, können wir damit in der Regel nur schlecht abschätzen. Besser ist es, die average-case Analyse zu verwenden. Hier bekommen wir eine sehr gute Aussage, allerdings ist es auch sehr schwierig und aufwendig, die Funktion $T_{\text{avg}}(n)$ zu bestimmen – zumal es zudem fraglich ist, ob die Instanzen in der Praxis auch wirklich gleichverteilt auftreten. Die Worst-Case-Analyse ist deshalb oft die bessere Methode. Häufig können wir eine Abschätzung von $T_{\text{worst}}(n)$ leicht vornehmen. Wir erwarten zusätzlich eine treffendere Beschreibung gegenüber $T_{\text{best}}(n)$. Natürlich wird für viele Instanzen die Laufzeit schneller sein, als wir es mit der Worst-Case-Analyse abgeschätzt haben. Wir gehen aber davon aus, dass sich die Laufzeit einer durchschnittlichen Instanz nicht deutlich von der Worst-Case-Laufzeit unterscheiden wird. Zudem erwarten wir, dass eine höhere Worst-Case-Laufzeit auch eine höhere average-case Laufzeit impliziert. Diese Annahme ist natürlich nicht immer gegeben, insbesondere wenn wir hinzunehmen, dass die in der Praxis zu erwartenden Instanzen sich stark von denen unterscheiden, die den Worst-Case realisieren. Wir greifen diese Problematik noch einmal später in Kurseinheit 6 auf. Abschließend kann man sagen, dass uns die Worst-Case-Laufzeit ein verlässliches und akzeptiertes Maß gibt, mit dem man die Laufzeit von Algorithmen messen kann.

1.3.1 Die Groß-O-Notation

Ein Vorteil der Angabe der Laufzeit als Worst-Case (auch Average-Case oder Best-Case) ist es, dass das eigentliche Resultat eine Funktion $T(n) \colon \mathbb{N} \to \mathbb{N}$ ist. Mit solchen Funktionen können wir viel einfacher umgehen als beispielsweise mit Funktionen des Typs $\mathcal{I} \to \mathbb{N}$. Trotzdem ist die genaue Angabe dieser Funktion $T(n)$ schwierig. Hinzu kommt, dass wir ja auch gar nicht eine genaue Bestimmung vornehmen können, da uns dazu die Details aus dem Word-RAM-Programm fehlen. Deshalb reicht es an dieser Stelle aus, auf eine asymptotische Schranke für die Laufzeit zurückzugreifen. Für die Angabe von asymptotischen Schranken nutzen wir die Groß-O-Notation.

Für eine Funktion $f \colon \mathbb{R} \to \mathbb{R}$ bezeichnen wir mit $O(f)$ eine Menge von Funktionen des Typs $\mathbb{R} \to \mathbb{R}$, die als asymptotische obere Schranke die Funktion f haben. Vereinfacht gesprochen, berücksichtigt die Groß-O-Notation zwei Aspekte: Für kleine n wird keine Aussage getroffen und konstante Faktoren werden vernachlässigt. Formal ausgedrückt ergibt sich

$$O(f) := \{g \mid \exists (c > 0)\ \exists n_0\ \forall (n \geq n_0) \colon g(n) \leq c \cdot f(n)\}.$$

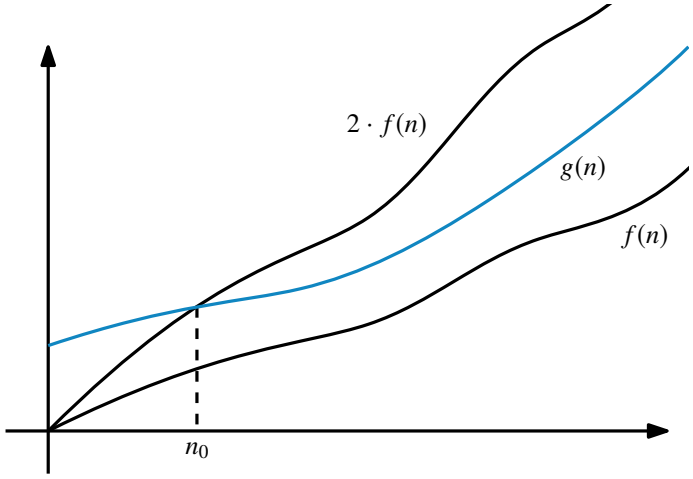

Abbildung 1.1: Skizze zur asymptotischen Beschränkung. Im Bild ist $g = O(f)$ (Konstante $c = 2$).

Beachten Sie, dass das c nicht vom gewählten n_0 abhängen darf. Die Abbildung 1.1 illustriert die Groß-O-Notation.

Obwohl $O(f)$ eine Menge ist, schreiben wir $g = O(f)$ statt eigentlich richtig $g \in O(f)$. Diese streng genommen inkorrekte Notation hat sich als Schreibweise etabliert und wir wollen nicht vom Standard abweichen. Eine weitere Konvention ist es, für $f(n) = c$ (für eine Konstante c) statt $O(f)$ nur kurz $O(1)$ zu schreiben. $O(1)$ bezeichnet also alle Funktionen, die sich durch eine Konstante beschränken lassen. Des Weiteren beschreiben wir oft das f in $O(f)$ durch eine Funktion in der Variablen n, also zum Beispiel $O(n^2)$, während andere Zeichen konstante Parameter sind, also zum Beispiel $O(n^d)$ für ein festes d.

Mitunter möchte man auch ausdrücken, dass eine Funktion eine *echte* obere Schranke für eine andere Funktion ist. Dies kann man mit der *Klein-O-Notation* beschreiben. In diesem Fall gilt

$$o(f) := \{g \mid \forall(c > 0)\ \exists n_0\ \forall(n \geq n_0)\colon g(n) < c \cdot f(n)\}.$$

Offensichtlich folgt aus $f = o(g)$ auch $f = O(g)$. Analog zu den oberen Schranken können wir auch asymptotische untere Schranken angeben. Man benutzt an dieser Stelle

$$\Omega(f) := \{g \mid \exists(c > 0)\ \exists n_0\ \forall(n \geq n_0)\colon g(n) \geq c \cdot f(n)\}.$$

Gelten $g = O(f)$ und $g = \Omega(f)$, schreiben wir $g = \Theta(f)$ als scharfe asymptotische Schranke.

Beispiel 1.1 Seien $g(n) = 1000n^x$ und $f(n) = n^y$ mit $y > x$ (zum Beispiel $g(n) = 1000n^2$ und $f(n) = n^3$), dann gilt $g = O(f)$. Um dies zu zeigen, wählen wir $n_0 = 1$ und $c = 1000$. Wegen $y > x$, gilt nun

$$g(n) = 1000n^x \leq 1000n^y = cn^y = c \cdot f(n),$$

und $g = O(f)$ ist gezeigt.

Test 1.1

Zeigen Sie, dass für $g(n) = 1000n + 1\,000\,000$ und $f(n) = n \log n$ gilt, dass $g = O(f)$.

Aus der Definition der Groß-O-Notation können wir folgende Regeln ableiten, welche Umformungen und Abschätzungen erleichtern. Wir verzichten auf die Beweise, die allesamt durch elementare Umstellungen der Definitionen folgen.

- Für zwei Werte c, d gilt $g(n) = c \cdot f(n) + d = O(f)$.
- Sei g ein Polynom vom Grad k, also $g(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots a_1 n + a_0$, mit $a_k > 0$. Dann ist $g = \Theta(n^k)$.
- Für $k' > k$ gilt $n^k = o(n^{k'})$ (vgl. Beispiel 1.1).
- Für jedes $r > 1$ und $d > 0$ gilt $n^d = o(r^n)$ (polynomielles Wachstum ist immer kleiner als exponentielles).
- Für jedes $b > 1$, $p \geq 1$ und $c > 0$ gilt $(\log_b n)^p = o(n^c)$ (logarithmisches Wachstum ist immer kleiner als polynomielles).
- Für zwei Funktionen g_1, g_2 gilt, dass $g_1(n) + g_2(n) = O(g_1 + g_2)$ und $g_1(n) \cdot g_2(n) = O(g_1 \cdot g_2)$.
- Wenn $g = O(f)$ und $h = O(g)$, dann ist $h = O(f)$.

Die letzten zwei aufgeführten Punkte brauchen wir häufig für die Analyse von Algorithmen. Besteht beispielsweise ein Algorithmus aus einer Schleife, die $O(\log n)$-mal durchlaufen wird, und innerhalb der Schleife beträgt der Aufwand $O(n^2)$, dann ist der Gesamtaufwand beschränkt durch $O(n^2 \log n)$. Besteht ein Algorithmus hingegen aus zwei Teilen mit Aufwand $O(n^3)$ und $O(n)$, die nacheinander ausgeführt werden, so ist der Gesamtaufwand durch $O(n^3 + n) = O(n^3)$ beschränkt.

Test 1.2

Gilt für folgende Funktionen f_i und g_i die Beziehung $g_i \in O(f_i)$ oder $g_i = \Omega(f_i)$ oder gelten beide Beziehungen (also $g_i = \Theta(f_i)$)?

(a) $f_1(n) = n^3 + n^2$, $\quad g_1(n) = n^3 - n^2$,

(b) $f_2(n) = 10n^2$, $\quad g_2(n) = n\sqrt{n}$,

(c) $f_3(n) = n \log n^2$, $\quad g_3(n) = n \log n^4$.

(d) $f_4(n) = \log_{10} n$, $\quad g_4(n) = \log_2 n$.

Häufig wird die Groß-O-Notation genutzt, um von Konstanten zu abstrahieren. Das geschieht immer dann, wenn der Groß-O-Ausdruck Teil eines Terms ist, wie zum Beispiel $f(n) = 2^{O(n)}$. In diesem Fall meint man, dass es eine Konstante $c > 0$ gibt, sodass $f(n) \leq 2^{c \cdot n}$, was nicht das Gleiche ist wie $f = O(2^n)$. Gleiches gilt für die Klein-O-Notation.

1.3.2 Das Mastertheorem

Da effiziente Algorithmen oft rekursiv arbeiten, sind Rekurrenzgleichungen in der Laufzeitanalyse keine Seltenheit. Das Auflösen dieser Rekurrenzen ist nicht immer einfach. Aus diesem Grund stellen wir einen Satz vor, der es uns erlaubt, die Lösungen für viele Rekurrenzgleichungen asymptotisch abzuschätzen. Zur Motivation dieses Satzes (und als Vorbereitung für dessen Beweis) sehen wir uns den Mergesort-Algorithmus (Algorithmus 2) noch einmal an.

Zuerst wollen wir den Aufruf von $\textsc{Merge}(X, Y)$ analysieren. Wir sehen, dass wir bei $X = \emptyset$ oder $Y = \emptyset$ nur $O(1)$ Zeit benötigen. In allen anderen Fällen gibt es genau einen rekursiven Aufruf, denn die Zeilen 4 und 7 enthalten sich gegenseitig ausschließende Bedingungen. Wir messen den Aufwand von $\textsc{Merge}(X, Y)$ relativ zur Länge n von $X \circ Y$ und erhalten damit für die Laufzeit T_{merge}

$$\begin{aligned} T_{\text{merge}}(n) &\leq T_{\text{merge}}(n-1) + c \\ T_{\text{merge}}(1) &\leq c, \end{aligned}$$

wobei c eine hinreichend große Konstante ist. Wenn wir die Rekkurrenz auffalten, erhalten wir

$$T_{\text{merge}}(n) \leq \underbrace{c + c + c + \cdots + c}_{n \text{ mal}} = n \cdot c = O(n).$$

Nun kommen wir zur Abschätzung der eigentlichen Laufzeit für Mergesort. Die Rekurrenz ergibt sich direkt aus den rekursiven Aufrufen. Wir messen den Aufwand relativ zur Länge der zu sortierenden Liste. Um eine Liste mit n Elementen zu sortieren, müssen wir rekursiv zwei Listen der halben Länge sortieren und dann mischen. Wir nehmen der Einfachheit halber an, dass die Länge n der Liste eine Zweierpotenz ist. Es folgt

$$\begin{aligned} T_{\text{mergesort}}(n) &\leq 2T_{\text{mergesort}}(n/2) + T_{\text{merge}}(n) \leq 2T_{\text{mergesort}}(n/2) + cn \\ T_{\text{mergesort}}(1) &\leq c, \end{aligned}$$

wobei c auch hier wieder eine ausreichend große Konstante ist und wir das Ergebnis der Analyse von Merge eingesetzt haben. Um die Abschätzung vorzunehmen, sehen wir uns den zugehörigen **Rekursionsbaum** an. Dies ist ein Baum, dessen Knoten den in der Berechnung auftretenden Teilproblemen entsprechen. Werden bei einem Teilproblem X die rekursiven Lösungen für $Y_1, Y_2, \ldots$ benötigt, dann werden $Y_1, Y_2, \ldots$ die Kinder von X. Die Wurzel des Baumes repräsentiert das ursprüngliche Problem. Wir beschriften die Knoten des Baumes mit den Kosten der entsprechenden Teilprobleme *ohne* die rekursiven Aufrufe (im Folgenden *Aufwand* genannt). Der schematische Rekursionsbaum für Mergesort ist in Abbildung 1.2 zu sehen.

Wir nummerieren die Ebenen des Rekursionsbaums so durch, dass die Wurzel sich auf Ebene 0 befindet. Im Rekursionsbaum zu Mergesort erkennen wir, dass auf der nullten Ebene ein Gesamtaufwand von cn entsteht. Die erste Ebene enthält zwei Knoten mit Aufwand jeweils $cn/2$, was auch wieder einen Gesamtaufwand von cn erfordert. Auf der zweiten Ebene haben wir nun vier Knoten mit Aufwand $cn/4$ und schon wieder Gesamtaufwand cn. Das ist kein Zufall, denn wie man leicht sehen kann, haben wir auf

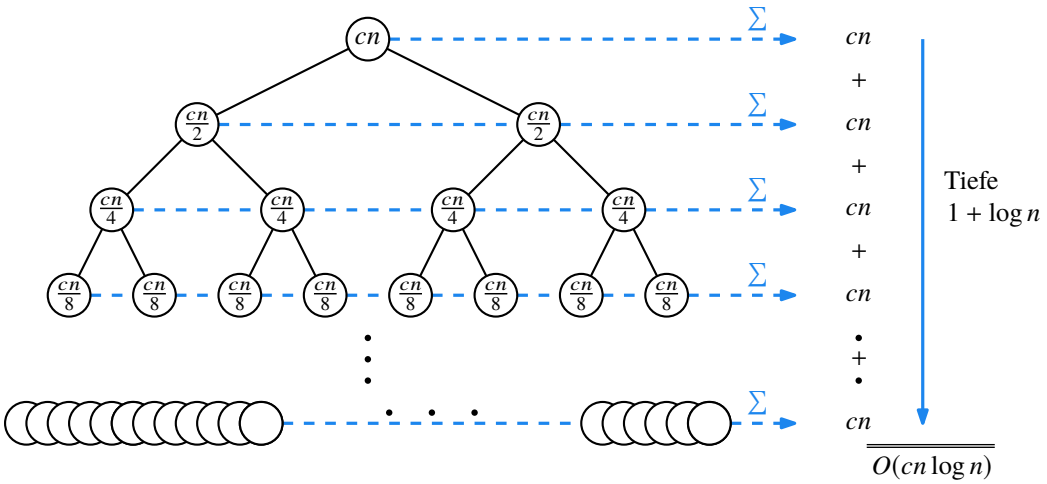

Abbildung 1.2: Rekursionsbaum für Mergesort.

der Ebene k genau 2^k Knoten, die jeweils einen Aufwand von $cn/2^k$ verursachen, was zusammen cn ausmacht. Der binäre Logarithmus (einer Zweierpotenz) gibt an, wie lange wir diese Zahl durch 2 teilen müssen, um auf 1 zu kommen. Der Rekursionsbaum hat deshalb $\log n + 1$ viele Ebenen. Summieren wir den Aufwand für jede Ebene auf, erhalten wir insgesamt $cn + cn \log n$. Es folgt also

$$T_{\text{mergesort}}(n) = O(n \log n).$$

Test 1.3

Skizzieren Sie den schematischen Aufbau des Rekursionsbaumes für das Teilproblem MERGE.

Der Rekursionsbaum für Mergesort ist im gewissen Sinne ein Spezialfall, da sich auf jeder Ebene der gleiche Gesamtaufwand ergibt. Dies muss für andere Rekursionen nicht zwangsweise so sein. Der Gesamtaufwand könnte steigen oder fallen. Der folgende Satz analysiert diesen Aspekt und liefert uns dadurch ein wichtiges Hilfsmittel für das Auflösen von Rekurrenzgleichungen.

Satz 1.1 — Mastertheorem.
Sei

$$T(n) \leq aT(n/b) + O(n^d),$$

wobei $a \in \mathbb{N}$, $b > 1$ und $d \geq 0$ frei wählbare Konstanten sind . Dann gilt:

- Wenn $a/b^d < 1$, dann ist $T(n) = O(n^d)$.
- Wenn $a/b^d = 1$, dann ist $T(n) = O(n^d \log n)$.
- Wenn $a/b^d > 1$, dann ist $T(n) = O(n^{\log_b a})$.

Beweis. Wir nehmen zunächst an, dass n eine Potenz von b ist. Für die Analyse der Rekurrenz betrachten wir den dazugehörigen Rekursionsbaum. Da jeder Knoten im

Rekursionsbaum a Kinder hat, gibt es auf der Ebene k demnach a^k Knoten. Zwischen den Ebenen werden die Teilprobleme um den Faktor $1/b$ skaliert. Also haben wir auf der Ebene k jeweils Teilprobleme der Größe n/b^k. Der Gesamtaufwand auf der Ebene k ist demnach

$$a^k \cdot O((n/b^k)^d) = O(n^d) \cdot (a/b^d)^k.$$

Um nun $T(n)$ abzuschätzen, summieren wir über alle Ebenen (von denen es $t = 1 + \log_b n$ viele gibt).

$$T(n) \leq O(n^d) \cdot \left((a/b^d)^0 + (a/b^d)^1 + (a/b^d)^2 + \cdots + (a/b^d)^t\right) \tag{1.1}$$

Die in der Klammer auftretende Summe bildet eine geometrische Reihe. Wie wir gleich sehen werden, hängt das asymptotische Wachstum einer geometrischen Reihe nur vom größten Summanden ab. Da die Summanden einer geometrischen Reihe eine schwach monoton wachsende oder fallende Folge beschreiben, gibt es hier 3 Möglichkeiten: Der echt größte Summand ist $(a/b^d)^0 = 1$, alle Summanden sind gleich groß, oder der echt größte Summand ist $(a/b^d)^t$. Wir gehen jeden der drei Fälle einzeln durch.

- Der echt größte Summand ist $(a/b^d)^0$. Das bedeutet, dass $a/b^d < 1$. In diesem Fall konvergiert die geometrische Reihe gegen eine Konstante und damit gilt

  $$T(n) = O(n^d).$$

- Alle t Summanden sind genau dann gleich groß, was heißt, dass $a/b^d = 1$. In diesem Fall ist

  $$T(n) = (t+1) \cdot O(n^d) = O(n^d \log n).$$

- Der echt größte Summand ist $(a/b^d)^t$. Dann muss gelten $a/b^d > 1$. Für die geometrische Reihe gilt $\sum_{i=0}^{k} q^i = (1-q^{k+1})/(1-q) = O(q^{k+1}) = O(q^k)$. Demzufolge ist

  $$\left((a/b^d)^0 + (a/b^d)^1 + (a/b^d)^2 + \cdots + (a/b^d)^t\right) = O((a/b^d)^t).$$

  Damit gilt für die Summe (1.1)

  $$T(n) = O\left(n^d \left(\frac{a}{b^d}\right)^{\log_b n}\right) = O\left(n^d \frac{a^{\log_b n}}{b^{d \log_b n}}\right) = O\left(n^d \frac{a^{\log_b n}}{b^{\log_b n^d}}\right)$$
  $$= O\left(n^d \frac{a^{\log_b n}}{n^d}\right) = O(a^{\log_b n}) = O(n^{\log_b a}).$$

  Für den letzten Schritt haben wir die hilfreiche Regel $x^{\log_z y} = y^{\log_z x}$ benutzt, die gilt, da $x^{\log_z y} = z^{\log_z (x^{\log_z y})} = z^{\log_z y \cdot \log_z x} = \left(z^{\log_z y}\right)^{\log_z x} = y^{\log_z x}$.

Mit diesen drei Fällen decken wir also genau die drei Fälle aus der Aussage des Mastertheorems ab.

Am Anfang des Beweises hatten wir die Annahme getroffen, dass n eine Potenz von b ist. Ist dies nicht der Fall, wählen wir als n' die zu n nächstgrößere Potenz von b. Offensichtlich unterscheiden sich n und n' nur um einen konstanten Faktor (kleiner b). Falls nun $a/b^d < 1$, dann ist $T(n) \leq T(n') = O(n'^d) = O(n^d)$. Für die anderen beiden Fälle kann man analog argumentieren. Somit gilt das Mastertheorem also für alle Werte n.

■

Beispiel 1.2 Wir wollen mit dem Mastertheorem die folgende Rekursion asymptotisch abschätzen.

$$T(n) \leq 3T(n/4) + O(\sqrt{n})$$
$$T(1) = O(1).$$

Als Erstes bestimmen wir die Parameter, die wir für die Anwendung des Mastertheorems benötigen. Wir sehen, dass $a = 3$, $b = 4$ und $d = 1/2$. Es folgt, dass $a/b^d = 3/2 > 1$. In diesem Fall liefert uns das Mastertheorem $T(n) = O(n^{\log_4 3}) \subset O(n^{0.793})$.

Test 1.4

Schätzen Sie mit dem Mastertheorem folgende Rekursion asymptotisch ab.

$$T(n) \leq 8T(n/2) + O(n^3)$$
$$T(1) = O(1).$$

An dieser Stelle sei erwähnt, dass es auch alternative Formulierungen des Mastertheorems gibt. So gibt es eine Version, die etwas allgemeiner ist und nicht voraussetzt, dass der nicht-rekursive Aufwand von einem Polynom n^d beschränkt wird, sondern von einer beliebigen Funktion $f(n)$. Für unsere Zwecke genügt jedoch die Abschätzung durch ein Polynom. Eine andere Version erlaubt den Einsatz des Auf- oder Abrundens bei der Größe der Teilprobleme. Wie beim Beweis des Mastertheorems diskutiert, verlieren wir nichts, wenn wir annehmen, dass n eine Potenz von b ist. In diesem Fall würden wir aber auch nie bei der Größe der Teilprobleme runden. Man sieht also, dass das Auf- oder Abrunden keine Auswirkung auf die asymptotische Laufzeit hat. Wir verzichten an dieser Stelle auf eine gesonderte Formulierung des Mastertheorems mit Auf- oder Abrunden, behalten aber die besprochenen Zusammenhänge im Hinterkopf.

1.3.3 Amortisierte Analyse

Um effiziente Algorithmen zu entwerfen, benötigen wir Kenntnisse über effiziente Datenstrukturen. Es ist schwierig, eine klare Linie zwischen Datenstrukturen und Algorithmen zu ziehen, denn Algorithmen verwenden Datenstrukturen und auf der anderen Seite gehören zu jeder Datenstruktur auch ihre Algorithmen. Ein wichtiges Charakteristika von Datenstrukturen ist, dass Anfragen an die (fast) gleiche Datenbasis mehrfach durchgeführt werden. Setzen wir zum Beispiel einen binären Suchbaum ein, wird dieser in der Regel mehrfach angefragt werden, da sich sonst sein Aufbau gar nicht gelohnt hätte.

Die Operationen einer Datenstruktur sind Algorithmen. Bei der Messung der Laufzeit beobachtet man aber oft, dass es Sinn macht, statt einer einzelnen Anfrage, eine Sequenz von Anfragen zu betrachten. Der Grund dafür ist, dass eine einzelne Anfrage unter Umständen aufwendig ist, dies aber in einer beliebigen Sequenz nur selten vorkommen kann. In diesem Fall kann man die Kosten der „teureren Operationen“ auf die anderen Operationen umverteilen. Wir sprechen dann von **amortisierten Kosten** und von einer **amortisierten Analyse**.

Wir wollen an dieser Stelle etwas formaler sein. Sei $q_1, q_2, \ldots q_k$ eine Sequenz von Anfragen an eine Datenstruktur. Die (tatsächlichen) Kosten der Anfrage q_i bezeichnen wir mit c_i. Daneben wählen wir für jede Anfrage q_i die *amortisierten Kosten* $\hat{c}_i$. Diese können größer oder kleiner als die tatsächlichen Kosten sein und wir können sie frei verteilen, solange

$$\sum_{1 \le i \le k} \hat{c}_i \ge \sum_{1 \le i \le k} c_i \tag{1.2}$$

garantiert werden kann, der amortisierte Gesamtaufwand also mindestens so groß ist wie der tatsächliche Aufwand. Auf diese Weise können wir anfallende Kosten einer (teuren) Anfrage auf andere Anfragen umverteilen und so die Gesamtkosten besser abschätzen.

Die amortisierte Analyse interessiert uns in diesem Kurs deshalb, weil es Algorithmen gibt, für die wir die Laufzeit für eine Sequenz von Operationen messen wollen, deren Kosten variieren. In diesem Fall können wir eine genauere Abschätzung vornehmen, wenn wir die Kosten amortisieren.

Wir wollen uns als Beispiel das Aufzählen aller Wörter einer festen Länge n ansehen. Der Einfachheit halber betrachten wir nur Wörter mit den Zeichen `0` und `1`. Ein solches Wort w können wir auch als Binärzahldarstellung einer Zahl interpretieren. Wir wollen von $\texttt{00}\cdots\texttt{0}$ beginnend immer wieder eine 1 zur assoziierten Binärzahl hinzuaddieren, bis wir $\texttt{11}\cdots\texttt{1}$ erreicht haben. Die Zeichen des Wortes w sind in einem Array W gespeichert, wobei wir das i-te Zeichen von rechts in $W[i]$ ablegen.[2] Wir können in einem Schritt (mit konstantem Aufwand) also immer nur eine Stelle der Zeichenkette ändern. Für das binäre Addieren von $+1$ starten wir rechts in der Binärdarstellung und gehen solange eine Position nach links, bis wir eine `0` finden. Dabei ersetzen wir jede gefundene `1` durch eine `0` und die gefundene `0` durch eine `1`. Falls wir keine `0` finden, terminiert der Algorithmus. Das Vorgehen ist in Algorithmus 3 in Pseudocode angegeben.

Algorithmus 3 Addiere 1 zu einer als Array W gespeicherten Binärzahl

```
1: i ← 1
2: while W[i] = 1 do                ▷ W[i] ist i-tes Zeichen in w von rechts
3:     W[i] ← 0
4:     i ← i + 1
5: if i ≤ n then
6:     W[i] = 1
```

Sehen wir uns nun für die ersten Additionen den Aufwand an (wobei wir nur die Anzahl der geänderten Zeichen zählen). Im ersten Schritt ändern wir nur ein Zeichen, denn hier ist das Zeichen am rechten Rand ($W[1]$) gleich `0`. Das gilt insbesondere für jeden zweiten Schritt. Es kann aber auch notwendig sein, mehrere Zeichen zu ändern. Zum Beispiel erfordert der Übergang von $w_1 = \texttt{00111}$ auf $w_2 = \texttt{01000}$ (d.h. von $W_1 = [1,1,1,0,0]$ auf $W_2 = [0,0,0,1,0]$) vier geänderte Zeichen. Tabelle 1.1 zeigt den Aufwand für die ersten Schritte. Die teuersten Schleifendurchläufe erfordern bis zu n Schritte, während viele andere Additionen nur wenige Schritte benötigen. Klar ist, dass wir mindestens 2^n Durchläufe benötigen, um alle 2^n verschiedenen Wörter aufzuzählen.

[2] Auch als „little-endian“-Kodierung bekannt.

aktuelles Wort w	Array-Darstellung W	Aufwand
00000	[0, 0, 0, 0, 0]	1
00001	[1, 0, 0, 0, 0]	2
00010	[0, 1, 0, 0, 0]	1
00011	[1, 1, 0, 0, 0]	3
00100	[0, 0, 1, 0, 0]	1
00101	[1, 0, 1, 0, 0]	2
00110	[0, 1, 1, 0, 0]	1
00111	[1, 1, 1, 0, 0]	4
01000	[0, 0, 0, 1, 0]	1
....		

Tabelle 1.1: Aufwand (Anzahl geänderter Zeichen) beim Hochzählen eines Binärzählers für $n = 5$.

Die Analyse des Aufwandes einer vollständigen Aufzählung kann man mit etwas Überlegung direkt vornehmen (siehe Selbsttest 1.5). Wir wollen dieses Beispiel aber nutzen, um ein klassisches Verfahren für die amortisierte Analyse vorzustellen. Dieses Verfahren heißt **Potentialmethode**. Die Idee hierbei ist, jeder Instanz der Datenstruktur D (in unserem Fall das Array W) eine Zahl zuzuweisen, welche wir das Potential $\Phi(D)$ nennen. Anschaulich gesprochen, kann man das Potential als Summe der bisher vorausgezahlten Kosten interpretieren. Sei $D_0, D_1, D_2, \ldots D_k$ die Sequenz der Datenstruktur(-instanzen), wobei D_i der Zustand nach dem Ausführen von q_i ist. Wir fordern, dass

$$\begin{aligned} &\Phi(D_0) = 0, \\ \forall i: \quad &\Phi(D_i) \geq 0. \end{aligned}$$

Die amortisierten Kosten der i-ten Operation $\hat{c}_i$ ergeben sich aus ihren tatsächlichen Kosten c_i und der Potentialdifferenz, also

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

Dieses Vorgehen ist valide, denn es gilt

$$\begin{aligned} \sum_{1 \leq i \leq k} \hat{c}_i &= \sum_{1 \leq i \leq k} c_i + \Phi(D_i) - \Phi(D_{i-1}) && (\textit{Teleskopsumme}) \\ &= \Phi(D_k) - \Phi(D_0) + \sum_{1 \leq i \leq k} c_i \\ &\geq \sum_{1 \leq i \leq k} c_i \end{aligned}$$

und damit ist die Bedingung (1.2) erfüllt.

Wir wenden die Potentialmethode nun auf unser Beispiel an. Dazu definieren wir

$$\Phi(W) := \text{Anzahl 1en in } W.$$

Somit ist beispielsweise $\Phi([1,0,1,1]) = 3$. Das Abschätzen der amortisieren Kosten ist nun einfach. Hat das Array W_i auf der rechten Seite t_i 1en und davor eine 0, dann verändern wir beim Hochzählen $t_i + 1$ Zeichen, also $c_i = t_i + 1$. Wir ersetzen dabei t_i-mal eine 1 durch eine 0 und einmal eine 0 durch eine 1. Somit verändern wir die Anzahl der 1en um den Wert $-(t_i - 1)$. Wir erhalten

$$\hat{c}_i = c_i + \Phi(W_i) - \Phi(W_{i-1}) = t_i + 1 - (t_i - 1) = 2.$$

Wir sehen somit, dass die amortisierten Kosten beim kompletten Hochzählen des Zählers für einen Aufruf von Algorithmus 3 gleich 2 sind, es ergibt sich also ein konstanter amortisierter Aufwand pro Addition. Die Gesamtkosten für die 2^n Operationen liegen also in $O(2^n)$.

> **Test 1.5**
>
> Führen Sie die amortisierte Analyse des Hochzählens des Binärzählers durch, indem Sie die Kosten aufsummieren und nicht die Potentialmethode nutzen.

1.3.4 Effiziente Algorithmen

In diesem Kurs interessieren wir uns für *effiziente* Algorithmen. Wir möchten an dieser Stelle kurz diskutieren, was wir in diesem Zusammenhang unter *effizient* verstehen. Im Folgenden nutzen wir die einfache und etablierte Konvention:

Effizienter Algorithmus: Wir verstehen einen Algorithmus als *effizient*, wenn dessen (Worst-Case)-Laufzeit sich durch eine polynomielle Schranke asymptotisch abschätzen lässt.

Unsere Konvention ist durchaus diskussionswürdig. Es gibt beispielsweise Aufgaben, für die Algorithmen mit quadratischer Laufzeit keine praktische Relevanz haben, da die zu erwartenden Eingabeinstanzen so groß sind, dass die Berechnungen zu lange dauern würden. Zudem können in der asymptotischen Abschätzung riesige Konstanten versteckt sein, die einen erheblichen Einfluss auf die Laufzeit haben. Diese Einwände sind alle berechtigt. Da wir uns aber vordergründig für die theoretische Analyse der Algorithmen interessieren, spielen die praxisrelevanten Eingabegrößen eine untergeordnete Rolle, weil wir in der Analyse auch keine Kontrolle darüber haben. Gleiches gilt für die verborgenen Konstanten in der Groß-O-Notation. Diese spielen nur dann einen Rolle, wenn die Eingabeinstanzen zu klein sind, und werden bei entsprechend großen Instanzen vernachlässigbar. Es sei zudem daran erinnert, dass polynomielles und exponentielles Wachstum sich stark unterscheiden. Natürlich kann jeder Algorithmus irgendwann an seine Grenzen stoßen, bei einer superpolynomiellen Laufzeit passiert dies jedoch sehr schnell (siehe dazu Tabelle 1.2). Nicht zuletzt beruht unser Verständnis von Effizienz auch auf Erfahrungswerten.

An dieser Stelle sei erwähnt, dass wir nicht erwarten, dass es zu jedem Problem einen effizienten Algorithmus geben wird. Probleme mit einem effizienten Algorithmus bilden die Komplexitätsklasse P. Die Klasse XP enthält hingegen alle Probleme, für die wir einen Algorithmus mit exponentieller Laufzeit kennen. Es gibt auch Probleme, für die wir wissen,

n	$\lfloor n \log n \rfloor$	n^2	n^3	2^n
1	0	1	1	2
5	15	25	125	32
10	40	100	1000	1024
50	300	2500	125000	1125899906842624
100	700	10000	1000000	1267650600228229401496703205376

Tabelle 1.2: Gegenüberstellung von polynomiellem und exponentiellem Wachstum.

dass es überhaupt keinen Algorithmus geben kann. So kann man zum Beispiel die Frage, ob ein gegebenes Programm auf einer bestimmten Eingabe terminiert, algorithmisch nicht universell beantworten (Stichwort Halteproblem).

Eine Sonderstellung haben Probleme, für die es Algorithmen gibt, die eine mögliche Lösung effizient (also in polynomieller Zeit) *verifizieren* können. Für die Verifikation erhalten diese Algorithmen neben der Eingabeinstanz noch eine Hilfe (genannt Zertifikat/Beweis/Zeuge), die jedoch nur polynomielle Länge haben darf. Solche Probleme bilden die Klasse NP, welche eine Teilmenge von XP ist. Bislang ist nicht bekannt, ob P = NP. Man geht jedoch davon aus, dass es Probleme gibt, die in NP liegen und für die es keinen effizienten Algorithmus gibt. Ein prominentes Beispiel ist hierfür die Frage, ob es für ein boolesche Formel eine Belegung der Variablen gibt, sodass die Formel mit dieser Belegung zu `wahr` ausgewertet wird. Dieses *Erfüllbarkeitsproblem (SAT)* genannte Problem ist ein Beispiel für ein NP-vollständiges Problem. NP-vollständige Probleme haben die Eigenschaft, dass sich jedes Problem aus NP auf sie zurückführen lässt, wobei diese Überführung nur polynomiellen Mehraufwand erfordert. Für unsere Überlegung ist es an dieser Stelle nur wichtig, zu wissen, dass wir nicht erwarten, für NP-vollständige Probleme einen effizienten Algorithmus zu finden. Aspekte, wie den Nachweis von NP-Vollständigkeit, werden wir in diesem Kurs nicht beleuchten. Wir gehen aber in späteren Kapiteln darauf ein, welche Strategien es im Umgang mit NP-vollständigen Problemen gibt.

Lernziele

Nach dem Studium dieser Lerneinheit sollten Sie in der Lage sein

- das Berechnungsmodell Word-RAM zu erklären,
- Programme in Pseudocode zu lesen und anzugeben,
- mit asymptotischen unteren und oberen Schranken zu arbeiten,
- für einen in Pseudocode angegebenen Algorithmus die Laufzeitabschätzung vorzunehmen,
- das Mastertheorem wiederzugeben und anzuwenden,
- das Prinzip der amortisierten Analyse inklusive Potentialmethode zu verwenden.

Bibliografische Anmerkungen

Eine umfangreichere Einführung über die historischen Ursprünge von Algorithmen liefert das Buch von Jeff Erickson [4]. Das Modell der Word-RAM geht auf die bahnbrechende Arbeit von Fredman und Willard zu Fusionsbäumen zurück [5]. Eine Übersicht über alternative RAM-Modelle gibt van Emde Boas [3]. Die Groß-O-Notation wurde in der Mathematik um 1900 etabliert, insbesondere durch die Arbeiten von Bachmann [1] und Landau [7]. Das Lehrbuch von Kleinberg und Tardos enthält eine detaillierte Einführung zur asymptotischen Abschätzung [6]. Das Mastertheorem wird im Lehrbuch von Cormen et al. ausführlicher besprochen [2]. Dort findet sich auch ein Beweis für die allgemeinere Formulierung, in welcher der nicht-rekursive Aufwand kein Polynom sein muss. Das Prinzip der amortisieren Analyse wird in einer Arbeit von Tarjan vorgestellt [8].

Lösungsvorschläge zu den Selbsttestaufgaben Kurseinheit 1

Lösungsvorschlag zum Selbsttest 1.1

Wir wählen als Konstante $c = 1000$. Damit erhalten wir $c \cdot f(n) = 1000\, n \log n$. Nun sehen wir, dass

$$\begin{aligned} g(n) \leq c \cdot f(n) &\iff 1000n + 1\,000\,000 \leq 1000\, n \log n \\ &\iff 1000 \leq n(\log n - 1). \end{aligned}$$

Die Funktion $n(\log n - 1)$ ist monoton steigend für $n \geq 2$, und für $n = 256$ ist $n(\log n - 1) = 1792$. Also ist ab $n_0 = 256$ für alle $n \geq n_0$ die Ungleichung $g(n) \leq c \cdot f(n)$ erfüllt, und somit $g = O(f)$.

Lösungsvorschlag zum Selbsttest 1.2

Nach unseren Regeln ist $f_1 = \Theta(n^3)$ und $g_1 = \Theta(n^3)$, also auch $g_1 = \Theta(f_1)$. Die Funktion g_2 können wir auch $g_2(n) = n^{1.5}$ schreiben. Somit ist $g_2 = o(f_2)$ und es folgt, dass $g_2 = O(f_2)$, aber $g_2 \neq \Omega(f_2)$. Weiterhin ist $f_3 = n \log n^2 = 2n \log n$ und $g_3 = n \log n^4 = 4n \log n$. Beide Funktionen unterscheiden sich nur in einer Konstante, und deshalb gilt offensichtlich $g_3(n) = \Theta(f_3(n))$. Ähnliches gilt für das letzte Paar. Nach den Logarithmengesetzen ist $\log_b a = \log_x a / \log_x b$ und damit ist $\log_{10} n = \log_2 n / \log_2 10$. Daraus folgt, dass beide Funktionen sich nur um die multiplikative Konstante $\log_2 10$ unterscheiden. Es folgt, dass $g_4(n) = \Theta(f_4(n))$.

Lösungsvorschlag zum Selbsttest 1.3

Da wir bei MERGE pro rekursivem Aufruf nur einen Unteraufruf benutzen, ist der Rekursionsbaum nur ein Pfad. Abbildung 1.3 zeigt den schematischen Aufbau.

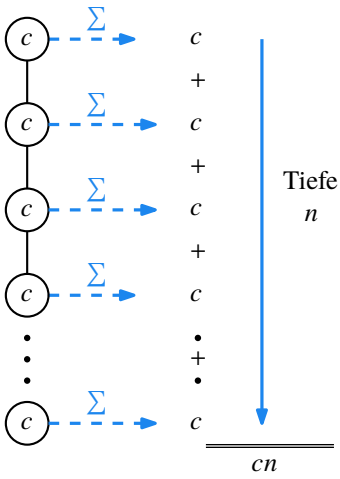

Abbildung 1.3: Rekursionsbaum für MERGE.

Lösungsvorschlag zum Selbsttest 1.4

Wir bestimmen die Werte a, b, d für das Mastertheorem und erhalten $a = 8, b = 2, d = 3$. In diesem Fall gilt also $a/b^d = 8/2^3 = 1$ und wir sind im zweiten Fall des Mastertheorems. Aus diesem Grund gilt $T(n) = O(n^3 \log n)$.

Lösungsvorschlag zum Selbsttest 1.5

Wie in der Potentialmethode beschrieben, benötigen wir einen Aufwand von $t + 1$, wenn $W[t + 1] \neq 1$ und alle Einträge im Array davor eine 1 enthalten. Daraus folgt, dass wir für jede zweite Zahl den Aufwand 1 haben ($t = 0$), jede vierte Zahl hat Aufwand 2 ($t = 1$), und allgemein, jede 2^k-te Zahl hat Aufwand k ($t = k - 1$). Etwas anders formuliert können wir auch sagen, dass jede Zahl einen Mindestaufwand von 1 hat, für jede zweite Zahl noch ein Aufwand von 1 dazukommt, für jede vierte Zahl kommt wiederum ein Aufwand von 1 dazu usw. Somit erhalten wir als Gesamtaufwand für die 2^n Operationen

$$2^n + 2^n/2 + 2^n/4 + 2^n/8 \cdots \leq \sum_{k=0}^{n} 2^n/2^k < 2^n\, 2 = 2^{n+1}.$$

Legt man die Gesamtkosten gleichmäßig auf die einzelnen Operationen um, ergibt sich ein amortisierter Aufwand von 2.

Literaturverzeichnis

[1] Paul Bachmann: *Analytische Zahlentheorie*. B. G. Teubner, 1894.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein: *Introduction to Algorithms*. MIT Press, 3. Auflage, 2009.

[3] Peter van Emde Boas: *Machine models and simulation*. In: Jan van Leeuwen (Herausgeber): *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, Seiten 1–66. Elsevier & MIT Press, 1990.

[4] Jeff Erickson: *Algorithms*. Eigenverlag, 1. Auflage, 2019. `http://algorithms.wtf`.

[5] Michael L. Fredman und Dan E. Willard: *Surpassing the information theoretic bound with fusion trees*. J. Comput. Syst. Sci., 47(3):424–436, 1993.

[6] Jon M. Kleinberg und Éva Tardos: *Algorithm design*. Addison-Wesley, 2006.

[7] Edmund Landau: *Handbuch der Lehre von der Verteilung der Primzahlen*. B. G. Teubner, 1909.

[8] Robert E. Tarjan: *Amortized computational complexity*. SIAM Journal on Algebraic and Discrete Methods, 6(2):306–318, 1985.