13 QUANTITATIVE ASPEKTE VON ALGORITHMEN

13.1 RESSOURCENVERBRAUCH BEI BERECHNUNGEN

Wir haben in Einheit 12 mit ersten Graphalgorithmen damit begonnen, festzustellen, wieviele arithmetische Operationen bei der Ausführung eines Algorithmus für eine konkrete Probleminstanz ausgeführt werden. Zum Beispiel hatten wir angemerkt, dass bei der Addition zweier $n \times n$-Matrizen mittels zweier ineinander geschachtelten **for**-Schleifen n^2 Additionen notwendig sind. Das war auch als ein erster Schritt gedacht in Richtung der Abschätzung von Laufzeiten von Algorithmen.

Laufzeit, Rechenzeit
Ressource
Speicherplatzbedarf
Komplexitätsmaß

Rechenzeit ist wohl die am häufigsten untersuchte *Ressource*, die von Algorithmen „verbraucht" wird. Eine zweite ist der *Speicherplatzbedarf*. Man spricht in diesem Zusammenhang auch von *Komplexitätsmaßen*. Sie sind Untersuchungsgegenstand in Vorlesungen über Komplexitätstheorie (engl. *computational complexity*) und tauchen darüber hinaus in vielen Gebieten (und Vorlesungen) zur Beurteilung der Qualität von Algorithmen auf.

Hauptgegenstand dieser Einheit wird es sein, das wichtigste Handwerkszeug bereitzustellen, das beim Reden über und beim Ausrechnen von z. B. Laufzeiten hilfreich ist und in der Literatur immer wieder auftaucht, insbesondere die sogenannte Groß-O-Notation.

Dazu sei als erstes noch einmal explizit daran erinnert, dass wir in Einheit 5 zum informellen Algorithmusbegriff festgehalten haben, dass ein Algorithmus für Eingaben beliebiger Größe funktionieren sollte: Ein Algorithmus zur Multiplikation von Matrizen sollte nicht nur für 3×3-Matrizen oder 42×42-Matrizen funktionieren, sollte für Matrizen mit beliebiger Größe $n \times n$. Es ist aber „irgendwie klar", dass dann die Laufzeit keine Konstante sein kann, sondern eine Funktion ist, die zumindest von n abhängt. Und zum Beispiel bei Algorithmus 12.1 zur Bestimmung der Wegematrix eines Graphen hatte auch nichts anderes Einfluss auf die Laufzeit.

Aber betrachten wir als ein anderes Beispiel das Sortieren von Zahlen und z. B. den Insertionsort-Algorithmus aus der Programmieren-Vorlesung, den wir in Algorithmus 13.1 noch mal aufgeschrieben haben. Wie oft die **while**-Schleife in der Methode *insert* ausgeführt wird, hängt nicht einfach von der Problemgröße $n = a.length$ ab. Es hängt auch von der konkreten Probleminstanz ab. Selbst bei gleicher Zahl n kann die Schleife unterschiedlich oft durchlaufen werden: Ist das Array a von Anfang an sortiert, wird die **while**-Schleife überhaupt nicht ausgeführt. Ist es genau in entgegengesetzter Richtung sortiert, wird ihr Schleifenrumpf insgesamt $\sum_{i=1}^{n-1} i = n(n-1)/2$ mal ausgeführt.

Meistens ist es aber so, dass man nicht für jede Probleminstanz einzeln angeben will oder kann, wie lange ein Algorithmus zu seiner Lösung benötigt. Man beschränkt sich auf eine vergröbernde Sichtweise und beschreibt z. B. die Laufzeit nur in

Algorithmus 13.1 Insertionsort aus der Vorlesung „Programmieren“

```
public class InsertionSort {
  public static void sort(long[] a) {
    for (int i ← 1; i < a.length; i++) {
      insert(a, i);
    }
  }
  private static void insert(long[] a, int idx) {
    int i ← idx;
    // Tausche a[idx] nach links bis es einsortiert ist
    while (i > 0 ∧ a[i − 1] > a[i]) {
      long tmp ← a[i − 1];
      a[i − 1] ← a[i];
      a[i] ← tmp;
      i − −;
    }
  }
}
```

Abhängigkeit von der Problemgröße n. Es bleibt die Frage zu klären, was man dann angibt, wenn die Laufzeit für verschiedene Instanzen gleicher Größe variiert: Den Durchschnitt? Den schnellsten Fall? Den langsamsten Fall?

Am weitesten verbreitet ist es, als Funktionswert für jede Problemgröße n den jeweils schlechtesten Fall (engl. *worst case*) zu nehmen. Eine entsprechende Analyse eines Algorithmus ist typischerweise deutlich einfacher als die Berechnung von Mittelwerten (engl. *average case*), und wir werden uns jedenfalls in dieser Vorlesung darauf beschränken.

worst case

average case

13.2 GROSS-O-NOTATION

Die Aufgabe besteht also im allgemeinen darin, bei einem Algorithmus für jede mögliche Eingabegröße n genau anzugeben, wie lange der Algorithmus für Probleminstanzen der Größe n im schlimmsten Fall zur Berechung des Ergebnisses benötigt. Leider ist es manchmal so, dass man die exakten Werte nicht bestimmen will oder kann.

Dass man es nicht *will*, muss nicht unbedingt Ausdruck von Faulheit sein. Vielleicht sind gewisse Ungenauigkeiten (die man noch im Griff hat) für das Verständnis nicht nötig. Oder die genauen Werte hängen von Umständen ab, die sich ohnehin „bald“ ändern. Man denke etwa an die recht schnelle Entwicklung bei Prozessoren in den vergangenen Jahren. Dass ein Programm für gewisse Eingaben auf einem bestimmten Prozessor soundso lange braucht, ist unter Umständen schon nach wenigen Monaten uninteressant, weil ein neuer Prozessor viel schneller ist. Das

muss nicht einfach an einer höheren Taktrate liegen (man könnte ja auch einfach Takte zählen statt Nanosekunden), sondern z. B. an Verbesserungen bei der Prozessorarchitektur.

Darüber hinaus *kann* man die Laufzeit eines Algorithmus mitunter gar nicht exakt abschätzen. Manchmal könnte man es vielleicht im Prinzip, aber man ist zu dumm. Oder die vergröbernde Darstellung nur der schlechtesten Fälle führt eben dazu, dass die Angabe für andere Fälle nur eine obere Schranke ist. Oder man erlaubt sich bei der Formulierung von Algorithmen gewisse Freiheiten bzw. Ungenauigkeiten, die man (vielleicht wieder je nach Prozessorarchitektur) unterschiedlich bereinigen kann, weil man eben an Aussagen interessiert ist, die von Spezifika der Prozessoren unabhängig sind.

Damit haben wir zweierlei angedeutet:

- Zum einen werden wir im weiteren Verlauf dieses Abschnittes eine Formulierungshilfe bereitstellen, die es erlaubt, in einem gewissen überschaubaren Rahmen ungenau über Funktionen zu reden.
- Zum anderen werden wir in einer späteren Einheit auf Modelle für Rechner zu sprechen kommen. Ziel wird es sein, z. B. über Laufzeit von Algorithmen in einer Weise reden zu können, die unabhängig von konkreten Ausprägungen von Hardware und trotzdem noch aussagekräftig ist.

13.2.1 *Ignorieren konstanter Faktoren*

Wie ungenau wollen wir über Funktionen reden?

Ein erster Aspekt wird dadurch motiviert, dass man das so tun möchte, dass z. B. Geschwindigkeitssteigerungen bei Prozessoren irrelevant sind. Etwas genauer gesagt sollen konstante Faktoren beim Wachstum von Funktionen keine Rolle spielen.

Wir bezeichnen im folgenden mit $\mathbb{R}^+$ die Menge der positiven reellen Zahlen (also *ausschließlich* 0) und mit $\mathbb{R}_0^+$ die Menge der nichtnegativen rellen Zahlen, also $\mathbb{R}_0^+ = \mathbb{R}^+ \cup \{0\}$. Wir betrachten Funktionen $f : \mathbb{N}_0 \to \mathbb{R}_0^+$.

asymptotisches Wachstum
größenordnungsmäßiges Wachstum

Wir werden im folgenden vom *asymptotischen Wachstum* oder auch *größenordnungsmäßigen Wachstum* von Funktionen sprechen (obwohl es das Wort „größenordnungsmäßig“ im Deutschen gar nicht gibt — zumindest steht es nicht im Duden; betrachten wir es als Terminus technicus). Eine Funktion $g : \mathbb{N}_0 \to \mathbb{R}_0^+$ wächst *größenordnungsmäßig genauso schnell* wie eine Funktion $f : \mathbb{N}_0 \to \mathbb{R}_0^+$, wenn gilt:

$$\exists c, c' \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geqslant n_0 : cf(n) \leqslant g(n) \leqslant c'f(n) \; .$$

$f \asymp g$

Wir schreiben in diesem Fall auch $f \asymp g$ oder $f(n) \asymp g(n)$. Zum Beispiel gilt $3n^2 \asymp 10^{-2}n^2$. Denn einerseits gilt für $c = 10^{-3}$ und $n_0 = 0$:

$$\forall n \geqslant n_0 : cf(n) = 10^{-3} \cdot 3n^2 \leqslant 10^{-2}n^2 = g(n)$$

Andererseits gilt z. B. für $c' = 1$ und $n_0 = 0$:

$$\forall n \geqslant n_0 : g(n) = 10^{-2}n^2 \leqslant 3n^2 = c'f(n)$$

Die eben durchgeführte Rechnung lässt sich leicht etwas allgemeiner durchführen. Dann zeigt sich, dass man festhalten kann:

13.1 (Rechenregel) Für alle $f : \mathbb{N}_0 \to \mathbb{R}_0^+$ gilt:

$$\forall a, b \in \mathbb{R}^+ : af(n) \asymp bf(n)$$

Damit können wir uns als zweites Beispiel $f(n) = n^3 + 5n^2$ und $g(n) = 3n^3 - n$ ansehen und nun schon recht leicht einsehen, dass $f \asymp g$ ist. Denn einerseits ist für $n \geqslant 0$ offensichtlich $f(n) = n^3 + 5n^2 \leqslant n^3 + 5n^3 = 6n^3 = 9n^3 - 3n^3 \leqslant 9n^3 - 3n = 3(3n^3 - n) = 3g(n)$. Andererseits ist $g(n) = 3n^3 - n \leqslant 3n^3 \leqslant 3(n^3 + 5n^2) = 3f(n)$.

Es gibt auch Funktionen, für die $f \asymp g$ nicht gilt. Als einfaches Beispiel betrachten wir $f(n) = n^2$ und $g(n) = n^3$. Die Bedingung $g(n) \leqslant c'f(n)$ aus der Definition ist für $f(n) \neq 0$ gleichbedeutend mit $g(n)/f(n) \leqslant c'$. Damit $f \asymp g$ gilt, muss insbesondere $g(n)/f(n) \leqslant c'$ für ein $c' \in \mathbb{R}^+$ ab einem n_0 für alle n gelten. Es ist aber $g(n)/f(n) = n$, und das kann durch keine Konstante beschränkt werden.

Das Zeichen $\asymp$ erinnert an das Gleichheitszeichen. Das ist auch bewusst so gemacht, denn die Relation $\asymp$ hat wichtige Eigenschaften, die auch auf Gleichheit zutreffen:

13.2 Lemma. Die Relation $\asymp$ ist eine Äquivalenzrelation.

13.3 Beweis. Wir überprüfen die drei definierenden Eigenschaften von Äquivalenzrelationen (siehe Unterabschnitt 11.2.1).

- *Reflexitvität*: Es ist stets $f \asymp f$, denn man $c = c' = 1$ und $n_0 = 0$ wählt, dann gilt für $n \geqslant n_0$ offensichtlich $cf(n) \leqslant f(n) \leqslant c'f(n)$.
- *Symmetrie:* Wenn $f \asymp g$ gilt, dann auch $g \asymp f$: Denn wenn für positive Konstanten c, c' und alle $n \geqslant n_0$

  $$cf(n) \leqslant g(n) \leqslant c'f(n)$$

  gilt, dann gilt für die gleichen $n \geqslant n_0$ und die ebenfalls positiven Konstanten $d = 1/c$ und $d' = 1/c'$:

  $$d'g(n) \leqslant f(n) \leqslant dg(n)$$

- *Transitivität:* Wenn $f \asymp g$ ist, und $g \asymp h$, dann ist auch $f \asymp h$: Es gelte für Konstanten $c, c' \in \mathbb{R}_+$ und alle $n \geqslant n_0$

  $$cf(n) \leqslant g(n) \leqslant c'f(n)$$

  und analog für Konstanten $d, d' \in \mathbb{R}_+$ und alle $n \geqslant n_1$

  $$dg(n) \leqslant h(n) \leqslant d'g(n) \,.$$

  Dann gilt für alle $n \geqslant \max(n_0, n_1)$

  $$dcf(n) \leqslant dg(n) \leqslant h(n) \leqslant d'g(n) \leqslant d'c'f(n) \,,$$

  wobei auch die Konstanten dc und $d'c'$ wieder positiv sind.

■

Es ist üblich, für die Menge aller Funktionen, die zu einer gegebenen Funktion $f(n)$ im Sinne von $\asymp$ äquivalent sind, $\Theta(f)$ bzw. $\Theta(f(n))$ zu schreiben. Also: $\Theta(f)$

$$\begin{aligned}\Theta(f) &= \{g \mid f \asymp g\} \\ &= \{g \mid \exists c, c' \in \mathbb{R}^+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geqslant n_0 : cf(n) \leqslant g(n) \leqslant c'f(n)\}\end{aligned}$$

Aus Rechenregel 13.1 wird bei Verwendung von Θ:

13.4 (Rechenregel) Für alle $f : \mathbb{N}_0 \to \mathbb{R}_0^+$ und alle Konstanten $a, b \in \mathbb{R}_0^+$ gilt: $\Theta(af(n)) = \Theta(bf(n))$.

13.2.2 *Notation für obere und untere Schranken des Wachstums*

In Fällen wie der unbekannten Anzahl von Durchläufen der **while**-Schleife in Algorithmus 13.1 genügt es nicht, wenn man konstante Faktoren ignorieren kann. Man kennt nur den schlimmsten Fall: $\sum_{i=1}^{n-1} i = n(n-1)/2$. Dementsprechend ist im schlimmsten Fall die Laufzeit in $\Theta(n^2)$; im allgemeinen kann sie aber auch kleiner sein. Um das bequem ausdrücken und weiterhin konstante Faktoren ignorieren zu können, definiert man: $O(f)$, $\Omega(f)$

$$\begin{aligned}O(f(n)) &= \{g(n) \mid \exists c \in \mathbb{R}^+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geqslant n_0 : g(n) \leqslant cf(n)\} \\ \Omega(f(n)) &= \{g(n) \mid \exists c \in \mathbb{R}^+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geqslant n_0 : g(n) \geqslant cf(n)\}\end{aligned}$$

Man schreibt gelegentlich auch $g \preceq f$, $g \succeq f$

$$\begin{aligned}g \preceq f &\text{ falls } g \in O(f) \\ g \succeq f &\text{ falls } g \in \Omega(f)\end{aligned}$$

und sagt, dass g *asymptotisch höchstens so schnell wie* f wächst (falls $g \preceq f$) bzw. dass g *asymptotisch mindestens so schnell wie* f wächst (falls $g \succeq f$).

Betrachten wir drei Beispiele.

- Es ist $10^{90}n^7 \in O(10^{-90}n^8)$, denn für $c = 10^{180}$ ist für alle $n \geqslant 0$: $10^{90}n^7 \leqslant c \cdot 10^{-90}n^8$.
  Dieses Beispiel soll noch einmal deutlich machen, dass man in $O(\cdot)$ usw. richtig große Konstanten „verstecken" kann. Ob ein hypthetischer Algorithmus mit Laufzeit in $O(n^8)$ in der Praxis wirklich tauglich ist, hängt durchaus davon ab, ob die Konstante c bei der oberen Schranke cn^8 eher im Bereich 10^{-90} oder im Bereich 10^{90} ist.
- Mitunter trifft man auch die Schreibweise $O(1)$ an. Was ist das? Die Definition sagt, dass das alle Funktionen $g(n)$ sind, für die es eine Konstante $c \in \mathbb{R}^+$ gibt und ein $n_0 \in \mathbb{N}_0$, so dass für alle $n \geqslant n_0$ gilt: $O(1)$

  $$g(n) \leqslant c \cdot 1 = c$$

  Das sind also alle Funktionen, die man durch Konstanten beschränken kann. Dazu gehören etwa alle konstanten Funktionen, aber auch Funktionen wie $3 + \sin(n)$. (So etwas habe ich aber noch nie eine Rolle spielen sehen.)

- Weiter vorne hatten wir benutzt, dass der Quotient n^2/n nicht für alle hinreichend großen n durch eine Konstante beschränkt werden kann. Also gilt *nicht* $n^2 \preceq n$. Andererseits gilt (machen Sie sich das bitte kurz klar) $n \preceq n^2$. Die Relation $\preceq$ ist also *nicht* symmetrisch.
  Allgemein gilt für positive reelle Konstanten $a < b$, dass $n^a \preceq n^b$ ist, aber nicht umgekehrt.

Man muss nur in der Ungleichung $g(n) \leqslant cf(n)$ die Konstante auf die andere Seite bringen und schon kann man sich davon überzeugen, dass gilt:

13.5 (Rechenregel) Für alle Funktionen $f : \mathbb{N}_0 \to \mathbb{R}_0^+$ und $g : \mathbb{N}_0 \to \mathbb{R}_0^+$ gilt:

$$g(n) \in O(f(n)) \iff f(n) \in \Omega(g(n)), \quad \text{also} \quad g \preceq f \iff f \succeq g$$

Man kann auch zeigen:

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$
$$\text{also} \qquad g \asymp f \iff g \preceq f \wedge g \succeq f$$

Das war in diesem Semester erst als Übungsaufgabe gedacht. Nun ist es doch keine.

13.2.3 *Die furchtbare Schreibweise*

Damit Sie bei Lektüre von Büchern und Aufsätzen alles verstehen, was dort mit Hilfe von $\Theta(\cdot)$, $O(\cdot)$ und $\Omega(\cdot)$ aufgeschrieben steht, müssen wir Ihnen nun leider noch etwas mitteilen. Man benutzt eine (unserer Meinung nach) sehr unschöne (um nicht zu sagen irreführende) Variante der eben eingeführten Notation. Aber weil sie so verbreitet ist, muten wir sie Ihnen zu. Man schreibt nämlich

$$\begin{aligned} g(n) &= O(f(n)) \quad \text{statt} \quad g(n) \in O(f(n)), \\ g(n) &= \Theta(f(n)) \quad \text{statt} \quad g(n) \in \Theta(f(n)), \\ g(n) &= \Omega(f(n)) \quad \text{statt} \quad g(n) \in \Omega(f(n)). \end{aligned}$$

Die Ausdrücke auf der linken Seite sehen zwar aus wie Gleichungen, *aber es sind keine!* Lassen Sie daher bitte immer *große Vorsicht* walten:

- Es ist *falsch*, aus $g(n) = O(f_1(n))$ und $g(n) = O(f_2(n))$ zu folgern, dass $O(f_1(n)) = O(f_2(n))$ ist.
- Es ist *falsch*, aus $g_1(n) = O(f(n))$ und $g_2(n) = O(f(n))$ zu folgern, dass $g_1(n) = g_2(n)$ ist.

Noch furchtbarer ist, dass manchmal etwas von der Art $O(g) = O(f)$ geschrieben wird, *aber nur die Inklusion* $O(g) \subseteq O(f)$ *gemeint ist.*

Auch Ronald Graham, Donald Knuth und Oren Patashnik sind nicht begeistert, wie man den Ausführungen auf den Seiten 432 und 433 ihres Buches *Concrete Mathematics* (Graham, Knuth und Patashnik 1989) entnehmen kann. Sie geben vier Gründe an,

warum man das doch so macht. Der erste ist Tradition; der zweite ist Tradition; der dritte ist Tradition. Der vierte ist, dass die Gefahr, etwas falsch zu machen, oft eher klein ist. Also dann: toi toi toi.

13.2.4 *Rechnen im O-Kalkül*

Ist $g_1 \preceq f_1$ und $g_2 \preceq f_2$, dann ist auch $g_1 + g_2 \preceq f_1 + f_2$. Ist umgekehrt $g \preceq f_1 + f_2$, dann kann man g in der Form $g = g_1 + g_2$ schreiben mit $g_1 \preceq f_1$ und $g_2 \preceq f_2$. Das schreiben wir auch so:

13.6 Lemma. Für alle Funktionen $f_1, f_2 : \mathbb{N}_0 \to \mathbb{R}_0^+$ gilt:

$$O(f_1) + O(f_2) = O(f_1 + f_2)$$

Dabei muss allerdings erst noch etwas definiert werden: die „Summe“ von Mengen (von Funktionen). So etwas nennt man manchmal *Komplexoperationen* und definiert sie so: Sind M_1 und M_2 Mengen von Elementen, die man addieren bzw. multiplizieren kann, dann sei

Komplexoperationen

$$M_1 + M_2 = \{g_1 + g_2 \mid g_1 \in M_1 \wedge g_2 \in M_2\}$$
$$M_1 \cdot M_2 = \{g_1 \cdot g_2 \mid g_1 \in M_1 \wedge g_2 \in M_2\}$$

Für Funktionen sei Addition und Multiplikation (natürlich?) argumentweise definiert. Dann ist z. B.

$$O(n^3) + O(n^3) = \{g_1(n) + g_2(n) \mid g_1 \in O(n^3) \wedge g_2 \in O(n^3)\}$$

z. B.

$$(2n^3 - n^2) + 7n^2 = 2n^3 + 6n^2 \in O(n^3) + O(n^3)$$

Wenn eine der Mengen M_i einelementig ist, lässt man manchmal die Mengenklammern darum weg und schreibt zum Beispiel bei Zahlenmengen

$$\text{statt} \qquad \{3\} \cdot \mathbb{N}_0 + \{1\} \qquad \text{kürzer} \qquad 3\mathbb{N}_0 + 1$$

oder bei Funktionenmengen

$$\text{statt} \qquad \{n^3\} + O(n^2) \qquad \text{kürzer} \qquad n^3 + O(n^2)$$

Solche Komplexoperationen sind übrigens nichts Neues für Sie. Die Definition des Produkts formaler Sprachen passt genau in dieses Schema (siehe Unterabschnitt **??**).

13.7 Beweis. (von Lemma 13.6) Wir beweisen die beiden Inklusionen getrennt.

„$\subseteq$“: Wenn $g_1 \in O(f_1)$, dann existiert ein $c_1 \in \mathbb{R}^+$ und ein n_{01}, so dass für alle $n \geqslant n_{01}$ gilt: $g_1(n) \leqslant c_1 f_1(n)$. Und wenn $g_2 \in O(f_2)$, dann existiert ein $c_2 \in \mathbb{R}^+$ und ein n_{02}, so dass für alle $n \geqslant n_{02}$ gilt: $g_2(n) \leqslant c_2 f_2(n)$.

Folglich gilt für alle $n \geqslant n_0 = \max(n_{01}, n_{02})$ und für $c = \max(c_1, c_2) \in \mathbb{R}_0^+$:

$$\begin{aligned} g_1(n) + g_2(n) &\leqslant c_1 f_1(n) + c_2 f_2(n) \\ &\leqslant c f_1(n) + c f_2(n) \\ &= c(f_1(n) + f_2(n)) \end{aligned}$$

„$\supseteq$“: Wenn $g \in O(f_1 + f_2)$ ist, dann gibt es $c \in \mathbb{R}^+$ und ein n_0, so dass für alle $n \geqslant n_0$ gilt: $g(n) \leqslant c(f_1(n) + f_2(n))$.

Man definiere nun eine Funktion $g_1 : \mathbb{N}_0 \to \mathbb{R}_0^+$ vermöge

$$g_1(n) = \begin{cases} g(n) & \text{falls } g(n) \leqslant c f_1(n) \\ c f_1(n) & \text{falls } g(n) > c f_1(n) \end{cases}$$

Dann ist offensichtlich $g_1 \in O(f_1)$.

Außerdem ist $g_1(n) \leqslant g(n)$ und folglich $g_2(n) = g(n) - g_1(n)$ stets größer gleich 0. Behauptung: $g_2 \in O(f_2)$. Sei $n \geqslant n_0$. Dann ist

$$\begin{aligned} g_2(n) &= g(n) - g_1(n) \\ &= \begin{cases} 0 & \text{falls } g(n) \leqslant c f_1(n) \\ g(n) - c f_1(n) & \text{falls } g(n) > c f_1(n) \end{cases} \\ &\leqslant \begin{cases} 0 & \text{falls } g(n) \leqslant c f_1(n) \\ c(f_1(n) + f_2(n)) - c f_1(n) & \text{falls } g(n) > c f_1(n) \end{cases} \\ &= \begin{cases} 0 & \text{falls } g(n) \leqslant c f_1(n) \\ c f_2(n) & \text{falls } g(n) > c f_1(n) \end{cases} \\ &\leqslant c f_2(n) \,, \end{aligned}$$

also $g_2 \in O(f_2)$. Also ist $g = g_1 + g_2 \in O(f_1) + O(f_2)$.

■

13.8 (Rechenregel) Wenn $g_1 \preceq f_1$ ist, und wenn $g_1 \asymp g_2$ und $f_1 \asymp f_2$, dann gilt auch $g_2 \preceq f_2$.

13.9 (Rechenregel) Wenn $g \preceq f$ ist, also $g \in O(f)$, dann ist auch $O(g) \subseteq O(f)$ und $O(g + f) = O(f)$.

Es gibt noch eine Reihe weiterer Rechenregeln für $O(\cdot)$ und außerdem ähnliche für $\Theta(\cdot)$ und $\Omega(\cdot)$ (zum Beispiel Analoga zu Lemma 13.6). Wir verzichten hier darauf, sie alle aufzuzählen.

13.3 MATRIXMULTIPLIKATION

Wir wollen uns nun noch einmal ein bisschen genauer mit der Multiplikation von $n \times n$-Matrizen beschäftigen, und uns dabei insbesondere für

- die Anzahl $N_{add}(n)$ elementarer Additionen ist und
- die Anzahl $N_{mult}(n)$ elementarer Multiplikationen

interessieren. Deren Summe bestimmt im wesentlichen (d. h. bis auf konstante Faktoren) die Laufzeit.

13.3.1 *Rückblick auf die Schulmethode*

Die „Schulmethode“ für die Multiplikation von 2×2-Matrizen geht so:

		b_{11}	b_{12}
		b_{21}	b_{22}
a_{11}	a_{12}	$a_{11}b_{11} + a_{12}b_{21}$	$a_{11}b_{12} + a_{12}b_{22}$
a_{21}	a_{22}	$a_{21}b_{11} + a_{22}b_{21}$	$a_{21}b_{12} + a_{22}b_{22}$

Wie man sieht ist dabei

- $N_{mult}(2) = 2^2 \cdot 2 = 8$ und
- $N_{add}(2) = 2^2 \cdot (2-1) = 4$.

Wenn n gerade ist (auf diesen Fall wollen uns im folgenden der einfacheren Argumentation wegen beschränken), dann ist die Schulmethode für $n \times n$ Matrizen äquivalent zum Fall, dass man 2×2 Blockmatrizen mit Blöcken der Größe $n/2$ vorliegen hat, die man nach dem gleichen Schema wie oben multiplizieren kann:

		B_{11}	B_{12}
		B_{21}	B_{22}
A_{11}	A_{12}	$A_{11}B_{11} + A_{12}B_{21}$	$A_{11}B_{12} + A_{12}B_{22}$
A_{21}	A_{22}	$A_{21}B_{11} + A_{22}B_{21}$	$A_{21}B_{12} + A_{22}B_{22}$

Das sind 4 Additionen von Blockmatrizen und 8 Multiplikationen von Blockmatrizen. Die Anzahl elementarer Operationen ist also

- $N_{mult}(n) = 8 \cdot N_{mult}(n/2)$ und
- $N_{add}(n) = 8 \cdot N_{add}(n/2) + 4 \cdot (n/2)^2 = 8 \cdot N_{add}(n/2) + n^2$.

Wir betrachten den Fall $n = 2^k$ (die anderen Fälle gehen im Prinzip ähnlich). Dann ergibt sich aus $N_{mult}(n) = 8 \cdot N_{mult}(n/2)$:

$$\begin{aligned} N_{mult}(2^k) &= 8 \cdot N_{mult}(2^{k-1}) = 8 \cdot 8 \cdot N_{mult}(2^{k-2}) = \cdots = 8^k \cdot N_{mult}(1) \\ &= 8^k = 8^{\log_2(n)} = 2^{3\log_2(n)} = 2^{\log_2(n)\cdot 3} = n^3 \end{aligned}$$

Dass man statt der Pünktchen einen Induktionsbeweis führen kann, ist Ihnen inzwischen klar, richtig?

Aus $N_{add}(n) = 8 \cdot N_{add}(n/2) + n^2$ ergibt sich analog:

$$\begin{aligned} N_{add}(2^k) &= 8 \cdot N_{add}(2^{k-1}) + 4^k \\ &= 8 \cdot 8 \cdot N_{add}(2^{k-2}) + 8 \cdot 4^{k-1} + 4^k = \cdots \\ &= 8 \cdot 8 \cdot N_{add}(2^{k-2}) + 2 \cdot 4^k + 4^k = \cdots \\ &= 8^k N_{add}(2^0) + (2^{k-1} + \cdots 1) \cdot 4^k = \\ &= 2^k \cdot 4^k \cdot 1 + (2^k - 1) \cdot 4^k = \\ &= 2 \cdot 2^k \cdot 4^k - 4^k = 2n^3 - n^2 \end{aligned}$$

13.3.2 *Algorithmus von Strassen*

Nun kommen wir zu der Idee von Strassen (1969). Er hat bemerkt, dass man die Blockmatrizen C_{ij} des Matrixproduktes auch wie folgt berechnen kann:

$$\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}$$

und dann

$$\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{aligned}$$

Das sieht erst einmal umständlicher aus, denn es sind 18 Additionen von Blockmatrizen statt nur 4 bei der Schulmethode. Aber es sind nur 7 Multiplikationen von Blockmatrizen statt 8! Und das zahlt sich aus, denn im Gegensatz zum skalaren Fall sind Multiplikationen aufweniger als Additionen. Für die Anzahl elementarer Operationen ergibt sich:

- $N_{mult}(n) = 7 \cdot N_{mult}(n/2)$
- $N_{add}(n) = 7 \cdot N_{add}(n/2) + 18 \cdot (n/2)^2 = 7 \cdot N_{add}(n/2) + 4.5 \cdot n^2$

Für den Fall $n = 2^k$ ergibt sich:

$$\begin{aligned}
N_{mult}(2^k) &= 7 \cdot N_{mult}(2^{k-1}) = 7 \cdot 7 \cdot N_{mult}(2^{k-2}) = \cdots = 7^k \cdot N_{mult}(1) \\
&= 7^k = 7^{\log_2(n)} = 2^{\log_2 7 \cdot \log_2(n)} = n^{\log_2 7} \approx n^{2.807\ldots}
\end{aligned}$$

Analog erhält man auch für die Anzahl der Additionen $N_{add}(n) \in \Theta(n^{\log_2 7})$. Die Gesamtzahl elementarer arithmetischer Operationen ist also in $\Theta(n^{\log_2 7}) + \Theta(n^{\log_2 7}) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807\ldots})$.

Es gibt sogar Algorithmen, die asymptotisch noch weniger Operationen benötigen. Das in dieser Hinsicht beste Ergebnis stammt von Coppersmith und Winograd (1990), die mit $O(n^{2.376\ldots})$ elementaren arithmetischen Operationen auskommen. Auch dieses Verfahren benutzt wie das von Strasse eine Vorgehensweise, die man in vielen Algorithmen wiederfindet: Man teilt die Probleminstanz in kleinere Teile auf, die man wie man sagt rekursiv nach dem gleichen Verfahren bearbeitet und die Teilergebnisse dann benutzt, um das Resultat für die ursprüngliche Eingabe zu berechnen. Man spricht von *„teile und herrsche“* (engl. *divide and conquer*).

Sie werden im Laufe der kommenden Semester viele Algorithmen kennenlernen, bei denen wie bei Strassens Algorithmus für Matrixmultiplikation das Prinzip „Teile und Herrsche“ benutzt wird. In den einfacheren Fällen muss man zur Bearbeitung eines Problems der Größe n eine konstante Anzahl a von Teilprobleme gleicher Größe n/b lösen. Die zusätzlichen Kosten zur Berechnung des eigentlichen Ergebnisses mögen zusätzlich einen Aufwand $f(n)$ kosten. Das beinhaltet auch den unter Umständen erforderlichen Aufwand zum Erzeugen der Teilprobleme.

Dann ergibt sich für Abschätzung (z. B.) der Laufzeit $T(n)$ eine Rekursionsformel, die grob gesagt von der Form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

ist. Dabei ist sinnvollerweise $a \geqslant 1$ und $b > 1$.

Obige Rekursionsformel ist unpräzise, denn Problemgrößen sind immer ganzzahlig, n/b im allgemeinen aber nicht. Es zeigt sich aber, dass sich jedenfalls in den nachfolgend aufgeführten Fällen diese Ungenauigkeit im folgenden Sinne nicht auswirkt: Wenn man in der Rekursionsformel n/b durch $\lfloor n/b \rfloor$ oder durch $\lceil n/b \rceil$ ersetzt oder gar durch $\lfloor n/b + c \rfloor$ oder durch $\lceil n/b + c \rceil$ für eine Konstante c, dann behalten die folgenden Aussagen ihre Gültigkeit.

Wenn zwischen den Konstanten a und b und der Funktion $f(n)$ gewisse Zusammenhänge bestehen, dann kann man ohne viel Rechnen (das schon mal jemand anders für uns erledigt hat) eine Aussage darüber machen, wie stark $T(n)$ wächst.

Es gibt drei wichtige Fälle, in denen jeweils die Zahl $\log_b a$ eine Rolle spielt:

FALL 1: Wenn $f(n) \in O(n^{\log_b a - \varepsilon})$ für ein $\varepsilon > 0$ ist, dann ist $T(n) \in \Theta(n^{\log_b a})$.

FALL 2: Wenn $f(n) \in \Theta(n^{\log_b a})$ ist, dann ist $T(n) \in \Theta(n^{\log_b a} \log n)$.

FALL 3: Wenn $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ für ein $\varepsilon > 0$ ist, und wenn es eine Konstante d gibt mit $0 < d < 1$, so dass für alle hinreichend großen n gilt $af(n/b) \leqslant df(n)$, dann ist $T(n) \in \Theta(f(n))$.

Dass die Aussagen in diesen drei Fällen richtig sind, bezeichnet man manchmal als *Mastertheorem*, obwohl es sich sicherlich um keine sehr tiefschürfenden Erkenntnisse handelt. *Mastertheorem*

Betrachten wir als Beispiele noch einmal die Matrixmultiplikation. Als „Problemgröße“ n benutzen wir die Zeilen- bzw. Spaltenzahl. Der Fall von $n \times n$-Matrizen wird auf den kleineren Fall von $n/2 \times n/2$-Matrizen zurückgeführt.

Bei der Schulmethode haben wir $a = 8$ Multiplikationen kleinerer Matrizen der Größe $n/2$ durchzuführen; es ist also $b = 2$. In diesem Fall ist $\log_b a = \log_2 8 = 3$. Der zusätzliche Aufwand

besteht in 4 kleinen Matrixadditionen, so dass $f(n) = 4 \cdot n^2/4 = n^2$. Damit ist $f(n) \in O(n^{3-\varepsilon})$ (z. B. für $\varepsilon = 1/2$) und der erste Fall des Mastertheorems besagt, dass folglich $T(n) \in \Theta(n^3)$. (Und das hatten wir uns vorhin tatsächlich auch klar gemacht.)

Bei Strassens geschickterer Methode sind nur $a = 7$ Multiplikationen kleinerer Matrizen der Größe $n/2$ durchzuführen (es ist also wieder $b = 2$). In diesem Fall ist $\log_b a = \log_2 7 \approx 2.807\ldots$. Der zusätzliche Aufwand besteht in 18 kleinen Matrixadditionen, so dass $f(n) = 18 \cdot n^2/4 \in \Theta(n^2)$. Auch hier gilt für ein geeignetes ε wieder $f(n) \in O(n^{\log_b a - \varepsilon}) = O(n^{\log_2 7 - \varepsilon})$. Folglich benötigt Strassens Algorithmus $T(n) \in \Theta(n^{\log_2 7}) = \Theta(n^{2.807\ldots})$ Zeit.

13.5 AUSBLICK

Algorithmen, bei denen die anderen beiden Fälle des Mastertheorems zum Tragen kommen, werden Sie im kommenden Semester in der Vorlesung „Algorithmen 1“ kennenlernen.

Manchmal wird „Teile und Herrsche“ auch in etwas komplizierterer Form angewendet (zum Beispiel mit deutlich unterschiedlich großen Teilproblemen). Für solche Situationen gibt Verallgemeinerungen obiger Aussagen (Satz von Akra und Bazzi).

LITERATUR

Coppersmith, Don und Shmuel Winograd (1990). “Matrix Multiplication via Arithmetic Progressions”. In: *Journal of Symbolic Computation* 9, S. 251–280.

Graham, Ronald L., Donald E. Knuth und Oren Patashnik (1989). *Concrete Mathematics*. Addison-Wesley.

Strassen, Volker (1969). “Gaussian Elimination Is Not Optimal”. In: *Numerische Mathematik* 14, S. 354–356.