

Frage: Können wir mit weniger als $\Theta(n^2)$ Bitoperationen auskommen?

Ansatz: „**Divide-and-Conquer**“ (lat.: *divide et impera* – deutsch: „teile und herrsche“)

1. Falls $|x|$ (Größe des Inputs) kleiner als eine Trivialitätsschranke s_0 , löse das Problem für x direkt.
2. Falls $|x| > s_0$, zerlege den Input x (geschickt) in mehrere Teile y_1, \dots, y_b („**Teile**“)
3. Löse die Aufgabe für jeden der Teile separat – **rekursiv**;
Ergebnisse: r_1, \dots, r_b
4. Setze aus r_1, \dots, r_b eine Lösung r für die Eingabe x zusammen.
(„**Kombiniere**“)

Konkret:

1. Falls $n = 1$: Multipliziere Bits x und y .
(Falls $n \leq n_0$: Benutze Hardware-Multiplikation, oder benutze Schulmethode)
2. Sonst: Setze $k = n/2$.
3. Zerlege Zahldarstellung x in 2 Teile A, B mit $x = A \cdot 2^k + B$
und zerlege Zahldarstellung y in 2 Teile C, D mit $y = C \cdot 2^k + D$
4. Berechne $A \cdot C, A \cdot D, B \cdot C, B \cdot D$ **rekursiv**
5. Setze zusammen:

$$(2) \quad x \cdot y = (A \cdot C) \cdot 2^{2k} + (A \cdot D + B \cdot C) \cdot 2^k + (B \cdot D).$$

Wieviele Bitoperationen („Kosten“, „cost“) insgesamt?

$$(3) \quad C(n) \leq \begin{cases} 1 & \text{falls } n = 1 \\ 4 \cdot C(n/2) + c \cdot n, & \end{cases}$$

für eine Konstante $c > 0$ (diese linearen Kosten decken die Bitoperationen für die Additionen ab).

Mit $c = 10$ (nur ein Beispiel) bekommen wir:

n	Schranke für $C(n)$
1	1
2	$4 \cdot 1 + 10 \cdot 2 = 24$
4	$4 \cdot 24 + 10 \cdot 4 = 136$
8	$4 \cdot 136 + 10 \cdot 8 = 624$
\vdots	
$n = 2^h$???

Die Ungleichung (3) ist eine sogenannte **Rekurrenzgleichung**, aus der wir folgern können: $C(n) = O(n^2)$. (Später: Das Master-Theorem ermöglicht es oft, einfach eine Lösung abzulesen.)

Dies ist nicht besser als die Schulmethode. Zu einfach gedacht!

Wir müssen auf jeden Fall $A \cdot C$ und $B \cdot D$ berechnen. Mit $E := A - B$ und $F := C - D$ beobachten wir

$$(4) \quad (A \cdot D + B \cdot C) = (A \cdot C + B \cdot D) - E \cdot F,$$

also

$$(5) \quad x \cdot y = (A \cdot C) \cdot 2^{2k} + ((A \cdot C + B \cdot D) - E \cdot F) \cdot 2^k + (B \cdot D).$$

Es genügt, **3** Produkte von Zahlen der halben Länge zu berechnen!

Bemerkung: Eigentlich wird die Instanz (x, y) nicht in drei (disjunkte) Teile „zerlegt“, sondern die Lösung für (x, y) wird auf die Lösung dreier halb so großer Instanzen „reduziert“ – ähnliches gilt für viele D-a-C-Algorithmen.

Bemerkung: E und F lassen sich mit $n/2$ Bits darstellen, können aber negativ sein. Das Vorzeichen müssen gesondert behandeln. Dazu wird die Signumsfunktion benutzt:

$$\text{sign}(x) = \begin{cases} -1 & \text{falls } x < 0 \\ 0 & \text{falls } x = 0 \\ 1 & \text{falls } x > 0. \end{cases}$$

Algorithmus 2 (Multiplikation nach Karatsuba und Ofman)

- (1) **Eingabe:** Natürliche Zahlen x und y , jede mit n Bits geschrieben, $n = 2^h$.
- (2) Falls $n = 1$, **return** $x \cdot y$; (* sonst: *)
- (3) Zerlege $x = A \cdot 2^k + B$, $y = C \cdot 2^k + D$, A, B, C, D mit $n/2$ Bits;
- (4) $E := A - B$; $F := C - D$;
- (5) $G := A \cdot C$; $H := B \cdot D$; $I := |E| \cdot |F|$; (* **rekursiv!** *)
- (6) **return** $G \cdot 2^{2k} + ((G + H) - \text{sign}(E)\text{sign}(F) \cdot I) \cdot 2^k + H$.

Wie viele Bitoperationen? Zunächst: Additionen ignorieren.

Wenn $n = 1$: $M(1) = 1$ \wedge -Operation.

Wenn $n = 2^h$, $h \geq 1$: $M(n) = 3M(n/2)$ \wedge -Operationen.

n	$M(n)$
1	1
2	$3 \cdot 1 = 3$
4	$3 \cdot 3 = 9$
8	$3 \cdot 3^2 = 3^3$
\vdots	
$n = 2^h$	3^h .

Also ist $M(n) = M(2^h) = 3^h = 2^{(\log_2 3)h} = n^{\log_2 3}$. Dabei: $\log_2 3 = \frac{\ln 3}{\ln 2} \approx 1.585$. Besser als quadratisch!

Nun wollen wir auch die Kosten der Additionen berücksichtigen. Die Zerlegung von x und y in zwei Teile ist (fast) umsonst, da hier nur Bitsegmente abzuschneiden sind. Eine Addition bzw. Subtraktion von Zahlen der Länge $O(n)$ kostet $O(n)$. Es fallen in einer Rekursionsstufe nur konstant viele solche Operationen an, also kosten diese alle zusammen $\leq cn$ Bitoperationen.

Rekurrenzungleichung:

$$(6) \quad C(n) \leq \begin{cases} 1 & \text{falls } n = 1 \\ 3 \cdot C(n/2) + c \cdot n, & \end{cases}$$

für eine Konstante $c > 0$.

Wir rechnen:

$$\begin{aligned}
C(n) &= \#(\text{Operationen bei Karatsuba-Ofman für } n \text{ Ziffern}) \\
&= 3 \cdot C(n/2) + cn \\
&= 3 \cdot C(2^{h-1}) + c \cdot 2^h \\
&= 3 \cdot (3 \cdot C(2^{h-2}) + c \cdot 2^{h-1}) + c \cdot 2^h \\
&= 3^2 \cdot C(2^{h-2}) + 3c \cdot 2^{h-1} + c \cdot 2^h \\
&= 3^3 \cdot C(2^{h-3}) + 3^2c \cdot 2^{h-2} + 3c \cdot 2^{h-1} + c \cdot 2^h \\
&\vdots \\
&= 3^h \cdot C(2^{h-h}) + 3^{h-1}c \cdot 2^1 + 3^{h-2}c \cdot 2^2 + \dots + 3^1c \cdot 2^{h-1} + c \cdot 3^0 \cdot 2^h \\
&= 3^h + 3^h \cdot c \cdot \left(\left(\frac{2}{3}\right)^1 + \left(\frac{2}{3}\right)^2 + \dots + \left(\frac{2}{3}\right)^h \right) \\
&< 3^h + 3^h \cdot c \cdot \frac{2}{3} \cdot \frac{1}{1 - \frac{2}{3}} \\
&< 3^h + 3^h \cdot c \cdot 2 \\
&= (1 + 2c) \cdot 3^h = O(n^{\log_2 3}).
\end{aligned}$$

Satz 3 Beim Karatsuba-Ofman-Multiplikationsalgorithmus beträgt die Anzahl der Bitoperationen und die Rechenzeit $O(n^{\log_2 3}) = O(n^{1.585})$.

Es sei bemerkt, dass es andere Multiplikationsalgorithmen mit (asymptotisch) noch geringerer Operationsanzahl gibt. Genannt sei der Algorithmus von Schönhage und Strassen (1971), der $O(n(\log n) \log \log n)$ Zeit/Bitoperationen benötigt, sowie der Algorithmus von Fürer (Martin Fürer: Faster integer multiplication, STOC 2007 Proceedings, S. 57-66), der mit $O(n(\log n) \cdot 2^{\log^* n})$ Bitoperationen¹ auskommt. Diese Algorithmen sind allerdings nur von theoretischem Wert; theoretisch interessant und sehr schwierig ist auch die Frage, ob $O(n \log n)$ Bitoperationen genügen.

Bei der praktischen Verwendung des Karatsuba-Ofman-Multiplikationsalgorithmus sollte man die sogenannte „gemischte Methode“ verwenden, die auf

¹ $\log^* n$ ist die kleinste Höhe eines Zweierpotenzturms $2^{2^{\dots^2}}$, dessen Wert n erreicht. Dies ist eine extrem kleine Zahl, viel kleiner als $\log n$.

kleine Instanzen (weniger als n_0 Bits), auch in der Rekursion, die Schulmethode anwendet, die ihrerseits wieder für Zahlen mit 32 oder mit 64 Bits die Hardware benutzt. Auf diese Weise lassen sich schon bei Zahlen mit wenig mehr als 1000 Bits Einsparungen gegenüber der Schulmethode erzielen. Was das richtige n_0 ist, kann eventuell von der Hardware abhängen.

1.2 Die Mergesort-Rekurrenz

Der Algorithmus Mergesort sortiert (bekanntlich) n Zahlen folgendermaßen:

1. Wenn $n = 1$, ist nichts zu tun; wenn $n = 2$, sortiere mit 1 Vergleich. Sonst:
2. Teile die Zahlenmenge in zwei Teile der Größe $\lceil n/2 \rceil$ und $\lfloor n/2 \rfloor$. Sortiere diese **rekursiv**.
3. Füge die beiden sortierten Folgen aus 2. zu einer sortierten Folge zusammen, mit einem „Reißverschlussverfahren“, das man „Merge“ oder „Mischen“ nennt. Kosten: Maximal $n - 1$ Vergleiche.

Vergleichszahlen im schlechtesten Fall:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
$C(n)$	0	1	3	5	8	11	14	17	21	25	29	33	37	41	...

Rekurrenzgleichung:

$$(7) \quad C(n) \leq \begin{cases} 0 & \text{falls } n = 1 \\ C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + n - 1 & \text{, sonst.} \end{cases}$$

Lösung: $C(n) \leq n \lceil \log n \rceil - (2^{\lceil \log n \rceil} - 1)$.

Dies kann man durch Induktion beweisen. (Bemerkung: Die Induktion geht über $k = \lceil \log n \rceil$, und man muss die Induktionsbehauptung für $n = 2^k - s$ sorgfältig formulieren.)

Im nächsten Abschnitt lernen wir eine Methode kennen, mit der man viele Rekurrenzgleichungen rasch und durch „Nachsehen“ lösen kann, ohne eigene Rechnungen oder Induktionsbeweise durchzuführen.

1.3 Das Master-Theorem für D-a-C-Rekurrenzen

Wir betrachten Rekurrenzen der folgenden Form:

$$(8) \quad B(n) \leq \begin{cases} g & \text{falls } n = 1 \\ a \cdot B(n/b) + f(n) & \text{, sonst.} \end{cases}$$

Dabei ist $a \geq 1$ eine ganze Zahl, $b > 1$ ist eine Konstante, $f(n)$ ist eine monoton wachsende Funktion. Falls n/b keine ganze Zahl ist, sollte man sich an dieser Stelle $B(\lceil n/b \rceil)$ denken.

Eine solche Rekurrenz ergibt sich bei der Analyse eines D-a-C-Algorithmus, der für einen trivialen Basisfall (Größe 1) höchstens Kosten g hat und aus einer Instanz der Größe $n > 1$ genau a Teilinstanzen der Größe n/b (passend gerundet) bildet, dann den Algorithmus rekursiv aufruft und die Lösungen wieder kombiniert.

O.B.d.A. können wir annehmen, dass $B(n)$ monoton wachsend ist. Andernfalls definieren wir einfach

$$\hat{B}(n) = \max\{B(i) \mid 1 \leq i \leq n\}.$$

Dann ist $\hat{B}(n)$ monoton und erfüllt ebenfalls die Rekurrenzungleichung (weil $f(n)$ als monoton wachsend angenommen wurde).

Um die Überlegungen zu vereinfachen und anschaulicher zu gestalten, nehmen wir an, dass $b > 1$ ganzzahlig ist. Die resultierenden Formeln gelten aber auch für nicht ganzzahlige b . Ebenso nehmen wir an, dass n eine Potenz von b ist. Dies lässt sich mit der Monotonie von $f(n)$ und von $B(n)$ rechtfertigen.

Sei also $n = b^\ell$. Um $B(n)$ abzuschätzen, bilden wir einen „Rekursionsbaum“, wie folgt:

Die Wurzel sitzt auf Level 0, sie hat den Eintrag $f(n)$ und hat a Kinder auf Level 1. Wenn v ein Knoten auf Level $i - 1 < \ell$ ist, hat v genau a Kinder auf Level i . Falls $i < \ell$, haben diese Kinder den Eintrag $f(n/b^i)$; Knoten auf Level ℓ haben den Eintrag g .

(Skizze: siehe Vorlesung)

Lemma 4 *Wenn v Knoten auf Level i ist, dann gilt: $B(n/b^i) \leq$ Summe der Einträge des Unterbaums, der v als Wurzel hat.*

Also: $B(n) \leq$ Summe der Einträge im gesamten Baum.

Dieses Lemma beweist man leicht durch Induktion über $i = \ell, \ell - 1, \dots, 0$ (bottom-up).

Wenn man bemerkt, dass es auf Level i genau a^i Knoten gibt, die alle denselben Eintrag $f(n/b^i)$ bzw. g haben, ergibt sich folgende Abschätzung:

$$(9) \quad B(n) \leq \sum_{0 \leq i < \ell} a^i \cdot f(n/b^i) + a^\ell \cdot g =: B_1(n) + B_2(n).$$

Wir formen zunächst den zweiten Term um:

$$(10) \quad B_2(n) = a^\ell \cdot g = (b^{\log_b a})^\ell \cdot g = (b^\ell)^{\log_b a} \cdot g = n^{\log_b a} \cdot g.$$

Dieser Term beschreibt den Beitrag zu den Gesamtkosten, der von den Blättern herrührt – algorithmisch gesehen sind das die Kosten für die Bearbeitung der a^ℓ Basisfälle.

Nun wenden wir uns dem ersten Term zu. Hier gibt es mehrere Fälle.

1.3.1 Erster Fall: Untere Bauebene

$f(m) = O(m^\alpha)$ mit $\alpha < \log_b a$.

In diesem Fall wächst f relativ langsam mit n . Die Beiträge aus den unteren Ebenen des Rekursionsbaums (kleine n) dominieren die Laufzeit – durch ihre Vielzahl, nicht die Größe der einzelnen Einträge.

$$\begin{aligned}
B_1(n) &= \sum_{0 \leq i < \ell} a^i \cdot f(n/b^i) \\
&= O\left(\sum_{0 \leq i < \ell} a^i \cdot \left(\frac{n}{b^i}\right)^\alpha\right) \\
&= O\left(n^\alpha \cdot \sum_{0 \leq i < \ell} \left(\frac{a}{b^\alpha}\right)^i\right) \\
&= O\left(n^\alpha \cdot \frac{(a/b^\alpha)^\ell}{(a/b^\alpha) - 1}\right) \\
&= O\left(n^\alpha \cdot a^\ell \cdot \frac{1}{(b^\ell)^\alpha}\right) \\
&= O(a^\ell).
\end{aligned}$$

In diesem Fall ist also

$$(11) \quad B(n) \leq B_1(n) + B_2(n) = O(a^\ell) + O(a^\ell) = O(a^\ell) = O(n^{\log_b a}).$$

Man sieht auch an der Rechnung, dass der Beitrag $B_2(n)$ (direkt von den Blättern) den Beitrag $B_1(n)$ von den inneren Knoten dominiert.

Die Karatsuba-Ofman-Rekurrenz ist von diesem Typ!

1.3.2 Zweiter Fall: Balancierte Baumebenen

$$f(m) = O(m^{\log_b a}).$$

In diesem Fall wächst f mit n (höchstens) mit einer Rate, die das Wachsen der Baumebenen gerade ausgleicht. Die Beiträge aus jeder Ebene des Rekursionsbaums zur Abschätzung sind in etwa gleich.

$$\begin{aligned}
B_1(n) &= \sum_{0 \leq i < \ell} a^i \cdot f(n/b^i) \\
&= O\left(\sum_{0 \leq i < \ell} a^i \cdot \left(\frac{n}{b^i}\right)^{\log_b a}\right) \\
&= O\left(\sum_{0 \leq i < \ell} a^i \cdot \frac{n^{\log_b a}}{a^i}\right) \\
&= O(\ell \cdot n^{\log_b a}).
\end{aligned}$$

In diesem Fall ist also

(12)

$$B(n) \leq B_1(n) + B_2(n) = O(\ell \cdot n^{\log_b a}) + O(n^{\log_b a}) = O(a^\ell) = O((\log n) \cdot n^{\log_b a}).$$

Der Beitrag zur Summe in jeder Ebene des Rekursionsbaums ist gleich, und ist gleich dem Beitrag von den Blättern. Der Mergesort-Algorithmus ist das typische Beispiel für einen Algorithmus, der eine Rekurrenz dieser Art liefert.

1.3.3 Dritter Fall: Übergewicht der Wurzel

$f(m) = \Omega(m^\alpha)$, mit $\alpha > \log_b a$.

^

(Regularitätsbedingung: f wächst stets mit der entsprechenden Rate)

Es gibt ein $c < 1$ mit: $c \cdot f(n) \geq a \cdot f(n/b)$.

Wenn man die Größe des Inputs von n/b auf n erhöht, wachsen die Kosten im Knoten von $f(n/b)$ auf $f(n)$ mindestens um den Faktor $a/c > a$.

Wir rechnen:

$$\begin{aligned}
f(n/b) &\leq \frac{c}{a} \cdot f(n) \\
f(n/b^2) &\leq \left(\frac{c}{a}\right)^2 \cdot f(n) \\
&\vdots \\
f(n/b^i) &\leq \left(\frac{c}{a}\right)^i \cdot f(n).
\end{aligned}$$

Dies benutzen wir, um $B_1(n)$ abzuschätzen:

$$\begin{aligned}
 B_1(n) &= \sum_{0 \leq i < \ell} a^i \cdot f(n/b^i) \\
 &= \sum_{0 \leq i < \ell} a^i \cdot \left(\frac{c}{a}\right)^i \cdot f(n) \\
 &= \sum_{0 \leq i < \ell} c^i \cdot f(n) \\
 &= f(n) \cdot \sum_{0 \leq i < \ell} c^i \\
 &= O(f(n)),
 \end{aligned}$$

weil $\sum_{0 \leq i < \ell} c^i = \frac{1-c^{\ell}}{1-c} = O(1)$.

In diesem Fall ist also

$$(13) \quad B(n) \leq B_1(n) + B_2(n) = O((\log n) \cdot n^{\log_b a}) + O(f(n)) = O(f(n)).$$

Die Summe wird vom Beitrag der Wurzel dominiert: die Kosten für den „divide“- und den „combine“-Schritt der Eingabe dominieren die Kosten sämtlicher rekursiver Aufrufe. (Algorithmen mit dieser Eigenschaft sind eher selten.)

Das Master-Theorem (Einfache Form) Es gelte

$$(14) \quad B(n) \leq \begin{cases} g & \text{falls } n = 1 \\ a \cdot B(n/b) + f(n) & \text{, sonst,} \end{cases}$$

wobei $b > 1$ und a ganzzahlige Konstante sind. Dann gilt für $n = b^\ell$:

1. Falls $f(m) = O(m^\alpha)$ mit $\alpha < \log_b a$, dann ist $B(n) = O(n^{\log_b a})$.
2. Falls $f(m) = O(m^{\log_b a})$, dann ist $B(n) = O(n^{\log_b a} \cdot \log n)$.
3. Falls $f(m) = \Omega(m^\alpha)$ mit $\alpha > \log_b a$ und $f(m) \geq \frac{a}{c} \cdot f(m)$, für $c < 1$ konstant, dann ist $B(n) = O(f(n))$.

Das Master-Theorem kann auf beliebige n erweitert werden. Man nimmt dann an, dass $f(n)$ und $B(n)$ monoton wachsend sind, und dass man in der Voraussetzung anstelle von $B(n/b)$ einen Ausdruck $B(\lceil n/b \rceil + d)$ stehen hat, für eine Konstante d . Der Beweis ist nur technisch etwas aufwendiger, die Grundgedanken bleiben aber dieselben wie in unserem einfachen Fall.

Weiterhin kann man analoge untere Schranken beweisen, wenn eine Rekurrenzgleichung „nach unten“ gegeben ist, d. h.

$$(15) \quad B(n) \geq \begin{cases} g & \text{falls } n = 1 \\ a \cdot B(n/b) + f(n) & , \text{sonst,} \end{cases}$$

Die Beweise laufen völlig analog. (Eine volle Version dieser Beweise findet man im Buch von Cormen, Leiserson, Rivest und Stein.)

1.4 Matrixmultiplikation nach Strassen

Wir wollen zwei $n \times n$ -Matrizen über einem Körper (z. B. \mathbb{Q}) oder einem Ring (z. B. \mathbb{Z}) multiplizieren:

$$A \cdot B = C.$$

Wenn $A = (a_{ij})_{1 \leq i, j \leq n}$ und $B = (b_{jk})_{1 \leq j, k \leq n}$, so ist bekanntermaßen $C = (c_{ik})_{1 \leq i, k \leq n}$ mit

$$(16) \quad c_{ik} = \sum_{1 \leq j \leq n} a_{ij} \cdot b_{jk}, \text{ für } 1 \leq i, k \leq n.$$

Man sieht sofort, dass bei direkter Implementierung dieser Methode n^3 Multiplikationen und $n^2(n-1)$ Additionen anfallen. Es handelt sich um eine $O(n^3)$ -Methode, was nicht ganz so schlimm ist wie es aussieht, da ja die Eingabe Umfang n^2 hat. Dennoch kann man sich fragen, ob man Matrizen mit weniger als kubisch vielen Operationen multiplizieren kann. Die etwas überraschende Antwort: dies ist möglich, wenn die Operation Subtraktion einsetzt, die bei der Definition des Matrixprodukts gar nicht vorkommt.

Der Einfachheit halber nehmen wir an, dass $n = 2^h$ für eine natürliche Zahl h ist. Wenn die Matrizen A und B weniger als n_0 Zeilen/Spalten haben, multiplizieren wir mit der naiven Formel (16).

Wenn die Zeilen/Spaltenzahl größer oder gleich n_0 ist, zerlegen wir die beiden Matrizen in 4 „Quadranten“, also in 4 Teilmatrizen mit Umfang $(n/2)$ Zeilen und Spalten:

$$(17) \quad \begin{pmatrix} a & b \\ c & d \end{pmatrix} \text{ und } \begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

Wenn wir nun die Produktmatrix naiv aus 4 Teilen zusammensetzen:

$$(18) \quad \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} aA + bC & aB + bD \\ cA + dC & cB + dD \end{pmatrix},$$

so sind in der Rekursion 8 Teilprobleme der Größe $n/2$ zu lösen; wieder lässt sich relativ leicht feststellen, dass dies zu einem Gesamtaufwand von $O(n^3)$ führt. Damit wird also kein Gewinn gegenüber der naiven Methode erzielt.

Der Trick (von Strassen, 1969) ist der, eine Folge von Hilfsmatrizen zu definieren, aus denen man das Produkt zusammensetzen kann, wobei man mit 7 (statt 8) Multiplikationen kleinerer Matrizen auskommt. Hinzu kommen cn^2 Additionen/Subtraktionen. Strassen hatte einfache Formeln vorgeschlagen, die 18 Additionen von $(n/2) \times (n/2)$ -Matrizen benötigt, siehe Folien.

Die folgende Formel geht auf Shmuel Winograd zurück. Sie verringert die Zahl der Additionen auf 15. Es wird nicht verlangt, dass man selber darauf kommt oder sie auswendig weiß ... Setze $w = aA - (a - c - d)(A - C + D)$; dann sieht die Matrix insgesamt so aus:

$$(19) \quad \begin{pmatrix} aA + bC & w + (c + d)(C - A) + (a + b - c - d)D \\ w + (a - c)(D - C) - d(A - B - C + D) & w + (a - c)(D - C) + (c + d)(C - A) \end{pmatrix}$$

Es ist kein großes Problem, durch Ausmultiplizieren zu verifizieren, dass dies tatsächlich die gewünschte Produktmatrix liefert.

Die folgenden Definitionen erlauben es, die Einträge in (19) mit 7 Multipli-

kationen und 15 Additionen zu berechnen.

$$\begin{aligned}
f &:= aA \\
g &:= bC \\
h &:= f + g &&= aA + bC \\
i &:= a - c \\
j &:= i - d &&= a - c - d \\
k &:= A - C \\
l &:= k + D &&= A - C + D \\
m &:= j \cdot l \\
w &:= f - m &&= aA - (a - c - d)(A - C + D) \\
n &:= c + d \\
o &:= k \cdot n &&= -(c + d)(C - A) \\
p &:= j + b &&= a + b - c - d \\
q &:= p \cdot D \\
r &:= w - o \\
s &:= r + q \\
t &:= D - C \\
u &:= i \cdot t &&= (a - c)(D - C) \\
v &:= l - B &&= A - B - C + D \\
x &:= w + u \\
y &:= x - o \\
z &:= x - d \cdot v
\end{aligned}$$

Für diese Berechnung werden 7 Multiplikationen und 15 Additionen von $(n/2) \times (n/2)$ -Matrizen benötigt (letzteres entspricht $\frac{15}{4}n$ Elementaradditionen). Die Produktmatrix ist dann:

$$(20) \quad \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} aA + bC & aB + bD \\ cA + dC & cB + dD \end{pmatrix} = \begin{pmatrix} h & s \\ z & y \end{pmatrix},$$

wie man durch sorgfältiges Nachrechnen verifiziert.

Für die Anzahl der Elementarmultiplikationen erhalten wir die Rekurrenzgleichung

$$(21) \quad M(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 7 \cdot M(n/2) & \text{falls } n = 2^h > 1 \end{cases}$$

Dies löst sich wie bei der Ganzzahl-Multiplikation leicht zu

$$M(n) = 7^h = 2^{\log_2 7 h} = n^{\log_2 7}.$$

Dabei ist $2,80 < \log_2 7 < 2,81$. Für die Anzahl der Elementaroperationen (Multiplikationen und Additionen) ergibt sich

$$(22) \quad T(n) = \begin{cases} 0 & \text{falls } n = 1 \\ 7 \cdot T(n/2) + O(n^2) & \text{, sonst.} \end{cases}$$

Wir wenden das Master-Theorem an. Hier ist $a = 7$, $b = 2$, und $f(n) = O(n^2)$ mit $2 < \log_2 7$. Daraus folgt (1. Fall im Master-Theorem): $T(n) = O(n^{\log_2 7}) = O(n^{2,81})$.

Bemerkung: Der asymptotisch schnellste bekannte Algorithmus zur Multiplikation von Matrizen ist der von Coppersmith und Winograd², der eine Laufzeitschranke von $O(n^{2,376})$ aufweist. Eine Laufzeit von n^2 kann nicht unterschritten werden, da jeder Algorithmus jede der $2n^2$ Inputkomponenten ansehen muss. Manche Forscher vermuten, dass eine Laufzeit von $O(n^2 \log n)$ erreicht werden kann, hiervon ist man aber noch weit entfernt.

²Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. Journal of Symbolic Computation, 9:251–280, 1990.