

# Schnelle Multiplikation

$O(n^2)$  ist für Anfänger

Johannes Hahn

06.06.25 – 09.06.25

## Inhaltsverzeichnis

<b>0</b>	<b>Wiederholung: Gruppen &amp; Ringe</b>	<b>2</b>
0.1	Gruppen . . . . .	2
0.2	Ringe . . . . .	4
0.2.1	Matrizen . . . . .	5
<b>1</b>	<b>Level 0: <math>O(n^2)</math> wie in der Grundschule</b>	<b>8</b>
1.1	Das Problem . . . . .	8
Blatt 1	. . . . .	13
<b>2</b>	<b>Level 1: <math>O(n^{1.585})</math> Karatsuba</b>	<b>14</b>
Blatt 2	. . . . .	17
<b>3</b>	<b>Level 2: <math>O(n^{1+\epsilon})</math> Toom-Cook</b>	<b>18</b>
Blatt 3	. . . . .	23

## 0 Wiederholung: Gruppen & Ringe

### 0.1 Gruppen

#### 0.1 Definition (Gruppen):

Eine Gruppe  $(G, *)$  besteht aus

- einer Menge  $G$  und
- einer Abbildung  $*$  :  $G \times G, (g, h) \mapsto g \cdot h$ , genannt Multiplikation,

die die „Gruppen-Axiome“ erfüllen:

(G1) Assoziativität:  $\forall x, y, z \in G : x * (y * z) = (x * y) * z$ .

(G2) Neutrales Element:  $\exists e \in G \forall x \in G : x * e = e * x = x$ .

(G3) Inverse Elemente:  $\forall x \in G \exists x' \in G : x * x' = e = x' * x$ .

Die Gruppe heißt abelsch<sup>1</sup> oder kommutativ, falls zusätzlich dazu die folgende Bedingung erfüllt ist:

(G4) Kommutativität:  $\forall x, y \in G : x * y = y * x$ .

Der Kürze halber schreibt man oft nur  $G$  statt  $(G, *)$ , falls klar ist, welche Operation  $*$  gemeint ist.

**0.2:** Es gibt zwei Klassen von Gruppen. Die einen sind Gruppen, bei denen man traditionell eine multiplikative Schreibweise wählt, d.h. der Name der Gruppenverknüpfung  $*$  erinnert an ein Multiplikationssymbol, etwa  $*$ ,  $\cdot$ ,  $\otimes$ ,  $\circ$  etc. Oder sogar, wie bei der gewöhnlichen Multiplikation: Man lässt das Symbol ganz weg und schreibt nur noch  $xy$ . Praktisch alle nichtabelschen Gruppen schreibt man multiplikativ. In dieser Schreibweise nennt man das (es gibt wirklich nur eines, siehe Lemma 0.5.a) neutrale Element dann 1 statt  $e$  und das (ebenfalls eindeutige, siehe Lemma 0.5.b) von  $x$  nennt man  $x^{-1}$  statt  $x'$ . Die zweite Klasse sind Gruppen, die man traditionell additiv schreibt, d.h. der Name der Gruppenverknüpfung erinnert an ein Additionssymbol, etwa  $+$  oder  $\oplus$ . Dies wird fast ausschließlich bei abelschen Gruppen angewandt. In dieser Schreibweise nennt man das neutrale Element dann 0 statt  $e$  und das inverse Element von  $x$  nennt man  $-x$  statt  $x'$ . Es gibt aber natürlich keinen inhaltlichen Unterschied zwischen diesen Schreibweisen. Die Wahl, wie wir etwas aufschreiben, ist ja nur eine Frage der Ästhetik, sie hat keine inhaltlichen Konsequenzen.

**0.3 Beispiel:** a.)  $G = \mathbb{Z}$  mit  $x * y = x + y$ . Es ist  $e = 0$ ,  $x' = -x$ . Diese Gruppe ist abelsch. Analog sind auch  $(\mathbb{Q}, +)$ ,  $(\mathbb{R}, +)$  und  $(\mathbb{C}, +)$  abelsche Gruppen.

b.)  $G = \mathbb{R}_{>0} = \{x \in \mathbb{R} \mid x > 0\}$  mit  $x * y = xy$  (Multiplikation). Es ist  $e = 1$ ,  $x' = \frac{1}{x}$ .

---

<sup>1</sup>Niels Hendrik Abel, norwegischer Mathematiker, 1802–1829

- c.) Die triviale Gruppe  $G = \{1\}$ , mit  $1 * 1 = 1$ . Neutrales Element 1, und  $1' = 1$ .
- d.) Die Menge  $G = \{+1, -1\}$ , mit Multiplikation. Es ist  $e = 1$ ,  $x' = \frac{1}{x} = x$ . Dies ist eine endliche Gruppe:  $|G| = 2$ .
- e.) Die Matrizen Gruppen  $GL_n(K)$ ,  $SL_n(K)$ ,  $O_n(\mathbb{R})$ ,  $SO_n(\mathbb{R})$ ,  $U_n(\mathbb{C})$ ,  $SU_n(\mathbb{C})$ , ... sind wichtig für Geometrie und Physik:

$O_n(\mathbb{R})$  beschreibt Drehungen und Spiegelungen im  $n$ -dimensionalen Raum;  $SO_n(\mathbb{R})$  beschreibt nur die Drehungen.  $U_1(\mathbb{C})$  z.B. beschreibt die Phasenverschiebung zwischen zwei rotierenden Systemen;  $SU_2(\mathbb{C})$  ist z.B. für die Beschreibung der elektromagnetischen Kraft zuständig und ist u.A. für das quantenphysikalische Phänomen des Spins verantwortlich;  $SU_3(\mathbb{C})$  spielt in der Beschreibung der starken Kernkraft eine Rolle.

Andere Gruppen wie z.B.  $O_{3,1}(\mathbb{R})$  kommen in der Relativitätstheorie vor (Sie erkennen drei Raum- und eine Zeitdimension). Diese Gruppe beschreibt sogenannte „Lorentz-Boosts“.

Diese Gruppen sind fast alle nichtabelsch und unendlich.

**0.4 Beispiel:** a.)  $(\mathbb{N}, +)$  ist hingegen keine Gruppe. (G1) und (G2) sind zwar erfüllt, aber (G3) nicht, denn nicht *jedes* Element hat ein inverses Element. 0 hat eines, 1 aber nicht, denn es gibt keine natürliche Zahl, die  $n + 1 = 1 + n = 0$  erfüllt. (Es gibt eine ganze Zahl, die das erfüllt, aber keine natürliche)

- b.) Aus ähnlichen Gründen ist auch  $(\mathbb{R}, \cdot)$  keine Gruppe. Das Element  $0 \in \mathbb{R}$  ist nicht invertierbar, denn keine reelle Zahl erfüllt  $x \cdot 0 = 1$ .

**0.5 Lemma** (Eindeutigkeit des neutralen Elements und der inversen Elemente):

Sei  $(G, *)$  eine Gruppe.

- a.) Es gibt in  $G$  genau ein neutrales Element.
- b.) Zu jedem  $x \in G$  gibt es genau ein Inverses  $x' \in G$ . Dieses nennt man daher  $x^{-1}$ .

*Beweis.* Übung. □

**0.6 Lemma und Definition** (Potenzschreibweise):

Zusätzlich zur Schreibweise  $g^{-1}$  für inverse Elemente führen wir allgemeiner Potenzen mit ganzzahligen Exponenten für alle Gruppenelemente ein.

Ist  $(G, \cdot)$  eine (multiplikativ notierte) Gruppe und  $g \in G$ , so definieren wir  $g^k$ , indem wir

$$g^0 := 1 \quad g^{k+1} := g^k \cdot g \quad \text{und} \quad g^{k-1} := g^k \cdot g^{-1}$$

für alle  $k \in \mathbb{Z}$  festlegen. Mit dieser Bezeichnung gilt dann:

$$\text{a.) } \forall g \in G \forall n, m \in \mathbb{Z} : g^{n+m} = g^n \cdot g^m$$

$$\text{b.) } \forall g \in G \forall n, m \in \mathbb{Z} : g^{nm} = (g^n)^m$$

Falls  $(G, \cdot)$  eine abelsche Gruppe ist, dann gilt außerdem

$$\text{c.) } \forall g_1, g_2 \in G \forall n \in \mathbb{Z} : (g_1 \cdot g_2)^n = g_1^n \cdot g_2^n$$

*Beweis.* Übung. □

**0.7:** In additiv geschriebenen Gruppen benutzt man normalerweise nicht die Potenzschreibweise  $g^n$ , sondern die Schreibweise  $ng$ . Dann schreiben sich die drei Potenzgesetze aber ebenso wiedererkennbar einfach als

$$(n+m)g = ng + mg, \quad (nm)g = n(mg), \quad n(g_1 + g_2) = ng_1 + ng_2$$

## 0.2 Ringe

### 0.8 Definition:

Ein Ring  $R = (R, +, \cdot)$  besteht aus einer Menge  $R$  zusammen mit Abbildungen  $+$  :  $R \times R \rightarrow R$ ,  $(x, y) \mapsto x + y$ , genannt „Addition“, und  $\cdot$  :  $R \times R \rightarrow R$ ,  $(x, y) \mapsto x \cdot y = xy$ , genannt „Multiplikation“, welche die folgenden Bedingungen erfüllen:

(R1)  $(R, +)$  ist eine abelsche Gruppe. Das neutrale Element bezeichnen wir mit  $0_R$  und nennen es Nullelement des Rings.

(R1) Assoziativität der Multiplikation:

$$\forall x, y, z \in R : (xy)z = x(yz)$$

(R3) Neutrales Element der Multiplikation:

$$\exists 1_R \in R \forall x \in R : x \cdot 1_R = x = 1_R \cdot x$$

Dieses Element nennen wir Einselement des Rings.

(R4) Distributivgesetze:

$$\forall x, y, z \in R : x(y + z) = xy + xz \wedge (x + y)z = xz + yz$$

Der Ring heißt kommutativ, falls zusätzlich

(R5) Kommutativität:  $\forall x, y \in R : xy = yx$ .

gilt.

Der Kürze halber schreibt man oft nur  $R$  statt  $(R, +, \cdot)$ , wenn aus dem Kontext klar ist, welche Addition und Multiplikation gemeint sind.

**0.9 Beispiel:** a.)  $\mathbb{Z}$ ,  $\mathbb{Q}$  und  $\mathbb{R}$  sind Ringe bezüglich der üblichen Addition und Multiplikation.

b.) Der Nullring  $R = \{0\}$  mit  $0 + 0 = 0 \cdot 0 = 0$  ist ein Ring, mit  $1_R = 0_R = 0$ . In der Tat ist das der einzige Ring, in dem  $0_R = 1_R$  gilt (Übung).

c.) Ist  $R$  ein Ring und  $n \in \mathbb{N}$ , so ist die Menge  $R^{n \times n}$  aller  $n \times n$ -Matrizen über  $R$  ein Ring bezüglich Matrixaddition und -multiplikation (s. unten).

d.)  $\mathbb{Z}/n$  der Ringe der „Restklassen modulo  $n$ “ besteht aus den Äquivalenzklassen

$$[a] := \{ b \in \mathbb{Z} \mid a \equiv b \pmod{n} \}$$

zusammen mit der Addition  $[a] + [b] := [a + b]$  und der Multiplikation  $[a] \cdot [b] := [ab]$ .

e.) Die Menge  $R[X]$  aller Polynome mit Koeffizienten aus  $R$  ist ein Ring zusammen mit der üblichen Addition:

$$\begin{aligned} \sum_{i=0}^n a_i X^i + \sum_{i=0}^n b_i X^i &:= \sum_{i=0}^n (a_i + b_i) X^i \\ \left( \sum_{i=0}^n a_i X^i \right) \left( \sum_{j=0}^m b_j X^j \right) &:= \sum_{k=0}^{n+m} \left( \sum_{\substack{i,j \\ i+j=k}} a_i b_j \right) X^k \end{aligned}$$

Ist  $R$  kommutativ, dann ist auch  $R[X]$  ein kommutativer Ring.

### 0.10 Lemma:

Sei  $R$  ein Ring. Dann:

- a.) Es gibt nur ein Einselement in  $R$ .
- b.)  $\forall x \in R : 0 \cdot x = x \cdot 0 = 0$ .
- c.)  $\forall x, y \in R : x \cdot (-y) = (-x) \cdot y = -(xy)$ .
- d.)  $(-1)^2 = 1$ .

*Beweis.* Übung. □

### 0.2.1 Matrizen

#### 0.11 Definition:

Es sei  $R$  ein Ring und  $n, m \in \mathbb{N}$ . Eine  $n \times m$ -Matrix mit Einträgen aus  $R$  ist eine rechteckige Anordnung von  $nm$  Elementen von  $R$  in  $n$  Zeilen und  $m$  Spalten. Im Deutschen Sprachgebrauch ist es üblich, runde Klammern um eine Matrix zu schreiben.

Die Menge aller  $n \times m$ -Matrizen mit Einträgen aus  $R$  bezeichnen wir mit  $R^{n \times m}$ .

Ist  $A$  eine solche Matrix, so bezeichnet man mit  $A_{ij}$  den Eintrag an der Stelle  $(i, j)$ , d.h. in der  $i$ ten Zeile und der  $j$ ten Spalte.

Eine Matrix heißt quadratisch, wenn die Anzahl der Zeilen und der Spalten gleich sind.

**0.12 Beispiel:**

Hier ist eine  $(3 \times 2)$ -Matrix mit Einträgen aus  $\mathbb{R}$ :  $\begin{pmatrix} 1 & 2 \\ 4 & -7 \\ 0,5 & \pi \end{pmatrix}$ .

**0.13 Beispiel:**

Eine  $1 \times 0$ -Matrix:  $()$ .

**0.14 Beispiel:**

Für  $A = \begin{pmatrix} 1 & 3 & 7 & 4 \\ 3 & 1 & 2 & 9 \\ 8 & 0 & 7 & 3 \end{pmatrix}$  ist  $A_{23} = 2$ .

**0.15 Definition** (Matrixaddition und -multiplikation):

Für Matrizen  $A, B \in R^{m \times n}$  wird die Summe  $A + B \in R^{m \times n}$  definiert durch

$$(A + B)_{ij} := A_{ij} + B_{ij}$$

für alle  $1 \leq i \leq m, 1 \leq j \leq n$ .

Sind  $A \in R^{m \times n}$  und  $B \in R^{n \times p}$  Matrizen, so wird das Produkt  $AB \in R^{m \times p}$  definiert durch

$$(AB)_{ik} := A_{i1}B_{1k} + A_{i2}B_{2k} + A_{i3}B_{3k} + \cdots + A_{in}B_{nk} = \sum_{j=1}^n A_{ij}B_{jk}$$

für alle  $1 \leq i \leq m, 1 \leq j \leq n$ .

Ist  $A \in R^{m \times n}$  eine Matrix und  $\lambda \in R$  ein Skalar, so ist  $\lambda A \in R^{m \times n}$  definiert durch

$$(\lambda A)_{ij} := \lambda A_{ij}$$

Die Nullmatrix  $0 = 0_{n \times m} \in R^{n \times m}$  ist definiert durch

$$(0_{n \times m})_{ij} := 0$$

Die Einheitsmatrix  $1_{n \times n}$  ist die  $n \times n$ -Matrix, deren Eintrag an der Stelle  $(i, j)$  genau

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{sonst} \end{cases}$$

ist.

**0.16 Beispiel:**

$$\begin{pmatrix} 1 & 2 & 4 \\ 2 & 5 & 7 \end{pmatrix} + \begin{pmatrix} 0 & 3 & 1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 5 & 5 \\ 3 & 6 & 8 \end{pmatrix}.$$

**0.17 Beispiel:**

$$\begin{pmatrix} 1 & 0 \\ 3 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{pmatrix} = \begin{pmatrix} 1 \cdot 0 + 0 \cdot 1 & 1 \cdot 2 + 0 \cdot 3 & 1 \cdot 4 + 0 \cdot 5 \\ 3 \cdot 0 + 1 \cdot 1 & 3 \cdot 2 + 1 \cdot 3 & 3 \cdot 4 + 1 \cdot 5 \end{pmatrix} = \begin{pmatrix} 0 & 2 & 4 \\ 1 & 9 & 17 \end{pmatrix}$$

**0.18:** Beachten Sie: das Produkt  $AB$  ist nur dann definiert, wenn  $A$  die gleiche Anzahl von Spalten hat, wie  $B$  Zeilen hat.

**0.19 Lemma** (Ringeigenschaften):

Matrizenringe  $R^{n \times n}$  sind wirklich Ringe. Allgemeiner gilt selbst für rechteckige Matrizen:

a.)  $(R^{n \times m}, +)$  ist eine abelsche Gruppe.

b.) Matrixmultiplikation ist assoziativ:

$$\forall A \in R^{n \times m}, B \in R^{m \times p}, C \in R^{p \times q} : (AB)C = A(BC)$$

c.) Matrizenmultiplikation ist distributiv:

$$\forall A \in R^{n \times m}, B, C \in R^{m \times p} : A(B + C) = AB + AC$$

$$\forall A; B \in R^{n \times m}, C \in R^{m \times p} : (A + B)C = AC + BC$$

d.) Einheitsmatrizen sind Einselemente:

$$\forall A \in R^{n \times m} : 1_{n \times n} \cdot A = A = A \cdot 1_{m \times m}$$

e.) Multiplikation mit Skalaren ist assoziativ:

$$\forall A \in R^{n \times m}, B \in R^{m \times p} \forall \lambda \in R : (\lambda A)B = \lambda(AB)$$

sowie, wenn  $R$  kommutativ ist

$$\forall A \in R^{n \times m}, B \in R^{m \times p} \forall \lambda \in R : \lambda(AB) = A(\lambda B)$$

*Beweis.* Übung. □

**0.20 Beispiel:**

Schon  $R^{2 \times 2}$  und alle größeren Matrizenringe sind nicht kommutativ, selbst wenn  $R$  noch kommutativ war, denn

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \quad \text{aber} \quad \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}.$$

# 1 Level 0: $O(n^2)$ wie in der Grundschule

## 1.1 Das Problem

**1.1:** Wir wollen effizient multiplizieren. Was „effizient“ heißt, hängt von der Problemstellung ab. Konkret ist unser Problem folgendes:

Gegeben zwei natürliche Zahlen  $A, B$ , dargestellt durch ihre  $n$  Ziffern in Basis  $g$

$$A = \sum_{i=0}^{n-1} a_i g^i, \quad B = \sum_{i=0}^{n-1} b_i g^i$$

mit  $0 \leq a_i, b_i < g$ , berechne die Ziffern  $c_0, c_1, \dots, c_{2n-1}$  des Produkts  $C := A \cdot B$ .

Wir messen die Laufzeit von Multiplikationsalgorithmen in Abhängigkeit von  $n$ . Wir nehmen an, dass Operationen auf einzelnen Ziffern eine konstante Laufzeit haben.

**1.2:** Die Annahme, dass Operationen mit einzelnen Ziffern eine konstante Laufzeit haben, ist in der Theorie dadurch gerechtfertigt, dass man eine Additions- und Multiplikationstabelle vorberechnen kann. Dann hat jede Addition oder Multiplikation von zwei Ziffern die Laufzeit von „Schau einen Eintrag in eine  $n \times n$  Tabelle nach“.

So funktionierte das letztendlich auch in der Grundschule. Das Einmaleins haben wir auswendig gelernt und schriftliches Addieren und Multiplizieren von größeren Zahlen nutzt den „Lookup“ im Gedächtnis.

**1.3:** In Computerhardware sind i.A. keine Lookup-Tabellen implementiert. Die offensichtliche Wahl der Basis wäre  $g = 2$ . Die Addition und Multiplikation von einzelnen Bits sind einfach<sup>2</sup> direkt in Hardware zu implementieren, sodass kein Lookup nötig ist.

Etwas weniger offensichtlich ist es, auf „Wörtern“ statt Bits zu arbeiten, d.h.  $g = 2^8$ ,  $2^{16}$  (richtig alte Hardware),  $2^{32}$  (alte Hardware und einige moderne Mobilgeräte) oder  $2^{64}$  (moderne Hardware, insb. PCs) zu wählen. Manche Hardware unterstützt sogar noch höhere Wortgrößen.

Während Speicher für eine  $256 \times 256$  große Tabelle mit je 2 B großen Einträgen sicher noch verkraftbar wäre, ist eine  $2^{32} \times 2^{32}$  Tabelle schon zu groß und eine  $2^{64} \times 2^{64}$  Lookup-Tabelle völlig unrealistisch:  $2^{64}$  viele 64 bit große Einträge sind  $1.476 \times 10^8$  TB und  $2^{128}$  viele 128 bit große Einträge sind  $\approx 5.445 \times 10^{27}$  TB. Kein Datacenter der Welt hat diese Kapazität! Das ist ein Vielfaches der Gesamtgröße des Internets.

Also wird auch in diesen Fällen das Ergebnis einer einzelnen Multiplikation mit einer speziellen Instruktion direkt von der Hardware berechnet. Die Schaltkreise sind allerdings schon deutlich komplexer. Und in der Tat sind die Schaltkreise für größere Wörter i.d.R. zusammengesetzt aus Schaltkreisen, die Addition und Multiplikation kleinerer Wörter implementieren, d.h. es wird ein passender Multiplikationsalgorithmus in Hardware implementiert.

---

<sup>2</sup>Wie „einfach“ es auch sein mag, wenige hundert Atome große Schaltkreise zu bauen...



Solange alle Multiplikation von zwei Wörtern stets durch den gleichen Schaltkreis ermittelt wird, kann aber ebenfalls davon ausgegangen werden, dass die Laufzeitkosten konstant sind.

**1.4:** Es ist jedoch nicht unbedingt klar, dass die beiden Schaltkreise für Addition und Multiplikation gleich schnell operieren. In der Tat war auf echter Hardware lange ein deutlicher Unterschied zwischen den Laufzeiten einer Additions- und einer Multiplikationsinstruktion vorhanden. Teilweise war Multiplikation eine Größenordnung langsamer als Addition. Aus diesem Grund hat es sich auch in der theoretischen Analyse eingebürgert, nicht alle Operationen eines Multiplikationsalgorithmus zu zählen, sondern nur die Multiplikationen, weil diese mehr ins Gewicht fallen.

Wir werden dieser Tradition auch folgen, aber trotzdem Buchführen, wie viele Additionen ein Algorithmus jeweils benötigt, wenn auch im Zweifelsfall etwas weniger genau. Insbesondere werden wir keine Additionen zählen, die Schleifenvariablen inkrementieren. Die sind Teil des Konstrukts „Schleife“, nicht unseres spezifischen Algorithmus. (Und wie die Schleifen tatsächlich ausgeführt werden, ist außerdem zu schwer vorhersagbar, um konkrete Anzahlen an Additionen vorhersagen zu können)

**1.5:** Der offensichtliche Algorithmus für Multiplikation beliebiger natürlicher Zahlen ergibt sich direkt durch Anwendung des Distributivgesetzes:

$$\left(\sum_{i=0}^{n-1} a_i g^i\right) \cdot \left(\sum_{j=0}^{n-1} b_j g^j\right) = \sum_{i,j} (a_i b_j) g^{i+j} = \sum_{k=0}^{2n-2} \left(\sum_{i+j=k} a_i b_j\right) g^k$$

Indem wir alle  $n^2$  Produkte  $a_i b_j$  berechnen und geeignet aufsummieren, erhalten wir das Produkt  $A \cdot B$ .

In der Grundschule lernt man meist ein tabellarisches Verfahren, das schematisch in etwa so abläuft, dass in jeder Zeile das Produkt einer einzelnen Ziffer  $a_i$  von  $A$  mit  $B$  ausgerechnet wird, um  $i$  viele Dezimalstellen verschoben wird (was die Multiplikation mit  $10^i$  darstellt), sowie am Ende alle Zeilen aufsummiert werden.

In jeder Zeile werden dafür die Produkte  $a_i \cdot b_j$  ausgerechnet. Grob vereinfacht würde das also so aussehen:

$a_2$	$a_1$	$a_0$	$\cdot$	$b_3$	$b_2$	$b_1$	$b_0$
				$a_0 b_3$	$a_0 b_2$	$a_0 b_1$	$a_0 b_0$
			$a_1 b_3$	$a_1 b_2$	$a_1 b_1$	$a_1 b_0$	0
		$a_2 b_3$	$a_2 b_2$	$a_2 b_1$	$a_2 b_0$	0	0
	$c_6$	$c_5$	$c_4$	$c_3$	$c_2$	$c_1$	$c_0$

Natürlich sind die Produkte  $a_i \cdot b_j$  der einzelnen Ziffern i.A. größer als eine Ziffer, sodass man in jeder Zeile und von einer Zeile zur nächsten jeweils die Überträge beachten muss.

**A1 Algorithmus** (Schriftliches Multiplizieren):

Wir benutzen die Schulmethode direkt. Einziger Unterschied ist, dass wir ein laufendes Zwischenergebnis pflegen und dies kontinuierlich updaten, anstatt  $n$  viele Zwischenergebnisse zu speichern und erst am Ende alle aufzuaddieren.

```

Array<Digit> multiply(Array<Digit> a, Array<Digit> b){
    // initialize with a.length + b.length zeros
    Array<Digit> c := {0,0,...,0};

    for(i := 0; i<a.length; i++){
        Digit z := 0;

        for(j := 0; j<b.length; j++){
            // p := c[i+j] + a_i * b_j + z is at most a 2-digit number !!
            Digits {p_1,p_0} := c[i+j] + a[i] * b[j] + z;

            c[i+j]      := p_0;
            z           := p_1;
        }
        c[i + b.length] := z;
    }

    return c;
}

```

**1.6 Proposition** (Korrektheit):

Der Algorithmus **A1** ist korrekt. Er benötigt  $n^2$  Multiplikation,  $O(n^2)$  Additionen und benötigt  $n + O(1)$  Speicher.

*Beweis.* Das meiste ist relativ klar. Wir beginnen, indem wir  $C$  mit den Ziffern  $000 \dots 000$  initialisieren. Dann addieren wir für jedes  $i$  die Zahl  $a_i g^i \cdot B$  zu  $C$ , indem wir  $a_i \cdot B$  berechnen und auf die Ziffern  $\dots c_{i+2}c_{i+1}c_i$  addieren, und speichern das Ergebnis wieder in  $C$ .

Wir addieren in jeder Iteration der inneren Schleife die 1er Ziffer des Produkts  $p := a_i \cdot b_j$  an die korrekte Stelle  $c_{i+j}$ . Wir addieren außerdem den Übertrag der letzten Iteration an diese Stelle und merken uns den Übertrag, den wir dabei erhalten, für die nächste Iteration.

Damit das funktioniert, müssen wir die mit !! markierte Behauptung beweisen:

Das Produkt zweier Ziffern  $a_i, b_j \leq g - 1$  ist höchstens  $(g - 1)^2 = g^2 - 2g + 1$  und die Addition von zwei weiteren Ziffern  $c, z \leq g - 1$  führt zu einem Maximum von  $g^2 - 1$ , was echt kleiner  $g^2$  ist, also wirklich maximal zwei Ziffern hat. Der Übertrag  $z$  ist also auch in der nächsten Iteration maximal eine Ziffer groß.  $\square$

**1.7:** Wie wir eben gezeigt haben, ist es möglich, einen Schaltkreis zu bauen, der vier Wörter  $a, b, c, z$  als Input nimmt, und in einer einzigen Operation das 2-Wort breite Ergebnis  $ab + c + z$  berechnet.

Es ist von der konkreten Prozessorarchitektur abhängig, ob es tatsächlich eine solche Instruktion gibt. Wenn es sie gibt, sind überhaupt keine Additionsinstruktionen zusätzlich zu den Multiplikationen für unseren Algorithmus nötig.

Wenn es diesen Schaltkreis nicht gibt, dann gibt es oft trotzdem  $(a, b, c) \mapsto ab + c$ , womit wir nur noch eine einzige Addition zusätzlich zur Multiplikation benötigen.

**1.8:** Der Algorithmus ist offenbar nicht davon abhängig, dass  $A$  und  $B$  die gleiche Anzahl an Ziffern haben. Etwas präziser könnten wir die Laufzeiten also als  $O(nm)$  ausdrücken, wobei  $n$  und  $m$  die Längen der beiden Zahlen sind.

Darin enthalten ist die Teilaussage, dass die Multiplikation mit einer kleinen Konstanten in  $O(n)$  realisierbar ist. Das merken wir uns für später, weil wir es verwenden werden.

**1.9:** Man beachte auch, dass die asymptotische Laufzeit nicht von der Wahl von  $g$  abhängig ist. Wenn wir eine andere Basis  $g'$  wählen, dann sind  $A$  und  $B$  Zahlen mit  $\approx \frac{\log(g)}{\log(g')}n$  Ziffern. Die Basis zu ändern, ändert also die Laufzeit nur um einen konstanten Faktor.

**1.10:** Es ist nicht einmal notwendig, die beiden Inputs  $A$  und  $B$  bzgl. derselben Basis auszudrücken. In der Tat ist das genau das Prinzip hinter der „ägyptischen Multiplikation“, auch „(russische) Bauernmultiplikation“ genannt:

Drückt man eine der beiden Zahlen in Basis 2 aus, etwa  $A$ , so reduziert sich das Problem darauf, aus  $B$  die Zahlen  $2B, 4B, 8B, \dots$  zu berechnen, da die Multiplikationen  $a_i \cdot B$  entweder 0 oder  $B$  ist (denn  $a_i$  ist ja 0 oder 1 in Basis 2), also keinen zusätzlichen Rechenaufwand verursachen:

$$(a_{n-1}2^{n-1} + \dots + a_22^2 + a_12 + a_0) \cdot B = \sum_{\substack{0 \leq i \leq n-1 \\ a_i=1}} 2^i B$$

Man kann also mit  $B$  beginnen und die Zahl immer wieder zu sich selbst addieren, um  $2B, 4B, \dots$  zu erhalten. Und während dessen, jedes Mal, wenn das passende Bit in  $A$  gesetzt ist, eine weitere Addition auf das Zwischenergebnis durchführen.

## A2 Algorithmus (Double-and-add):

Input:  $A$  in Binärdarstellung  $a_{n-1}2^{n-1} + \dots + a_22^2 + a_12 + a_0$ ,  $B$  in beliebiger Basis  $b_{m-1}g^{m-1} + \dots + b_2g^2 + b_1g + b_0$ .

Output: Das Produkt  $C = A \cdot B$  in Basis  $g$ .

Pseudocode:

```
Array<Digit> multiply(Array<Bit> a, Array<Digit> b){
    // calculate the number of base-g-digits in A
    n := floor( a.length * log(2)/log(g) ) + 1;
    c := new Array<Digit>[n + b.length];

    // stores 2^i*b
    b2 := b;
    for(i := 0; i < a.length; i++){
```

```

        if(a[i] == 1){
            c := add(c, b2);
        }
        b2 := add(b2, b2);
    }

    return c;
}

```

### 1.11 Proposition (Korrektheit und Laufzeit):

Der Algorithmus ist korrekt, hat Laufzeit  $O(n^2)$  (alles Additionen) und Speicherbedarf  $n + O(1)$ .

*Beweis.* Übung. □

**1.12:** Da man in der Praxis eben nicht mit beliebigen  $g$ , sondern sehr häufig mit Zweierpotenzen  $g = 2^k$  arbeitet, ist für die Berechnung von  $\frac{\log(2)}{\log(g)}$  keine Fließkomma-Arithmetik nötig, da sich alles zu  $\frac{1}{k}$  kürzt. Die Länge von  $A$  in bits ist typischerweise klein genug, damit die Division `a.length / k` in einer einzigen Hardware-Instruktion berechnet werden kann.

Mehr noch: Wenn  $g$  selbst eine Zweierpotenz ist, dann besteht eigentlich kein Unterschied zwischen `Array<Bit>` und `Array<Digit>` und es ist überhaupt keine zusätzliche Aktion nötig.

## Aufgaben

### Aufgabe 1.1. – Parallelisierung

Beschleunige Algorithmus [A1](#) durch Parallelisieren einiger Schritte um einen konstanten Faktor zu beschleunigen, wobei der Speicherverbrauch weiterhin  $O(n)$  bleibt. Wie?

### Aufgabe 1.2. – Square-and-multiply

Es sei ein Monoid  $(M, \cdot)$  gegeben und eine Black-box, die die Multiplikation berechnet.

- a.) Finde einen Algorithmus, der für gegebenes  $x \in M$  und  $N \in \mathbb{N}$  die Potenz  $x^N$  in  $O(\log(N))$  Multiplikationen berechnet.
- b.) Was hat das mit dem Problem des Multiplizierens zu tun?

### Aufgabe 1.3.

Beweise Proposition [1.11](#).

## 2 Level 1: $O(n^{1.585})$ Karatsuba

**2.1:** Über Jahrtausende ist keine Methode bekannt gewesen, wesentlich schneller als  $O(n^2)$  zu multiplizieren. Erst 1960 wurde von Anatoly Karatsuba<sup>3</sup> eine schnellere Methode entdeckt.

Karatsubas Methode arbeitet rekursiv wie alle anderen, die wir uns ansehen werden: Anstatt in der eigentlich beabsichtigten kleinen Zahlenbasis  $g$  (z.B.  $g = 2$  oder  $g = 10$ ) zu arbeiten, werden die Zahlen zunächst bzgl. einer größeren Basis  $G$  dargestellt. Um nun zwei Zahlen in Basis  $G$  zu multiplizieren, sind mehrere Multiplikationen von Zahlen  $< G$  zu erwarten. Diese werden rekursiv mit dem gleichen Algorithmus, aber bzgl. einer Basis  $g \leq G' < G$  berechnet. Oft werden die anderen Basen  $G, G', \dots$  als Potenzen von  $g$  gewählt, sodass keine zusätzliche Rechenarbeit notwendig ist, um zwischen  $G$ -Ziffern und  $g$ -Ziffern umzurechnen. Beispiel: Die Ziffern zur Basis  $G = 1000$  der Zahl  $A = 123456789$  sind  $a_2 = 123, a_1 = 456, a_0 = 789$ .

Speziell für Karatsubas Methode wählt man  $G$  groß genug, dass die beiden Inputs  $A$  und  $B$  nur noch aus zwei Ziffern bestehen. Für Zahlen mit zwei Ziffern  $A = a_1G + a_0, B = b_1G + b_0$  gilt:

$$AB = \underbrace{a_1b_2}_{=c_2} G^2 + \underbrace{(a_1b_0 + a_0b_1)}_{=c_1} G + \underbrace{a_0b_0}_{=c_0}$$

In dieser Form aufgeschrieben, sehen wir genau die  $2^2$  Multiplikationen, die der klassische Algorithmus benötigt. Karatsuba sah, dass der mittlere Term auch gleich

$$c_1 = (a_1 + a_0)(b_1 + b_0) - a_1b_1 - a_0b_0 = (a_1 + b_1)(b_1 + b_0) - c_2 - c_0$$

ist. Somit kann man  $c_1$  anstatt mit zwei Multiplikationen und einer Addition auch durch eine Multiplikation und zwei Subtraktionen berechnen. Additionen und Subtraktionen haben typischerweise annähernd die gleiche Laufzeit.

### A3 Algorithmus (Karatsuba's algorithm):

```

Array<Digit> multiply(Array<Digit> a, Array<Digit> b) {
    int n := max(a.length, b.length);
    if(n <= CUTOFF){
        return slow_multiply(a,b); // algorithm A1
    }

    int k := ceil(n/2); // round up if n is odd

    low    := multiply(a[0..k-1], b[0..k-1]);
    high   := multiply(a[k..n-1], b[k..n-1]);

    mixed_a := add(a[0..k-1], a[k..n-1]);
    mixed_b := add(b[0..k-1], b[k..n-1]);

```

---

<sup>3</sup>russ. Mathematiker, 1937–2008

```

    product := multiply(mixed_a, mixed_b);
    middle := subtract(subtract(product, high), low);

    return add(low, (0,middle), (0,0,high));
}

```

**2.2:** In der Praxis wählt man den Cutoff nicht so klein wie möglich, um möglichst viel mit der „besseren“ Karatsuba-Methode zu arbeiten, sondern ein kleines bisschen größer, da für sehr kleine Anzahlen von Ziffern die Schul-Methode in der Praxis doch schneller ist als die Karatsuba-Methode. Selbst wenn nur einer der beiden Faktoren wenig Ziffern hat, insbesondere wenn ein Faktor nur eine Ziffer hat, ist die Schul-Methode schneller.

Die Standard-Library von Java nutzt beispielsweise  $g = 2^{32}$  als Basis, d.h. 32-bit Integer werden direkt per Hardware-Instruktion multipliziert, und  $n = 80$  als Cutoff<sup>4</sup>. Sobald einer der Faktoren weniger als 80 Worte groß ist, wird mit der Schul-Methode multipliziert.

Die C-Bibliothek GMP (GNU Multiprecision Arithmetic library) definiert den Cutoff hardware-abhängig zwischen  $n = 10$  und 34 Wörtern, z.B.  $n = 26$  für ARM64 Apple's M1 Prozessor.

**2.3:** Man beachte, dass  $(a_1 + a_0)(b_1 + b_0)$  nicht ein Produkt von zwei Zahlen  $< G$ , sondern  $< 2G$  ist. Wir sind also ganz leicht über die Grenze gegangen, bis zu der wir gehen wollten. Man kann auch nicht einfach  $G$  etwas größer zu wählen, sodass die führenden Ziffern  $a_1$  und  $b_1$  kleiner werden, z.B. wird das mit  $A = 199999$  nicht funktionieren: Egal, welche (sinnvolle) Potenz  $G = 10^k$  wir wählen, die Ziffer  $a_0$  wird immer 9.99 sein und somit bei Addition zum Overflow führen.

Da man i.d.R. nicht mit  $g = 2$  arbeitet (sondern eher  $g = 2^{32}$ ), benötigt man für die Summen  $a_1 + a_0$  und  $b_1 + b_0$  selbst im Extremfall nur eine weitere Ziffer. Man verliert also in der Praxis nicht wirklich viel. Insbesondere dann, wenn man die Karatsuba-Methode sowieso nur für die größeren Produkte einsetzt; ob man von 1000 zu 500 oder 501 Ziffern reduziert, ist nicht so furchtbar relevant.

Wenn man es unbedingt möchte, ist es aber möglich, stattdessen

$$c_1 = -(a_1 - a_0)(b_1 - b_0) + a_1b_1 + a_0b_0$$

zu verwenden, denn da  $0 \leq a_0, a_1 \leq G - 1$  gilt, wird auch stets  $|a_1 - a_0| \leq G$  sein. Man zahlt dafür den Preis, ein bisschen mehr Buchhaltung mit den Vorzeichen betreiben zu müssen.

## 2.4 Proposition:

Algorithmus **A3** ist korrekt, benötigt  $O(n^{\log(3)/\log(2)})$  Multiplikationen, Additionen und Subtraktionen, sowie  $O(n)$  Speicher.

<sup>4</sup>Diese Zahlen sind Implementierungsdetails, die sich von einer Java-Version zur nächsten auch mal ändern können. Ich habe sie im OpenJDK 21 geprüft. Wer selbst nachschauen will: Es geht um die Klasse `java.math.BigInteger` und ihre Methode `multiply(BigInteger)`.

*Beweis.* Die Abschätzungen ergeben sich direkt aus dem Mastertheorem (siehe Aufgabe 2.2), denn die Anzahl der Multiplikationen erfüllt die Rekursion  $M(n) = 3M(\frac{n}{2})$ , die Anzahl der Additionen/Subtraktionen  $A(n) = 3A(\frac{n}{2}) + 2\frac{n}{2} + 2n$ .

Der Speicherbedarf ergibt sich daraus, dass wir für jede Stufe der Rekursion Speicher für `high`, `low`, `mixed_a`, `mixed_b`, `product`, `middle` sowie das Endergebnis benötigen, die jeweils  $n$ ,  $n$ ,  $\frac{n}{2} + 1$ ,  $\frac{n}{2} + 1$ ,  $n + 1$ ,  $n$ ,  $2n$  groß sind. Wir runden das zu  $7n + O(1)$ .<sup>5</sup>

Wenn wir sequentiell ausführen, also jeweils nur einen Zweig im Rekursionsbaum gleichzeitig im Speicher halten müssen, haben also demnach einen Speicherbedarf von

$$(7n + O(1)) + \left(7\frac{n}{2} + O(1)\right) + \left(7\frac{n}{4} + O(1)\right) + \dots = 14n + O(\log(n)) \quad \square$$

**2.5:** Solange wir nur endlich viele Prozessoren haben und nicht maximal parallel arbeiten, benötigen wir auch nur ein konstantes Vielfaches der sequentiellen Speichermenge. Wenn wir maximal parallelisieren und somit alle Zweige der Rekursion gleichzeitig im Speicher halten müssen, dann erfüllt der Speicherbedarf auch eine Rekursion wie  $S(n) = 3S(\frac{n}{2}) + 5n$ , womit sich dann ein Speicherbedarf von  $O(n^{\log(3)/\log(2)})$  ergibt.

In der Praxis hat keine Maschine unendlich viele Prozessoren, sodass man niemals voll parallelisieren kann. Andererseits stoßen wir in der Praxis auch nicht auf Zahlen mit beliebig vielen Ziffern, sondern ausschließlich  $\leq 2^{32}$  Ziffern (wenn die Zifferngröße  $g = 2^{32}$  ist, dann wäre eine einzige solche Zahl schon  $\approx 4$  GB groß!), sodass die Rekursionstiefe  $\log(n)$  als praktisch konstant betrachtet werden kann und in der Tat auch in der Nähe einer real vorhandenen Anzahl an Prozessoren liegt.

---

<sup>5</sup>In der Tat kann man clever sein: Für `high`, `low` und `middle` benötigt man keinen zusätzlichen Speicher, wenn man `high`, `low` direkt im Ergebnis speichert und die Subtraktionen bzw. Addition jeweils in-place ausführt. Dann ist der konstante Faktor 5 statt 7.



## Aufgaben

### Aufgabe 2.1.

Finde eine explizite Abbildungsvorschrift für eine Funktion  $T : \mathbb{R}_{\geq 1} \rightarrow \mathbb{R}$  an, die die Rekursion

$$T(n) = aT\left(\frac{n}{b}\right) + cn, T(1) = t_1$$

mit  $a, b \geq 2, c \geq 0$  erfüllt.

Hinweis: Betrachte  $n = b^k$ .

### Aufgabe 2.2. – Mastertheorem

Beweise das *Mastertheorem für divide-and-conquer-Algorithmen*:

Gegeben sei eine Rekursion der Form

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

für Zahlen  $a, b > 1$  und eine nichtnegative Funktion  $f$ . Definiere  $e_{crit} := \frac{\log(a)}{\log(b)}$ . Dann gilt:

- a.) Wenn  $f \in O(n^c)$  mit  $c < e_{crit}$ , dann ist  $T \in \Theta(n^{e_{crit}})$ .
- b.) Wenn  $f \in O(n^{e_{crit}})$ , dann ist  $T \in \Theta(n^{e_{crit}} \log(n))$ .

### 3 Level 2: $O(n^{1+\varepsilon})$ Toom-Cook

**3.1:** Der nächste große Durchbruch kam schnell nach Karatsuba: 1963 bereits beschrieb Toom<sup>6</sup> eine Verallgemeinerung von Karatsubas Algorithmus; 1966 beschrieb Cook<sup>7</sup> eine vereinfachte Version, die heutzutage Toom-Cook-Algorithmus genannt wird.

„Der“ Toom-Cook-Algorithmus ist tatsächlich eine Familie von Algorithmen, parametrisiert durch eine natürliche Zahl  $k \geq 2$ . In der Praxis verwendet wird jedoch nur die  $k = 3$  Version, welche dementsprechend auch „der“ Toom-Cook-Algorithmus genannt wird.

Die grundlegende Idee ist es, die Input-Zahlen nicht in 2, sondern  $k$  Teile zu zerlegen und die  $G$ -Ziffern der Inputs

$$A = a_{k-1}G^{k-1} + \dots + a_1G + a_0, \quad B = b_{k-1}G^{k-1} + \dots + b_1G + b_0$$

als Koeffizienten von zwei Polynomen

$$\alpha(X) = a_{k-1}X^{k-1} + \dots + a_1X + a_0, \quad \beta(X) = b_{k-1}X^{k-1} + \dots + b_1X + b_0$$

aufzufassen. Die Zahlen  $A$  und  $B$  sind Auswertungen dieser Produkte an der Stelle  $X = G$ ; das Produkt  $C := A \cdot B$  ist entsprechend die Auswertung des Polynoms  $\gamma(X) := \alpha(X)\beta(X)$  an der Stelle  $X = G$ . Wenn wir die Koeffizienten von  $\gamma$  aus den Koeffizienten von  $\alpha$  und  $\beta$  berechnen können, dann ist die Ermittlung von  $C$  sehr einfach.

Das Produkt  $\alpha(X)\beta(X)$  mittels der Definition

$$\alpha(X)\beta(X) = \sum_{i,j} a_i b_j X^{i+j}$$

auszurechnen, würde uns direkt wieder zur Schulmultiplikation führen. Die entscheidende Einsicht ist aber, dass ein Polynom vom Grad  $d$  bereits eindeutig durch seine Werte an  $d + 1$  beliebigen Stützstellen festgelegt ist. Wir benötigen  $2k - 1$  Stützstellen für  $\gamma$ . Wenn wir an Stelle der relativ großen Zahl  $G$  sehr kleine Stützstellen, z.B.

$$s \in \{-k - 1, \dots, -1, 0, 1, \dots, k - 1\}$$

verwenden, dann sind die Ergebnisse  $\alpha(s)$  und  $\beta(s)$  eher in der Größenordnung  $\text{const} \cdot G$ , nicht  $G^k$ .

Wir können also die Produkte  $\alpha(s)\beta(s)$  mit einer kleineren Multiplikation bestimmen und so einen rekursiven Multiplikationsalgorithmus bauen, wenn wir es schaffen, aus den Stützstellen  $\gamma(s)$  relativ schnell wieder die Koeffizienten von  $\gamma$  zu bestimmen.

#### 3.2 Lemma (Vandermonde-Matrix):

<sup>6</sup>Andrei Leonovich Toom, 1942–2022, russischer Mathematiker

<sup>7</sup>Stephen Arthur Cook, geb. 1939, amerikanisch-kanadischer Mathematiker

Evaluation an  $q$  Stützstellen  $s_0, s_1, \dots, s_q \in \mathbb{R}$  ist eine lineare Abbildung  $\mathbb{R}[X]_{\leq d} \rightarrow \mathbb{R}^{q+1}$

$$\begin{pmatrix} \alpha(s_0) \\ \alpha(s_1) \\ \vdots \\ \alpha(s_q) \end{pmatrix} = \underbrace{\begin{pmatrix} s_0^0 & s_0^1 & \cdots & s_0^d \\ s_1^0 & s_1^1 & \cdots & s_1^d \\ \vdots & \vdots & \ddots & \vdots \\ s_q^0 & s_q^1 & \cdots & s_q^d \end{pmatrix}}_{=: V \in \mathbb{R}^{(q+1) \times (d+1)}} \cdot \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_d \end{pmatrix}$$

Für  $q = d$  ist die Matrix  $V$  invertierbar.

*Beweis.* Übung. □

#### A4 Algorithmus (Toom-Cook- $k$ ):

Parameter:  $k \in \mathbb{N}_{\geq 2}$ , paarweise verschiedene Stützstellen  $\{s_\ell \mid 0 \leq \ell < 2k-1\} \subset \mathbb{Q}$ .

Vorbereitung:

- Die Evaluationsmatrix  $\mathcal{E} \in \mathbb{Q}^{(2k-1) \times k}$  mit  $\mathcal{E}_{\ell,m} = s_\ell^m$  für  $0 \leq \ell < 2k-1, 0 \leq m < k$ .
- Die Interpolationsmatrix  $\mathcal{I} \in \mathbb{Q}^{(2k-1) \times (2k-1)}$  mit  $(\mathcal{I}^{-1})_{\ell,m} = s_\ell^m$  mit  $0 \leq \ell, m < 2k$ .

```

Array<Digit> multiply(Array<Digit> a, Array<Digit> b){
    n := max(a.length, b.length);
    if(n < CUTOFF){
        return slow_multiplication(a,b); // algorithms A1 and/or A3
    }
    n_prime := ceil(n/k); // n' is the size of the individual chunks

    for(int m:=0; m<k; m++) {
        // not an actual computation, only a naming convention
        alpha[m] := a[m*n_prime .. (m+1)n_prime-1];
        beta[m] := b[m*n_prime .. (m+1)n_prime-1];
    }

    // Step 1: evaluate alpha and beta at the chosen points
    eval_alpha := new Array<>[2k-1]; // each of these entries is itself
    eval_beta := new Array<>[2k-1]; // an array of n_prime+0(1) digits

    for(int l:=0; l<2k-1; l++) {
        for(int m:=0; m<k; m++) {
            // multiplications with small, known constants
            // => no recursion necessary here
            eval_alpha[l] := add(eval_alpha[l],
                                multiply(E[l][m], alpha[m]));
        }
    }

    // Step 2: pointwise multiplication

```

```

// each of the entries is itself an array of 2*n_prime+O(1) digits
eval_gamma := new Array<>[2k-1];

for(int l:=0; l<2k-1; l++) {
    // recursion here
    eval_gamma[l] := multiply(eval_alpha[l], eval_beta[l]);
}

// Step 3: recover coefficients of gamma
gamma := new Array<>[2k-1];

for(int l:=0; l<2k-1; l++) {
    for(int m:=0; m<2k-1; m++) {
        // multiplication with constant rationals, i.e.
        // multiplication and division with small, known constants
        // => no recursion necessary here
        gamma[l] := add(gamma[l],
                        multiply(I[l][m], eval_gamma[m]));
    }
}

// Step 4: evaluate gamma(g^k)
result := new Array<Digit>[2n];
for(int l:=0; l<2k-1; l++) {
    result[l*n_prime...] := add(result[l*n_prime...], gamma[l]);
}

return result;
}

```

### 3.3 Proposition:

Der obige Algorithmus ist korrekt, hat Laufzeit  $O(n^e)$  mit  $e := \frac{\log(2k-1)}{\log(k)}$  und Speicherbedarf  $O(n)$ .

*Beweis.* Analog zum Beweis für den Karatsuba-Algorithmus folgt dies im Wesentlichen aus dem Mastertheorem, da wir die Rekursion  $T(n) = (2k-1)T(\frac{n}{k}) + O(n)$  für die Laufzeit haben. Dabei benutzen wir, dass Multiplizieren mit kleinen Konstanten Laufzeit  $O(n)$  hat.

Wir benötigen Speicher für `eval_alpha`, `eval_beta`, `eval_gamma`, `gamma` und `result`. Das sind viermal  $2n + O(1)$  und einmal  $2n$  Speicher, die wir benötigen. Das  $O(1)$  kommt jeweils davon, dass wir bei den diversen Additionen Platz für einen Übertrag bereitstellen müssen.

Insgesamt kommen wir also auf  $10n + O(1)$  pro Rekursionsstufe. Insgesamt benötigen wir also

$$(10n + O(1)) \left( 1 + \frac{1}{k} + \frac{1}{k^2} + \dots \right) = 10n \frac{k}{k-1} + O(\log(n))$$

Speicher. □

**3.4:** Wir benötigen  $O(k^2)$  viele Multiplikationen/Divisionen mit kleinen Konstanten. Die  $O$ -Konstante in  $O(n)$  für die Multiplikationen/Divisionen hängt aber empfindlich davon ab, welche Konstanten das genau sind. Auch wenn es für die asymptotische Laufzeit nicht relevant ist, muss man sich also doch etwas Mühe geben bei der Wahl der Stützstellen.

Besonders einfach ist es, Polynome in  $X = 0$  auszuwerten, weil dafür gar keine Rechnung notwendig ist, also wählt man 0 immer als eine der Stützstellen. Ebenfalls kostenlos ist die „Auswertung bei  $X = \infty$ “. Damit meint man für ein Polynom vom Grad  $d$  die Berechnung von  $\lim_{x \rightarrow \infty} \frac{p(x)}{x^d}$ , was genau den führenden Koeffizienten des Polynoms ergibt. In unserer Notation also  $a_{k-1}$ ,  $b_{k-1}$  bzw.  $\gamma_{2k-2}$ . Man kann sich leicht überlegen, dass  $\gamma(\infty) = \alpha(\infty)\beta(\infty)$  ebenfalls erfüllt ist, sodass unsere Rechnung trotzdem funktionieren.

**3.5:** Für  $k = 2$  erhält man Karatsubas Algorithmus als Spezialfall, indem man die Stützstellen  $0, 1, \infty$  wählt. Das ergibt nämlich genau die Produkte

$$\begin{aligned}\gamma(0) &= \alpha(0)\beta(0) = a_0b_0 \\ \gamma(1) &= \alpha(1)\beta(1) = (a_11^1 + a_01^0)(b_11^1 + b_01^0) \\ \gamma(\infty) &= \alpha(\infty)\beta(\infty) = a_1b_1\end{aligned}$$

die wir für Karatsubas Algorithmus benutzt haben.

**3.6:** Für  $k = 3$  erhalten wir einen Algorithmus in  $O(n^{\log(5)/\log(3)}) \subseteq O(n^{1.465})$ .

**3.7 (Große  $k$ ):** Für  $k \rightarrow \infty$  ist  $\frac{\log(2k-1)}{\log(k)} \searrow 1$ . Durch hinreichend groß gewähltes  $k$  können wir also für jedes  $\varepsilon > 0$  einen Multiplikationsalgorithmus in  $O(n^{1+\varepsilon})$  hinschreiben.

In der Praxis verwendet man aber keine großen Werte für  $k$ , da der Overhead extrem schnell anwächst, während der Break-even-point, ab dem die theoretische asymptotische Laufzeit die tatsächliche Laufzeit bestimmt, viel zu schnell wächst: Wenn wir  $k$  so groß genug wählen wollen, damit  $\frac{\log(2k-1)}{\log(k)} \approx \frac{\log(2k)}{\log(k)} \leq 1 + \frac{1}{e}$  erfüllt ist, dann müssen wir in der Tat  $k \geq 2^e$  wählen.

Wenn wir das aber tun, dann ist der Overhead für einen einzigen Rekursionsschritt in der Größenordnung  $Ck^2n = C2^{2e}n$ . Für extrem große  $n$  wird das zwar dominiert von  $C'n^{1+1/e}$ , aber tatsächlich geschieht das erst bei  $n \geq \left(\frac{C}{C'}\right)^e 2^{2e^2}$ .

Selbst wenn wir nur  $O(n^{1.1})$  erreichen wollen (also  $e = 10$ ), dann müssen wir  $k = 2^{10}$  und Zahlen mit vielen hundert Bit betrachten, bevor wir überhaupt eine Chance haben, dass der Toom-Cook- $k$ -Algorithmus sich tatsächlich auszahlt. Mehr noch: Wenn  $k = 2^{10}$  ist, dann sind die Potenzen  $s_\ell^m$ , die wir für die Vandermonde-Matrix ausrechnen müssen, schon im Bereich von bis zu  $10 \cdot (2k - 1) \approx 20000$  bits groß. Das stellt die Behauptung in Frage, wie „klein“ die kleinen Konstanten eigentlich sind, die da in den drei Matrix-Vektor-Multiplikationen vorkommen. Wir müssen also in Wahrheit eher zehntausende Bits, vielleicht sogar hunderttausende Bits große Zahlen betrachten, bevor sich dieser Algorithmus praktisch auszahlt.

**3.8:** Nichts desto trotz ist diese Überlegung für die Theorie interessant. Wenn man für jedes  $k$  einen geeigneten Cutoff vorberechnet, dann kann man alle Toom-Cook-Algorithmen zu einem gemeinsamen Algorithmus kombinieren, der für hinreichend große Zahlen Toom-Cook- $k$  mit immer größer werdendem  $k$  verwendet. Dieser Algorithmus ist dann offenbar asymptotisch schneller als jeder einzelne Toom-Cook-Algorithmus, also in  $o(n^{1+\varepsilon})$  für jedes  $\varepsilon > 0$ .

Es ist nicht genau bekannt, welche Komplexitätsklasse man mit diesem Ansatz erreichen kann, aber eine Implementierung von Knuth<sup>8</sup> erzielt eine Komplexität von  $O(n2^{\sqrt{2\log(n)}}\log(n))$ , aber es ist nicht bekannt, ob das die beste Implementierung ist.

**3.9:** Deshalb werden nur sehr kleine  $k$  praktisch eingesetzt und selbst dann nur für sehr große  $n$ .

Die<sup>9</sup> Java Implementierung wählt z.B.  $n = 240$  als Cutoff, d.h. erst wenn beide Faktoren mehr als 240 Worte groß sind, wird die Toom-Cook-3-Multiplikation gegenüber der Karatsuba-Methode bevorzugt. Ein höheres  $k$  wird gar nicht verwendet in der Implementierung.

GMP verwendet  $k = 3, 4, 6, 8$  in gewissen Grenzen und den (asymptotisch noch schnelleren) Schönhage-Strassen-Algorithmus darüber.

---

<sup>8</sup>Donald Ervin Knuth, geb. 1938, amerikanischer Mathematiker und Informatiker, Autor der „The Art of Computer Programming“ Buchreihe.

<sup>9</sup>Wieder OpenJDK 21 in diesem Fall; andere Implementierungen der Standardbibliothek könnten theoretisch davon abweichen

## Aufgaben

### Aufgabe 3.1. – Vandermonde-Determinante

Beweise, dass die Vandermonde<sup>10</sup>-Matrix

$$V := \begin{pmatrix} s_0^0 & s_0^1 & \cdots & s_0^d \\ s_1^0 & s_1^1 & \cdots & s_1^d \\ \vdots & \vdots & \ddots & \vdots \\ s_d^0 & s_d^1 & \cdots & s_d^d \end{pmatrix}$$

die Determinante

$$\det(V) = \prod_{0 \leq i < j \leq d+1} (s_j - s_i)$$

hat. Insbesondere ist  $V$  genau dann invertierbar, wenn die  $s_i$  paarweise verschieden sind.

### Aufgabe 3.2.

Verallgemeinere den Toom-Cook- $k$ -Algorithmus zum Toom-Cook- $k_1, k_2$ -Algorithmus, der  $A$  in  $k_1$  und  $B$  in  $k_2$  kleinere Stücke zerlegt.

**3.10:** Der Toom-Cook-2,3-Algorithmus wird manchmal auch als Toom-Cook-2.5 bezeichnet.

---

<sup>10</sup> Alexandre-Théophile Vandermonde, 1735–1796, französischer Mathematiker, Musiker und Chemiker