SS 2021

Algorithmen und Datenstrukturen

9. Kapitel

Divide-and-Conquer-Algorithmen

Martin Dietzfelbinger

Juli 2021

Kapitel 9 Divide-and-Conquer (D-a-C)

Das **Algorithmenparadigma** hinter **MergeSort**

Vorgehen eines D-a-C-Algorithmus $\mathcal{A}$ für Berechnungsproblem $\mathcal{P} = (\mathcal{I}, \mathcal{O}, f)$
auf Instanz $x \in \mathcal{I}$:

0) Trivialitätstest: Wenn $\mathcal{P}$ für x einfach direkt zu lösen ist, tue dies. – Sonst:

1) Teile: Bilde aus x ($a \geq 1$ viele) Instanzen $x_1, \ldots, x_a \in \mathcal{I}$.

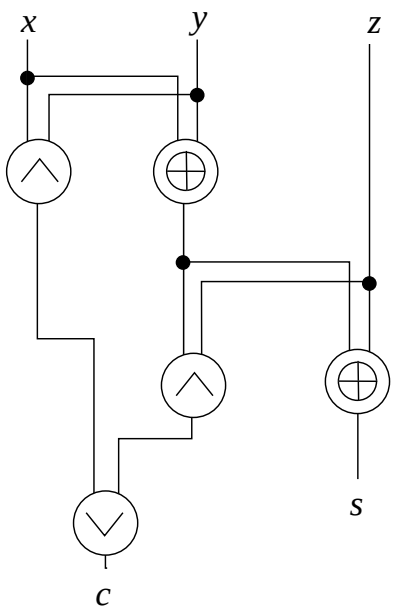
2) Rekursion: Löse $\mathcal{P}$ für $x_1, \ldots, x_a$, separat, durch
rekursive Verwendung von $\mathcal{A}$. Teillösungen: $r_1, \ldots, r_a$.

3) Kombiniere: Baue aus x und $r_1, \ldots, r_a$ und $x_1, \ldots, x_a$
eine Lösung r für $\mathcal{P}$ auf x auf.

9.1 Multiplikation ganzer Zahlen

Zahlen in Binärdarstellung.
(Methoden funktionieren im Prinzip für jede beliebige Basis.)

Bekannt:



Volladdierer (5 zweistellige Bitoperationen)
liefert zu 3 Bits x, y, z die zwei Bits
$(c, s) = \text{fulladd}(x, y, z)$ mit

$$c = (x \wedge y) \vee (y \wedge z) \vee (z \wedge x)$$

(Übertragsbit, **Carry**bit) und

$$s = d \oplus e \oplus f$$

(**Summen**bit).

Bekannt: Serielle **Binäraddition**.

Input: Binärzahlen/-strings $a_{n-1} \ldots a_0$ und $b_{n-1} \ldots b_0$, ≥ 0

$\mathtt{c}_0 \leftarrow 0$; // Carry, Übertrag
for $\mathtt{i}$ **from** 0 **to** $n-1$ **do** $(\mathtt{c}_{\mathtt{i}+1}, \mathtt{s}_\mathtt{i}) \leftarrow \text{fulladd}(a_\mathtt{i}, b_\mathtt{i}, \mathtt{c}_\mathtt{i})$;
$\mathtt{s}_n \leftarrow \mathtt{c}_n$;
Ergebnis: $\mathtt{s}_n \ldots \mathtt{s}_0$.

Kosten: Nicht mehr als $5n = O(n)$ Bitoperationen.

Bekannt: Ganze Zahlen: Notation als Paar (Vorzeichen, Betrag (in Binärdarst.)),
z. B. $10, -1001, 101010, -11110$.

Rechnerintern: **Zweierkomplementdarstellung**.
Addition und Subtraktion auf die **Addition vorzeichenloser Zahlen** zurückführbar.
Kosten für n-Bit-Zahlen: $\leq 10n$ Bitoperationen.

Multiplikation zweier natürlicher Zahlen

„Schulmethode“ SM(x, y)

Faktoren x, y:	1	0	0	1	1	0	·	1	1	0	0	1	1
			1	0	0	1	1	0	0	0	0	0	0
				1	0	0	1	1	0	0	0	0	0
					0	0	0	0	0	0	0	0	0
						0	0	0	0	0	0	0	0
							1	0	0	1	1	0	0
		+						1	0	0	1	1	0
Produkt:			1	1	1	1	0	0	1	0	0	1	0

Multiplikation $\widehat{=}$ Addition von n Binärzahlen.

Allgemein:

Input: Binärzahlen $x = a_{n-1} \dots a_0$ und $y = b_{n-1} \dots b_0$

Bilde n Binärzahlen $d^{(0)}, \dots, d^{(n-1)}$:

$$d^{(i)} = (a_{n-1} \cdot b_i) \dots (a_0 \cdot b_i) \underbrace{0 \dots 0}_{i \text{ Nullen}}$$

und addiere alle diese.

$\leq n-1$ Additionen von Zahlen mit nicht mehr als $2n$ Bits:

$O(n^2)$ Bitoperationen.

Überlege: Was ändert sich bei Ziffernsatz $\{0, 1, \dots, b-1\}$ statt $\{0, 1\}$? (Z. B. $b = 10$.)

Geht es billiger als in Zeit $O(n^2)$?

Multiplikation mit **Divide-and-Conquer**-Strategie:

Eingabe: n-Bit-Binärzahlen x und y, eventuell Vorzeichen.

Falls $n \leq n_0$: Benutze Schulmethode.

Falls $n > n_0$: („**Teile**“)

Setze $k = \lceil n/2 \rceil$.

Schreibe $x = \underbrace{a_{n-1} \dots a_k}_{A} \underbrace{a_{k-1} \dots a_0}_{B}$

und $y = \underbrace{a_{n-1} \dots a_k}_{C} \underbrace{a_{k-1} \dots a_0}_{D}$

Dann $x = A \cdot 2^k + B$ und $y = C \cdot 2^k + D$.

Also

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

Erste Idee: Berechne **rekursiv** $A \cdot C$, $A \cdot D$, $B \cdot C$, $B \cdot D$,

und füge die Produkte durch einige Additionen zum Resultat $x \cdot y$ zusammen.

Kosten für n-Bit-Zahlen (für eine Konstante c):

$$C(n) \leq \begin{cases} 1 & \text{für } n = 1 \\ 4 \cdot C(n/2) + c \cdot n & \text{für } n > 1. \end{cases}$$

Der Summand $4 \cdot C(n/2)$ erfasst die Kosten der vier rekursiven Aufrufe, der Summand cn die Kosten der Additionen.

Man **kann** zeigen (machen wir später, „Master-Theorem“):

Die Anzahl der Bitoperationen ist wieder $\Theta(n^2)$, nicht besser als Schulmethode.

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

Trick:

$E := A - B$ und $F := C - D$ (sieht sinnlos aus . . .)

Bemerke: $|E|$ und $|F|$ haben als Betrag der Differenz von zwei nichtnegativen k-Bit-Zahlen höchstens k Bits. – Dann:

$E \cdot F = (A - B) \cdot (C - D) = A \cdot C + B \cdot D - (A \cdot D + B \cdot C)$.

Also:

$$A \cdot D + B \cdot C = A \cdot C + B \cdot D - E \cdot F.$$

Eingesetzt:

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

Nur noch **drei** rekursive Multiplikationen von k-Bit-Zahlen nötig!

Algorithmus Karatsuba(x, y)

Eingabe: Zwei n-Bit-Zahlen x und y, nichtnegativ
if $n \leq n_0$ **then return** SM(x, y) // Multiplikation mit Schulmethode
else
$k \leftarrow \lceil n/2 \rceil$;
zerlege $x = A \cdot 2^k + B$ und $y = C \cdot 2^k + D$; // vier k-Bit-Zahlen
$E \leftarrow A - B$ und $F \leftarrow C - D$; // k-Bit-Zahlen, mit Vorzeichen
$G \leftarrow$ Karatsuba(A, C); // Rekursion
$H \leftarrow$ Karatsuba(B, D); // Rekursion
$I \leftarrow$ Karatsuba$(|E|, |F|)$; // Rekursion
return $G \cdot 2^{2k} + (G + H - \text{sign}(E) \cdot \text{sign}(F) \cdot I) \cdot 2^k + H$.

(sign(a) ist das Vorzeichen der ganzen Zahl a.
Multiplikationen mit ± 1 und Zweierpotenzen sind „umsonst“.)

Beispiel: Mit Dezimalzahlen, $n_0 = 2$.

(Methode funktioniert mit jeder Basis $b \geq 2$. In der Informatik interessante Basiszahlen b:
2, 8, 10, 16 (Hexzahlen), 256 (Ziffern sind Bytes), 2^{16}, 2^{32} (Ziffern sind (Halb-)Wörter), . . .)

$n = 8$, $x = 76490358$, $y = 35029630$.

$A = 7649$, $B = 0358$, $C = 3502$, $D = 9630$.

$E = A - B = 7291$, $F = C - D = -6128$, $\mathsf{sign}(E) \cdot \mathsf{sign}(F) = 1 \cdot (-1) = -1$.

Jeweils ≤ 4 Dezimalziffern.

Rekursion für $A \cdot C$:
$a = 76, b = 49, c = 35, d = 02$.
$e = a - b = 27$, $f = c - d = 33$.

Weil z.B. $n_0 = 2$ ist, wird direkt multipliziert:
$g = a \cdot c = 76 \cdot 35 = 2660$, $h = b \cdot d = 98$, $i = |e| \cdot |f| = 27 \cdot 33 = 891$.

3 Multiplikationen von 2-Bit-Zahlen!

Ergebnis:

$$G = A \cdot C = 2660 \cdot 10^4 + (2660 + 98 - 891) \cdot 10^2 + 98 = 26786798.$$

Analog, jeweils rekursiv:
$H = B \cdot D = 03447540$, $I = |E| \cdot |F| = 44679248$.

Ergebnis:

$$x \cdot y = 26786798 \cdot 10^8 + (26786798 + 03447540 - (-1) \cdot 44679248) \cdot 10^4 + 03447540 = 267942893907540$$

Multiplikation mit 10^k: Anhängen von Nullen. $\Rightarrow$ Beim Kombinationsschritt gibt es nur Additionen!

Laufzeitanalyse:

Es sei $n_0 = 1$ und n sei eine Zweierpotenz: $\boldsymbol{n = 2^\ell}$.

$T_{\mathsf{Ka}}(n) :=$ Anzahl der Bit-Operationen, die der Algorithmus von Karatsuba auf einer Eingabe aus zwei n-Bit-Zahlen macht, mit $n_0 = 1$. Klar: $T_{\mathsf{Ka}}(1) = 1$.

Für einen Input der Größe $n > n_0$ müssen wir (rekursiv) $a = 3$ Teilinstanzen bearbeiten, für Parametergröße $\lceil n/b \rceil = \lceil n/2 \rceil$, also $b = 2$,
und im Kombinationsschritt einige Additionen und Subtraktionen von Zahlen mit maximal $2n$ Bits durchführen, was zusätzlich $O(n)$ Bitoperationen erfordert.

Rekurrenzungleichung:

$$T_{\mathsf{Ka}}(n) \leq \begin{cases} 1 & \text{für } n = 1, \\ 3 \cdot T_{\mathsf{Ka}}(n/2) + c \cdot n & \text{für } n > 1, \end{cases}$$

wobei c konstant ist.

Direkte Rechnung,

wobei die Rekurrenzungleichung für n, $n/2$, $n/4$, $n/8$, ... an der Stelle von n benutzt wird:

$$
\begin{aligned}
T_{\mathsf{Ka}}(n) &\leq 3 \cdot T_{\mathsf{Ka}}(n/2) + c \cdot n \\
&\leq 3 \cdot (3 \cdot T_{\mathsf{Ka}}((n/2)/2) + c \cdot n/2) + c \cdot n \\
&= 3^2 \cdot T_{\mathsf{Ka}}(n/2^2) + c \cdot 3 \cdot n/2 + c \cdot n \\
&\leq 3^3 \cdot T_{\mathsf{Ka}}(n/2^3) + c \cdot 3^2 \cdot n/2^2 + c \cdot 3 \cdot n/2 + c \cdot n \\
&\;\;\vdots \\
&\leq 3^\ell \cdot T_{\mathsf{Ka}}(n/2^\ell) + c \cdot \sum_{0 \leq j < \ell} 3^j \cdot n/2^j. \\
&= 3^\ell \cdot T_{\mathsf{Ka}}(1) + cn \cdot \sum_{0 \leq j < \ell} (3/2)^j.
\end{aligned}
$$

(Fortsetzung folgt)

Wir setzen fort, unter Benutzung der Formel für geometrische Reihen:

$$T_{\mathsf{Ka}}(2^\ell) \leq 3^\ell \cdot \underbrace{T_{\mathsf{Ka}}(1)}_{=1} + cn \cdot \sum_{0 \leq j < \ell} (3/2)^j$$

$$= 3^\ell + c \cdot 2^\ell \cdot \frac{(\frac{3}{2})^\ell - 1}{\frac{3}{2} - 1}$$

$$< 3^\ell \cdot (1 + 2c).$$

Nun gilt $3^\ell = (2^{\log_2 3})^\ell = 2^{(\log_2 3)\ell} = 2^{\ell \log_2 3} = (2^\ell)^{\log_2 3} = n^{\log_2 3}$.

$\Rightarrow T_{\mathsf{Ka}}(n) \leq (1 + 2c) n^{\log_2 3} = O(n^{1{,}585})$.

Satz 9.1.1

Beim Karatsuba-Multiplikationsalgorithmus beträgt die Anzahl der Bitoperationen und die Rechenzeit $O(n^{\log_2 3}) = O(n^{1{,}585})$.

Bemerkungen

$n^{\log_2 3}$ ist viel kleiner als n^2!

Dieselbe Rechnung mit „4“ an der Stelle von „3“ beweist, dass das naive rekursive Verfahren mit vier rekursiven Aufrufen (Folie 7) Rechenzeit $O(n^{\log_2 4}) = O(n^2)$ hat.

In der Praxis (Implementierung in Software): für $w =$ Wortlänge des Rechners (z. B. $w = 32$) für Operationen auf Ziffern die eingebaute Multiplikations-Hardware benutzen. (D. h.: mit Basis $b = 2^w$ rechnen.)

Für Zahlen bis zu einer Länge von n_0 Worten: Schulmethode. (Auch in der Rekursion!)

Nur für noch längere Zahlen D-a-C-Verfahren benutzen.
Welches n_0 optimal ist, hängt von der Hardware, vom Compiler und von Programmierdetails ab.
Ein gutes n_0 kann man experimentell bestimmen. (Oft gut: $24 \leq n_0 \leq 32$.)

(Studie hierzu: Buch [D./Mehlhorn/Sanders].)

Beispiele: 1024 Binärziffern, gut 300 Dezimalziffern.

Will man mit so riesigen Zahlen rechnen?

Ja! – Kryptographie!

Ignoriere Additionen, setze $n_0 = 32$.
$M'_{\mathsf{Ka}}(n) = \#$(32-Bit-Mult. bei Karatsuba für zwei n-Bit-Zahlen), mit $n_0 = 32$.

Rekurrenzgleichung:

$$M'_{\mathsf{Ka}}(n) = \begin{cases} 1 & \text{für } n \leq n_0, \\ 3 \cdot M'_{\mathsf{Ka}}(n/2) & \text{für } n > n_0, \end{cases}$$

Daraus: $M'_{\mathsf{Ka}}(2^\ell) = 3^{\ell-5} \cdot M'_{\mathsf{Ka}}(2^5) = 3^{\ell-5}$.

$2^{10} = 1024$ Binärziffern, $\ell = 10$:

Karatsuba: $3^5 = 243$ Multiplikationen$_{32}$;
Schulmethode: $(2^{\ell-5})^2 = 1024$ Multiplikationen$_{32}$.

$2^{15} = 32768$ Binärziffern, $\ell = 15$, ca. 9900 Dezimalziffern:

Karatsuba: $3^{10} = 59049$ Multiplikationen$_{32}$;
Schulmethode:
$(2^{15-5})^2 = 2^{20}$ Multiplikationen$_{32}$, mehr als 1 Million!

Ersparnis: Faktor 18.

Geht es noch besser? Theoretisch ja, praktisch eigentlich kaum.

Mitteilung:

- Schönhage-Strassen (1971): Multiplikation zweier n-Bit-Zahlen in Zeit $O((n \log n) \cdot \boldsymbol{\log \log n})$.

  [A. Schönhage und V. Strassen: Schnelle Multiplikation großer Zahlen, Computing 7, 1971, Springer-Verlag, S. 281–292]
- Fürer (2007), De *et al.* (2008): Multiplikation zweier n-Bit-Zahlen in Zeit $O((n \log n) \cdot \boldsymbol{2^{\log^* n}})$. ($\log^* n$ wächst extrem langsam.)

  [M. Fürer: Faster integer multiplication, STOC 2007, S. 57–66].
  [A. De, P. P. Kurur, C. Saha, R. Saptharishi: Fast integer multiplication using modular arithmetic. STOC 2008, S. 499–506 und SIAM J. Comput. 42(2): 685-699 (2013)].
- David Harvey, Joris Van Der Hoeven (2019/21): Multiplikation zweier n-Bit-Zahlen in Zeit $\boldsymbol{O(n \log n)}$.
  [Annals of Mathematics Vol. 193, No. 2 (March 2021), pp. 563–617. `https://doi.org/10.4007/annals.2021.193.2.4`]

Dabei ist $\log^* n$ definiert als die kleinste Zahl i mit

$$\underbrace{\log \log \ldots \log}_{i\text{-mal}} n \leq 1.$$

Also: $\log^* 2 = 1$, $\log^* 4 = 2$, $\log^* 16 = 3$,

$\log^*(65536) = 4$, $\log^*(2^{65536}) = 5$, $\log^*(2^{2^{65536}}) = 6$.

$2^{2^{65536}}$ ist schon eine **sehr** große Zahl.

$\Rightarrow$ Zahlen n mit $\log^* n > 6$ kommen in der Praxis nicht vor.

9.2 Matrixmultiplikation

Es sei R irgendein **Ring**.[1]

$A = (a_{ij})_{1 \le i,j \le n}$, $B = (b_{ij})_{1 \le i,j \le n}$ seien $n \times n$-Matrizen über R.

Aufgabe: Berechne die Produktmatrix $P = A \cdot B$, d.h. $P = (p_{ij})_{1 \le i,j \le n}$ mit

$$p_{ij} = \sum_{1 \le k \le n} a_{ik} b_{kj}.$$

Eine **naive** Implementierung gemäß dieser Formel kostet n^3 Ring-Multiplikationen und $n^2(n-1)$ Ring-Additionen, hat also Rechenzeit $\boldsymbol{O(n^3)}$.

Strassen (1969): Es geht mit weniger Multiplikationen! Ansatz: Divide-and-Conquer.

[1] Man kann addieren, subtrahieren, multiplizieren, gemäß Standard-Rechenregeln. Bsp.: $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$.

Wir nehmen an: $n = 2^\ell$, Zweierpotenz.

Eingabe: Zwei $n \times n$-Matrizen A und B.

Falls $n \leq n_0$: Berechne $A \cdot B$ mit der direkten Methode. Kosten n_0^3 Multiplikationen.

Falls $n > n_0$, zerlege A, B in jeweils 4 quadratische $(\frac{n}{2} \times \frac{n}{2})$-Teilmatrizen:

$$A = \left(\begin{array}{c|c} C & D \\ \hline E & F \end{array}\right), \qquad B = \left(\begin{array}{c|c} G & H \\ \hline K & L \end{array}\right).$$

Dann gilt (wie nicht schwer nachzukontrollieren ist):

$$A \cdot B = \left(\begin{array}{c|c} C \cdot G + D \cdot K & C \cdot H + D \cdot L \\ \hline E \cdot G + F \cdot K & E \cdot H + F \cdot L \end{array}\right).$$

Haben:

$$A \cdot B = \left(\begin{array}{c|c} C \cdot G + D \cdot K & C \cdot H + D \cdot L \\ \hline E \cdot G + F \cdot K & E \cdot H + F \cdot L \end{array} \right).$$

Dies suggeriert einen rekursiven Ansatz, in dem 8 Multiplikationen von $(\frac{n}{2} \times \frac{n}{2})$-Teilmatrizen durchgeführt werden.

Einfache Analyse ergibt: Dies führt zu n^3 Multiplikationen in R, kein Gewinn.

(Unten zeigen wir mit dem Master-Theorem: $O(n^3)$.)

Strassen-Trick: 7 Multiplikationen genügen (mit einigen Additionen/Subtraktionen).

$$P_1 = C \cdot (H - L)$$
$$P_2 = (C + D) \cdot L$$
$$P_3 = (E + F) \cdot G$$
$$P_4 = F \cdot (K - G)$$
$$P_5 = (C + F) \cdot (G + L)$$
$$P_6 = (D - F) \cdot (K + L)$$
$$P_7 = (C - E) \cdot (G + H)$$

Dann:

$$A \cdot B = \left(\begin{array}{c|c} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ \hline P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{array} \right)$$

Von Hand nachzukontrollieren!

18 Additionen von $(\frac{n}{2} \times \frac{n}{2})$-Matrizen.

(Alternative Methode, etwas komplizierter, benötigt nur 15 Additionen.)

Aufwandsanalyse:

Rekurrenzungleichung für die Anzahl der Operationen:

$$T_{\mathsf{Str}}(n) \leq \begin{cases} 1 & \text{für } n = 1, \\ 7 \cdot T_{\mathsf{Str}}(n/2) + c \cdot n^2 & \text{für } n > 1. \end{cases}$$

$n = 1$: Eine Ringmultiplikation.

$n > 1$: Neben den rekursiven Aufrufen genau 18 Additionen von $(\frac{n}{2} \times \frac{n}{2})$-Matrizen, mit Kosten $18 \cdot (n/2)^2 = 4{,}5 \cdot n^2$.

Also: $c = 4{,}5$ ist geeignete Konstante.

Nun: Rechnung für $n = 2^\ell$ wie bei der Analyse des Karatsuba-Algorithmus.

$$
\begin{aligned}
T_{\mathsf{Str}}(n) &\leq 7 \cdot T_{\mathsf{Str}}(n/2) + c \cdot n^2 \\
&\leq 7^2 \cdot T_{\mathsf{Str}}(n/4) + c \cdot 7 \cdot (n/2)^2 + c \cdot n^2 \\
&\leq 7^3 \cdot T_{\mathsf{Str}}(n/8) + c \cdot 7^2 \cdot (n/4)^2 + c \cdot 7 \cdot (n/2)^2 + c \cdot n^2 \\
&\quad \vdots \\
&\leq 7^\ell \cdot \underbrace{T_{\mathsf{Str}}(n/2^\ell)}_{=1} + c \cdot n^2 \cdot \sum_{0 \leq j < \ell} \frac{7^j}{(2^j)^2} \\
&= 7^\ell + c \cdot n^2 \cdot \sum_{0 \leq j < \ell} \left(\frac{7}{4}\right)^j < 7^\ell + c \cdot (2^\ell)^2 \cdot \frac{(\frac{7}{4})^\ell}{\frac{7}{4} - 1} \\
&= 7^\ell \cdot (1 + 4c/3).
\end{aligned}
$$

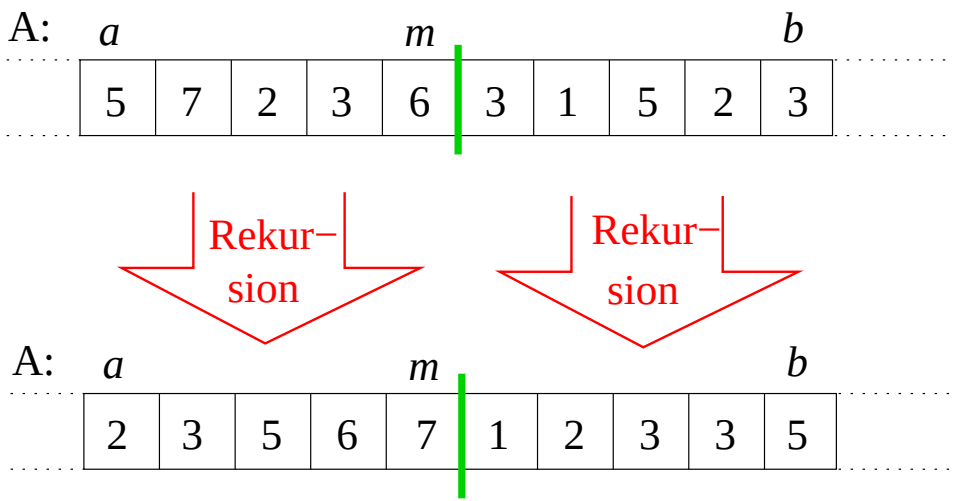
Wie beim Karatsuba-Algorithmus: $T_{\mathsf{Str}}(n) = T_{\mathsf{Str}}(2^\ell) = O(7^\ell)$.

Dabei: $7^\ell = 2^{\ell \log_2 7} = n^{\log_2 7}$ mit $\log_2 7 \approx 2{,}81$.

Satz 9.1.2

Der Algorithmus von **Strassen** für die Matrixmultiplikation führt bei $n \times n$-Matrizen über dem Ring R als Eingabe höchstens $n^{\log_2 7}$ Ringmultiplikationen und $O(n^{\log_2 7})$ Ringadditionen und -subtraktionen aus. Dabei ist $\log_2 7 \approx 2{,}81$.

9.3 Erinnerung: Mergesort (Kap. 6.2)



Prozedur r_MergeSort(a, b):

0) Falls $\boldsymbol{n} := b - a + 1 \leq n_0$: Insertionsort.

1) $m := \lfloor (a+b)/2 \rfloor$; Aufteilen von $\mathtt{A}[a \ldots b]$ in $\mathtt{A}[a \ldots m]$ und $\mathtt{A}[m+1 \ldots b]$ (Längen $\lceil n/2 \rceil$ und $\lfloor n/2 \rfloor$).

2) Diese beiden Segmente werden **rekursiv** sortiert.

3) **Mischen** von $\mathtt{A}[a \ldots m]$ und $\mathtt{A}[m+1 \ldots b]$: **Merge**(a, m, b). ($\leq n-1$ Vergleiche.)

Rekurrenzgleichung für Vergleichsanzahl $C(n)$ für Sortieren eines Teilarrays der Länge n:

$$C(n) = \begin{cases} 0, & \text{für } n = 0, \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + n - 1, & \text{für } n \geq 1. \end{cases}$$

Gezeigt in 6.2, mit ad-hoc-Beweis: $C(n) = n\lceil \log n \rceil - (2^{\lceil \log n \rceil} - 1) \leq n \log n$.

Worst-Case-Rechenzeit: $O(n \log n)$.

9.4 Das Master-Theorem

Wir betrachten Rekurrenzungleichungen der folgenden Form:

$$B(n) \leq \begin{cases} g & \text{, falls } n \leq n_0 \\ a \cdot B(n/b) + f(n) & \text{, sonst.} \end{cases}$$

Dabei: $a \geq 1$ und $b > 1$ und g und n_0 sind Konstante,
$f : \mathbb{N} \to \mathbb{N}$ ist eine monoton wachsende Funktion.

Falls n/b keine ganze Zahl ist, sollte man sich an Stelle von $B(n/b)$ z. B. $B(\lceil n/b \rceil)$ geschrieben denken. Es muss dann gelten: $n > n_0 \Rightarrow \lceil n/b \rceil < n$.

Ziel: „Geschlossene“ Abschätzung von $B(n)$ nach oben.

Gesehen: Eine solche Rekurrenzungleichung ergibt sich bei der Analyse eines Divide-and-Conquer-Algorithmus mit:

- Basisfall (Größe $\leq n_0$) hat höchstens Kosten g,
- aus Instanz der Größe $n > n_0$ werden a Teilinstanzen der Größe n/b (passend gerundet) gebildet („**teile**“),
- es erfolgen a **rekursive Aufrufe**,
- und die a Lösungen werden zusammengesetzt („**kombiniere**“).
- Kosten für das Aufspalten und das Kombinieren: $f(n)$.

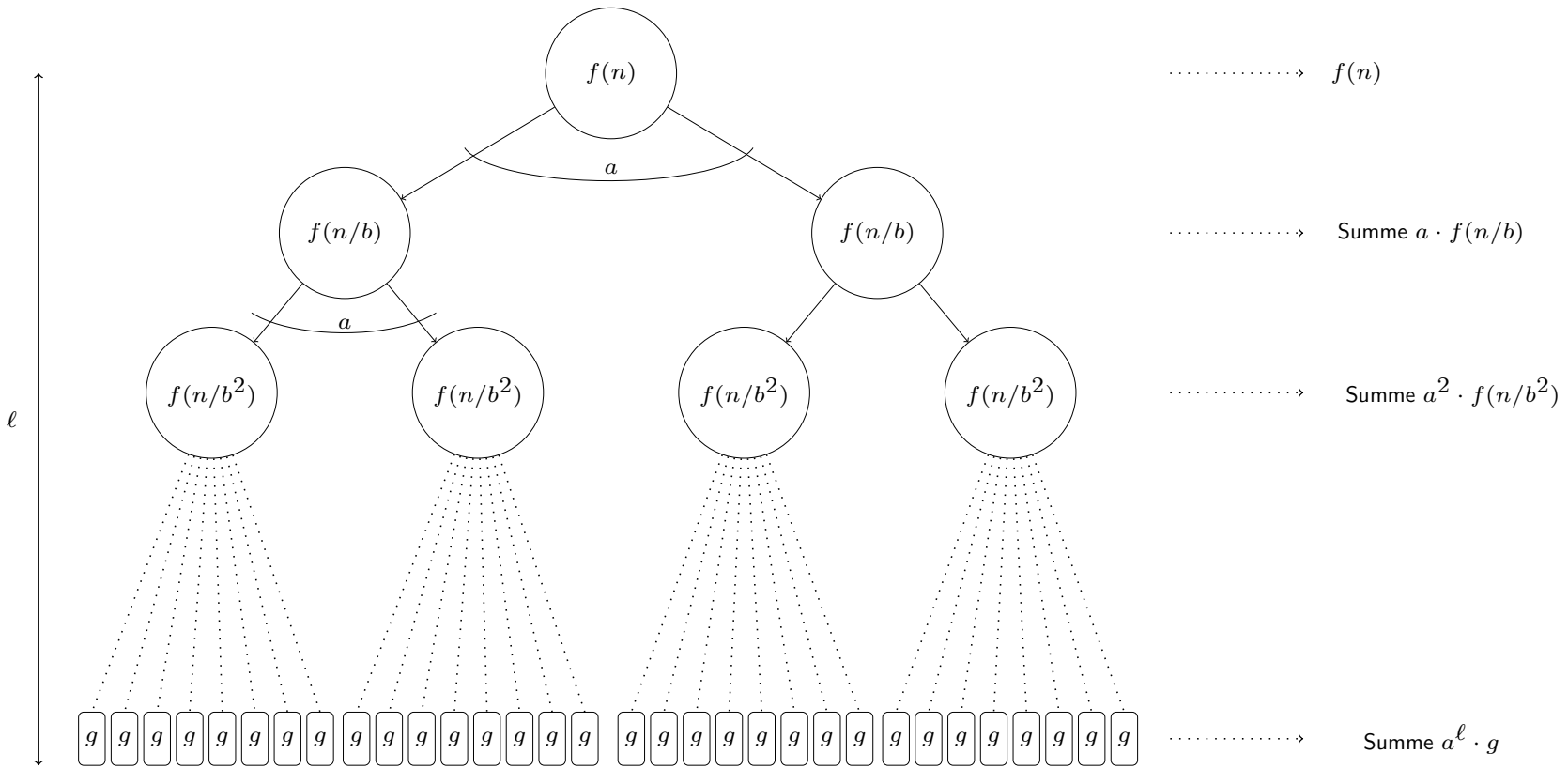
O.B.d.A.: $B(n)$ monoton wachsend. – Sonst definiere:

$$\hat{B}(n) := \max\{B(i) \mid 1 \leq i \leq n\}, \text{ für } n \geq 0.$$

$\Rightarrow \hat{B}(n)$ ist monoton und erfüllt dieselbe Rekurrenzungleichung, und $B(n) \leq \hat{B}(n)$.

Vereinfachende Annahmen (nicht wesentlich): $n_0 = 1$ und $n = b^\ell$.
$b > 1$ und $a \geq 1$, beide sind ganzzahlig.

„Rekursionsbaum“: Veranschaulicht Kostenaufteilung, (mathematisch nicht notwendig).
Level 0: Wurzel, hat Eintrag $f(n)$ und hat a Kinder auf Level 1.
Knoten v auf Level $i < \ell$ hat Eintrag $f(n/b^i)$ und hat a Kinder auf Level $i+1$.
Knoten auf Level ℓ sind Blätter, sie haben Eintrag g.



Lemma 9.4.1

Wenn v ein Knoten auf Level i ist, dann gilt:

$B(n/b^i) \leq$ Summe der Einträge im Unterbaum unter v.

(Beweis durch Induktion über $i = \ell, \ell - 1, \ldots, 1$.)

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

Auf Level i gibt es a^i Knoten mit Eintrag $f(n/b^i)$. Summation liefert:

$$\boxed{B(n) \leq \sum_{0 \leq i < \ell} a^i \cdot f(n/b^i) + a^\ell \cdot g.}$$

Erster Term $B_1(n)$: Beitrag zu Gesamtkosten aus dem Inneren des Baums.

Zweiter Term $B_2(n)$: Beitrag von den Blättern. (Algorithmisch: Die a^ℓ Basisfälle.)

Beobachte: $a^\ell = (b^{\log_b a})^\ell = (b^\ell)^{\log_b a} = n^{\log_b a}$, also $B_2(n) = O(n^{\log_b a})$.

Herleitung der Ungleichung in der Box durch iterierte Anwendung der Rekurrenzungleichung, ohne Veranschaulichung durch Baum (vgl. auch die konkreten Fälle in 6.1 und 6.2):

$$
\begin{aligned}
B(n) &\leq a \cdot B(n/b) + f(n) \\
&\leq a \cdot (a \cdot B((n/b)/b) + f(n/b)) + f(n) \\
&= a^2 \cdot B(n/b^2) + a \cdot f(n/b) + f(n) \\
&\leq a^3 \cdot B(n/b^3) + a^2 \cdot f(n/b^2) + a \cdot f(n/b) + f(n) \\
&\quad \vdots \\
&\leq a^j \cdot B(n/b^j) + a^{j-1} \cdot f(n/b^{j-1}) + \cdots + a \cdot f(n/b) + f(n) \\
&\quad \vdots \\
&\leq a^\ell \cdot \underbrace{B(n/b^\ell)}_{=B(1)=g} + \sum_{0 \leq i < \ell} a^i f(n/b^i) = B_2(n) + B_1(n).
\end{aligned}
$$

Man erkennt hier, dass man nicht annehmen muss, dass a ganzzahlig ist.

Erster Term: $B_1(n) = \sum_{0 \leq i < \ell} a^i \cdot f(n/b^i)$.

3 Fälle (je nach Verhalten des Gesamtaufwandes $a^i \cdot f(n/b^i)$ auf Level i, für $i = 0, \ldots, \ell - 1$):

Intuitiv:

1. Fall: $a^i \cdot f(n/b^i)$ wächst mit i an.

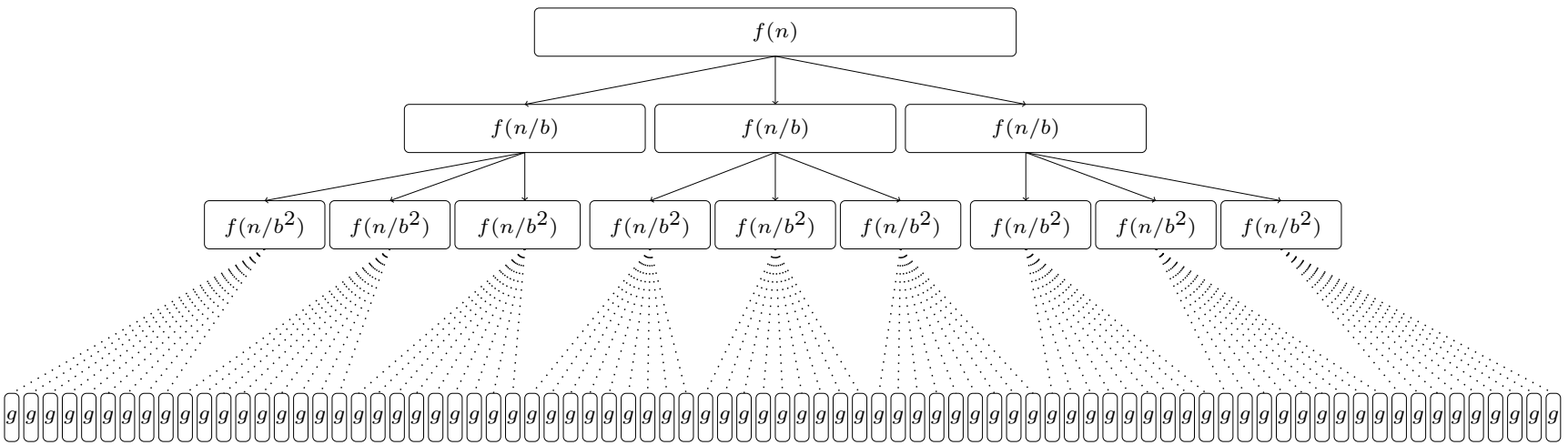
2. Fall: $a^i \cdot f(n/b^i)$ bleibt in etwa gleich über alle i.

3. Fall: $a^i \cdot f(n/b^i)$ schrumpft mit i.

Genaueres folgt.

1. Fall: $f(n) = O(n^\alpha)$ mit $b^\alpha < a$, oder, äquivalent $\alpha < \log_b a$.

Die Beiträge aus den unteren Baumebenen (kleine Instanzen) dominieren.
(Fallende Größe wird durch wachsende Anzahl überkompensiert. Im Bild: Die Breite des $f(n/b^i)$-Knotens ist $f(n/b^i)$.)



Wir benutzen mehrfach die Summenformel für geometrische Reihen: $\sum_{0 \le i < \ell} q^i = \frac{q^\ell - 1}{q-1}$, für $q \neq 1$, s. Kapitel 1, Folie 37. Für $q > 0$, $q \neq 1$, folgt:

$$\sum_{0 \le i < \ell} q^i < \frac{q^\ell}{q-1}. \qquad (*)$$

$$B_1(n) = \sum_{0 \le i < \ell} a^i \cdot f(n/b^i)$$

$$= O\Big(\sum_{0 \le i < \ell} a^i \cdot \Big(\frac{n}{b^i}\Big)^\alpha\Big) = O\Big(n^\alpha \cdot \sum_{0 \le i < \ell} \Big(\frac{a}{b^\alpha}\Big)^i\Big)$$

$$\overset{(*)}{=} O\left(n^\alpha \cdot \frac{(a/b^\alpha)^\ell}{(a/b^\alpha) - 1}\right) \qquad \text{(geom. Reihe)}$$

$$= O\left(n^\alpha \cdot a^\ell \cdot \frac{1}{(b^\ell)^\alpha}\right) \qquad \text{(Nenner ist konstant)}$$

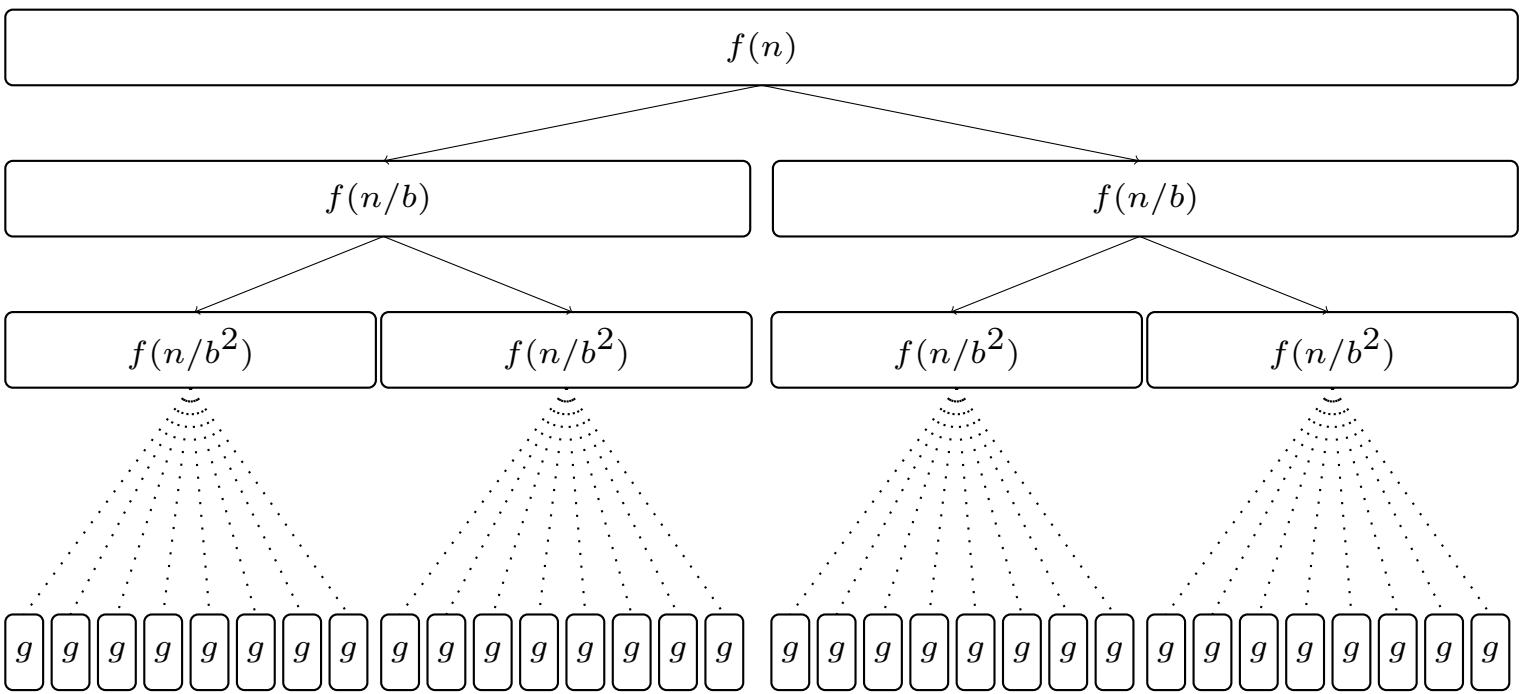
$$= O(a^\ell). \qquad (b^\ell = n, \text{ Kürzen})$$

Also: $B(n) \le B_1(n) + B_2(n) = O(a^\ell) = O(n^{\log_b a})$.

Beispiele: **Karatsuba**-Algorithmus ($b = 2$, $a = 3$, $f(n) = O(n)$),
Strassen-Algorithmus ($b = 2$, $a = 7$, $f(n) = O(n^2)$).

2. Fall: $f(n) = O(n^{\log_b a})$, oder, äquivalent, $f(n) = O(n^\alpha)$ mit $\alpha = \log_b a$.

$f(n/b^i)$ wächst mit n/b^i, $i = \ell - 1, \ldots, 0$, höchstens mit einer Rate, die durch das Schrumpfen der Größe der Baumebene ausgeglichen wird. Der Gesamtaufwand ist beschränkt durch den Aufwand für die Ebene direkt über den Blättern, multipliziert mit der Anzahl der Levels.



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

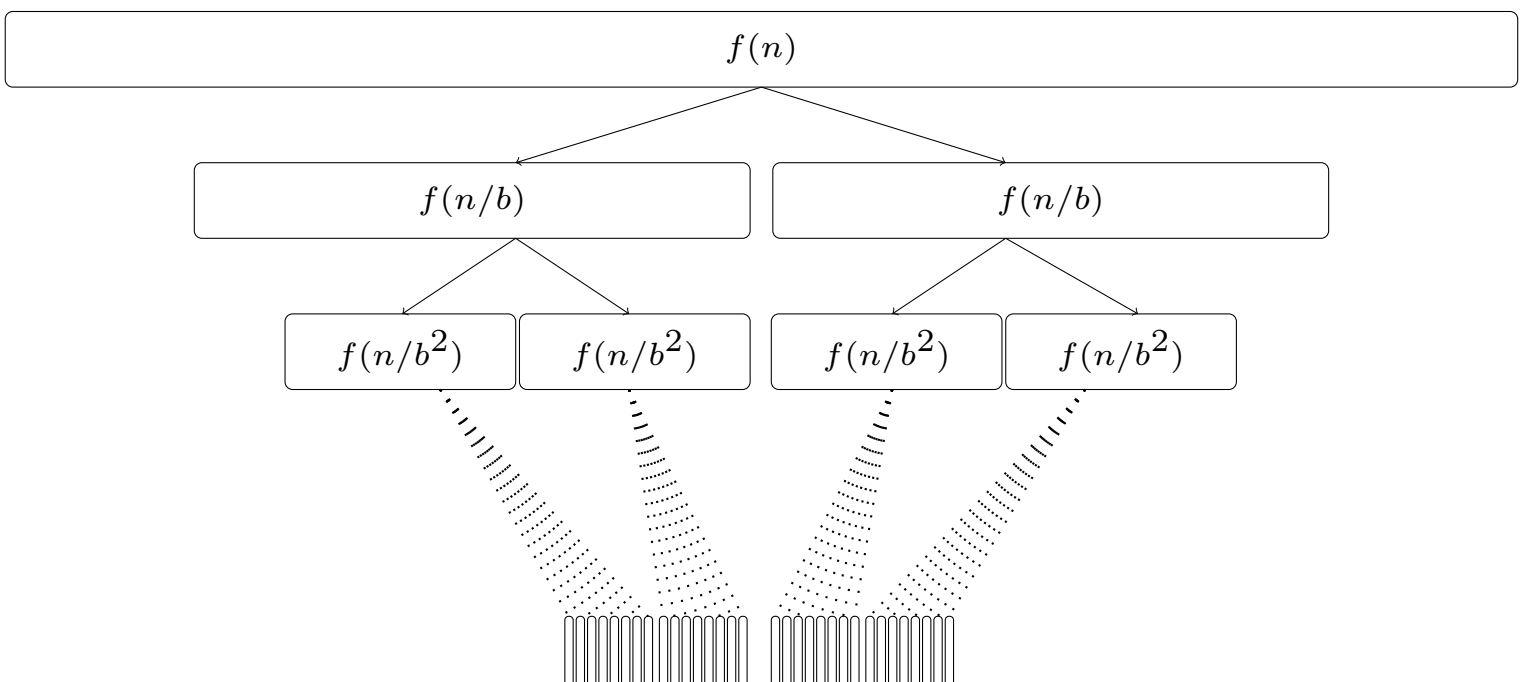
Also: $B(n) \le B_1(n) + B_2(n) = O(n^{\log_b a} \cdot \log n) + O(n^{\log_b a}) = O(n^{\log_b a} \cdot \log n)$.

Beispiele: **Mergesort** ($b = a = 2$), Binäre Suche ($b = 2$, $a = 1$, $\log_b a = 0$, $n^0 = 1$).

3. Fall: $f(n) = \Omega(n^\alpha)$, mit $b^\alpha > a$

und es gibt $c < 1$ mit: $f(n/b) \leq (c/a) \cdot f(n)$, für alle $n \geq 1$ („Regularitätsbedingung“).

Wenn die Größe des Inputs von n auf n/b sinkt, fallen die Kosten im Baumknoten garantiert mindestens um den Faktor c/a. $f(n/b^i)$ fällt dann so rasch mit wachsendem i, dass die wachsende Anzahl a^i der Baumknoten (über)kompensiert wird, und die unteren Baumebenen vernachlässigbares Gewicht haben.



Mit der Regularitätsbedingung erhalten wir nacheinander:
$f(n/b) \leq \frac{c}{a} \cdot f(n)$, $f(n/b^2) \leq \left(\frac{c}{a}\right)^2 \cdot f(n)$, $\ldots$, $f(n/b^i) \leq \left(\frac{c}{a}\right)^i \cdot f(n)$. Damit:

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

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

Satz 9.4.2 Das Master-Theorem (Einfache Form)

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

für ganzzahlige konstante $b > 1$ und $a > 0$. Dann gilt für $n = b^\ell$:

1. Fall: $f(n) = O(n^\alpha)$ mit $\alpha < \log_b a$. Dann: $B(n) = O(n^{\log_b a})$.

2. Fall: $f(n) = O(n^\alpha)$ mit $\alpha = \log_b a$. Dann: $B(n) = O(n^{\log_b a} \cdot \log n)$.

3. Fall: $f(n) = \Omega(n^\alpha)$ mit $\alpha > \log_b a$ **und** $f(n/b) \leq \frac{c}{a} \cdot f(n)$, für $c < 1$ konstant.
Dann gilt $B(n) = O(f(n))$.

Verschiedene Einschränkungen, die wir im Beweis benutzt haben, sind nicht wirklich nötig. Dieselben Formeln gelten in folgenden Fällen:

- Beliebige $n \in \mathbb{N}$, nicht nur $n = b^\ell$.
- Verallgemeinerte Relation
  $B(n) \leq a \cdot B(n') + f(n)$, $n' \leq \lceil n/b \rceil + d$, für d konstant.
- $b > 1$, $a > 0$ beliebig reell, Schranke n_0 in der Basisbedingung,
  wenn nur $\lceil n/b \rceil + d < n$ für $n > n_0$.
- Analoge untere Schranken.

Genaueres dazu kann man in den Büchern von [D./Mehlhorn/Sanders] oder [Cormen, Leiserson, Rivest und Stein] nachlesen.

9.5 Das Auswahlproblem („selection“)

Beispiel: Finde den drittkleinsten Eintrag in $(7, 3, 8, 5, 4, 9, 2, 10, 4, 8, 7, 12, 9, 5)$!

Gegeben ist eine Folge $(a_1, \ldots, a_n)$ von n Objekten aus einer totalen Ordnung $(D, <)$ (in Array oder als Liste), sowie eine Zahl k, $1 \leq k \leq n$.

Aufgabe: Finde das Element x der Folge, das **Rang k** hat, d. h. das Objekt x in der Liste mit $|\{i \mid a_i < x\}| < k \leq |\{i \mid a_i \leq x\}|$.

Spezialfall:
Der **Median** einer Folge mit n Einträgen ist das Element mit Rang $\lceil n/2 \rceil$.
(Median$((2, 4, 7, 9)) = 4$, Median$((4, 7, 9)) = 7$.)

Einfache Lösung: Sortiere die Folge, mit Ergebnis $(b_1, \ldots, b_n)$, dann wähle $x = b_k$.
Kosten hierfür: $n \log n$ Vergleiche, Zeit $\Theta(n \log n)$.

Wir betrachten einen **randomisierten Algorithmus** für das Auswahlproblem:

Quickselect (C. A. R. („Tony“) Hoare, wie Quicksort)

Ansatz: Wie bei Quicksort.

Input: Folge $(a_1, \ldots, a_n)$, Zahl k, $1 \leq k \leq n$.

Vereinfachende Annahme, vorerst: Die a_i sind verschieden.

Falls $n = 1$, ist nichts zu tun.

Falls $n = 2$, sortiere mit einem Vergleich, Ergebnis (b_1, b_2), gib Element b_k zurück.

Falls $n \geq 3$:

Wähle einen Eintrag x aus $(a_1, \ldots, a_n)$ als partitionierendes Element **zufällig**.

Zerlege $(a_1, \ldots, a_n)$ mit $n-1$ Vergleichen in eine Teilfolge $(b_1, \ldots, b_{p-1})$, alle $< x$, in das Element x, und eine Teilfolge $(c_{p+1}, \ldots, c_n)$, alle $> x$.

1. Fall: $p = k$.
Das Ergebnis ist x.

2. Fall: $p > k$.
Finde (rekursiv) in $(b_1, \ldots, b_{p-1})$ das Element vom Rang k.

3. Fall: $p < k$.
Finde (rekursiv) in $(c_{p+1}, \ldots, c_n)$ das Element vom Rang $k - p$.

Prozedur rqSelect(a, b, k)

// Rekursive Prozedur im Quickselect-Algorithmus, $1 \leq a \leq b \leq n$, $a \leq k \leq b$.
// Vorbedingung: Alle Einträge vom Rang $< a$ [$> b$] links [rechts] von `A[`$a..b$`]`
// Nachbedingung: Eintrag vom Rang k in `A[`k`]`, kleinere links, größere rechts davon.

(1) **if** $a = b\ (= k)$ **then return**;
(2) `s` $\leftarrow$ ein zufälliges Element von $\{a, \ldots, b\}$;
(3) **if** $(a <$ `s`$)$ **then** vertausche `A[`a`]` und `A[s]`;
(4) **partition**$(a, b,$ `p`$)$; // wie bei `rqsort`, Abschnitt 6.3, Folie 30
(5) **if** $k =$ `p` **then return**;
(6) **if** $k <$ `p` **then rqSelect**$(a,$ `p` $- 1, k)$
(7) **else rqSelect**$($`p` $+ 1, b, k)$.

Mögliche Anpassungen:
(a) Sortiere z.B. mit Insertionsort, wenn $b - a$ sehr klein ist.
(b) Anstelle von Rekursion benutze Iteration.

Korrektheit: Klar.

Zu analysieren: (Erwartete) Rechenzeit.

Sei n die Eingabelänge, $k \in \{1, \ldots, n\}$ der gegebene Rang.

Klar: Rechenzeit ist proportional zur erwarteten Anzahl $\mathbf{E}(\boldsymbol{C_{n,k}})$ von Vergleichen in dieser Situation.

(Wahrscheinlichkeitsmodell: Zufällige Pivotwahl.)

Vernachlässige die Abhängigkeit von k:

$\boldsymbol{U_n} := \max\{\mathbf{E}(C_{n,k}) \mid 1 \leq k \leq n\}$.

Klar: $U_1 = 0$, $U_2 = 1$.

Behauptung: $U_n \leq c \cdot n$, für eine Konstante $c > 0$.

Beweis: Induktion über n.

(Trick: Wir wählen c **später** so, dass der Beweis funktioniert. Der Induktionsanfang ist mit der Bedingung $c \geq 1$ schon gesichert.)

Sei $n \geq 2$.

Das Pivotelement hat Rang $p \in \{1, \ldots, n\}$, jedes p mit Wahrscheinlichkeit $\frac{1}{n}$.

Das Partitionieren kostet $n - 1$ Vergleiche.

1. Fall: $p = k$. Fertig.

2. Fall: $k < p$. – Rekursion in $(b_1, \ldots, b_{p-1})$.

Erwartete Vergleichsanzahl ist $\leq U_{p-1} \leq c(p-1)$, nach I.V.

3. Fall: $k > p$. – Rekursion in $(c_{p+1}, \ldots, c_n)$.

Erwartete Vergleichsanzahl ist $\leq U_{n-p} \leq c(n-p)$, nach I.V.

Also . . .

$$\mathbf{E}(C_{n,k}) < n + \frac{1}{n} \sum_{k<p\leq n} c(p-1) + \frac{1}{n} \sum_{1\leq p<k} c(n-p)$$

$$< n + \frac{c}{n}\left((n-k)\cdot\frac{k+(n-1)}{2} + (k-1)\frac{(n-1)+(n-(k-1))}{2}\right).$$

Benutzt wurde die Formel für arithmetische Reihen:
$a + (a+d) + (a+2d) + \cdots + (a+(r-1)d) = r \cdot \frac{1}{2}(a + (a+(r-1)d))$.
(Anzahl Summanden mal Mittelwert des ersten und letzten Summanden!)

$$\mathbf{E}(C_{n,k}) < n + \frac{c((n-k)(k+n-1) + (k-1)(2n-k))}{2n}.$$

Ausmultiplizieren:

$$\mathbf{E}(C_{n,k}) \leq n + \frac{c(n^2 - 3n - 2k^2 + 2k + 2nk)}{2n}.$$

Wähle k so, dass rechte Seite maximal wird, nämlich $k = \frac{n+1}{2}$:

$$U_n \leq n + \frac{c \cdot (\frac{3}{2}n^2 - 2n + \frac{1}{2})}{2n} < (1 + (3c/4))n.$$

Ohne c zu kennen, geht es nicht weiter.

Wähle $c = 4$, verstärke Ind.-Beh. zu: $U_n \leq 4n$.

Trivial für $n = 1$ und $n = 2$.

Induktionsschritt: $U_n \overset{\text{I.V.}}{\leq} (1 + (3c/4))n = (1 + 3)n = 4n = cn$. Fertig! □

Mitteilungen:

(a) Eine genauere Analyse ergibt für $\alpha = k/n$ konstant eine erwartete Vergleichsanzahl von $2(1 + H(\alpha)\ln 2)n < (3{,}3863 + o(1)) \cdot n$.

Dabei ist $H(\alpha) = -\alpha \log \alpha - (1-\alpha)\log(1-\alpha)$ die **„binäre Entropie“** der Wahrscheinlichkeitsverteilung $(\alpha, 1-\alpha)$.
$H(\alpha)$ liegt zwischen 0 und 1; das Maximum 1 ist bei $\alpha = \frac{1}{2}$, was der Suche nach dem Median entspricht.
$2(1 + \ln 2) \approx 3{,}386294.$

(b) Quickselect ist in der Praxis sehr, sehr schnell.
Achtung! Falls es identische Einträge gibt, muss man entsprechende Vorkehrungen treffen (3-Wege-Partition, s. Kapitel 6, Folie 42).

(c) Die beste theoretische Schranke für die erwartete Vergleichsanzahl bei Algorithmen für das Auswahlproblem, nämlich $\frac{3}{2}n + o(n)$, erreicht ein anderer randomisierter Algorithmus. („Random Sampling“, siehe Vorlesung „Randomisierte Algorithmen“.)

Nun: Ein **deterministischer** Algorithmus mit Aufwand $O(n)$.

(Erfinder: M. **Blum**, R. W. **Floyd**, V. R. **Pratt**, R. L. **Rivest**, R. E. **Tarjan**: lauter Pioniere der Algorithmik!)

„Stufenweises“ Divide-and-Conquer. – Wie bei Quickselect:

- Finde ein partitionierendes Element x.
- Verschiebe Einträge im Array, so dass alle Elemente $< x$ links von x stehen, alle Elemente $> x$ rechts.
- Lese ab, in welchem Teil das Element vom Rang k sitzt.
- Rufe den Algorithmus rekursiv auf diesem Teil auf.

Zentral: **Deterministisch**, unter Benutzung von Rekursion ein „günstiges“ Element x best

Algorithmus BFPRT$(a_1, \ldots, a_n, k)$ // O.B.d.A.: $a_1, \ldots, a_n$ verschieden

(0) Falls $k = 1$ oder $k = n$: Bestimme Minimum/Maximum direkt. **Sonst**:
(1) Falls $n \leq n_0$: Sortiere mit Mergesort, fertig. **Sonst**:
(2) Teile $(a_1, \ldots, a_n)$ in $m = \lceil n/5 \rceil$ Gruppen mit 4 bzw. 5 Elementen auf.
(3) Bestimme in jeder Gruppe den Median (z. B. mit Mergesort).
Sei $(a_1^*, \ldots, a_m^*)$ die Liste dieser Mediane.
(4) Suche mit **BFPRT rekursiv** den **Median** x von $(a_1^*, \ldots, a_m^*)$.
(5) Zerlege $(a_1, \ldots, a_n)$ in eine Teilfolge $b_1, \ldots, b_{p-1}$, alle $< x$, in das Element x, und eine Teilfolge $c_{p+1}, \ldots, c_n$, alle $> x$.
(6) Falls $k = p$: Rückgabe x.
(7) Falls $k < p$: **BFPRT**$(b_1, \ldots, b_{p-1}, k)$. // **Rekursion**
(8) Falls $k > p$: **BFPRT**$(c_{p+1}, \ldots, c_n, k - p)$ // **Rekursion**

Das Pivotelement x bezeichnet man als „*Median der Mediane*“.
Dieses partitionierende Element wird in den Schritten (2)–(4) gefunden.

Korrektheit: Klar, durch Induktion über rekursive Aufrufe.

Laufzeit: Die Laufzeit ist proportional zur Anzahl der durchgeführten Vergleiche.

Wir definieren: $\boldsymbol{C(n)}$:= maximale Anzahl der Vergleiche bei Aufruf **BFPRT**$(a_1, \ldots, a_\ell, k)$, $\ell \leq n$, $1 \leq k \leq \ell$.

(Durch Maximieren über „$\ell \leq n$“ wird die Funktion $C(n)$ monoton.)

Zeile (0) (Fall $k \in \{1, n\}$) benötigt $n - 1$ Vergleiche, und wir sind fertig.

Zeile (1) (Fall $n \leq n_0$) benötigt höchstens $n \log n$ Vergleiche.

Zeile (3): Median von 5 Einträgen: mit Mergesort in 8 Vergleichen zu ermitteln.

Direktes Verfahren (Übung) findet Median mit 6 Vergleichen.

Für 4 Einträge genügen 4 Vergleiche.

Zeile (3) benötigt daher $\leq 6n_5$ Vergleiche für die n_5 Fünfergruppen und $4n_4$ Vergleiche für die $n_4 \leq 4$ Vierergruppen.

Da $n = 5n_5 + 4n_4$, ist dies zusammen $\leq 6n/5$.

Zeile (4): Höchstens $C(\lceil n/5 \rceil)$ Vergleiche.

Zeile (5): Exakt $n - 1$ Vergleiche, ebenso wie bei Quickselect.

Zeilen (7)/(8): Es wird nur eine dieser beiden Zeilen ausgeführt. Wir zeigen, dass in beiden Fällen die Anzahl der beteiligten Einträge nicht größer als $7n/10 + 4$ ist.

1. Fall: $p > k$. Bei der rekursiven Suche nach dem Eintrag mit Rang k werden alle Einträge $\geq x$ weggelassen.

Wie viele Einträge sind dies **mindestens**?

Mit $G_1, \ldots, G_m$ bezeichnen wir die Gruppen (4 oder 5 Elemente).
a_j^* ist der Median von G_j. Dann definieren wir:

$$A := \{a_i \mid \exists j : a_i \text{ in Gruppe } G_j \text{ und } a_j^* \geq x \text{ und } a_i \geq a_j^*\}.$$

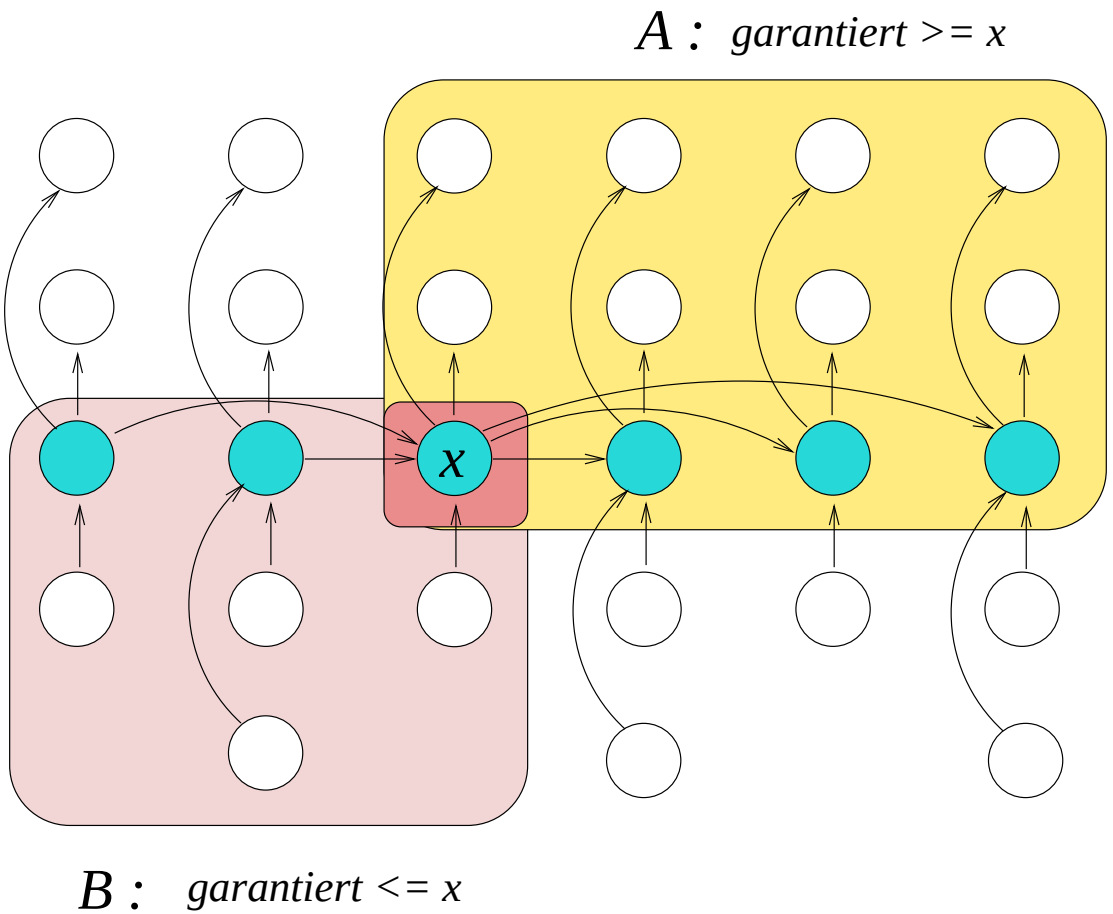
Dann sind alle Elemente von A mindestens so groß wie x (s. Bild).

Also $n - p + 1 \geq |A| \geq 3(\lfloor m/2 \rfloor + 1) \geq 3n/10$, also $p - 1 \leq 7n/10$.

Die Kosten für den rekursiven Aufruf in Zeile (7) sind also maximal $C(\lfloor 7n/10 \rfloor)$.

Spalten: Gruppen nach Umsortieren, Kriterium: a_j^* $</=/>x$

Innerhalb jeder Spalte: unten kleiner als der Median, oben größer



2. Fall: $p < k$. Bei der rekursiven Suche nach dem Eintrag mit Rang k werden alle Einträge $\leq x$ weggelassen.

Definiere

$$B := \{a_i \mid \exists j : a_i \text{ in Gruppe } G_j \text{ und } a_j^* \leq x \text{ und } a_i \leq a_j^*\}.$$

Dann sind alle Elemente von B höchstens so groß wie x (s. Bild).

Weil es mindestens $\lceil m/2 \rceil$ Gruppen G_j mit $a_j^* \leq x$ und maximal 4 Gruppen mit 4 Elementen gibt, folgt:

$p \geq |B| \geq 3\lceil m/2 \rceil - 4 \geq 3n/10 - 4$,

also betrifft der rekursive Aufruf in Zeile (8) höchstens $n - p \leq 7n/10 + 4$ Einträge.

Die Kosten sind also maximal $C(\lfloor 7n/10 \rfloor + 4)$.

Wir erhalten die folgende Rekurrenzungleichung:

$$C(n) \leq \begin{cases} n \log n & \text{für } n \leq n_0, \\ C(\lceil n/5 \rceil) + C(\lfloor 7n/10 \rfloor + 4) + 11n/5 & \text{für } n > n_0. \end{cases}$$

Dabei schätzt der Term $11n/5$ die Beiträge von Zeilen (3) und (5) zusammen ab.

Leider: Unser Mastertheorem nicht anwendbar.

Wir lösen die Rekurrenz direkt, indem wir eine passende Induktionsbehauptung beweisen.

Behauptung:
$C(n) \leq cn$ für alle n und eine passende Konstante c.

Die $n \leq n_0$ werden erledigt, indem man $c \geq \log n_0$ wählt.

Konkret: $n_0 = 500$; jedes $c \geq 9$ erfüllt die Behauptung in diesem Fall.

Nun sei $n > n_0$. Wir rechnen:

$$\begin{aligned} C(n) &\leq C(\lceil n/5 \rceil) + C(\lfloor 7n/10 \rfloor + 4) + 11n/5, \\ &\overset{\text{I.V.}}{\leq} c\lceil n/5 \rceil + c(7n/10 + 4) + 11n/5 \\ &\leq cn/5 + c + 7cn/10 + 4c + 11n/5 \\ &\leq cn + (-cn/10 + 5c + 11n/5). \end{aligned}$$

Entscheidend: $C(n) \leq \frac{9}{10}cn + O(n)$.

Wir wählen c so, dass $cn/10 \geq 5c + 11n/5$ ist, was für $c \geq 25$ und $n \geq n_0 = 500$ der Fall ist (nachrechnen!).

Für ein solches c lässt sich der Induktionsschritt durchführen; damit gilt die Behauptung $C(n) \leq cn$ für alle n.

Wir haben gezeigt:

Satz 9.5.2

Der BFPRT-Algorithmus löst das Auswahlproblem und hat eine Laufzeit von $O(n)$ im schlechtesten Fall.

Bemerkung: (a) Durch eine viel genauere Analyse kann die Konstante in der Vergleichsanzahl noch verbessert werden.
(b) Der beste bekannte deterministische Algorithmus für das Auswahlproblem (anderer Ansatz!) benötigt $(2{,}95 + o(1))n$ Vergleiche. Es ist bekannt, dass jeder deterministische Algorithmus $\geq 2n$ Vergleiche benötigt.

9.6 Die schnelle Fourier-Transformation (FFT)

Aufgabe: Multiplikation von **Polynomen**, z. B.

$(5x^4 - 7x^3 - 4x + 2) \cdot (3x^2 + x + 1) = \ldots$

Polynom in Koeffizientendarstellung:

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1} = \sum_{0 \leq i \leq n-1} a_i x^i,$$

mit **Koeffizienten** $a_0, \ldots, a_{n-1} \in$ „Zahlenbereich“ $\mathbb{Z}$ oder $\mathbb{R}$ oder $\mathbb{C}$.

Für die Durchführung benötigen wir **komplexe Zahlen**. (Siehe Mathematikvorlesung.)

Am Schluss wird eine Alternative skizziert, die man für Berechnungen über $\mathbb{Z}$ benutzen kann, ohne dabei zu komplexen Zahlen überzugehen.

Aufgabe: Gegeben zwei Polynome $A(x)$ und $B(x)$, als

$$A(x) = \sum_{0 \le i \le d-1} a_i x^i \quad \text{und} \quad B(x) = \sum_{0 \le j \le d-1} b_j x^j,$$

berechne das **Polynomprodukt**

$$A(x) \cdot B(x) = C(x) = \sum_{0 \le k \le 2d-2} c_k x^k,$$

d. h. berechne die Koeffizienten $c_k = \sum_{i,j \,:\, i+j=k} a_i b_j$, für $0 \le k \le 2d-2$.

Beispiel: $(a_0 + a_1 x + a_2 x^2) \cdot (b_0 + b_1 x + b_2 x^2) =$
$\underline{a_0 b_0} + (\underline{a_0 b_1 + a_1 b_0}) x + (\underline{a_0 b_2 + a_1 b_1 + a_2 b_0}) x^2 + (\underline{a_1 b_2 + a_2 b_1}) x^3 + \underline{a_2 b_2} x^4.$

Die Folge $(c_0, \ldots, c_{2d-2})$ heißt auch **Konvolution** $(a_0, \ldots, a_{d-1}) \circ (b_0, \ldots, b_{d-1})$.

Naive Benutzung der Formel für die c_k liefert $\Theta(d^2)$-Algorithmus.

Ziel: $O(d \log d)$. – Methode: Divide-and-Conquer.

Zentraler Trick: Benutze eine weitere Darstellung von Polynomen, nämlich die

Stützstellen-Darstellung.

Betrachte Folge $(x_0, \ldots, x_{n-1}) \in \mathbb{C}^n$ von beliebigen **verschiedenen** „Stützstellen“, für $n \geq 1$.

Zu einem Polynom $A(x) = \sum_{0 \leq i \leq n-1} a_i x^i$ betrachten wir den zugehörigen „**Wertevektor**“ $(A(x_0), \ldots, A(x_{n-1}))$. – Umkehrung:

Fakt 9.6.1 („**Interpolation**“ von Polynomen über Körpern)

Zu jedem beliebigen Wertevektor $(r_0, \ldots, r_{n-1}) \in \mathbb{C}^n$ **gibt es genau ein** Polynom $A(x) = \sum_{0 \leq i \leq n-1} a_i x^i$, also genau einen Koeffizientenvektor $(a_0, \ldots, a_{n-1})$, mit $A(x_k) = r_k$ für $0 \leq k \leq n-1$.

Es ist also gleichgültig, ob man für die Darstellung eines Polynoms (vom Grad $< n$) seine Koeffizienten oder einen Wertevektor benutzt.

Beweis von Fakt 9.6.1: Betrachte die „**Vandermonde-Matrix**“

$$V(x_0, \ldots, x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}.$$

Offensichtlich ist

$$\begin{pmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{pmatrix} = V(x_0, \ldots, x_{n-1}) \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

Die Matrix $V(x_0, \ldots, x_{n-1})$ hat Determinante $\prod_{0 \le k < \ell \le n-1} (x_\ell - x_k) \neq 0$, ist also regulär. Daher hat das Gleichungssystem

$$V(x_0, \ldots, x_{n-1}) \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{n-1} \end{pmatrix}$$

genau eine Lösung

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = V(x_0, \ldots, x_{n-1})^{-1} \cdot \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{n-1} \end{pmatrix}.$$

Bemerkung: Aus der Darstellung der n-fachen Polynomauswertung als Matrix-Vektor-Produkt folgt auch:

Die Umrechnung von $(a_0, \ldots, a_{n-1})$ in $(r_0, \ldots, r_{n-1})$ ist eine (bijektive) **lineare Abbildung** von $\mathbb{C}^n$ nach $\mathbb{C}^n$.

Zurück zu Polynommultiplikation: Um durch Interpolation die $2d-1$ Koeffizienten des Produktpolynoms

$$A(x) \cdot B(x) = C(x) = \sum_{0 \leq k \leq 2d-2} c_k x^k$$

zu erhalten, müssen wir mit mindestens $2d-1$ Argument-Werte-Paaren arbeiten. Aus technischen Gründen verwenden wir $n = 2^{\lceil \log(2d) \rceil} \geq 2d$ viele.

Algorithmenplan für die Polynommultiplikation:

Eingabe: Zwei Polynome $A(x)$ und $B(x)$ vom Grad $< d$ als Koeffizientenvektoren $(a_0, \ldots, a_{d-1})$ und $(b_0, \ldots, b_{d-1})$.

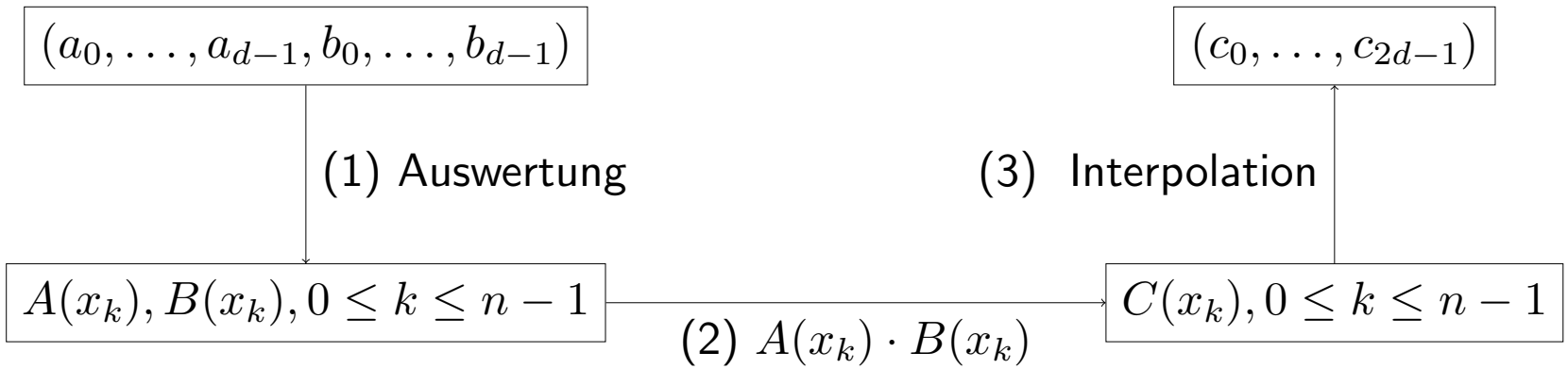
Setze $n = 2^{\lceil \log(2d) \rceil} \geq 2d$, Zweierpotenz, $2d \leq n < 4d$. Also: $O(n) = O(d)$.

$x_0, x_1, \ldots, x_{n-1}$ seien verschiedene Argumentwerte.

(1) Auswertung:
Berechne $A(x_k)$ und $B(x_k)$, für $k = 0, \ldots, n-1$.

(2) Berechne durch **punktweise Multiplikation** die n Werte des Produktpolynoms $C(x)$ an den Stützstellen:
$C(x_k) := A(x_k) \cdot B(x_k)$, für $k = 0, \ldots, n-1$.

(3) Interpolation:
Berechne aus $(C(x_0), C(x_1), \ldots, C(x_{n-1}))$ die Koeffizienten $(c_0, \ldots, c_{2d-1})$ von $C(x)$.



Kosten:

(1) **??** Naiv: Jeden Wert $A(x_k)$ separat berechnen, z. B. mit dem Horner-Schema: $A(x_k) = ((\ldots(a_{n-1} \cdot x_k + a_{n-2}) \cdot x_k \ldots) \cdot x_k + a_1) \cdot x_k + a_0$. Kosten: $O(d^2)$.

(2) Kosten: $O(d)$.

(3) **??** (Auch hier: $O(d^2)$ recht leicht zu erreichen.)

Unser Ziel: $O(d \log d)$ für (1) und (3).

Zunächst: **Auswertung**, für Grad $< n$ und $\boldsymbol{n = 2^L}$ Stützstellen.

Input: $A(x)$ als $(a_0, \ldots, a_{n-1})$, Stützstellen $(x_0, \ldots, x_{n-1})$.

Output: $(r_0, \ldots, r_{n-1}) = (A(x_0), \ldots, A(x_{n-1}))$.

Ansatz: **Divide-and-Conquer**.

Wenn $n = 1$, ist das Ergebnis $(r_0) = (a_0)$.

Wenn $n > 1$, teilen wir $A(x)$ in zwei Teilpolynome auf:

$$A(x) = (a_0 + a_2x^2 + a_4x^4 + \cdots + a_{n-2}x^{n-2}) + x(a_1 + a_3x^2 + a_5x^4 + \cdots + a_{n-1}x^{n-2}),$$

mit den Abkürzungen („**g**erade“, “**u**ngerade“)

$A_{\mathsf{g}}(x) := a_0 + a_2x + \cdots + a_{n-2}x^{n/2-1}$ und $A_{\mathsf{u}}(x) := a_1 + a_3x + \cdots + a_{n-1}x^{n/2-1}$
also

$$A(x) = A_{\mathsf{g}}(x^2) + xA_{\mathsf{u}}(x^2). \tag{1}$$

$$A(x) = A_{\mathsf{g}}(x^2) + xA_{\mathsf{u}}(x^2).$$

$A_{\mathsf{g}}(x^2)$ und $A_{\mathsf{u}}(x^2)$ haben jeweils nur noch $n/2$ Koeffizienten.
(Inputgröße halbiert!)

Originalproblem: Werte das Polynom $A(x)$ an den Stützstellen $x_0, \ldots, x_{n-1}$ aus.

Müssen nun rekursiv $A_{\mathsf{g}}(x_j^2)$ und $A_{\mathsf{u}}(x_j^2)$ berechnen, für $0 \leq j < n$.

Das passt aber nicht mit der Rekursion zusammen.
(Ein Polynom mit $n/2$ Koeffizienten muss an $n/2$ Stellen ausgewertet werden.)

Zentraler Trick: Sorge dafür, dass

$$x_0^2, \ldots, x_{n-1}^2$$

nur $n/2$ verschiedene Stützstellen sind.

Wie kann das funktionieren?

Wähle $x_0, \ldots, x_{n/2-1}$ verschieden und $\neq 0$ und betrachte als Stützstellen

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1},$$

mit folgende Anordnung im Stützstellenvektor:

$$(x_0, \ldots, x_{n/2-1}, -x_0, \ldots, -x_{n/2-1}).$$

Vorerst nehmen wir an, dass die Stützstellen auf jeder Ebene der Rekursion diese Eigenschaft erfüllen.

Auswertung des Polynoms $A(x) = \sum_{0 \le j < n} a_j x^j$ auf Stützstellen $(x_0, \ldots, x_{n-1})$:

Teile: Stelle Koeffizientenvektoren $(a_0, a_2, \ldots, a_{n-2})$ von $A_{\mathsf{g}}(x)$ und $(a_1, a_3, \ldots, a_{n-1})$ von $A_{\mathsf{u}}(x)$ zusammen
und finde $(x'_0, x'_1, \ldots, x'_{n/2-1}) = (x_0^2, x_1^2, \ldots, x_{n/2-1}^2)$.

Rekursion: Werte $A_{\mathsf{g}}(x)$ mit $(x'_0, x'_1, \ldots, x'_{n/2-1})$ und $A_{\mathsf{u}}(x)$ mit $(x'_0, x'_1, \ldots, x'_{n/2-1})$ rekursiv aus. Resultat: Zwei Wertevektoren

$$A_{\mathsf{g}}(x_0^2), A_{\mathsf{g}}(x_1^2), \ldots, A_{\mathsf{g}}(x_{n/2-1}^2) \quad \text{und}$$

$$A_{\mathsf{u}}(x_0^2), A_{\mathsf{u}}(x_1^2), \ldots, A_{\mathsf{u}}(x_{n/2-1}^2).$$

Kombiniere: Berechne nun:

$$A(x_j) = A_{\mathsf{g}}(x_j^2) + x_j \cdot A_{\mathsf{u}}(x_j^2), \text{ für } 0 \le j < n/2;$$

$$A(x_{n/2+j}) = A_{\mathsf{g}}(x_j^2) - x_j \cdot A_{\mathsf{u}}(x_j^2), \text{ für } 0 \le j < n/2.$$

Beobachte: $n/2$ Multiplikationen und n Additionen im Kombinierschritt.

Unter der Annahme, dass auf jeder Rekursionsebene geeignete Stützstellen gefunden werden, schätzen wir die Laufzeit ab:

$C(n)$: Rechenzeit bei Eingaben der Länge n, vernachlässige konstante Faktoren.

Rekurrenzungleichung:

$$C(n) \leq \begin{cases} 1 & \text{, falls } n = 1 \\ 2 \cdot C(n/2) + cn & \text{, sonst,} \end{cases}$$

für eine Konstante c. Mit dem Master-Theorem, 2. Fall, ergibt sich

$$C(n) = O(n \log n).$$

Problem: Auf oberster Rekursionsstufe: $\pm x_0, \ldots, \pm x_{n/2-1}$ als „Plus-Minus“-Paare gewählt.

Im rekursiven Aufruf: $x_0^2, \ldots, x_{n/2-1}^2$ müssen wieder in „Plus-Minus“-Paare aufgeteilt werden.

Wenn die Stützstellen reell sind, kann dies nicht funktionieren: Quadrate sind nie negativ.

Trick: Nutze komplexe Zahlen!

ω sei **primitive n-te Einheitswurzel** in $\mathbb{C}$, d. h.

(i) $\omega^n = 1$,

(ii) für $1 \leq k \leq n-1$ gilt $\sum_{0 \leq j \leq n-1} (\omega^k)^j = 0$.

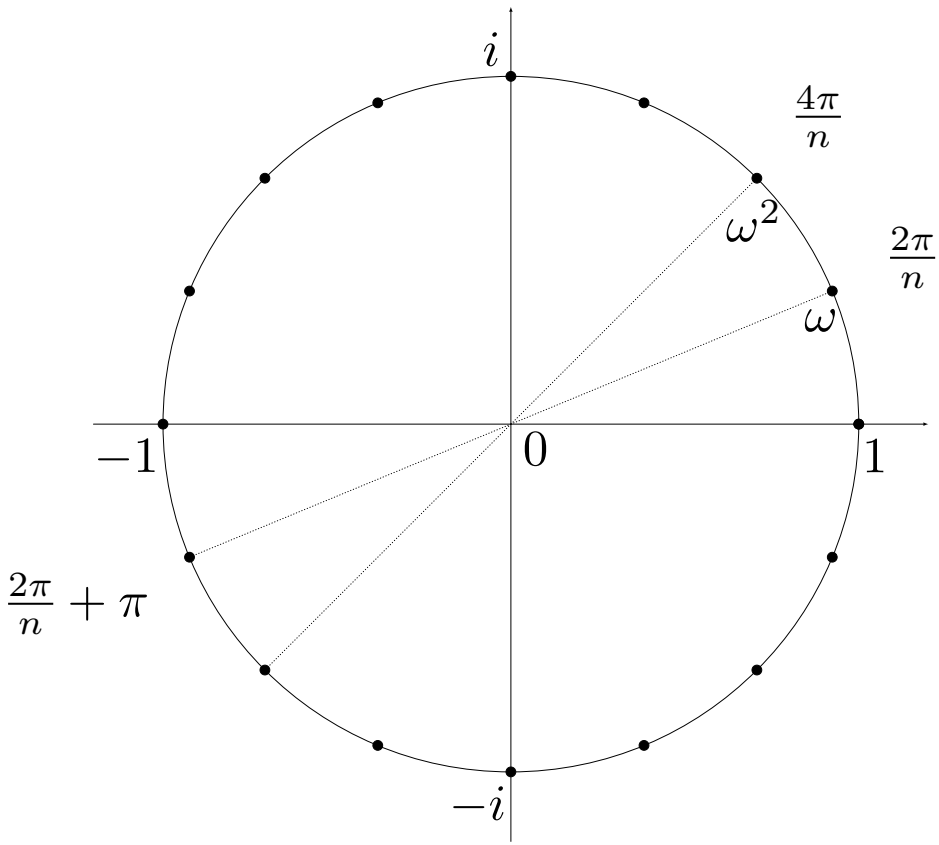
(iii) $\omega^k \neq 1$, für $1 \leq k \leq n-1$.

Unter einfachen weiteren Voraussetzungen sind (ii) und (iii) äquivalent:

– Wenn $\omega^k = 1$ ist, dann folgt $\sum_{0 \leq j \leq n-1} (\omega^k)^j = n$. Daher folgt (iii) aus (ii) (und $n \neq 0$).

– Wenn $\omega^k \neq 1$ gilt, schreiben wir: $(\omega^k - 1)(\sum_{0 \leq j \leq n-1} (\omega^k)^j) = (\omega^k)^n - 1 = 0$.

Daher folgt (ii), wenn (iii) gilt und zudem die Inversen $(\omega^k - 1)^{-1}$ existieren.



In $\mathbb{C}$: Eine primitive n-te Einheitswurzel ist

$$\omega := e^{\frac{2\pi i}{n}} = \cos\left(\frac{2\pi}{n}\right) + i \sin\left(\frac{2\pi}{n}\right),$$

wobei hier i die imaginäre Einheit ist. In der komplexen Zahlenebene liegt der Punkt ω also auf dem Einheitskreis, von 1 aus um den Winkel $2\pi/n$ gegen den Uhrzeigersinn verdreht.
Die Potenzen ω^k, $0 \leq k \leq n-1$, liegen in gleichen Abständen auf dem Einheitskreis.

Wieso ist ω primitive n-te Einheitswurzel?

(i) $\omega^n = (e^{\frac{2\pi i}{n}})^n = e^{2\pi i} = 1$.

(ii) Für jedes beliebige y gilt

$$\sum_{0 \le j \le n-1} y^j = (1+y)(1+y^2)(1+y^4)\cdots(1+y^{2^{L-1}}). \tag{2}$$

(Ausmultiplizieren des Produkts ergibt $n = 2^L$ Summanden y^j, bei denen jeder Exponent $j \in \{0, \ldots, 2^L - 1\}$ genau einmal vorkommt, wegen der Eindeutigkeit der Binärdarstellung.)

Wir betrachten (2) für $y = \omega^k$, für $1 \le k \le n-1$ beliebig.

Schreibe $k = u \cdot 2^\ell$, mit u ungerade und $0 \le \ell < L$. Dann ist

$$(\omega^k)^{2^{L-\ell-1}} = \omega^{u(2^{L-1})} = (\omega^{n/2})^u = (-1)^u = -1,$$

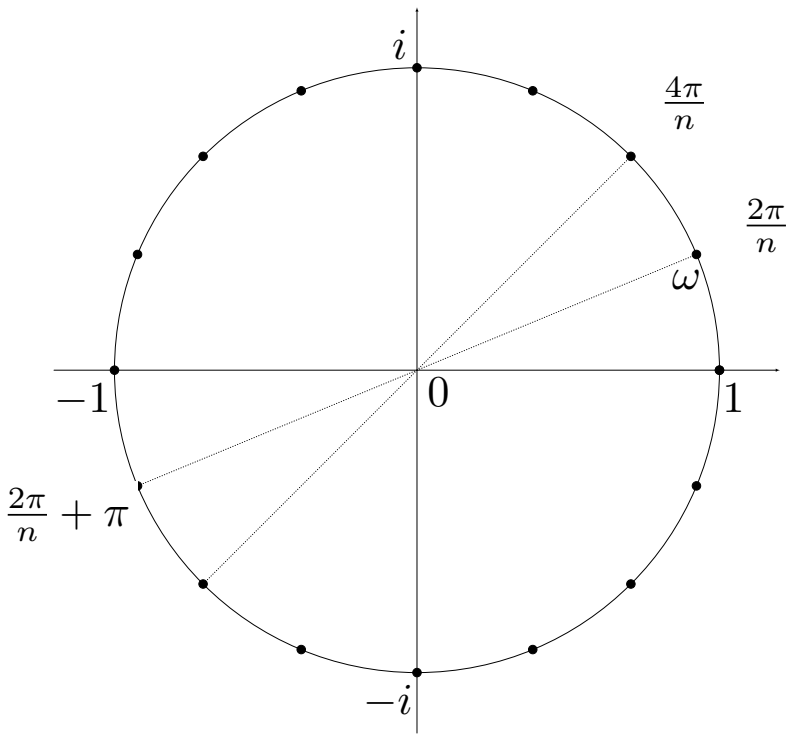
weil u ungerade ist. Also ist der Faktor $(1 + (\omega^k)^{2^{L-\ell-1}})$ in (2) gleich 0, also ist $\sum_{0 \le j \le n-1} (\omega^k)^j = 0$.

Sei $n = 2^L$ und sei ω eine beliebige primitive n-te Einheitswurzel. Dann ist unser Problem mit den $\pm$-Paaren gelöst:

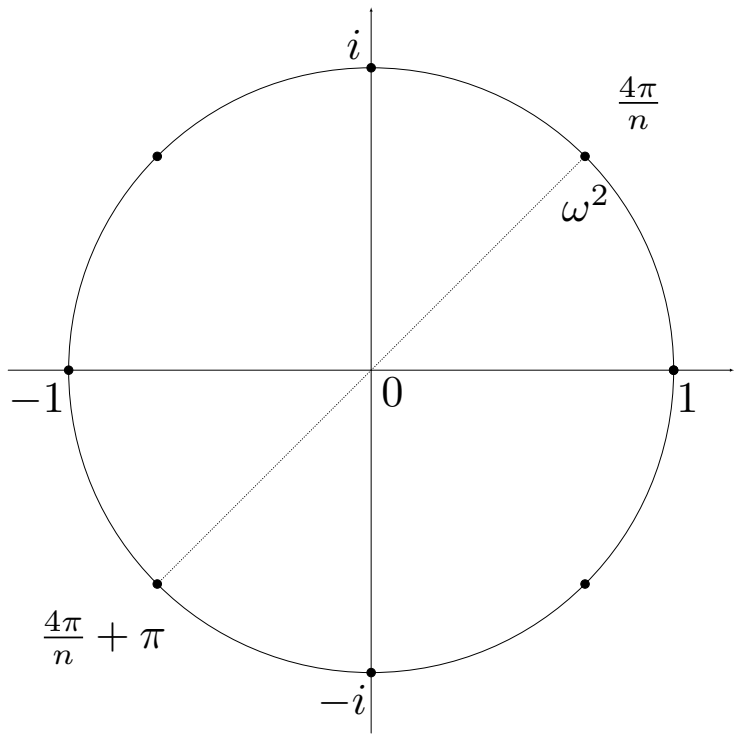
- $\omega^0 = 1, \omega^{n/2} = -1$.
- Für $j \in \{0, \ldots, n/2 - 1\}$ gilt: $\omega^j = -\omega^{n/2+j}$. Wir finden also $\pm$-Paare!
- ω^2 ist selbst eine primitive Einheitswurzel für $n/2$.
- Die Potenzen $(\omega^0)^2, \ldots, (\omega^{n/2-1})^2$ sind $(\omega^2)^0, \ldots, (\omega^2)^{n/2-1}$.
  Im Rekursionsschritt werden wir also wieder $\pm$-Paare finden.

Beispiel: $n = 16$:

16 Stützstellen.



Rekursion: 8 Stützstellen.



Als $(x_0, x_1, \ldots, x_{n-1})$ wählen wir $(\omega^0, \omega^1, \ldots, \omega^{n-1})$.

(Beachte: $\omega^0 = 1$, $\omega^1 = \omega$.)

Gegeben $A(x) = \sum_{0 \leq i \leq n-1} a_i x^i$ als Koeffizientenvektor $(a_0, a_1, \ldots, a_{n-1})$, wollen wir dann die „**diskrete Fourier-Transformierte**“

$(r_0, r_1, \ldots, r_{n-1}) = (A(1), A(\omega), A(\omega^2), \ldots, A(\omega^{n-1}))$

von $(a_0, \ldots, a_{n-1})$ berechnen.

Die Operation

$$\boxed{(a_0, \ldots, a_{n-1}) \mapsto (A(1), A(\omega), A(\omega^2), \ldots, A(\omega^{n-1}))}$$

heißt die **diskrete Fourier-Transformation**.

Es handelt sich dabei um eine bijektive lineare Abbildung von $\mathbb{C}^n$ nach $\mathbb{C}^n$.

(Es gibt auch eine Fourier-Transformation für Funktionen, die auf Integralen beruht.)

Wir berechnen zunächst **rekursiv** die n Werte

$$(s_0, \ldots, s_{n/2-1}) = (A_{\mathsf{g}}((\omega^2)^0), A_{\mathsf{g}}((\omega^2)^1), \ldots, A_{\mathsf{g}}((\omega^2)^{n/2-1}))$$

und

$$(t_0, \ldots, t_{n/2-1}) = (A_{\mathsf{u}}((\omega^2)^0), A_{\mathsf{u}}((\omega^2)^1), \ldots, A_{\mathsf{u}}((\omega^2)^{n/2-1})).$$

Wie schon festgestellt, ist ω^2 eine primitive $(n/2)$-te Einheitswurzel, so dass wir tatsächlich rekursiv vorgehen können.

Wie sollen wir jetzt $(r_0, r_1, \ldots, r_{n-1})$ berechnen?

Für $j = 0, \ldots, n/2 - 1$, mit (1):

$$r_j = A(\omega^j) = A_{\mathsf{g}}((\omega^j)^2) + \omega^j \cdot A_{\mathsf{u}}((\omega^j)^2) = s_j + \omega^j \cdot t_j.$$

(Jetzt kommt der Clou, die Anwendung des Plus-Minus-Tricks!)

Für $j = 0, \ldots, n/2 - 1$ gilt $\omega^{n/2+j} = \underbrace{\omega^{n/2}}_{=-1} \cdot \omega^j = -\omega^j$.

Also:

$$(\omega^{n/2+j})^2 = (-\omega^j)^2 = (\omega^j)^2.$$

Daher:

$$\begin{aligned} r_{n/2+j} &= A(\omega^{n/2+j}) = A_{\mathsf{g}}((\omega^{n/2+j})^2) + \omega^{n/2+j} \cdot A_{\mathsf{u}}((\omega^{n/2+j})^2) \\ &= A_{\mathsf{g}}((\omega^2)^j) - \omega^j \cdot A_{\mathsf{u}}((\omega^2)^j) \\ &= s_j - \omega^j \cdot t_j. \end{aligned}$$

Wir können die Ergebnisse der rekursiven Aufrufe also ein zweites Mal benutzen, um $(r_{n/2}, \ldots, r_{n-1})$ zu berechnen!

Algorithmus FFT (Schnelle Fourier-Transformation)

Eingabe: (Koeffizienten-)Vektor $(a_0, \ldots, a_{n-1})$, für Zweierpotenz n;
primitive n-te Einheitswurzel ω.

if $n = 1$ **then return** (a_0);

$(s_0, \ldots, s_{n/2-1}) \leftarrow \mathsf{FFT}((a_0, a_2, \ldots, a_{n-2}), \omega^2)$;

$(t_0, \ldots, t_{n/2-1}) \leftarrow \mathsf{FFT}((a_1, a_3, \ldots, a_{n-1}), \omega^2)$;

for j **from** 0 **to** $n/2 - 1$ **do**

$r_j \leftarrow s_j + \omega^j \cdot t_j$;

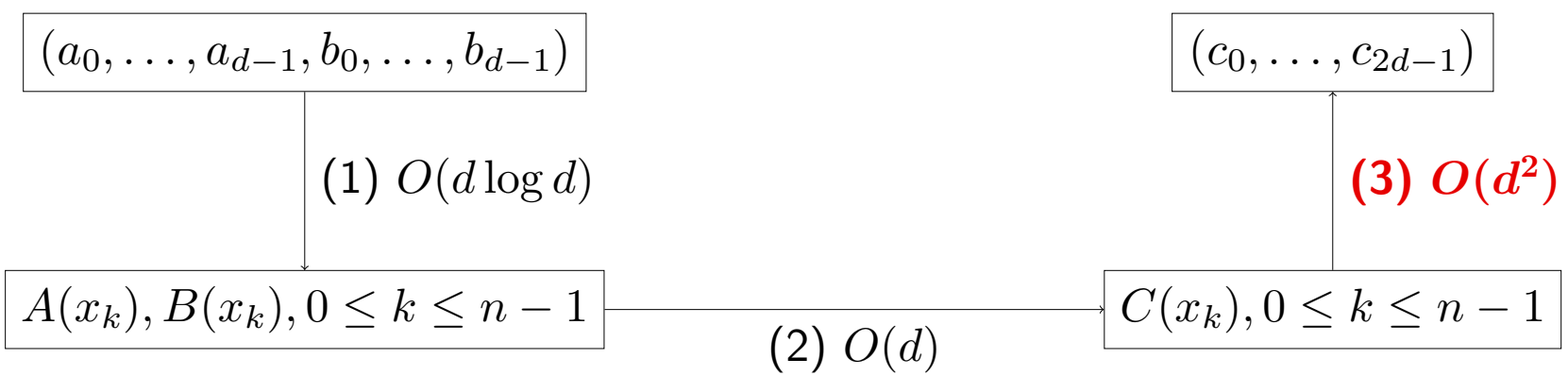
$r_{n/2+j} \leftarrow s_j - \omega^j \cdot t_j$;

return $(r_0, r_1, \ldots, r_{n-1})$.

Satz 9.6.2

Algorithmus FFT berechnet die diskrete Fouriertransformierte eines Koeffizientenvektors im Bezug auf die Argumente $(1, \omega, \omega^2, \ldots, \omega^{n-1})$, wobei ω eine primitive n-te Einheitswurzel ist, in Zeit $O(n \log n)$.

Aktueller Stand bei der Polynommultiplikation $(4d > n \geq 2d)$:



Es fehlt noch: **(3) Interpolation**.

Es fehlt noch: **Interpolation**. Auch hier ein schöner **Trick**.

Gegeben ist $(r_0, \ldots, r_{n-1})$, gesucht der Koeffizientenvektor $(a_0, \ldots, a_{n-1})$, der $M(\omega) \cdot (a_0, \ldots, a_{n-1})^{\mathsf{T}} = (r_0, \ldots, r_{n-1})^{\mathsf{T}}$ erfüllt, für die Matrix $M(\omega)$ der diskreten Fourier-Transformation (die Vandermonde-Matrix von $(\omega^0, \omega, \ldots, \omega^{n-1})$):

$$
\begin{aligned}
M(\omega) &= ((\omega^i)^j)_{0 \le i,j \le n-1} \\
&= \begin{pmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\
1 & \omega^2 & (\omega^2)^2 & \cdots & (\omega^2)^{n-1} \\
\vdots & \vdots & \vdots & & \vdots \\
1 & \omega^{n-1} & (\omega^{n-1})^2 & \cdots & (\omega^{n-1})^{n-1}
\end{pmatrix} .
\end{aligned}
$$

Man kann die zu $M(\omega)$ inverse Matrix direkt bestimmen.

Setze $\hat{\omega} := \omega^{n-1}$.

Dann gilt $\omega \cdot \hat{\omega} = \omega^n = 1$, also ist $\hat{\omega} = \omega^{-1}$.

Betrachte

$$
\begin{aligned}
M(\hat{\omega}) &= ((\hat{\omega}^i)^j)_{0 \le i,j \le n-1} \\
&= \begin{pmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & \hat{\omega} & \hat{\omega}^2 & \cdots & \hat{\omega}^{n-1} \\
1 & \hat{\omega}^2 & (\hat{\omega}^2)^2 & \cdots & (\hat{\omega}^2)^{n-1} \\
\vdots & \vdots & \vdots & & \vdots \\
1 & \hat{\omega}^{n-1} & (\hat{\omega}^{n-1})^2 & \cdots & (\hat{\omega}^{n-1})^{n-1}
\end{pmatrix}
\end{aligned}
$$

Wir berechnen den Eintrag z_{ij} an Stelle (i,j) der Produktmatrix $M(\omega) \cdot M(\hat{\omega})$:

$$z_{ij} = \sum_{0 \le k \le n-1} (\omega^i)^k \cdot (\hat{\omega}^k)^j = \sum_{0 \le k \le n-1} (\omega^{i+(n-1)j})^k.$$

Es gibt zwei Fälle. Wenn $i = j$ gilt, ist $\omega^{i+(n-1)j} = \omega^n = 1$, und die Summe ist n.

Wenn $i > j$ gilt, ist $\omega^{i+(n-1)j} = \omega^\ell$ für $1 \le \ell = i - j \le n-1$.
Wenn $i < j$ gilt, ist $\omega^{i+(n-1)j} = \omega^\ell$ für $1 \le \ell = n - (j-i) \le n-1$.

Wegen Bedingung (ii):

$$z_{ij} = \sum_{0 \le k \le n-1} (\omega^\ell)^k = 0.$$

Also: $M(\omega) \cdot M(\hat{\omega}) = n \cdot I_n$, für die $n \times n$-Einheitsmatrix I_n.

Das heißt: $M(\omega)^{-1} = n^{-1} \cdot M(\hat{\omega})$.

Wenn also $(r_0, \ldots, r_{n-1})$ der für die Interpolation gegebene Wertevektor ist, so erhalten wir den Koeffizientenvektor als

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = n^{-1} \cdot M(\hat{\omega}) \cdot \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{n-1} \end{pmatrix}.$$

Für die Zahl $\hat{\omega}$ beobachten wir:

(i) $\hat{\omega}^n = (\omega^{n-1})^n = (\omega^n)^{n-1} = 1^{n-1} = 1$,

(ii) für $1 \leq k \leq n-1$ gilt $\hat{\omega}^k = (\omega^{n-1})^k = \omega^{n(k-1)}\omega^{n-k} = \omega^{n-k}$,
mit $1 \leq n-k \leq n-1$, also $\sum_{0 \leq j \leq n-1}(\hat{\omega}^k)^j = 0$ wegen (ii) für ω.

Das heißt: Auch $\hat{\omega}$ ist eine primitive n-te Einheitswurzel.

Multiplikation mit $M(\hat{\omega})$ entspricht der FFT-Operation mit $\hat{\omega}$ an Stelle von ω.

Wir erhalten für die Interpolation:

$$(a_0, \ldots, a_{n-1}) = n^{-1} \cdot \mathsf{FFT}((r_0, \ldots, r_{n-1}), \hat{\omega}).$$

Zeitaufwand: $O(n \log n)$, also $O(d \log d)$. Damit: Plan von Folie 68 vollständig!

Algorithmus FFT-PM (Polynommultiplikation)

Eingabe: (Koeffizienten-)Vektoren $(a_0, \ldots, a_{d-1})$, $(b_0, \ldots, b_{d-1})$,
Setze $n = 2^{\lceil \log(2d) \rceil}$ $(\geq 2\,d)$.

Berechne n-te Einheitswurzeln $\omega \leftarrow e^{2\pi i/n}$ und $\hat{\omega} \leftarrow \omega^{n-1}$.

$(r_0^A, \ldots, r_{n-1}^A) \leftarrow \mathsf{FFT}((a_0, \ldots, a_{d-1}, 0, \ldots, 0), \omega)$; // auf Länge n auffüllen

$(r_0^B, \ldots, r_{n-1}^B) \leftarrow \mathsf{FFT}((b_0, \ldots, b_{d-1}, 0, \ldots, 0), \omega)$; // auf Länge n auffüllen

for j **from** 0 **to** $n-1$ **do** $r_j^C \leftarrow r_j^A \cdot r_j^B$;

$(c_0, c_1, \ldots, c_{n-1}) \leftarrow \frac{1}{n} \cdot \mathsf{FFT}((r_0^C, \ldots, r_{n-1}^C), \hat{\omega})$;

return $(c_0, c_1, \ldots, c_{2d-2})$.

Satz 9.6.3

Algorithmus FFT-PM berechnet die Koeffizienten des Produktes zweier durch Koeffizientenvektoren gegebener Polynome vom Grad $< d$ in Zeit $O(d \log d)$.

Beweis: Siehe vorherige Überlegungen.
(Rechenaufwand: $O(2d \log(2d)) = O(d \log d)$.)

Anmerkung: Um Polynome vom Grad $k-1$ und vom Grad $\ell-1$ zu multiplizieren, genügt $n=2^L$ für $L=\lceil \log_2(k+\ell-1)\rceil$.

Für Zuhause: Es folgen weitere Anmerkungen zur Geschichte und Anwendung der FFT. **(Nicht prüfungsrelevant.)**

Zum Üben: Berechnen Sie $(x+1)\cdot(x^2+1)$ mittels FFT!

Der FFT-Ansatz ermöglicht es auch, die diskrete Fourier-Transformatierte eines Vektors sehr effizient parallel zu berechnen, entweder mit mehreren Prozessoren oder sogar in Hardware.
In Algorithmus FFT können die beiden rekursiven Aufrufe unabhängig voneinander parallel durchgeführt werden (in Hardware: zwei Kopien der gleichen Schaltung nötig). Die Berechnung des Resultates $(r_0,\ldots,r_{n-1})$ kann sogar für alle n Werte gleichzeitig erfolgen.
Eine Beispielschaltung findet man auf Seite 69 im Buch von S. Dasgupta, C. Papadimitriou, U. Vazirani, **Algorithms**, McGraw-Hill, 2007.

Bemerkung für $\mathbb{C}$-Vektorraum-Spezialist/inn/en:

Die DFT ist eine Koordinatentransformation für einen Basiswechsel im $\mathbb{C}$-Vektorraum P_n der Polynome vom Grad bis zu $n-1$.

Man betrachtet ein inneres Produkt in P_n:

$$\left\langle \sum_i a_i X^i, \sum_i b_i X^i \right\rangle = \sum_{0 \le i \le n-1} a_i \overline{b_i}.$$

Definiere $f_j(X)$ als das (eindeutig bestimmte) Polynom, das $f_j(\omega^k) = [j = k]$ erfüllt, für $0 \le k \le n-1$. Dann ist die „Standardbasis“ $B_n = (1, X, X^2, \ldots, X_{n-1})$ Orthonormalbasis von P_n und die „Fourierbasis“ $F_n = (f_0(X), f_1(X), \ldots, f_{n-1}(X))$ Orthogonalbasis von P_n.

Die Matrix $M(\omega)$ ist die Matrix für den Basiswechsel von B_n (Koeffizienten $(a_0, \ldots, a_{n-1})$ nach F_n (Koeffizienten $(A(\omega^0), \ldots, A(\omega^{n-1}))$), die Matrix $M(\omega)^{-1} = \frac{1}{n} M(\hat{\omega})$ ist für den umgekehrten Basiswechsel zuständig.

Wozu eigentlich Polynommultiplikation oder „Faltung“

$$c_k = \sum_{i,j\,:\,i+j=k} a_i b_j, \quad \text{für } 0 \leq k \leq 2n-1 \text{ ?}$$

Zentrales Hilfsmittel bei der „Digitalen Signalverarbeitung“.

Offizielle Publikation: [Cooley/Tukey 1965].

Form des Algorithmus 1805 von C. F. Gauß entworfen (und zur Berechnung von Asteroidenbahnen benutzt).

Erstmalig publiziert wurde eine Variante des Algorithmus von C. Runge (1903/05).

(Quelle hierfür: Wikipedia – kann stimmen, muss aber nicht.)

Wir stellen uns ein System vor, in dem regelmäßig Signale anfallen und (zu Ausgabesignalen) verarbeitet werden.
Abtastzeitpunkte $t_0, t_1, \ldots$ mit festem Abstand $\Delta = t_{i+1} - t_i$ liefern (reelle, komplexe) Signalwerte $a_0, a_1, a_2, \ldots$.
Ein Verarbeitungsmechanismus soll diese Messwertfolge in eine Ausgabefolge $c_0, c_1, c_2, \ldots$ umsetzen, mit Ausgabezeiten $t'_0, t'_1, t'_2, \ldots$, ebenso mit Abstand Δ.

Die einfachsten Mechanismen sind **linear** (wenn man zwei Signalfolgen addiert, addieren sich die Ausgabefolgen, ebenso bei Multiplikation mit konstanten Faktoren)
und **zeitinvariant** (wenn die gleiche Signalfolge um einen Zeitschritt Δ versetzt auftritt, ergibt sich die gleiche um Δ versetzte Ausgabefolge).
Leicht zu sehen: Bei linearen, zeitinvarianten Signalverarbeitungs-Systemen ist durch die Ausgabefolge $(b_0, b_1, b_2, \ldots)$, die von einem Signal der Größe 1 bei t_0 und sonst nur Nullsignalen ausgelöst wird, die Ausgabefolge $(c_0, c_1, c_2, \ldots)$ auf einer beliebigen Signalfolge $(a_0, a_1, \ldots)$ eindeutig bestimmt, durch:

$$c_k = \sum_{i+j=k} a_i b_j.$$

Das ist gerade die Folge der Koeffizienten des Produktpolynoms! D.h.: Lineare, zeitinvariante Reaktion auf Messsignale führt unmittelbar zum Problem „Polynommultiplikation“.

„Multiplikation ganzer Zahlen“:
Wir wollen zwei Zahlen x, y mit je N Bits multiplizieren, N ist extrem groß.
Wir zerlegen die Binärdarstellung von x und y in Blöcke der Länge $\sqrt{N}$. Indem wir jeweils jeden dritten Block nehmen, bilden wir aus x und y jeweils drei Zahlen, so dass

$$x = x_0 + x_1 + x_2 \text{ und } y = y_0 + y_1 + y_2$$

gilt und so dass $x_i \cdot y_j$ als Multiplikation von Zahlen mit $\sqrt{N}$ Ziffern im Bereich $[\sqrt{N}]$ aufgefasst werden kann. Durch die großen eingestreuten Blöcke von Nullen gibt es keine Überträge.
(Bild: Tafel/Handzeichnung.)
Die „Ziffern“ der Produkte $x_i \cdot y_j$ stellen sich als die Koeffizienten eines Produktpolynoms heraus.
Mit der FFT könnte man diese Ziffern der Teilprodukte in $O(\sqrt{N} \log N)$ Multiplikationen von komplexen Zahlen ermitteln.

Unter der Annahme, dass eine solche Multiplikation durch geeignete Rundung auf komplexe Zahlen mit Darstellungslänge $O(\sqrt{N})$ in Zeit $O(N)$ möglich ist, erhalten wir einen Gesamtzeitaufwand von $O(N^{3/2} \log N)$, sogar besser als der Zeitbedarf des Karatsuba-Multiplizierers.
Durch Einziehen von Rekursionsstufen lässt sich die Zeit weiter drücken; wenn man es konsequent durchführt, auf $O(N^{1+\varepsilon})$ für beliebige konstante $\varepsilon > 0$ (Verfahren von Toom/Cook, z. B. im Buch „The Art of Computer Programming, Vol. 2: Seminumerical Algorithms“ von D. Knuth beschrieben).
Für den Multiplikationsalgorithmus von Schönhage und Strassen mit Kosten $O(N \log N \log \log N)$ sind neben der FFT noch weitere Ideen nötig.

Für die Arbeit mit ganzen Zahlen ist die oben beschriebene FFT sehr unbequem, weil man mit komplexen Zahlen rechnen muss.
In gewissen Situationen kann man FFT auch „modulo m“ durchführen, für geeignete „Moduli“ m, und ganz im diskreten Bereich bleiben.

Fakt 9.6.4
Wenn $w = 2$ und n eine Zweierpotenz ist, dann gilt für $m = 2^{n/2} + 1$:

(i) $w^n \bmod m = 1$ und $w^{n/2} \bmod m = -1$.

(ii) Für $1 \leq k \leq n-1$ gilt $\sum_{0 \leq j \leq n-1} (w^k)^j = 0$.

(iii) Für $1 \leq k \leq n-1$ gilt $w^k \neq 1$.

(iv) Es gibt $\hat{n}, \hat{w}$ mit $(n \cdot \hat{n}) \bmod m = 1$ und $(w \cdot \hat{w}) \bmod m = 1$.

Die Zahl w spielt also im Ring $\mathbb{Z}_m$ die Rolle einer primitiven n-ten Einheitswurzel, sie besitzt eine Inverse $\hat{w}$, und auch n hat eine Inverse $\hat{n}$.

Beispiel: $w = 2, n = 32, m = 2^{16} + 1 = 65537, \hat{w} = 32769, \hat{n} = 63489$.
Man kann sich überlegen, dass in dieser Situation alle Überlegungen, die wir oben für $\omega, \hat{\omega}, n, n^{-1}$ angestellt haben, für $w, \hat{w}, n, \hat{n}$ ebenso funktionieren, wir also den FFT-Algorithmus auch mit Addition und Multiplikation „modulo m“ benutzen können, obwohl $\mathbb{Z}_m$ normalerweise kein Körper ist. Damit lassen sich die Koeffizienten von Produkten ganzzahliger Polynome mit FFT ohne Umweg über die komplexen Zahlen berechnen, und auch in der Situation der Multiplikation von ganzen Zahlen ist diese FFT-Version günstiger. Einzige Voraussetzung: m muss so groß sein, dass die Koeffizienten c_k des Produktpolynoms schon durch den Wert $c_k \bmod m$ eindeutig bestimmt sind.