

5 Divide-and-Conquer-Rekursionen

5.1 Szenario, statische DC-Rekursionen

Die Technik des algorithmischen Problemlösens durch rekursive Zerlegung eines gegebenen Problems in kleinere Teilprobleme gleichen Typs, bis hin zu direkt lösbaren “kleinen” Fällen und anschliessender Synthese der Gesamtlösung aus den Teillösungen sollte bekannt sein. Ein Schema für eine Algorithmus dieser Art sieht etwa so aus:

Algorithm 3 Divide-and-Conquer

```

procedure DCSOLVE( $B$ )                                ▷ Solve Problem  $B$ 
  if  $\text{size}(B) < \text{bound}$  then
    return(ESOLVE( $B$ ))                                  ▷ Solve directly if  $B$  is small
  end if
   $\langle B_1, B_2, \dots, B_\ell \rangle \leftarrow \text{decompose}(B)$     ▷ Generate smaller subproblems
  for  $j \leftarrow 1, \ell$  do
     $S_j \leftarrow \text{DCSOLVE}(B_j)$                         ▷ Solve subproblems recursively
  end for
   $S \leftarrow \text{compose}(S_1, S_2, \dots, S_\ell)$         ▷ Synthesize solution
  return  $S$ 
end procedure

```

Hier geht es um die Beurteilung des Aufwandes eines solchen Verfahrens. Offensichtlich gilt, wenn man Laufzeit misst, arithmetische Operationen zählt usw.:

$$\text{COST}_{\text{DCSOLVE}}(B) = \begin{cases} \text{COST}_{\text{ESOLVE}}(B) & \text{falls } \text{size}(B) < \text{bound} \\ \text{COST}_{\text{decompose}}(B) \\ + \sum_{j=1}^{\ell} \text{COST}_{\text{DCSOLVE}}(B_j) \\ + \text{COST}_{\text{compose}}(B_1, B_2, \dots, B_\ell) & \text{sonst} \end{cases}$$

Dabei kann die Anzahl und die Grösse der Subprobleme B_1, \dots, B_ℓ von der Instanz B abhängen, nicht nur von deren Grösse. Sie kann auch noch von anderen Parametern abhängen, etwa dem Resultat eines Zufallsprozesses, der bei der Zerlegung mitspielt. Man spricht dann von einer *dynamischen* Situation.

Einfacher zu handhaben und zu beurteilen sind *statische* Situationen, bei denen die Anzahl der Teilprobleme und deren Grösse nur von der Grösse des Inputproblems B abhängt. Bezeichnet man dann mit

$$T_{\mathcal{A}}(n) := \max \{ \text{COST}_{\mathcal{A}}(B) ; \text{size}(B) = n \}$$

die maximalen Kosten eines rekursiven Algorithmus \mathcal{A} auf Inputs der Grösse n , so ergibt sich aus dem Rekursionsschema eine Rekursion

$$T_{\mathcal{A}}(n) = \begin{cases} c & \text{falls } n < \textit{bound} \\ \sum_{1 \leq j \leq \ell} T_{\mathcal{A}}(n_j) + g(n) & \text{sonst} \end{cases}$$

mit einem festen Wert von ℓ , wobei in den sog. *overhead*-Kosten $g(n)$ die Kosten für *decompose* und *compose* zusammengefasst sind.

Auch hier sind noch verschiedene Situationen möglich, je nachdem, wie sich die Grössen n_1, \dots, n_ℓ der Subprobleme B_1, \dots, B_ℓ aus der Grösse n des Inputproblems B ergeben. Natürlich sollte jeweils $n_j < n$ gelten. Zwei Fälle sind häufig anzutreffen:

- Aus B werden ℓ Probleme B_j generiert, deren Grösse n_j jeweils in einem festen Verhältnis α_j zur Grösse n von B steht, d.h., $n_j = \alpha_j \cdot n$ mit einem $0 < \alpha_j < 1$.²
- Aus B werden ℓ Probleme B_j generiert, deren Grösse n_j jeweils um einen bestimmten ganzzahligen Betrag r_j kleiner ist als n , d.h., $n_j = n - r_j$ mit einem $0 < r_j < n$.

Im ersten Fall soll folgende (unwesentliche) Vereinfachung vorgenommen werden: es wird angenommen, dass alle vorkommenden Problemgrössen n bzw. n_j Potenzen einer festen Basis b sind, also

$$n = b^m \quad \text{und} \quad n_j = \alpha_j n = b^{m + \log_b \alpha_j},$$

wobei $0 < s_j = -\log_b \alpha_j \leq m$ ganzzahlig ist. In der Rekursion

$$T_{\mathcal{A}}(n) = \sum_{1 \leq j \leq \ell} T_{\mathcal{A}}(n_j) + g(n)$$

kann man dann $T_{\mathcal{A}}(n)$ durch $t_{\mathcal{A}}(m)$ ersetzen und die Rekursionsgleichung schreibt sich nun

$$t_{\mathcal{A}}(m) = \sum_{1 \leq j \leq \ell} t_{\mathcal{A}}(m - s_j) + g(b^m).$$

Im zweiten Fall gilt mit

$$T_{\mathcal{A}}(n) = \sum_{1 \leq j \leq \ell} T_{\mathcal{A}}(n - r_j) + g(n)$$

²Damit die Grösse eine ganze Zahl ist müsste man eigentlich $n_j = \lfloor \alpha_j \cdot n \rfloor$ oder $n_j = \lceil \alpha_j \cdot n \rceil$ setzen.

eine Rekursion vom gleichen Typ: eine C-Rekursion, die aber nicht homogen ist, da noch der Einfluss der Overheadkosten $g(n)$ zu berücksichtigen ist.

Zusammengefasst: Bezeichnet a_j die Anzahl der Subprobleme der Grösse n_j , so führt der erste Fall auf eine Rekursion vom Typ

$$t_m = a_1 t_{m-1} + a_2 t_{m-2} + \cdots + a_k t_{m-k} + g(b^m)$$

und der zweite Fall auf eine Rekursion vom Typ

$$t_n = a_1 t_{n-1} + a_2 t_{n-2} + \cdots + a_k t_{n-k} + g(n).$$

Dabei sind die $a_j \in \mathbb{N}$, es ist $\sum_j a_j = \ell$ und es kann ohne Einschränkung $a_k \neq 0$ angenommen werden. Für die weitere Analyse ist die Ganzzahligkeit der a_j weniger wichtig, die Tatsache der Nichtnegativität $a_j \geq 0$ dagegen sehr!

Die im Kontext der Divide-and-Conquer-Rekursionen auftretenden Rekursionspolynome bzw. charakteristischen Polynome zeichnen, sich durch die Besonderheit aus, dass die Koeffizienten $a_i \geq 0$ sind. Das hat interessante Folgen:

Lemma 26. Sei $\chi_{\mathbf{a}}(z) = z^k - a_1 z^{k-1} - a_2 z^{k-2} - \cdots - a_k$ das charakteristische Polynom einer C-Rekursion \mathbf{a} mit $a_j \geq 0$ ($1 \leq j \leq k$) und $a_k \neq 0$. Dann gilt:

1. $\chi_{\mathbf{a}}(z)$ hat genau eine reelle Nullstelle $\lambda_1 > 0$ und diese ist einfach.
2. Den Sonderfall $\chi_{\mathbf{a}}(z) = z^k - a_k$ ausgenommen, ist λ_1 dominierende Nullstelle, d.h. für alle anderen Nullstellen η von $\chi_{\mathbf{a}}(z)$ gilt $|\eta| < \lambda_1$.

Für den Beweis ist es bequemer, das Rekursionpolynom $a(z) = 1 - a_1 z - \cdots - a_k z^k$ in Augenschein zu nehmen. Beachte: ist λ eine Nullstelle von $\chi_{\mathbf{a}}(z)$, so ist λ^{-1} eine Nullstelle von $a(z)$.

Bemerkung 12. Tatsächlich interessiert für unsere Anwendungen nur der Fall ganzzahliger $a_j \geq 0$. Dann ist $\lambda_1 \leq 1$ und der Fall $\lambda = 1$ kann nur in der Ausnahmesituation $\chi_{\mathbf{a}}(z) = z^k - 1$ eintreten, die fortan ausgeblendet werden soll.

Im Bezug auf die C-Rekursion

$$x_n = a_1 x_{n-1} + a_2 x_{n-2} + \cdots + a_k x_{n-k} \quad (n \geq k)$$

ist also λ_1 eine *dominierende* Nullstelle! Das bedeutet, dass sich Folgen $(x_n)_{n \geq 0}$, die dieser (homogenen!) Rekursion genügen, typischerweise asymptotisch wie

$$x_n \sim c \cdot \lambda_1^n$$

verhalten. Im Bezug auf die inhomogene Rekursion

$$x_n = a_1 x_{n-1} + a_2 x_{n-2} + \cdots + a_k x_{n-k} + y_{n-k} \quad (n \geq k)$$

gilt, wie im Beispiel in Abschnitt 3.11. ausgeführt,

$$\sum_{n \geq 0} x_n z^n = \frac{b(z) c(z) + z^k d(z)}{a(z) c(z)},$$

wenn $(y_n)_{n \geq 0}$ eine C-rekursive Folge ist:

$$\sum_{n \geq 0} y_n z^n = \frac{d(z)}{c(z)},$$

sodass das asymptotische Verhalten von $\mathbf{x} = (x_n)_{n \geq 0}$ davon abhängt, wie sich die grösste Nullstelle γ von $\chi_c(z)$ und λ_1 zueinander verhalten.

5.2 Beispiele

5.2.1 Einfache Divide-and-Conquer-Rekursionen, Master-Theorem

Der am häufigsten auftretende Rekursionstyp ist

$$f(n) = a \cdot f(n/b) + c \cdot n^d$$

mit positiven Konstanten a, b, c, d , wobei $b > 1$ ganzzahlig sein soll.

Lässt man die Rekursion über die Potenzen von bn laufen, setzt also $n = b^k$, $g(k) = f(n)$, so geht diese Rekursion über in

$$g(k) = a \cdot g(k-1) + c \cdot b^{kd}$$

Schreibt man diese Gleichung noch einmal für $k-1$ an Stelle von k hin

$$g(k-1) = a \cdot g(k-2) + c \cdot b^{(k-1)d}$$

und subtrahiert das b^d -fache der zweiten Gleichung von der ersten, hat man den inhomogenen Anteil eliminiert und erhält

$$g(k) = (a + b^d) \cdot g(k-1) + a \cdot b^d \cdot g(k-2)$$

Das charakteristische Polynom ist

$$\chi(z) = z^2 - (a + b^d)z + a \cdot b^d = (z - a)(z - b^d)$$

mit den Nullstellen a und b^d . Im Fall $a = b^d$ liegt eine doppelte Nullstelle vor!

Damit gilt für die allgemeine Lösung

$$g(k) = \begin{cases} \alpha \cdot a^k + \beta \cdot (b^d)^k & \text{falls } a \neq b^d \\ \alpha \cdot a^k + \beta \cdot k \cdot a^k & \text{falls } a = b^d \end{cases}$$

Die Konstanten α und β hängen von den Anfangsbedingungen ab, die hier nicht spezifiziert wurden. Im allgemeinen (falls $\alpha, \beta \neq 0$) gilt also

$$\begin{array}{llll} a > b^d & \Rightarrow & g(k) \sim \alpha \cdot a^k & \Rightarrow & f(n) \in \Theta(n^{\log_b a}) \\ a = b^d & \Rightarrow & g(k) \sim \beta \cdot k \cdot a^k & \Rightarrow & f(n) \in \Theta(n^d \cdot \log n) \\ a < b^d & \Rightarrow & g(k) \sim \beta \cdot (b^d)^k & \Rightarrow & f(n) \in \Theta(n^d) \end{array}$$

Aussagen wie diese findet man in der Literatur in verschiedenster Spezialisierung (und meist mit ganz anders gearteten ad-hoc-Beweisen versehen) als “Master-Theorem”. Hier ist eine Version aus dem Buch [6] von HEUN, Theorem 2.17, siehe auch die Abschnitte 4.3 und 4.4 in dem Buch [5] von CORMEN, LEISERSON, RIVEST, STEIN, die man mit den hier behandelten Techniken ebenfalls leicht beweisen kann:

Sei

$$f(n) = a \cdot f(n/b) + t(n) \quad \text{mit } f(1) = t(1) > 0$$

mit $a > 0, b \geq 2$, so gilt

$$f(n) \in \begin{cases} \Theta(t(n)) & \text{falls } t(n) \in \Omega(n^{\log_b(a)+\epsilon}) \text{ und } a \cdot t(n/b) \leq c \cdot t(n) \\ \Theta(n^{\log_b a} \log n) & \text{falls } t(n) \in \Theta(n^{\log_b a}) \\ \Theta(n^{\log_b a}) & \text{falls } t(n) \in O(n^{\log_b(a)-\epsilon}) \end{cases}$$

wobei $\epsilon > 0$ und $c < 1$ konstant sind. Das deckt schon sehr viele Fälle ab. Hier eine Auswahl:

$$\begin{array}{ll} f(n) = f(n/2) + c & \Rightarrow f(n) \in \Theta(\log n) \\ f(n) = 2f(n/2) + cn & \Rightarrow f(n) \in \Theta(n \log n) \\ f(n) = 2f(n/2) + cn^2 & \Rightarrow f(n) \in \Theta(n^2) \\ f(n) = 4f(n/2) + cn^2 & \Rightarrow f(n) \in \Theta(n^2 \log n) \\ f(n) = 7f(n/2) + cn^2 & \Rightarrow f(n) \in \Theta(n^{\log_2 7}) \\ f(n) = 2f(n/2) + \log n & \Rightarrow f(n) \in \Theta(n) \\ f(n) = 3f(n/2) + n \log n & \Rightarrow f(n) \in \Theta(n^{\log_2 3}) \\ f(n) = 2f(n/2) + n \log n & \Rightarrow f(n) \in \Theta(n \log^2 n) \\ f(n) = 5f(n/2) + (n \log n)^2 & \Rightarrow f(n) \in \Theta(n^{\log_2 5}) \end{array}$$

Schliesslich soll noch der Rekursionstyp

$$F(n) = F(\alpha_1 n) + F(\alpha_2 n) + \cdots + F(\alpha_s n) + G(n)$$

mit $G(n) \in \Theta(n^s \log^t n)$ und festen $0 < \alpha_j < 1$ angesprochen werden.

Falls man die Rekursionen über die Potenzen einer festen Basis b laufen lässt, hat man eine Rekursion

$$F(n) = \sum_{i=1}^k a_i \cdot F(n/b^i) + \Theta(n^s \cdot \log_b^t n)$$

mit $a_1, a_2, \dots, a_k \geq 0, a_k > 0$ und $s, t, b \in \mathbb{N}, b \geq 2$.

Mit $n = b^m$ und $f_m = F(b^m)$ wird daraus

$$f_m = \sum_{i=1}^k a_i \cdot f_{m-i} + \Theta(b^{ms} \cdot m^t)$$

Betrachte dies als eine inhomogene (forcierte) lineare Rekursion!

$$f_m = \sum_{i=1}^k a_i \cdot f_{m-i} + c \cdot (b^{ms} \cdot m^t)$$

Die forcierende Folge (“overhead”) ist C-rekursiv:

$$\sum_{m \geq 0} b^{ms} m^t z^m = \sum_{m \geq 0} m^t (b^s z)^m = \frac{r(z)}{(1 - b^s \cdot z)^{t+1}}$$

mit einem Polynom $r(z)$ vom Grad $\leq t$.

Das charakteristische Polynom

$$\chi(z) = z^k - a_1 z^{k-1} - \cdots - a_k$$

hat eine dominierende Nullstelle λ .

Für das asymptotische Verhalten der Folge $(f_m)_{m \geq 0}$ ist also entscheidend, wie sich die Nullstellen λ der Rekursion und b^s der overhead-Rekursion zueinander verhalten.

$$\begin{array}{llll} \lambda > b^s & \Rightarrow & t_m \in \Theta(\lambda^m) & \Rightarrow & f(n) \in \Theta(n^{\log_b \lambda}) \\ \lambda = b^s & \Rightarrow & t_m \in \Theta(b^{ms} \cdot m^{t+1}) & \Rightarrow & f(n) \in \Theta(n^s \log_b^{t+1} n) \\ \lambda < b^s & \Rightarrow & t_m \in \Theta(b^{ms} \cdot m^t) & \Rightarrow & f(n) \in \Theta(n^s \cdot \log_b^t n) \end{array}$$

Das entspricht dem “Master-Theorem” im Kapitel 1.9 des Buches [11] von SCHÖNING.

5.2.2 Mergesort

Dieser Algorithmus sollte bekannt sein.

Algorithm 4 Merge

```

procedure MERGE( $A :: list, B :: list$ )           ▷ Merge two (sorted) lists
   $a \leftarrow \text{length}(A)$ 
   $b \leftarrow \text{length}(B)$ 
  if  $a = 0$  then
    return( $B$ )
  end if
  if  $b = 0$  then
    return( $A$ )
  end if
  if  $\text{head}(A) \leq \text{head}(B)$  then             ▷ Compare head elements
    return( $[\text{head}(A), \text{MERGE}(\text{tail}(A), B)]$ )
  else
    return( $[\text{head}(B), \text{MERGE}(A, \text{tail}(B))]$ )
  end if
end procedure

```

Algorithm 5 Mergesort

```

procedure MERGESORT( $A :: list$ )                 ▷ Sort a list
   $\ell \leftarrow \text{length}(A)$ 
   $h \leftarrow \lfloor \ell/2 \rfloor$ 
  if  $\ell < 2$  then return( $A$ )
  end if
   $B \leftarrow A[1..h]$                            ▷ Recursively sort first half of  $A$ 
   $C \leftarrow A[h+1..\ell]$                        ▷ Recursively sort second half of  $A$ 
  return( $\text{MERGE}(B, C)$ )                           ▷ Merge results
end procedure

```

Die Komplexitätsanalyse für diesen Algorithmus ist einfach: bezeichnet $V_{\text{mergesort}}(n)$ die maximale Anzahl von Vergleichsoperationen, die der Algorithmus MERGESORT auf Listen der Länge n benötigt, so ergibt sich

$$V_{\text{mergesort}}(n) = V_{\text{mergesort}}(\lfloor n/2 \rfloor) + V_{\text{mergesort}}(\lceil n/2 \rceil) + \mathcal{O}(n).$$

Der Term $\mathcal{O}(n)$ beinhaltet die Anzahl der Vergleichsoperationen für den Algorithmus MERGE, ausgeführt auf zwei Listen der Längen k und ℓ mit $k + \ell = n$: das ist klarerweise $\leq n$.

Falls die Listenlänge n eine Potenz von 2 ist, wird man auf eine einfache Divide-and-Conquer-Rekursion

$$V(n) = 2V(n/2) + c \cdot n$$

geführt und erhält daraus die bekannte Aussage

$$V_{\text{mergesort}}(n) \in \Theta(n \log n).$$

Man kann sich fragen, ob die Beschränkung auf Listenlängen $n = 2^m$ bei dieser Situation (und ähnlichen) nicht verfälschend wirkt? Das ist nicht der Fall, aber im konkreten technischen Nachweis etwas lästig. Hier soll exemplarisch gezeigt werden, dass man die maximale Anzahl der Vergleichsoperationen für MERGESORT für beliebige Listenlänge *exakt* bestimmen kann und ein völlig analoges Ergebnis wie im Spezialfall erhält.

Betrachten wir also die Rekursion

$$V(n) = V(\lfloor n/2 \rfloor) + V(\lceil n/2 \rceil) + n - 1, \quad V(1) = 0$$

Es wird nun behauptet, dass

$$V(n) = n \cdot \lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1.$$

gilt. Der Beweis wird durch Induktion über n geführt:

- Für $n = 1$ ist alles klar:

$$0 = 1 \cdot \lceil \log 1 \rceil - 2^{\lceil \log 1 \rceil} + 1$$

- Für gerades $n = 2m > 1$ gilt

$$\begin{aligned} V(n) &= V(m) + V(m) + 2m - 1 \\ &= 2(m \cdot \lceil \log m \rceil - 2^{\lceil \log m \rceil} + 1) + 2m - 1 \\ &= 2m(\lceil \log m \rceil + 1) - 2^{\lceil \log m \rceil + 1} + 1 \\ &= n \cdot \lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1 \end{aligned}$$

- Für ungerades $n = 2m + 1 > 1$ gilt

$$\begin{aligned} V(n) &= V(m + 1) + V(m) + 2m \\ &= (m + 1)\lceil \log(m + 1) \rceil - 2^{\lceil \log(m + 1) \rceil} + 1 + m\lceil \log m \rceil - 2^{\lceil \log m \rceil} + 1 + 2m \end{aligned}$$

wobei man nun zu unterscheiden hat:

$$- 2^{k-1} < m < 2^k \Rightarrow \lceil \log(m+1) \rceil = k = \lceil \log m \rceil, \lceil \log n \rceil = k+1$$

$$\begin{aligned} V(n) &= (m+1)k - 2^k + 1 + mk - 2^k + 1 + 2m \\ &= (2m+1)(k+1) - 2^{k+1} + 1 \\ &= n \lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1 \end{aligned}$$

$$- m = 2^{k-1} \Rightarrow \lceil \log(m+1) \rceil = k = \lceil \log m \rceil + 1, \lceil \log n \rceil = k+1$$

$$\begin{aligned} V(n) &= (m+1)k - 2^k + 1 + m(k-1) - 2^{k-1} + 1 + 2m \\ &= (2m+1)(k+1) - 2^{k+1} + 1 \\ &= n \lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1 \quad \square \end{aligned}$$

5.2.3 Rekursive Multiplikation von Zahlen und Polynomen

Die möglichst effiziente Realisierung von arithmetischen Operationen ist ein Hauptthema der *Computeralgebra*, siehe das Buch [13] für eine interessante, breite und fundierte Einführung. Selbst so "alltägliche" Aufgaben, wie das Multiplizieren von Zahlen und Polynomen, und ihre traditionellen algorithmischen Lösungen, hier als "Schulmethoden" bezeichnet, müssen hinterfragt werden, wenn es um das Operieren mit *sehr grossen* Zahlen bzw. Polynomen *sehr grossen Grades* geht.

Hat man zwei Polynome³ von einem Grad $< n$

$$f(X) = \sum_{i=0}^{n-1} f_i X^i, \quad g(X) = \sum_{j=0}^{n-1} g_j X^j,$$

miteinander zu multiplizieren, so besagt die Formel

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

dass bei deren direkten Übertragung in einen Algorithmus jeder der n Koeffizienten a_j mit jedem der Koeffizienten b_j multipliziert werden muss: das sind also insgesamt n^2 Multiplikationen im Bereich der Koeffizienten. Hinzu kommen noch etwa ebensoviele Additionen. Die "Schulmethode" ist also ein Verfahren mit arithmetischer Komplexität $\mathcal{O}(n^2)$. Man hat lange geglaubt, dass dies auch die wahre

³Hier werden Polynome behandelt. Die Methoden und Komplexitätsaussagen gelten auch für das Multiplizieren von ganzen Zahlen, deren übliche Basisdarstellung ja nichts anderes als die Auswertung eines Polynoms an der Basis ist. Das Phänomen des *Übertrags*, das bei Polynomen nicht auftritt, macht die Sache aber etwas komplizierter.

untere Schranke für die Komplexität des Multiplizierens sei — bis ihm Jahr 1862 ein junger russischer Student, A. KARATSUBA, ein Schüler des legendären N. KOLMOGOROV, herausfand, dass dies nicht richtig ist.

Nach der Idee von A. KARATSUBA (im russischen Original 1962, die Arbeit [7] ist die Übersetzung ins Englische) kann die Multiplikation zweier Polynome

$$f(X) = \sum_{i=0}^{2n-1} f_i X^i \quad , \quad g(X) = \sum_{j=0}^{2n-1} g_j X^j$$

vom Grad $\deg f, \deg g < 2n$ wird zurückgeführt werden auf die drei (statt vier!) Multiplikationen von Polynomen, deren Grad $< n$ ist:

$$\begin{aligned} f(X) &= a(X) + X^n b(X) \\ g(X) &= c(x) + x^n d(X) \\ f(X) \cdot g(X) &= a(X) \cdot c(X) + X^n (a(X) \cdot d(X) + b(X) \cdot c(X)) + X^{2n} \cdot b(X) \cdot d(X) \\ &= u(X) + x^n (w(X) - u(X) - v(X)) + X^{2n} v(X) \end{aligned}$$

wobei

$$\begin{aligned} u(X) &:= a(X) \cdot c(X) \\ v(X) &:= b(X) \cdot d(X) \\ w(X) &:= (a(X) + b(X)) \cdot (c(X) + d(X)) \end{aligned}$$

Für die *rekursive* Anwendung dieser Idee im Maple-Programm *karatsuba* wird angenommen, daß n eine Potenz von 2 ist, d.h. $\deg f, \deg g < 2^m$. Der Parameter m wird im rekursiven Programm mitgeführt. Startet man mit beliebigen input-Polynomen, so wird in *kara_mult* das kleinste m bestimmt, sodaß diese Annahmen zutreffen.

Idee und Programm lassen sich auf die Multiplikation von ganzen Zahlen übertragen, wobei das Phänomen der “Überträge” (das es bei Polynomen nicht gibt) die Sache technisch etwas verkompliziert, ohne dass sich am Grundsätzlichen etwas ändert.

```
karatsuba := proc(f,g,X,m)
local a,b,c,d,u,v,w,deg;
if m=0 then RETURN(f*g) fi;
deg := 2^(m-1);
a := sum('coeff(f,X,i)*X^i', 'i'=0..deg-1);
b := sum('coeff(f,X,i+deg)*X^i', 'i'=0..deg-1);
c := sum('coeff(g,X,i)*X^i', 'i'=0..deg-1);
```

```

d := sum('coeff(g,X,i+deg)*X^i','i'=0..deg-1);
u := karatsuba(a,c,X,m-1);
v := karatsuba(b,d,X,m-1);
w := karatsuba(a+b,c+d,X,m-1);
RETURN(eXpand(u+(w-u-v)*X^deg+v*X^(2*deg)));
end;

```

```

kara_mult := proc(f,g,var)
local df,dg,bound;
df := degree(f,var);
dg := degree(g,var);
bound := ceil(simplify(log[2](max(df,dg)+1)));
karatsuba(f,g,var,bound);
end;

```

Komplexitätsanalyse (für $n = 2^m$):

$$t(n) := \begin{cases} \text{Anzahl der Additionen und Multiplikationen} \\ \text{im Koeffizientenbereich bei input-Grad} < n \end{cases}$$

Dann hat man die Rekursion

$$t(2n) = 3 \cdot t(n) + \Theta(n) \quad , \quad t(1) = \Theta(1)$$

und damit

$$\Rightarrow \boxed{t(n) \in \Theta(n^{\log_2 3})}$$

im Gegensatz zu $\Theta(n^{\log_2 4}) = \Theta(n^2)$ für die “Schulmethode”.

Weitere Bemerkungen zur Geschichte....

[10]

5.2.4 Rekursive Multiplikation von Matrizen

Die Multiplikation von zwei $(n \times n)$ -Matrizen erfordert, wenn man einfach die Definition des Matrixproduktes algorithmisiert, einen Aufwand von n^3 Multiplikationen von Matrixkoeffizienten und etwa genausoviele Additionen. Als Konsequenz kann man die meisten der üblichen Matrixoperationen, wie “inverse Matrix berechnen”, “lineares Gleichungssystem lösen” mit einem Aufwand von $\mathcal{O}(n^3)$ arithmetischen Operationen im Koeffizientenring oder -körper lösen. Es geht aber besser! Dies stellte zum ersten Mal V. STRASSEN im Jahr 1969 in der Arbeit [12] heraus.

Ausgangspunkt: die Multiplikation von zwei (2×2) -Matrizen (über irgendeinem (!) Ring) lässt sich mit 7 Multiplikationen im Koeffizientenbereich ausführen — statt mit 8 Multiplikationen nach der “Schulmethode”:

$$\begin{aligned} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} &= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix} \\ &= \begin{bmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{bmatrix} \end{aligned}$$

Man berechnet zunächst 7 Produkte

$$\begin{aligned} c_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\ c_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ c_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\ c_4 &= (a_{11} + a_{12})b_{22} \\ c_5 &= a_{11}(b_{12} - b_{22}) \\ c_6 &= a_{22}(b_{21} - b_{11}) \\ c_7 &= (a_{21} + a_{22})b_{11} \end{aligned}$$

und danach die d_{11}, \dots, d_{22} durch

$$\begin{aligned} d_{11} &= c_1 + c_2 - c_4 + c_6 & d_{12} &= c_4 + c_5 \\ d_{21} &= c_6 + c_7 & d_{22} &= c_2 - c_3 + c_5 - c_7 \end{aligned}$$

Der Witz der (rekursiven) Angelegenheit: die Koeffizienten a_{ij}, b_{ij}, \dots können selbst wieder Matrizen sein, d.h.

Reduktion einer Matrixmultiplikation für zwei $(2n \times 2n)$ -Matrizen auf 7 Matrixmultiplikationen von $(n \times n)$ -Matrizen, dazu 18 Additionen/Subtraktionen von $(n \times n)$ -Matrizen

Das nachfolgende Maple-Program benutzt diese Idee rekursiv und führt die Strassen-Multiplikation für zwei $(n \times n)$ -Matrizen mit $n = 2^m$ aus:

```
strassen := proc(A::Matrix,B::Matrix,m)
local dim,A11,A12,A21,A22,
      B11,B12,B21,B22,
      C1,C2,C3,C4,C5,C6,C7,
      D11,D12,D21,D22;
if m=0 then RETURN(A*B) fi;
```

```

dim := 2^(m-1);
A11 := A[1..dim,1..dim];
A12 := A[1..dim,dim+1..2*dim];
A21 := A[dim+1..2*dim,1..dim];
A22 := A[dim+1..2*dim,dim+1..2*dim];
B11 := B[1..dim,1..dim];
B12 := B[1..dim,dim+1..2*dim];
B21 := B[dim+1..2*dim,1..dim];
B22 := B[dim+1..2*dim,dim+1..2*dim];
C1 := strassen( A12-A22, B21+B22 , m-1);
C2 := strassen( A11+A22, B11+B22, m-1);
C3 := strassen( A11-A21, B11+B12, m-1);
C4 := strassen( A11+A12, B22, m-1);
C5 := strassen( A11, B12-B22, m-1);
C6 := strassen( A22, B21-B11, m-1);
C7 := strassen( A21+A22 , B11, m-1);
D11 := C1+C2-C4+C6 ;
D12 := C4+C5 ;
D21 := C6+C7 ;
D22 := C2-C3+C5-C7 ;
RETURN( [[D11,D12],[D21,D22]]);
end;

```

Komplexitätsanalyse (für $n = 2^m$):

$$t(n) := \begin{cases} \text{Anzahl der arithmetischen Operationen im Koeffizientenbereich für} \\ \text{die Multiplikation von zwei } (n \times n)\text{-Matrizen} \end{cases}$$

Es gilt die Rekursion

$$t(2n) = 7 \cdot t(n) + 18 \cdot n^2, \quad t(1) = 1$$

und die Lösung verhält sich so:

$$\Rightarrow t(n) \in \Theta(n^{\log_2 7})$$

Beachte: $\log_2 7 = 2.808 \dots$! Im Anschluss an die Arbeit von STRASSEN haben er selbst und viele andere Autoren versucht, den Exponenten für die Komplexität der Matrixmultiplikation (obere Schranke) soweit wie möglich herunterzudrücken - siehe dazu Abschnitt 7.8 im Buch [11]. Seit 1982 steht der Rekord bei $\mathcal{O}(n^{2.38\dots})$, gehalten von D. COPPERSMITH und S. WINOGRAD. Die Monografie [2] von P. BÜRGISSER, M. CLAUSEN und M. A. SHOKROLLAHI bearbeitet dieses schwierige Thema profund.

Siehe auch [1].

5.2.5 Ein SAT-Algorithmus

Das Problem SATISFIABILITY oder kurz SAT ist *das* NP-vollständige Problem. Es sind bis heute keine effizienten Algorithmen bekannt, um die Erfüllbarkeit von aussagenlogischen Formeln (gegeben in Klauselform, konjunktiver Normalform) testen. Bei n Variablen benötigt das *brute-force*-Durchprobieren einen Aufwand von $\mathcal{O}(2^n)$. Es geht aber besser. Ein wesentliche Schritt hin zu besseren SAT-Algorithmen war das in [8] präsentierte Verfahren von B. MONIEN und E. SPECKENMEYER.

Es bezeichne $\mathbb{B} = \{0, 1\} = \{\text{false}, \text{true}\} = \{\text{f}, \text{t}\} = \{\top, \perp\}$ die zweielementige boolesche Algebra mit den Operationen der Konjunktion (“und”) \wedge , Disjunktion (“oder”) \vee , und Negation (“nicht”) \neg .

$X_n = \{x_1, x_2, \dots, x_n\}$ sei eine Menge von (booleschen) Variablen. Mit $\overline{X}_n = \{\overline{x}_1, \overline{x}_2, \dots, \overline{x}_n\}$ werden die negierten Variablen notiert, $U_n = X_n \cup \overline{X}_n$ ist die Menge der *Literale*. Die Negation von negierten Literalen ist mit $\overline{\overline{x}} = x$ erklärt.

Formeln (in konjunktiver Normalform, CNF) sind folgendermassen definiert:

- Eine Klausel (über X_n) ist eine Menge von Literalen $C = \{u_1, u_2, \dots, u_s\} \subseteq U_n$, wobei komplementäre Paare x_i, \overline{x}_i nicht auftreten sollen. Klauseln verstehen sich als *Disjunktion ihrer Literale*, d. h. sie sind als

$$C = u_1 \vee u_2 \vee \dots \vee u_s$$

zu lesen. Für $s = 0$ wird die leere Klausel $C = \emptyset$ mit **false** identifiziert (= neutrales Element bezgl. Disjunktion), oft auch als \square notiert.

$\mathcal{C}(X_n)$ bezeichnet die Menge der Klauseln über X_n

- Eine CNF-Formel (über X_n) ist eine Menge $F = \{C_1, C_2, \dots, C_t\}$ von verschiedenen Klauseln über X_n , zu verstehen als *Konjunktion ihrer Klauseln*

$$F = C_1 \wedge C_2 \wedge \dots \wedge C_t$$

Für $t = 0$ wird die leere Formel $F = \emptyset$ mit **true** identifiziert (= neutrales Element bezgl. Konjunktion).

$\mathcal{F}(X_n)$ ist die Menge der Formeln über X_n .

Jedes Element $\mathbf{b} = (b_1, b_2, \dots, b_n) \in \mathbb{B}^n$ definiert eine *Bewertung* der Variablen durch

$$\phi_{\mathbf{b}} : X_n \rightarrow \mathbb{B} : x_i \mapsto b_i \quad (1 \leq i \leq n)$$

Daraus ergibt sich eine Bewertung der Klauseln und Formeln, wie üblich,

- Fortsetzung von $\phi_{\mathbf{b}}$ zu einer Bewertung der Literale:

$$\phi_{\mathbf{b}} : U_n \rightarrow \mathbb{B} \text{ mittels } \phi_{\mathbf{b}}(\bar{x}) = \overline{\phi_{\mathbf{b}}(x)} \quad (x \in X_n)$$

- Fortsetzung von $\phi_{\mathbf{b}}$ zu einer Bewertung der Klauseln:

$$\phi_{\mathbf{b}} : \mathcal{C}(X_n) \rightarrow \mathbb{B} : \phi_{\mathbf{b}}(u_1 \vee u_2 \vee \dots \vee u_s) = \phi_{\mathbf{b}}(u_1) \vee \phi_{\mathbf{b}}(u_2) \vee \dots \vee \phi_{\mathbf{b}}(u_s)$$

- Fortsetzung von $\phi_{\mathbf{b}}$ zu einer Bewertung der Formeln:

$$\phi_{\mathbf{b}} : \mathcal{F}(X_n) \rightarrow \mathbb{B} : \phi_{\mathbf{b}}(C_1 \wedge C_2 \wedge \dots \wedge C_t) = \phi_{\mathbf{b}}(C_1) \wedge \phi_{\mathbf{b}}(C_2) \wedge \dots \wedge \phi_{\mathbf{b}}(C_t)$$

- Jede boolesche Formel $F \in \mathcal{F}(X_n)$ definiert somit eine n -stellige boolesche Funktion

$$\langle F \rangle : \mathbb{B}^n \rightarrow \mathbb{B} : \mathbf{b} \mapsto \phi_{\mathbf{b}}(F)$$

Theorem 27. *Zu jeder n -stelligen booleschen Funktion $f : \mathbb{B}^n \rightarrow \mathbb{B}$ gibt es eine Formel $F \in \mathcal{F}(X_n)$ mit $\langle F \rangle = f$.*

Das Problem SAT (SATISFIABILITY)

- Instanzen sind boolesche Formeln $F \in \mathcal{F}(X_n)$
($n =$ Instanzengrösse)

- Entscheide, ob F erfüllbar ist oder nicht,
d.h. ob es ein $\mathbf{b} \in \mathbb{B}^n$ gibt mit $\phi_{\mathbf{b}}(F) = \langle F \rangle(\mathbf{b}) = \mathbf{true}$ oder nicht.

k -SAT ist analog zu SAT, aber mit höchstens k Literalen pro Klausel.

Theorem 28. *SAT und k -SAT für $k \geq 3$ sind NP-vollständige Probleme. 2-SAT ist effizient entscheidbar.*

Es gibt also (derzeit) keine effizienten Algorithmen für SAT. Das *brute-force*-Durchprobieren aller 2^n Bewertungen hat eine worst-case-Komplexität von 2^n . es geht aber (etwas) besser. Dazu betrachten man *partielle Bewertungen*, also Funktionen, die nur einigen der Variablen einen Wahrheitswert zuordnen.

Eine partielle Bewertungen von X_n ist eine Funktion $\psi : U_n \rightarrow \mathbb{B} \cup U_n$ mit

$$\psi(u) \in \{u, 0, 1\} \text{ und } \psi(\bar{u}) = \overline{\psi(u)} \quad (u \in U_n).$$

- Fortsetzung zu einer partiellen Bewertung der Klauseln:

$$\begin{aligned} \psi : \mathcal{C}(X_n) &\rightarrow \mathcal{C}(X_n) \cup \{\mathbf{t}\} : \\ \psi(u_1 \vee u_2 \vee \dots \vee u_s) &= \psi(u_1) \vee \psi(u_2) \vee \dots \vee \psi(u_s) \end{aligned}$$

Algorithm 6 MS-SAT

```

procedure TEST( $F$ )                                ▷ Test  $F$  for Satisfiability
  if  $F = \emptyset$  then                               ▷ Trivial case
    return(true)
  end if
  if  $F = \square$  then                                 ▷ Trivial case
    return(false)
  end if
  select shortest  $C = u_1 \vee u_2 \vee \dots \vee u_s \in F$   ▷ Select clause
  for  $i = 1..s$  do
    if TEST( $\psi_j(F)$ ) then                           ▷ Test subcases recursively
      return(true)
    end if
  end for
  return(false)
end procedure

```

Dann gilt im Fall von k -SAT-Formeln:

$$\begin{aligned} T(n) &\leq T(n-1) + T(n-2) + \dots + T(n-s) \\ &\leq T(n-1) + T(n-2) + \dots + T(n-k) \end{aligned}$$

Definiert man die C-rekursive Folge $(t_k(n))_{n \geq 0}$ mittels

$$t_k(n) = t_k(n-1) + t_k(n-2) + \dots + t_k(n-k) \quad (n \geq k)$$

und mit Anfangswerten $t_k(0), t_k(1), \dots, t_k(k-1)$, für die $T(j) \leq t_k(j)$ ($0 \leq j < k$), so gilt

$$T(n) \leq t_k(n) \quad \text{für alle } n \geq 0.$$

Ist nun α_k die (eindeutige) im Intervall $0 < x < 2$ liegende reelle Nullstelle des Polynoms

$$z^k - z^{k-1} - z^{k-2} - \dots - z - 1,$$

so gilt

$$t_k(n) \sim \alpha_k^n \quad \text{für } n \rightarrow \infty.$$

Einige Zahlenwerte:

$$\begin{aligned}
 k = 1 : & \quad \alpha_1 = 1 \\
 k = 2 : & \quad \alpha_2 = \phi = \frac{1 + \sqrt{5}}{2} = 1.6181 \dots \\
 k = 3 : & \quad \alpha_3 = 1.8393 \dots \\
 k = 4 : & \quad \alpha_4 = 1.9276 \dots \\
 k = 5 : & \quad \alpha_5 = 1.966 \dots
 \end{aligned}$$

Einie etwas genauere Betrachtung liefert für k -SAT-Formeln sogar

$$T(n) \leq t_{k-1}(n) \sim \alpha_{k-1}^n \quad (n \rightarrow \infty)$$

Satz 31. *Der Berechnungsaufwand $T(n)$ der divide-and-conquer-Methode in Anhängigkeit von der Anzahl der Variablen n , angewendet auf k -SAT-Formeln, verhält sich wie*

$$T(n) \leq t_{k-1}(n) \sim \alpha_{k-1}^n \quad (n \rightarrow \infty)$$

Für das NP-vollständige Problem 3-SAT hat man also die Schranke mit dem Goldenen Schnitt ϕ :

$$T(n) \leq t_2(n) \sim \phi^n = (1.6181 \dots)^n \quad (n \rightarrow \infty).$$

Auch die *divide-and-conquer*-Methode hat exponentielles Verhalten in der Variablenanzahl n , aber der relative Aufwand (gemessen and der *brute-force*-Methode) verringert sich exponentiell!

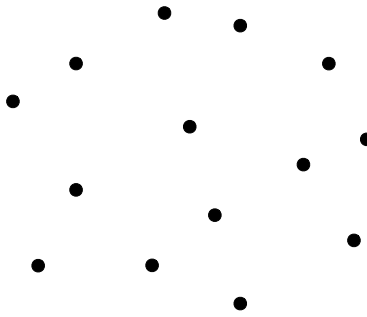
Einige Werte für $k = 3$:

n	$(\phi/2)^n$
5	0.3465678100...
10	0.1201092469...
20	0.0144262311...
50	0.0000249966...
100	$0.6248 \dots \cdot 10^{-9}$

5.2.6 Das Closest-Pair-Problem

Dieser Abschnitt orientiert sich an dem entsprechenden Kapitel aus dem Buch [9] von Th. OTTMANN.

- Die Problemstellung:

Abbildung 9: Eine Punktwolke in \mathbb{R}^2

– Gegeben: P endliche Menge von Punkten $p = (p_x, p_y) \in \mathbb{R}^2$

– Gesucht:

$$\delta(P) = \min \{ \text{dist}(p, q) ; p, q \in P, p \neq q \}$$

wobei $\text{dist}(p, q)$ den euklidischer Abstand von $p, q \in \mathbb{R}^2$ bezeichnet.

Allgemeiner definiert man für endliche $P, Q \subseteq \mathbb{R}^2$

$$\delta(P, Q) = \min \{ \text{dist}(p, q) ; p \in P, q \in Q, p \neq q \}$$

also $\delta(P) = \delta(P, P)$

- Algorithmisches Problem

- berechne $\delta(P)$ möglichst effizient

- (evtl. auch: bestimme ein Paar $p, q \in P$ mit $\text{dist}(p, q) = \delta(P)$)

Eine naive Lösung der Aufgabe geht so:

- berechne alle paarweisen Distanzen

$$\{ \text{dist}(p, q) ; p, q \in P, p \neq q \}$$

- berechne das Minimum dieser Menge

Die Komplexität dieses Verfahrens ist

- für $\#P = n$: $\binom{n}{2}$ Distanzen berechnen: $\mathcal{O}(n^2)$

- Minimum in $\mathcal{O}(n^2)$ Zahlen suchen: $\mathcal{O}(n^2)$ Vergleiche

- insgesamt ein $\mathcal{O}(n^2)$ -Verfahren

Man kann aber auch eine Divide-and-Conquer Ansatz verfolgen:

- Zerlege P in zwei (ungefähr) gleich grosse Teile P_ℓ, P_r
- berechne (rekursiv) $\delta(P_\ell), \delta(P_r)$
- berechne $\delta(P_\ell, P_r)$
- berechne $\delta(P) = \min(\delta(P_\ell), \delta(P_r), \delta(P_\ell, P_r))$

Dieser Ansatz führt — bei naivem Vorgehen zur Berechnung von $\delta(P_\ell, P_r)$ — auf eine Rekursion für den Aufwand $t(n)$ an Operationen (bei $\#P = n$)

$$t(n) = 2 \cdot t(n/2) + 2 \cdot \left(\frac{n}{2}\right)^2 + 1$$

$$t(2) = 1$$

mit einer Lösung $t(n) \in \mathcal{O}(n^2)$

Die eben beschriebene Divide-and-Conquer-Lösung nützt nicht die Tatsache aus, dass es sich bei den Punkten um Elemente aus \mathbb{R}^2 handelt!

Ein geometrisches Divide-and-Conquer-Verfahren berücksichtigt diesen speziellen Aspekt:

1. Zerlege P mit Hilfe des Medians m der x -Koordinaten der $p \in P$ in zwei (ungefähr) gleich grosse Teile

$$P_\ell = \{p \in P; p_x \leq m\} \quad P_r = \{p \in P; m < p_x\}$$

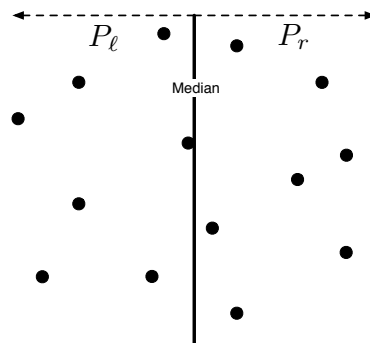
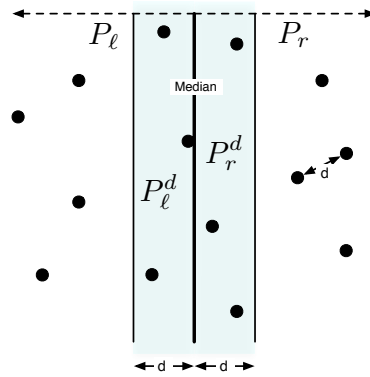


Abbildung 10: Median-Zerlegung der Punktmenge

Bemerkung: Median-Berechnung für n Elemente geht in $\mathcal{O}(n)$

2. Berechne (rekursiv) $\delta(P_\ell), \delta(P_r)$ und $d = \min(\delta(P_\ell), \delta(P_r))$

Abbildung 11: Rekursive Berechnung von $\delta(P)$

3. Bestimme

$$P_\ell^d = \{p \in P; m - d \leq p_x \leq m\} \quad P_r^d = \{p \in P; m < p_x \leq m + d\}$$

4. Berechne $\delta(P_\ell^d, P_r^d)$ und somit

$$\delta(P) = \min(d, \delta(P_\ell, P_r)) = \min(d, \delta(P_\ell^d, P_r^d))$$

Vorsicht: auch P_ℓ^d und P_r^d können noch $\mathcal{O}(n)$ Elemente enthalten!

ABER: zu jedem $p \in P_\ell^d$ kann es nur sehr wenige $q \in P_r^d$ geben, die "nahe" ($\approx d$) von p liegen:

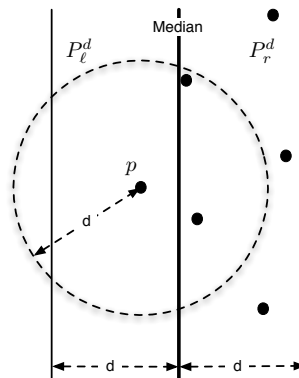


Abbildung 12: Situation im zentralen Streifen

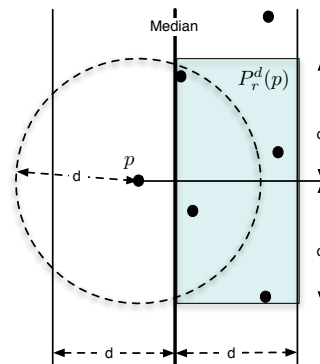


Abbildung 13: Wenige Nachbarn im zentralen Streifen!

Aus der geometrische Situation folgt:

$$\begin{aligned} \text{dist}(p, q) < d &\Rightarrow p_y - d < q_y < p_y + d \\ &\Rightarrow q \in \underbrace{P_r^d \cap \{q; p_y - d < q_y < p_y + d\}}_{P_r^d(p)} \end{aligned}$$

und damit hat man für $\delta(P_\ell, P_r)$:

$$\begin{aligned} \delta(P_\ell, P_r) &= \delta(P_\ell^d, P_r^d) \\ &= \min_{p \in P_\ell^d} \delta(p, P_r^d) = \min_{p \in P_\ell^d} \delta(p, P_r^d(p)) \end{aligned}$$

Aus der Geometrie folgt (grosszügig abgeschätzt!)

$$(*) \quad \#P_r^d(p) \leq 7$$

d.h. zu jedem $p \in P_\ell^d$ gibt es höchstens sieben $q \in P_r^d$ mit $\text{dist}(p, q) < d$,

Folgerung: Man muss bei geschickter (!!) Organisation der Daten nur $\mathcal{O}(n)$ Operationen durchführen, um $\delta(P_\ell, P_r)$ zu berechnen!

Eine geometrische Überlegung zum Beweis von (*):

- Wieviele disjunkte, offene Kreisscheiben mit Radius R kann man in einem $(a \times b)$ -Rechteck unterbringen?

Die Anzahl ist $k(a, b, R) \leq \frac{a \cdot b}{\pi \cdot R^2}$.

- Wieviele verschiedene Punkte, die paarweise einen Abstand $\geq d$ haben, kann man in einem $(a \times b)$ -Rechteck unterbringen?

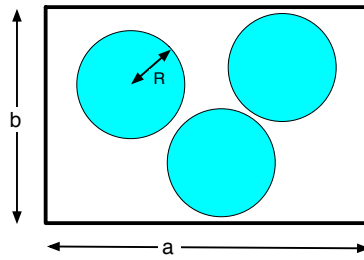


Abbildung 14: Disjunkte Kreisscheiben in einem Rechteck

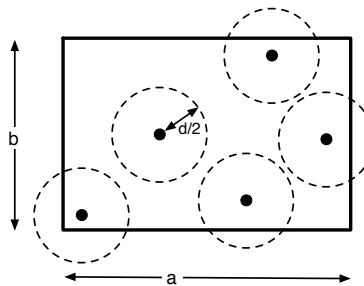


Abbildung 15: Punkte mit Mindestabstand in einem Rechteck

Die Anzahl ist $\leq k(a + d, b + d, d/2) \leq \frac{(a+d)(b+d)}{\pi \cdot (d/2)^2}$

Sind also die Elemente von P_ℓ^d nach y -Koordinaten sortiert

$$P_\ell^d : \begin{array}{ccccccc} p^{(1)} & p^{(2)} & p^{(3)} & \dots & p^{(s)} \\ p_y^{(1)} \leq & p_y^{(2)} \leq & p_y^{(3)} \leq & \dots & < p_y^{(s)} \end{array}$$

und die Elemente von P_r^d nach y -Koordinaten sortiert

$$P_r^d : \begin{array}{ccccccc} q^{(1)} & q^{(2)} & q^{(3)} & \dots & q^{(t)} \\ q_y^{(1)} \leq & q_y^{(2)} \leq & q_y^{(3)} \leq & \dots & < q_y^{(t)} \end{array}$$

so treten Kandidaten für $\text{dist}(p, q)$ in Blöcken der Länge ≤ 7 auf:

$$\begin{array}{ccccccc} P_\ell^d : & p^{(1)} & p^{(2)} & p^{(3)} & \dots & \boxed{p} & \dots & p^{(s)} \\ P_r^d : & q^{(1)} & q^{(2)} & q^{(3)} & \dots & \underbrace{q^{(i)} \ q^{(i+1)} \ \dots \ q^{(i+j)}}_{P_r^d(p)} & \dots & q^{(s)} \end{array}$$

Der Divide-and-Conquer Closest-Pair-Algorithmus sieht also so aus:

1. Zerlege P mit Hilfe des Medians m der x -Koordinaten der $p \in P$ in zwei (ungefähr) gleich grosse Teile

$$P_\ell = \{p \in P; p_x \leq m\} \quad P_r = \{p \in P; m < p_x\}$$

2. Berechne (rekursiv) $\delta(P_\ell), \delta(P_r)$ und $d = \min(\delta(P_\ell), \delta(P_r))$
3. Bestimme

$$P_\ell^d = \{p \in P; m - d \leq p_x \leq m\} \quad P_r^d = \{p \in P; m < p_x \leq m + d\}$$

4. Sortiere P_ℓ^d und P_r^d nach den y -Koordinaten ihrer Elemente
5. Berechne $\delta(P_\ell^d, P_r^d)$ mit $\mathcal{O}(n)$ Operationen
6. $\delta(P) = \min(d, \delta(P_\ell^d, P_r^d))$

Die Komplexität $t(n)$ des cp-Algorithmus ergibt sich nun so:

- Medianberechnung und Zerlegung $P = P_\ell \uplus P_r : \mathcal{O}(n)$
- Herausziehen von P_ℓ^d und $P_r^d : \mathcal{O}(n)$
- Sortieren von P_ℓ^d und $P_r^d : \mathcal{O}(n \log n)$
- Berechnen von $\delta(P_\ell^d, P_r^d) : \mathcal{O}(n)$
- Divide-and-Conquer Rekursionsgleichung

$$t(n) = 2t(n/2) + \mathcal{O}(n \log n), \quad t(2) = 1$$

- Verhalten der Lösung der Rekursionsgleichung

$$t(n) \in \mathcal{O}(n \log^2 n)$$

Bemerkung: Man kann das Sortieren auf jeder Rekursionsstufe durch einmaliges Sortieren zu Beginn und geschickte Verwaltung dieser Information ersetzen, dadurch sogar Gesamtkomplexität $t(n) \in \mathcal{O}(n \log n)$ erreichen

5.2.7 Schnelle Fouriertransformation

Die Schnelle Fouriertransformation (FFT) wird im Rahmen der Algorithmen der Arithmetik noch genau behandelt. Hier soll nur auf die Divide-and-Conquer-Idee eingegangen werden.

Ist $a(X) = a_0 + a_1 X + a_2 X^2 + \dots + a_n X^n$ ein (reelles oder komplexes) Polynom n -ten Grades und ist ξ eine komplexe Zahl, so nennt man das Berechnen der

Zahl

$$a(\xi) = a_0 + a_1 \xi + a_2 \xi^2 + \cdots + a_n \xi^n$$

die *Auswertung von $a(X)$ an der Stelle ξ* . Führt man die Berechnung entsprechend dem sogenannten ‘‘Hornerschema’’

$$a(\xi) = a_0 + (a_1 + \cdots + (a_{n-1} + (a_{n-1} + a_n \xi) \xi) \xi \dots) \xi$$

durch, so benötigt man dafür n Multiplikationen und n Additionen von komplexen Zahlen. Man kann zeigen, dass dies optimal ist! Dies ist einer der seltenen Fälle, in dem man die untere Schranke für die arithmetische Aufgabe exakt kennt.

Wertet man das Polynom $a(X)$ an $n + 1$ *verschiedenen* Stellen $\xi_0, \xi_1, \dots, \xi_n \in \mathbb{C}$ *simultan* aus, d.h. betrachtet man die Abbildung

$$a(\xi) = a_0 + a_1 \xi + a_2 \xi^2 + \cdots + a_n \xi^n \longmapsto \langle a(\xi_0), a(\xi_1), \dots, a(\xi_n) \rangle$$

so kann man das natürlich mit insgesamt n^2 Additionen und n^2 Multiplikationen machen, indem man jeden dieser Werte separat berechnet

Diese simultane Auswertung lässt sich übersichtlich als lineare Transformation von \mathbb{C}^{n+1} darstellen:

$$\begin{bmatrix} 1 & \xi_0 & \xi_0^2 & \xi_0^3 & \cdots & \xi_0^n \\ 1 & \xi_1 & \xi_1^2 & \xi_1^3 & \cdots & \xi_1^n \\ 1 & \xi_2 & \xi_2^2 & \xi_2^3 & \cdots & \xi_2^n \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \xi_n & \xi_n^2 & \xi_n^3 & \cdots & \xi_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} a(\xi_0) \\ a(\xi_1) \\ a(\xi_2) \\ \vdots \\ a(\xi_n) \end{bmatrix}$$

deren Transformationsmatrix

$$V(\xi_0, \xi_1, \xi_2, \dots, \xi_n) = [\xi_i^j]_{0 \leq i, j \leq n}$$

eine VANDERMONDE-Matrix ist mit der

$$\det V(\xi_0, \xi_1, \xi_2, \dots, \xi_n) = \prod_{0 \leq i < j \leq n} (\xi_j - \xi_i).$$

Die Verschiedenheit der Interpolationsstellen garantiert gerade die Invertierbarkeit der Transformation.

Wichtig ist, dass man aus diesem Datensatz $\langle a(\xi_0), a(\xi_1), \dots, a(\xi_n) \rangle$ das Polynom $a(X)$ (d.h., die Folge (a_0, a_1, \dots, a_n) seiner Koeffizienten) wieder eindeutig zurückgewinnen kann: das ist die Aufgabe der *Interpolation*. Einen klassischer Weg weist die *Interpolationsformel von LAGRANGE*:

$$a(X) = \sum_{0 \leq i \leq n} a(\xi_i) \cdot \frac{\prod_{j \neq i} (X - \xi_j)}{\prod_{j \neq i} (\xi_i - \xi_j)},$$

die quantitativ die Tatsache zum Ausdruck bringt, dass ein Polynom vom Grad $\leq n$ durch seine Werte an $n + 1$ Interpolationsstellen eindeutig bestimmt ist. Dabei ist man (bis auf die Verschiedenheit) in der Wahl der Interpolationsstellen $\xi_0, \xi_1, \dots, \xi_n \in \mathbb{C}$ völlig frei.

Die Diskrete Fouriertransformation (DFT) legt, motiviert durch die Absicht, die klassische kontinuierliche Fouriertransformation diskret zu approximieren, eine spezielle Wahl der Interpolationsstellen fest:

Für $N \geq 1$ hat die Gleichung $X^N = 1$ genau N komplexe Lösungen:

$$1 = \omega_N^0, \omega_N, \omega_N^2, \omega_N^3, \dots, \omega_N^{N-1}, \quad \text{wobei } \omega_N = e^{2\pi i/N} = \cos \frac{2\pi}{N} + i \cdot \sin \frac{2\pi}{N}.$$

Diese Menge nennt man die Menge der *komplexen n -ten Einheitswurzeln*, kurz mit \mathcal{R}_N bezeichnet. Es gilt also

$$DFT_N : \mathbb{C}^N \rightarrow \mathbb{C}^N : (a_0, a_1, \dots, a_{N-1}) \mapsto \langle a(\omega_N^0), a(\omega_N), a(\omega_N^2), \dots, a(\omega_N^{N-1}) \rangle,$$

in Matrixform (mit $\omega = \omega_N$) geschrieben:

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \dots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{4(N-1)} & \dots & \omega^{(N-1)^2} \end{bmatrix}}_{V(\omega^0, \omega^1, \omega^2, \dots, \omega^{N-1})} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{N-1} \end{bmatrix} = \begin{bmatrix} a(1) \\ a(\omega) \\ a(\omega^2) \\ \vdots \\ a(\omega^{N-1}) \end{bmatrix}$$

wobei also

$$V(\omega^0, \omega^1, \omega^2, \dots, \omega^{N-1}) = [\omega^{i \cdot j}]_{0 \leq i, j < N}$$

ist. Interessanterweise ist die inverse Matrix (bis auf einen skalaren Faktor) vom gleichen Typ:

$$V(\omega^0, \omega^1, \omega^2, \dots, \omega^{N-1})^{-1} = N \cdot [\omega^{-i \cdot j}]_{0 \leq i, j < N} = N \cdot V(\omega^0, \omega^{-1}, \omega^{-2}, \dots, \omega^{-(N-1)}).$$

Beispiele für DFT-Matrizen

- $n = 2$

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

- $n = 3$

$$DFT_3 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega_3 & \omega_3^2 \\ 1 & \omega_3^2 & \omega_3^4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega_3 & \omega_3^2 \\ 1 & \omega_3^2 & \omega_3 \end{bmatrix}$$

mit $\omega_3 = \frac{-1+i\sqrt{3}}{2}, \omega_3^2 = \frac{-1-i\sqrt{3}}{2}$

- $n = 4$

$$DFT_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & i^2 & i^3 \\ 1 & i^2 & i^4 & i^6 \\ 1 & i^3 & i^6 & i^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & i^2 & i^3 \\ 1 & i^2 & i^0 & i^2 \\ 1 & i^3 & i^2 & i^1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

- $n = 5$

$$DFT_5 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_5 & \omega_5^2 & \omega_5^3 & \omega_5^4 \\ 1 & \omega_5^2 & \omega_5^4 & \omega_5^6 & \omega_5^8 \\ 1 & \omega_5^3 & \omega_5^6 & \omega_5^9 & \omega_5^{12} \\ 1 & \omega_5^4 & \omega_5^8 & \omega_5^{12} & \omega_5^{16} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_5 & \omega_5^2 & \omega_5^3 & \omega_5^4 \\ 1 & \omega_5^2 & \omega_5^4 & \omega_5 & \omega_5^3 \\ 1 & \omega_5^3 & \omega_5 & \omega_5^4 & \omega_5^2 \\ 1 & \omega_5^4 & \omega_5^3 & \omega_5^3 & \omega_5 \end{bmatrix}$$

mit

$$\omega_5 = \frac{\sqrt{5} - 1 + i\sqrt{2}\sqrt{5 + \sqrt{5}}}{4}$$

- $n = 6$

$$DFT_6 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_6 & \omega_6^2 & \omega_6^3 & \omega_6^4 & \omega_6^5 \\ 1 & \omega_6^2 & \omega_6^4 & \omega_6^6 & \omega_6^8 & \omega_6^{10} \\ 1 & \omega_6^3 & \omega_6^6 & \omega_6^9 & \omega_6^{12} & \omega_6^{15} \\ 1 & \omega_6^4 & \omega_6^8 & \omega_6^{12} & \omega_6^{16} & \omega_6^{20} \\ 1 & \omega_6^5 & \omega_6^{10} & \omega_6^{15} & \omega_6^{20} & \omega_6^{25} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_6 & \omega_6^2 & -1 & \omega_6^4 & \omega_6^5 \\ 1 & \omega_6^2 & \omega_6^4 & 1 & \omega_6^2 & \omega_6^4 \\ 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & \omega_6^4 & \omega_6^2 & 1 & \omega_6^4 & \omega_6^2 \\ 1 & \omega_6^5 & \omega_6^4 & 1 & \omega_6^2 & \omega_6 \end{bmatrix}$$

mit

$$\omega_6 = \frac{1 + i \cdot \sqrt{3}}{2}$$

$$\omega_6^2 = \frac{-1 + i \cdot \sqrt{3}}{2}$$

$$\omega_6^4 = \frac{-1 - i \cdot \sqrt{3}}{2} = -\omega_6$$

$$\omega_6^5 = \frac{1 - i \cdot \sqrt{3}}{2} = -\omega_6^2$$

- $n = 8$

$$DFT_8 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_8 & i & \omega_8^3 & -1 & \omega_8^5 & -i & \omega_8^7 \\ 1 & i & -1 & -i & 1 & i & -1 & -i \\ 1 & \omega_8^3 & -i & \omega_8 & -1 & \omega_8^7 & i & \omega_8^5 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & \omega_8^5 & i & \omega_8^7 & -1 & \omega_8 & -i & \omega_8^3 \\ 1 & -i & -1 & i & 1 & -i & -1 & i \\ 1 & \omega_8^7 & -i & \omega_8^5 & -1 & \omega_8^3 & i & \omega_8 \end{bmatrix}$$

mit

$$\begin{aligned} \omega_8 &= \frac{1+i}{\sqrt{2}} & \omega_8^3 &= \frac{-1+i}{\sqrt{2}} = \omega_8 - \sqrt{2} \\ \omega_8^5 &= \frac{-1-i}{\sqrt{2}} = -\omega_8 & \omega_8^7 &= \frac{1-i}{\sqrt{2}} = \omega_8 - i \cdot \sqrt{2} \end{aligned}$$

Die *Schnelle Fouriertransformation* beruht nun auf der Tatsache (vermutlich zuerst von C. F. GAUSS 1805 bemerkt, aber erst im Nachlass 1866 veröffentlicht und nicht beachtet, "neu erfunden" in dem vielzitierten Artikel [4] von J. W. COOLEY und J. W. TUKEY, siehe auch [3]), dass man bei dieser Wahl der Interpolationsstellen eine interessante Rekursion einleiten kann:

Will man ein Polynom $a(X) = \sum_{i=0}^{2N-1} a_i X^i$ vom Grad $< 2N$ an den $2N$ Interpolationsstellen ω_{2N}^j ($0 \leq j < 2N$) auswerten, so kann man $a(X)$ zerlegen:

$$\begin{aligned} a(X) &= \underbrace{a_0 + a_2 X^2 + a_4 X^4 + \cdots + a_{2N-2} X^{2N-2}}_{a_{\text{even}}(X^2)} \\ &\quad + X \underbrace{(a_1 + a_3 X^2 + a_5 X^4 + \cdots + a_{2N-1} X^{2N-2})}_{a_{\text{odd}}(X^2)} \end{aligned}$$

Damit ist aber

$$a(\omega_{2N}^j) = a_{\text{even}}(\omega_{2N}^{2j}) + \omega_{2N}^j \cdot a_{\text{odd}}(\omega_{2N}^{2j})$$

Beachtet man nun noch die simple Tatsache

$$\omega_{2N}^{2k} = e^{2\pi \cdot (2k)/(2N)} = e^{2\pi \cdot k/N} = \omega_N^k$$

und

$$\omega_{2N}^{N+j} = \omega_{2N}^N \cdot \omega_{2N}^j = \omega_2 \cdot \omega_{2N}^j = -\omega_{2N}^j,$$

so ergibt sich folgendes Schema für die Berechnung von DFT_{2N} :

- berechne $(a_{\text{even}}(\omega_{2N}^j))_{0 \leq j < N} = (a'_0, a'_1, \dots, a'_{N-1})$: das ist eine DFT_N -Berechnung
- berechne $(a_{\text{odd}}(\omega_{2N}^j))_{0 \leq j < N} = (a'_0, a'_1, \dots, a'_{N-1})$: das ist eine DFT_N -Berechnung
- berechne

$$(a(\omega_{2N}^j))_{0 \leq j < N} = (a'_j + \omega_{2N}^j \cdot a''_j)_{0 \leq j < N}$$

und

$$(a(\omega_{2N}^{N+j}))_{0 \leq j < N} = (a'_j - \omega_{2N}^j \cdot a''_j)_{0 \leq j < N}$$

Symbolisch geschrieben:

$$DFT_{2N}(\mathbf{a}) = DFT_N(\mathbf{a}_{\text{even}}) \bowtie_{\omega^k} DFT_N(\mathbf{a}_{\text{odd}})$$

wobei \bowtie_{ω^k} ist die sogenannte "butterfly-Operation" ist:

$$\begin{pmatrix} a(\omega_{2N}^k) \\ a(\omega_{2N}^{N+k}) \end{pmatrix} = \begin{pmatrix} 1 & \omega^k \\ 1 & -\omega^k \end{pmatrix} \begin{pmatrix} a_{\text{even}}(\omega_N^k) \\ a_{\text{odd}}(\omega_N^k) \end{pmatrix} \quad (0 \leq k < N)$$

Eine grafische Darstellung im Fall $N = 4$

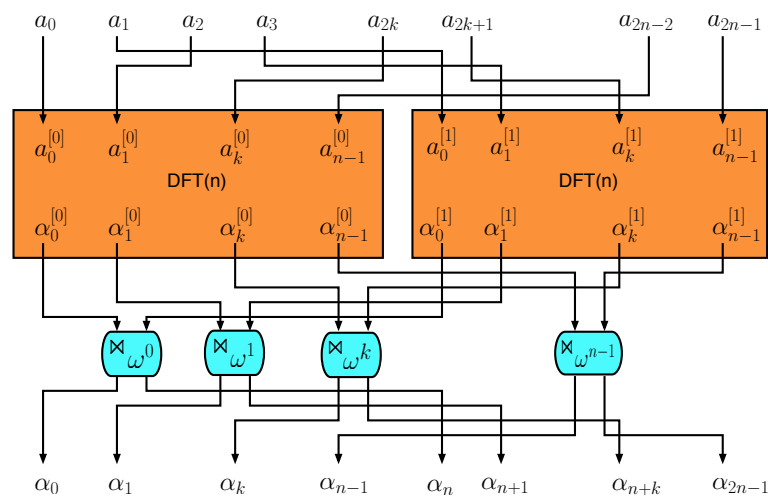


Abbildung 16: FFT-Rekursionsschritt

Die rekursive Anwendung dieser Idee bezeichnet man als *Schnelle Fouriertransformation* (FFT) - siehe Algorithmus 7.

Eine grafische Darstellung der FFT für $N = 8$ als Schaltkreis:

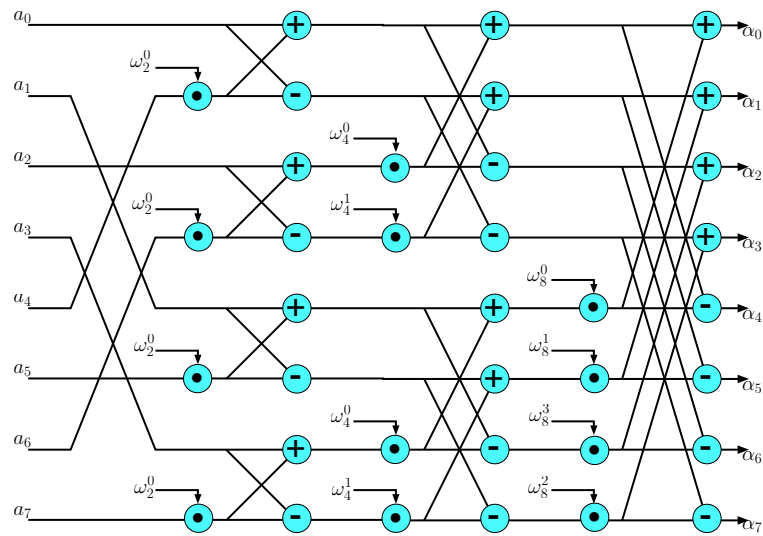


Abbildung 17: FFT-Schaltkreis

Algorithm 7 Schnelle Fouriertransformation

```

procedure FFT( $A :: list, k :: integer$ )  ▷ Fast Fourier Transform of order  $2^k$ 
   $N \leftarrow 2^k$ 
  if  $k = 0$  then  ▷ base case
    return( $A$ )
  end if
   $\omega_n \leftarrow \exp(2\pi i/N)$ 
   $\omega \leftarrow 1$ 
   $a_{even} \leftarrow [A[0], A[2], \dots, A[N-2]]$ 
   $a_{odd} \leftarrow [A[1], A[3], \dots, A[N-1]]$ 
   $y_{even} \leftarrow FFT(a_{even}, k-1)$   ▷ recursive call
   $y_{odd} \leftarrow FFT(a_{odd}, k-1)$   ▷ recursive call
  for  $t = 0..(N/2) - 1$  do  ▷ butterfly operation
     $y[t] \leftarrow y_{even}[t] + \omega \cdot y_{odd}[t]$ 
     $y[t + (N/2)] \leftarrow y_{even}[t] - \omega \cdot y_{odd}[t]$ 
     $\omega \leftarrow \omega \cdot \omega_n$ 
  end for
  return( $y$ )
end procedure

```

Der Aufwand $F(N)$ zur Berechnung von DFT_N bei rekursiver Verwendung dieser Idee, gemessen in arithmetischen Operationen mit komplexen Zahlen, genügt also einer Rekursionsgleichung

$$F(2N) = 2 \cdot F(N) + \mathcal{O}(N)$$

und das führt auf

$$F(N) \in \Theta(N \cdot \log N).$$

5.3 Dynamische Rekursionen: das Beispiel Quicksort

5.4 Eine nichtlineare Rekursion für binäre Bäume

5.5 Kommentare, Literatur, Ausblicke

Literatur

- [1] Dario Bini and Victor Y. Pan. *Polynomial and matrix computations. Vol. 1*. Progress in Theoretical Computer Science. Birkhäuser Boston Inc., Boston, MA, 1994. Fundamental algorithms.
- [2] P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic Complexity Theory*, volume 315 of *Grundlehren der Mathematischen Wissenschaften*. Springer-Verlag, 1996.
- [3] James W. Cooley. How the FFT gained acceptance. In *A history of scientific computing (Princeton, NJ, 1987)*, ACM Press Hist. Ser., pages 133–140. ACM, New York, 1990.
- [4] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.
- [5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition, 2001.
- [6] Volker Heun. *Grundlegende Algorithmen*. Vieweg, 2. edition, 2003.
- [7] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics-Doklady*, 7:595–586, 1863.
- [8] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Appl. Math.*, 10(3):287–295, 1985.

-
- [9] Thomas Ottmann. *Prinzipien des Algorithmenentwurfs*. Spektrum Akademischer Verlag, 1997.
- [10] A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing (Arch. Elektron. Rechnen)*, 7:281–292, 1971.
- [11] Uwe Schöning. *Algorithmik*. Spektrum Akademischer Verlag, 2001.
- [12] Volker Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.
- [13] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, 1999.