Algorithmentechnik

Version vom
15. Februar 2001

Manuel Kauers

nach einer Vorlesung von

Prof. Dr. rer. nat. Thomas Beth

Wintersemester 2000/2001
Universität Karlsruhe (TH)

Inhaltsverzeichnis

1 Einführung

Algorithmen, also Verfahren zur Lösung von Problemen, spielen in der gesamten Informatik eine herausragende Rolle. Die Vorlesung *Algorithmentechnik* beschäftigt sich mit dem Entwurf und der Analyse dieser Algorithmen. Dabei liegt der Schwerpunkt mehr auf der Entwicklung angepaßter Algorithmen als darauf, bestehende „universelle“ Algorithmen kennenzulernen. Zur Präsentation fertiger Algorithmen ist nach Ansicht des Dozenten keine eigene Vorlesung nötig,[1] da es hierzu hinreichend viele gute Bücher gibt, etwa [Knu81] oder [Sed98]. Ziel dieser Vorlesung ist es stattdessen, die Fähigkeit zu entwickeln, zu einem vorgegebenen Problem einen „*optimalen*“ Algorithmus zu entwickeln.

Algorithmus

optimaler Algorithmus

Dabei ist a priori keineswegs klar, wann ein Algorithmus optimal ist. War es noch vor wenigen Jahrzehnten das vorrangige Ziel der Algorithmentechnik, aus naiven Algorithmen mit einer Laufzeit von z. B. $O(n^2)$ neue Algorithmen etwa der Laufzeit $O(n \log n)$ oder gar $O(n)$ zu entwickeln, so ist dies heute weniger wichtig. Man interessiert sich inzwischen eher für Verbesserungen von exponentieller Laufzeit auf subexponentielle oder gar polynomielle Laufzeit. Aufgrund der sich überproportional entwickelnden Rechnertechnologie sind nämlich die geringeren Entwicklungskosten bei der Anwendung naiver polynomieller Algorithmen oft entscheidend, weil der Geschwindigkeitsvorteil der vermeintlich „besseren“ Algorithmen fast vernachlässigbar ist.

Wir betrachten die Algorithmentechnik unter dem Gesichtspunkt der *Taxonomie*, d. h. der Klassifizierung, von Problemlösungsverfahren. Es gibt grundlegend verschiedene Ansätze zum Entwurf von Algorithmen:

Taxonomie

1. *Analogie-Schlüsse*
2. *Extrapolation* und *Interpolation*
3. *Vertafelung* und *Vorausberechnung*
4. Verwendung von *Heuristiken*
5. *Neuronale Netze*
6. *Fuzzy Logic*
7. Optische Verfahren
8. DNA-Verfahren (*Chemical computing*)
9. *Quanten-Computing*
10. *Probabilistische Verfahren*
11. ...

Ebenfalls in den Bereich der Algorithmentechnik fällt auch der gesamte Bereich des Entwurfs von *Protokollen*.

[1] Natürlich kommen auch in dieser Vorlesung einige Standard-Algorithmen vor. Eine Zusammenstellung dieser findet sich in Anhang A ab Seite 53.

Bsp. 1 (Analogie-Schlüsse) Betrachte die Multiplikation zweier Ganzzahlen mittels *shift and add*:

Algorithmus 1.1: shift and add

```
function multiply(a, b)
  k = 0;
  while b ≠ 0 do
    if b.isOdd() then
      k = k + a;
    b = ½b;
    a = a + a;
  return k;
```

Dieser Algorithmus sei als bekannt vorausgesetzt und gesucht ist ein Algorithmus zur Potenzierung. In shift and add wird die Halbgruppenstruktur von $(\mathbb{Z}, +)$ verwendet. Tauscht man diese durch $(\mathbb{Z}\backslash\{0\}, \cdot)$ aus, erhält man den gewünschten Algorithmus. □

Algorithmus 1.2: square and multiply

```
function power(a, b)
  k = 1;
  while b ≠ 0 do
    if b.isOdd() then
      k = k · a;
    b = ½b;
    a = a · a;
  return k;
```

Bsp. 2 (Interpolation und Vertafelung) Gesucht ist ein guter Algorithmus zur Approximation von $\sin x$ für einen Gleitkommawert $x \in [0, N]$. Die Genauigkeit der Approximation soll 10^{-K} betragen.

Ansatz: In einer Tabelle werden die Werte von $\sin(n)$ für $n = 0, 1, \ldots, N$ sowie von $\sin(i\cdot 10^{-k})$ für $i = 1, \ldots, 9$ und $k = 0, \ldots, K$ vertafelt. Aus der Mathematik ist das Additionstheorem

$$\sin(a + b) = \sin a \cos b + \cos a \sin b \qquad (a, b \in \mathbb{R})$$

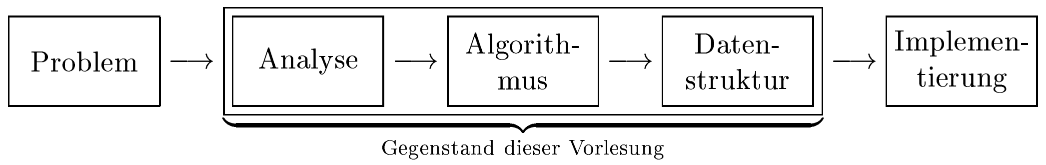
bekannt. Sei $x \approx \sum_{k=0}^{K} c_k 10^{-k}$ mit $c_0 \in \{0, \ldots, N\}$ und $c_1, \ldots, c_k \in \{0, 1, \ldots, 9\}$. Dann gilt:

$$\begin{aligned}\sin(x) &\approx \sin\left(\sum_{k=0}^{K} c_k 10^{-k}\right) = \sin\left(\sum_{k=0}^{K-1} c_k 10^{-k} + c_K 10^{-K}\right)\\ &= \sin\left(\sum_{k=0}^{K-1} c_k 10^{-k}\right)\cos(c_K 10^{-K}) + \cos\left(\sum_{k=0}^{K-1} c_k 10^{-k}\right)\sin(c_K 10^{-K}).\end{aligned}$$

Unter Verwendung von $\cos x = \sin(x + \frac{\pi}{2})$ läßt sich daraus leicht ein rekursiver Approximationsalgorithmus entwerfen. Wegen $|\sin'(x)| \leq 1$ $(x \in \mathbb{R})$ überträgt sich die Genauigkeit der Eingabe auf das Ergebnis. $\square$

Zur Algorithmentechnik im weiteren Sinne gehören nicht nur Analyse und Entwicklung von Algorithmen. Vielmehr ist dies nur ein Schritt von vielen auf dem Weg von einem Problem zu dessen Lösung. Eine umfassendere Darstellung dieses Weges könnte z. B. wie im folgenden Diagramm aussehen:

Abbildung 1.1: Das Thema der Algorithmentechnik



Anmerkungen des Autors

Dieses Skript will in folgenden Punkten wesentlich von der zugrunde liegenden Vorlesung abweichen: Verständlichkeit, Ausführlichkeit und Konsistenz der Darstellung. Insbesondere habe ich nach reiflicher Überlegung auf eine vierfarbige Darstellung des Fließtextes verzichtet. Dennoch sollte alles, was in der Vorlesung zur Sprache kam, irgendwo in diesem Skript in angemessenem Umfang enthalten sein, was gelegentlich zu einem Zielkonflikt führte …

An dieser Stelle möchte ich all jenen danken, die durch Korrekturen und kritische Anmerkungen zur Qualität dieses Skriptes beigetragen haben. Neben vielen anderen sind hier besonders Dominique Unruh und Alexander Sung zu nennen.

2 Das Prinzip der Wiederverwendbarkeit

2.1 Von Euklid zu ELBA

ggT

Bereits in der Antike war bekannt, daß der größte gemeinsame Teiler (ggT) zweier natürlicher Zahlen u, v der Gleichung

$$\mathrm{ggT}(u, v) = \mathrm{ggT}(u, v \bmod u)$$

genügt: Wegen $u \bmod v \leq u$ folgt daraus bereits unmittelbar die Korrektheit des Euklidischen Algorithmus zur Bestimmung von $\mathrm{ggT}(u, v)$:

Algorithmus 2.1: Berechnung des größten gemeinsamen Teilers nach Euklid

```
function gcd(u, v)
  if u mod v = 0 then
    return v;
  return gcd(v, u mod v);
```

Wir nehmen diesen Algorithmus als Ausgangspunkt für weitere Überlegungen.

Zunächst fällt auf, daß die von Euklid verwendeten Gesetze, auf denen der Algorithmus beruht, in jedem euklidischen Ring gelten (vgl. Anhang B, S. 60). Folglich kann man den Algorithmus ohne Veränderung in all jenen Ringen ausführen, solange man die mod-Operation implementieren kann.

Aus der Algebra ist außerdem bekannt, daß jeder euklidische Ring R Hauptidealring ist (vgl. Anhang B, S. 61). Damit läßt sich zeigen, daß es zu je zwei Elementen $u, v \in R$ eindeutig bestimmte Elemente $r, s \in R$ gibt mit

$$u \cdot r + v \cdot s = \mathrm{ggT}(u, v) =: g.$$

Es stellt sich die Frage, ob sich der Euklidische Algorithmus so erweitern läßt, daß er neben g auch r und s liefert. Dies ist in der Tat möglich:

O. B. d. A. sei nun $u > v$. Die Folge (t_i) sei rekursiv definiert durch

$$t_0 := u, \qquad t_1 := v, \qquad t_i := t_{i-2} \bmod t_{i-1}$$

Dann ist (t_i) (bzw. allgemein (∂t_i)) streng monoton fallend in $\mathbb{N}$, also abbrechend. t_i ist gerade das u der Rekursionstiefe i in Algorithmus 2.1. Das letzte t_i, das nicht 0 ist, ist also der größte gemeinsame Teiler von u und v.

Da also $t_{i_0} = \mathrm{ggT}(u, v)$ für ein gewisses i_0 gilt, kann man versuchen, in jedem Iterationsschritt Elemente r_i und s_i so zu wählen, daß die Gleichung

$$u \cdot r_i + v \cdot s_i = t_i$$

schleifeninvariant bleibt. Dann nämlich sind $r = r_{i_0}$ und $s = s_{i_0}$ die gesuchten Elemente.

Es bleibt zu überlegen, wie r_i und s_i zu setzen sind. Wegen $t_0 = u$ und $t_1 = v$ kann man $(r_0, s_0) = (1, 0)$ und $(r_1, s_1) = (0, 1)$ vorgeben. Betrachten wir zwei aufeinanderfolgenden Gleichungen:

$$\left.\begin{aligned} u \cdot r_i + v \cdot s_i &= t_i \\ u \cdot r_{i+1} + v \cdot s_{i+1} &= t_{i+1}. \end{aligned}\right\} \quad (1)$$

Setzt man $q := t_i \operatorname{div} t_{i+1}$ und zieht das q-fache der unteren Gleichung von der oberen ab, so erhält man

$$\underbrace{t_i - (t_i \operatorname{div} t_{i+1}) \cdot t_{i+1}}_{= t_i \bmod t_{i+1} =: t_{i+2}} = u \underbrace{(r_i - q \cdot r_{i+1})}_{=: r_{i+2}} + v \underbrace{(s_i - q \cdot s_{i+1})}_{=: s_{i+2}}$$

ELBA

Damit ergibt sich ohne weiteres Zutun der Algorithmus *ELBA*:

Algorithmus 2.2: ELBA

```
function elba(u, v)
  return elba(u, 1, 0, v, 0, 1);
function elba(u, r, s, v, r', s')
  if u mod v = 0 then
    return (v, r', s');
  else
    q = u div v;
    return elba(v, r', s', u − q · v, r − q · r', s − q · s');
```

Da die Anzahl der Schleifendurchläufe identisch ist mit der Anzahl der Schleifendurchläufe im ursprünglichen Algorithmus, ändert sich die Komplexität beim Übergang von Algorithmus 2.1 zu Algorithmus 2.2 nur minimal.

Die Algorithmensammlung im Anhang enthält iterative Varianten von Algorithmus 2.1 und Algorithmus 2.2.

Betrachten wir nochmals das Gleichungssystem (1). Eine zu (1) äquivalente Darstellung ist

$$\begin{pmatrix} r_i & s_i \\ r_{i+1} & s_{i+1} \end{pmatrix} \cdot \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} t_i \\ t_{i+1} \end{pmatrix}.$$

Setzt man

$$T_{(a,b)} := \begin{pmatrix} 0 & 1 \\ 1 & -a \operatorname{div} b \end{pmatrix},$$

so läßt sich der Iterationsschritt in Algorithmus 2.2 in der Form

$$(t_{i+1}, t_{i+2}) = \underbrace{T_{(t_i, t_{i+1})}}_{=: T_i} \cdot (t_i, t_{i+1})$$

schreiben. Damit gilt nach Konstruktion des Algorithmus

$$\begin{aligned} (\operatorname{ggT}(u, v), 0) &= T_{i_0 - 1} \cdot \dots \cdot T_1 \cdot T_0 \cdot (u, v), \\ \begin{pmatrix} r_{i_0} & s_{i_0} \\ r_{i_0+1} & s_{i_0+1} \end{pmatrix} &= T_{i_0 - 1} \cdot \dots \cdot T_1 \cdot T_0 \cdot I. \end{aligned}$$

Das Matrizenwort $T_{i_0-1} \cdots T_1 T_0$ enthält die *Kettenbruchdarstellung* von u und v im Sinn des folgenden Beispiels.

Kettenbruch-darstellung

Bsp. 3 (Analogiebeispiel) Zu je zwei Zahlen $u, v \in \mathbb{N}$ gibt es eine Kettenbruchdarstellung[2] von $\frac{u}{v}$. Zum Beispiel hat $(u, v) = (64, 29)$ die Darstellung

$$\frac{64}{29} = 2 + \frac{6}{29} = 2 + \cfrac{1}{4 + \cfrac{5}{6}} = 2 + \cfrac{1}{4 + \cfrac{1}{1 + \cfrac{1}{5}}}.$$

Die Zahlen $2, 4, 1, 5$ dieser Darstellung sind nun genau das (-1)-fache der Zahlen, die in den Matrizen T_i unten rechts stehen. Demnach ist $i_0 = 4$ und

$$\begin{pmatrix} r_4 & s_4 \\ r_5 & s_5 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -5 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -4 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -2 \end{pmatrix} = \begin{pmatrix} 5 & -11 \\ -29 & 64 \end{pmatrix},$$

also

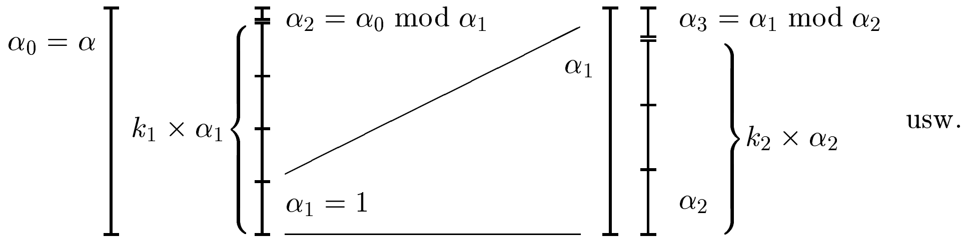
$$\mathrm{ggT}(64, 29) = 64 \cdot 5 + 29 \cdot (-11) = 1,$$

wie erwartet (64 und 29 sind teilerfremd). Man beachte auch, daß $u \cdot r_5 + v \cdot s_5 = 0$ ist, der Algorithmus also tatsächlich an dieser Stelle terminiert hat. □

Bsp. 4 (Längenbestimmung mit Euklids Algorithmus) Ein Stab habe die unbekannte Länge α. Mit Hilfe eines Maßstabs der Länge 1 soll die Länge α bestimmt werden, wobei der Maßstab die Messung von Bruchteilen der Länge 1 nicht zuläßt.

$\alpha_1 := 1$ gehe k_1 mal in $\alpha_0 := \alpha$, der Rest habe die Länge $\alpha_2 = \alpha_0 \bmod \alpha_1$. Wir verwenden nun α_2 als Maßstab zur Messung von α_1 und erhalten entsprechend k_2 und einen Rest α_3 (vgl. Abb. 2.1).

Abbildung 2.1: Meßverfahren



[2] Für $(u, v) \in \mathbb{N}^2$ ist gewährleistet, daß der Kettenbruch endlich ist. Allerdings ist die Kettenbruchdarstellung einer Zahl nicht eindeutig, z. B. gilt

$$5 + \frac{1}{3} = 5 + \cfrac{1}{2 + \cfrac{1}{1}}$$

Dieses Verfahren wiederholt man, bis α_k verschwindet oder hinreichend klein ist. Es ergibt sich eine Folge (k_i), aus der sich die Länge α bezüglich 1 angeben läßt:

$$\frac{\alpha_0}{\alpha_1} = k_1 + \frac{\alpha_2}{\alpha_1} = k_1 + \frac{1}{\frac{\alpha_1}{\alpha_2}} = k_1 + \frac{1}{k_2 + \frac{\alpha_3}{\alpha_2}} = k_1 + \frac{1}{k_2 + \frac{1}{k_3 + \cdots}}$$

$$\Longrightarrow \quad \alpha = k_1 + \frac{1}{k_2 + \frac{1}{k_3 + \cdots}}$$

Nonius

Die Funktionsweise des *Nonius* an einem Meßschieber beruht auf diesem Prinzip (Rekursionstiefe 1). □

2.2 Das Birthday-Phänomen und die Pollardsche ϱ-Methode

Seien X und Y endliche Mengen ($|X| = n$, $|Y| = m$) und $f\colon X \to Y$ eine zufällig ausgewählte Funktion.[3] Die Wahrscheinlichkeit p dafür, daß es $i, j \in X$ gibt mit $f(i) = f(j)$, ist unerwartet hoch: Es ist $p = 1 - P(f \text{ ist injektiv})$, und weil es aus kombinatorischen Gründen genau

$$n! \cdot \binom{m}{n} = \prod_{i=0}^{n-1} (m - i)$$

injektive Funktionen $f\colon X \to Y$ gibt, ist folgende Rechnung möglich:

$$P(f \text{ ist injektiv}) = \prod_{i=0}^{n-1} \left(1 - \frac{i}{m}\right) \le \prod_{i=0}^{n-1} \mathrm{e}^{-\frac{i}{m}} \le \mathrm{e}^{-\sum_{i=0}^{n-1} \frac{i}{m}} = \mathrm{e}^{-\frac{(n-1)n}{2m}} \tag{2}$$

(Beachte $\mathrm{e}^x \ge 1 + x$ $(x \in \mathbb{R})$) Wie groß muß nun z. B. X sein, damit p größer als 50% ist? Nach (2) gilt

$$\begin{aligned} & P(f \text{ ist injektiv}) < \tfrac{1}{2} \\ \Longleftarrow \quad & \mathrm{e}^{-\frac{(n-1)n}{2m}} < \tfrac{1}{2} \\ \Longleftarrow \quad & -\frac{(n-1)n}{2m} < -\ln 2, \end{aligned}$$

also kann man im Fall

$$n > \tfrac{1}{2} + \tfrac{1}{2}\sqrt{1 + 8 \ln 2 \cdot m} = O(\sqrt{m}).$$

mit großer Sicherheit sagen, daß f nicht injektiv ist.

Birthday-Phänomen

Bsp. 5 (Birthday-Phänomen) Sei Y die Menge der Tage eines Jahres (also $|Y| = 365$) und X eine Personengruppe. Die Wahrscheinlichkeit, daß zwei Personen aus X am gleichen Tag Geburtstag haben, ist nach obiger Rechnung bereits dann größer als 50%, wenn $|X| \ge 23$ ist. □

[3] Für fest vorgegebenes $g\colon X \to Y$ ist $P(f = g) = \frac{1}{m^n}$.

Ziel dieses Abschnitts ist die Entwicklung eines Faktorisierungsalgorithmus. Dabei werden wir das Birthday-Phänomen weiter unten zur Abschätzung der mittleren Laufzeit heranziehen.

Faktorisierungsalgorithmen spielen in der Kryptoanalyse eine große Rolle. Da viele Kryptoverfahren darauf beruhen, daß die Multiplikation wesentlich schneller möglich ist als eine Primfaktorzerlegung (vgl. Abschnitt 2.3), ist es wichtig zu wissen, wie schnell man eine Primfaktorzerlegung durchführen kann. Eine Methode zur Faktorisierung wurde von Pollard vorgeschlagen (vgl. [Cor90]):

Es seien $p, q \in \mathbb{Z}$ prim und $n = p \cdot q$.

Algorithmus 2.3: Faktorisierung mit der Pollardschen ϱ-Methode

```
function factorize(n)
  i = 1;  k = 1;
  x_1 = random(2, n − 2);
  while true do
    i = i + 1;  x_i = x_{i−1}^2 − 1 mod n;
    d = gcd(x_k − x_i, n);
    if d ≠ 1 and d ≠ n then
      return [d, n/d];
    if i = 2 · k then
      k = i;
```

Zunächst ist klar, daß der Algorithmus nie ein falsches Ergebnis liefert, da $gcd(x_k - x_i, n)$ unabhängig von x_k und x_i stets ein Teiler von n ist. Unklar ist also nur noch Komplexität und Terminierung des Algorithmus. Betrachte die Eingabe $n = 2 \cdot 5 = 10$ und den Startwert $x_1 = 4$. Man rechnet leicht nach, daß in diesem Fall $x_{2i-1} = 4$ und $x_{2i} = 5$ $(i \in \mathbb{N})$ gilt. Die Variable x_k beinhaltet stets das letzte berechnete Folgenglied, dessen Index k eine Zweierpotenz war. Im Beispiel ist ab $k > 2$ also $x_k = 5$ konstant. Es ist also stets $x_k - x_i = 5 - 4 = 1$ oder $x_k - x_i = 5 - 5 = 0$, im ersten Schleifendurchlauf gilt $x_k - x_i = 4 - 4 = 0$. Es gilt also jedesmal $d = 1$ oder $d = n$, d. h. der Algorithmus terminiert nicht.[4]

Algorithmus 2.3 ist also offensichtlich als *Las-Vegas-Algorithmus* einzustufen: Alle Ausgaben des Algorithmus sind richtig, aber es ist nicht garantiert, daß der Algorithmus terminiert.

Las-Vegas-Algorithmus

Allerdings liefert Pollards Algorithmus in der Praxis erstaunlich gute Ergebnisse, so daß eine Abschätzung der mittleren Laufzeit (im Falle eines Terminierens) lohnend erscheint. Wählt man z. B. im obigen Beispiel zufällig $x_1 = 3$ statt $x_1 = 4$ als Startelement, so erhält man $x_2 = x_1^2 - 1 \bmod 10 = 8$ und damit wegen $d = 3 - 8 = -5$ den Teiler 5 bereits im zweiten Schritt.

Die Folge (x_n) kann nach Konstruktion nur endlich viele Werte annehmen, daher gibt es mit Sicherheit zwei Indizes i und j mit $x_i = x_j$ $(i \neq j)$. Faßt man (x_n) als Folge von Zufallszahlen auf, so ist nach dem Birthday-Phänomen eine Kollision im Mittel nach $O(\sqrt{n})$ Schritten zu erwarten. Aufgrund der rekursiven

[4] Das hätte man einfacher haben können: Ist n prim, so hat der Algorithmus keine Chance und sucht ewig nach Teilern.

Schlaufe

Definition von (x_n) folgt $x_{i+t} = x_{j+t}$ $(t \in \mathbb{N})$ aus $x_i = x_j$. Die Folgenglieder lassen sich also in einer ϱ-förmigen *Schlaufe* anordnen, wie in Abbildung 2.2 dargestellt.

Ist p der kleinste Teiler von n, so gilt $(x \bmod n) \bmod p = x \bmod p$ für jedes x. Für die Folge $(\widetilde{x}_i)$ mit $\widetilde{x}_i := x_i \bmod p$ überträgt sich daher die Rekursion der x_i: Es gilt $\widetilde{x}_i = \widetilde{x}_{i-1}^2 - 1 \bmod p$ $(i \in \mathbb{N})$. Wie oben argumentiert man mit dem Birthday-Phänomen, daß sich die Folge $(\widetilde{x}_i)$ nach $O(\sqrt{p})$ Schritten „aufrollt", d. h., daß es $i \neq j$ gibt mit $\widetilde{x}_{i+t} = \widetilde{x}_{j+t}$ $(t \in \mathbb{N})$.

Ist t der kleinste Index, für den ein Folgenglied von $(\widetilde{x}_i)$ wiederkehrt, und ist u die Länge der p-Schlaufe, so gilt $\widetilde{x}_{t+i} = \widetilde{x}_{t+i+u}$ und damit nach Definition $p \mid (x_{t+i+u} - x_{t+i})$ $(i \in \mathbb{N})$.

Für $k \geq t$ liegt $y = x_k$ in der p-Schlaufe. Ist zudem $k > u$, so wird y mit jedem Schlaufenelement verglichen, eine Kollision ist also sicher. Für jedes $i \geq k$ mit $x_i \equiv x_k \bmod p$ und $x_i \not\equiv x_k \bmod n$ findet der Algorithmus einen nichttrivialen Faktor.

$$\begin{aligned}
& & x_i^2 - 1 &\equiv x_k^2 - 1 \bmod n \\
&\Longrightarrow & x_i^2 - x_k^2 &\equiv 0 \bmod n \\
&\Longrightarrow & (x_i - x_k)(x_i + x_k) &\equiv 0 \bmod n \\
&\Longrightarrow & p = \operatorname{ggT}(x_i - x_k, x_i + x_k) &\mid n.
\end{aligned}$$

Dies ist mit großer Wahrscheinlichkeit nach $O(\sqrt{p}) = O(\sqrt{\sqrt{n}}) = O(\sqrt[4]{n})$ Schritten der Fall.

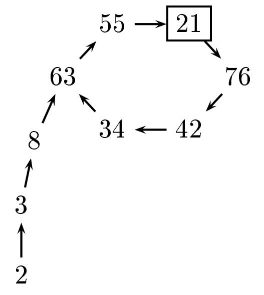
Die Analyse zeigt, daß bei kleineren Faktoren mit kürzeren Laufzeiten zu rechnen ist. Außerdem läßt sich der Algorithmus auch auf den allgemeineren Fall $n = p_1 \cdots p_k$ (p_i prim) anwenden, indem man die Ergebnisse rekursiv solange weiter zerlegt, bis sie prim sind.[5] Auch in diesem Fall liegt die erwartete Laufzeit im Bereich von $O(\sqrt[4]{n})$. Man beachte außerdem, daß das Verfahren eine Speicherkomplexität von $O(1)$ hat, denn von der Folge (x_i) brauchen in jedem Iterationsschritt nur die aktuellen Werte von x_i und x_k bekannt zu sein (vgl. Darstellung im Anhang).

Wie bereits im einführenden Beispiel gesehen, gibt es Abhilfe im Fall einer nicht endenden Schleife. Zum einen kann die Wahl eines anderen Startwerts x_1 zum Erfolg führen, zum anderen kann man prinzipiell aber auch das gesamte Polynom $x^2 - 1$ durch ein (geeignetes) anderes austauschen, etwa durch $x^2 - 3$ oder $x^2 - 4$.

Bsp. 6 Wir faktorisieren $n = 91$ mit Hilfe der Pollardschen ϱ-Methode. Abbildung 2.2 zeigt die zum Startwert 2 gehörige ϱ-Schlaufe. Bei Erreichen des Glieds 21 ist $x_k = x_4 = 63$ und wegen $\operatorname{ggT}(63 - 21, 91) = 7$ haben wir einen Teiler gefunden. Modulo 7 hat die Folge die Form $2 \to 3 \to 1 \to 0 \rightleftarrows 6$. Der Teiler wird also wie vorhergesagt genau dort gefunden, wo sich die Folge modulo 7 beginnt aufzuwickeln. □

[5] Primzahltests mit polynomieller Laufzeit sind bekannt

Abbildung 2.2: Das ϱ der Pollardschen ϱ-Methode



Das Beispiel zeigt, daß die Länge der Schlaufen sowie die Komplexität des Algorithmus mit einer gewissen Toleranz in den erwarteten Größenordnungen liegt. Tabelle 2.3 zeigt, daß die Ergebnisse bei größeren Zahlen wesentlich genauer getroffen werden. Angegeben ist für Zahlen n verschiedener Größenordnung die Anzahl der berechneten Folgenglieder, die Größe der n-Schlaufe sowie die Schlaufengröße des gefundenen Faktors. Als Startwert x_1 wurde 2 gewählt. Die in Klammern angegebenen Werte sind eine Mittelung über die entsprechenden Größen bei den zehn nächstgelegenen Werten. Während der gesamten Meßreihe trat keine einzige Endlosschleife auf.

Abbildung 2.3: Laufzeitanalyse der Pollardschen ϱ-Methode

$n = p \cdot q$	Iterationen		n-Schlaufe		p-Schlaufe		q-Schlaufe	
$106 = 2 \cdot 53$	4	(4.3)	11	(7.2)	2	(1.8)	5	(4.1)
$1\,003 = 17 \cdot 59$	8	(5.1)	5	(13.4)	4	(2.2)	1	(9.4)
$10\,001 = 71 \cdot 137$	12	(6.2)	9	(38.2)	4	(2.4)	8	(28.5)
$100\,001 = 11 \cdot 9\,091$	5	(7.7)	35	(88.4)	1	(3.3)	35	(57.6)
$1\,000\,001 = 101 \cdot 99$	28	(11.0)	445	(514.5)	12	(4.0)	74	(282.1)
$10\,000\,001 = 11 \cdot 909\,091$	5	(14.4)	133	(752.8)	1	(4.6)	133	(476.1)
$100\,000\,001 = 17 \cdot 5\,882\,353$	8	(30.5)	2 117	(1 876.1)	4	(10.7)	529	(1 035.6)
$1\,000\,000\,006 = 2 \cdot 500\,000\,003$	4	(27.8)	30 921	(12 276.6)	2	(9.6)	30 921	(5 648.0)

2.3 Auf der Suche nach Kryptomorphismen

Ist R ein Ring und $\sim$ eine *Kongruenzrelation* auf R mit $|W \cap \ker(\sim)| \leq 1$, so wird R nach dem Einbettungssatz (vgl. Anhang B, S. 62) durch $R/_{\ker(\sim)}$ treu modelliert. Weil $\ker(\sim)$ ein Ideal ist, folgt im Fall eines euklidischen Rings unmittelbar

$$\exists\, g \in R: \quad \ker(\sim) = g \cdot R = (g),$$

denn nach Satz 6 (Anhang B, S. 61) ist R dann insbesondere ein Hauptidealring. Es gibt weiter ein Vertretersystem von $R/_{(g)}$, das durch einen maximalen Grad d beschränkt ist, d. h. R ist ist durch $R/_{(g)}$ treu modelliert.

Ist nun $g = p_1^{e_1} \cdot \dots \cdot p_s^{e_s}$ eine Primfaktorzerlegung von g, so gibt es nach dem Chinesischen Restsatz (Anhang B, S. 62) einen Isomorphismus

$$\alpha\colon R/_{(g)} \longrightarrow R/_{(p_1^{e_1})} \times \dots \times R/_{(p_s^{e_s})}.$$

parallele Algorithmen

Diese Tatsache läßt sich leicht zum Entwurf paralleler Algorithmen einsetzen. Statt daß ein Prozessor ein Problem in $R/_{(g)}$ löst, lösen s Prozessoren je ein Teilproblem in $R/_{(p_i^{e_i})}$ $(i = 1, \ldots, s)$. Als Isomorphismus ist α bijektiv, so daß sich die einzelnen Ergebnisse garantiert wieder zu einem Element von $R/_{(g)}$ zusammensetzen lassen (vgl. auch Abschnitt 4.3).

Weitere Anwendungsmöglichkeiten des Chinesischen Restsatzes liefert die Kryptologie:

RSA-Algorithmus

Bsp. 7 (Der RSA-Algorithmus) Seien p und q zwei große[6] Primzahlen und $n = p \cdot q$. Dann gilt $\mathbb{Z}_n \cong \mathbb{Z}_p \times \mathbb{Z}_q$ und $\mathbb{Z}_n^\times \cong \mathbb{Z}_p^\times \times \mathbb{Z}_q^\times$. Es kodiere $m \in \mathbb{Z}_n$ eine Nachricht. Wir betrachten für ein festes $e \in \mathbb{Z}$ mit $0 \leq e < n$ und

$$\mathrm{ggT}(e, (p-1)(q-1)) = 1$$

die Funktion

$$E\colon \mathbb{Z}_n \to \mathbb{Z}_n, \qquad E(m) := m^e.$$

Die Berechnung von $E(m)$ ist mit einem Aufwand von $O((\log n)^2)$ möglich, während zur Berechnung von $E^{-1}(m)$ ohne Kenntnis von p und q keine effizienten Algorithmen bekannt sind (vgl. Abschnitt 2.2).

Kennt man p und q, so kann man wegen $\mathrm{ggT}(e, (p-1)(q-1)) = 1$ Elemente $d, s \in \mathbb{Z}$ bestimmen mit

$$d \cdot e + s \cdot (p-1)(q-1) = 1 \tag{3}$$

(vgl. 2.1 und Algorithmus 3). Nach dem Chinesischen Restsatz (vgl. Anhang B, S. 62) gilt $(m^e)^d \equiv m \bmod n$ genau dann, wenn

$$(m^e)^d \equiv m \bmod p \quad \text{und} \quad (m^e)^d \equiv m \bmod q$$

gelten. Diese Gleichungen sind genau dann erfüllt, wenn auch

$$e \cdot d \equiv 1 \bmod (p-1) \quad \text{und} \quad e \cdot d \equiv 1 \bmod (q-1)$$

trap-door

gilt, und beides ist nach Konstruktion in (3) der Fall. Somit kann man e als *public key* veröffentlichen und d als *trap-door* geheim halten. (Einzelheiten bei [Gei00]) $\square$

Die Sicherheit des in Beispiel 7 dargestellten Verfahrens beruht allein auf der Tatsache, daß E^{-1} wesentlich härter zu berechnen ist als E (wenn man d nicht kennt). Es erscheint also lohnend, vom konkreten Beispiel zu abstrahieren, Funktionen wie E einen eigenen Namen zu geben, nach anderen Beispielen für solche Funktionen zu suchen und die zu Grunde liegenden Verfahren allgemein zu formulieren. Dies soll im Folgenden geschehen.

Krypto-morphismus

Definition Injektive Abbildungen E zwischen Datenstrukturen heißen *Kryptomorphismen*, wenn deren Funktionswerte mit geringem Aufwand berechenbar sind, während E^{-1} nur mit zusätzlichem Wissen leicht berechenbar ist, das der „Eigentümer" von E geheim halten kann.

Einweg-funktion

Bei Abbildungen E zwischen Datenstrukturen, deren Umkehrung in jedem Fall (also insbesondere auch für den Eigentümer) schwer zu berechnen ist, spricht man von *Einwegfunktionen* oder *Komplexitätsdioden*. In diesem Fall verwendet man gelegentlich die Schreibweise $E\colon A \twoheadrightarrow B$. □

Der Rest dieses Abschnitts widmet sich der Suche nach solchen Funktionen, während der folgende Abschnitt sich mit der Verallgemeinerung der Verfahren beschäftigt.

Merkle-Hellman-Knapsack

Bsp. 8 (Merkle-Hellman-Knapsack) Seien $V > 0$ und $v_1, \ldots, v_n > 0$ gegeben. Gesucht sind Zahlen $\lambda_i \in \mathbb{N}_0$, für die $\sum\limits_{i=1}^{n} \lambda_i v_i$ maximal ist, aber durch V beschränkt bleibt. Das Finden solcher Zahlen λ_i ist unter der Bezeichnung *Knapsack-Problem* bekannt. Dieses Problem ist NP-vollständig, und daher liegt es nahe zu versuchen, dies für einen Krypto-Algorithmus auszunutzen.

Der Sender wählt ganze Zahlen $k_0 < k_1 < \cdots < k_{n-1}$, ein primes $p \in \mathbb{Z}$ mit $p > \sum\limits_{i=0}^{n-1} m_i 2^{k_i}$ sowie ein beliebiges $a \in \mathbb{Z}_p^\times$. Als Public-Key werden die Zahlen $\omega_i := a \cdot 2^{k_i}$ verwendet.

Ist $m = \sum\limits_{i=0}^{n-1} m_i 2^i$ eine Nachricht, so wird

$$E(m) := \sum_{i=0}^{n-1} m_i \omega_i$$

übertragen. Nach Wahl der k_i gilt

$$\sum_{i=0}^{l-1} 2^{k_i} \leq 2^{k_l} \qquad (l = 1, \ldots, n-1),$$

weswegen man mit Kenntnis von a und den k_i via

$$a^{-1} E(m) \equiv \sum_{i=0}^{n-1} m_i 2^{k_i} \mod p$$

leicht die ursprüngliche Darstellung von m berechnen kann. Für Angreifer, die a und die k_i nicht kennen, stellt sich die Berechnung der m_i als Knapsack-Problem dar. Da dieses Problem NP-vollständig ist, sollte man das Verfahren als sicher einstufen.

Das Verfahren ist aber *nicht* sicher! Die Definition von NP-Vollständigkeit beruht auf Worst-Case-Komplexitätsbetrachtungen und anscheinend stellen die ω_i nicht den Worst-Case für das Knapsackproblem dar. Tatsächlich wurde im Jahre 1986 der sogenannte *LLL-Algorithmus* entwickelt, der m aus $E(m)$ ohne Kenntnis von a oder $k_0, \ldots, k_{n-1}$ mit einem Aufwand von $O(n^3)$ berechnet. □

[6] „Große" Zahlen sind heute mindestens 600 Bit lang.

Bsp. 9 (Wortproblem) Ist Ω ein Alphabet, so ist Ω^* zusammen mit der Konkatenation $*$ ein Monoid. Durch die Hinzunahme formaler inverser Elemente zu Ω kann Ω^* zu einer Gruppe erweitert werden. Ist nun zusätzlich $\Pi \subseteq \Omega^*$ eine Menge von *Relatoren* und

$$N = \langle \omega r \omega^{-1} : \omega \in \Omega^*, r \in \Pi \rangle,$$

so wird durch $G := {}^{\Omega^*}\!/_{N}$ eine (i. a. nicht-abelsche) Gruppe definiert (zu Relatoren vgl. auch Anhang B, S. 63). Ω und Π seien so gewählt, daß das *Wortproblem* in G algorithmisch schwer zu lösen ist, d. h. es schwer zu bestimmen ist, ob zwei Elemente in G gleich sind.

Wähle nun $\omega_0, \omega_1 \in \Omega$ mit $\langle \omega_0, \omega_1 \rangle \cap N = \{\varepsilon\}$ und wähle als Public Key Ableitungen $\pi_0, \pi_1 \in \langle \omega_0, \omega_1 \rangle$. Sei $m = (m_i)_{i=0}^{n-1}$ eine binär codierte Nachricht (d. h. $m_i \in \{0, 1\}$, $0 \leq i \leq n-1$) und

$$E\colon G \to G \qquad E(m) := \pi_{m_0} * \pi_{m_1} * \cdots * \pi_{m_{n-1}},$$

wobei $E(m)$ in einer Form dargestellt wird, die die Zerlegung in die π_i nicht mehr erkennen läßt. Dies erreicht man durch beliebige Anwendung der Relatoren in Π. Um eine Nachricht m aus $E(m)$ zu bestimmen, muß man zumindest das Wortproblem in G lösen können, was aber nach Voraussetzung schwer ist.

Damit man m mit Kenntnis der geheim gehaltenen Wörter ω_0 und ω_1 effizient berechnen kann, braucht man eine Untergruppe $\widetilde{N} \leq G$ mit $\langle \omega_0, \omega_1 \rangle \cap \widetilde{N} = \{\varepsilon\}$, in der die Zerlegung eines Wortes in ω_0 und ω_1 einfach ist. Für den kanonischen Homomorphismus $\varphi\colon G \to {}^{G}\!/_{\widetilde{N}}$ gilt $\varphi(\omega_0) \neq \varphi(\omega_1)$.

Um die Sicherheit weiter zu erhöhen, wäre wünschenswert, daß es nur sehr wenige Untergruppen von G gäbe, die als $\widetilde{N}$ verwendet werden können, d. h. für die meisten Untergruppen $\widetilde{N} \leq G$ gelte $\omega_0 \equiv \omega_1 \bmod \widetilde{N}$.

Konkrete Beispiele für Konstruktionen dieser Art sind derzeit noch nicht bekannt. □

DIFFIE-HELLMAN-Protokoll

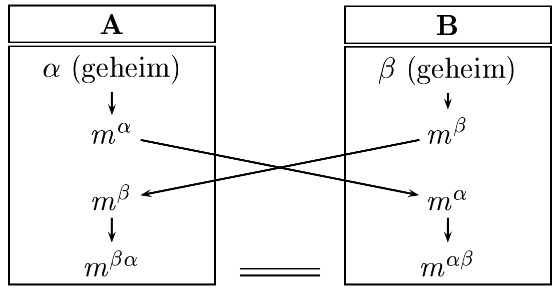
Bsp. 10 (DIFFIE-HELLMAN-Protokoll) Das Protokoll von DIFFIE und HELLMAN eignet sich dazu, zwischen zwei Parteien A und B auf sichere Weise gemeinsame Schlüssel herzustellen, ohne dabei eine abhörsichere Verbindung vorauszusetzen. A und B einigen sich zuvor auf eine zyklische multiplikative Gruppe C (z. B. $C := \mathbb{Z}_p^\times$) und einen Erzeuger $m \in C$.

A wählt nun ein geheimes Element $\alpha \in \{0, \ldots, |C|-1\}$ und überträgt m^α. B wählt $\beta \in \{0, \ldots, |C|-1\}$ und überträgt m^β. Durch Austausch der öffentlichen Schlüssel m^α und m^β erhalten A und B aus

$$(m^\alpha)^\beta = m^{\alpha \cdot \beta} = m^{\beta \cdot \alpha} = (m^\beta)^\alpha$$

einen gemeinsamen geheimen Schlüssel (vgl. Abb. 2.4) □

Abbildung 2.4: Diffie-Hellman-Protokoll



Das Verfahren von Diffie und Hellman (Bsp. 10) beruht darauf, daß der diskrete Logarithmus schwer zu berechnen ist. Man ist darum an Gruppen interessiert, in denen das diskrete Logarithmieren besonders schwer ist. $(\mathbb{Z}_n, +)$ ist offensichtlich kein gutes Beispiel für solch eine Gruppe. Ist p groß und prim, dann kostet in $(\mathbb{Z}_p, \cdot)$ der diskrete Logarithmus nach bisherigem Kenntnisstand $\mathrm{e}^{O(1)\sqrt[3]{\log n(\log\log p)^2}}$ Schritte. Es gibt aber noch „schlimmere" Gruppen. Auf *elliptischen Kurven* kann man eine Addition „+" definieren, die eine Gruppe $(EC, +)$ definiert (vgl. Anhang B, S. 64). In dieser Gruppe kostet Logarithmieren zum Teil gar $\sqrt[2]{|EC|}$ Schritte. Dies führt dazu, daß man bei einer Verwendung von 150 Bit-Zahlen Logarithmierungskomplexitäten erhält, für die man in $(\mathbb{Z}_p, \cdot)$ 1 000 Bit-Zahlen verwenden müßte.

elliptische Kurven

2.4 Public-Key-Infrastrukturen

Als *Public-Key-Infrastrukturen* bezeichnet man die Anwendung eines Kryptomorphismus in einer Weise, daß mit der öffentlichen Version einer Datenstruktur gearbeitet werden kann, aber die Inversion ohne einen *trap-door*-Schlüssel praktisch unmöglich ist.

Public-Key-Infrastruktur

Nachdem in Abschnitt 2.3 konkrete Beispiele für Kryptomorphismen genannt wurden, sollen hier Kryptoverfahren in allgemeinerer Form formuliert werden.

Im Folgenden sei $E\colon G \to G$ ein Kryptomorphismus und $D = E^{-1}$. Dabei ist stets vorausgesetzt (Definition des Kryptomorphismus), daß D nur bei Kenntnis von Geheimnissen in vernünftiger Zeit berechenbar ist. In diesem Fall bezeichnet man G auch als *sichere Gruppe* (bzw. allgemein als sichere Datenstruktur).

sichere Gruppe

Encryption

Man stelle sich einen Pool von adressierten Truhen vor. Jeder Truhe sei ein geöffnetes Vorhängeschloß beigelegt, jedoch befinde sich der zugehörige Schlüssel beim Adressaten. Um in dieser Situation eine Nachricht sicher zu verschicken, genügt es, sie in die richtige Truhe zu legen, deren Schloß zuschnappen zu lassen und die Truhe abzuschicken. Findet die Truhe den Weg zum Empfänger, so kann dieser sie öffnen und die Nachricht lesen, während allen anderen die Nachricht verborgen bleibt.

Dieses Verfahren wird mathematisch nachgemacht. Ist m eine zu verschickende Nachricht, so entspricht $E(m)$ der Nachricht in der verschlossenen Truhe und D entspricht dem Schlüssel, der nur dem Adressaten bekannt ist (Abb. 2.5).

Abbildung 2.5: Encryption

Sender — Empfänger

$$\boxed{\text{Nachricht } m} \xrightarrow{\quad E(m) \quad} \boxed{D(E(m)) = m}$$

totale Anonymität

Wenn es gelänge, D und E in der gleichen Geschwindigkeit zu berechnen, mit der Daten über ein Netz übertragen werden können, könnte man *totale Anonymität* in diesem Netz erzeugen. Dann würden einfach alle Pakete nach dem obigen Schema verschlüsselt und „nach überall" verschickt. Jeder Netzteilnehmer bekäme so eine Flut von Paketen, die er alle mit seinem geheimen Schlüssel D zu öffnen versucht. All jene Pakete, die er öffnen kann, waren für ihn bestimmt.

Zu bemerken ist jedoch, daß eine Realisierung die Leistung des Netzes stark beeinträchtigen würde, da jedes Paket an jeden Teilnehmer versandt werden muß.

Authentifikation und Signatur

Ein ähnliches Verfahren läßt sich zur Authentifikation einsetzen. Hier wendet der Sender seine geheime Funktion D auf m an und versendet $(m, D(m))$ (vgl. Abb. 2.6). Der Empfänger berechnet $E(D(m))$ mit der öffentlich bekannten, bijektiven Funktion E des Senders und kann im Fall $E(D(m)) = m$ ziemlich sicher sein, daß der Sender der ist, für den er sich ausgibt.

Abbildung 2.6: Authentifikation

Sender — Empfänger

$$\boxed{\text{Nachricht } m} \xrightarrow{\quad (m, D(m)) \quad} \boxed{E(D(m)) \stackrel{?}{=} m}$$

Ein anderer Ansatz für eine digitale Signatur:

El-Gamal-Signatur

Bsp. 11 (Die El-Gamal-Signatur) Sei $\alpha \in \mathbb{Z}_p^\times$ von hoher Ordnung und $E\colon G \twoheadrightarrow G$, $E(x) = \alpha^x$. Weiter sei $N = \{0, \dots, p-2\}$.

Der Sender wählt ein geheimes Element $x \in N$ und veröffentlicht $E(x)$. Zur Signatur einer Nachricht $m \in N$ wird ein zufälliges $k \in N$ gewählt und $r := E(k)$ gesetzt. k sei invertierbar modulo $p-1$. Dann wird das eindeutig bestimmte $s \in N$ berechnet, für das gilt

$$m \equiv r \cdot x + k \cdot s \bmod p - 1.$$

Als signierte Nachricht wird das Tupel (m, r, s) übertragen.

Der Empfänger verifiziert den Absender dann dadurch, daß er prüft, ob

$$E(x)^r \cdot r^s \stackrel{?}{\equiv} E(m) \bmod p$$

gilt, denn dies ist wegen

$$E(x)^r \cdot r^s \equiv (\alpha^x)^r \cdot (\alpha^k)^s \equiv \alpha^{x \cdot r + k \cdot s} = \alpha^m \bmod p$$

zu erwarten. □

Das folgende Beispiel zeigt, daß man die spezielle Einwegfunktion

$$E\colon G \to G, \quad E(x) := \alpha^x \qquad (\alpha \in G \text{ fest})$$

(G eine sichere Gruppe) auch zum chiffrieren verwenden kann:

Bsp. 12 (El-Gamal-Chiffrierverfahren) Der Empfänger wählt ein y und veröffentlicht $Y := \alpha^y$. Zum Senden einer Nachricht m wählt der Sender ein zufälliges k, setzt $z := Y^k$, $r := \alpha^k$ und verschickt $(m \cdot z, r)$. Der Empfänger kann m wegen

El-Gamal-Chiffrierverfahren

$$m = \frac{m \cdot z}{z} = \frac{m \cdot z}{Y^k} = \frac{m \cdot z}{\alpha^{y \cdot k}} = \frac{m \cdot z}{r^y}$$

mit Hilfe seines geheimen y rekonstruieren. □

Bei den Verfahren der Beispiele 11 und 12 ist zu beachten, daß nach Möglichkeit jedesmal ein anderes Element k zu wählen ist.

Bsp. 13 Global seien Zahlen p, q, g und h wie folgt vorgegeben:

- p ist prim und sehr groß.
- q ist ein Primfaktor von $p - 1$.
- $h < p - 1$ und $g := h^{(p-1)/q} \neq 1 \bmod p$.

$H\colon \mathbb{Z}_q \to \mathbb{Z}_q$ sei eine kryptographische Hashfunktion.

Dem Teilnehmer A wird zur Identifikation ein privater Schlüssel $x \in \mathbb{Z}_q$ zugeordnet. Als öffentliches Gegenstück zu diesem Schlüssel dient $y := g^x$. Um eine Nachricht $m \in \mathbb{Z}_q$ zu signieren, wählt A ein zufälliges Element $k \in \mathbb{Z}_q$ und bestimmt

$$r := (g^k \bmod p) \bmod q, \qquad s := (k^{-1}(H(m) + x \cdot r)) \bmod q.$$

Als signierte Nachricht wird dann (m, r, s) verschickt.

Der Empfänger bestimmt $w = s^{-1} \bmod q$ und

$$v := \Big((g^{(H(m) \cdot w) \bmod q} \cdot y^{(r \cdot w) \bmod q}) \bmod p\Big) \bmod q.$$

Die Signatur ist verifiziert, wenn $v = r$ gilt. □

Die „Paßbehörde“

Keines der vorgestellten Protokolle ist in der angegebenen Form sicher vor der *man-in-the-middle-attack*: Ein unabhängiger Dritter kann sich für den Sender ausgeben und von seinem eigenen E behaupten, es sei das E des wirklichen Senders. Dann kann der Empfänger diesen Dritten nicht vom wirklichen Sender unterscheiden, woran auch Encryption und Signatur nichts ändern.

man-in-the-middle-attack

Dies kann umgangen werden, indem man eine globale Zentrale einrichtet, die die Zuordnungen der Funktionen E zu den Benutzern verwaltet. Eine solche

SKIA

Zentrale wird als *SKIA* (secure key issuing authority, auch *CA* für certification authority) oder *Paßbehörde* bezeichnet. Jeder Netzbenutzer muß sich bei dieser Behörde anmelden und bekommt E und D zugeteilt (bzw. läßt sich seine eigenen Konstruktionen zertifizieren), nachdem er sich auf konventionelle Art und Weise identifiziert hat (etwa durch einen Personalausweis).

Selane

Es ist möglich, mehrere Authentifikationszentralen in einer Hierarchie zu etablieren. Im *Selane*-Projekt des E. I. S. S. wird ein Protokoll zur Verfügung gestellt, das eine SKIA der obersten Ebene hat, wobei jeder Benutzer, der sich bei dieser SKIA anmeldet, gleichzeitig die Funktion einer „Unter-SKIA" übernehmen kann. Zwei Netzteilnehmer kommunizieren dann unter Verwendung der nächsthöheren gemeinsamen SKIA als Authentifikationszentrale. Für nähere Informationen sei auf den Artikel [$\beth$95] verwiesen.

Einen tieferen Einblick in die hier vorgestellten Kryptoideen vermittelt [Gei00].

3 Der Rijndael-Algorithmus

3.1 Vorbemerkung

Am 2. Oktober 2000 wurde vom NIST der *Advanced Encryption Standard* (AES) veröffentlicht, der den bisher gebräuchlichen *Data Encryption Standard* (DES) in Zukunft ablösen soll.

Advanced Encryption Standard

Abbildung 3.1 zeigt eine verkürzte historische Übersicht über das Auswahlverfahren.

Abbildung 3.1: Auswahlverfahren des AES

02. 01. 97	Internationaler Aufruf des NIST, bis zum 12. 09. 97 Algorithmenvorschläge für einen neuen Encryption Standard einzureichen.
20. 08. 98	15 ausgewählte Algorithmen werden veröffentlicht mit der Einladung zur Kommentierung.
22. 03. 99	Die ausgewählten Algorithmen werden auf einer Konferenz in Rom diskutiert.
15. 04. 99	Aus den 15 Algorithmen werden fünf verbleibende ausgewählt, die bis zum 15.05.2000 nochmals international kommentiert werden können.
07. 08. 00	Die NSA entwickelt VHDL-Modelle zur Hardware-Performance-Evaluation.
02. 10. 00	Die Entscheidung fällt auf den Rijndael-Algorithmus.

Der neue Standard beruht auf einem Verfahren, das in Belgien von Daemen und Rijmen entwickelt wurde und als *Rijndael-Algorithmus* bezeichnet wird. Die Grundideen dieses Verfahrens sollen in diesem Abschnitt grob skizziert werden. Für eine ausführliche Darstellung sei auf [DaRi00] verwiesen.

Rijndael-Algorithmus

3.2 Datenstrukturen

Beim Rijndael-Algorithmus handelt es sich um ein symmetrisches *Blockchiffre*. Die Größe der Blöcke sowie die Länge des Schlüssels können (unabhängig voneinander) zu 128, 192 oder 256 Bit festgelegt werden.

Im Feld *state* werden je vier Byte eines Blocks derart zu einer Spalte zusammengefaßt, daß $state[i][j]$ das $(i + 4 \cdot j)$-te Byte des Blocks bezeichnet. Damit ist *state* ein zweidimensionales Array der Größe $4 \times b$, wobei b von der Größe des Blocks abhängt ($b = \frac{1}{32} \cdot$ Blocklänge).

Analog wird der Schlüssel in einem zweidimensionalen Feld *key* der Größe $4 \times k$ abgelegt. Allerdings wird dieser Schlüssel nicht selbst verwendet, sondern aus *key* wird ein größeres Feld *expandedKey* berechnet, das (theoretisch) die Größe $4 \times \infty$ hat, wobei für jede Runde (s. u.) des Algorithmus ein Stück der Größe $4 \times b$ von diesem Feld genommen wird.

Die 4-Tupel eines Feldes werden im Folgenden als Spalten bezeichnet, die Tupel der anderen Dimension als Zeilen.

3.3 Der Algorithmus

Der Algorithmus besteht aus einer gewissen Anzahl r von Runden, in denen das Feld *state* in immer der gleichen Weise manipuliert wird. Die Rundenanzahl r ist bestimmt durch b und k.

Algorithmus 3.1: Der Rijndael-Algorithmus

```
procedure rijndael(byte[][] state, byte[][] key)
  byte[][] expandedKey = expand(key);
  state = state ⊕ selectKey(expandedKey, 0);
  for i = 1 to r
    byteSubstitution(state);
    shiftRows(state);
    if i < r then
      mixColumns(state);
    state = state ⊕ selectKey(expandedKey, b · i);
```

Die Funktion *expand* erweitert den Schlüssel *key* nach folgendem Prinzip: Die ersten k Spalten entsprechen dem Schlüssel *key*. Die nachfolgenden Spalten werden danach jeweils aus den vorangehenden k Spalten berechnet. Da insgesamt $r+1$ mal auf *expandedKey* zugegriffen wird und bei jedem Zugriff b Spalten verbraucht werden, müssen insgesamt $b \cdot (r+1)$ Spalten von *expandedKey* berechnet werden:

Algorithmus 3.2: Schlüsselexpansion

```
function expand(byte[][] key)
  for i = 0 to k − 1
    expandedKey[i] = key[i];
  for i = k to b · (r + 1)
    tmp = shake(expandedKey[i − 1][0]) ⊕ x^i;
    for j = 1 to 3
      expandedKey[i][j] = expandedKey[i − k][j] ⊕ tmp;
  return expandedKey;
```

Der Aufruf *shake* bewirkt eine Permutation der Bits des übergebenen Bytes, auf die hier nicht näher eingegangen werden soll, und x ist eine beliebige Konstante mit $0 \neq x \neq 1$.

Die Schlüssel-Expansion in dieser Form kann vorausberechnet werden. Außerdem ist zu bemerken, daß das Expansionsverfahren vom Rest des Algorithmus unabhängig ist, so daß es problemlos durch andere Verfahren ersetzt werden könnte. Ignoriert man die AES-Spezifikation, die die Schlüsselgröße vorschreibt, kann man z. B. als expandierten Schlüssel auch Sequenzen einer Zufallsfolge verwenden, wodurch das Verfahren insgesamt sicherer werden dürfte.

Die nächste Funktion in Algorithmus 3.1 ist *selectKey*. Sie liefert ein Feld von der gleichen Größe wie *state*, das mit dem angegebenen Offset aus *expandedKey* ausgeschnitten wird, also etwa wie folgt:

Algorithmus 3.3: Bestimmung des Rundenschlüssels aus *expandedKey*

```
function selectKey(byte[][] expandedKey, offset)
  for i = 0 to b
    selectedKey[i] = expandedKey[i + offset];
  return selectedKey;
```

Jeder Eintrag des Feldes *state* wird nun mit dem entsprechenden Eintrag des von *selectKey* gelieferten Schlüssels mit $\oplus$ verknüpft.

Runden

Nach diesem „Vorspiel“ beginnen nun die eigentlichen *Runden* des Rijndael-Algorithmus. Jede Runde besteht aus je einem Aufruf von *byteSubstitution*, *shiftRows*, *mixColumns* und der Addition eines weiteren Teils von *expandedKey*. Der Aufruf von *mixColumns* unterbleibt im letzten Durchlauf vermutlich aus Gründen der Asymmetrie.

Die Funktion *byteSubstitution* ersetzt zunächst jeden Eintrag in *state*, aufgefaßt als Element von $\mathbb{F}_{2^8} = \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$, durch sein multiplikatives Inverses

$$state[i][j] := (state[i][j])^{-1}$$

(mit $0^{-1} := 0$). Danach wird darauf eine feste affine Abbildung ausgeführt, wobei nun die Bytes als Elemente von $\mathbb{F}_2^8$ aufgefaßt werden:

$$state[i][j] := \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot state[i][j] + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}.$$

shiftRows bewirkt eine zyklische Verschiebung der Zeilen von *state* um einen Betrag, der von der Zeile und der Blockgröße abhängt.

Schließlich werden in *mixColumns* noch die Spalten in *state* manipuliert. Dazu wird jede Spalte als Element von $\mathbb{F}_{2^8}[x]/(x^4 + 1)$ aufgefaßt und mit dem festen Polynom

$$c(x) = 3x^3 + x^2 + x + 2$$

multipliziert. Da c und $x^4 + 1$ koprim sind, ist die Multiplikation invertierbar. Es gilt:

$$c^{-1}(x) = 11x^3 + 13x^2 + 9x + 14.$$

4 Algorithmenanalyse und Klassen von Algorithmen

4.1 Amortisierte Analyse

Bekanntlich kann man die Laufzeit eines Algorithmus dadurch nach oben abschätzen, daß man die Laufzeit des Algorithmus im schlimmsten Fall, d. h. im *worst case* betrachtet. Diese Laufzeit wird als *worst-case-Komplexität* bezeichnet.

worst-case-Komplexität

Viele Funktionen bestehen im wesentlichen aus einer Folge von Funktionen niedrigeren Abstraktionsniveaus. So wird bei square-and-multiply (Algorithmus 1.2) die Exponentiation auf eine Folge von Multiplikationen zurückgeführt, wobei die Multiplikation ihrerseits möglicherweise mittels shift-and-add (Algorithmus 1.1) als Folge von Additionen und Shifts dargestellt wird. Ist man an der Anzahl der Additionen interessiert, die in diesem Fall eine Exponentiation kostet, so ist es ungerechtfertigt, im shift-and-add-Algorithmus für jede Addition die größtmögliche Laufzeit anzunehmen. Sinnvoller ist es zu versuchen, die Additionsfolge „als Ganzes" zu untersuchen, um so eine schärfere obere Schranke für die worst-case-Komplexität zu erhalten. Dieser Ansatz wird als *amortisierte Analyse* bezeichnet.

Definition Ein Prozeß bestehe aus einer Folge von n Grundoperationen und habe die Gesamtkomplexität $\varkappa(n)$. Dann heißt

amortisierte Komplexität

$$\alpha(n) := \frac{\varkappa(n)}{n}$$

die *amortisierte Komplexität* einer Grundoperation. □

Bsp. 14 (Inkrementieren eines Bit-Strings) Betrachte ein Feld a von $\log n$ booleschen Werten, die eine Ganzzahl binär codieren. a sei mit 0 initialisiert und soll schrittweise durch folgende Funktion inkrementiert werden, bis a den Wert $n-1$ enthält.[7]

Algorithmus 4.1: Inkrementieren eines Bit-Strings

```
procedure increment(a[])
  i = 0;
  while i < a.length and a[i] = 1 do
    a[i] = 0; i = i + 1;
  if i < a.length then
    a[i] = 1;
```

Offensichtlich hat dieser Algorithmus eine worst-case-Komplexität von $O(\log n)$.

[7] Man wird solche Bit-Operationen heute kaum noch aus Laufzeit-Gründen optimieren. Trotzdem lohnt sich Optimierung auf dieser Ebene, nämlich um die bei Schaltvorgängen freiwerdende Wärme zu reduzieren.

Sei $d_H\colon \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ die Funktion, die je zwei natürlichen Zahlen die Hammingdistanz[8] ihrer Binärdarstellung zuordnet. In der folgenden Rechnung werden Zahlen mit ihrer Binärdarstellung identifiziert, insbesondere meint $i[k]$ das k-te Bit der Binärdarstellung von i. Für die Laufzeit $\varkappa(n)$ gilt dann:

$$\varkappa(n) = \sum_{i=0}^{n-1} d_H(i, i+1) = \sum_{i=0}^{n-1} \sum_{k=0}^{\log n} d_H(i[k], (i+1)[k]) = \sum_{k=0}^{\log n} \frac{n}{2^k} \leq 2n.$$

Damit gilt für die amortisierte Komplexität einer Inkrementierung

$$\alpha(n) = \frac{\varkappa(n)}{n} \leq 2 = O(1).$$

□

Zur Bestimmung der amortisierten Komplexität gibt es verschiedene Methoden.

Aggregations-methode

Die gerade verwendete Methode wird als *Aggregationsmethode* (*aggregate method*) bezeichnet: Man bestimmt eine obere Schranke $\varkappa(n)$ für die Summe der Kosten von n Operationen und erhält

$$\alpha(n) = \frac{\varkappa(n)}{n}$$

als amortisierte Komplexität.

Buchungs-methode

Einen etwas anderen Ansatz bietet die *Buchungsmethode* (*accounting method*). Hierbei stellt man sich vor, es handelt sich bei den Kosten von Operationen tatsächlich um finanzielle Kosten, z. B. ein Euro pro Operation. Statt nun eine Operation dann zu „bezahlen“, wenn sie durchgeführt werden soll, kann man billige Operationen etwas verteuern, um auf teurere Operationen zu „sparen“.

Bsp. 15 Ein spezieller Stack stelle neben den Operationen *push* und *pop* noch die Operation *multipop* zur Verfügung, mit der eine angegebene Anzahl von Elementen gleichzeitig vom Stack genommen werden kann. Dies sei implementiert durch einen iterierten Aufruf der elementaren Methode *pop*.

Ordnet man *push* und *pop* jeweils die Kosten 1 zu, so hat *multipop* eine worst-case-Komplexität von $O(n)$.

Beachtet man jedoch, daß man nur höchstens so viele *pop*- wie *push*-Aufrufe durchführen kann, so kann man die Kosten wie folgt umverteilen: *push* koste 2 Einheiten und *pop* sowie *multipop* werden als kostenlos modelliert. Die Idee dabei ist, beim *push*en eines Objekts auf den Stack das *pop*pen gleich mitzubezahlen.

Offensichtlich ergibt sich so für *push*, *pop* und *multipop* eine amortisierte Komplexität von $O(1)$. □

Eine dritte Ansatzweise zur Bestimmung der amortisierten Komplexität ist die *Potentialmethode*.

Potential-methode

[8] Die *Hammingdistanz* von $a, b \in \mathbb{F}_2^n$ ist die Anzahl der Kanten des kürzesten Pfades, durch den a und b auf dem Hyperkubus $\mathbb{F}_2^n$ miteinander verbunden werden können. Dies ist genau die Anzahl der Dimensionen, in denen die Koordinaten von a und b nicht übereinstimmen.

Man betrachtet verschiedene Zustände D_i der Datenstruktur, in Beispiel 15 etwa die Anzahl der Elemente, die sich gerade auf dem Stack befinden. Für den Übergang von Zustand D_i in Zustand D_{i+1} seien c_i die tatsächlichen Kosten einer Operation. Weiter sei $\Phi\colon \{D_0, D_1, \dots\} \to \mathbb{R}$ eine *Potentialfunktion*, über die via

Potential-
funktion

$$\widehat{c}_i := c_i + \Phi(D_i) - \Phi(D_{i-1}), \qquad \widehat{c}_0 := \Phi(D_0)$$

die amortisierten Kosten des Zustandsübergangs $D_i \to D_{i+1}$ definiert werden. Ziel ist es, Φ so zu wählen, daß sich die amortisierten Kosten $\widehat{c}_i$ für große i möglichst gut den tatsächlichen Kosten c_i annähern. Für die *amortisierten Gesamtkosten* A gilt stets

$$A(n) = \sum_{i=1}^{n} \widehat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).$$

Diese Kosten bilden dann eine obere Schranke für die tatsächlichen Kosten, wenn $\Phi(D_n) \geq \Phi(D_0)$ gilt. In diesem Fall gilt insbesondere $\alpha(n) = \frac{1}{n}A(n)$ für die amortisierten Kosten pro Operation.

Die Potentialmethode kann als Verallgemeinerung des in Beispiel 15 dargestellten Vorgehens verstanden werden: Es bedeutet $\Phi(D_i) > \Phi(D_{i-1})$, daß etwas „vorausbezahlt“ wird und $\Phi(D_i) < \Phi(D_{i-1})$, daß das Guthaben abgeschmolzen bzw. ein Kredit aufgenommen wird.

Die folgenden Beispiele zeigen eine umfangreiche amortisierte Analyse mit Hilfe der Potentialmethode.

Bsp. 16 (dynamische Listen) Wir betrachten eine Datenstruktur T zur Verwaltung von Elementen. Die Methode *insert* dient zum Einfügen von Elementen in T. Intern verfüge T über ein Feld der festen Größe $T.size$, in dem die eingefügten Objekte abgelegt werden. $T.num$ bezeichne die Anzahl der aktuell in T enthaltenen Objekte. Ist $T.num = T.size$, so wird bei erneutem Aufruf von *insert* das Datenfeld durch ein neues Feld der Größe $2 \cdot T.size$ ersetzt (vgl. Algorithmus 4.2). Aufgrund des dann nötigen Umkopierens gilt für die Kosten c_i der Operation *insert*

$$c_i = \begin{cases} i & \text{falls } i-1 \text{ eine Potenz von 2 ist} \\ 1 & \text{sonst} \end{cases}$$

Es sei

$$\Phi(T) := 2 \cdot T.num - T.size$$

die Potentialfunktion. Dann gilt unmittelbar nach einer Tabellenerweiterung $\Phi(T) = 0$ und unmittelbar vorher $\Phi(T) = T.num$. Setzt man

$$r(T) := \frac{T.num}{T.size},$$

so gilt $r(T) \geq \frac{1}{2}$ für jedes T (wegen $T.num \geq \frac{1}{2}T.size$). Insbesondere folgt $\Phi(T) \geq 0$ für alle T und schließlich

$$A(n) = \sum_{i=0}^{n} \widehat{c}_i \geq \sum_{i=0}^{n} c_i = \varkappa(n).$$

Damit majorisieren die amortisierten Kosten die tatsächlichen Kosten.

Sei nun T_i der Zustand der Liste nach dem i-ten Aufruf von *insert*. Abkürzend werden im Folgenden die Schreibweisen $s_i := T_i.size$, $n_i := T_i.num$, $r_i := r(T_i)$ und $\Phi_i := \Phi(T_i)$ verwendet.

Wir betrachten nun den i-ten Aufruf von *insert*. Falls bei diesem Aufruf *keine* Tabellenerweiterung stattfindet, gilt $s_i = s_{i-1}$ und darum

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1}\\ &= 1 + (2n_i - s_i) - (2n_{i-1} - s_{i-1})\\ &= 1 + (2n_i - s_i) - (2(n_i - 1) - s_i) = 3.\end{aligned}$$

Im anderen Fall, also wenn eine Erweiterung nötig ist, gilt $\frac{1}{2}s_i = s_{i-1} = n_i - 1 = n_{i-1}$ und damit

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1}\\ &= n_i + (2n_i - s_i) - (2n_{i-1} - s_{i-1})\\ &= n_i + (2n_i - (2n_i - 2)) - ((2n_i - 2) - (n_i - 1)) = 3.\end{aligned}$$

Damit ist die amortisierte Komplexität von *insert* konstant 3. □

Algorithmus 4.2: Einfügen nach Beispiel 16

```
procedure insert(Element x)
  if size = 0 then
    array = new Element[1];
  else if num = size then
    newArray = new Element[2 · size];
    for i = 0 to size − 1
      newArray[i] = array[i];
    array = newArray;
    size = 2 · size;
  array[num] = x;
  num = num + 1;
```

Bsp. 17 (dynamische Listen – Fortsetzung) Die Datenstruktur des vorangegangenen Beispiels soll um eine Methode *delete* erweitert werden, die das Löschen von Objekten aus der Liste erlaubt.

Damit T nicht unnötig viel Speicherplatz belegt, soll bei wiederholtem Löschen von Elementen das Datenfeld durch ein geeignetes kleineres Feld ersetzt werden. Es mag naheliegend erscheinen, analog zum Einfügen mit *insert* die Methode *delete* so zu implementieren, daß $T.size$ dann halbiert wird, wenn $r(T) < \frac{1}{2}$, also $T.num < \frac{1}{2}T.size$ gilt. Dies ist aber „nicht kompatibel“ zur Implementierung von *insert*. Es könnte nach (un-)geeigneter Vorgeschichte eine Situation entstehen, in der bei wiederholtem doppelten Einfügen und doppeltem Löschen das Feld ständig vergrößert und verkleinert wird, so daß die amortisierten Kosten für diesen Fall $O(n)$ betrügen.

Geschickter ist es, wenn man erst bei $r(T) < \frac{1}{4}$ zu einem kleineren Feld übergeht. Dann sind die amortisierten Kosten für *insert* und *delete* konstant, wie im folgenden dargelegt wird.[9]

Sei Φ nun wie folgt definiert:

$$\Phi(T) := \begin{cases} 2 \cdot T.num - T.size & \text{falls } r(T) \geq \frac{1}{2} \\ \frac{1}{2} \cdot T.size - T.num & \text{falls } r(T) < \frac{1}{2} \end{cases}$$

Dann gilt unter anderem $\Phi(T) = T.num$ im Fall $r(T) = 1$, $\Phi(T) = 0$ im Fall $r(T) = \frac{1}{2}$ und $\Phi(T) = T.num$ im Fall $r(T) = \frac{1}{4}$.

Fall 1: Die i-te Operation ist *insert*.
Fall 1.a: $r_{i-1} \geq \frac{1}{2}$. Wie in Beispiel 16 gezeigt, gilt in diesem Fall $\widehat{c}_i = 3$.
Fall 1.b: $r_{i-1} < \frac{1}{2}$. Es findet keine Expansion statt, d. h. es gilt $s_i = s_{i-1}$, $n_i = n_{i-1} + 1$.
Fall 1.b.(i): $r_i < \frac{1}{2}$. Dann gilt

$$\begin{aligned} \widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (\tfrac{1}{2}s_i - n_i) - (\tfrac{1}{2}s_{i-1} - n_{i-1}) \\ &= 1 + (\tfrac{1}{2}s_i - n_i) - (\tfrac{1}{2}s_i - (n_i - 1)) = 0. \end{aligned}$$

Fall 1.b.(ii): $r_i \geq \frac{1}{2}$. Mit $n_{i-1} \leq \frac{1}{2}s_{i-1}$ (wegen $r_{i-1} < \frac{1}{2}$) gilt:

$$\begin{aligned} \widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2n_i - s_i) - (\tfrac{1}{2}s_{i-1} - n_{i-1}) \\ &= 1 + (2(n_{i-1} + 1) - s_{i-1}) - (\tfrac{1}{2}s_{i-1} - n_{i-1}) \\ &= 3 + 3n_{i-1} - \tfrac{3}{2}s_{i-1} \leq 3. \end{aligned}$$

Fall 2: Die i-te Operation ist *delete*.
Fall 2.a: $r_{i-1} \geq \frac{1}{2}$. In diesem Fall findet keine Kontraktion statt, d. h. es gilt $s_i = s_{i-1}$ und $n_i = n_{i-1} - 1$.
Fall 2.a.(i): $r_i < \frac{1}{2}$. Mit $n_i \geq \frac{1}{2}s_i - 1$ (wegen $r_{i-1} \geq \frac{1}{2}$) gilt:

$$\begin{aligned} \widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (\tfrac{1}{2}s_i - n_i) - (2n_{i-1} - s_{i-1}) \\ &= 1 + \tfrac{1}{2}s_{i-1} - (n_{i-1} - 1) - 2n_{i-1} + s_{i-1} \\ &= 2 + \tfrac{3}{2}s_{i-1} - 3n_{i-1} \leq 2 \end{aligned}$$

Fall 2.a.(ii): $r_i \geq \frac{1}{2}$. Dann gilt

$$\begin{aligned} \widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2n_i - s_i) - (2n_{i-1} - s_{i-1}) \\ &= 1 + 2n_i - s_i - 2(n_i + 1) + s_i = -1. \end{aligned}$$

[9] Wie ändert sich die Komplexität, wenn man die Schwelle r kontinuierlich von $\frac{1}{2}$ nach $\frac{1}{4}$ verschiebt?

Fall 2.b: $r_{i-1} < \frac{1}{2}$.
Fall 2.b.(i): $r_i < \frac{1}{2}$ (keine Kontraktion).

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1}\\ &= 1 + (\tfrac{1}{2}s_i - n_i) - (\tfrac{1}{2}s_{i-1} - n_{i-1})\\ &= 1 - (n_{i-1} - 1) + n_{i-1} = 2.\end{aligned}$$

Fall 2.b.(ii): $r_i \geq \frac{1}{2}$ (Kontraktion). Es gilt $c_i = n_i$, $s_{i-1} = 2s_i$, $n_i = \frac{1}{2}s_i$, $n_{i-1} = n_i + 1$ und somit

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1}\\ &= n_i + (2n_i - s_i) - (\tfrac{1}{2}s_{i-1} - n_{i-1})\\ &= 3n_i - 2s_i + n_i + 1 = 2s_i - 2s_i + 1 = 1.\end{aligned}$$

Insgesamt hat in der dynamischen Listenverwaltung jede Operation eine amortisierte Komplexität von $O(1)$. □

Die hier vorgestellten Beispiele sind entnommen aus dem Kapitel über amortisierte Analyse des Buchs [Cor90].

4.2 Raffke-Algorithmen[10]

Inzidenz-
struktur

Es sei E eine endliche Menge und $U \subseteq \mathcal{P}(E)$. $T := (E, U)$ heißt *Inzidenzstruktur* und die Elemente von U heißen *zulässig* bezüglich T.

T heißt *monoton gefiltert* oder einfach *Filter*[11], falls gilt $\varnothing \neq U$ und

$$\forall\, B \in U\ \forall\, A \in \mathcal{P}(E): \quad A \subseteq B \implies A \in U.$$

Der Definitionsbereich einer Gewichtsfunktion $w\colon E \to \mathbb{R}$ ($E \notin U$) sei durch

$$w\colon \mathcal{P}(E) \to \mathbb{R}, \qquad w(X) = \sum_{x \in X} w(x)$$

auf $\mathcal{P}(E)$ fortgesetzt.

Optimierungs-
problem

Das *Optimierungsproblem* auf T zur Gewichtsfunktion w ist wie folgt definiert: Finde unter allen Mengen $X \in U$, für die $w(X)$ maximal wird, eine Menge M mit einer maximalen Anzahl von Elementen:

$$M \in \{X \in \{X \in U : w(X) \text{ ist maximal}\} : |X| \text{ ist maximal}\}$$

Dieses Problem löst der *kanonische Raffke-Algorithmus*:

[10] auch *Greedy-Algorithmen* genannt
[11] Gelegentlich definiert man Filter auch mit mit umgekehrtem Inklusionszeichen

Algorithmus 4.3: Der kanonische Raffke-Algorithmus

```
function raffke(IncidenceStructure (E, U), Function w)
  sort(E, w);
  // Nun gelte w(e_1) ≥ w(e_2) ≥ ··· ≥ w(e_|E|)
  M = ∅;
  for k = 1 to |E|
    if M ∪ {e_k} ∈ U and w(e_k) ≥ 0 then
      M = M ∪ {e_k};
  return T;
```

Es gibt Fälle, in denen der Algorithmus in der angegebenen Form versagt. Ist z. B. $E = \{a, b, c\}$, $U = \{\varnothing, \{a\}, \{b\}, \{c\}, \{b, c\}\}$ und $w\colon E \to \mathbb{R}$ definiert durch $w(a) = 3$, $w(b) = w(c) = 2$, so liefert Algorithmus 4.3 das Ergebnis $M = \{a\}$, obwohl wegen

$$w(M) = 3 < 4 = w(\{b, c\})$$

die Menge $M' := \{b, c\}$ „optimaler“ ist als M.

Um solche Fälle auszuschließen, geht man von allgemeinen Inzidenzstrukturen auf eine speziellere Datenstruktur über.

Definition Sei $T = (E, U)$ ein monoton gefiltertes Teilmengensystem mit der *Austauscheigenschaft* Matroid

$$\forall\ A, B \in U : \quad |A| < |B| \implies \exists\, x \in B \setminus A : \ A \cup \{x\} \in U.$$

Dann heißt T *Matroid.* □

Satz Sei $T = (E, U)$ ein monoton gefiltertes Teilmengensystem. Dann gilt: Der kanonische Raffke-Algorithmus (Algorithmus 4.3) liefert genau dann das Optimum in T bezüglich jeder beliebigen Gewichtsfunktion w, wenn T ein Matroid ist.

Beweis (nach [Sch97])
„$\Rightarrow$“ Angenommen, die Austauscheigenschaft gilt nicht, d. h.

$$\exists\ A, B \in U, |A| < |B|\ \forall\ b \in B \setminus A : \quad A \cup \{b\} \notin U.$$

Zu $r := |B|$ betrachte die Gewichtsfunktion

$$w\colon E \to \mathbb{R}, \qquad w(x) := \begin{cases} r + 1 & x \in A \\ r & x \in B \setminus A \\ 0 & \text{sonst} \end{cases}$$

Dann wählt der kanonische Raffke-Algorithmus eine Menge M mit $A \subseteq M$ und $M \cap (B \setminus A) = \varnothing$. Für dieses M gilt

$$w(M) = (r + 1) \cdot |A| \leq r^2 - 1.$$

Für Teilmengensysteme M' mit $B \subseteq M'$ gilt jedoch

$$w(M') \geq r \cdot |B| = r^2,$$

so daß der Algorithmus nicht die optimale Lösung geliefert hat. Widerspruch.

„$\Leftarrow$“ (E, U) sei ein Matroid, w eine Gewichtsfunktion und die Elemente von E seien bereits geordnet. O. B. d. A. gelte

$$w(e_1) \geq w(e_2) \geq \cdots \geq w(e_{|E|}).$$

Weiter sei $M := \{e_{i_1}, \ldots, e_{i_k}\}$ die vom kanonischen Raffke-Algorithmus gefundene Lösung. Angenommen, $M' := \{e_{j_1}, \ldots, e_{j_k}\}$ ist eine weitere optimale Lösung (Bem.: $|M| = |M'|$ gilt, weil T ein Matroid ist und $w(e_{i_k}), w(e_{j_k}) \geq 0$ gilt).

Nun existiert ein Index μ mit $w(e_{j_\mu}) > w(e_{i_\mu})$, also $j_\mu < i_\mu$. Betrachte $A = \{e_{i_1}, \ldots, e_{i_{\mu-1}}\}$ und $B = \{e_{j_1}, \ldots, e_{j_\mu}\}$: Wegen $|A| < |B|$ existiert ein $e_{j_\sigma} \in B \setminus A$ mit $A \cup \{e_{j_\sigma}\} \in U$, also

$$w(e_{j_\sigma}) \geq w(e_{j_\mu}) > w(e_{i_\mu}).$$

Dann hätte der kanonische Raffke-Algorithmus e_{j_σ} vor e_{j_μ} ausgewählt und wäre nicht zur Lösung M gelangt. Widerspruch. ■

Da nun die Korrektheit des Raffke-Algorithmus in einem allgemeinen Rahmen gesichert ist, gestalten sich die Korrektheitsbeweise für konkrete Raffke-Algorithmen erheblich leichter.

Bsp. 18 (Der Kruskal-Algorithmus) Sei $G = (V, E)$ ein Graph und $w\colon E \to \mathbb{R}$ eine Gewichtsfunktion. Der Algorithmus von Kruskal (Algorithmus 5) ist ein Raffke-Algorithmus zur Bestimmung eines minimalen Spannbaums von G.

Die Korrektheit des Kruskal-Algorithmus soll nun mit Hilfe des gerade gezeigten Satzes gezeigt werden. Sei also U die Menge aller zyklenfreien Untergraphen von G. Dann ist nur zu zeigen, daß (E, U) ein Matroid ist. Dies ist der Fall:

Offensichtlich ist (E, U) ein Filter. Seien $A, B \subseteq E$ zyklenfrei mit $|A| < |B|$. A zerlegt V in „Zusammenhangskomponenten“ V_i:

$$V = V_1 \,\dot\cup\, V_2 \,\dot\cup\, \cdots \,\dot\cup\, V_k.$$

Für jede Kante $e \in B$ lassen sich zwei Fälle unterscheiden:
1. Fall: e verbindet zwei Knoten einer Knotenmenge V_i.
2. Fall: e verbindet zwei Knoten $v_1 \in V_i$ und $v_2 \in V_j$ mit $i \neq j$.
Die Anzahl der Kanten in B, für die der erste Fall zutrifft, ist beschränkt durch

$$|A| = \sum_{i=1}^{k} (|V_i| - 1),$$

da B zyklenfrei ist. Somit existiert wegen $|B| > |A|$ mindestens eine Kante e vom Typ des zweiten Falls und für diese ist $A \cup \{e\}$ noch immer zyklenfrei. Damit ist U ein Matroid. □

4.3 Ansätze für parallele Algorithmen

Ein chinesischer Bauer hat einen Haufen Melonen geerntet und will diesen nun zum Transport auf kleine Wagen verladen, wobei auf jedem Wagen die gleiche Anzahl Melonen liegen soll. Allerdings gestaltet sich dies schwieriger als erwartet, denn die Melonen lassen sich so weder auf zwei, noch auf drei, vier, fünf oder sechs Wagen verladen: Stets bleibt genau eine Frucht übrig. Erst auf sieben Wagen lassen sie sich gleichmäßig verteilen. Wieviele Melonen hat der Bauer (mindestens) geerntet? (*Antwort*: 301)

Allgemeiner und mathematischer läßt sich das obige Problem wie folgt notieren: Seien $r_1, r_2, \ldots, r_n \in \mathbb{N}$ gegeben und $m_1, m_2, \ldots, m_n \in \mathbb{N}$ koprim. Gesucht ist eine Zahl $x \in \mathbb{N}$ mit

$$x = r_i \bmod m_i \qquad (i = 1, \ldots, n).$$

Gleichungssysteme diesen Typs werden als *Chinesische Restesysteme* bezeichnet. Darin heißen die Zahlen r_i die *Reste* und die Zahlen m_i die *Moduli* des Systems. Über die Lösbarkeit Chinesischer Restesysteme macht der Chinesische Restsatz (siehe Anhang B, S. 62, zweite Formulierung) eine Existenzaussage. Demnach gibt es genau eine Lösung im Bereich $0, \ldots, \prod_{i=1}^{n} m_i - 1$. Modulus

Der Chinesische Restsatz bietet weitere Anwendungsmöglichkeiten: z. B. konstruierte Lehmer 1942 eine Faktorisierungsmaschine auf der Basis dieses Satzes. Wie bereits in Abschnitt 2.3 angeklungen, kann man den Chinesischen Restsatz auch zur Parallelisierung verwenden. Dies soll hier näher untersucht werden.

Sei R ein Ring und $I_1, \ldots, I_n \lhd R$ mit $I_i + I_j = R$ $(i \neq j)$. Nach dem Chinesischen Restsatz gibt es einen Homomorphismus $\psi\colon R \to {}^{R}\!/_{I_1} \times \cdots \times {}^{R}\!/_{I_n}$ und für diesen gilt nach dem Einbettungssatz

$$\begin{tikzcd} R \arrow[r, "\psi"] \arrow[d, "\pi"'] & {}^{R}\!/_{I_1} \times \cdots \times {}^{R}\!/_{I_n} \\ {}^{R}\!/_{\ker\psi} = {}^{R}\!/_{\bigcap_{k=1}^{n} I_k} \arrow[ur, "\text{isomorph}"'] & \end{tikzcd}$$

Jede Datenstruktur $A \subseteq R$ mit $|A \cap \ker(\psi)| \leq 1$ wird durch

$$P := {}^{R}\!/_{I_1} \times \cdots \times {}^{R}\!/_{I_n}$$

treu modelliert. Das heißt: Wenn es möglich ist, eine Rechnung in ${}^{R}\!/_{\ker\psi}$ statt in A durchzuführen, dann ist dies auch in P möglich. Die Idee ist also, ein gegebenes Problem, das sich in ${}^{R}\!/_{\ker\psi}$ modellieren läßt, parallel mit n Prozessoren zu lösen. Der k-te Prozessor löst dann das Problem in ${}^{R}\!/_{I_k}$.

Anschließend müssen die Ergebnisse der n Prozessoren wieder zu einem Element von ${}^{R}\!/_{\ker\varphi}$ zusammengesetzt werden. Ist

$$\varphi\colon {}^{R}\!/_{\bigcap_{k=1}^{n} I_k} \to \prod_{k=1}^{n} {}^{R}\!/_{I_k}$$

der Isomorphismus, so wird in Ermangelung eines besseren Namens für die Umkehrfunktion φ^{-1} die Bezeichnung 算 (*suan*[12]) verwendet. suan

[12]Das sinojapanische Zeichen 算 bedeutet „rechnen". Es setzt sich zusammen aus den drei Radikalen 竹 (*zhu*, „Bambus"), 目 (*mu*, „Auge") und 廾 (*gong*, „zwei Hände").

Ein Algorithmus für 算

Wir betrachten exemplarisch die Zerlegung $\mathbb{Z}_{p\cdot q} \to \mathbb{Z}_p \times \mathbb{Z}_q$ mit $\mathrm{ggT}(p,q) = 1$, in anderen euklidischen Ringen geht die Konstruktion vollkommen analog. Die Zerlegung eines Elements $x \in \mathbb{Z}_{p\cdot q}$ ist einfach: $x_p := x \bmod p$ und $x_q := x \bmod q$. Aber wie konstruiert man die Abbildung 算?

Seien also x_p und x_q gegeben und das zugehörige x gesucht. Wenn $x = x_p \bmod p$ gilt, dann gilt auch $x + s\cdot p = x_p \bmod p$ für jedes s. Die Idee ist daher, zunächst ein x zu finden, das die erste Gleichung löst, und dieses dann um ein geeignetes Vielfaches von p zu variieren. Trivialerweise kann man mit $x = x_p$ beginnen und s dann wie folgt berechnen:

$$\begin{aligned} s\cdot p + x_p &\equiv x_q \bmod q \\ s &\equiv ((x_q - x_p)\cdot p^{-1}) \bmod q. \end{aligned}$$

Damit erhält man $x = \text{算}(x_p, x_q) = x_p + s\cdot p$. Wie man $p^{-1} \bmod q$ mit Hilfe des ELBA-Algorithmus bestimmt, ist auf Seite 54 beschrieben.

Hat man mehr als zwei Gleichungen zu erfüllen, kann man die gerade vorgestellte Rechnung iteriert anwenden. Seien also r_k und m_k $(k = 0,\ldots,n-1)$ mit $\mathrm{ggT}(m_i, m_j) = 1$ $(i \neq j)$ gegeben und ein x mit

$$x = r_k \bmod m_k \qquad (k = 0,\ldots,n-1)$$

gesucht. Setzt man $M_k := \prod_{i=0}^{k} m_k$, so erfüllt eine Lösung x offensichtlich auch die Gleichungen $x = r_k \bmod M_k$ $(k = 0,\ldots,n-1)$. Dies verwendet der folgende Algorithmus:

Algorithmus 4.4: Ein Algorithmus zur Berechnung von 算

```
function suan(r[], m[])
  M := 1;  x = r[0] mod m[0];
  for k = 1 to n − 1 do
    M := M · m[k − 1];
    s := (r[k] − (x mod m[k])) · (M^−1 mod m[k]) mod m[k];
    x := x + s · M;
  return x;
```

Zur Komplexitätsanalyse: Die Zahlen $r[i]$, $m[i]$ $(i = 0,\ldots,n-1)$ und n seien in ihrer Länge beschränkt, wohingegen x und M als Langzahlen aufgefaßt werden sollen. Addition, Multiplikation und die Modulo-Operation auf Kurzzahlen – Langzahlen der Länge $O(1)$ – mögen $O(1)$ Schritte kosten, Addition und Modulo von Langzahlen a, b koste $O(\log\max\{a,b\})$ Schritte, die Multiplikation $O(\log^2\max\{a,b\})$ Schritte. Die Berechnung von $a^{-1} \bmod b$ für Langzahlen a, b ist mit Hilfe von Euklid in $O(\log\max\{a,b\})$ Schritten möglich.

Die Anweisung $M := M\cdot m[k-1]$ benötigt $O(\log^2 M)$ Schritte. Die Berechnung von $x \bmod m[k]$ kostet $O(\log x)$, wobei das Ergebnis eine Kurzzahl ist, also in $O(1)$ von $r[k]$ abgezogen werden kann. $M^{-1} \bmod m[k]$ zu berechnen braucht $O(\log M)$ Operationen, das Ergebnis ist eine Kurzzahl, die in $O(1)$ mit der Kurzzahl $r[k] - x \bmod m[k]$ multipliziert werden kann. Die letzte Zeile der **for** Schleife braucht $O(\log x) + O(\log^2 M)$ Schritte. Insgesamt hat die Schleife also die Komplexität $O(n \cdot (\log x + \log^2 M))$.

Nun gilt in der k-ten Iteration

$$O(\log M) = O(\log \prod_{i=0}^{k} m[k]) = O(\sum_{i=0}^{k} \log m[k]) = \sum_{i=0}^{k} O(1) = O(k)$$

und

$$O(x) = O(\sum_{i=0}^{k} M) = \sum_{i=0}^{k} O(k) = O(k^2).$$

Zusammen erhält man

$$\varkappa(n) = \sum_{k=0}^{n-1} O(\log x + \log^2 M) = \sum_{k=0}^{n-1} O(k^2) = O(n^3)$$

als Komplexität für die Berechnung einer Lösung des Chinesischen Resteproblems mit n Gleichungen.[13]

Hat man viele Chinesische Restesysteme zu immer den selben Moduli zu berechnen, dann empfiehlt sich eine Vorausberechnung von

$$a_i := \text{算}(0, \ldots, 0, \underset{\substack{\uparrow \\ i\text{-te Stelle}}}{1}, 0, \ldots, 0).$$

Für einen Restevektor $(r_0, \ldots, r_{n-1})$ erhält man die Lösung x dann via

$$x = \sum_{i=0}^{n-1} r_i a_i.$$

Bsp. 19 Gesucht ist ein u mit

$$u \equiv 1 \bmod 3, \qquad u \equiv 3 \bmod 5, \qquad u \equiv 2 \bmod 8.$$

Nach der Initialisierung $M = 1$ und $x = r_0 \bmod m_0 = 1$ tritt Algorithmus 4.4 in die Schleife ein. Während der Schleife nehmen M und x die folgenden Werte an: (Die erste Zeile beinhaltet die Startwerte)

Abbildung 4.1: Zwischenergebnisse im Algorithmus 4.4

k	r_k	m_k	M	M^{-1}	s	x
0	1	3	1	1	–	1
1	3	5	3	2	4	13
2	2	8	15	7	3	$\boxed{58}$

Wie man leicht durch Nachrechnen bestätigt, ist 58 eine Lösung. Wegen $3 \cdot 5 \cdot 8 = 120$ ist damit $58 + 120\mathbb{Z}$ die Menge aller Lösungen des Systems. $\square$

[13] Betrachtet man auch x und M als Kurzzahlen mit der Komplexität $O(1)$ für Addieren, Multiplizieren und Modulo, so erhält man $\varkappa(n) = O(n^2)$.

Man macht sich leicht klar, daß Algorithmus 4.4 die kleinste positive Lösung des Problems liefert. Mit der Kenntnis, daß alle Lösungen periodisch zu $p := \prod_k m_k$ auftreten, läßt sich auch leicht die betragsmäßig kleinste Lösung angeben: Liegt nämlich die kleinste positive Lösung x jenseits von $p/2$, so ist $x - p$ die betragsmäßig kleinste Lösung.

Paralleles Lösen Linearer Gleichungssysteme in $\mathbb{Z}$

lineares Gleichungssystem

Seien $A \in \mathbb{Z}^{n \times n}$ und $b \in \mathbb{Z}^n$. Gesucht ist ein Vektor $x \in \mathbb{Q}^n$ mit $A \cdot x = b$ (*lineares Gleichungssystem*). Bekanntlich ist die Bestimmung einer Lösung x mit dem Algorithmus von Gauss in $O(n^3)$ Schritten möglich, solange man mit Fließkommazahlen arbeitet. Ist man allerdings – wie wir – an einer exakten Lösung interessiert, führt das rasche Wachstum der Zahlen in Zähler und Nenner im worst case bei genauer Betrachtung zu einer exponentiellen Laufzeit. Aus diesem Grunde mag es gerechtfertigt erscheinen, nach Alternativen zum Gauss-Algorithmus zu suchen.

Cramersche Regel

Bezeichne A_i die Matrix, die aus A hervorgeht, indem man die i-te Spalte in A durch b ersetzt. Weiter gelte $\det A \neq 0$. Dann ist nach der *Cramerschen Regel* die Lösung $x = (x_0, \ldots, x_n - 1)$ gegeben durch

$$x_i = \frac{\det A_i}{\det A} \qquad (i = 0, \ldots, n-1).$$

Im Gegensatz zum Gauss-Algorithmus geschieht hier der größte Teil der Rechnung in $\mathbb{Z}$, nur der jeweils letzte Schritt bei der Bestimmung eines x_i ist eine Division.

Kennt man ein primes $p > 2\max\{\det A, \det A_i\}$, so kann man $\mathbb{Z}$ problemlos durch $\mathbb{Z}_p$ modellieren.

Bsp. 20 (Lösen Linearer Gleichungssysteme in $\mathbb{Z}_p$) Wir suchen eine (exakte) Lösung $x \in \mathbb{Q}^2$ für $A \cdot x = b$ mit

$$A := \begin{pmatrix} 2 & -7 \\ 5 & -3 \end{pmatrix}, \qquad b := \begin{pmatrix} 4 \\ -1 \end{pmatrix}.$$

Man rechnet leicht nach, daß $\det A = 29$, $\det A_1 = -19$ und $\det A_2 = -22$ gilt und damit $x = -\frac{1}{29}(19, 22)$ die Lösung ist.

Auf diese Lösung kommt man auch, wenn man in $\mathbb{Z}_{71}$ rechnet (bea.: $71 \geq 2\max\{|29|, |-19|, |-22|\}$ ist prim). Hier ist

$$(A \mid b) \equiv \left(\begin{array}{cc|c} 2 & 64 & 4 \\ 5 & 68 & 70 \end{array}\right)$$

und man erhält

$$\det A \equiv 29, \qquad \det A_0 \equiv 52, \qquad \det A_1 \equiv 49.$$

Zur Rücktransformation ist jeweils der betragsmäßig kleinste Vertreter zu wählen, also $\det A_0 \equiv -19$ und $\det A_1 \equiv -22$, was zum bereits bekannten Ergebnis führt. □

Nun kann man die Kenntnis des Chinesischen Restsatzes dazu anwenden, ein lineares Gleichungssystem parallel zu lösen. Statt in $\mathbb{Z}_p$ zu rechnen, rechnet man unabhängig voneinander in $\mathbb{Z}_{p_1}$, $\mathbb{Z}_{p_2}, \ldots, \mathbb{Z}_{p_k}$ mit

$$\prod_{i=0}^{k-1} p_i \geq 2 \max\{\det A, \det A_i\}$$

und setzt die einzelnen Ergebnisse anschließend mit Hilfe der 算-Abbildung wieder zusammen.

Bsp. 21 (Lösen Linearer Gleichungssysteme in $\mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_k}$) Das Beispiel 20 soll fortgeführt werden. Wegen

$$7 \cdot 5 \cdot 2 = 70 \geq 2 \max\{\det A, \det A_0, \det A_1\}$$

kann man das System parallel in $\mathbb{Z}_7$, $\mathbb{Z}_5$ und $\mathbb{Z}_2$ betrachten. Man erhält dann

$$\begin{aligned}(A \mid b) &\equiv \left(\begin{array}{cc|c} 2 & 0 & 4 \\ 5 & 4 & 6 \end{array}\right) \bmod 7, \\ &\equiv \left(\begin{array}{cc|c} 2 & 3 & 4 \\ 0 & 2 & 4 \end{array}\right) \bmod 5, \\ &\equiv \left(\begin{array}{cc|c} 0 & 1 & 0 \\ 1 & 1 & 1 \end{array}\right) \bmod 2,\end{aligned}$$

und damit die drei Chinesischen Restesysteme

$$\begin{array}{rlrlrl} \det A & \equiv 1 \bmod 7 & \det A_0 & \equiv 2 \bmod 7 & \det A_1 & \equiv 6 \bmod 7 \\ & \equiv 4 \bmod 5 & & \equiv 1 \bmod 5 & & \equiv 3 \bmod 5 \\ & \equiv 1 \bmod 2 & & \equiv 1 \bmod 2 & & \equiv 0 \bmod 2. \end{array}$$

Algorithmus 4.4 liefert hierfür die kleinsten positiven Lösungen $\det A = 29$, $\det A_0 = 52$ und $\det A_1 = 49$. Daraus ergeben sich die betragsmäßig kleinsten Lösungen $\det A_0 = -19$ und $\det A_1 = -22$, die unsere Lösung abermals bestätigen. □

Nun ist es so, daß wir vor der Bestimmung der Determinanten Moduli p_i wählen müssen, das seinerseits von den Determinanten abhängt. Die einfachste Lösung dieses Problems ist es, p in der Größenordnung der Wortlänge des Rechners zu wählen, da ein Überlauf andernfalls ohnehin zu falschen Ergebnissen bzw. Fehlermeldungen führt. Man kann p aber auch aufgrund von allgemeinen Abschätzungen wählen, die Aussagen über die Größenordnung von Determinanten machen.

Neben der einfachen Abschätzung

$$|\det A| \leq n! \left(\max_{0 \leq i,j < n} |a_{ij}| \right)^n$$

gilt zum Beispiel die *Hadamardsche Ungleichung*: Hadamardsche Ungleichung

$$|\det A|^2 \leq \prod_{i=0}^{n-1} \sum_{j=0}^{n-1} |a_{ij}|^2 \qquad (A = ((a_{ij}))_{i,j=0}^{n-1} \in \mathbb{R}^{n \times n}).$$

Bei der Verwendung solcher Formeln ist allerdings damit zu rechnen, daß man eine Schranke erhält, die wesentlich größer ist, als eigentlich erforderlich wäre.

Der Chinesische Restsatz und Interpolation

Interpolation

Die Suche nach dem (eindeutig bestimmten) Polynom $p \in \mathbb{F}[x]$ kleinsten Grades mit $p(x_i) = y_i$ für Punkte $(x_i, y_i) \in \mathbb{F}^2$ mit $x_i \neq x_j$ $(i \neq j)$ heißt *Interpolation*.

Lagrange-Interpolation

Sind $(x_i, y_i) \in \mathbb{F}^2$ für $i = 0, \ldots, n-1$ gegeben, so erhält man das gesuchte Polynom leicht über die *Lagrangesche Interpolationsformel*:

$$p(x) = \sum_{k=0}^{n-1} y_k \cdot \prod_{\substack{i=0\\ i\neq k}}^{n-1} \frac{x - x_i}{x_k - x_i}.$$

Die Bestimmung des Interpolationspolynoms nach Lagrange hat aber Nachteile. So ist es z. B. nicht möglich, nachträglich ein Polynom an einen Punkt (x_n, y_n) „anzupassen“.

Die Bestimmung eines Interpolationspolynoms kann man als spezielles Chinesisches Resteproblem auffassen. Statt $\mathbb{Z}$ legt man den euklidischen Ring $\mathbb{F}[x]$ zugrunde. Die Suche nach einem $p \in \mathbb{F}[x]$ mit $p(x_i) = y_i$ $(i = 0, \ldots, n-1)$ ist dann nichts anderes als die Suche nach einem $p \in \mathbb{F}[x]$ mit

$$p \equiv y_i \bmod (x - x_i) \qquad (i = 1, \ldots, n).$$

Newton-Interpolation

Algorithmus 4.4 liefert also unmittelbar einen neuen Interpolationsalgorithmus. Man spricht in diesem Zusammenhang von der Newton-*Interpolation*.

Algorithmus 4.5: Interpolation nach Newton

```
function newtonPolynom(x[], y[])
  M(x) := 1;  p(x) = y[0];
  for k = 1 to n − 1 do
    M(x) := M(x) · (x − x[k − 1]);
    s := (y[k] − p(x[k]))/M(x[k]);
    p(x) := p(x) + s · M(x);
  return p(x);
```

Bsp. 22 (Interpolation nach Newton) Gesucht ist das Polynom $p \in \mathbb{Q}[x]$ kleinsten Grades mit $p(1) = 2$, $p(3) = 0$ und $p(4) = 3$.

Abbildung 4.2: Zwischenergebnisse im Algorithmus 4.5

k	x_k	y_k	$M(x)$	$M^{-1}(x_k)$	s	$p(x)$
0	1	2	1	1	–	2
1	3	0	$x-1$	$\frac{1}{2}$	-1	$2-(x-1)$
2	4	3	$(x-1)(x-3)$	$\frac{1}{3}$	$\frac{4}{3}$	$\boxed{2-(x-1)+\frac{4}{3}(x-1)(x-3)}$

Man überzeugt sich durch Einsetzen leicht von der Richtigkeit des Ergebnisses. Anders als bei der Interpolation nach Lagrange kann man bei Newton leicht die Spezifikation erweitern. Will man z. B., daß zusätzlich $p(7) = 1$ gilt, so erhält man das gewünschte Polynom, indem man die Tabelle weiterführt: Es ist dann $M(x) = (x-1)(x-3)(x-4)$, $M(x_k)^{-1} = \frac{1}{72}$, $s = -\frac{3}{8}$ und schließlich

$$p(x) = 2 - (x-1) + \tfrac{4}{3}(x-1)(x-3) - \tfrac{3}{8}(x-1)(x-3)(x-4). \qquad \square$$

Wie im allgemeinen Fall (vgl. S. 4.3) kann man auch hier Rechenzeit durch Vorausberechnung einsparen, wenn die x_i fest und nur die y_i variabel sind.

Man erhält dann

$$a_k(x) = \text{算}(0, \ldots, 0, \underset{\substack{\uparrow \\ k\text{-te Stelle}}}{1}, 0, \ldots, 0) = 1 \cdot \prod_{\substack{i=0 \\ i \neq k}}^{n-1} \frac{x - x_i}{x_k - x_i}.$$

Ein Vektor $y = (y_0, \ldots, y_{n-1})$ wird damit an den Stützstellen $x_0, \ldots, x_{n-1}$ interpoliert durch das Polynom

$$p(x) = \sum_{k=0}^{n-1} y_k a_k = \sum_{k=0}^{n-1} y_k \cdot \prod_{\substack{i=0 \\ i \neq k}}^{n-1} \frac{x - x_i}{x_k - x_i},$$

was gerade die Formel von Lagrange ist.

Weitere Einzelheiten zu Anwendungen des Chinesischen Restsatzes findet man bei [Lip81].

5 Zur schnellen Fouriertransformation

Ein häufiges Problem in der Signalverarbeitung ist es, zu gegebenen Vektoren[14] $f,\ g \in \mathbb{C}^n$ den *Faltungsvektor* $f * g \in \mathbb{C}^{2n-1}$ mit

Faltung

$$f * g = (f_0 \cdot g_0, f_1 \cdot g_0 + f_0 \cdot g_1, \ldots, \sum_{k=0}^{n-1} f_k \cdot g_{i-k}, \ldots, f_{n-1} \cdot g_{n-1})$$

zu bestimmen. Vielfältige Anwendungen der Faltungsoperation wurden in Info IV behandelt und sind u. a. in [DniQ00] nachzulesen.

Die Berechnung von $f * g$ mit einem naiven Algorithmus benötigt $O(n^2)$ Schritte. Bekanntlich kann man mit Hilfe der diskreten Fouriertransformation[15] die Faltung in $O(n \log n)$ Schritten berechnet:

$$\begin{tikzcd}
f \arrow[r, "*g"] \arrow[d, "\mathrm{DFT}_N"'] & f * g \\
\widehat{f} \arrow[r, "\odot \widehat{g}"] & \widehat{f} \odot \widehat{g} \arrow[u, "\mathrm{DFT}_N{}^{-1}"']
\end{tikzcd}$$

Dabei entsteht $f \odot g$ durch komponentenweise Multiplikation von f und g.

Die Größe $N \geq 2n - 1$ der DFT-Matrix wird dabei so gewählt, daß sich die Multiplikation eines Vektors mit DFT_N in $O(N \log N) = O(n \log n)$ Schritten berechnen läßt. Ist z. B. N eine Potenz von 2, so kann man den bekannten Rekursionsalgorithmus verwenden:

Seien $n = 2^k$, $f \in \mathbb{C}^n$ und ω eine n-te Einheitswurzel in $\mathbb{C}$. Es gilt (vgl. [DniQ00] oder Abschnitt 5.1)

$$\mathrm{DFT}_{2n} = \underbrace{\begin{pmatrix} I_n & I_n \\ I_n & -I_n \end{pmatrix}}_{=\mathrm{DFT}_2 \otimes I_n} \cdot \underbrace{\begin{pmatrix} I & \\ & \Delta \end{pmatrix}}_{=:T} \cdot \underbrace{\begin{pmatrix} \mathrm{DFT}_n & \\ & \mathrm{DFT}_n \end{pmatrix}}_{=I_2 \otimes \mathrm{DFT}_n} \cdot P$$

mit $\Delta = \mathrm{diag}(\omega^0, \omega^1, \ldots, \omega^{n-1})$ und einer geeigneten Permutationsmatrix P. Diese Gleichung ist die Grundlage für Algorithmus 5.1. Die Permutation P entspricht der Aufspaltung von $f = (f_0, \ldots, f_{n-1})$ in $(f_0, f_2, \ldots, f_{n-2})$ und $(f_1, f_3, \ldots, f_{n-1})$. Damit ist $(g, h) = (I_2 \otimes \mathrm{DFT}_n) \cdot P \cdot f$. Die **for** Schleife ist wie folgt zu interpretieren: Die Vorfaktoren ω^i und 1 liefern $T \cdot (g, h) = (I_n g, \Delta h)$. Die erste Zeile der Schleife ist schließlich für die „obere Hälfte“ und die zweite Zeile für die „untere Hälfte“ der Matrix $(\mathrm{DFT}_2 \otimes I_n)$ zuständig.

[14]Ist V ein Vektorraum der Dimension n und $a \in V$, so ist stets $a = (a_0, \ldots, a_{n-1})$ die Darstellung und Indizierung der Komponenten von a. Überdies identifizieren wir Zeilen- und Spaltenvektoren miteinander, um uns das ständige Hin- und Hertransponieren zu ersparen.

[15]Ist $a \in \mathbb{C}^n$, so stehe $\widehat{a} := \mathrm{DFT}_n \cdot a$ für die Fouriertransformierte von a. Weitere Bezeichnungsweisen werden im Anhang B ab Seite 65 definiert.

Algorithmus 5.1: Die schnelle Fouriertransformation für Zweierpotenzen

function $fft(f[\,], \omega)$
  if $f.length = 1$ **then**
    return $[f_0]$;
  $g := fft([f_0, f_2, f_4, \ldots, f_{n-2}], \omega^2)$;
  $h := fft([f_1, f_3, f_5, \ldots, f_{n-1}], \omega^2)$;
  for $i = 0$ **to** $n/2 - 1$
    $\widehat{f}_i = g_i + \omega^i h_i$;
    $\widehat{f}_{i+n/2} = g_i - \omega^i h_i$;
  return $\widehat{f}$;

Butterfly

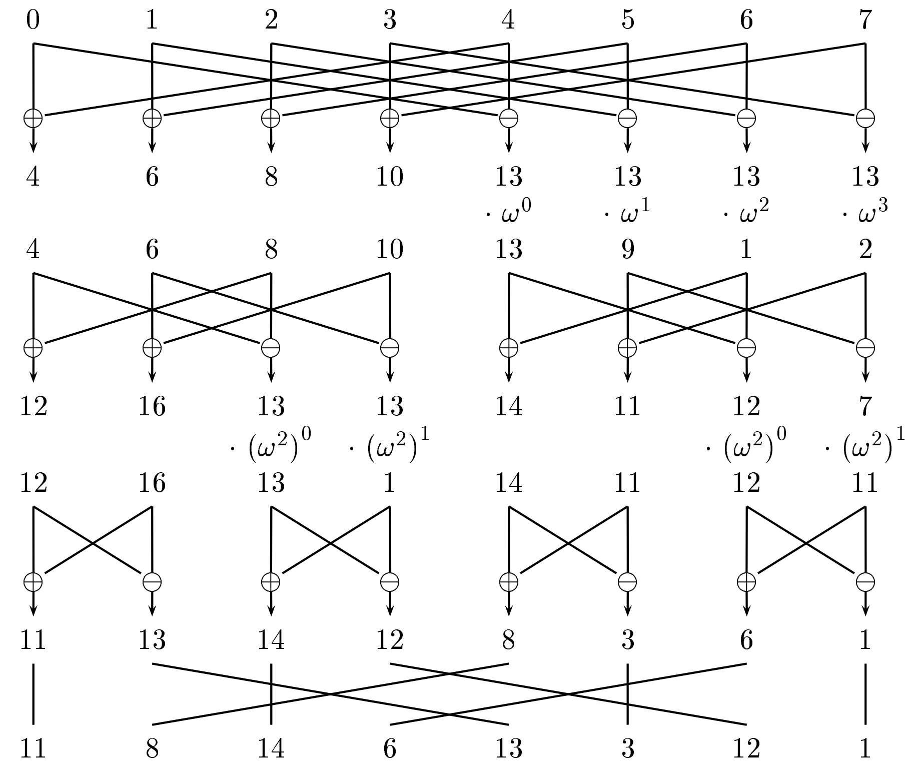
Spart man sich die Permutationen bis zum Schluß auf, so erhält man den *Butterfly*-Algorithmus, der sich sowohl zum Schaltungsentwurf als auch zur Handrechnung eignet.

Bsp. 23 In $\mathbb{Z}_{17}$ ist $\omega = 2$ eine primitive achte Einheitswurzel, es ist

i	0	1	2	3	4	5	6	7
ω^i	1	2	4	8	16	15	13	9

Gesucht ist die Fouriertransformierte $\widehat{f}$ des Vektors $f = (0, 1, 2, 3, 4, 5, 6, 7)$. Abbildung 5.1 zeigt die Berechnung mit Hilfe des Butterfly-Algorithmus. Demnach ist $\widehat{f} = (11, 8, 14, 6, 13, 3, 12, 1)$. □

Abbildung 5.1: Der Butterfly-Algorithmus



Man erhält die Permutation, die nach Ausführung des Butterfly-Algorithmus aufgelaufen ist, indem man den i-ten Eintrag an die j-te Stelle verschiebt, wobei j aus i entsteht, indem man die Binärdarstellung von i „rückwärts" liest (vgl. Abbildung 5.2). Die so definierte Permutation wird daher auch als *Bitreversal* bezeichnet.

Bitreversal

Abbildung 5.2: Bitreversal

$$\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
000 & 100 & 010 & 110 & 001 & 101 & 011 & 111 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
0 & 4 & 2 & 6 & 1 & 5 & 3 & 7
\end{array}$$

Wenn $2n-1$ selbst nur wenig größer als eine Zweierpotenz ist, dann ist das nächste in Frage kommende N fast doppelt so groß wie eigentlich nötig wäre. Daher stellt sich die Frage, ob es neben dem obigen noch weitere schnelle Algorithmen gibt, die auch für andere N funktionieren. Das Ziel dieses Kapitels ist es, eine Sammlung verschiedener Algorithmen zur schnellen Berechnung der Fouriertransformation vorzustellen. Allerdings ist es schwer festzustellen, ob ein Verfahren „besser" oder „schlechter" ist als ein anderes, da dies unter anderem auch von der zugrunde liegenden Rechnerarchitektur abhängt.

Die folgende Übersicht zeigt die möglichen Zerlegungsverfahren zur Berechnung der DFT_N:

- $N \lessapprox 16$: direkte Berechnung
- $N = n \cdot m$: FFT nach COOLEY-TUKEY
- $N = p_1 \cdots p_k$ mit $\mathrm{ggT}(p_i, p_j) = 1$ $(i \neq j)$: FFT nach GOOD-THOMAS
- N prim: FFT nach RADER
- N beliebig: FFT nach BLUESTEIN

Es ist zu bemerken, daß die obigen Verfahren auch kombiniert werden können: In jedem Rekursionsschritt ist dynamisch zu entscheiden, welches der zur Verfügung stehenden Verfahren am sinnvollsten anzuwenden ist.

Alle Verfahren haben eine Laufzeit von $O(N \log N)$. Einzelheiten zur Herleitung und weiterführende Verallgemeinerungen findet man in [$\beth$84].

5.1 FFT nach COOLEY-TUKEY

Es sei $N = m \cdot n$ $(m, n \geq 2)$ und $f = (f_0, \ldots, f_{N-1}) \in \mathbb{C}^N$. Weiter sei ω eine N-te Einheitswurzel. Dann gilt

$$\mathrm{DFT}_N \cdot f = (\widehat{f}_0, \ldots, \widehat{f}_{N-1})$$

mit

$$\widehat{f}_\mu = \sum_{\nu=0}^{N-1} f_\nu \omega^{\mu\nu} \qquad (\mu = 0, \ldots, N-1). \tag{4}$$

Ähnlich wie in Algorithmus 5.1 soll der Vektor f nun geeignet aufgespalten werden. Dazu setze $\mu = k \cdot n + l$ und $\nu = i \cdot m + j$ ($k, j = 0, \ldots, m-1$, $l, i = 0, \ldots, n-1$). Die Indizes μ eines gemeinsamen l bilden zusammen eine *Nebenklasse* der zyklischen Gruppe $\mathbb{Z}_n$ (analog für ν). Aus diesem Grund wird die Methode von Cooley-Tukey auch als *Nebenklassenmethode* bezeichnet.

Nebenklassen-methode

Mit der genannten Setzung für μ und ν wird Gleichung 4 zu

$$\widehat{f}_\mu = \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} f_{im+j} \cdot \underbrace{\omega^{(im+j)(kn+l)}}_{=\omega^{jkn}\omega^{iml}\omega^{jl}} = \sum_{j=0}^{m-1} \underbrace{\left(\sum_{i=0}^{n-1} f_{im+j} (\omega^m)^{il}\right)}_{=:b_{lm+j}} \cdot \omega^{jl} \cdot (\omega^n)^{jk}. \tag{5}$$

Da ω nach Voraussetzung eine N-te Einheitswurzel ist, ist ω^m eine n-te und ω^n eine m-te Einheitswurzel. Damit gilt offensichtlich

$$(b_{lm+j})_{l=0}^{n-1} = \mathrm{DFT}_n^{(\omega^m)} \cdot (f_{im+j})_{i=0}^{n-1}.$$

Sei nun P die Permutationsmatrix, die die Koordinaten von f nach Nebenklassen „sortiert“, d. h.

$$P \cdot (f_0, f_1, \ldots, f_{N-1}) = (\underbrace{\underbrace{f_0, f_n, \ldots, f_{(m-1)n}}_{m \text{ Komponenten}}, \underbrace{f_1, f_{n+1}, \ldots, f_{(m-1)n+1}}_{m \text{ Komponenten}}, \ldots\ldots}_{n \text{ Nebenklassen}})$$

Den Vektor $(b_{lm+j})_{l=0}^{n-1}$ erhält man nun durch parallele Anwendung je einer $\mathrm{DFT}_n^{(\omega^m)}$ auf jede Nebenklasse:

$$(b_p)_{p=0}^{N-1} = \underbrace{\begin{pmatrix} \mathrm{DFT}_n & & \\ & \ddots & \\ & & \mathrm{DFT}_n \end{pmatrix}}_{=I_m \otimes \mathrm{DFT}_n} \cdot P \cdot f.$$

In Gleichung 5 werden die nun berechneten b_{jm+l} mit einem Faktor ω^{jl} multipliziert. Diese Faktoren werden als *Twiddlefaktoren* bezeichnet. Ist

$$\Delta = \mathrm{diag}(\omega^0, \omega^1, \ldots, \omega^{n-1}),$$

so entspricht die Multiplikation von b_{lm+j} mit ω^{jl} insgesamt der Multiplikation des Vektors $(b_{lm+j})_{l=0}^{n-1}$ mit der Matrix

$$T = \mathrm{diag}(\Delta^0, \Delta^1, \ldots, \Delta^{m-1}).$$

Twiddlematrix

Die Matrix T wird folgerichtig als *Twiddlematrix* bezeichnet.

Es bleibt also noch die Multiplikation mit $(\omega^n)^{jk}$ sowie die äußere Summe in Gleichung 5 zu untersuchen. Ohne das k im Exponenten bliebe also nichts weiter als eine $\mathrm{DFT}_m^{(\omega^n)}$ der „getwiddelten" Komponenten b_{lm+j} stehen. Aus der Bedeutung des k im Index μ leitet sich nun ab, daß es sich mit dem k um eine Anwendung der $\mathrm{DFT}_m^{(\omega^n)} \otimes I_n$ handelt.

Insgesamt gilt also

$$\mathrm{DFT}_{mn}^{(\omega)} = (\mathrm{DFT}_m^{(\omega^n)} \otimes I_n) \cdot T \cdot (I_m \otimes \mathrm{DFT}_n^{(\omega^m)}) \cdot P$$

mit T und P wie oben definiert.

Bsp. 24 Es gilt $4 = 2 \cdot 2$. $\omega = \mathrm{i}$ ist primitive vierte Einheitswurzel in $\mathbb{C}$. Man verifiziert durch Nachrechnen die obige Formel:

$$\underbrace{\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \mathrm{i} & -1 & -\mathrm{i} \\ 1 & -1 & 1 & -1 \\ 1 & -\mathrm{i} & -1 & \mathrm{i} \end{pmatrix}}_{\mathrm{DFT}_4} = \underbrace{\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}}_{\mathrm{DFT}_2 \otimes I_2} \cdot \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \mathrm{i} \end{pmatrix}}_{T} \cdot \underbrace{\begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix}}_{I_2 \otimes \mathrm{DFT}_2} \cdot \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{P}$$

□

5.2 FFT nach GOOD-THOMAS

Der komplexitätstheoretisch unangenehmste Teil in der Formel von COOLEY-TUKEY ist die Anwendung der Twiddlematrix. Für den Fall, daß m und n teilerfremd sind, bietet das Verfahren von GOOD-THOMAS eine Zerlegung, die ohne diese Matrix auskommt.

Sei also $n = p \cdot q$ mit $\mathrm{ggT}(p, q) = 1$. Die Funktion

$$\varphi\colon \mathbb{Z}_n \to \mathbb{Z}_p \times \mathbb{Z}_q, \qquad \varphi(x) := (x \bmod p, x \bmod q)$$

ist ein Isomorphismus zwischen $\mathbb{Z}_n$ und $\mathbb{Z}_p \times \mathbb{Z}_q$ (vgl. Abschnitt 4.3). Zu jedem $\mu \in \mathbb{Z}_n$ gibt es damit insbesondere genau eine Darstellung $\mu = kp + lq$.

Sei $f = (f_0, \ldots, f_{n-1}) \in \mathbb{C}^n$ und $\omega \in \mathbb{C}$ eine n-te Einheitswurzel. In Gleichung 4 (Seite 44) setzen wir $\mu = kp + lq$ und $\nu = ip + jq$ ($k, i = 0, \ldots, q-1$, $l, j = 0, \ldots, p-1$). Dies führt auf

$$\widehat{f}_\mu = \sum_{i=0}^{q-1} \sum_{j=0}^{p-1} f_{ip+jq} \underbrace{\omega^{(kp+lq)(ip+jq)}}_{=\omega^{ljq^2}\omega^{kip^2}} = \sum_{i=0}^{q-1} \underbrace{\left(\sum_{j=0}^{p-1} f_{ip+jq} \left(\omega^{q^2}\right)^{lj} \right)}_{=:b_{ip+l}} \cdot \left(\omega^{p^2}\right)^{ki}. \tag{6}$$

Vergleicht man diese Gleichung mit Gleichung 5, so fällt auf, daß die Twiddlefaktoren jetzt nicht mehr auftreten, weil $\omega^{(kj+li)pq} = 1$ ist.

Für die innere Summe gilt wieder

$$(b_{ip+l})_{l=0}^{p-1} = \mathrm{DFT}_p^{(\omega^{q^2})} \cdot (f_{ip+jq})_{j=0}^{p-1}.$$

Die Permutationsmatrix P mit

$$P \cdot (f_0, f_1, \ldots f_{n-1}) = (f_0, f_q, \ldots, f_{(p-1)q}, f_p, f_{p+q}, \ldots, f_{p+(p-1)q}, \ldots\ldots)$$

liefert

$$(b_m)_{m=0}^{n-1} = (I_q \otimes \mathrm{DFT}_p^{(\omega^{q^2})} \cdot P \cdot f.$$

Fährt man fort wie im vorangegangenen Abschnitt und macht am Schluß die Permutation wieder rückgängig, so erhält man

$$\mathrm{DFT}_{pq}^{(\omega)} = P^{-1} \cdot (\mathrm{DFT}_q^{(\omega^{p^2})} \otimes I_p) \cdot (I_q \otimes \mathrm{DFT}_p^{(\omega^{q^2})}) \cdot P.$$

5.3 FFT nach Rader

Die Verfahren von Cooley-Tukey und Good-Thomas sind anwendbar, wenn sich die Problemgröße p faktorisieren läßt. Es stellt sich nun die Frage nach einem Verfahren, das die DFT_p schnell berechnet, wenn p prim ist.

Ist $p > 3$ prim, so ist $p-1$ mit Sicherheit nicht prim. Wenn es also gelingt, die DFT_p für primes p auf die DFT_{p-1} zu reduzieren, dann kann man jene mit den bekannten Methoden weiter behandeln.

In der ersten Zeile (wie der ersten Spalte) einer jeden DFT-Matrix sind alle Einträge 1. Aus diesem Grund ist die Berechnung von $\widehat{f}_0$ ohne eine Multiplikation möglich:

$$\widehat{f}_0 = \sum_{i=0}^{p-1} f_i.$$

Zu DFT_p sei $W_p \in \mathbb{F}^{(p-1)\times(p-1)}$ die Matrix, die aus DFT_p durch Streichen der ersten Zeile und der ersten Spalte hervorgeht. Um einander entsprechende Elemente in DFT_p und W_p gleich zu indizieren, wird W_p ab 1 (statt ab 0) indiziert. Es ist dann $W_p = ((\omega^{ij}))_{i,j=1}^{p-1}$.

Weil p prim ist, ist $\mathbb{Z}_p$ ein Körper und daher $(\mathbb{Z}_p^\times, \cdot)$ eine zyklische Gruppe der Ordnung $p-1$. Damit gibt es ein Element $a \in \mathbb{Z}_p^\times$, so daß sich jedes Element $e \in \mathbb{Z}_p^\times$ eindeutig als Potenz von a darstellen läßt.

Ist P die zur Permutation

$$\pi\colon \mathbb{Z}_p^\times \to \mathbb{Z}_p^\times, \qquad \pi(l) := a^l \bmod p$$

gehörige Permutationsmatrix, so gilt

$$PW_pP^{-1} = P((\omega^{ij}))_{i,j=1}^{p-1}P^{-1} = ((\omega^{a^i a^j}))_{i,j=1}^{p-1} = ((\omega^{a^{i+j}}))_{i,j=1}^{p-1}.$$

Wegen

$$\omega^{a^{i+j}} = \omega^{a^{i+j} \bmod p} = \omega^{a^{i+j \bmod p-1}}$$

ist PW_pP^{-1} also fast eine Zirkulante:

$$PW_pP^{-1} = \begin{pmatrix} \omega^{a^2} & \omega^{a^3} & \omega^{a^4} & \cdots \\ \omega^{a^3} & \omega^{a^4} & \iddots & \\ \omega^{a^4} & \iddots & & \\ \vdots & & & \end{pmatrix}.$$

Die Multiplikation des Vektors $(f_1, f_2, \ldots, f_{p-1})$ mit dieser Matrix entspricht gerade der zyklischen Faltung des Vektors mit

$$(\omega^{a^2}, \omega^{a^1}, \omega^{a^{p-1}}, \ldots, \omega^{a^3}).$$

Diese Faltung läßt sich mit Hilfe zweier DFT_{p-1} realisieren, so daß

$$\varkappa(\mathrm{DFT}_p) \leq O(p) + 2\varkappa(\mathrm{DFT}_{p-1})$$

gilt, wenn man die Komplexität für die Permutationen vernachlässigt.

Bsp. 25 Es sei ω eine fünfte Einheitswurzel. Dann gilt

$$\mathrm{DFT}_5 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \omega^4 \\ 1 & \omega^2 & \omega^4 & \omega^1 & \omega^3 \\ 1 & \omega^3 & \omega^1 & \omega^4 & \omega^2 \\ 1 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{pmatrix}, \quad \underbrace{\boxed{\begin{matrix} \omega^1 & \omega^2 & \omega^3 & \omega^4 \\ \omega^2 & \omega^4 & \omega^1 & \omega^3 \\ \omega^3 & \omega^1 & \omega^4 & \omega^2 \\ \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{matrix}}}_{=W_p}.$$

Die Abbildung

$$\pi \colon \mathbb{Z}_5^\times \to \mathbb{Z}_5^\times, \qquad \pi(x) = 2^x$$

definiert die Permutation $(1\,2\,4)$ und es ist $\pi^{-1} = (1\,4\,2)$. Die letzte Permutation angewandt auf die Spalten von W_p liefert

$$W_p {P_{(1\,2\,4)}}^{-1} = \begin{pmatrix} \omega^2 & \omega^4 & \omega^3 & \omega^1 \\ \omega^4 & \omega^3 & \omega^1 & \omega^2 \\ \omega^1 & \omega^2 & \omega^4 & \omega^3 \\ \omega^3 & \omega^1 & \omega^2 & \omega^4 \end{pmatrix}.$$

Die inverse Permutation wendet man noch auf die Zeilen an und erhält schließlich

$$P_{(1\,2\,4)} W_p {P_{(1\,2\,4)}}^{-1} = \begin{pmatrix} \omega^4 & \omega^3 & \omega^1 & \omega^2 \\ \omega^3 & \omega^1 & \omega^2 & \omega^4 \\ \omega^1 & \omega^2 & \omega^4 & \omega^3 \\ \omega^2 & \omega^4 & \omega^3 & \omega^1 \end{pmatrix}.$$

Die Matrix $P_{(1\,2\,4)} W_p {P_{(1\,2\,4)}}^{-1}$ hat also tatsächlich die vorausgesagte Form. $\square$

5.4 FFT nach Bluestein

Die bis jetzt vorgestellten Verfahren erlauben zusammen die Berechnung der diskreten Fouriertransformierten für jede beliebige Eingabegröße mit der Komplexität $O(n \log n)$.

Die Methode, die in diesem Abschnitt erläutert werden soll, ist eine Alternative, auf die man z. B. in den worst-case-Fällen der anderen Verfahren zurückgreifen kann.

Bsp. 26 Die Zahl 719 läßt bisher nur die folgende Zerlegung zu:

$$\begin{aligned}
719 &= 2 \cdot 359 + 1 && \text{(Rader)}\\
&= 2 \cdot (2 \cdot 179 + 1) + 1 && \text{(Rader)}\\
&= 2 \cdot (2 \cdot (2 \cdot 89 + 1) + 1) + 1 && \text{(Rader)}\\
&= 2 \cdot (2 \cdot (2 \cdot (2 \cdot 44 + 1) + 1) + 1) + 1 && \text{(Rader)}\\
&= 2 \cdot (2 \cdot (2 \cdot (2 \cdot (4 \cdot 11) + 1) + 1) + 1) + 1 && \text{(Cooley-Tukey oder}\\
& && \text{Good-Thomas)}\\
&= 2 \cdot (2 \cdot (2 \cdot (2 \cdot (4 \cdot (2 \cdot 5 + 1)) + 1) + 1) + 1) + 1 && \text{(Rader)}
\end{aligned}$$

Berechnet man DFT_4 und DFT_5 direkt, so hat der Zerlegungsbaum von 719 eine Tiefe von 6. □

Mit dem Verfahren von Bluestein wird die DFT_n zunächst auf eine DFT_m mit $m \geq 2n - 1$ „reduziert“, wobei man hierfür ein m wählt, das sich wesentlich leichter zerlegen läßt als n, z. B. die nächsthöhere Zweierpotenz.

Experimente bestätigen, daß der Einsatz dieses Verfahrens die Laufzeiten um bis zu 80% reduziert.

Seien $m, n \in \mathbb{N}$ mit $m \geq 2n - 1$. ω_n sei eine n-te Einheitswurzel und ω_{2n} eine $2n$-te Einheitswurzel mit $\omega_{2n}^2 = \omega_n$. Es gilt dann

$$\mathrm{DFT}_n = ((\omega_n^{ij}))_{i,j=0}^{n-1} = ((\omega_{2n}^{2ij}))_{i,j=0}^{n-1}.$$

Für

$$D = \mathrm{diag}(1, \omega_{2n}, \omega_{2n}^4, \omega_{2n}^9, \ldots, \omega_{2n}^{(n-1)^2})$$

hat

$$D^{-1}\mathrm{DFT}_n D^{-1} = ((\omega_{2n}^{-i^2} \omega_{2n}^{2ij} \omega_{2n}^{-j^2}))_{i,j=0}^{n-1} = ((\omega_{2n}^{-(i-j)^2}))_{i,j=0}^{n-1}$$

fast die Form einer Zirkulante. Bei Matrizen dieser Form spricht man von Toepliz-*Matrizen*. Jede Toepliz-Matrix $A = ((a_{ij}))_{i,j=0}^{n-1}$ läßt sich einbetten in eine Zirkulante $B \in \mathbb{F}^{(m-1)\times(m-1)}$ mit

$$B = \begin{pmatrix} A & * \\ * & * \end{pmatrix},$$

wähle z. B.

$$B = C_{Z_m}(a_{0,0}, a_{0,1}, \ldots, a_{0,n-1}, 0, \ldots, 0, a_{n-1,0}, a_{n-2,0}, \ldots, a_{1,0}).$$

Für $f \in \mathbb{F}^n$ gilt

$$\widehat{f} = \mathrm{DFT}_n f = (DD^{-1})\mathrm{DFT}_n(D^{-1}D)f = D(\underbrace{D^{-1}\mathrm{DFT}_n D^{-1}}_{=:A})(Df).$$

Der Vektor Df wird mit Nullen bis zur Länge m aufgefüllt und statt mit A mit B multipliziert. Da B eine Zirkulante ist, läßt sich diese Operation schnell mit Hilfe zweier DFT_m ausführen. Die ersten n Komponenten des Ergebnisvektors ergeben mit D multipliziert die Fouriertransformierte $\widehat{f}$.

Bsp. 27 Wählt man im Fall $n = 719$ beispielsweise $m = 1470 > 1438 = 2 \cdot 719$, so läßt sich diese Länge wie folgt zerlegen:

$$1470 = 35 \cdot 42 = (5 \cdot 7) \cdot (6 \cdot 7)$$

Dieser Baum ist nur noch zwei Ebenen tief. $\square$

6 Lasset uns bethen

Beth unser im I A K S
Verschlüsselt werde dein Name
Dein Recall komme
Dein Folienstift schreibe
Wie in Erlangen so in Karlsruhe
Unsere tägliche Vorlesung gib uns heute
Und vergib uns unsere Unkenntnis
Wie auch wir vergeben unseren früheren Dozenten
Und führe uns nicht in die Übung
Sondern erlöse uns von Robert Zeier
Denn dein sind die Studenten
Und die Mitarbeiter
Und die Drittmittel
In Ewigkeit.
Amen.

A Algorithmensammlung

Die folgende Zusammenstellung von Algorithmen ist nicht streng geordnet. Die Reihenfolge orientiert sich grob am Vorlesungsverlauf.

A.1 Der Euklidische Algorithmus

Sei R ein euklidischer Ring. Dann existiert zu $u, v \in R$ ein größter gemeinsamer Teiler (ggT, greatest common divisor, gcd). Dieser läßt sich mit Hilfe des Euklidischen Algorithmus bestimmen.

EUKLIDS Algorithmus

Die Laufzeit des Algorithmus ist nach oben beschränkt durch $O(\log n)$, wobei $n = \max\{\partial u, \partial v\}$ ist. Diese Laufzeit wird erreicht, wenn u und v zwei aufeinanderfolgende FIBONACCI-Zahlen sind.

Eine Verallgemeinerung des Algorithmus von EUKLID ist Algorithmus 3.

Algorithmus 1: Der Euklidische Algorithmus

```
function gcd(u, v)
  while v ≠ 0 do
    (u, v) = (v, u mod v);
  return u;
```

A.2 Die Pollardsche ϱ-Methode

Algorithmus 2: Faktorisierung mit der Pollardschen ϱ-Methode

```
function factorize(n)
  if isPrime(n) then
    return {n};
  i = 1;  x = random(2, n − 2);
  y = x;  k = 2;
  while true do
    i = i + 1;
    x = x^2 − 1 mod n;
    d = gcd(y − x, n);
    if d ≠ 1 and d ≠ n then
      return factorize(d) ∪ factorize(n/d);
    if i = k then
      y = x;  k = 2 · k;
```

Algorithmus 2 bestimmt die Primfaktorzerlegung einer vorgelegten Zahl n mit Hilfe eines probabilistischen Verfahrens. Im Mittel terminiert der Algorithmus nach einer Laufzeit von $O(\sqrt[4]{n})$.

Weitere Einzelheiten zu diesem Algorithmus in Abschnitt 2.2.

A.3 ELBA

Sei R ein euklidischer Ring und $u, v \in R$. Dann existieren Elemente $r, s \in R$ mit

$$\mathrm{ggT}(u, v) = t = u \cdot r + v \cdot s.$$

ELBA

ELBA bestimmt zu vorgegebenen u und v die zugehörigen Ringelemente r, s und t. Dazu ist ein Aufwand von $O(\log n)$ nötig. ELBA ist eine Verallgemeinerung des Euklidischen Algorithmus (Algorithmus 1, vgl. auch Abschnitt 2.1).

multipl. Inverse

Mit Hilfe dieses Algorithmus lassen sich insbesondere auch *multiplikative Inverse* in $\mathbb{Z}_p$ ($p \in \mathbb{N}$ prim) bestimmen: Ist etwa $m \in \mathbb{Z}_p$, so gilt $\mathrm{ggT}(m, p) = 1$ und damit

$$1 = \mathrm{ggT}(m, p) = m \cdot r + \underbrace{p \cdot s}_{\equiv 0 \bmod p},$$

also $m^{-1} = r \bmod p$.

Algorithmus 3: ELBA

function *elba*(u, v)
  $(t, r, s, t', r', s') = (u, 1, 0, v, 0, 1)$;
  while $t' \neq 0$ **do**
    $q = t \operatorname{div} t'$;
    $(t, r, s, t', r', s') = (t', r', s', t - q \cdot t', r - q \cdot r', s - q \cdot s')$;
  return (t, r, s);

A.4 Polynom- und Langzahlmultiplikation nach Karatsuba

p und q seien Polynome über einem Körper $\mathbb{F}$, o. B. d. A. sei $\partial p = \partial q = 2^n =: d$. Die Berechnung von $p \cdot q$ mit einem naiven Algorithmus hat einen Aufwand von $O(n^2)$. Durch Einsparung einer Multiplikation pro Rekursionsschritt kommt Karatsubas Algorithmus mit $O(n^{\log_2 3}) \approx O(n^{1.58})$ aus.

Setzt man

$$p = \sum_{k=0}^{d} p_k x^k = \underbrace{x^{d/2} \sum_{k=d/2}^{d} p_k x^{k-d/2}}_{=:P_1} + \underbrace{\sum_{k=0}^{d/2-1} p_k x^k}_{=:P_2}$$

$$q = \sum_{k=0}^{d} q_k x^k = \underbrace{x^{d/2} \sum_{k=d/2}^{d} q_k x^{k-d/2}}_{=:Q_1} + \underbrace{\sum_{k=0}^{d/2-1} q_k x^k}_{=:Q_2},$$

so gilt

$$\begin{aligned} p \cdot q &= P_1 Q_1 x^d + (P_1 Q_2 + P_2 Q_1) x^{d/2} + P_2 Q_2 \\ &= P_1 Q_1 x^d + (P_1 Q_1 + P_2 Q_2 + P_1 Q_2 - P_2 Q_2 - P_1 Q_1 + P_2 Q_1) x^{d/2} + P_2 Q_2 \\ &= P_1 Q_1 x^d + (P_1 Q_1 + P_2 Q_2 + (P_1 - P_2)(Q_2 - Q_1)) x^{d/2} + P_2 Q_2. \end{aligned}$$

Algorithmus 4: Polynommultiplikation nach Karatsuba

function *multiply*(p, q)
  $d := max(\partial p, \partial q)$; // Zweierpotenz
  if $d = 1$ **then**
    return $p_0 \cdot q_0$;
  $P_1 := p_{d/2} + \cdots + p_d x^{d/2}$; $P_2 := p_0 + \cdots + p_{d/2-1} x^{d/2-1}$;
  $Q_1 := q_{d/2} + \cdots + q_d x^{d/2}$; $Q_2 := q_0 + \cdots + q_{d/2-1} x^{d/2-1}$;
  $R_1 := multiply(P_1, Q_1)$;
  $R_2 := multiply(P_2, Q_2)$;
  $R_3 := multiply(P_1 - P_2, Q_2 - Q_1)$;
  return $R_1 x^d + (R_1 + R_2 + R_3) x^{d/2} + R_2$

Der Algorithmus kann zur Multiplikation von Langzahlen eingesetzt werden. Hierzu ersetzt man x durch die Basis des Zahlensystems (z. B. 2 oder 10) und überträgt evtl. entstehende Überläufe in den nächsthöheren Koeffizienten.

A.5 Der Rijndael-Algorithmus

Siehe Kapitel 3 (S. 21) oder [DaRi00].

A.6 Der Kruskal-Algorithmus

Sei $G = (V, E, w)$ ein ungerichteter, gewichteter Graph. $w\colon E \to \mathbb{R}$ sei eine Gewichtungsfunktion der Kanten E.

Der Kruskal-Algorithmus bestimmt einen minimalen Spannbaum von G. Ein Untergraph U von G heißt *Spannbaum*, wenn U ein Baum (d. h. zyklenfrei) ist, der alle Knoten aus V umfaßt. Ein Spannbaum heißt *minimal*, wenn für ihn die Summe der Kantengewichte minimal unter allen Spannbäumen ist.

minimaler Spannbaum

Ohne Beschränkung der Allgemeinheit sei im Folgenden G ein vollständiger Graph (Existiert eine Kante (u, v) nicht, setze $w(u, v) := 0$).

Algorithmus 5: Der Kruskal-Algorithmus

function *kruskal*(V, w)
  $T := \varnothing$; $v := minimumEdge(V, w)[0]$;
  for $u \in V$ **do**
    $u.cost := w(v, u)$; $u.neighbour := v$; $u.marked := false$;
  $v.marked := true$;
  while $\exists\, v \in V : \neg v.marked$ **do**
    $v := minimumVertex(V); v.marked = true$;
    $T := T \cup \{(v, v.neighbour)\}$;
    for $z \in V$ **do**
      if not $u.marked$ **and** $w(v, u) < u.cost$ **then**
        $u.cost = w(v, u)$;
        $u.neighbour = v$;
  return (V, T);

Die Methode *minimumEdge* liefere das Paar $(v_1, v_2) \in V \times V$, für die $w(v_1, v_2)$ minimal wird. *minimumVertex* liefert den Knoten $v \in V$, für den $v.cost$ minimal unter den noch nicht markierten Knoten ist.

Die Komplexität des Algorithmus beträgt $O(|V| + |E|)$.

A.7 Flüsse in Netzwerken

Wir betrachten einen gerichteten Graph $G = (V, E)$ mit einer Gewichtsfunktion $c\colon E \to [0, \infty)$. Ein Knoten $q \in V$ heißt *Quelle* von G, wenn von ihm aus ein Pfad zu jedem anderen Knoten, jedoch von keinem anderen Knoten eine Kante nach q existiert. Ein Knoten $s \in V$ heißt *Senke* von G, wenn von jedem Knoten ein Pfad nach s existiert, jedoch von s selbst keine Kanten ausgehen. Verfügt G über eine Quelle und eine Senke, so heißt G *Netzwerk*.

Netzwerk

Sei G ein Netzwerk. Dann gilt: G ist zusammenhängend und G hat genau eine Quelle und genau eine Senke.

Eine Funktion $f\colon E \to \mathbb{R}$ heißt *Fluß*, falls folgendes gilt:

$$\begin{aligned} \forall\, e \in E: &\quad |f(e)| \le c(e), \\ \forall\, u, v \in V: &\quad f(u,v) = -f(v,u), \\ \forall\, v \in V \setminus \{q, s\}: &\quad \sum_{(u,v) \in E} f(u,v) = 0. \end{aligned}$$

Der Wert $|f|$ eines Flusses f ist definiert durch $|f| := \sum_{v \in V} f(s, v)$.

Gesucht ist zu gegebenen $G = (V, E)$ und c der Fluß f, für den $|f|$ maximal wird.

Es sei $[q, s]$ die Menge aller Pfade von q nach s. Die *Restkapazität* $c_f(p)$ eines Pfades p bezüglich eines Flusses f ist definiert als

$$c_f(p) := \min_{(u,v) \in p} \{c(u,v) - f(u,v)\}.$$

Zu jedem Pfad p sei schließlich

$$f_p\colon V \times V \to \mathbb{R}, \qquad f_p(u,v) := \begin{cases} c_f(p) & (u,v) \in p \\ -c_f(p) & (v,u) \in p \\ 0 & \text{sonst} \end{cases}$$

Mit den nun zur Verfügung stehenden Bezeichnungsweisen läßt sich der maximale Fluß eines Netzwerks durch den folgenden Algorithmus bestimmen:

Algorithmus 6: Bestimmung des maximalen Flusses nach Ford und Fulkerson

function *maximumFlow*(V, E, q, s, c)
  $f := 0$;
  while $\exists\, p \in [q, s] : c_f(p) \neq 0$ **do**
    $f = f + f_p$;
  return f;

A.8 Interpolation nach Newton

Es sei $\mathbb{F}$ ein Körper und $(x_i, y_i) \in \mathbb{F}^2$ $(i = 1, \ldots, n)$ mit $x_i \neq x_j$ $(i \neq j)$. Gesucht ist das eindeutig bestimmte Polynom $p \in \mathbb{F}[x]$ kleinsten Grades mit $p(x_i) = y_i$ $(i = 1, \ldots, n)$.

Der folgende Interpolationsalgorithmus löst dieses Problem mit einem Aufwand von $O(n^2)$. In Abschnitt 4.3 erscheint das Interpolationsproblem und der hier vorgestellte Algorithmus als Spezialfall eines allgemeineren Zusammenhangs.

Algorithmus 7: Interpolation nach Newton

```
function newtonPolynom(x[], y[])
    M(x) := 1;  p(x) = y[0];
    for k = 1 to n − 1 do
        M(x) := M(x) · (x − x[k − 1]);
        s := (y[k] − p(x[k]))/M(x[k]);
        p(x) := p(x) + s · M(x);
    return p(x);
```

B Theoretische Grundlagen und mathematische Hilfsmittel

Dieser Abschnitt enthält mathematische Definitionen und Sätze, die in der Vorlesung verwendet wurden. Mathematische Aussagen, die hier nicht aufgeführt sind, sind möglicherweise bei [DniQ00] zu finden. Die mathematische Nomenklatur entspricht den Schreibweisen in [Kau00]. Für eine Einführung in Algebra ist das Lehrbuch [Bos99] zu empfehlen.

B.1 Algebraische Strukturen

Definition 1 Sei R ein Ring[16]. Die multiplikative Gruppe

$$R^\times := \{a \in R : \; \exists\, b \in R : \; a \cdot b = b \cdot a = 1\}$$

heißt *Einheitengruppe ($R^\times$)* von R. Die Elemente von $R^\times$ heißen *Einheiten*. Einheit

Bsp. 28 $\mathbb{Z}^\times = \{-1, 1\}$, $\mathbb{F}[x]^\times = \mathbb{F}^\times = \mathbb{F} \setminus \{0\}$ ($\mathbb{F}$ beliebiger Körper). □

Definition 2 Sei R ein Ring und $p \in R$.

1. p heißt *prim*, falls $p \notin R^\times \cup \{0\}$ und prim

$$\forall\, u, v \in R : \quad p \mid u \cdot v \implies p \mid u \;\vee\; p \mid v.$$

2. p heißt *irreduzibel* , falls $p \notin R^\times \cup \{0\}$ und irreduzibel

$$\forall\, u, v \in R : \quad p = u \cdot v \implies u \in R^\times \vee v \in R^\times.$$

Satz 3 Sei R ein Ring. Dann gilt:

$$\forall\, p \in R : \quad p \text{ prim} \implies p \text{ irreduzibel}.$$

In faktoriellen Ringen sind die Begriffe prim und irreduzibel äquivalent. Jedes Element $n \in R$ hat eine eindeutige *Primfaktorzerlegung*, d. h. es gibt eine Einheit $e \in R^\times$ und Primelemente $p_0, \ldots, p_k \in R$ mit

$$n = e \cdot \prod_{i=0}^{k} p_i.$$

Die Elemente p_i sind bis auf Permutation eindeutig bestimmt.

Definition 4 Sei R ein Ring.

Ideal

1. Eine Menge $I \subseteq R$ heißt *Ideal*, falls $(I,+)$ Untergruppe von $(R,+)$ ist und folgendes gilt:

$$\forall\, r \in R\ \forall\, a \in I: \quad r \cdot a \in I.$$

Ist I ein Ideal von R, so schreibt man $I \trianglelefteq R$.

Erzeugenden-system

2. Sei $I \trianglelefteq R$. $E \subseteq I$ heißt *Erzeugendensystem* von I, falls gilt

$$I = \sum_{e \in E} R \cdot e.$$

Ist E ein Erzeugendensystem von I, so schreibt man $I = (E)$.

Hauptideal

3. Ein Ideal $I \trianglelefteq R$ heißt *Hauptideal*, falls es von einem einzigen Element $e \in R$ erzeugt wird. Man schreibt dann auch $(e) := (\{e\}) = I$ und nennt e den *Erzeuger* von I.

Bsp. 29 Es gilt z. B. $\mathbb{R}[x]/(x^2+1) \cong \mathbb{C}$. $\square$

Definition 5 Sei R ein Ring.

Eins

1. R heißt *Ring mit Eins*, falls ein Element $1 \in R$ existiert mit

$$\forall\, r \in R: \quad r \cdot 1 = r.$$

In diesem Fall heißt 1 *Eins*.

nullteilerfrei

2. R heißt *Integritätsbereich* oder *nullteilerfrei* oder *integer*, falls gilt:

$$\forall\, a, b \in R: \quad a \cdot b = 0 \implies a = 0 \ \vee\ b = 0.$$

faktoriell

3. R heißt *faktoriell*, falls zu jedem Element $r \in R \setminus R^{\times}$ eine eindeutige Primfaktorzerlegung existiert.

4. R heißt *Hauptidealring*, falls jedes Ideal $I \trianglelefteq R$ von einem Element erzeugt ist.

Hauptidealring

euklidisch

5. R heißt *euklidisch*, falls es eine Funktion $\partial\colon R \to \mathbb{N}_0 \cup \{\infty\}$ gibt, so daß für alle $a, b \in R$ mit $b \neq 0$ gilt $\partial(a \cdot b) \geq \partial(a)$ und

$$\exists\, q, r: \quad a = q \cdot b + r \ \wedge\ (r = 0 \ \vee\ \partial r < \partial b).$$

Man schreibt $q = a \operatorname{div} b$ und $r = a \bmod b$ und nennt ∂ die *Gradfunktion* von R.

Körper

6. R heißt *Körper*, falls $R^{\times} = R \setminus \{0\}$ gilt.[17]

[16]In dieser Vorlesung betrachten wir nur kommutative Ringe.

[17]Diese Definition stimmt mit der üblichen überein.

Satz 6 Es gilt (vgl. Abb. B.1)

1. Jeder Körper ist ein euklidischer Ring.
2. Jeder euklidische Ring ist Hauptidealring.
3. Jeder Hauptidealring ist faktoriell.
4. Jeder faktorielle Ring ist nullteilerfrei.

Bsp. 30

1. $\mathbb{F}_2$ ist ein Körper.
2. $\mathbb{Z}$ ist ein euklidischer Ring, aber kein Körper.
3. $\mathbb{Z}[\frac{1}{2}(1+\sqrt{-14})]$ ist ein Hauptidealring, aber nicht euklidisch.
4. $\mathbb{Z}[x]$ ist faktoriell, aber kein Hauptidealring.
5. $\mathbb{Z}[-\sqrt{5}]$ ist nullteilerfrei, aber nicht faktoriell.
6. ${}^{\mathbb{Q}[x]}/_{(x^2)}$ ist Ring mit Eins, aber nicht nullteilerfrei.
7. $2\mathbb{Z}$ ist ein Ring ohne Eins. □

B.2 Ausgewählte Sätze und Definitionen

Definition 7 Sei R ein faktorieller Ring und $a, b \in R$. Ein Element g heißt *größter gemeinsamer Teiler* von a und b, falls gilt

ggT

$$\forall\, x \in R: \quad x \mid g \implies x \mid a \;\wedge\; x \mid b.$$

In diesem Fall schreibt man $g = \mathrm{ggT}(a, b)$.

Satz 8 Sei R ein euklidischer Ring. Dann gilt:

1. Zu $a, b \in R$ existiert genau ein $g \in R$ mit $aR + bR = gR$. Es ist $g = \mathrm{ggT}(a, b)$.[18]
2. Zu $a, b \in R$ existieren Elemente $r, s \in R$ mit

$$a \cdot r + b \cdot s = \mathrm{ggT}(a, b).$$

Chinesischer Restsatz

Satz 9 (Chinesischer Restsatz)

1. Sei R Hauptidealring, $a \in R$ und $p_1^{\nu_1} \cdots p_r^{\nu_r}$ eine Primelementzerlegung von a. Dann gilt

$$R/(a) \cong R/(p_1^{\nu_1}) \times \cdots \times R/(p_r^{\nu_r}).$$

算

Ist φ ein Isomorphismus, so wird die Umkehrabbildung φ^{-1} als 算 (*suan*) bezeichnet.

2. Sei R ein euklidischer Ring. $m_1, \ldots, m_r \in R$ seien koprim, d. h.

$$\forall\ i \neq j: \quad \mathrm{ggT}(m_i, m_j) = 1.$$

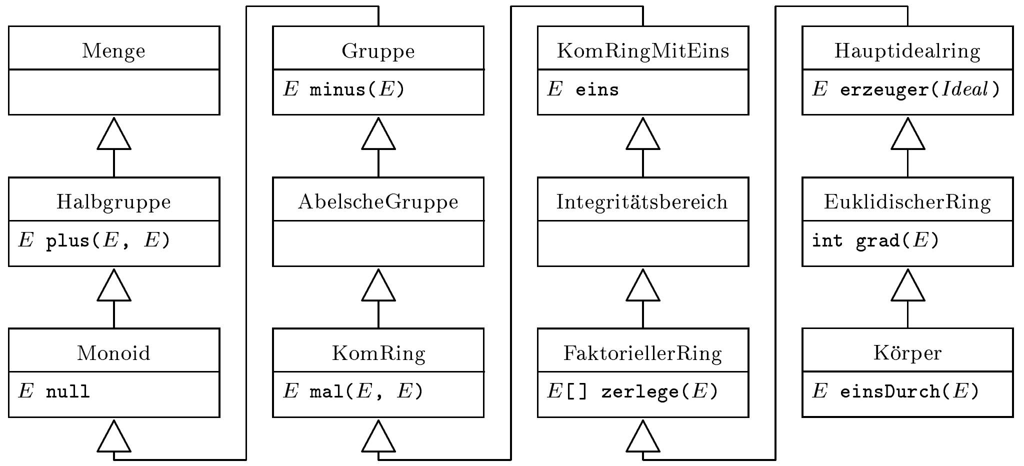
Für $i = 1, \ldots, r$ seien weiter Elemente x_i mit $\partial x_i \leq \partial m_i$ vorgegeben. Dann besitzt das Gleichungssystem

$$x = x_i \bmod m_i \qquad (i = 1, \ldots, r)$$

genau eine Lösung x mit $\partial x < \partial \prod_{i=1}^{r} m_i$.

3. Der Chinesische Restsatz läßt sich allgemein für beliebige Ringe formulieren. Vergleiche dazu [Bos99], Seite 49.

Abbildung B.1: Abhängigkeiten zwischen algebraischen Strukturen als UML-Diagramm



Homomorpiesatz

Satz 10 (Homomorphiesatz, Einbettungssatz) Seien R und S Datenstrukturen und $\varphi\colon R \to S$ ein Homomorphismus. Weiter sei $\pi\colon R \to R/\ker\varphi$ die

[18] Insbesondere ist $\mathrm{ggT}(a, b)$ in euklidischen Ringen damit eindeutig bestimmt. Im Allgemeinen ist das nicht der Fall.

kanonische Abbildung. Dann existiert ein Isomorphismus $\widetilde{\varphi}\colon {}^{R}\!/_{\ker\varphi} \to S$, für den das Diagramm

$$\begin{tikzcd} W \arrow[r] & R \arrow[rr, "\varphi"] \arrow[dr, "\pi"'] & & S \\ & & {}^{R}\!/_{\ker\varphi} \arrow[ur, "\widetilde{\varphi}"'] & \end{tikzcd}$$

kommutiert. Jede partielle Struktur $W \subseteq R$ wird durch S treu modelliert, falls $|W \cap (\ker\varphi)| \leq 1$ gilt.

Definition 11 (Eulersche φ-Funktion) Die Funktion

$$\varphi\colon \mathbb{N} \to \mathbb{N}, \qquad \varphi(n) := |\mathbb{Z}_n^\times|$$

heißt *Eulersche φ-Funktion.*

Satz 12

1. Ist $p \in \mathbb{N}$ prim, so gilt $\varphi(p) = p - 1$. Allgemein gilt

$$\varphi(n) = n \cdot \prod_{\substack{p \mid n \\ p \text{ prim}}} \left(1 - \frac{1}{p}\right).$$

2. (Satz von Fermat) FERMAT

$$\forall\, p \in \mathbb{N} \text{ prim } \forall\, a: \quad p \nmid a \implies a^{p-1} \equiv 1 \bmod p.$$

3. (Satz von Euler) EULER

$$\forall\, n \in \mathbb{N}\ \forall\, a \in \mathbb{N}: \quad \operatorname{ggT}(a, n) = 1 \implies a^{\varphi(n)} \equiv 1 \bmod n.$$

B.3 Algebra bis zum Abwinken

13. (Relatoren) Sei Ω eine Menge von Symbolen. Die Verknüpfung $*$ sei definiert als „Aneinanderhängung" (*Konkatenation*) von Symbolsequenzen, etwa Relatoren

$$\texttt{thomas} * \texttt{beth} = \texttt{thomasbeth}.$$

Derartige Symbolsequenzen heißen *Worte*. Mit ε sei das *leere Wort* bezeichnet.

$$\Omega^* := \{\varepsilon\} \cup \Omega \cup \{p_0 * \cdots * p_n : p_i \in \Omega\ (i = 0, \ldots, n)\}$$

ist dann die Menge aller endlichen Worte auf dem *Alphabet* Ω. $(\Omega^*, *)$ ist ein Monoid.

Sei $\overline{\Omega} := \Omega \cup \{\overline{a} : a \in \Omega\}$, wobei für jedes $a \in \Omega$ mit $\overline{a}$ ein neues, noch nicht in Ω liegendes Symbol gemeint ist, für das definiert wird $\overline{a} * a := a * \overline{a} := \varepsilon$. Dann ist $(\overline{\Omega}^*, *)$ eine Gruppe, wenn man für $\omega = \omega_1 * \omega_2 \in \overline{\Omega}^*$ definiert

$$\omega^{-1} = (\omega_1 * \omega_2)^{-1} := \omega_2^{-1} * \omega_1^{-1}$$

und $\overline{a}^{-1} := a, a^{-1} := \overline{a}$ für $a \in \Omega$.

Zur Notation einer Faktorgruppe $N \leq \overline{\Omega}^*$ kann man Elemente aus $\overline{\Omega}^*$ vorgeben, die in N verschwinden sollen. Diese Elemente werden als *Relatoren* oder *definierende Relationen* bezeichnet.

Bsp. 31 Sei $\Omega = \{\mathtt{a}, \mathtt{b}\}$. Dann ist

$$\overline{\Omega}^* = \{\varepsilon, \mathtt{a}, \overline{\mathtt{a}}, \mathtt{b}, \overline{\mathtt{b}}, \mathtt{aa}, \mathtt{a}\overline{\mathtt{a}}, \mathtt{ab}, \mathtt{a}\overline{\mathtt{b}}, \mathtt{ba}, \mathtt{b}\overline{\mathtt{a}}, \dots\}$$

Ist $R = \{\mathtt{ab}\overline{\mathtt{a}}\overline{\mathtt{b}}\}$, so ist $\overline{\Omega}^*/(R) \cong \mathbb{Z} \times \mathbb{Z}$, denn $\mathtt{ab}\overline{\mathtt{a}}\overline{\mathtt{b}}$ bedeutet $\mathtt{ab} = \mathtt{ba}$ und damit läßt sich jedes Wort in $\overline{\Omega}^*$ in eine der Formen $\mathtt{a}^i\mathtt{b}^j$, $\overline{\mathtt{a}}^i\mathtt{b}^j$, $\mathtt{a}^i\overline{\mathtt{b}}^j$ oder $\overline{\mathtt{a}}^i\overline{\mathtt{b}}^j$ $(i, j \in \mathbb{N}_0)$ überführen. □

elliptische Kurven

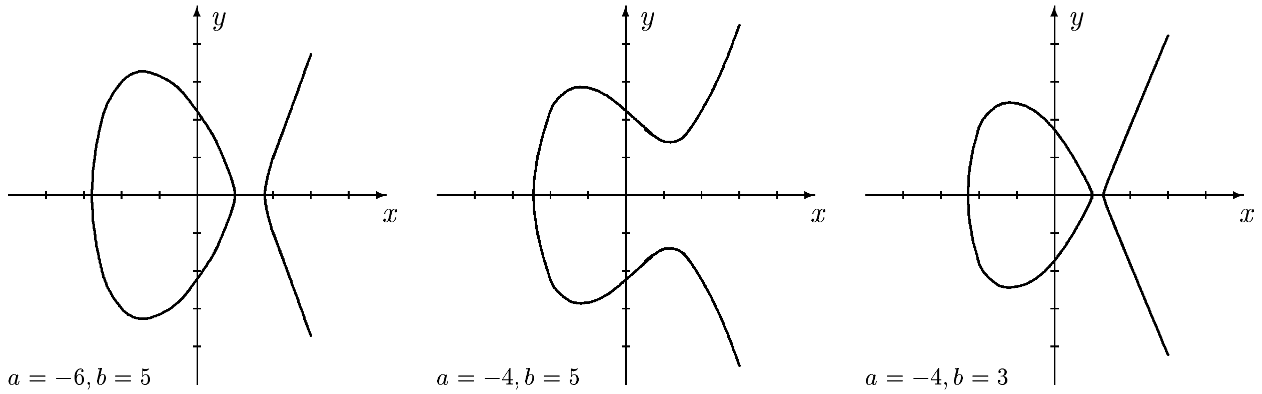
14. (Elliptische Kurven) Sei $\mathbb{F}$ ein Körper mit $2 \nmid |\mathbb{F}|$ und $3 \nmid |\mathbb{F}|$ sowie und $a, b \in \mathbb{F}$. Dann heißt die Menge

$$EC_{a,b} := E(\mathbb{F}) := \{(x, y) \in \mathbb{F}^2 : y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$$

elliptische Kurve über $\mathbb{F}$ zu a und b, falls das Polynom $x^3 + ax + b$ keine mehrfachen Nullstellen in $\mathbb{F}$ hat, d. h. falls $4a^3 + 27b^2 \neq 0$ ist.

Im Fall $\mathbb{F} = \mathbb{R}$ haben elliptische Kurven etwa die folgende Gestalt:

Abbildung B.2: Einige elliptische Kurven über $\mathbb{R}$



Der Punkt $\mathcal{O}$ kann als im Unendlichen liegend angenommen werden.

Zu einem Punkt r auf einer elliptischen Kurve sei $-r$ der Punkt, der durch Spiegelung von r an der x-Achse entsteht. (Elliptische Kurven sind stets symmetrisch zur x-Achse.)

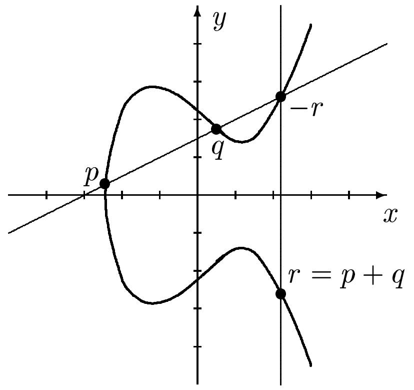
Sind p, q $(p \neq q)$ Punkte auf einer elliptischen Kurve, so kann man in den meisten Fällen eine Gerade durch p und q legen, die die Kurve in einem dritten Punkt $-r$

schneidet.[19] Definiert man in diesen Fällen $p+q := r$ und in allen anderen Fällen $p+q := \mathcal{O}$, dann ist $(EC_{a,b}, +)$ eine abelsche Gruppe mit Neutralelement $\mathcal{O}$.

Nach dem Satz von HASSE-WEIL gilt für die Kardinalität n einer elliptischen Kurve über $\mathbb{F}_q$ die Abschätzung

$$|n - q + 1| \leq 2\sqrt{q}.$$

Abbildung B.3: Addition auf elliptischen Kurven



B.4 Grundlagen der Fouriertransformation

Definition 15 Seien $A \in \mathbb{F}^{n\times n}$ und $B \in \mathbb{F}^{m\times m}$.

1. Die *direkte Summe* von A und B ist definiert durch

$$A \oplus B := \begin{pmatrix} A & \\ & B \end{pmatrix}.$$

2. Das KRONECKER-*Produkt* (auch *Tensorprodukt*) von A und B ist definiert durch

Tensorprodukt

$$A \otimes B := \begin{pmatrix} a_{0,0}B & \cdots & a_{0,n-1}B \\ \vdots & \ddots & \vdots \\ a_{n-1,0}B & \cdots & a_{n-1,n-1}B \end{pmatrix}.$$

In diesem Fall heißen die Koordinaten von A die *Grobkoordinaten* und die Koordinaten von B die *Feinkoordinaten* von $A \otimes B$.

Sind $A, C \in \mathbb{F}^{n\times n}$ und $B, D \in \mathbb{F}^{m\times m}$, so gilt:

1. $(A \otimes B) \cdot (C \otimes D) = (A \cdot C) \otimes (B \cdot D)$
2. $(A \oplus C) \otimes B = (A \otimes B) \oplus (C \otimes B)$
3. $A \otimes B = (I_n \otimes B) \cdot (A \otimes I_m) = (A \otimes I_m) \cdot (I_n \otimes A)$

[19] Im Fall $p = q$ betrachte die Tangente an q als Gerade.

Definition 16

1. Sei $a = (a_0, a_1, \ldots, a_{n-1}) \in \mathbb{F}^n$. Dann heißt
$$\mathrm{diag}(a) := \begin{pmatrix} a_0 & & \\ & \ddots & \\ & & a_{n-1} \end{pmatrix} \in \mathbb{F}^{n\times n}$$
die *Diagonalmatrix* bezüglich a. Die Matrix $I_n := \mathrm{diag}(1, 1, \ldots, 1) \in \mathbb{F}^{n\times n}$ heißt *Einheitsmatrix*.
Die gleiche Schreibweise wird für *Blockdiagonalmatrizen* verwendet.

2. Es sei S_n die *symmetrische Gruppe* der Ordnung n und $\sigma \in S_n$. Dann heißt
$$P_\sigma := ((\delta_{\sigma(i),j}))_{i,j=0}^{n-1} \in \mathbb{F}^{n\times n}$$
die *Permutationsmatrix* zu σ. Dabei bezeichnet
$$\delta_{ij} := \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$
das KRONECKER-*Symbol*. P_σ geht aus I_n hervor, indem man σ auf die Spalten von I_n anwendet.

Zirkulante

3. Matrizen A der Form $A = ((a_{j-i \bmod n}))_{i,j=0}^{n-1}$ heißen *Zirkulanten*. Schreibweise: $C_{Z_n}(a_0, \ldots, a_{n-1}) := A$.

4. Matrizen A der Form $A = ((a_{i+j \bmod n}))_{i,j=0}^{n-1}$ heißen *Antizirkulanten*.

Bsp. 32
$$C_{Z_4}(a, b, c, d) = \begin{pmatrix} a & b & c & d \\ d & a & b & c \\ c & d & a & b \\ b & c & d & a \end{pmatrix}, \qquad P_{(1\,3\,2\,4)} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}.$$
□

Einheitswurzel

Definition 17 Ein Element $\omega \in \mathbb{F}$ mit $\omega^n = 1$ heißt *n-te Einheitswurzel* in $\mathbb{F}$. Eine n-te Einheitswurzel heißt *primitiv*, falls $\omega^k \neq 1$ für $1 \leq k \leq n$ ist.

Ist ω eine n-te Einheitswurzel, so heißt die Matrix
$$\mathrm{DFT}_n^{(\omega)} := \mathrm{DFT}_n := ((\omega^{i\cdot j}))_{i,j=0}^{n-1} \in \mathbb{F}^{n\times n}$$
die *diskrete Fouriertransformation*.

Zu $a \in \mathbb{F}^n$ heißt $\widehat{a} := \mathrm{DFT}_n \cdot a$ die *Fouriertransformierte* von a.

Bsp. 33 Im Fall $\mathbb{F} = \mathbb{C}$ sind u. a. die Zahlen $\omega := \mathrm{e}^{-\frac{2\pi \mathrm{i}}{n}}$ und $\omega := \mathrm{e}^{+\frac{2\pi \mathrm{i}}{n}}$ primitive n-te Einheitswurzeln. Die Zahlen ω^i $(i = 0, \ldots, n-1)$ unterteilen den Einheitskreis der *Gaußschen Zahlenebene* in n gleich große Sektoren. □

Satz 18

1. Ist ω eine primitive n-te Einheitswurzel, so gilt

$$x^n - 1 = \prod_{i=0}^{n-1} (x - \omega^i),$$

   d. h. die Zahlen ω^i sind gerade die Nullstellen des Polynoms $x^n - 1$.

2. Ist ω eine primitive n-te Einheitswurzel, so gilt

$$(\{\omega^i : i = 0, \ldots, n-1\}, \cdot) \cong (\mathbb{Z}_n, +),$$

   d. h. die Zahlen ω^i bilden eine multiplikative zyklische Gruppe der Ordnung n. Existiert eine n-te Einheitswurzel, so existieren genau $\varphi(n)$ primitive Einheitswurzeln und genau n Einheitswurzeln.

3. Für jeden Vektor $a \in \mathbb{F}^n$ gilt

$$\mathrm{DFT}_n{}^{-1} \cdot C_{Z_n}(a) \cdot \mathrm{DFT}_n = \mathrm{diag}(\widehat{a}).$$

4. Es gilt

$$\mathrm{DFT}_n{}^{-1} \cdot P_{(1\,2\,\cdots\,n)} \cdot \mathrm{DFT}_n = \mathrm{diag}(1, \omega, \ldots, \omega^{n-1}).$$

Definition 19 Eine Matrix $A = ((a_{ij}))_{i,j=0}^n \in \mathbb{F}^{n \times n}$ heißt TOEPLIZ-*Matrix*, wenn gilt

$$a_{i,j} = a_{i+1,j+1} \qquad (i, j = 0, \ldots, n-2).$$

C Literaturverzeichnis

[ℶ84] Thomas Beth[20]:
Verfahren der schnellen Fouriertransformation
Teubner, 1984

[ℶ95] Thomas Beth:
Sichere offene Datennetze
Spektrum der Wissenschaft, Mai 1995

[Bos99] Siegfried Bosch:
Algebra
Springer-Verlag, 1999

[Cor90] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest:
Introduction to Algorithms
MIT Press, Juni 1990

[DaRi00] Joan Daemen, Vincent Rijmen:
AES Proposal: Rijndael
http://csrc.nist.gov/encryption/aes/, 2000

[DniQ00] Dominique Unruh:
Informatik IV – Versuch einer Gliederung
http://www.unruh.de/DniQ/~ueb/info_iv_vorlesung.ps.gz, 2000

[Gei00] Willi Geiselmann:
Public Key Kryptographie
I A K S Karlsruhe, Februar 2000

[Kau00] Manuel Kauers:
Vorlesungsmitschriften
http://www.kauers.de/skripten.html, 1999–2000

[Knu81] Donald E. Knuth:
The Art of Computer Programming
second edition, Addison-Wesley, January 1981

[Lip81] John D. Lipson:
Elements of Algebra and Algebraic Computing
Addison-Wesley, 1981

[Sch97] Uwe Schöning:
Algorithmen – kurz gefasst
Spektrum Hochschultaschenbuch, 1997

[Sed98] Robert Sedgewick:
Algorithms in C++
Addison-Wesley, 1998

[20] Das Symbol ℶ (*beth*) ist nach ℵ (*aleph*) der zweite Buchstabe im hebräischen α-ℶ.

D Index

Symbolverzeichnis

$O(f(n))$	O-Kalkül-Abschätzung von $f(n)$	$R^{n\times m}$	Ring der $n \times m$-Matrizen über R
$\mathbb{Z}$	Menge der ganzen Zahlen	DFT_n	Diskrete *Fouriertransformation* der Größe n
$\mathbb{Z}_m$	Restklassenring der Größe m	$f * g$	*Faltung* von f und g
ggT	*größter gemeinsamer Teiler*	$f \odot g$	komponentenweise Multiplikation
$a \bmod b$	$b \cdot (a \operatorname{div} b)$	$\widehat{f}$	*Fouriertransformierte* von f
$a \operatorname{div} b$	Ganzzahldivision	I_n	*Einheitsmatrix*
$\lvert M\rvert$	Mächtigkeit der Menge M	$A \otimes B$	*Tensorprodukt*
$P(X)$	Wahrscheinlichkeit für X	$\mathrm{diag}(a)$	*Diagonalmatrix*
$a \equiv b \bmod m$	$a = b$ in $\mathbb{Z}_m$	P_σ	*Permutationsmatrix*
$m \mid n$	m teilt n	$C_{Z_n}(a)$	*Zirkulante*
$\ker(\sim)$	Kern der Kongruenzrelation $\sim$	$I \trianglelefteq R$	I ist *Ideal* von R
$R^\times$	*Einheitengruppe* von R	(E)	von E erzeugtes Ideal
$E\colon A \twoheadrightarrow B$	*Einwegfunktion*	$\langle E\rangle$	allg. Erzeugnis von E
$\mathbb{F}_p$	Körper mit p Elementen	∂	*Gradfunktion*
$R[x]$	Polynomring über R	φ	*Eulersche φ-Funktion*
$a \oplus b$	bitweises **xor** (in Alg. 3.2)	$a * b$	*Konkatenation*
$\varkappa(n)$	*Komplexität* (worst-case)	ε	*leeres Wort*
$\alpha(n)$	*Komplexität* (amortisierte)	Ω^*	Menge der *Worte* über Ω
$d_H(a,b)$	*Hamming-Distanz* zw. a und b	$EC_{a,b}, E(\mathbb{F})$	*elliptische Kurve*
$\mathcal{P}(M)$	Potenzmenge von M	$\mathcal{O}$	Neutralelement in $E(\mathbb{F})$
算	*suan*	$A \oplus B$	*direkte Matrixsumme*
竹	*zhu*	$\delta_{i,j}$	*Kronecker-Symbol*
目	*mu*	ב	*beth*
廾	*gong*	א	*aleph*
$\det A$	Determinante von A		
$((a_{ij}))_{i,j=0}^{n-1}$	Matrix mit den Einträgen a_{ij}		

Stichwortverzeichnis