Technische Universität Ilmenau
Fachgebiet Komplexitätstheorie und Effiziente Algorithmen
Univ.-Prof. Dr. Martin Dietzfelbinger

Sommersemester 2021
Modul „Randomisierte Algorithmen“ (3V + 1Ü, 5 LP)
Online-Kurs

Präsentation von Vorlesung und Übung in Videos, nicht an fixe Zeiten gebunden.

Abschluss für Bachelorstudiengang Informatik: Mündliche Prüfung (30 Min.)

Andere Interessent/inn/en: sehr willkommen.

Gute Basis für „Kryptographie“ im WS 2022 und für weitere Arbeiten (auch Bachelorarbeit, Vertiefung im Masterstudiengang) im Fachgebiet.

Erwünschte Vorkenntnisse: Algorithmen und Datenstrukturen, Stochastik für Informatiker, Grundlagen und Diskrete Strukturen, Mathematik-Grundlagen

Inhalt

Kapitel 1 - Einführung und Beispiele

- Was sind randomisierte Algorithmen? Wozu sind sie gut?
- MinCut
- Verifikation von Matrixprodukten
- Gleichheit von Polynomprodukten (eine Variable)
- Quicksort

Kapitel 2 - Grundlagen aus der Wahrscheinlichkeitsrechnung

- W-Räume, Zufallsvariable, Erwartungswerte, Varianz, bedingte Wahrscheinlichkeiten, Unabhängigkeit
- Grundlegende Ungleichungen: Markov, Chebychev(-Cantelli), Jensen, Chernoff/Hoeffding.

Kapitel 3 - Modellierung und Transformationen

- Maschinenmodell und Wahrscheinlichkeitsräume für randomisierte Algorithmen
- Monte-Carlo-Algorithmen (machen Fehler)
- Wahrscheinlichkeitsverbesserung

- Las-Vegas-Algorithmen (fehlerfrei)

Kapitel 4 - Randomisiertes Suchen

- Zweifach unabhängige Suche
- Suchen in Listen
- Mediansuche
- Erfüllende Belegung für erfüllbare Boolesche Formeln in 3-KNF

Kapitel 5 - Algorithmen für Probleme aus der Zahlentheorie

- Zahlentheoretische Konzepte und Algorithmen
- Primzahltests, Primzahlerzeugung
- Quadratwurzeln modulo p

Kapitel 6 - Algebraische Methoden

- Satz von Schwartz-Zippel: Nullstellen von multivariaten Polynomen
- Polynomdeterminanten
- Vergleich von Polynomprodukten (mehrere Variable)
- Textvergleich,Textsuche
- Äquivalenz von Branchingprogrammen
- Existenz und Berechnung von perfekten Matchings in [bipartiten] Graphen

Literatur:

- **Skript** wird kapitelweise zur Verfügung gestellt.
- R. Motwani, P. Raghavan, „Randomized Algorithms“, Cambrigde University Press, 1995 (ein ziemlich umfassendes Standardwerk).
- J. Hromkovic, "Randomisierte Algorithmen", Teubner, 2004 (einführend)
- M. Mitzenmacher, E. Upfal, „Probability and Computing“, 2. Auflage, Cambridge University Press, 2017 (Fokussiert auf Wahrscheinlichkeitsmodelle für die Informatik, aber auch Algorithmen, viel Information über diskrete Situationen, startet bei Grundlagen, führt weit in die Theorie hinein).
- U. Schöning, „Algorithmik“, Spektrum Akademischer Verlag, 2001 (Kapitel 12).
- T. Cormen, C. Leiserson, R. Rivest, C. Stein, „Introduction to Algorithms“, MIT Press, 2001 (Umfassendes Standardwerk zu Algorithmen, eine deutschsprachige Ausgabe ist im Oldenbourg-Verlag erhältlich).

Offizielle Informationen: `https://www.tu-ilmenau.de/modultafeln/Informatik/Bachelor/2013/fach/4350/`

1 Einführung und Beispiele

Die Vorlesung „Randomisierte Algorithmen“ befasst sich mit Algorithmen, die Zufallsexperimente durchführen. Dabei stellt man sich vor, dass dem Algorithmus bzw. dem ausführenden Rechner eine Operation „Wähle eine zufällige Zahl aus $\{1, \ldots, k\}$“ zur Verfügung steht, mit k als Parameter. Während der Ausführung des Algorithmus können viele solche Experimente durchgeführt werden. Oft bestimmt dann die Eingabe x die Ausgabe $\mathcal{A}(x)$ von Algorithmus $\mathcal{A}$ auf x nicht mehr eindeutig; vielmehr ist dies eine Zufallsgröße. Von der Verwendung randomisierter Algorithmen verspricht man sich einen Effizienzgewinn, meistens bezüglich der Rechenzeit. – Wir beginnen mit *Beispielen* für den Entwurf und die Analyse solcher Verfahren.[1]

1.1 Randomisierter MinCut-Algorithmus

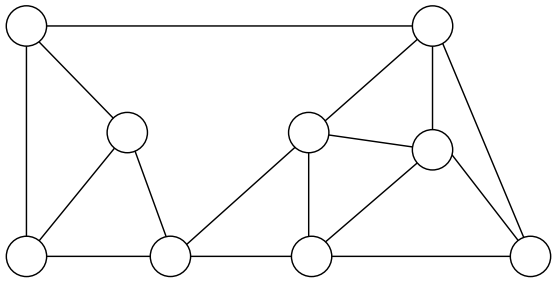


Abbildung 1.1.1: Ein zusammenhängender Graph $G = (V, E)$.

In diesem Abschnitt betrachten wir zusammenhängende (ungerichtete) Graphen, bezeichnet mit $G = (V, E)$, s. Abb. 1.1.1 für ein Beispiel. Wie üblich bezeichnen $n = |V|$

[1] In diesem einführenden Kapitel verwenden wir Konzepte und Tatsachen aus der Wahrscheinlichkeitsrechnung ohne weiteren Kommentar. Grundlagen aus der Wahrscheinlichkeitsrechnung werden in Kapitel 2 im Detail dargestellt bzw. wiederholt.

die Knotenzahl und $m = |E|$ die Kantenzahl. Ein *Schnitt* in G ist eine Menge von Kanten, deren Entfernen den Zusammenhang zerstört.[2]

Definition 1.1.1
Ein **Schnitt** in einem zusammenhängenden Graphen $G = (V, E)$ ist eine Kantenmenge $C \subseteq E$ mit der Eigenschaft, dass $(V, E - C)$ nicht zusammenhängend ist.
Ein Schnitt C heißt **minimal**, wenn es keinen Schnitt C' mit $|C'| < |C|$ gibt (Schnitt minimaler Kardinalität).

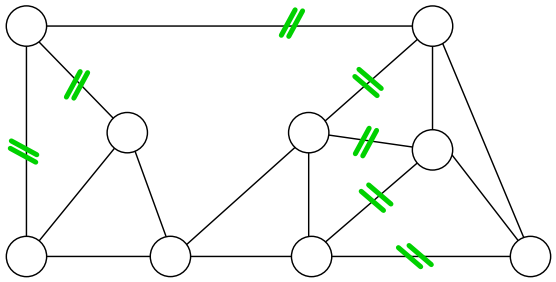


Abbildung 1.1.2: (Grün) Markierte Kanten: Schnitt C, nicht minimal.

In diesem Abschnitt betrachten wir das **MinCut-Problem**, das ist die Aufgabe, in einem gegebenen Graphen einen minimalen Schnitt zu finden. In Abb. 1.1.2 ist ein nicht minimaler Schnitt eingezeichnet, in Abb. 1.1.3 ein minimaler Schnitt.

Wir benutzen einen randomisierten Ansatz, das heißt, dass der Algorithmus „Zufallsexperimente ausführt“. Wir bauen schrittweise einen Graphen (V, R) mit $R \subseteq E$ auf. Wir beginnen mit $R = \emptyset$. Dann führen wir wiederholt den folgenden Schritt („Runde“) aus:

[2]Die technische Relevanz von Schnitten etwa für Graphen, die Kommunikationsnetzwerke darstellen, sollte offensichtlich sein. Netzwerke, die keine kleinen Schnitte haben, sind robuster gegen Verbindungsausfälle als solche, bei denen das Entfernen von wenigen Kanten dazu führt, dass das Netzwerk in Teile zerfällt. Wenn man einen sehr kleinen Schnitt findet, muss man das Netzwerk noch verbessern; wenn es keinen sehr kleinen Schnitt gibt, kann man das Netzwerk als „robust“ betrachten (gegen den Ausfall einzelner Verbindungen).

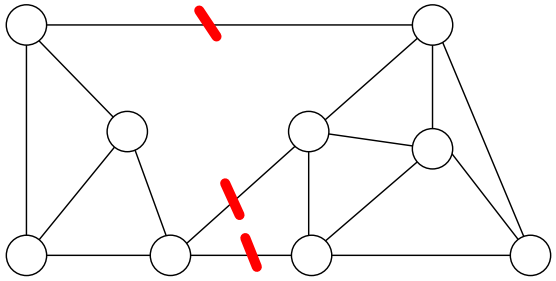


Abbildung 1.1.3: (Rot) Markierte Kanten: Minimaler Schnitt C.

> Wähle *zufällig* eine Kante e aus denjenigen Kanten aus, die zwei Zusammenhangskomponenten von (V, R) verbinden. Füge e zu R hinzu.
> (Dadurch werden zwei Zusammenhangskomponenten zu einer vereinigt.)

Dies wird iteriert, bis (V, R) genau zwei Zusammenhangskomponenten hat. Weil man mit n Zusammenhangskomponenten startet, gibt es genau $n - 2$ Runden. Man sieht sofort, dass der Algorithmus die Menge R überhaupt nicht benutzt, sondern nur die Zusammenhangskomponenten. Der Algorithmus führt R daher gar nicht mit. Abb. 1.1.4 stellt einen möglichen Zwischenstand im Ablauf des Algorithmus dar.

Wenn S und $V - S$ die Knotenmengen der beiden verbleibenden Zusammenhangskomponenten sind, ist die Ausgabe

$$C = \{(v, w) \in E \mid v \in S, w \in V - S\}.$$

Die Idee ist als Algorithmus 1.1.2 dargestellt.

Mit einer deutlichen Anlehnung an den *Algorithmus von Kruskal* können wir diese Skizze wie folgt implementieren. Wir benutzen eine *Union-Find-Datenstruktur*.[3] Die Menge R der gewählten Kanten muss man, anders als im Algorithmus von Kruskal, nicht mitführen. (Die [...]-Teile im Algorithmus werden weggelassen.) In der Mengenvariablen F speichert man alle Kanten, die noch nicht betrachtet worden

[3] Für den Algorithmus von Kruskal und die Union-Find-Datenstruktur siehe Vorlesung „Algorithmen und Datenstrukturen“ im SS 2020, `https://moodle2.tu-ilmenau.de/course/view.php?id=2465`, Abschnitte 11.3. und 11.4. Die Details der Implementierung sind für das Verständnis der Wahrscheinlichkeitsanalyse des Algorithmus aber gar nicht wichtig.

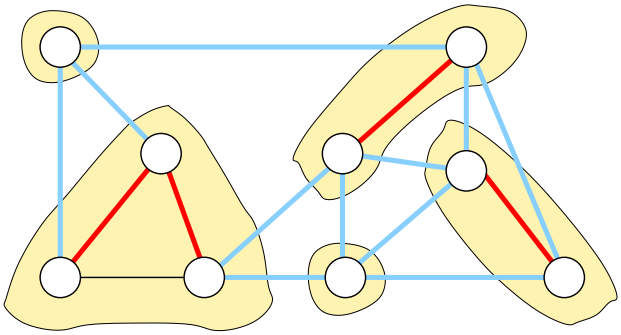


Abbildung 1.1.4: Mögliche Situation nach 4 Runden. Weiß: 9 Knoten. Rot: 4 schon gewählte Kanten. Ocker: $5 = 9 - 4$ von den roten Kanten gebildete Zusammenhangskomponenten. Blau: 11 noch verfügbare Kanten. Jede davon hat in der nächsten Runde dieselbe Wahrscheinlichkeit $\frac{1}{11}$, gewählt zu werden.

sind, und lässt sie bei der Zufallsauswahl mit zu. Wenn eine Kante gewählt wird, die innerhalb einer Komponente verläuft (mit der Union-Find-Datenstruktur leicht festzustellen, Zeilen **6** und **7**), wird sie ignoriert. Am Ende kann man die Union-Find-Datenstruktur noch benutzen, um die Kanten zu ermitteln, die zwischen den zwei verbleibenden Komponenten verlaufen (Zeilen **9**–**12**).

Algorithmus 1.1.2 ***BasicMinCut-Skizze***
Input: Zusammenhängender Graph $G = (V, E)$ mit $n = |V|$
Methode:
1 Jeder Knoten in V bildet für sich eine Zusammenhangskomponente;
2 $R \leftarrow \emptyset$;
3 **repeat** $n - 2$ **times**
4 ***wähle*** Kante e ***zufällig*** aus den Kanten, die zwei Zsh.-Komp. von (V, R)
5 verbinden; $R \leftarrow R \cup \{e\}$ // vereinigt zwei Zsh.-Komp. zu einer neuen
// es verbleiben zwei Zusammenhangskomponenten S und $V - S$
6 **return** $C =$ Menge der Kanten aus E zwischen S und $V - S$.

Algorithmus 1.1.3 ***BasicMinCut***
Input: Zusammenhängender Graph $G = (V, E)$ mit $n = |V|$
Methode:
1 $[R \leftarrow \emptyset;]$
2 $F \leftarrow E$;
3 Initialisiere eine Union-Find-Datenstruktur mit V als Objektmenge;
4 **while** in der Union-Find-Datenstruktur gibt es mehr als 2 Klassen **do**
5 wähle Kante (v, w) aus F zufällig; entferne (v, w) aus F;
6 $r \leftarrow \mathit{find}(v)$; $s \leftarrow \mathit{find}(w)$;
7 **if** $r \neq s$ **then** // (v, w) verbindet zwei Zsh.-Komp., neue Runde
8 $[$füge (v, w) zu R hinzu;$]$ $\mathit{union}(r, s)$;
9 $C \leftarrow \emptyset$;
10 **for** Kante $(v, w) \in F$ **do**
11 **if** $\mathit{find}(v) \neq \mathit{find}(w)$ **then** füge (v, w) zu C hinzu;
12 **return** C.

Abbildung 1.1.5 zeigt eine mögliche Situation am Ende des Algorithmus mit der resultierenden Ausgabe C.

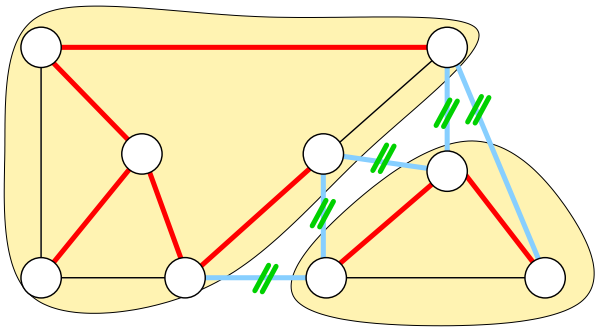


Abbildung 1.1.5: Mögliche Situation am Ende des Algorithmus BasicMinCut. Rot: Die gewählten Kanten, Menge R. Diese bilden zwei Zusammenhangskomponenten. Hellblau: Die ausgegebenen Kanten, d. h. der Schnitt C. Dies sind die Kanten, die die beiden Komponenten verbinden. Alle hellblauen Kanten sind sicher in der Menge F, schwarze Kanten können in F liegen, müssen aber nicht.

Eine kurze Bemerkung zur Rechenzeit: Wenn wir die Union-Find-Datenstruktur benutzen, die für $n-2$ Union-Operationen Zeit $O(n \log n)$ benötigt und $O(1)$ für die Find-Operationen (Array-basiert, mit Listen für die Mengen), erhalten wir Rechenzeit $O(n \log n + m)$ für den Algorithmus. Wenn die Kantenzahl nicht ganz klein ist, also $m \geq n \log n$ gilt, ist dies ein *Linearzeitalgorithmus*. Alternativ können wir Union-Find mit wurzelgerichteten Bäumen implementieren, mit Union-by-Rank und Pfadkompression, und erhalten eine Rechenzeit von $O(m \log^* n)$, auch für sehr dünne Graphen, also Kantenzahlen, die nahe bei n liegen. (Details: Vorlesung „Algorithmen und Datenstrukturen“ im SS 2019.) Die Details der Rechenzeitanalyse sind hier aber nicht relevant.

Nun wenden wir uns der Frage der Korrektheit zu. Gibt es Anlass zu glauben, dass der ausgegebene Schnitt C besonders wenige Kanten hat?

Definition 1.1.4
$\mathrm{mc}(G) := \min\{|C| \mid C \text{ Schnitt in } G\}$, die Größe eines minimalen Schnitts.

Sei C_0 mit $|C_0| = k = \mathrm{mc}(G)$ ein fest gewählter minimaler Schnitt. (Für ein Beispiel siehe Abb. 1.1.3. Beachte, dass C_0 nicht eindeutig bestimmt sein muss.) Wir werden die Wahrscheinlichkeit dafür, dass Algorithmus **BasicMinCut** genau C_0 ausgibt, nach unten abschätzen.

Wir beobachten: $(V, E - C_0)$ hat genau zwei Zusammenhangskomponenten, und C_0 ist genau die Menge der Kanten zwischen diesen beiden. (Der Beweis hierfür ist eine

Übungsaufgabe. Veranschaulichung: Abb. 1.1.3.) Wir nennen S_0 die Knotenmenge der einen und $\overline{S}_0 = V - S_0$ die Knotenmenge der anderen Komponente.

Lemma 1.1.5
Die Ausgabe ist $C_0 \Leftrightarrow$ Der Algorithmus wählt nie eine Kante aus C_0.

(Der *Beweis* ist eine Übungsaufgabe.) Wir müssen nun also die Wahrscheinlichkeit dafür abschätzen, dass in keiner der Runden $i = 1, \ldots, n-2$ eine Kante aus C_0 gewählt wird. Dazu definieren wir $n-2$ *Ereignisse*[4], die ausdrücken, dass in Runde i keine Kante aus C_0 gewählt wird[5], für $i = 1, \ldots, n-2$.

$$\mathcal{E}_i := \{\text{die Kante, die in Runde } i \text{ gewählt wird, ist nicht in } C_0\}. \tag{1.1.1}$$

Lemma 1.1.6
$\mathbf{Pr}(\text{Algorithmus } \mathbf{BasicMinCut} \text{ liefert } C_0 \text{ als Ausgabe}) > 2/n^2$.

Beweis: Nach Lemma 1.1.5 gibt der Algorithmus genau dann C_0 aus, wenn in keiner der $n-2$ Runden eine Kante aus C_0 gewählt wird, und das heißt, dass alle $\mathcal{E}_i$, $1 \leq i \leq n-2$, eintreten, formal: $\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2}$ tritt ein. Wir schätzen die Wahrscheinlichkeit $\mathbf{Pr}(\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2})$ nach unten ab.

Nach der Grundformel $\mathbf{Pr}(A \cap B) = \mathbf{Pr}(A \mid B)\mathbf{Pr}(B)$ der Wahrscheinlichkeitsrechnung gilt:

$$\mathbf{Pr}(\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2}) = \mathbf{Pr}(\mathcal{E}_1)\mathbf{Pr}(\mathcal{E}_2 \mid \mathcal{E}_1) \cdots \mathbf{Pr}(\mathcal{E}_{n-2} \mid \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-3}). \tag{1.1.2}$$

(Bedingte Wahrscheinlichkeiten werden in Kapitel 2 wiederholend diskutiert.) Wir müssen also

$$\mathbf{Pr}(\mathcal{E}_i \mid \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{i-1})$$

abschätzen. Das heißt: Wir nehmen an, dass in Runden $1, \ldots, i-1$ keine Kante aus C_0 gewählt wurde, und fragen nach der Wahrscheinlichkeit, dass dies auch in Runde i so ist. Sei $E_{i-1} \subseteq E$ die Menge der Kanten nach Runde $i-1$, die zwei Zusammenhangskomponenten von (V, R) verbinden. Jede der Kanten in E_i hat die gleiche Wahrscheinlichkeit, in Runde i gewählt zu werden. (Für ein Beispiel siehe Abb.1.1.4.)

[4]Siehe Kapitel 2 für den Begriff des Ereignisses und die verwendete Notation.
[5]„$\mathcal{E}$“ ist ein kalligraphisches „E“ (für „Ereignis“), nicht ein zu groß geratenes „epsilon“.

Wir beobachten: (1) Aus jeder Zusammenhangskomponente von (V, R) führen mindestens k Kanten hinaus. (Grund: Diese Kanten bilden einen Schnitt.) (2) Es gibt $n - i + 1$ Zusammenhangskomponenten in (V, R). (Grund: Am Anfang waren es n viele; in jeder der bisherigen $i - 1$ Runden hat sich die Anzahl um 1 verringert.) Daraus folgt:

$$|E_{i-1}| \geq k(n - i + 1)/2.$$

(Man muss durch 2 dividieren, da jede Kante zu zwei Zusammenhangskomponenten gehört.) Die Wahrscheinlichkeit, in Runde i eine Kante aus C_0 zu wählen (d. h. dass $\overline{\mathcal{E}_i}$ eintritt), ist also höchstens

$$\frac{|C_0|}{k(n-i+1)/2} = \frac{2k}{k(n-i+1)} = \frac{2}{n-i+1}.$$

(Man beachte den „magischen Schritt“ im Beweis: Der Wert k kürzt sich weg.) Übergang zur Wahrscheinlichkeit für das Komplementereignis liefert:

$$\mathbf{Pr}(\mathcal{E}_i \mid \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{i-1}) \geq 1 - \frac{2}{n-i+1} = \frac{n-i-1}{n-i+1}.$$

Nach (1.1.2) folgt

$$\mathbf{Pr}(\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2}) \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \cdot \cdots \cdot \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3}. \tag{1.1.3}$$

Durch Kürzen verschwinden im Zähler und im Nenner die Faktoren $n-2$, $n-3$, …, 4, 3, und wir erhalten

$$\mathbf{Pr}(\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2}) \geq \frac{2}{n(n-1)} > \frac{2}{n^2}.$$

Das ist die Behauptung. □

Die Wahrscheinlichkeit, dass die Ausgabe C von Algorithmus 1.1.3 *nicht* gleich C_0 ist, ist also höchstens $1 - 2/n^2$, eine Zahl, die leider sehr nahe an 1 liegt.

Um die Irrtumswahrscheinlichkeit zu verringern, *wiederholen* wir Algorithmus **Ba-**

sicMinCut ℓ-mal und geben den kleinsten dabei produzierten Schnitt aus.[6]

Algorithmus 1.1.7 ***MinCut***
Input: Zusammenhängender Graph $G = (V, E)$ mit $V = \{1, \ldots, n\}$, $\ell \geq 1$.
Methode:
```
1  C ← {(1, j) | (1, j) ∈ E}; // diese Kanten bilden einen Schnitt
2  h ← |C|;
3  repeat ℓ times
4      CC ← BasicMinCut(G);
5      if |CC| < h then
6          C ← CC; h ← |CC|;
7  return C.
```

Die Rechenzeit dieses Algorithmus ist ℓ mal die Rechenzeit von **BasicMinCut**. (Wir diskutieren weiter unten, wie ℓ zu wählen ist.) Es seien $C_1, \ldots, C_\ell$ die Ausgaben der ℓ Aufrufe, und es sei C^* die Ausgabe von **MinCut**. (Alle diese Kantenmengen sind Zufallsgrößen.) Wir analysieren: Wenn C^* kein minimaler Schnitt ist, dann kann C_0 nicht in $\{C_1, \ldots, C_\ell\}$ vorkommen. Also gilt:

$$\begin{aligned} &\mathbf{Pr}(C^* \text{ ist kein minimaler Schnitt}) \\ &\quad\leq\ \mathbf{Pr}(C_0 \notin \{C_1, \ldots, C_\ell\}) \\ &\quad=\ \mathbf{Pr}(C_1 \neq C_0) \cdots \mathbf{Pr}(C_\ell \neq C_0) \\ &\quad\leq\ \left(1 - \frac{2}{n^2}\right)^\ell . \end{aligned} \tag{1.1.4}$$

(Beim Übergang von der zweiten zur dritten Zeile wird die *Unabhängigkeit* der Zufallsexperimente in den verschiedenen Aufrufen von **BasicMinCut** benutzt, d. h., dass bei jedem Aufruf neue Zufallsexperimente durchgeführt werden. Das wahrscheinlichkeitstheoretische Konzept „Unabhängigkeit“ wird in Kap. 2 diskutiert.) Wenn wir z. B. $\ell = \lceil n^2/2 \rceil$ setzen, ergibt sich

$$\mathbf{Pr}(|C^*| > k) \leq \left(1 - \frac{2}{n^2}\right)^{n^2/2} < e^{-1} \approx 0.3679, \tag{1.1.5}$$

[6] Dem Trick, Algorithmen unabhängig mehrfach zu wiederholen, um die Wahrscheinlichkeit für eine falsche Ausgabe zu verringern, werden wir wieder und wieder begegnen.

also ist

$$\mathbf{Pr}(C^* \text{ ist minimaler Schnitt}) > 1 - e^{-1} > 0.632.$$

Wir haben hier die folgende wichtige Ungleichung benutzt:

$$1 + z \leq e^z, \text{ für alle } z \in \mathbb{R}, \text{ mit Gleichheit genau für } z = 0.$$

Daraus folgt mit den Potenz-Rechenregeln

$$(1 + z)^y \leq e^{zy}, \text{ für alle } z \geq -1,\ y > 0.$$

Dies wurde für $z = -2/n^2$ und $y = n^2/2$ angewandt.
Diese und weitere Ungleichungen werden am Ende von Kapitel 2 diskutiert.

Wenn wir $\ell = \lceil c \cdot n^2 \cdot \ln(n)/2 \rceil$ setzen, für eine Konstante c, dann ergibt sich

$$\mathbf{Pr}(|C^*| > k) \leq \left(1 - \frac{2}{n^2}\right)^{c \cdot n^2 \cdot \ln n/2} < e^{-c \ln n} = n^{-c}. \tag{1.1.6}$$

Mit einer relativ kleinen Wiederholungszahl lässt sich die Irrtumswahrscheinlichkeit also sogar auf einen „polynomiell kleinen“ Wert drücken!

Man beachte, dass die Gesamtrechenzeit des Algorithmus **MinCut** auf G mit n Knoten und m Kanten und mit Wiederholungszahl $\ell = \lceil c \cdot n^2 \cdot (\ln n)/2 \rceil$ durch $O((n \log n + m) \cdot c \cdot n^2 \cdot \ln n) = O(n^3 (\log n)^2 + m n^2 \log n)$ beschränkt ist. Auf jeden Fall ist die Rechenzeit polynomiell.

Satz 1.1.8
Algorithmus 1.1.7 **MinCut** mit $\ell = \frac{1}{2}n^2$ $[\ell = \frac{1}{2}n^2 \ln n;\ \ell = \frac{c}{2}n^2 \ln n]$ liefert mit Wahrscheinlichkeit größer als $1 - 1/e > 0.632$ $[1 - \frac{1}{n};\ 1 - \frac{1}{n^c}]$ einen Schnitt minimaler Kardinalität. Der Algorithmus hat in jedem Fall polynomielle Rechenzeit.

Mitteilung: Man kann den Algorithmus **MinCut** so modifizieren und implementieren, dass die Rechenzeit geringer wird als für sämtliche bekannten deterministischen Algorithmen. Dies wurde von Karger und Stein 1993 geleistet. Ihr Algorithmus hat eine Rechenzeit von $O(n^2 (\log n)^4)$. Karger zeigte 1996, dass man mit $O(m(\log n)^3)$ Zeit auskommt. Dies wurde erst im Jahr 2020 auf $O(m(\log n)^2)$ verbessert (Pawel Gawrychowski, Shay Mozes, Oren Weimann: Minimum Cut in $O((m \log^2 n)$ Time. ICALP 2020: 57:1-57:15).

Bemerkung: Das MinCut-Problem besitzt effiziente Algorithmen, die ohne Randomisierung auskommen. Manche dieser Algorithmen beruhen auf „Flussberechnungen“ (siehe Veranstaltung „Effiziente Algorithmen“ im Masterstudiengang oder [Cormen, Leiserson, Rivest und Stein, Introduction to Algorithms, Kapitel „Maximum Flow“]) und haben Rechenzeiten der Größenordnung $O(nm \log(n^2/m)) = O(nm \log n)$. Diese Algorithmen funktionieren sogar, wenn die Kanten mit Gewichten ≥ 0 versehen sind. Andere deterministische Algorithmen kommen ohne Flussberechnungen aus und haben Rechenzeiten von $O(nm)$ [Stoer/Wagner 1997]. In [Brinkmeier 2007] wird ein Algorithmus vorgestellt, der Rechenzeit $O(n^2\delta_G)$ hat, wobei δ_G der kleinste Knotengrad in G ist. Beachte: $m \geq n\delta_G/2$, also ist dies immer mindestens so gut wie $O(nm)$.
Wir werden im weiteren Verlauf oft der Situation begegnen, dass unsere randomisierten Algorithmen Berechnungsprobleme lösen, für die es *auch* recht effiziente deterministische Algorithmen gibt. Der Vorteil der randomisierten Verfahren ist oft, dass die Algorithmen einfacher sind als deterministische, oder dass sie etwas schneller laufen, auch in der praktischen Anwendung.

Bemerkung: Das MinCut-Problem hat also deterministische Polynomialzeitalgorithmen und einen etwas schnelleren randomisierten Algorithmus. In der Vorlesung „Automaten, Sprachen und Komplexität“ lernt man NP-vollständige Entscheidungsprobleme kennen, zusammen mit der Vermutung, dass es für diese *keinen* deterministischen Polynomialzeitalgorithmus gibt. Zu diesen Entscheidungsproblemen gehören „NP-schwere“ Optimierungsprobleme, zum Beispiel das Problem, in einem vollständigen Graphen mit Kantengewichten eine TSP-Tour mit kleinsten Kosten zu finden oder die Knoten eines Graphen mit möglichst wenigen Farben „legal“ zu färben (d. h. so dass die Endpunkte jeder Kante verschiedene Farben haben), oder auch der, zu einem zusammenhängenden Graphen einen Schnitt *maximaler* Kardinalität zu finden. Kann Randomisierung hier helfen? Leider kennt man kein einziges NP-schweres Optimierungsproblem, das einen randomisierten Polynomialzeitalgorithmus zulässt. Aufgrund von Resultaten in der Komplexitätstheorie gibt es sogar Anlass zu der Vermutung, dass es solche Probleme nicht gibt. Der Ansatz „Randomisierung“ kann also effizientere (d. h. etwas schnellere) Algorithmen liefern; vermutlich wird er aber nicht dazu führen, dass man NP-schwere Probleme effizient lösen kann. In der Vorlesung „Approximationsalgorithmen“ kann man lernen (Master-Studium!), wie Randomisierung in Verbindung mit anderen Techniken dabei hilft, effizient *näherungsweise* optimale Lösungen für NP-schwere Probleme zu berechnen.

1.2 Gleichheit von Polynomprodukten

Beispiel: Gilt

$$(X^4+4X^3+6X^2+4X+1)(X^4-4X^3+6X^2-4X+1) = (X^2-1)(X^2-1)(X^2-1)(X^2-1)\ ? \tag{1.2.7}$$

Dabei ist natürlich die Gleichheit zwischen Polynomen gemeint. Allgemeiner könnte man $(\sum_{1\le i\le n}\binom{n}{i}X^i)(\sum_{1\le i\le n}(-1)^i\binom{n}{i}X^i)$ mit dem Produkt von n Kopien von (X^2-1) vergleichen. (Erst nachher lesen: Lösung unter[7].)

Allgemein betrachten wir die folgende Aufgabenstellung: Gegeben seien Polynome $f_1(X),\ldots,f_s(X)$ und $g_1(X),\ldots,g_t(X)$ über einem Körper $\mathbb{F}$ (etwa $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$ oder $\mathbb{Z}_p=\mathbb{Z}/p\mathbb{Z}$ für eine Primzahl p).

Frage: Gilt

$$f_1(X)\cdot\ldots\cdot f_s(X) = g_1(X)\cdot\ldots\cdot g_t(X)\ ? \tag{1.2.8}$$

Die naheliegende Methode ist, die Polynome auf beiden Seiten *auszumultiplizieren*, zu vereinfachen und dann zu vergleichen. Dabei beobachtet man Folgendes: Die Multiplikation zweier Polynome vom Grad d benötigt bei naivem Vorgehen $\Theta(d^2)$ Körperoperationen. Mit der FFT-Methode (siehe Vorlesung „Algorithmen und Datenstrukturen“) erreicht man für manche Grundbereiche (z. B. $\mathbb{R}$ oder $\mathbb{C}$) Kosten von $O(d\log d)$. Ähnliche Kosten ergeben sich bei der Multiplikation von d Kopien eines Polynoms vom Grad 2. Es ist kein deterministischer Algorithmus bekannt, der für solche Multiplikationsaufgaben mit linearem Aufwand auskommt.

Wir versuchen es mit folgendem Trick: Wir wählen ein zufälliges Argument, setzen es in die einzelnen Polynome ein und werten diese aus, multiplizieren dann die Werte und vergleichen die Produkte. (Damit werden zu keiner Zeit die Koeffizienten eines Produktpolynoms berechnet.)

Dies heißt genauer: Sei z. B. $\mathbb{F}$ gleich $\mathbb{Q}$ (oder $\mathbb{R}$ oder $\mathbb{C}$). Bestimme eine Zahl k aufgrund der Eingabe und der angestrebten Irrtumswahrscheinlichkeit. Wähle eine Menge $S\subseteq\mathbb{F}$ der Größe k, zum Beispiel die Menge $\{1,\ldots,k\}$. Dann wähle a aus S rein zufällig. Berechne $b_1=f_1(a),\ldots,b_s=f_s(a),c_1=g_1(a),\ldots,c_t=g_t(a)$ durch Einsetzen und Auswerten, und berechne dann $u=b_1\cdot\ldots\cdot b_s$ und $v=c_1\cdot\ldots\cdot c_t$. Falls $u\neq v$, weiß man sicher, dass die Produktpolynome verschieden sind. Falls $u=v$, muss man mit der Interpretation des Ergebnisses etwas vorsichtiger sein.

[7] Mit der binomischen Formel $(Y+Z)^n=\sum_i\binom{n}{i}Y^iZ^{n-i}$ sieht man, durch Einsetzen von 1 für Y und X bzw. $(-X)$ für Z, dass dabei $(1+X)^n(1-X)^n$ mit $(1-X^2)^n=((1+X)(1-X))^n$ verglichen wird. Es herrscht also Gleichheit.

Wesentlich ist die Betrachtung der Grade. Sei

$$d := \max\Big\{ \sum_{1\le i\le s} \deg(f_i(X)), \sum_{1\le j\le t} \deg(g_j(X))\Big\}.$$

Dann hat das Polynom

$$h(X) := f_1(X)\cdot \ldots \cdot f_s(X) - g_1(X)\cdot \ldots \cdot g_t(X) \tag{1.2.9}$$

auf keinen Fall einen Grad größer als d. Offensichtlich gilt für alle $a \in \mathbb{F}$:

$$h(a) = 0 \quad \Leftrightarrow \quad f_1(a)\cdot \ldots \cdot f_s(a) = g_1(a)\cdot \ldots \cdot g_t(a). \tag{1.2.10}$$

Bei der angedeuteten Berechnung erhalten wir also $u = v$ genau dann, wenn $h(a) = 0$ ist. Wenn die Polynomprodukte gleich sind, ist $h(X)$ das Nullpolynom, und es gilt $h(a) = 0$ für alle $a \in \mathbb{F}$. Andernfalls ist $h(X)$ nicht das Nullpolynom.

Fakt 1.2.1
Ein Polynom vom Grad höchstens d über einem beliebigen Körper (etwa $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, $\mathbb{Z}_p$ für eine Primzahl p), das nicht das Nullpolynom ist, hat in diesem Körper nicht mehr als d Nullstellen.[8]

Daher hat $h(X)$ nicht mehr als d Nullstellen in $\mathbb{F}$. Wenn wir $k \ge d$ wählen, erhalten wir für zufällig aus S gewähltes a:

$$\mathbf{Pr}(h(a) = 0) = \frac{|\{a \in S \mid h(a) = 0\}|}{k} \le \frac{d}{k}.$$

Durch geeignete Wahl von $k = |S|$ können wir diese Wahrscheinlichkeit klein machen.

[8] Die Beweisidee ist folgende: Wenn das Polynom $f(X)$ die verschiedenen Nullstellen $a_1, \ldots, a_r$ hat, dann kann man $f(X) = (X - a_1)\ldots(X - a_r)\cdot h(X)$ schreiben, für ein Polynom $h(X)$, man kann also für jede Nullstelle einen linearen Faktor abspalten. Daraus sieht man sofort, dass $f(X)$ Grad mindestens r haben muss.

Algorithmus 1.2.2 ***PPV – Polynomproduktvergleich***
Input: Listen $f_1(X), \ldots, f_s(X)$ und $g_1(X), \ldots, g_t(X)$ von Polynomen über $\mathbb{F}$
Methode:
1 $\mathtt{d} \leftarrow \max\{\sum_{1\le i\le s} \deg(f_i(X)), \sum_{1\le j\le t} \deg(g_j(X))\}$;
2 $\mathtt{k} \leftarrow 2 \cdot \mathtt{d}$;
3 Wähle eine Teilmenge S von $\mathbb{F}$ mit $|S| = \mathtt{k}$;
4 Wähle $\mathtt{a}$ aus S *zufällig* // (uniforme Verteilung)
5 $\mathtt{u} \leftarrow f_1(\mathtt{a}) \cdot \ldots \cdot f_s(\mathtt{a})$; // gerechnet in $\mathbb{F}$
6 $\mathtt{v} \leftarrow g_1(\mathtt{a}) \cdot \ldots \cdot g_t(\mathtt{a})$;
7 **if** $\mathtt{u} \neq \mathtt{v}$ **then return** 1
8 **else return** 0.

Die zentralen Eigenschaften von Algorithmus 1.2.2 sind folgende:

- Die Anzahl der Multiplikationen von Körperelementen ist höchstens $4d + s + t$, die Rechenzeit ist $O(d)$, wenn man die einzelnen Polynome mit Hilfe des Horner-Schemas auswertet.
- Wenn $f_1(X) \cdot \ldots \cdot f_s(X) = g_1(X) \cdot \ldots \cdot g_t(X)$, ist die Ausgabe sicher 0.
- Wenn $f_1(X) \cdot \ldots \cdot f_s(X) \neq g_1(X) \cdot \ldots \cdot g_t(X)$, gilt: $\mathbf{Pr}(\text{Ausgabe ist } 0) \le \frac{1}{2}$ (oder, bei anderer Wahl von k im Algorithmus: $\ldots \le d/k$).

Wenn man höhere Zuverlässigkeit möchte, könnte man viel größere k wählen. Für Körper, bei denen eine Rechenoperation tatsächlich mit Zeit $O(1)$ zu veranschlagen ist, ist dies ohne große Rechenzeitveränderung möglich. Schwierigkeiten entstehen, wenn der Körper nur endlich viele Elemente hat oder wenn, wie im Fall $K = \mathbb{Q}$, Rechnen mit größeren Zahlen größeren Aufwand hat. Alternativ führt man Algorithmus 1.2.2 mehrfach (ℓ-mal) durch, mit Ergebnissen $b_1, \ldots, b_\ell$, und liefert $\max\{b_1, \ldots, b_\ell\}$ als Gesamtergebnis. Für diesen Algorithmus gilt:

- Die Anzahl der Multiplikationen von reellen Zahlen ist höchstens $(4d + s + t)\ell$.
- Wenn $f_1(X) \cdot \ldots \cdot f_s(X) = g_1(X) \cdot \ldots \cdot g_t(X)$, ist die Ausgabe sicher 0.
- Wenn $f_1(X) \cdot \ldots \cdot f_s(X) \neq g_1(X) \cdot \ldots \cdot g_t(X)$, gilt: $\mathbf{Pr}(\text{Ausgabe ist } 0) \le \frac{1}{2^\ell}$ (oder, bei anderer Wahl von k im Algorithmus: $\ldots \le (d/k)^\ell$).

Um zu sehen, dass Rechenzeitunterschiede drastisch sein können, betrachten wir abschließend noch folgendes Beispiel: $\mathbb{F} = \mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ für eine Primzahl p, mit Addition und Multiplikation modulo p. Es sei n ein Maß für die Eingabegröße. Die Frage lautet: Ist $(1+X)(1+X^2)(1+X^4)\ldots(1+X^{2^{n-1}})(1-X)$ gleich $1-X^{2^n}$?

Der naive Algorithmus berechnet das Produkt der linken Seite durch Multiplikation von links nach rechts. Da

$$\prod_{0 \le i < n} (1 + X^{2^i}) = \sum_{0 \le j < 2^n} X^j$$

gilt (Hinweis: Binärdarstellung der Exponenten j), hat das vorletzte Zwischenergebnis 2^n viele Terme, bevor sich bei der letzten Multiplikation fast alles wegkürzt. Mit geschickter Auswertung kann man das Produkt für ein Körperelement a aber viel schneller ausrechnen: Mit $b_0 = a$, $b_1 = b_0^2 = a^2$, $b_2 = b_1^2 = a^4$, $b_3 = b_2^2 = a^8$, ... (immer modulo p) erhält man die benötigten Potenzen durch nur n Multiplikationen modulo p.

1.3 Verifikation von Matrixprodukten

Gegeben seien drei $n \times n$-Matrizen A, B und C über einem Körper $\mathbb{F}$ (zum Beispiel $\mathbb{F} = \mathbb{Q}$ oder $\mathbb{F} = \mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ mit Operationen modulo p, wobei p eine Primzahl ist). Jemand behauptet, dass $A \cdot B = C$ ist. Wir wollen dies überprüfen.

Eine naheliegende Methode ist, das Produkt $A \cdot B$ zu berechnen und das Resultat mit C zu vergleichen. Mit der Schulmethode für die Multiplikation dauert dies Zeit $O(n^3)$, mit der Strassen-Methode (Vorlesung „Algorithmen und Datenstrukturen“) Zeit $O(n^{\log 7})$, wobei $\log 7 \approx 2.81$.[9] Wir zeigen hier, dass ein sehr einfacher randomisierter Algorithmus mit Zeit $O(n^2)$ auskommt.

[9] Schnellere Methoden: Coppersmith und Winograd (1987/90), Zeit $O(n^{2.376})$; Stothers (2010), Williams (2011), Le Gall (2014): Zeit $O(n^{2.3728639})$. (Diese Algorithmen gelten allerdings als rein theoretisch wichtig; für realistisch große n sind andere Verfahren schneller.)

Algorithmus 1.3.1 ***VerifyMatrixProduct***
Input: $n \times n$-Matrizen A, B, C über Körper $\mathbb{F}$.
Methode:
1 Wähle endliche Menge $S \subseteq \mathbb{F}$ mit $0, 1 \in S$;
2 Wähle Vektor $\mathtt{v} = (v_1, \ldots, v_n) \in S^n$ *zufällig* // (uniforme Verteilung)
3 $\mathtt{u} \leftarrow B \cdot \mathtt{v}$;
4 $\mathtt{w} \leftarrow A \cdot \mathtt{u}$;
5 $\mathtt{x} \leftarrow C \cdot \mathtt{v}$;
6 **if** $w \neq x$ **then return** 1 **else return** 0.

Die Rechenzeit dieses Algorithmus ist $O(n^2)$, weil drei Matrix-Vektor-Multiplikationen auszuführen sind. Da die Größe der Eingabe $\Theta(n^2)$ ist, ist dies lineare Rechenzeit.

Wie steht es mit dem Ein-Ausgabe-Verhalten? Wir bezeichnen die Vektoren in `v`, `w`, `x` mit v, w, x. Offensichtlich gilt Folgendes: Wenn $A \cdot B = C$ ist, dann gilt

$$x = C \cdot v = (A \cdot B) \cdot v = A \cdot (B \cdot v) = A \cdot u = w,$$

also ist die Ausgabe 0. (In der Rechnung haben wir die Assoziativität im Produkt $A \cdot B \cdot v$ benutzt. Der Trick ist, dass wir auf diese Weise Information über $A \cdot B$ erhalten können, ohne dieses Matrixprodukt auszurechnen!)

Ab hier gelte $A \cdot B \neq C$. Wir definieren

$$D := A \cdot B - C,$$

mit $D = (d_{ij})_{1 \leq i,j \leq n}$. Weil $A \cdot B \neq C$, gibt es einen Eintrag $d_{i_0 j_0} \neq 0$. Wir überlegen: Wenn $v_1, \ldots, v_{j_0-1}, v_{j_0+1}, \ldots, v_n$ irgendwelche Elemente von S sind, dann gibt es in S entweder ein oder kein Element v_{j_0} mit

$$d_{i_0 1} v_1 + \cdots + d_{i_0, j_0-1} v_{j_0-1} + d_{i_0 j_0} v_{j_0} + d_{i_0, j_0+1} v_{j_0+1} + \cdots + d_{i_0 n} v_n = 0. \qquad (1.3.11)$$

Dies liegt daran, dass man wegen $d_{i_0 j_0} \neq 0$ die Gleichung (1.3.11) nach v_{j_0} auflösen kann. Eine Lösung gibt es, wenn das Ergebnis in S liegt, keine Lösung sonst. Damit erhalten wir: Wenn v aus S^n zufällig gewählt wird, ist die Wahrscheinlichkeit dafür, dass (1.3.11) gilt, höchstens $1/|S|$.

Wir können also abschätzen:

$$\begin{aligned}
&\mathbf{Pr}(\text{Ausgabe ist } 0)\\
&= \mathbf{Pr}(A \cdot B \cdot v = C \cdot v) \qquad (\text{für den Zufallsvektor } v)\\
&= \mathbf{Pr}(D \cdot v = 0)\\
&\leq \mathbf{Pr}(\text{Gleichung (1.3.11) gilt})\\
&\leq \frac{1}{|S|} \leq \frac{1}{2}. \qquad (1.3.12)
\end{aligned}$$

Wir fassen zusammen: Algorithmus 1.3.1 hat folgendes Verhalten: Wenn $A \cdot B = C$ ist, ist die Ausgabe 0; wenn $A \cdot B \neq C$ ist, entsteht die (unerwünschte) Ausgabe 0 höchstens mit Wahrscheinlichkeit $1/|S|$.

Spezialfall: Wenn $\mathbb{F} = \mathbb{Z}_p$ für eine Primzahl p, wird man $S = \mathbb{Z}_p$ wählen; die Irrtumswahrscheinlichkeit ist dann (genau) $1/p$.

Wenn die Irrtumswahrscheinlichkeit $1/|S|$ zu groß ist, können wir ebenso wie in Abschnitt 1.1 durch ℓ-fache Wiederholung die Irrtumswahrscheinlichkeit auf $(1/|S|)^\ell$ drücken. Beispielsweise genügt $(c \log n)$-fache Wiederholung, um eine Irrtumswahrscheinlichkeit von höchstens $|S|^{-c \log n} = n^{-c \cdot \log |S|}$ zu erhalten. Dann hat der Algorithmus eine Rechenzeit von $O(n^2 \log n)$, immer noch viel weniger als die deterministischen Verfahren.

1.4 Randomisiertes Quicksort

Der Algorithmus „Quicksort“ wurde schon in mehreren Vorlesungen betrachtet: „Algorithmen und Programmierung“ und „Algorithmen und Datenstrukturen“. Hier konzentrieren wir uns auf die „randomisierte“ Variante und auf eine clevere Analysemethode, die einige interessante Konzepte aus der Wahrscheinlichkeitsrechnung benutzt.

Wir gehen von folgender idealisierter Situation aus: In einem Array `A[1..`n`]` stehen verschiedene Zahlen[10] $a_1, \ldots, a_n$, in dieser Reihenfolge. Die Aufgabe ist, diese Zahlen umzusortieren, in eine Reihenfolge $b_1 < \cdots < b_n$, wobei b_i in `A[`i`]` steht, für $1 \leq i \leq n$.

[10] In tatsächlichen Anwendungen hat man Datensätze mit Schlüsseln aus einem total geordneten Universum, nicht unbedingt Zahlen. Dabei kann es natürlich auch identische Schlüssel geben, und man muss entsprechende Vorkehrungen treffen. Eine Möglichkeit ist 3-Wege-Partitionierung, siehe weiter unten. Wir betrachten die Situation mit verschiedenen Zahlen, um bei der Analyse den Überblick zu behalten.

Randomisiertes Quicksort geht so vor: Wenn $n = 1$ ist, passiert nichts. Wenn $n > 1$ ist, wird eine Prozedur **rqsort**$(1, n)$ aufgerufen. Dabei ist **rqsort**(l, r) so angelegt, dass gilt: Die Einträge in `A[`$l..r$`]` werden sortiert, die Einträge in `A[`$1..l-1$`]` und `A[`$r+1..n$`]` bleiben unverändert. Die Prozedur **rqsort**(l, r) geht wie folgt vor: Wenn $l \geq r$ gilt, passiert nichts. Wenn $l < r$ gilt, wird ein Index $t \in \{l, \ldots, r\}$ ***zufällig*** gewählt. Der Eintrag x in `A[`t`]` wird das „Pivotelement“ oder „partitionierende Element“. Durch eine Prozedur „**partition**“ werden die Einträge in `A[`$l..r$`]` wie folgt verschoben: Der Eintrag x landet in einer Position `A[`p`]` mit $l \leq p \leq r$, alle Einträge in `A[`$l..p-1$`]` sind kleiner als x, alle Einträge in `A[`$p+1..r$`]` sind größer als x.[11] Dieser Vorgang erfordert genau $r - l$ Vergleiche und $\Theta(r - l)$ Zeit. Anschließend wird rekursiv **rqsort**$(l, p-1)$ aufgerufen (falls $l < p - 1$ ist) und **rqsort**$(p + 1, r)$ (falls $p + 1 < r$ ist).

Durch einen einfachen Induktionsbeweis über die Größe $k = r - l$ zeigt man, dass **rqsort**(l, r) tatsächlich das Teilarray `A[`$l..r$`]` sortiert. Die Frage ist, wie groß die erwartete Anzahl von Schlüsselvergleichen und wie groß die erwartete Rechenzeit ist. Hierbei wird der Erwartungswert über die Zufallsentscheidungen des Algorithmus (bei der Wahl der Pivotelemente) gebildet, nicht etwa über verschiedene Inputs.

[11] 3-Wege-Partitionierung erzeugt drei Segmente in `A[`$l..r$`]`: In `A[`$l..p_1 - 1$`]` stehen Einträge $< x$, in `A[`$p_1..p_2$`]` stehen Einträge mit Schlüssel x, in `A[`$p_2 + 1..r$`]` stehen Einträge $> x$. Der erste Teil `A[`$l..p_1 - 1$`]` und der dritte Teil `A[`$p_2 + 1..r$`]` werden rekursiv sortiert, wenn sie mehr als einen Eintrag haben. Man kann relativ leicht zeigen, dass die für den Spezialfall ermittelte mittlere Anzahl von Vergleichen eine obere Schranke für die Anzahl von Vergleichen im allgemeinen Fall ist.

Wir geben den Algorithmus nochmals im Zusammenhang an:

Algorithmus 1.4.1 ***RandomizedQuicksort***
Input: Ein Array `A[1..n]` mit Einträgen $a_1, \ldots, a_n$, alle verschieden.
Aufruf: if $n > 1$ **then rqsort**$(1, n)$.

Dabei ist **rqsort** die folgende rekursive, randomisierte Prozedur:

Prozedur rqsort(ℓ, r) // Vorbedingung: $1 \leq \ell < r \leq n$
1 wähle *zufällig* ein $t \in \{\ell, \ell+1, \ldots, r\}$ // (uniforme Verteilung)
2 $x \leftarrow$ `A`$[t]$; // x heißt *Pivotelement* oder *partitionierendes Element*
3 Aufruf der Prozedur **partition**(ℓ, r, t, p), mit Ausgabeparameter p:
Verschiebe die `A[`i`]`, $\ell \leq i \leq r$, mit `A[`i`]` $< x$ nach `A[`$\ell..p-1$`]`,
verschiebe x nach `A[`p`]`,
verschiebe die `A[`i`]`, $\ell \leq i \leq r$, mit `A[`i`]` $> x$ nach `A[`$p+1..r$`]`;
4 **if** $\ell < p-1$ **then rqsort**$(\ell, p-1)$;
5 **if** $p+1 < r$ **then rqsort**$(p+1, r)$;

Zur Analyse beobachten wir:

- Ein Aufruf **rqsort**(l, r) kostet $r-l$ Vergleiche und Zeit $\Theta(r-l)$, wenn man die rekursiven Aufrufe nicht mitzählt.
- Es gibt insgesamt maximal $n-1$ Aufrufe von **rqsort**. (Dies liegt daran, dass ein Eintrag, der im Aufruf **rqsort**(l, r) Pivotelement war, nicht in davon ausgelösten rekursiven Aufrufen vorkommen kann, und die Mindestlänge des Arrays in einem Aufruf 2 ist. Es kann tatsächlich $n-1$ Aufrufe geben.)

Zur Veranschaulichung betrachte man Abb. 1.4.6.

Den gesamten Zeitaufwand erhalten wir durch Summation über alle Aufrufe. Die fixen Kosten aller Aufrufe liefern Zeitaufwand $O(n)$, die variablen Kosten sind

$$\sum_{\substack{1 \leq l < r \leq n \\ \text{Aufruf } \mathbf{rqsort}(l,r) \text{ erfolgt}}} \Theta(r-l) = \Theta\Bigg(\underbrace{\sum_{\substack{1 \leq l < r \leq n \\ \text{Aufruf } \mathbf{rqsort}(l,r) \text{ erfolgt}}} (r-l)}_{=: C} \Bigg)$$

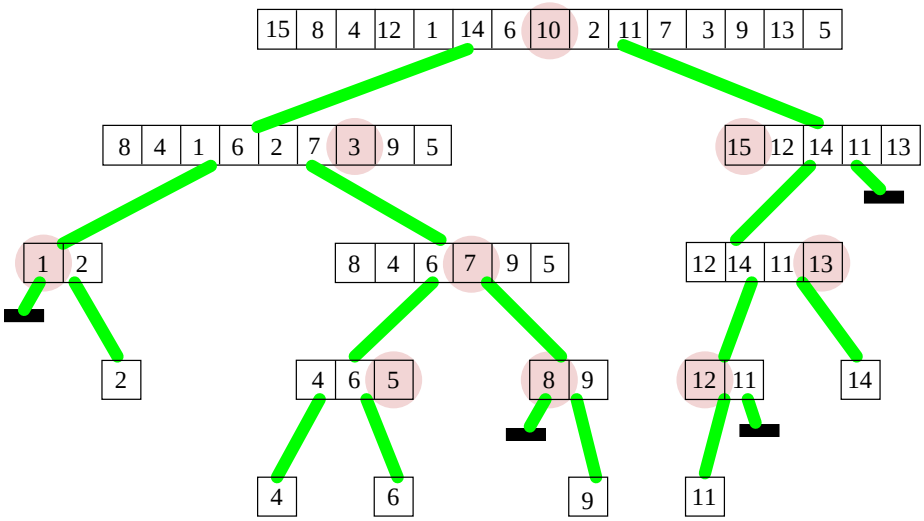


Abbildung 1.4.6: Ablauf von Quicksort als Baum. Die zu sortierenden Schlüssel sind $1, \ldots, 15$ in irgendeiner Reihenfolge. Die aufgrund der zufallsgesteuerten Pivotwahl entstehenden Aufrufe von **rqsort** sind als Baum dargestellt, wobei in einem Knoten das Teilarray `A[`ℓ`..`r`]` steht, für das ein solcher Aufruf erfolgt, in den Kindern entsprechend die Teilarrays, die den beiden rekursiven Aufrufen entsprechen. Man beachte, dass der Baum vom Zufall abhängt. (Die Aufrufreihenfolge entspricht einem Präorderdurchlauf durch den Baum.) Pivots sind rosa dargestellt. Kosten: Der Knoten für `A[`ℓ`..`r`]` kostet Zeit $O(1)$ plus Zeit proportional zu $r - \ell$; zu ihm gehören $r - \ell$ Vergleiche in der Partitionierungsprozedur.

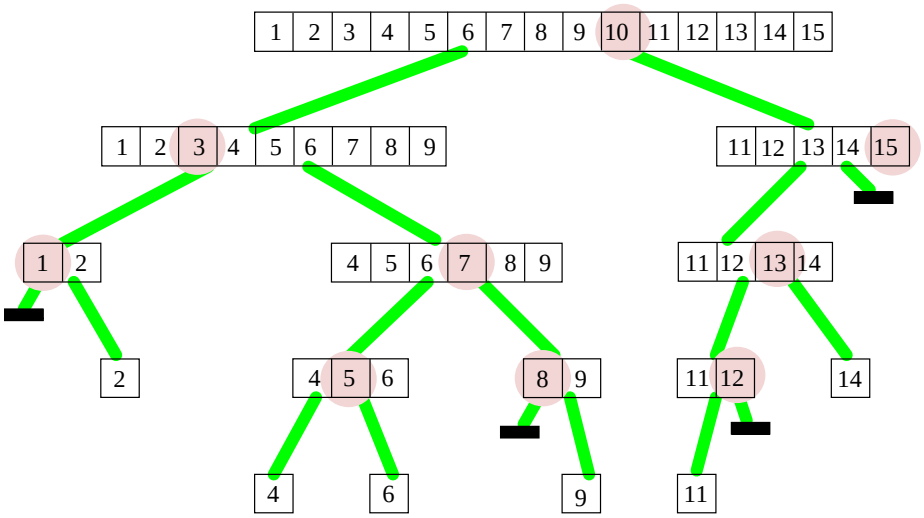


Abbildung 1.4.7: Ablauf von Quicksort als Baum. Wir betrachten die ideale Situation, wo der Input schon aufsteigend sortiert ist (also $(a_1, \ldots, a_{15}) = (b_1, \ldots, b_{15}) = (1, \ldots, 15)$ gilt). Die Wahrscheinlichkeiten für die Pivotwahlen sind exakt dieselben wie in Abb. 1.4.6, weil jeder Eintrag in $\texttt{A}[\ell..r]$ genau dieselbe Wahrscheinlichkeit $1/(r-\ell+1)$ hat, als Pivot gewählt zu werden, ganz gleich wie die Zahlen angeordnet sind. Wenn der Ablauf wie in dieser Abbildung ist, gilt $C = 14+8+4+1+5+3+2+1+1 = 39$ und $X_{4,10} = 1$, $X_{5,11} = 0$, $X_{4,7} = 1$, $X_{3,8} = 1$, $X_{4,9} = 0$, für die im Text definierten Indikatorvariablen X_{ij}. Es sei nochmals betont, dass der Baum von den zufällig gewählten Pivots bestimmt wird. Andere Pivots führen zu einem anderen Baum und zu anderen Werten für C und für die X_{ij}.

Dabei gibt die Zufallsvariable C die Anzahl aller ausgeführten Vergleiche an. Die erwartete Anzahl von Vergleichen ist der Erwartungswert $\mathbf{E}(C)$. Diesen werden wir im Folgenden berechnen. Die erwartete Rechenzeit ist $\Theta(n + \mathbf{E}(C))$.

Erinnerung: $b_1 < \cdots < b_n$ ist die Folge der Eingabezahlen *in sortierter Reihenfolge*. Wir definieren:[12]

$$X_{ij} := [b_i \text{ und } b_j \text{ werden verglichen}] := \begin{cases} 1 & \text{, falls } b_i \text{ und } b_j \text{ verglichen werden,} \\ 0 & \text{, andernfalls.} \end{cases}$$

Die X_{ij} sind sogenannte ***Indikator-Zufallsvariablen***. Offensichtlich gilt

$$C = \sum_{1 \le i < j \le n} X_{ij} \text{ , also } \mathbf{E}(C) = \sum_{1 \le i < j \le n} \mathbf{E}(X_{ij}), \tag{1.4.13}$$

wegen der Linearität des Erwartungswertes (siehe Kapitel 2). Zur Veranschaulichung siehe Abb. 1.4.7.

Da X_{ij} nur die Werte 0 und 1 annimmt, gilt

$$\mathbf{E}(X_{ij}) = \mathbf{Pr}(X_{ij} = 1) = \mathbf{Pr}(b_i \text{ und } b_j \text{ werden verglichen}).$$

Wir müssen also nur die letztgenannten Wahrscheinlichkeiten bestimmen und dann summieren.

Betrachte $1 \le i < j \le n$ und dazu die Menge $I_{ij} = \{b_i, b_{i+1}, \ldots, b_j\}$ der Eingabezahlen zwischen b_i und b_j. Beobachte den Ablauf des Algorithmus, wie er sich durch die rekursiven Aufrufe hindurch entwickelt. Solange kein Element von I_{ij} als Pivotelement gewählt wird, werden bei der Partitionierung stets alle Elemente von I_{ij} im gleichen Teilarray landen. Genau in dem Moment, in dem zum ersten Mal ein Element von I_{ij} Pivotelement wird, fällt die Entscheidung darüber, ob b_i und b_j verglichen werden oder nicht. Wenn entweder b_i oder b_j dieses Pivotelement ist, erfolgt der Vergleich, wenn einer der Einträge $\{b_{i+1}, \ldots, b_{j-1}\}$ das erste ist, landen b_i und b_j bei der Partitionierung in verschiedenen Teilarrays und werden nie verglichen.

Weil Pivotelemente uniform zufällig gewählt werden, hat jedes der $j - i + 1$ Elemente von I_{ij} die gleiche Wahrscheinlichkeit, das erste Pivotelement in dieser Menge zu sein. Damit gilt:[13]

$$\mathbf{Pr}(b_i \text{ und } b_j \text{ werden verglichen}) = \frac{|\{i, j\}|}{|\{i, i+1, \ldots, j\}|} = \frac{2}{j - i + 1}. \tag{1.4.14}$$

[12] Wir benutzen hier und später die sogenannten „Iverson-Klammern“ $[\ldots]$, die Wahrheitswerte in Zahlen übersetzen. Wenn φ eine Aussage ist, dann ist $[\varphi] = 1$, wenn φ zutrifft und $[\varphi] = 0$ sonst.

[13] In Abb. 1.4.8 wird diese Überlegung anhand von Illustrationen anschaulich begründet.

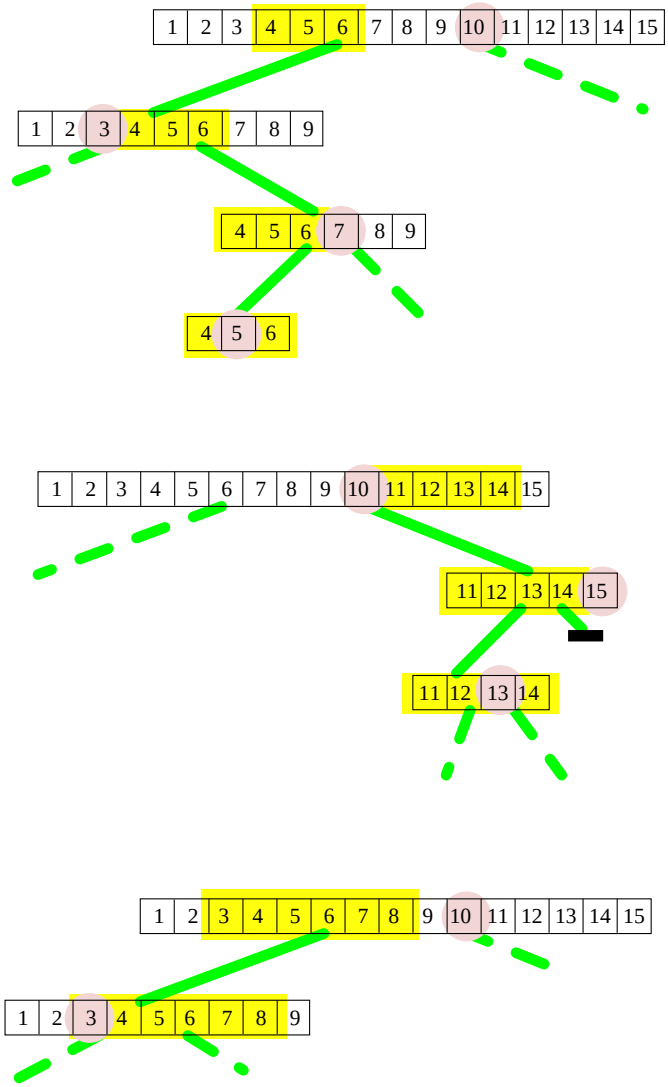


Abbildung 1.4.8: Verläufe für verschiedene Intervalle I_{ij}. Input ist (idealisiert) gleich $\{1, \ldots, 15\}$, also ist $I_{ij} = \{i, \ldots, j\}$.
Oben: $I_{4,6} = \{4, 5, 6\}$. $\mathbf{Pr}(4 \text{ und } 6 \text{ werden verglichen}) = \frac{2}{|\{4,5,6\}|} = \frac{2}{3}$. In diesem Beispiel (Pivot 5) passiert dies nicht, $X_{4,6} = 0$.
Mitte: $I_{11,14} = \{11, 12, 13, 14\}$. $\mathbf{Pr}(11 \text{ und } 14 \text{ werden verglichen}) = \frac{2}{|\{11,12,13,14\}|} = \frac{2}{4}$. In diesem Beispiel (Pivot 13) passiert dies nicht, $X_{11,14} = 0$.
Unten: $I_{3,8} = \{3, 4, 5, 6, 7, 8\}$. $\mathbf{Pr}(3 \text{ und } 8 \text{ werden verglichen}) = \frac{2}{|\{3,4,5,6,7,8\}|} = \frac{2}{6}$. In diesem Beispiel (Pivot 3) passiert dies, $X_{3,8} = 1$.

Wenn wir die „2“ auf Paare $i<j$ und $j<i$ verteilen, erhalten wir

$$\mathbf{E}(C) = \sum_{1\le i\ne j\le n} \frac{1}{j-i+1}, \tag{1.4.15}$$

also ist $\mathbf{E}(C)$ die Summe aller Einträge in der folgenden Matrix:

$$\begin{pmatrix}
0 & \frac12 & \frac13 & \frac14 & \cdots & \frac{1}{n-1} & \frac1n \\
\frac12 & 0 & \frac12 & \frac13 & \ddots & \cdots & \frac{1}{n-1} \\
\frac13 & \frac12 & 0 & \frac12 & \ddots & & \vdots \\
\vdots & \frac13 & \frac12 & 0 & \ddots & & \frac14 \\
\vdots & & \ddots & \ddots & \ddots & & \frac13 \\
\frac{1}{n-1} & & & \ddots & \ddots & \ddots & \frac12 \\
\frac1n & \frac{1}{n-1} & \cdots & \cdots & \frac13 & \frac12 & 0
\end{pmatrix}.$$

Den Wert der Summe kann man leicht direkt hinschreiben, indem man entlang der (Neben-)Diagonalen addiert und die Symmetrie beachtet. Summand $\frac12$ etwa kommt $2(n-1)$-mal vor, Summand $\frac13$ kommt $2(n-2)$-mal vor, usw. Allgemein gilt: Summand $\frac1k$ kommt exakt $2(n-k+1)$-mal vor, für $2\le k\le n$. Also gilt:

$$\mathbf{E}(C) = \left(\sum_{2\le k\le n} \frac{2(n-k+1)}{k}\right) = 2(n+1)\cdot \sum_{2\le k\le n} \frac1k - 2(n-1).$$

Wir setzen

$$H_n := 1+\frac12+\frac13+\cdots+\frac1n, \text{ für } n\ge 1 \quad (n\text{-}te\ harmonische\ Zahl),$$

und erhalten

$$\mathbf{E}(C) = 2(n+1)(H_n-1) - 2(n-1) = 2(n+1)H_n - 4n.$$

Man weiß, dass $\ln n < H_n < 1+\ln n$ gilt.[14] Damit erhalten wir $\mathbf{E}(C) = 2n\ln n - O(n)$ oder $\mathbf{E}(C) = (2\ln 2)n\log_2 n - O(n)$, wobei $2\ln 2 = 1.386\ldots$ die „Quicksort-Konstante“ ist.

[14] Genaueres über Abschätzungen von H_n findet man z. B. in `https://www.tu-ilmenau.de/fileadmin/public/iti/Lehre/AuD/SS19/AuD-Kap-4-statisch.pdf`, Folien 37–39.

Man kann sogar den linearen Term genau bestimmen: Mit dem Wissen

$$H_n = \ln n + \gamma + \frac{1}{2n} - O(1/n^2),$$

wobei $\gamma = 0.5772156649\ldots$ die „Euler-Mascheroni-Konstante“ ist, erhält man

$$\mathbf{E}(C) = (2\ln 2)n\log_2 n - (4 - 2\gamma)n + 2\ln n + O(1),$$

mit $4 - 2\gamma \geq 2.845$.

Bemerkung: Was ist der Unterschied zwischen der Analyse von Quicksort mit deterministischer Pivotauswahl („immer $x = A[l + \lfloor (r-l)/2 \rfloor]$ wählen“) und der randomisierten Variante? Rein rechnerisch ergibt sich genau die gleiche Formel, wenn man annimmt, dass jede Anordnung der Eingabe die gleiche Wahrscheinlichkeit hat. Bei deterministischem Quicksort gibt es aber immer (worst-case-)Inputs, auf denen sich das Verfahren schlecht verhält, das heißt, Zeit $\Omega(n^2)$ benötigt. Bei der randomisierten Variante gilt dies nicht mehr: Auf jedem beliebigen Input $(a_1, \ldots, a_n)$ ist die *erwartete* Rechenzeit $O(n \log n)$. Quadratische Rechenzeiten treten auch hier auf, aber mit verschwindend kleiner Wahrscheinlichkeit, *für jeden beliebigen festen Input.* Die Randomisierung führt also zur Eliminierung von worst-case-Inputs.

2 Grundlagen aus der Wahrscheinlichkeitsrechnung

In diesem Kapitel sind die wichtigsten Konzepte der Wahrscheinlichkeitsrechnung zusammengestellt, die für die Zwecke unserer Vorlesung wichtig sind. Sie beschränken sich der Einfachheit halber auf den Fall endlicher und abzählbar unendlicher Wahrscheinlichkeitsräume.

Eine sehr gute Einführung in die Thematik findet sich in den ersten Kapiteln des Buchs „Probability and Computing – Randomized Algorithms and Probabilistic Analysis“ von M. Mitzenmacher und E. Upfal (s. Literaturverzeichnis auf der Webseite).

Ein Wort noch zur Verwendung dieses Kapitels und zu seiner Rolle in der Abschlussprüfung. Eigentlich wird angenommen, dass die folgenden Grundbegriffe der Wahrscheinlichkeitsrechnung aus der Veranstaltung „Stochastik für Informatiker“ bekannt sind. Infolgedessen sind sie nicht in erster Linie Prüfungsstoff.

- Wahrscheinlichkeitsraum, Verteilung
- Zufallsvariable, Erwartungswert, Linearität des E-Wertes, Markov-Ungleichung
- Varianz, Chebychev-Ungleichung, Jensensche Ungleichung
- uniforme Verteilung, Binomialverteilung, geometrische Verteilung
- bedingte Wahrscheinlichkeiten, bedingte Erwartungswerte
- Unabhängigkeit von Ereignissen und Zufallsvariablen, Rechenregeln.

Diese Begriffe werden wiederholend dargestellt, eventuell etwas modifiziert, und mit Beispielen unterlegt. Der Unterschied zur Behandlung in einer Stochastik-Vorlesung ist insbesondere, dass nur endliche und abzählbar unendliche Wahrscheinlichkeitsräume betrachtet werden. Anhand von Beispielen, die nahe an algorithmischen Anwendungen liegen, wird illustriert, wie die abstrakten Konzepte und Rechenregeln in algorithmischen und diskreten Anwendungen verwendet werden, und diese Verwendung wird im konkreten Kontext von Algorithmen und diskreten Strukturen eingeübt.

Wer also die wahrscheinlichkeitstheoretischen Konzepte kennt, kann die Definitionen und bekannten Aussagen in den Abschnitten 2.1–2.5 nur überfliegen, um die hier verwendete Notation kennenzulernen, Unbekanntes zu identifizieren (dies könnte zum Beispiel die Jensensche Ungleichung oder Fakt 2.2.9 sein) und die Beispiele anzusehen und gut zu verstehen. Die Ungleichung vom arithmetischen und geometrischen Mittel (Prop. 2.3.9) sollte man kennen. Verteilungen, die man unbedingt kennen sollte, sind die (diskreten) uniformen Verteilungen, die Binomialverteilungen, die geometrischen Verteilungen. Die Hoeffding-Ungleichung in Abschnitt 2.6 und die Ungleichungen in Abschnitt 2.7 sind sicher neu – sie sind zentrales Werkzeug und auch Thema der Vorlesung. Anhang A.1 beinhaltet eine Sammlung nützlicher Ungleichungen aus der Analysis und der Kombinatorik, die man ohnehin kennen sollte, ohne dass sie direkt Stoff der Vorlesung sind. Anhang A.2 stellt das Konzept der Summierbarkeit von unendlich vielen Zahlen bereit, das für die Modellierung in Kapitel 3 grundlegend ist.

2.1 Grundbegriffe, Beispiele

Definition 2.1.1

Ein **Wahrscheinlichkeitsraum** (**W-Raum**) ist ein Paar (Ω, p), wobei Ω eine endliche oder abzählbar unendliche Menge und $p\colon \Omega \to [0,1]$ eine Funktion ist, mit[1]

$$\sum_{\omega\in\Omega} p(\omega) = 1.$$

Wir schreiben oft p_ω statt $p(\omega)$. Eine solche Funktion $p\colon \Omega \to [0,1]$ heißt auch „**Verteilung**“ oder „**Wahrscheinlichkeitsverteilung**“.

Ein Wahrscheinlichkeitsraum ist eine mathematisch exakte Formulierung für das (informale, intuitive) Konzept eines „*Zufallsexperiments*“: Es wird „zufällig“ ein Element aus Ω ausgewählt; dabei ist die Wahrscheinlichkeit, gerade ω zu erhalten, durch $p(\omega)$ gegeben. Die Elemente ω von Ω heißen *Ergebnisse* (gemeint ist „mögliche Ergebnisse des Zufallsexperiments“) oder *Elementarereignisse*. Man teste diese intuitive

[1] Wenn Ω unendlich ist, ist die Schreibweise $\sum_{\omega\in\Omega} p(\omega)$, die Summation ohne Berücksichtigung einer Reihenfolge ausdrückt, in den Mathematikvorlesungen nicht verwendet worden. Die Definition und einige Kommentare hierzu finden sich in Anhang A.2. Man kann die Schreibweise aber auch einfach benutzen und sich vorstellen, dass durch das Bestehen auf absoluter Konvergenz (unabhängig von irgendwelchen Reihenfolgen) nie Probleme beim Umgang mit solchen Summen auftreten können, und die Standardrechenregeln wie bei endlichen Summen gelten.

Auffassung an den folgenden Beispielen.

Beispiele 2.1.2

(a) Zur Modellierung des Zufallsexperiments, einen fairen Würfel einmal zu werfen, benutzt man den Wahrscheinlichkeitsraum (Ω, p) mit $\Omega = \{1, \dots, 6\}$ und $p(\omega) = \frac{1}{6}$ für jedes $\omega \in \{1, \dots, 6\}$.
Um das Werfen einer fairen Münze zu modellieren, wird man (mit „0“ für „Kopf“ und „1“ für „Zahl“) den W-Raum $\Omega = \{0, 1\}$ und $p(\omega) = \frac{1}{2}$ verwenden. Ist die Münze gefälscht, könnte man z. B. $p(0) = 0.55$ und $p(1) = 0.45$ setzen.

(b) Zur Modellierung des Zufallsexperiments, zwei Würfel zu werfen und die Summe der Augenzahlen als Ergebnis zu nehmen, wird man etwa $\Omega = \{2, \dots, 12\}$ und $p(2) = \frac{1}{36}$, $p(3) = \frac{2}{36}$, $p(4) = \frac{3}{36}$, $\dots$, $p(7) = \frac{6}{36}$, $p(8) = \frac{5}{36}$, $\dots$, $p(12) = \frac{1}{36}$ wählen. Man beachte, dass hier die Wahrscheinlichkeiten unterschiedlich sind.

(c) $U \neq \emptyset$ sei eine endliche Menge. Wir modellieren das Zufallsexperiment, ein Element aus U zu wählen, wobei jedes Element die gleichen Chancen haben soll, wie folgt: $\Omega = U$ und $p_\omega = \frac{1}{|U|}$, für alle $\omega \in \Omega$. Diese Wahrscheinlichkeitsverteilung heißt „*uniforme Verteilung*“ oder „(diskrete) Gleichverteilung“ auf U. Gewöhnlich ist implizit diese Verteilung gemeint, wenn über die Wahrscheinlichkeiten der einzelnen Elemente gar nichts gesagt wird oder wenn die Formulierung „wähle zufällig ein Element aus U“ benutzt wird.[2]

(d) Wir wollen wiederholt mit einem Würfel würfeln und warten, bis die erste „6“ erscheint. Das Ergebnis des Experiments soll die Anzahl der benötigten Würfe sein. Um dies zu modellieren, setzen wir $\Omega = \{1, 2, 3, \dots\}$ und $p_i = (\frac{5}{6})^{i-1} \cdot \frac{1}{6}$ als die Wahrscheinlichkeit, dass beim i-ten Versuch zum ersten Mal eine „6“ gewürfelt wird. (Bei den ersten $i - 1$ Versuchen keine „6“, jeweils mit Wahrscheinlichkeit $1 - \frac{1}{6} = \frac{5}{6}$, bei Versuch Nummer i eine „6“, mit Wahrscheinlichkeit $\frac{1}{6}$.) Man sieht, mit der Summenformel für geometrische Reihen:

$$\sum_{i \geq 1} p_i = \sum_{i \geq 1} \left(\frac{5}{6}\right)^{i-1} \cdot \frac{1}{6} = \frac{1}{6} \cdot \sum_{i \geq 1} \left(\frac{5}{6}\right)^{i-1} = \frac{1}{6} \cdot \frac{1}{1 - \frac{5}{6}} = 1.$$

[2] Man nennt die uniforme Verteilung auch *Laplace-Verteilung*, nach Pierre-Simon Laplace (1749–1827), einem französischen Mathematiker, der postulierte, dass man ohne Information, die ein Elementarereignis vor einem anderen bevorzugt, die uniforme Verteilung annehmen sollte (Indifferenzprinzip, `https://de.wikipedia.org/wiki/Indifferenzprinzip`).

Damit haben wir tatsächlich einen Wahrscheinlichkeitsraum definiert. Die hier definierte Verteilung heißt „*geometrische Verteilung* mit Parameter $p = \frac{1}{6}$“. (In Abschnitt 2.5.1 werden geometrische Verteilungen allgemein diskutiert.)

(e) Es sei $U \neq \emptyset$ eine endliche Menge und $n \geq 1$. Der W-Raum (Ω, p) mit

$$\Omega = U^n = \{(a_1, \ldots, a_n) \mid a_1, \ldots, a_n \in U\}$$

und $p_\omega = \frac{1}{|U|^n}$ für $\omega \in \Omega$, das ist also die uniforme Verteilung auf U^n, entspricht dem Zufallsexperiment, bei dem eine Folge von n Elementen aus U zufällig gewählt wird, bzw. n-mal hintereinander ein Element aus U zufällig gewählt wird.

(f) Es sei $U \neq \emptyset$ eine endliche Menge und $1 \leq n \leq |U|$. Wir wollen das Zufallsexperiment „Wähle eine zufällige n-elementige Teilmenge von U“ modellieren. Dazu wählen wir $\Omega = \{S \subseteq U \mid |S| = n\}$ als Grundmenge mit der Verteilung, die durch $p_S = 1/\binom{|U|}{n}$ für alle $S \in \Omega$ gegeben ist.

(g) Für die Durchschnittsanalyse von Sortierverfahren, die n Schlüssel aus dem angeordneten Universum $(U, <)$ sortieren, ist die folgende Verteilung zentral. Für Sortierverfahren, die auf Schlüsseln nur Vergleiche und keine anderen Operationen durchführen, ist der Ablauf des Verfahrens im Wesentlichen durch den „*Ordnungstyp*“ der Eingabe $(a_1, \ldots, a_n) \in U^n$ bestimmt, das ist die Permutation π von $\{1, \ldots, n\}$ mit $a_{\pi(1)} < \cdots < a_{\pi(n)}$. Diese ist eindeutig bestimmt, wenn $a_1, \ldots, a_n$ verschieden sind. Daher betrachten wir

$$\Omega = \{\pi \mid \pi \text{ Permutation von } \{1, \ldots, n\}\},$$

mit der durch $p(\pi) = 1/|\Omega| = 1/n!$ gegebenen Verteilung. Dieser W-Raum entspricht dem Experiment, für n beliebig vorgegebene Elemente von U die Anordnung rein zufällig zu wählen.

(h) Beim Hashing betrachtet man n Schlüssel $x_1, \ldots, x_n$ und n Funktionswerte $h(x_1)$, $\ldots, h(x_n)$ in $[m] := \{0, 1, \ldots, m-1\}$. Es gibt dabei verschiedene Wahrscheinlichkeitsannahmen, die zu verschiedenen Wahrscheinlichkeitsräumen führen. Wenn man etwa die „Uniformitätsannahme“ für eine Hashfunktion macht, meint man damit, dass der Hashwert eines jeden Schlüssels unabhängig von den anderen jeden Wert in $[m]$ mit derselben Wahrscheinlichkeit annimmt. Der zugehörige Wahrscheinlichkeitsraum ist

$$\Omega = [m]^n = \{(v_1, \ldots, v_n) \mid v_1, \ldots, v_n \in [m]\}$$

mit der durch

$$p((v_1, \ldots, v_n)) = \frac{1}{m^n}$$

definierten Verteilung. (Das ist derselbe Wahrscheinlichkeitsraum wie der in (e), wenn man $U = [m]$ setzt.)

Definition 2.1.3

Ein **Ereignis** ist eine Menge $A \subseteq \Omega$.
Die **Wahrscheinlichkeit** von A ist $\mathbf{Pr}(\boldsymbol{A}) := \sum_{\omega \in A} p_\omega$.

Bemerkung 2.1.4

Wenn $p\colon \Omega \to [0,1]$ eine Wahrscheinlichkeitsverteilung ist, dann ist $(p_\omega)_{\omega \in A}$ für jedes Ereignis $A \subseteq \Omega$ summierbar (s. Prop. A.2.3(a) im Anhang), also ist $\sum_{\omega \in A} p_\omega$ wohldefiniert.

Notation: Ist φ eine Eigenschaft oder (synonym) eine Aussage, die für ein Ergebnis $\omega \in \Omega$ gelten oder nicht gelten kann, so ist $A = A_\varphi = \{\omega \in \Omega \mid \varphi(\omega)\}$ ein Ereignis. Oft schreibt man hierfür kurz $\{\varphi\}$. Die Wahrscheinlichkeit $\mathbf{Pr}(A) = \mathbf{Pr}(\{\varphi\})$ wird dann als $\mathbf{Pr}(\varphi)$ abgekürzt.

In den folgenden Beispielen sieht man, dass der Name „Ereignis“ und die abkürzende Schreibweise für durch Aussagen gegebene Ereignisse und ihre Wahrscheinlichkeiten recht gut zur Intuition passt. Man beachte, dass in der Notation der W-Raum und auch der Bezug auf einzelne Ergebnisse unterdrückt wird, wann immer es geht.

Beispiel 2.1.5

(a) In Beispiel 2.1.2(b) ist

$$A = \{\omega \in \Omega \mid \omega \geq 6\} = \{\text{Augensumme} \geq 6\}$$

ein Ereignis, das die Situation modelliert, dass die Summe der Augen mindestens 6 beträgt. Man schreibt $\mathbf{Pr}(\text{Augensumme} \geq 6)$ für $\mathbf{Pr}(A)$. Es gilt

$$\mathbf{Pr}(A) = \mathbf{Pr}(\{6,7,8,9,10,11,12\}) = \frac{5}{36} + \frac{6}{36} + \frac{5}{36} + \frac{4}{36} + \frac{3}{36} + \frac{2}{36} + \frac{1}{36} = \frac{13}{18}.$$

(b) In Beispiel 2.1.2(h) ist

$$A = \{(v_1, \ldots, v_n) \mid v_1 = v_2 = v_3\}$$

ein Ereignis, das man auch als $\{h(x_1) = h(x_2) = h(x_3)\}$ schreiben kann. Es gilt

$$\mathbf{Pr}(A) = \mathbf{Pr}(h(x_1) = h(x_2) = h(x_3)) = |A|/m^n = m^{n-2}/m^n = 1/m^2.$$

Beachte *allgemein*: Ist (Ω, p) die ***uniforme Verteilung*** auf Ω, d. h. $p_\omega = 1/|\Omega|$ für alle $\omega \in \Omega$, so ist $\mathbf{Pr}(A) = |A|/|\Omega|$.

Fakt 2.1.6

(a) $\mathbf{Pr}(\emptyset) = 0$ („das unmögliche Ereignis“),
$\mathbf{Pr}(\Omega) = 1$ („das sichere Ereignis“),
$\mathbf{Pr}(\overline{A}) = \mathbf{Pr}(\Omega - A) = 1 - \mathbf{Pr}(A)$ („Komplementärereignis“),
$\mathbf{Pr}(\{\omega\}) = p_\omega$, für $\omega \in \Omega$.

(b) Sind $A_1, \ldots, A_n$ *disjunkte* Ereignisse, so ist

$$\mathbf{Pr}(A_1 \cup \cdots \cup A_n) = \sum_{1 \le i \le n} \mathbf{Pr}(A_i) \qquad (\textbf{\textit{Additivität}}).$$

Sind $A_1, A_2, \ldots$ (abzählbar unendlich viele) *disjunkte* Ereignisse, so ist

$$\mathbf{Pr}(\bigcup_{i \ge 1} A_i) = \sum_{i \ge 1} \mathbf{Pr}(A_i) \qquad (\textbf{\textit{σ-Additivität}}).$$

(c) Sind $A_1, \ldots, A_n$ *beliebige* Ereignisse, so ist

$$\mathbf{Pr}(A_1 \cup \cdots \cup A_n) \le \sum_{1 \le i \le n} \mathbf{Pr}(A_i).$$

Sind $A_1, A_2, \ldots$ (abzählbar unendlich viele) Ereignisse, so ist

$$\mathbf{Pr}(\bigcup_{i \ge 1} A_i) \le \sum_{i \ge 1} \mathbf{Pr}(A_i)$$

(***Vereinigungsschranke*** oder englisch ***Union Bound***).

(d) Ist $A_1 \subseteq A_2$, so ist $\mathbf{Pr}(A_1) \le \mathbf{Pr}(A_2)$ (***Monotonie***).

Die Gültigkeit der Aussagen in Fakt 2.1.6 kann man mittels Def. 2.1.3 nachkontrollieren. Man benutzt, dass Assoziativität, Distributivität und Monotonie auch bei

unendlichen Summen gelten, s. Prop. A.2.3 im Anhang.

Formel 2.1.6(d) wird oft folgendermaßen benutzt: Wenn für jedes $\omega \in \Omega$ aus der Aussage $\varphi(\omega)$ die Aussage $\psi(\omega)$ *folgt*, dann gilt $\{\varphi\} \subseteq \{\psi\}$ und damit $\mathbf{Pr}(\varphi) \leq \mathbf{Pr}(\psi)$.

Beispiel 2.1.7
In Beispiel 2.1.2(h) gilt für jedes $v \in [m] = \{0, 1, \ldots, m-1\}$:

$$\mathbf{Pr}(\exists i \in \{1, \ldots, n\} \colon h(x_i) = v) \leq \sum_{1 \leq i \leq n} \mathbf{Pr}(h(x_i) = v) = n \cdot \frac{1}{m}.$$

(*Übung*: Man mache die hier benutzten Ereignisse explizit und benenne die Regeln, die in den einzelnen Rechenschritten angewendet werden.)

2.2 Zufallsvariablen und Erwartungswerte

Definition 2.2.1
Sei (Ω, p) ein W-Raum und R eine beliebige Menge. Eine Funktion $X \colon \Omega \to R$ heißt eine **Zufallsfunktion**. Ist R numerisch (also $R \subseteq \mathbb{R}$), so heißt ein solches X eine **Zufallsvariable** (**ZV**), im Fall $R \subseteq \mathbb{R}^k$ für ein $k \geq 1$ auch ein **Zufallsvektor**.

Die Idee dabei ist natürlich, dass man ein $\omega \in \Omega$ zufällig wählt (gesteuert von der Verteilung $p \colon \Omega \to [0, 1]$), und dass dadurch auch ein zufälliger Wert $X(\omega)$ festgelegt wird.

Zur Schreibweise: Soweit möglich schreibt man X statt $X(\omega)$. Ist beispielsweise $R' \subseteq R$, betrachtet man das Ereignis $\{X \in R'\} = X^{-1}(R') = \{\omega \mid X(\omega) \in R'\}$, und die Wahrscheinlichkeit $\mathbf{Pr}(X \in R')$, usw. Achtung: „X“ sieht aus wie ein Wert, ist aber variabel (mit ω).

Bemerkung 2.2.2
Eine ZV X mit Wertebereich $\{0, 1\}$ bezeichnet man als Indikator(-zufallsvariable). Solche Zufallsvariablen werden mit Hilfe einer Aussage φ wie folgt konstruiert:

$$X(\omega) := \begin{cases} 1, & \text{falls } \varphi(\omega) \text{ wahr ist,} \\ 0, & \text{sonst.} \end{cases}$$

Um Indikatorzufallsvariablen kompakt zu notieren (und nicht jedes Mal die Fallunterscheidung hinschreiben zu müssen) hat sich die **Iverson-Notation** bewährt: Für das X wie oben schreibt man $[\varphi]$. Für die Aussage „Augensumme ≥ 6“ (Beispiel 2.1.5 (a)) könnte man also einen entsprechenden Indikator mit „[Augensumme ≥ 6]“ angeben.

Beispiel 2.2.3
Betrachte Beispiel 2.1.2(h), also $\Omega = [m]^n = \{\omega = (v_1, \ldots, v_n) \mid v_1, \ldots, v_n \in [m]\}$.

(a) Für $1 \leq i \leq n$ ist die Funktion $\omega \mapsto v_i = h(x_i)$ eine Zufallsvariable.

(b) Für $v \in [m]$ ist die Funktion $\omega \mapsto B_v = \{i \mid v_i = v\} = \{i \mid h(x_i) = v\}$ eine Zufallsfunktion (der Wert ist eine „zufällige Menge“ oder „Zufallsmenge“, die den Schlüsseln x_i entspricht, die von h auf den Wert v abgebildet werden); die Funktion $b_v \colon \omega \mapsto |B_v|$ der Anzahl dieser Schlüssel ist eine ZV.

Jede Zufallsfunktion $X \colon \Omega \to R$ induziert einen neuen Wahrscheinlichkeitsraum, wie folgt:

$$\Omega' := X[\Omega] = \{X(\omega) \mid \omega \in \Omega\} \subseteq R;\ p'(\alpha) := \mathbf{Pr}(X = \alpha) \text{ für } \alpha \in \Omega'. \tag{2.2.1}$$

Die Verteilung p' heißt die ***Verteilung von X***. Wenn es bequem ist, kann man auch eine (endliche oder abzählbare) Menge R' mit $X[\Omega] \subseteq R' \subseteq R$ als Grundmenge Ω' benutzen.

Bemerkung 2.2.4
Für jeden Wahrscheinlichkeitsraum (Ω, p) ist die Funktion $p \colon \Omega \to [0, 1]$ Verteilung einer Zufallsvariablen X. Man wählt einfach $X = \mathrm{id}_\Omega$, die Identität, die ω auf ω abbildet, und erhält $\Omega' = \Omega$ und $p' = p$.

Beispiel 2.2.5
(a) Beim Werfen von zwei fairen Würfeln ist folgender Wahrscheinlichkeitsraum mit einer uniformen Verteilung natürlich:

$$\Omega = \{1, \ldots, 6\} \times \{1, \ldots, 6\};\ \ p((i, j)) = \frac{1}{36} \text{ für } (i, j) \in \Omega.$$

Die durch $X((i, j)) := i + j$ definierte Abbildung $X \colon \Omega \to \{2, \ldots, 12\}$ ist eine Zufallsvariable. Die Verteilung von X ist gerade die Verteilung des in Beispiel 2.1.2(b) beschriebenen Wahrscheinlichkeitsraums.

(b) Beim Spiel „Würfeln, bis eine 6 erscheint“ ist folgender Wahrscheinlichkeitsraum natürlich:

$$\Omega = \{(a_1, \ldots, a_i) \mid i \geq 1, a_1 \ldots, a_{i-1} \in \{1, \ldots, 5\}, a_i = 6\};$$
$$p((a_1, \ldots, a_i)) = \left(\frac{1}{6}\right)^i, \text{ für } (a_1 \ldots, a_i) \in \Omega.$$

Ein Elementarereignis ist hier eine Folge von Ergebnissen einzelner Würfe, die abbricht, sobald die erste 6 erschienen ist. Jede solche Folge hat, intuitiv gesehen, die Wahrscheinlichkeit $(1/6)^i$. Die durch $X((a_1, \ldots, a_i)) = i$ gegebene Zufallsvariable zählt die Anzahl dieser Versuche. Ihre Verteilung liefert den Wahrscheinlichkeitsraum aus Beispiel 2.1.2(d).

Beispiel 2.2.6
Wir führen Beispiel 2.2.3 noch etwas weiter. Die Zufallsvariable $b_0 = |B_0|$ induziert eine Verteilung auf $b_0[\Omega] = \{0, 1, \ldots, n\}$. Dabei ist

$$\begin{aligned} p'(i) &= \frac{|\{(v_1, \ldots, v_n) \in \Omega \mid (v_1, \ldots, v_n) \text{ enthält genau } i \text{ Nullen}\}|}{m^n}. \\ &= \binom{n}{i} \cdot \frac{(m-1)^{n-i}}{m^n} = \binom{n}{i} \cdot \left(\frac{1}{m}\right)^i \cdot \left(1 - \frac{1}{m}\right)^{n-i}. \end{aligned}$$

(Dies ist eine *Binomialverteilung.*) Natürlich ergibt sich für jedes $v \in [m]$ anstelle von 0 dieselbe Verteilung.

Der *Erwartungswert* einer Zufallsvariablen ist ein sehr grundlegendes Konzept. Es modelliert den „durchschnittlichen Wert“ der Zufallsvariablen, indem er die Funktionswerte zu Elementarereignissen, mit der entsprechenden Wahrscheinlichkeit gewichtet, aufsummiert. Bei Bedarf schaue man sich das Konzept einer summierbaren Familie von Zahlen (s. Abschnitt A.2) an.

Definition 2.2.7
Für eine Zufallsvariable X, für die $(X(\omega) \cdot p_\omega)_{\omega \in \Omega}$ summierbar ist, definieren wir den **Erwartungswert** von X durch:

$$\mathbf{E}(\boldsymbol{X}) := \sum_{\omega \in \Omega} X(\omega) \cdot p_\omega = \sum_{\alpha \in X[\Omega]} \alpha \cdot \mathbf{Pr}(X = \alpha).$$

Die erste Summe betrachtet die Situation eher vom W-Raum (Ω, p) aus, die zweite eher vom Wertebereich und der Verteilung auf $X[\Omega]$ aus. Für nicht-negative Zufallsvariablen X lässt man mitunter auch den Fall $\mathbf{E}(X) = \infty$ zu (wenn Ω unendlich ist und die Menge $\{\sum_{\omega\in A} X(\omega)\cdot p_\omega \mid A \subseteq \Omega \text{ endlich}\}$ unbeschränkt ist).

Beispiele 2.2.8
(a) Es sei $\Omega = \mathbb{N}^+ = \{1, 2, 3, \dots\}$ und $p_i = 2^{-i}$ für $i \in \mathbb{N}^+$. Dann hat die Zufallsvariable X mit $X(i) = (-1)^i \cdot i^2$ für $i \in \mathbb{N}^+$ einen Erwartungswert (weil $\sum_{i=1}^\infty i^2/2^i$ konvergent ist), die Zufallsvariable Y mit $Y(i) = (-2)^i$, $i \in \mathbb{N}^+$, dagegen nicht (weil $\sum_{i=1}^\infty 2^i/2^i = \sum_{i\geq 1} 1$ divergent ist, also beliebig große endliche Teilsummen besitzt).
(b) Man erinnere sich an die bekannte Formel $\sum_{i=1}^\infty \frac{1}{i(i+1)} = 1$. (Das liegt daran, dass $\frac{1}{i(i+1)} = \frac{1}{i} - \frac{1}{i+1}$ gilt.) Mit $\Omega = \mathbb{N}^+$ und $p_i = \frac{1}{i(i+1)}$ für $i \in \mathbb{N}^+$ erhalten wir also einen W-Raum. In diesem W-Raum hat die Zufallsvariable X mit $X(i) = (-1)^{i-1}$ für $i \in \mathbb{N}^+$ einen Erwartungswert, nämlich $\mathbf{E}(X) = \sum_{i\geq 1} \frac{(-1)^{i-1}}{i(i+1)}$ $(= 2\ln 2 - 1)$. Dagegen hat die Zufallsvariable Y mit $Y(i) = (-1)^{i-1}\cdot i$ keinen Erwartungswert, da die Reihe $\sum_{i=1}^\infty \frac{(-1)^{i-1}\cdot i}{i(i+1)} = \sum_{i=1}^\infty \frac{(-1)^{i-1}}{i+1} = \frac{1}{2} - \frac{1}{3} + \frac{1}{4} - \frac{1}{5} \pm \dots$, zwar bedingt konvergent ist (mit Grenzwert $1 - \ln 2$), aber nicht summierbar ist.

Die zweite Darstellung des Erwartungswertes in Definition 2.2.7 lässt sich durch Umstellen und Gruppieren von Summen beweisen. Ein solches Umstellen und Umklammern ist hier kein Problem, weil alle auftretenden Reihen summierbar sind, siehe Anhang A.2. Man kann diese zweite Darstellung auch so auffassen: Man betrachtet die Verteilung von X, die jeder Zahl α im Wertebereich $X[\Omega]$ die Wahrscheinlichkeit $p'(\alpha) = \mathbf{Pr}(X = \alpha)$ zuordnet, und bildet den Mittelwert dieser Zahlen, gewichtet mit diesen Wahrscheinlichkeiten.

Die folgende Formel für nicht-negative, ganzzahlige Zufallsvariablen X ist äußerst nützlich, wenn sich $\mathbf{Pr}(X \geq i)$ bequem berechnen oder abschätzen lässt. Wir werden sie häufig benutzen.

Fakt 2.2.9
Sei $X\colon \Omega \to \mathbb{N}$ eine Zufallsvariable, deren Erwartungswert existiert. Dann gilt:

$$\mathbf{E}(X) = \sum_{i\geq 1} \mathbf{Pr}(X \geq i) \qquad \big(= \sum_{i\geq 0} \mathbf{Pr}(X \geq i+1)\big).$$

Beweis. Setze $p_j = \mathbf{Pr}(X = j)$, $q_i = \mathbf{Pr}(X \geq i)$. Dann gilt: $q_i = \sum_{j \geq i} p_j$, also

$$\mathbf{E}(X) = \sum_{j \geq 0} j \cdot p_j = \sum_{j \geq 1} j \cdot p_j = \sum_{j \geq 1} \sum_{1 \leq i \leq j} p_j = \sum_{i \geq 1} \sum_{j \geq i} p_j = \sum_{i \geq 1} q_i.$$

□

Beispiel: In Beispiel 2.1.2(d) (Würfeln, bis die erste „6“ erscheint) definieren wir die Zufallsvariable $X :=$ Anzahl der Würfe bis zur ersten 6 (einschließlich). Technisch ist das in diesem Wahrscheinlichkeitsraum einfach $X(i) := i$, für $i \geq 1$. Intuitiv sieht man, dass $\mathbf{Pr}(X \geq i) = (\frac{5}{6})^{i-1}$ sein muss (Misserfolg in den ersten $i-1$ Würfen). Zur Sicherheit rechnen wir das nach:

$$\begin{aligned}\mathbf{Pr}(X \geq i) &= \sum_{j \geq i} \mathbf{Pr}(X = j) = \sum_{j \geq i} \left(\frac{5}{6}\right)^{j-1} \cdot \frac{1}{6} = \frac{1}{6} \cdot \left(\frac{5}{6}\right)^{i-1} \cdot \sum_{j \geq i} \left(\frac{5}{6}\right)^{j-i} \\ &= \frac{1}{6} \cdot \left(\frac{5}{6}\right)^{i-1} \cdot \frac{1}{1 - \frac{5}{6}} = \left(\frac{5}{6}\right)^{i-1}.\end{aligned}$$

Mit Fakt 2.2.9 ergibt sich:

$$\mathbf{E}(X) = \sum_{i \geq 1} \mathbf{Pr}(X \geq i) = \sum_{i \geq 1} \left(\frac{5}{6}\right)^{i-1} = \frac{1}{1 - \frac{5}{6}} = 6.$$

Die Verallgemeinerung dieser Situation auf beliebige Erfolgswahrscheinlichkeiten p anstelle von $\frac{1}{6}$ führt zu der geometrischen Verteilung zu Parameter p, siehe Abschnitt 2.5.1.

Fakt 2.2.10

Für beliebige Zufallsvariablen $X, Y, X_1, \ldots, X_n$ gilt (unter der Voraussetzung, dass alle Erwartungswerte definiert sind):

(a) $X \leq Y$ (d. h. $\forall \omega \in \Omega$: $X(\omega) \leq Y(\omega)$) $\Rightarrow \mathbf{E}(X) \leq \mathbf{E}(Y)$ (***Monotonie***).

(b) $\mathbf{E}(\alpha X + \beta Y) = \alpha \mathbf{E}(X) + \beta \mathbf{E}(Y)$, für beliebige $\alpha, \beta \in \mathbb{R}$ (***Linearität des Erwartungswertes I***).

(c) $\mathbf{E}(X_1 + \cdots + X_n) = \mathbf{E}(X_1) + \cdots + \mathbf{E}(X_n)$ (***Linearität des Erwartungswertes II***).

(d) $X \in \{0, 1\}$ (d. h. $\forall \omega \in \Omega$: $X(\omega) \in \{0, 1\}$) $\Rightarrow \mathbf{E}(X) = \mathbf{Pr}(X = 1)$.

Die *Beweise* von (a), (b), (c) sind (einfache) Anwendungen der Rechenregeln für Summen. Für (d) beobachtet man $\mathbf{E}(X) = \mathbf{Pr}(X = 0) \cdot 0 + \mathbf{Pr}(X = 1) \cdot 1$.

Bemerkung 2.2.11
Für eine Indikatorvariable $[\varphi]$ gilt nach Fakt 2.2.10 $\mathbf{E}([\varphi]) = \mathbf{Pr}(\varphi)$.

Beispiel 2.2.12
Betrachte Bsp. 2.2.3(b). Wir berechnen $\mathbf{E}(|B_v|)$ mit Hilfe der Indikatorvariablen $[h(x_i) = v]$, für $i \in \{1, 2, \ldots, n\}$. Klar: $|B_v| = [h(x_1) = v] + \cdots + [h(x_n) = v]$. Also gilt

$$\mathbf{E}(|B_v|) = \sum_{1 \le i \le n} \mathbf{E}([h(x_i) = v]) = \sum_{1 \le i \le n} \mathbf{Pr}(h(x_i) = v) = \sum_{1 \le i \le n} \frac{1}{m} = \frac{n}{m}.$$

Ein anderes, sehr typisches Beispiel für die Anwendung der Technik der Zerlegung einer Zufallsvariablen in Summanden, die Indikatorvariablen sind, ist die Analyse von Quicksort in Abschnitt 1.4. (Hier ist der richtige Zeitpunkt, diese Analyse nochmals mit vollem Verständnis durchzusehen.)

2.3 Varianz und Ungleichungen von Markov, Chebychev und Jensen

Der Zweck der folgenden fundamentalen Ungleichung ist, bei gegebenem Erwartungswert einer nichtnegativen Zufallsvariablen Z die Wahrscheinlichkeit dafür zu begrenzen, dass Z sehr große Werte hat.

Fakt 2.3.1 (*Markoff/Markov-Ungleichung*)
Es sei $Z \ge 0$ eine beliebige Zufallsvariable, und $t > 0$ sei beliebig. Dann gilt:

$$\mathbf{Pr}(Z \ge t) \le \frac{\mathbf{E}(Z)}{t}.$$

Beweis. Offenbar gilt $Z \ge t \cdot [Z \ge t]$ (eine Ungleichung für jedes $\omega \in \Omega$), also auch $\mathbf{E}(Z) \ge t \cdot \mathbf{E}([Z \ge t]) = t \cdot \mathbf{Pr}(Z \ge t)$ (mit Monotonie und Linearität des Erwartungswertes und Bem. 2.2.11). Dividieren durch t liefert die Behauptung. □

Definition 2.3.2
Für eine beliebige Zufallsvariable X, für die $\mathbf{E}(X^2)$ existiert, definieren wir[3] die **Varianz** von X als

$$\mathbf{Var}(X) := \mathbf{E}((X - \mathbf{E}(X))^2).$$

Bemerkung 2.3.3
Für jedes $a \in \mathbb{R}$ gilt $\mathbf{Var}(X - a) = \mathbf{Var}(X)$. Insbesondere haben wir für $X' := X - \mathbf{E}(X)$ die Beziehungen $\mathbf{E}(X') = 0$ und $\mathbf{Var}(X') = \mathbf{Var}(X)$. Weiter gilt $\mathbf{Var}(a \cdot X) = a^2 \cdot \mathbf{Var}(X)$, für jedes $a \in \mathbb{R}$.

Man sieht sofort, dass gilt:

$$\mathbf{Var}(X) = \mathbf{E}(X^2 - 2X\mathbf{E}(X) + \mathbf{E}(X)^2) = \mathbf{E}(X^2) - 2\mathbf{E}(X)^2 + \mathbf{E}(X)^2 = \mathbf{E}(X^2) - \mathbf{E}(X)^2. \tag{2.3.2}$$

Als Erwartungswert von $(X - \mathbf{E}(X))^2 \geq 0$ ist $\mathbf{Var}(X) \geq 0$. Mit (2.3.2) folgt

$$\mathbf{E}(X)^2 \leq \mathbf{E}(X^2) \tag{2.3.3}$$

für jede Zufallsvariable X, deren Varianz existiert.

Die Varianz misst in einem gewissen Sinn die Tendenz einer Zufallsvariablen, Werte anzunehmen, die weit von ihrem Erwartungswert entfernt sind. (Schön ist immer, wenn sich die Werte einer Zufallsvariablen ganz in der Nähe ihres Erwartungswertes befinden. Dieser Situation entspricht eine kleine Varianz.) Durch das Quadrieren in der Definition der Varianz werden große Entfernungen stark betont. Eine der zentralen Anwendungen der Varianz ist die folgende Ungleichung, die bei gegebener Varianz $\mathbf{Var}(X)$ die Wahrscheinlichkeit dafür begrenzt, dass X weit von seinem Erwartungswert entfernt liegt. Für den Beweis wendet man einfach auf die Zufallsvariable $Z = (X - \mathbf{E}(X))^2 \geq 0$ die Markov-Ungleichung an.

Fakt 2.3.4 (*Chebychev/Tschebyscheff-Ungleichung*)
Es sei X eine Zufallsvariable, deren Varianz existiert. Dann gilt für jedes $t > 0$:

$$\mathbf{Pr}(|X - \mathbf{E}(X)| \geq t) \leq \frac{\mathbf{Var}(X)}{t^2}.$$

[3] Wenn $\mathbf{E}(X^2) = \sum_{\omega \in \Omega} p_\omega X(\omega)^2$ definiert ist, dann ist auch $\mathbf{E}(X)$ definiert. Wieso?

Beweis. Setze $Z := (X - \mathbf{E}(X))^2$. Dann gilt nach der Markov-Ungleichung:

$$\mathbf{Pr}(|X - \mathbf{E}(X)| \geq t) = \mathbf{Pr}(Z \geq t^2) \leq \frac{\mathbf{E}(Z)}{t^2} = \frac{\mathbf{Var}(X)}{t^2}.$$

□

Wir können die Markov-Ungleichung verallgemeinern:

Proposition 2.3.5

X sei eine beliebige Zufallsvariable, $D \subseteq \mathbb{R}$, $f\colon D \to \mathbb{R}^+$ sei monoton wachsend mit $D = \mathrm{Def}(f) \supseteq X[\Omega]$, so dass $\mathbf{E}(f(X))$ existiert. Dann gilt für jedes $t \in D$:

$$\mathbf{Pr}(X \geq t) \leq \frac{\mathbf{E}(f(X))}{f(t)}.$$

Beweis: Man wendet die Markov-Ungleichung auf die Zufallsvariable $f(X)$ an, und verwendet, dass wegen der Monotonie von f die Aussagen $X \geq t$ und $f(X) \geq f(t)$ äquivalent sind. □

Beispiele 2.3.6
Sei X eine Zufallsvariable.

(a) Sei $\alpha > 0$ beliebig, so dass $\mathbf{E}(|X|^\alpha)$ existiert. Dann gilt für $t > 0$:

$$\mathbf{Pr}(X \geq t) \leq \frac{\mathbf{E}(|X|^\alpha)}{t^\alpha}.$$

(b) Sei $k \geq 2$ eine gerade natürliche Zahl, so dass $\mathbf{E}(X^k)$ existiert. Dann gilt für $t > 0$:

$$\mathbf{Pr}(|X - \mathbf{E}(X)| \geq t) \leq \frac{\mathbf{E}((X - \mathbf{E}(X))^k)}{t^k}.$$

(Hier wird Prop. 2.3.5 auf die Zufallsvariable $Z = |X - \mathbf{E}(X)|$ und $f(x) = x^k$ angewendet.)

(c) Seien $t, c > 0$ beliebig, und sei X eine Zufallsvariable, deren Varianz existiert. Dann gilt

$$\mathbf{Pr}((X + c)^2 \geq (t + c)^2) \leq \frac{\mathbf{E}((X + c)^2)}{(t + c)^2}.$$

Diese Ungleichung kann man für den Beweis der Chebychev-Cantelli-Ungleichung (Prop. 2.7.2) benutzen (siehe Übung).

(d) Sei X reellwertig, sei $a > 0$, und sei $\mathbf{E}(e^{aX}) < \infty$. Dann gilt für jedes $t \in \mathbb{R}$:

$$\mathbf{Pr}(X \geq t) \leq \frac{\mathbf{E}(e^{aX})}{e^{at}}.$$

(Dies ist die ursprüngliche **„Chernoff-Schranke“** von 1952. Wir werden sie weiter unten benutzen, um eine spezialisierte Folgerung, die *Hoeffding-Schranke*, zu beweisen.)

Ungleichung (2.3.3) besagt, dass stets $\mathbf{E}(X)^2 \leq \mathbf{E}(X^2)$ gilt. Diese Ungleichung wollen wir verallgemeinern, indem wir anstelle der Funktion $x \mapsto x^2$ irgendeine *konvexe* Funktion benutzen. Wir erinnern an die Definition von konvexen Funktionen.

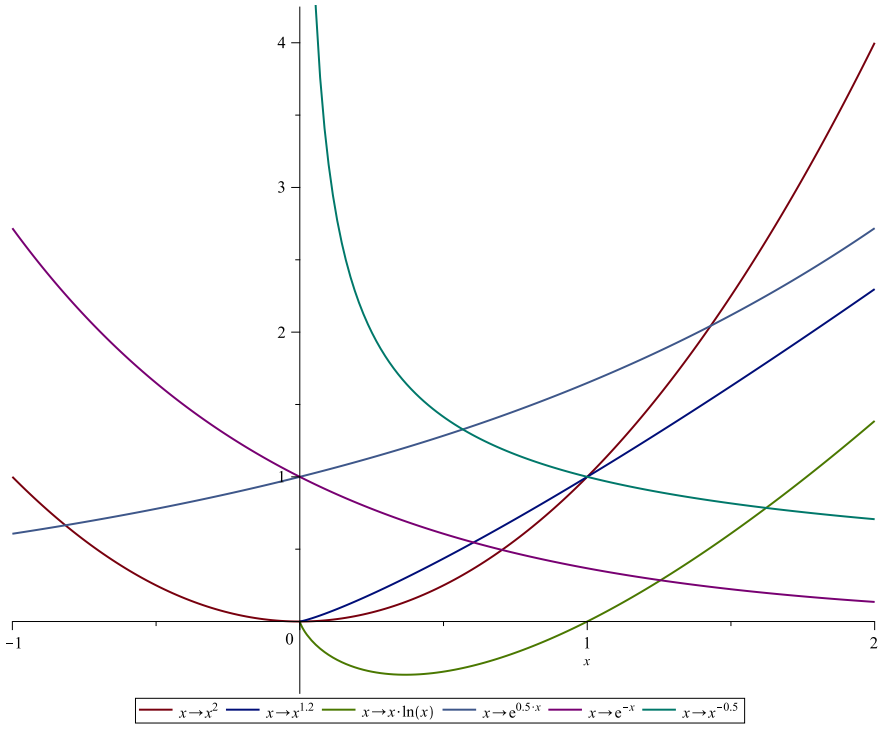


Abbildung 2.3.1: Einige konvexe Funktionen: $\mathbb{R} \ni x \mapsto x^2$, $[0,\infty) \ni x \mapsto x^{1.2}$, $[0,\infty) \ni x \mapsto x \ln x$, $\mathbb{R} \ni x \mapsto e^{x/2}$, $\mathbb{R} \ni x \mapsto e^{-x}$, $(0,\infty) \ni x \mapsto x^{-1/2}$.

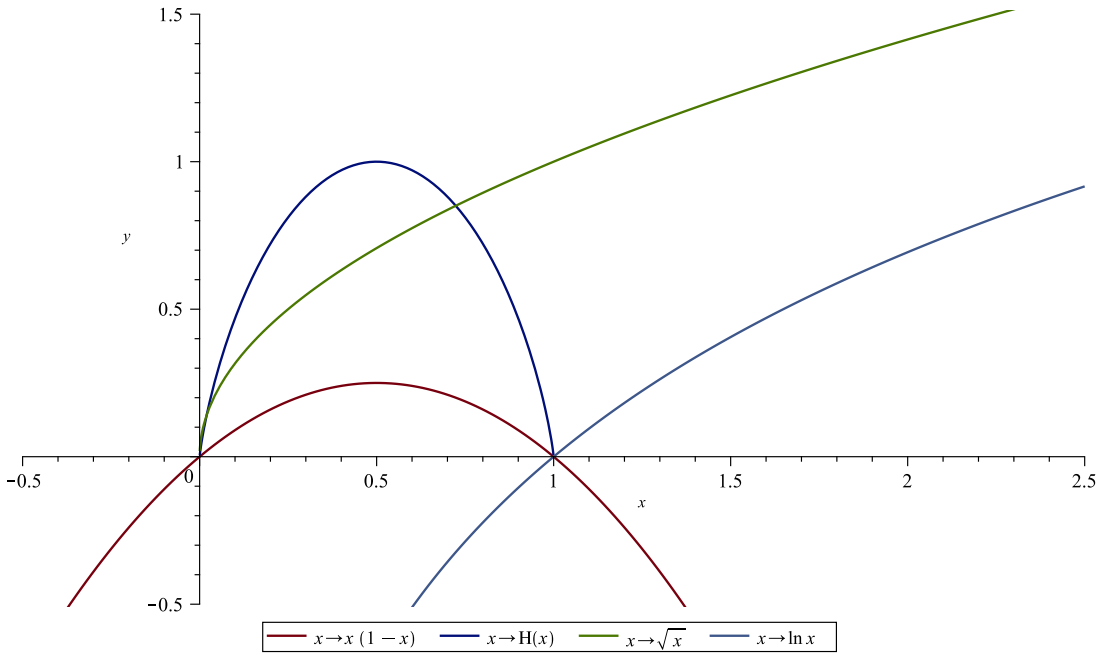


Abbildung 2.3.2: Einige konkave Funktionen: $\mathbb{R} \ni x \mapsto x(1-x)$, $[0,1] \ni x \mapsto \mathrm{H}(x) = -x\log_2 x - (1-x)\log_2(1-x)$ (binäre Entropie), $[0,\infty) \ni x \mapsto \sqrt{x}$, $(0,\infty) \ni x \mapsto \ln x$.

Definition 2.3.7

$D \subseteq \mathbb{R}$ sei ein Intervall. Eine Funktion $f\colon D \to \mathbb{R}$ heißt **konvex**, wenn gilt:

$$f((1-\lambda)x + \lambda y) \leq (1-\lambda)f(x) + \lambda f(y), \text{ für alle } x, y \in D \text{ und } \lambda \in [0,1].$$

Eine Funktion f heißt **konkav**, wenn $-f$ konvex ist.

Anschaulich, geometrisch gesehen ist eine Funktion konvex, wenn in jedem Teilintervall $[x,y]$ des Definitionsbereichs der Graph der Funktion unter der Verbindungsstrecke der Punkte $(x, f(x))$ und $(y, f(y))$ verläuft. – Aus der Schule oder aus der Analysis weiß man, dass für die Konvexität hinreichend ist, dass $f''(x)$ in D (bzw. im Inneren von D) existiert und nicht negativ ist. (Aber Achtung: Die Existenz der zweiten Ableitung ist nicht notwendig. Zum Beispiel ist die Funktion $x \mapsto |x|$ auf $\mathbb{R}$ konvex, aber in $x = 0$ hat sie keine Ableitung.)

Beispiele:

(i) Die Funktion $f\colon x \mapsto x^2$ ist konvex in $\mathbb{R}$. Allgemeiner gilt dies für $x \mapsto x^{2d}$, für jede natürliche Zahl $d > 0$.

(ii) Wenn $\alpha \in \mathbb{R}$, $\alpha \geq 1$, dann ist die Funktion $f_\alpha\colon x \mapsto x^\alpha$ konvex in $[0, \infty)$.

(iii) Wenn $\alpha \in \mathbb{R}$, $0 < \alpha \leq 1$, dann ist die Funktion $f_\alpha\colon x \mapsto x^\alpha$ konkav in $[0, \infty)$.

(iv) Wenn $\alpha \in \mathbb{R}$, $\alpha > 0$, dann ist die Funktion $g_\alpha\colon x \mapsto x^{-\alpha}$ konvex in $(0, \infty)$. (Differenziere zweimal: $g'_\alpha(x) = -\alpha/x^{\alpha+1}$, und dann: $g''_\alpha(x) = \alpha(\alpha+1)/x^{\alpha+2}$. Dies ist immer positiv.)

(v) Die Funktion $h\colon x \mapsto x \ln x$ ist konvex in $[0, \infty)$. (Differenziere zweimal: $h'(x) = \ln x + 1$, und $h''(x) = x^{-1} > 0$, für $x > 0$.)

(vi) Für $t \in \mathbb{R}$ ist die Funktion $k\colon x \mapsto e^{tx}$ konvex in $\mathbb{R}$.

(vii) Die Funktion $\mathrm{H}\colon x \mapsto -x \log_2 x - (1-x)\log_2(1-x)$ (*binäre Entropie* von x) ist konkav in $[0, 1]$.

Siehe Abb. 2.3.1 und Abb. 2.3.2 für Beispiele konvexer und konkaver Funktionen.

Proposition 2.3.8 (*Jensensche Ungleichung, allgemeine Form*)

Es sei X eine reellwertige Zufallsvariable und f eine Funktion mit $X[\Omega] \subseteq D := \mathrm{Def}(f)$. Wenn $\mathbf{E}(X)$ und $\mathbf{E}(f(X))$ definiert sind, dann gilt:

(a) Wenn f konvex ist: $f(\mathbf{E}(X)) \leq \mathbf{E}(f(X))$.

(b) Wenn f konkav ist: $f(\mathbf{E}(X)) \geq \mathbf{E}(f(X))$.

Beispiele: Unter der Voraussetzung, dass jeweils die Erwartungswerte definiert sind, gilt:

(i) $\mathbf{E}(X)^{2d} \leq \mathbf{E}(X^{2d})$.

(ii) Für $\alpha \geq 1$ und $X \geq 0$ gilt: $\mathbf{E}(X)^\alpha \leq \mathbf{E}(X^\alpha)$.

(iii) Für $0 < \alpha \leq 1$ und $X \geq 0$ gilt: $\mathbf{E}(X)^\alpha \geq \mathbf{E}(X^\alpha)$.

(iv) Für $\alpha > 0$ und $X > 0$ gilt $\mathbf{E}(X)^{-\alpha} \leq \mathbf{E}(X^{-\alpha})$.

(v) Für $X \geq 0$ gilt $\mathbf{E}(X)\ln(\mathbf{E}(X)) \leq \mathbf{E}(X \ln X)$.

(vi) Für $t \in \mathbb{R}$ gilt $e^{t\mathbf{E}(X)} \leq \mathbf{E}(e^{tX})$.

Beweis. (Jensensche Ungleichung.) Wir beweisen nur (a). ((b) folgt durch Multiplikation der Ungleichung mit -1.) Setze $x_0 := \mathbf{E}(X)$. Dann ist $x_0 \in D$, denn D ist ein Intervall. Nach einer Grundeigenschaft von konvexen Funktionen, die man in der Analysis beweist, hat der Graph von f im Punkt $(x_0, f(x_0))$ eine „untere Stützgerade“, das ist eine Gerade, die durch den Punkt verläuft und stets unterhalb des Funktionsgraphen bleibt. Das heißt: Es gibt ein $\alpha \in \mathbb{R}$ (die Steigung der Stützgeraden) derart dass

$$f(x_0) + \alpha(x - x_0) \leq f(x) \text{ , für alle } x \in \mathrm{Def}(f) \text{ .}$$

(Wenn f differenzierbar ist, wählt man $\alpha = f'(x_0)$.) Daraus folgt, mit der Linearität und der Monotonie des Erwartungswertes:

$$f(x_0) + \alpha(\mathbf{E}(X) - x_0) = \mathbf{E}(f(x_0) + \alpha(X - x_0)) \leq \mathbf{E}(f(X)).$$

Da $x_0 = \mathbf{E}(X)$ gewählt wurde, folgt die behauptete Ungleichung. □

Die Jensensche Ungleichung ist eine recht allgemeine Konvexitätsaussage. Um ihre Kraft zu demonstrieren, beweisen wir mit ihrer Hilfe die bekannte Ungleichung zwischen dem arithmetischen und dem geometrischen Mittel:

Proposition 2.3.9 (*Arithmetisches versus geometrisches Mittel*)
Für $a_1, \ldots, a_n \geq 0$ gilt:

$$\frac{a_1 + \cdots + a_n}{n} \geq (a_1 \cdots a_n)^{1/n}.$$

Allgemeiner: Wenn zudem $p_1, \ldots, p_n \geq 0$ sind mit $p_1 + \cdots + p_n = 1$, dann gilt:

$$p_1 a_1 + \cdots + p_n a_n \geq a_1^{p_1} \cdots a_n^{p_n}.$$

Beweis. Wir können o. B. d. A. annehmen, dass alle a_i strikt positiv sind. Dann betrachten wir eine Zufallsvariable X, die die Werte $a_1, \ldots, a_n$ mit Wahrscheinlichkeiten

$p_1, \ldots, p_n$ annimmt, sowie die konkave Funktion $f(t) = \ln t$ (mit $\mathrm{Def}(f) = (0, \infty)$). Nach Prop. 2.3.8(b) gilt $f(\mathbf{E}(X)) \geq \mathbf{E}(f(X))$. Wenn man dies ausschreibt und die Logarithmus-Rechenregeln anwendet, ergibt sich

$$\ln(p_1 a_1 + \cdots + p_n a_n) \geq p_1 \ln(a_1) + \cdots + p_n \ln(a_n) = \ln(a_1^{p_1} \cdots a_n^{p_n}).$$

Die Monotonie der Logarithmusfunktion liefert die Behauptung. □

2.4 Bedingte Wahrscheinlichkeiten und bedingte Erwartungswerte

Definition 2.4.1
Ist $B \subseteq \Omega$ ein Ereignis mit $\mathbf{Pr}(B) > 0$, setzen wir

$$\mathbf{Pr}(A \mid B) := \frac{\mathbf{Pr}(A \cap B)}{\mathbf{Pr}(B)},$$

und nennen dies die **bedingte Wahrscheinlichkeit von** A (unter der Bedingung B), für beliebige Ereignisse $A \subseteq \Omega$.

Eine Routinerechnung zeigt, dass Ω mit der durch $\mathbf{Pr}(\,\cdot \mid B)$ definierten Verteilung ebenfalls ein Wahrscheinlichkeitsraum ist. (Elementarwahrscheinlichkeiten: $p_\omega^B = p_\omega / \mathbf{Pr}(B)$ für $\omega \in B$ und $p_\omega^B = 0$ für $\omega \notin B$.) Auch in diesem Wahrscheinlichkeitsraum lassen sich Erwartungswerte von Zufallsvariablen X bilden (geschrieben $E(X \mid B)$). Man sieht leicht:

$$\mathbf{Pr}(B \mid B) = \mathbf{Pr}(\Omega \mid B) = 1; \quad \mathbf{E}(X \mid B) = \frac{1}{\mathbf{Pr}(B)} \cdot \sum_{\omega \in B} p_\omega X(\omega).$$

Fakt 2.4.2 (*Basisformel für bedingte Wahrscheinlichkeiten*)

$$\mathbf{Pr}(A \cap B) = \mathbf{Pr}(A \mid B) \cdot \mathbf{Pr}(B), \text{ für } A, B \subseteq \Omega \text{ mit } \mathbf{Pr}(B) > 0.$$

Im Fall $\mathbf{Pr}(B) = 0$ ist $\mathbf{Pr}(A \mid B)$ nicht definiert. Solange man bedingte Wahrscheinlichkeiten nur über die Basisformel aus Fakt 2.4.2 benutzt, kann man aber bei $\mathbf{Pr}(B) = 0$ so tun, als ob $\mathbf{Pr}(A \mid B)$ einen beliebigen Wert hätte. Die Formel kann man auf den Durchschnitt mehrerer Ereignisse verallgemeinern:

$$\mathbf{Pr}(A_1 \cap \cdots \cap A_n) = \mathbf{Pr}(A_1)\mathbf{Pr}(A_2 \mid A_1)\mathbf{Pr}(A_3 \mid A_1 \cap A_2) \cdots \mathbf{Pr}(A_n \mid A_1 \cap \cdots \cap A_{n-1}).$$

Diese Formel wurde in der Analyse des MinCut-Algorithmus in Abschnitt 1.1 benutzt.

Im Kontext von bedingten Wahrscheinlichkeiten gibt es recht einfache, aber grundlegende Formeln, die auch in der Analyse von Algorithmen benutzt werden. Dabei geht es darum, ein Ereignis oder eine Zufallsvariable in disjunkten Teilen des Wahrscheinlichkeitsraumes zu analysieren und dann die Ergebnisse zusammenzufassen. Das erste Ergebnis heißt etwas hochtrabend *Satz von der totalen Wahrscheinlichkeit*, das zweite hat keinen Namen, tut aber in etwa dasselbe für Erwartungswerte.

Fakt 2.4.3 (*Satz von der totalen Wahrscheinlichkeit*)

Für eine (endliche oder unendliche) Indexmenge seien B_i, $i \in I$, Ereignisse, die eine disjunkte Zerlegung des W-Raumes bilden, d. h. $B_i \cap B_j = \emptyset$ für $i, j \in I$ mit $i \neq j$ und $\bigcup_{i \in I} B_i = \Omega$. Dann gilt für jedes Ereignis A und jede Zufallsvariable X, deren Erwartungswert existiert:

$$\mathbf{Pr}(A) = \bigcup_{i \in I} \mathbf{Pr}(A \mid B_i)\mathbf{Pr}(B_i) \text{ und}$$
$$\mathbf{E}(X) = \sum_{i \in I} \mathbf{E}(X \mid B_i)\mathbf{Pr}(B_i).$$

Die Formel $\mathbf{E}(X) = \sum_{\alpha \in X[\Omega]} \alpha \cdot \mathbf{Pr}(X = \alpha)$ aus Definition 2.2.7 ist ein Spezialfall der zweiten Gleichung: $B_\alpha = \{X = \alpha\}$ und $\mathbf{E}(X \mid X = \alpha) = \alpha$. – In der Analyse von Algorithmen findet man oft das folgende Argumentationsmuster: Man legt eine Bedingung für das Ereignis A bzw. die Zufallsvariable X fest; damit wird eines der Ereignisse B_i ausgewählt. Unter dieser Bedingung beweist man $\mathbf{Pr}(A \mid B_i) \leq a$ bzw. $\mathbf{E}(X \mid B_i) \leq b$, für eine „obere Schranke“ a bzw. b. Fakt 2.4.3 liefert dann sofort $\mathbf{Pr}(A) \leq a$ bzw. $\mathbf{E}(X) \leq b$ (weil $\sum_{i \in I} \mathbf{Pr}(B_i) = 1$ gilt).

2.5 Unabhängigkeit von Ereignissen und von Zufallsvariablen

Definition 2.5.1

(a) Ereignisse A und B heißen **unabhängig**, falls $\mathbf{Pr}(A \cap B) = \mathbf{Pr}(A)\mathbf{Pr}(B)$.

(b) Ereignisse $A_1, \ldots, A_n$ heißen **unabhängig**, falls

$$\mathbf{Pr}\left(\bigcap_{i\in I} A_i \cap \bigcap_{i\in J} (\Omega - A_i)\right) = \prod_{i\in I} \mathbf{Pr}(A_i) \cdot \prod_{i\in J} (1 - \mathbf{Pr}(A_i)),$$

für beliebige *disjunkte* Mengen $I, J \subseteq \{1, \ldots, n\}$.

(c) Eine Familie $(A_i)_{i\in K}$ von Ereignissen heißt **unabhängig**, wenn für jede endliche Teilmenge $J \subseteq K$ die Familie $(A_i)_{i\in J}$ im Sinn von (b) unabhängig ist.

Bemerkung: Wenn $\mathbf{Pr}(A) = 0$ oder $\mathbf{Pr}(A) = 1$, dann sind A und B auf jeden Fall unabhängig. – Zwei Ereignisse A und B mit $\mathbf{Pr}(B) > 0$ sind unabhängig genau dann wenn $\mathbf{Pr}(A \mid B) = \mathbf{Pr}(A)$ gilt. (Denn nach Definition 2.5.1(a) ist $\mathbf{Pr}(A \mid B) \cdot \mathbf{Pr}(B) = \mathbf{Pr}(A \cap B)$.) Das ergibt die übliche intuitive Erklärung der Unabhängigkeit, nämlich dass sich die Wahrscheinlichkeit von A durch das „Wissen“, dass B eingetreten ist, nicht ändert.

In vielen Büchern findet man auch eine (auf den ersten Blick) andere Form von Definition 2.5.1(b): Man nennt $A_1, \ldots, A_n$ unabhängig, falls

$$\mathbf{Pr}\left(\bigcap_{i\in I} A_i\right) = \prod_{i\in I} \mathbf{Pr}\left(A_i\right), \tag{2.5.4}$$

für beliebige Teilmengen I von $\{1, 2, \ldots, n\}$. Diese Definition und Definition 2.5.1(b) sind jedoch äquivalent. Unsere Definition hat den Vorteil, dass man sofort Aussagen machen kann, bei denen Komplementereignisse $\overline{A_i} = \Omega - A_i$ vorkommen.

Beispiel 2.5.2
(a) In Bsp. 2.1.2(h) (Hashing) sind die Ereignisse $\{v_1 = v_1^0\}, \ldots, \{v_n = v_n^0\}$ unabhängig, für beliebige feste Werte $v_1^0, \ldots, v_n^0 \in \{0, \ldots, m-1\}$.

(b) In Bsp. 2.1.2(h) (Hashing) sind die Ereignisse $\{v_1 \neq 0\}, \ldots, \{v_n \neq 0\}$ unabhängig.

Definition 2.5.3

(a) Zufallsfunktionen $X_i\colon \Omega \to R_i$, $1 \le i \le n$, heißen **unabhängig**, wenn für beliebige $R_i' \subseteq R_i$ die Ereignisse $\{X_1 \in R_1'\}, \ldots, \{X_n \in R_n'\}$ unabhängig sind. Dies gilt genau dann, wenn

$$\mathbf{Pr}(X_i \in R_i' \text{ für } 1 \le i \le n) = \prod_{1 \le i \le n} \mathbf{Pr}(X_i \in R_i'),$$

für beliebige $R_i' \subseteq R_i$.

(b) Eine Familie $(X_i)_{i \in K}$ von Zufallsfunktionen ist **unabhängig**, wenn für jede endliche Teilmenge $J \subseteq K$ die Familie $(X_i)_{i \in J}$ im Sinn von (a) unabhängig ist.

Die folgende Charakterisierung der Unabhängigkeit für diskrete Wahrscheinlichkeitsräume ist durch die naheliegenden Summationen zu beweisen.

Bemerkung 2.5.4
Zufallsfunktionen $X_i\colon \Omega \to R_i$, $1 \le i \le n$, sind unabhängig genau dann wenn für beliebige $r_i \in R_i$ gilt

$$\mathbf{Pr}(X_i = r_i \text{ für } 1 \le i \le n) = \prod_{1 \le i \le n} \mathbf{Pr}(X_i = r_i).$$

Fakt 2.5.5
Sind $X_1, \ldots, X_n$ unabhängig und sind $g_i\colon R_i \to S_i$ beliebig, $1 \le i \le n$, dann sind die Zufallsfunktionen $g_1 \circ X_1, \ldots, g_n \circ X_n$ unabhängig.

Beispiel 2.5.6
Sind (Ω_i, p^i), $1 \le i \le n$, W-Räume, so wird durch (Ω, p) mit $\Omega := \Omega_1 \times \cdots \times \Omega_n$, $p := p^1 \times \cdots \times p^n$, wo $p(\omega) = p^1(\omega_1) \cdot \ldots \cdot p^n(\omega_n)$, für $\omega = (\omega_1, \ldots, \omega_n) \in \Omega$, ein neuer W-Raum (der „*Produktraum*“) definiert. Auf (Ω, p) sind die n Projektionsfunktionen $X_i\colon (\omega_1, \ldots, \omega_n) \mapsto \omega_i \in \Omega_i$ unabhängig; nach Fakt 2.5.5 ist also jede Folge $Y_1, \ldots, Y_n$ von Zufallsfunktionen, wo $Y_i = g_i \circ X_i$ (d. h. Y_i hängt *nur* von der i-ten Komponente ω_i ab), unabhängig. (*Beispiel*: Der Wahrscheinlichkeitsraum in Beispiel 2.1.2(h) ist ein Produktraum.)

Fakt 2.5.7

„Bei Unabhängigkeit multiplizieren sich Erwartungswerte, Varianzen addieren sich.“[4]

(a) Sind $X_1, \dots, X_n$ *unabhängige* Zufallsvariable, so gilt

$$\mathbf{E}(X_1 \cdot \ldots \cdot X_n) = \prod_{1 \le i \le n} \mathbf{E}(X_i).$$

(b) Sind $X_1, \dots, X_n$ *unabhängige* Zufallsvariable, so gilt

$$\mathbf{Var}(X_1 + \cdots + X_n) = \sum_{1 \le i \le n} \mathbf{Var}(X_i).$$

Dies gilt sogar, wenn nur X_i und X_j unabhängig sind für $i \neq j$ (*paarweise Unabhängigkeit*).

Beweis. (a) Wir beweisen die Aussage für zwei Zufallsvariable X und Y. Die Verallgemeinerung auf n Zufallsvariable ergibt sich durch vollständige Induktion.

$$\begin{aligned}
\mathbf{E}(X)\mathbf{E}(Y) &= \left(\sum_{\alpha \in X[\Omega]} \alpha \cdot \mathbf{Pr}(X = \alpha) \right) \left(\sum_{\beta \in Y[\Omega]} \beta \cdot \mathbf{Pr}(Y = \beta) \right) \\
&= \sum_{\substack{\alpha \in X[\Omega] \\ \beta \in Y[\Omega]}} \alpha\beta \cdot \mathbf{Pr}(X = \alpha)\mathbf{Pr}(Y = \beta) \\
&= \sum_{\delta \in XY[\Omega]} \left(\sum_{\substack{\alpha \in X[\Omega] \\ \beta \in Y[\Omega] \\ \alpha\beta = \delta}} \alpha\beta \cdot \mathbf{Pr}(X = \alpha)\mathbf{Pr}(Y = \beta) \right) \\
&\overset{(*)}{=} \sum_{\delta \in XY[\Omega]} \delta \cdot \left(\sum_{\substack{\alpha \in X[\Omega] \\ \beta \in Y[\Omega] \\ \alpha\beta = \delta}} \mathbf{Pr}(X = \alpha \wedge Y = \beta) \right)
\end{aligned}$$

[4] Additivität von Erwartungswerten gilt immer, siehe Fakt 2.2.10(c).

$$\overset{(**)}{=} \sum_{\delta \in XY[\Omega]} \delta \cdot \mathbf{Pr}(XY = \delta) = \mathbf{E}(XY).$$

Für $(*)$ wird die Unabhängigkeit benutzt, für $(**)$ die disjunkte Zerlegung

$$\{XY = \delta\} = \bigcup_{\substack{\alpha \in X[\Omega] \\ \beta \in Y[\Omega] \\ \alpha\beta = \delta}} \{X = \alpha \wedge Y = \beta\}.$$

(b) Definiere $X_i' := X_i - \mathbf{E}(X_i)$, für $1 \leq i \leq n$, und $X' = X_1' + \cdots + X_n' = X - \mathbf{E}(X)$. Dann gilt $\mathbf{E}(X_i') = 0$ und $\mathbf{Var}(X_i') = \mathbf{Var}(X_i)$, für $1 \leq i \leq n$, sowie $\mathbf{E}(X') = 0$ und $\mathbf{Var}(X') = \mathbf{Var}(X)$. Das heißt, dass wir o. B. d. A. annehmen können, dass $\mathbf{E}(X_i) = 0$ und $\mathbf{Var}(X_i) = \mathbf{E}(X_i^2)$ und $\mathbf{Var}(X) = \mathbf{E}(X^2)$ gelten. Wir haben dann:

$$\begin{aligned}
\mathbf{Var}(X) &= \mathbf{E}\Big(\big(\sum_{1 \leq i \leq n} X_i \big)^2 \Big) = \mathbf{E}\Big(\sum_{1 \leq i,j \leq n} X_i X_j \Big) = \sum_{1 \leq i,j \leq n} \mathbf{E}(X_i X_j) \\
&= \sum_{1 \leq i \leq n} \mathbf{E}(X_i^2) + \sum_{1 \leq i \neq j \leq n} \mathbf{E}\big(X_i X_j\big) \\
&\overset{(\dagger)}{=} \sum_{1 \leq i \leq n} \mathbf{E}(X_i^2) + \sum_{1 \leq i \neq j \leq n} \underbrace{\mathbf{E}(X_i)}_{=0} \underbrace{\mathbf{E}(X_j)}_{=0} = \sum_{1 \leq i \leq n} \mathbf{Var}(X_i).
\end{aligned}$$

Für $(\dagger)$ wird die Unabhängigkeit von X_i und X_j benutzt (aber auch nicht mehr). □

Beachte noch: Wenn X_i 0-1-wertig ist, ist $X_i^2 = X_i$, also $\mathbf{E}(X_i^2) = \mathbf{E}(X_i)$. Damit erhält man für $X = X_1 + \cdots + X_n$ die folgende nützliche Ungleichung, wenn die X_i paarweise unabhängig sind:

$$\mathbf{Var}(X) = \sum_{1 \leq i \leq n} \mathbf{Var}(X_i) = \sum_{1 \leq i \leq n} (\mathbf{E}(X_i^2) - \mathbf{E}(X_i)^2) \leq \sum_{1 \leq i \leq n} \mathbf{E}(X_i) = \mathbf{E}(X).$$

Gleichheit gilt nur, wenn alle X_i gleich 0 sind.

2.5.1 Zwei Verteilungen

Die Binomialverteilung(en). Die anschauliche Situation, die zu einer Binomialverteilung führt, ist folgende: Wir führen n Experimente durch, wobei bei jedem mit Wahrscheinlichkeit p „Erfolg“ und mit Wahrscheinlichkeit $q = 1 - p$ „Misserfolg“ auftritt. Die Experimente sind unabhängig. (Veranschaulicht wird dies mit dem

n-maligen Werfen einer Münze, die mit Wahrscheinlichkeit p „Kopf“ und mit Wahrscheinlichkeit $q = 1 - p$ „Zahl“ zeigt.) Uns interessiert die Anzahl der „Erfolge“ – eine Zufallsvariable mit Werten in $\{0, \ldots, n\}$. Die Verteilung dieser Zufallsvariablen heißt die *Binomialverteilung zu n und p*, kurz $B(n, p)$. (Es gibt also unendlich viele Binomialverteilungen.)

Technisch betrachtet man den Wahrscheinlichkeitsraum $(\Omega, p^{n,p})$ mit $\Omega = \{0, 1\}^n$, wobei die Idee der Unabhängigkeit durch die Produktverteilung realisiert wird:

$$p^{n,p}((a_1, \ldots, a_n)) = \prod_{1 \leq i \leq n} p^{a_i}(1-p)^{1-a_i}, \text{ für } (a_1, \ldots, a_n) \in \{0, 1\}^n.$$

Die 1-Positionen werden mit Wahrscheinlichkeit p realisiert, die 0-Positionen mit Wahrscheinlichkeit $q = 1 - p$. Die Projektionen $X_i((a_1, \ldots, a_n)) = a_i$ sind dann unabhängige 0-1-wertige Zufallsvariablen mit $\mathbf{Pr}(X_i = 1) = p$. Die Anzahl der „Erfolge“ wird durch die Summe $X = X_1 + \cdots + X_n$ modelliert. Die Verteilung von X auf $\mathbb{N}$ ist dann $B(n, p)$.

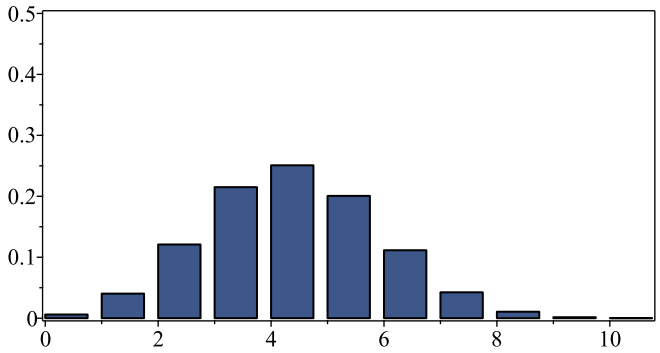


Abbildung 2.5.3: Binomialverteilung $B(10, \frac{2}{5})$. Höhe des Balkens rechts von k ist die Wahrscheinlichkeit $\binom{n}{k}(\frac{2}{5})^k(\frac{3}{5})^{10-k}$ für $k = 0, \ldots, 10$.

Was ist $\mathbf{Pr}(X = k)$, für ein $k \in \mathbb{N}$? Werte $k > n$ kann X nicht annehmen, also ist $\mathbf{Pr}(X = k) = 0$ für $k > n$. Für $k \in \{0, \ldots, n\}$ gilt: $X((a_1, \ldots, a_n)) = k$ genau dann wenn es genau k Positionen i mit $a_i = 1$ gibt. Es gibt genau $\binom{n}{k}$ viele Folgen $(a_1, \ldots, a_n)$ mit genau k Einsen; jede dieser Folgen hat Wahrscheinlichkeit $p^k(1-p)^{n-k}$. Daher gilt

$$\mathbf{Pr}(X = k) = \binom{n}{k} p^k (1-p)^{n-k}, \text{ für } 0 \leq k \leq n.$$

In Abb. 2.5.3 ist die Wahrscheinlichkeitsfunktion für die Binomialverteilung $B(10, 0.4)$ bildlich dargestellt. – Erwartungswert und Varianz einer binomialverteilten Zufallsvariablen sind nicht schwer zu berechnen.

Fakt 2.5.8
Sei X eine Zufallsvariable, die $B(n,p)$-verteilt ist, und sei $q = 1 - p$. Dann gilt:

$$\mathbf{E}(X) = np \quad \text{und} \quad \mathbf{Var}(X) = np(1-p) = npq.$$

Beweis. Nach der Konstruktion der Binomialverteilung können wir $X = X_1 + \cdots + X_n$ schreiben, für unabhängige 0-1-wertige Zufallsvariablen $X_1, \ldots, X_n$ mit Erwartungswert p. Es folgt $\mathbf{E}(X) = \sum_{1 \le i \le n} \mathbf{E}(X_i) = \sum_{1 \le i \le n} p = np$. Für die Varianz berechnen wir zunächst (beachte $X_i^2 = X_i$, weil X_i 0-1-wertig ist):

$$\mathbf{Var}(X_i) = \mathbf{E}(X_i^2) - \mathbf{E}(X_i)^2 = \mathbf{E}(X_i) - \mathbf{E}(X_i)^2 = p - p^2 = pq.$$

Da die X_i unabhängig sind, addieren sich nach Fakt 2.5.7(b) die Varianzen und wir erhalten

$$\mathbf{Var}(X) = \sum_{1 \le i \le n} \mathbf{Var}(X_i) = \sum_{1 \le i \le n} pq = npq.$$

□

Bemerkung 2.5.9 Schwaches Gesetz der großen Zahlen (Bernoulli)
Wir werfen wiederholt und unabhängig eine Münze, bei der mit Wahrscheinlichkeit p „Kopf“ (entspricht „1“) und mit Wahrscheinlichkeit $q = 1 - p$ „Zahl“ (entspricht „0“) auftritt. Wie wir gerade gesehen haben, ist die Anzahl der „Kopf“-Ergebnisse eine $B(n,p)$-verteilte Zufallsvariable, wenn n die Anzahl der Würfe ist. Wir stellen uns hier auf den Standpunkt, dass wir p nicht kennen und *schätzen* wollen. Es ist naheliegend, die relative Häufigkeit des Ergebnisses „Kopf“ als Schätzwert für p zu nehmen. Das schwache Gesetz der großen Zahlen besagt, dass dieser Wert nur mit kleiner Wahrscheinlichkeit weit von p entfernt ist – mit umso kleinerer Wahrscheinlichkeit, je größer die Anzahl der Münzwürfe ist. Technisch sieht das so aus: Sei $n \in \mathbb{N}$ beliebig. Für $1 \le i \le n$ sei X_i eine 0-1-wertige Zufallsvariable mit $\mathbf{Pr}(X_i = 1) = p$, und diese Zufallsvariablen seien unabhängig.[5] Der Schätzwert für p ist die Zufallsvariable

$$Y_n := \frac{X_1 + \cdots + X_n}{n}.$$

[5] Unsere abzählbar unendlichen Wahrscheinlichkeitsräume lassen es nicht zu, eine unendliche unabhängige Folge $X_1, X_2, \ldots$ zu betrachten. Daher nehmen wir einen W-Raum für jedes n.

Das *Schwache Gesetz der großen Zahlen* (Jakob I Bernoulli, 1655–1705) besagt dann, dass für jedes $\varepsilon > 0$ gilt: $\lim_{n\to\infty} \mathbf{Pr}(|Y_n - p| \geq \varepsilon) = 0$. Genauer gilt:

$$\mathbf{Pr}(|Y_n - p| \geq \varepsilon) \leq \frac{1}{4n\varepsilon^2}. \tag{2.5.5}$$

Der Beweis beruht auf der Chebychev-Ungleichung. Wir haben $\mathbf{E}(X_i) = p$ und $\mathbf{Var}(X_i) = pq = p(1-p) \leq \frac{1}{4}$. Nach Fakt 2.5.7(b) folgt[6] $\mathbf{Var}(Y_n) = (1/n^2)\mathbf{Var}(X_1 + \cdots + X_n) = (1/n^2) \cdot npq = pq/n \leq 1/(4n)$. Nun können wir die Chebychev-Ungleichung anwenden und erhalten:

$$\mathbf{Pr}(|Y_n - p| \geq \varepsilon) \leq \frac{\mathbf{Var}(Y_n)}{\varepsilon^2} \leq \frac{1}{4n\varepsilon^2},$$

wie behauptet.

Wir bemerken, dass das schwache Gesetz der großen Zahlen schon bei nur paarweiser Unabhängigkeit gilt (und in vielen anderen Situationen mit schwächeren Voraussetzungen).

Anwendung: Wir nehmen $p = \frac{1}{2}$ an, also einen fairen Münzwurf. Wieviele Münzwürfe genügen, damit der Schätzwert Y_n mit genügend großer Wahrscheinlichkeit im Intervall [0.45..0.55] liegt? In diesem Fall ist $\varepsilon = \frac{1}{20}$ zu wählen. Dann ist $1/(4\varepsilon^2) = 100$. Wir erhalten $\mathbf{Pr}(\left|Y_n - \frac{1}{2}\right| \geq \frac{1}{20}) \leq \frac{1}{4n\varepsilon^2} = \frac{100}{n}$. Mit 2000 Münzwürfen ist die Wahrscheinlichkeit, dass die Anzahl der Ergebnisse „Kopf“ nicht zwischen 900 und 1100 liegt, höchstens 0.05, mit 10000 Münzwürfen ist die Wahrscheinlichkeit, dass man nicht zwischen 4500 und 5500 mal „Kopf“ erhält, höchstens 0.01.

Die geometrische(n) Verteilung(en). Wieder führen wir wiederholt und unabhängig identische Bernoulli-Experimente durch, also Experimente, bei denen mit Wahrscheinlichkeit p „Erfolg“ und mit Wahrscheinlichkeit $q = 1 - p$ „Misserfolg“ auftritt. Nur ist hier die Anzahl der Experimente unbeschränkt. Wir halten an, sobald „Erfolg“ eintritt. Die Zufallsvariable, die uns interessiert, ist die Anzahl der Versuche. In Beispielen 2.1.2(d) und 2.2.5(b) hatten wir die Situation „Würfeln, bis die erste 6 erscheint“ betrachtet. Wir können hier einen ähnlichen W-Raum wie in Beispiel 2.2.5(b) benutzen:[7] $\Omega = \{(a_1, \ldots, a_i) \mid i \geq 1, a_1 = \cdots = a_{i-1} = 0 \text{ und } a_i = 1\}$

[6] Wir benutzen hier eine weitere einfache, aber nützliche Ungleichung: Für $0 \leq t \leq 1$ gilt $t(1-t) \leq \frac{1}{4}$. Das liegt daran, dass die quadratische Funktion $t \mapsto t(1-t)$ bei $t_0 = \frac{1}{2}$ ihr Maximum annimmt.

[7] Natürlich entspricht dieser Raum genau der Menge $\mathbb{N}^+$ mit der Verteilung analog zu Beispiel 2.1.2(d).

mit der Wahrscheinlichkeitsfunktion $p((a_1,\dots,a_i)) = (1-p)^{i-1}p$. Dass die Versuche unabhängig sind, wird durch die Multiplikation der Einzelwahrscheinlichkeiten modelliert. Die Zufallsvariable X bildet $(a_1,\dots,a_i)$ auf i ab. Die Verteilung von X ist eine W-Verteilung auf $\mathbb{N}$, sie heißt *geometrische Verteilung zu Parameter* p und ist gegeben durch $\mathbf{Pr}(X=0)=0$, $\mathbf{Pr}(X=k)=q^{k-1}p$ für $k\geq 1$. Eine Illustration einer geometrischen Verteilung ist in Abb. 2.5.4 angegeben. Wir berechnen den

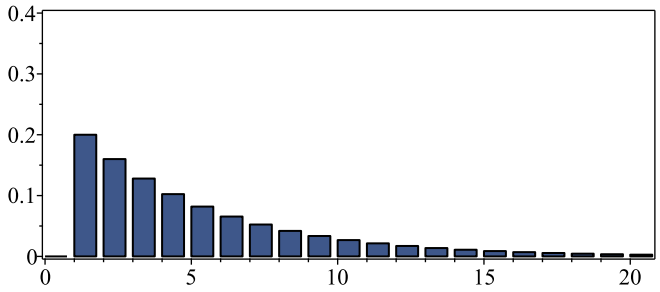


Abbildung 2.5.4: Geometrische Verteilung zum Parameter $p=0.2$, Wahrscheinlichkeiten für $k=0,\dots,20$. Höhe des Balkens rechts von k ist $\mathbf{Pr}(X=k)=0.8^{k-1}\cdot 0.2$.

Erwartungswert, mit Hilfe von Fakt 2.2.9. Zunächst rechnen wir (mit $p=1-q$):

$$\mathbf{Pr}(X\geq k)=\sum_{i\geq k} q^{i-1}p = pq^{k-1}\sum_{i\geq k} q^{i-k} = pq^{k-1}\cdot\frac{1}{1-q}=q^{k-1}.$$

(Dies entspricht der Tatsache, dass die Anzahl der Versuche genau dann $\geq k$ ist, wenn die ersten $k-1$ Versuche alle mit „Misserfolg“ enden, und der Intuition, dass die Wahrscheinlichkeit hierfür q^{k-1} ist.) Anwendung von Fakt 2.2.9 liefert:

$$\mathbf{E}(X)=\sum_{k\geq 1}\mathbf{Pr}(X\geq k)=\sum_{k\geq 1} q^{k-1}=\frac{1}{1-q}=\frac{1}{p}. \tag{2.5.6}$$

Wir wollen noch die Varianz von X bestimmen. Wir stellen dazu zunächst fest, dass $\mathbf{E}(X^2)=\sum_{k\geq 1}k^2q^{k-1}p<\infty$ gilt, sobald $q<1$, d. h. sobald die Erfolgswahrscheinlichkeit p positiv ist. Nach (2.3.2) gilt $\mathbf{Var}(X)=\mathbf{E}(X^2)-\mathbf{E}(X)^2$. Da wir den Erwartungswert kennen, können wir uns auf $\mathbf{E}(X^2)$ konzentrieren.

Wir benötigen eine kleine Vorüberlegung zur Differentiation von Potenzreihen. Betrachte $f(x):=\frac{1}{1-x}=\sum_{k\geq 0}x^k$, absolut konvergent für $|x|<1$. Aus der Analysis

weiß man, dass man gliedweise differenzieren darf. Das ergibt $f'(x) = \sum_{k\geq 1} kx^{k-1}$ und $f''(x) = \sum_{k\geq 2} k(k-1)x^{k-2}$, für x im Konvergenzbereich. Damit:

$$\mathbf{E}(X^2) = \sum_{k\geq 1} k^2 q^{k-1} p = p \sum_{k\geq 2} k(k-1)q^{k-1} + p \sum_{k\geq 1} kq^{k-1} = pq \cdot f''(q) + p \cdot f'(q).$$

Mit den geschlossenen Formeln

$$f'(x) = \frac{1}{(1-x)^2} \quad \text{und} \quad f''(x) = \frac{2}{(1-x)^3}, \quad \text{für } |x| < 1,$$

für die Ableitungen erhalten wir

$$\mathbf{E}(X^2) = pq \cdot \frac{2}{(1-q)^3} + p \cdot \frac{1}{(1-q)^2} = \frac{2pq}{p^3} + \frac{p}{p^2} = \frac{2(1-p)+p}{p^2} = \frac{2-p}{p^2}.$$

Damit ergibt sich die Varianz für die geometrische Verteilung mit Parameter p:

$$\mathbf{Var}(X) = \mathbf{E}(X^2) - \mathbf{E}(X)^2 = \frac{2-p}{p^2} - \left(\frac{1}{p}\right)^2 = \frac{1-p}{p^2}. \tag{2.5.7}$$

2.6 Die Hoeffding-Ungleichung

Die Hoeffding-Ungleichung (oder Chernoff-Hoeffding-Ungleichung) gibt, wie die Chebychev-Ungleichung, eine Schranke für die Wahrscheinlichkeit an, dass eine Zufallsvariable weit von ihrem Erwartungswert entfernt liegt. Sie liefert jedoch ungleich schärfere Abschätzungen als die Chebychev-Ungleichung, wenn X eine Summe von unabhängigen Zufallsvariablen ist, die alle jeweils nur einen kleinen Wertebereich überstreichen. Standardbeispiel für diese Situation ist die Binomialverteilung als Verteilung einer Summe von n unabhängigen 0-1-wertigen Zufallsvariablen.

Satz 2.6.1 (*Hoeffding-Ungleichung*)

$X_1, \ldots, X_n$ seien unabhängige Zufallsvariable mit Werten im Intervall $[0, 1]$. Definiere

$$\begin{aligned} X &:= X_1 + \cdots + X_n\,; \\ m &:= \mathbf{E}(X)\,. \end{aligned}$$

Dann gilt:[8]

$$\mathbf{Pr}(X \geq r) \leq \left(\frac{m}{r}\right)^r \left(\frac{n-m}{n-r}\right)^{n-r}, \text{ für } m \leq r \leq n; \tag{2.6.8}$$

$$\mathbf{Pr}(X \leq s) \leq \left(\frac{m}{s}\right)^s \left(\frac{n-m}{n-s}\right)^{n-s}, \text{ für } 0 \leq s \leq m. \tag{2.6.9}$$

Man sieht sofort, dass das am Anfang von Abschnitt 2.5.1 diskutierte wiederholte Bernoulli-Experiment mit n Münzwürfen, wo jeder mit Wahrscheinlichkeit p das Ergebnis „Kopf“ liefert, direkt zur Situation des Satzes führt. Die Zufallsvariable X_i ist das Ergebnis des i-ten Münzwurfs, und X zählt, wie oft „Kopf“ aufgetreten ist. In diesem Fall ist $m = \mathbf{E}(X) = np$.

Die Hoeffding-Ungleichung gehört zu der Familie der „tail inequalities“, das sind Ungleichungen, die obere Schranken für die Wahrscheinlichkeit liefern, dass Zufallsvariable Werte weit weg von ihrem Erwartungwert annehmen. Wir werden weiter unten sehen, dass die Hoeffding-Schranke relativ kräftig ist, wenn $m = \mathbf{E}(X)$ nicht zu klein ist. Grob gesprochen: Summen von *vielen* (auf $[0, 1]$) *beschränkten unabhängigen* Zufallsvariablen sind eng um ihren Erwartungswert konzentriert. Der Anwendungsbereich ist viel weiter als nur wiederholte Bernoulli-Experimente: Über die einzelnen X_i wird nichts weiter angenommen, als dass sie in $[0, 1]$ eingeschlossen sind. Insbesondere können sie auch ganz unterschiedliche Verteilungen haben und irgendwelche reellen Werte annehmen.

Beweis. (Von Satz 2.6.1.) Wir setzen $p := m/n$ und beweisen zunächst (2.6.8). Der Fall $r = m$ ist trivial, weil auf der linken Seite eine Wahrscheinlichkeit steht, auf der rechten Seite 1. Es sei jetzt also $m < r \leq n$ beliebig, aber fest. – Da der Beweis etwas länger ist, wird er in Schritte untergliedert.

[8] Lesehilfe: Wir benutzen die Konvention, dass $\alpha^0 = 1$ für alle $\alpha \geq 0$ gilt; Faktoren $(\alpha/\beta)^\beta$ haben also für $\alpha > \beta = 0$ den Wert 1.

1. Schritt: Chernoff-Schranke. – Betrachte eine beliebige reelle Zahl $u > 1$. Wir wenden Prop. 2.3.5, mit der monoton wachsenden Funktion $t \mapsto u^t$ an und erhalten

$$\mathbf{Pr}(X \geq r) \leq \frac{\mathbf{E}\big(u^X\big)}{u^r} = u^{-r} \cdot \mathbf{E}\big(u^{X_1+\cdots+X_n}\big) = u^{-r} \cdot \mathbf{E}\Big(\prod_{1\leq i\leq n} u^{X_i}\Big). \tag{2.6.10}$$

2. Schritt: Multipliziere Erwartungswerte. – Weil $X_1, \ldots, X_n$ unabhängig sind, sind auch $u^{X_1}, \ldots, u^{X_n}$ unabhängig (Fakt 2.5.5). Daher (Fakt 2.5.7(a)) multiplizieren sich in (2.6.10) die Erwartungswerte, und wir erhalten:

$$\mathbf{Pr}(X \geq r) \leq u^{-r} \cdot \prod_{1\leq i\leq n} \mathbf{E}(u^{X_i}). \tag{2.6.11}$$

3. Schritt: Abschätzung der einzelnen Erwartungswerte $\mathbf{E}(u^{X_i})$*, per Konvexität.* –

Lemma 2.6.2
Sei $u > 1$ beliebig. Dann gilt:

(i) $u^x \leq 1 + (u-1)x$, für $0 \leq x \leq 1$.

(ii) Für jede Zufallsvariable Y mit $0 \leq Y \leq 1$ gilt $\mathbf{E}(u^Y) \leq 1 + (u-1)\mathbf{E}(Y)$.

Beweis. (i) Da $u > 1$, ist die Funktion $x \mapsto u^x$ konvex. Daher verläuft der Graph dieser Funktion in $[0,1]$ unterhalb der Strecke durch $(0, u^0) = (0,1)$ und $(1, u^1) = (1, u)$, die durch $x \mapsto 1 + (u-1)x$ gegeben ist. Das liefert genau (i).
(ii) Wegen (i) gilt $u^Y \leq 1 + (u-1)Y$ als Ungleichung zwischen Zufallsvariablen. Die Behauptung folgt wegen der Monotonie und der Linearität des Erwartungswertes. □

Mit Lemma 2.6.2(ii) erhalten wir aus (2.6.11):

$$\mathbf{Pr}(X \geq r) \leq u^{-r} \cdot \prod_{1\leq i\leq n} \left(1 + (u-1)\mathbf{E}(X_i)\right). \tag{2.6.12}$$

4. Schritt: Arithmetisches und geometrisches Mittel. – Auf das Produkt in (2.6.12) wenden wir die Ungleichung zwischen dem arithmetischen und dem geometrischen Mittel (Prop. 2.3.9) an, mit $a_i = 1 + (u-1)\mathbf{E}(X_i)$. Dies liefert:

$$\mathbf{Pr}(X \geq r) \leq u^{-r} \cdot \left(\frac{1}{n} \sum_{1\leq i\leq n} \left(1 + (u-1)\mathbf{E}(X_i)\right)\right)^n.$$

Nun erinnern wir uns, dass $X = X_1 + \cdots + X_n$ und $m = \mathbf{E}(X) = np$ ist, und erhalten

$$\mathbf{Pr}(X \geq r) \leq u^{-r} \cdot \left(1 + (u-1) \cdot \frac{m}{n}\right)^n = u^{-r} \cdot (1 - p + pu)^n. \tag{2.6.13}$$

5. Schritt: Minimiere Schranke durch Variation von u. – Die rechte Seite $g(u) = u^{-r}(1-p+pu)^n$ in (2.6.13) hängt von dem beliebigen Wert $u > 1$ ab. Um diese Schranke möglichst gut auszunutzen, variieren wir u und suchen die Stelle, an der $g(u)$ minimal wird. Dazu verwenden wir die üblichen Techniken aus der Analysis. Ein bisschen Bruchrechnen erledigt den Rest.

Im Grenzfall $r = n$ gilt $g(u) = (u^{-1}(1-p)+p)^n \to p^n$ für $u \to \infty$, und damit $\mathbf{Pr}(X \geq r) \leq p^n = (m/r)^r$. Damit ist (2.6.8) für $r = n$ bewiesen, und wir können ab hier $m < r < n$ annehmen.

Wir orientieren uns grob: $\lim_{u\to 1} g(u) = 1$ und $\lim_{u\to\infty} g(u) = \infty$. Irgendwo dazwischen vermuten wir ein globales (also auch lokales) Minimum. Wir differenzieren mit der Produktregel:

$$g'(u) = -ru^{-(r+1)}(1-p+pu)^n + u^{-r}n(1-p+pu)^{n-1}p.$$

Nullsetzen von $g'(u)$ und Multiplizieren der entstehenden Gleichung mit $u^{r+1}(1-p+pu)^{-n+1}$ $(\neq 0)$ führt zu der Bedingung $r(1-p+pu) = npu$ für die Nullstelle u_0 von $g'(u)$ (an der wir das Minimum vermuten), und damit zu der Lösung

$$u_0 = \frac{r(1-p)}{(n-r)p} = \frac{r(1-\frac{m}{n})}{(n-r)\frac{m}{n}} = \frac{r(n-m)}{m(n-r)} > 1.$$

Einsetzen von u_0 in $g(u)$ liefert:

$$g(u_0) = \left(\frac{m(n-r)}{r(n-m)}\right)^r \cdot \left(1 - \frac{m}{n} + \frac{r(n-m)}{n(n-r)}\right)^n = \left(\frac{m(n-r)}{r(n-m)}\right)^r \cdot \left(\frac{n-m}{n-r}\right)^n.$$

Mit (2.6.13) ergibt sich

$$\mathbf{Pr}(X \geq r) \leq g(u_0) = \left(\frac{m}{r}\right)^r \cdot \left(\frac{n-m}{n-r}\right)^{n-r},$$

und das ist (2.6.8).

Um (2.6.9) zu beweisen, benutzen wir (2.6.8) auf geschickte Weise. Wir definieren „Komplementär-Zufallsvariablen“

$$\overline{X_i} := 1 - X_i, \quad 1 \leq i \leq n, \quad \text{und} \quad \overline{X} := \overline{X_1} + \cdots + \overline{X_n} = n - X.$$

Weiter setzen wir $\overline{m} := \mathbf{E}(\overline{X}) = \mathbf{E}(n - X) = n - \mathbf{E}(X) = n - m$ und $\overline{r} := n - s$. Dann gilt $\overline{m} \leq \overline{r} \leq n$. Die Zufallsvariablen $\overline{X_i}$, $1 \leq i \leq n$, sind wieder unabhängig, mit Werten in $[0, 1]$. Wir können also (2.6.8) anwenden und erhalten:

$$\mathbf{Pr}(\overline{X} \geq \overline{r}) \leq \left(\frac{\overline{m}}{\overline{r}}\right)^{\overline{r}} \cdot \left(\frac{n - \overline{m}}{n - \overline{r}}\right)^{n-\overline{r}}.$$

Wenn man diese Ungleichung wieder in die „X_i-Notation“ überführt, ergibt sich:

$$\mathbf{Pr}(X \leq s) \leq \left(\frac{n - m}{n - s}\right)^{n-s} \cdot \left(\frac{m}{s}\right)^{s},$$

und das ist gerade (2.6.9). □

Im Folgenden geben wir nützliche Varianten der Hoeffding-Ungleichung an.

Korollar 2.6.3
In der Situation von Satz 2.6.1 gilt:

$$\mathbf{Pr}(X \geq (1+\varepsilon)m) \leq \left(\frac{e^{\varepsilon}}{(1+\varepsilon)^{1+\varepsilon}}\right)^{m}, \text{ für } 0 \leq \varepsilon \leq \tfrac{n}{m} - 1; \qquad (2.6.14)$$

$$\mathbf{Pr}(X \leq (1-\varepsilon)m) \leq \left(\frac{e^{-\varepsilon}}{(1-\varepsilon)^{1-\varepsilon}}\right)^{m}, \text{ für } 0 \leq \varepsilon \leq 1. \qquad (2.6.15)$$

Beweis. Für (2.6.14) setzen wir $r := (1+\varepsilon)m$ und wenden (2.6.8) an, um zu erhalten:

$$\mathbf{Pr}(X \geq (1+\varepsilon)m) \leq \left(\frac{1}{1+\varepsilon}\right)^{(1+\varepsilon)m} \left(\frac{n-m}{n-r}\right)^{n-r}.$$

Wir müssen nur noch zeigen, dass der zweite Faktor nicht größer als $e^{\varepsilon m}$ ist. Für $r = n$ ist dies trivial, weil der zweite Faktor 1 ist; sei ab hier $r < n$. Die Ungleichung $(1 + \frac{x}{y})^y \leq e^x$, die nach Prop. A.1.2(b) für $y > 0$ und $x \geq -y$ gültig ist, liefert:

$$\left(\frac{n-m}{n-r}\right)^{n-r} = \left(1 + \frac{r-m}{n-r}\right)^{n-r} \leq e^{r-m} = e^{\varepsilon m}.$$

Für (2.6.15) setzen wir $s := (1-\varepsilon)m$, und erhalten mit (2.6.9):

$$\mathbf{Pr}(X \leq (1-\varepsilon)m) \leq \left(\frac{1}{1-\varepsilon}\right)^{(1-\varepsilon)m} \left(\frac{n-m}{n-s}\right)^{n-s}.$$

Die eben genannte Ungleichung aus Prop. A.1.2(b) liefert für den zweiten Faktor:

$$\left(\frac{n-m}{n-s}\right)^{n-s} = \left(1-\frac{m-s}{n-s}\right)^{n-s} \leq e^{-(m-s)} = e^{-\varepsilon m}.$$

□

Korollar 2.6.3 besagt Folgendes: Wenn man eine tolerierbare prozentuale Abweichung (z. B. $\varepsilon = 0.01$, was 1 Prozent entspricht) vorgibt, dann ist die Wahrscheinlichkeit, dass X weiter als diese Toleranz von seinem Erwartungswert $m = \mathbf{E}(X)$ abweicht, durch eine in m exponentiell fallende Funktion beschränkt. Je kleiner ε wird, desto näher an 1 liegt die Basis dieser Exponentialfunktion. Der Verlauf der Funktion $\varepsilon \mapsto \frac{e^{\varepsilon}}{(1+\varepsilon)^{1+\varepsilon}}$ ist in Abb. 2.6.5 angegeben.

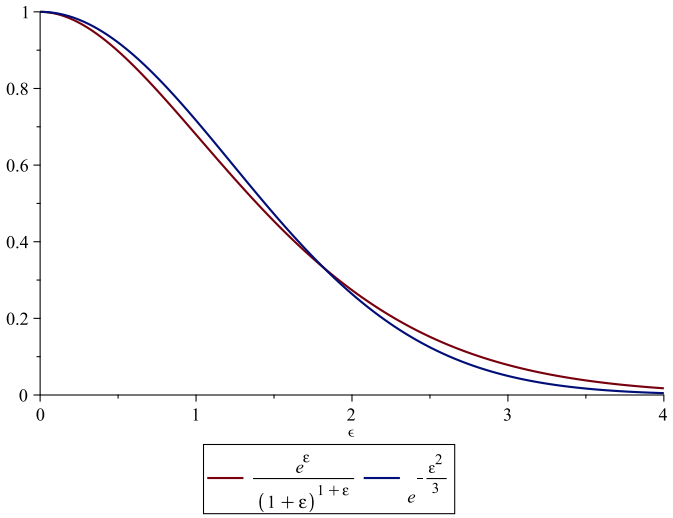


Abbildung 2.6.5: Funktionen $\varepsilon \mapsto \frac{e^{\varepsilon}}{(1+\varepsilon)^{1+\varepsilon}}$ und $\varepsilon \mapsto e^{-\varepsilon^2/3}$. Die erste Funktion ist für $0 \leq \varepsilon \leq 1.8$ durch die zweite beschränkt.

Wir notieren noch weitere nützliche und häufig benutzte Formen der Hoeffding-Ungleichungen.

Korollar 2.6.4
In der Situation von Korollar 2.6.3 gilt:

$$\mathbf{Pr}(X \geq (1+\varepsilon)m) \leq e^{-\varepsilon^2 m/3}, \text{ für } 0 \leq \varepsilon \leq 1.8; \tag{2.6.16}$$

$$\mathbf{Pr}(X \geq (1+\varepsilon)m) \leq e^{-\varepsilon^2 m/4}, \text{ für } 0 \leq \varepsilon \leq 4.1; \tag{2.6.17}$$

$$\mathbf{Pr}(X \leq (1-\varepsilon)m) \leq e^{-\varepsilon^2 m/2}, \text{ für } 0 \leq \varepsilon \leq 1; \tag{2.6.18}$$

$$\mathbf{Pr}(|X-m| \geq \varepsilon m) \leq 2e^{-\varepsilon^2 m/3}, \text{ für } 0 \leq \varepsilon \leq 1.8. \tag{2.6.19}$$

$$r \geq 5m \Rightarrow \mathbf{Pr}(X \geq r) \leq 2^{-r}. \tag{2.6.20}$$

Für eine Illustration der beiden Funktionen $e^{\varepsilon}/(1+\varepsilon)^{1+\varepsilon}$ und $e^{-\varepsilon^2/3}$ siehe Abb. 2.6.5. Aus dem Bild erkennt man auch, dass sich für $\varepsilon > 1.85$ die Beziehung zwischen den Funktionen umdreht. Der *Beweis* von (2.6.16) und (2.6.17) besteht in einer Diskussion des Verlaufs der Funktionen

$$\varepsilon \mapsto \ln\left(\frac{e^{-\varepsilon^2/K}}{e^{\varepsilon}/(1+\varepsilon)^{1+\varepsilon}}\right) = -\varepsilon^2/K - \varepsilon + (1+\varepsilon)\ln(1+\varepsilon),$$

für $K = 3$ und $K = 4$, aus der hervorgeht, dass diese Funktion im Intervall $[1, 1.8]$ (für $K = 3$) bzw. $[1, 4.1]$ (für $K = 4$) nicht negativ ist. Wir führen dies nicht im Detail durch, sondern begnügen uns mit einem Blick auf ein Bild für $K = 3$ (s. Abb. 2.6.6). Damit folgt die Behauptung direkt aus (2.6.14). Die dritte Ungleichung (2.6.17) folgt

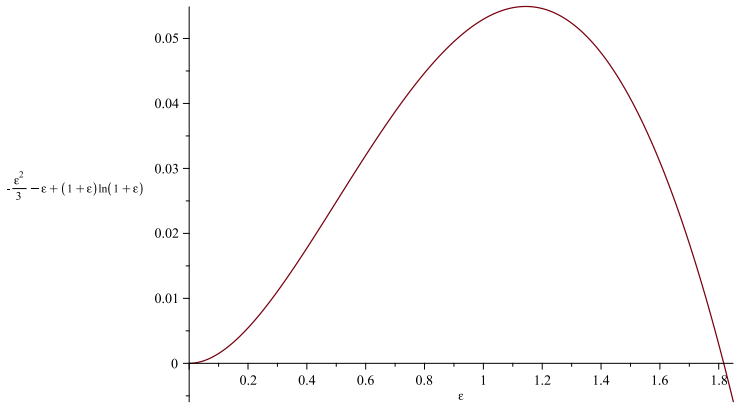


Abbildung 2.6.6: Funktion $\varepsilon \mapsto -\varepsilon^2/3 - \varepsilon + (1+\varepsilon)\ln(1+\varepsilon)$ ist in $[1, 1.8]$ nicht negativ.

ähnlich aus der Beobachtung, dass die Funktion $\varepsilon \mapsto \ln\bigl(e^{-\varepsilon^2/2}/(e^{-\varepsilon}/(1-\varepsilon)^{1-\varepsilon})\bigr) = -\varepsilon^2/2+\varepsilon+(1-\varepsilon)\ln(1-\varepsilon)$ im Intervall $[0,1]$ nicht negativ ist. Wir führen dies nicht durch. Die vierte Ungleichung (2.6.19) folgt mit der Vereinigungsschranke aus (2.6.16) und (2.6.18). Für die fünfte und letzte Ungleichung kehren wir zu $r = (1+\varepsilon)m$ und zu Schranke (2.6.14) zurück, um zu erhalten:

$$\mathbf{Pr}(X \geq r) \leq \left(\frac{e^{\varepsilon}}{(1+\varepsilon)^{1+\varepsilon}}\right)^m = \left(\left(\frac{e^{r/m-1}}{(r/m)^{r/m}}\right)^{m/r}\right)^r = \left(\frac{e^{1-m/r}}{r/m}\right)^r.$$

Die Funktion $g\colon t \mapsto e^{1-1/t}/t$ fällt strikt monoton mit $t \geq 1$. (Man betrachtet $\frac{\mathrm{d}}{\mathrm{d}t}\ln(g(t)) = \frac{\mathrm{d}}{\mathrm{d}t}(1-1/t-\ln t) = 1/t^2 - 1/t < 0$, für $t > 1$.) Wir betrachten $t_0 = 2e-1 = 4.43656... < 4.5$. Es gilt, mit Prop. A.1.3 im Anhang:

$$g(t_0) = \frac{e^{1-\frac{1}{2e-1}}}{2e-1} \leq \frac{\frac{e}{1+\frac{1}{2e-1}}}{2e-1} = \frac{1}{2}.$$

Also gilt $g(t) < \frac{1}{2}$ für alle $t \geq 2e-1$, und das liefert $\mathbf{Pr}(X \geq r) \leq (\frac{1}{2})^r$ für $r \geq (2e-1)m$.

Bemerkung 2.6.5
Um die Stärke der Hoeffding-Ungleichung(en) zu demonstrieren, betrachten wir nochmals das n-fache Werfen einer fairen Münze. Die Zufallsvariable X zählt, wie oft „Kopf“ auftaucht. In Bem. 2.5.9 (Schwaches Gesetz der großen Zahlen) wird gezeigt, wie mit der Chebychev-Ungleichung die Wahrscheinlichkeit für eine große (relative) Abweichung vom Erwartungswert beschränkt werden kann. Wir vergleichen dies mit Schranken, die sich aus der Hoeffding-Ungleichung ergeben.

Seien also $X_1, \ldots, X_n$ unabhängige 0-1-wertige Zufallsvariablen mit $\mathbf{E}(X_i) = \frac{1}{2}$ für alle i. Sei $X = X_1 + \cdots + X_n$. Sei $m = \mathbf{E}(X) = n/2$. Für eine beliebige Konstante $c > 0$ betrachten wir die (zu m) relative Abweichung $\varepsilon := \varepsilon(n) := 2\sqrt{c(\ln n)/n}$. Der Rest der Diskussion trifft nur für die n zu, für die $\varepsilon(n) \leq 1.8$ gilt, was keine ernsthafte Einschränkung ist. (Dieser relativen Abweichung entspricht eine absolute Abweichung von $\varepsilon m = 2\sqrt{c(\ln n)/n} \cdot \frac{n}{2} = \sqrt{cn\ln n}$. Wir sehen gleich in der Rechnung, weshalb es interessant ist, eine so merkwürdige Formel zu wählen.)

Wir wenden das schwache Gesetz der großen Zahlen mit $p = q = \frac{1}{2}$ an, wobei man achtgeben muss, dass in der Formulierung dort die Abweichung relativ zu n formuliert ist. Wir erhalten aus (2.5.5):

$$\mathbf{Pr}(|X - \mathbf{E}(X)| \geq \varepsilon m) = \mathbf{Pr}\left(\left|\tfrac{1}{n}X - \mathbf{E}(\tfrac{1}{n}X)\right| \geq \varepsilon/2\right) \leq \frac{1}{4(\varepsilon/2)^2 n} = \frac{1}{4c\ln n}.$$

Für jedes feste c geht diese Schranke mit $n \to \infty$ gegen 0, aber relativ gemächlich. Die Hoeffding-Ungleichung in der Form (2.6.19) liefert:

$$\mathbf{Pr}(|X - \mathbf{E}(X)| \geq \varepsilon m) \leq 2e^{-\left(2\sqrt{c(\ln n)/n}\right)^2 (n/2)/3} = 2e^{-4c(\ln n)/6} = \frac{2}{n^{2c/3}}.$$

Mit der Wahl $c = \frac{3}{2}$ ist die Schranke $2/n$, mit der Wahl $c = 3$ wird sie $2/n^2$. Dies wird mit wachsendem n rasch sehr klein.

Ein Zahlenbeispiel soll den Unterschied verdeutlichen. Wähle $c = 3/2$. Mit $n = 1\,000\,000$ gilt $\ln n = 13.8155...$, also $\varepsilon m = \sqrt{cn \ln n} \leq 4553$. Mit dem schwachen Gesetz der großen Zahlen erhalten wir für die Wahrscheinlichkeit, dass die beobachtete Anzahl von „Kopf"-Ergebnissen um mehr als 4553 (das ist weniger als 1%) vom Erwartungswert $m = 500\,000$ abweicht:

$$\mathbf{Pr}(|X - \mathbf{E}(X)| \geq 4553) \leq \frac{1}{4c \ln n} \approx 0.0121.$$

Mit der Hoeffding-Ungleichung ergibt sich

$$\mathbf{Pr}(|X - \mathbf{E}(X)| \geq 4553) \leq \frac{2}{n} = 0.000002.$$

Mit $c = 3$ vergrößert man die tolerierte Abweichung auf 6438, immer noch weniger als 1.3%. Die vom schwachen Gesetz der großen Zahlen gelieferte Wahrscheinlichkeitsschranke verringert sich auf ≈ 0.0061. Die von der Hoeffding-Ungleichung gelieferte Schranke fällt jedoch auf den winzigen Wert $2 \cdot 10^{-12}$.

Noch zwei Bemerkungen zum Schluss: (i) Auch die Hoeffding-Ungleichung verhindert nicht, dass Abweichungen von X von m um $\sqrt{n}$ oder mehr mit konstanter Wahrscheinlichkeit vorkommen. Dies ist einfach eine Eigenschaft der Binomialverteilung $B(n, \frac{1}{2})$ und hängt damit zusammen, dass die *Standardabweichung* $\sqrt{\mathbf{Var}(X)}$ gleich $\frac{1}{2}\sqrt{n}$ ist. (ii) Dass die Hoeffding-Ungleichung schärfere Schranken liefert, hat auch damit zu tun, dass das schwache Gesetz der großen Zahlen unter viel schwächeren Bedingungen als der vollständigen Unabhängigkeit gilt. (Paarweise Unabhängigkeit genügt!)

2.7 Weitere Ungleichungen

Proposition 2.7.1 (*Cauchy-Schwarz-Ungleichung*)
Für Zufallsvariablen X und Y, deren Erwartungswert und Varianz definiert ist, gilt:

$$|\mathbf{E}(XY)| \leq \sqrt{\mathbf{E}(X^2)\mathbf{E}(Y^2)}.$$

Beweis: Wir zeigen: $\mathbf{E}(XY)^2 \leq \mathbf{E}(X^2)\mathbf{E}(Y^2)$. – Für $\lambda \in \mathbb{R}$ betrachte

$$f(\lambda) := \mathbf{E}((\lambda X + Y)^2) = \lambda^2\mathbf{E}(X^2) + 2\lambda\mathbf{E}(XY) + \mathbf{E}(Y^2) = \alpha\lambda^2 + 2\beta\lambda + \mathbf{E}(Y^2),$$

mit $\alpha := \mathbf{E}(X^2) \geq 0$ und $\beta := \mathbf{E}(XY)$. Weil $(\lambda X + Y)^2 \geq 0$, gilt $f(\lambda) \geq 0$ für alle $\lambda \in \mathbb{R}$. Wenn $\alpha = 0$ ist, haben wir $2\beta\lambda + \mathbf{E}(Y^2) \geq 0$ für alle $\lambda \in \mathbb{R}$, was nur für $\mathbf{E}(XY) = \beta = 0$ möglich ist. In diesem Fall gilt also die Ungleichung. Nun nehmen wir $\alpha > 0$ an. Für $\lambda_0 := -\beta/\alpha$ (Minimalstelle der Funktion $f(\lambda)$) gilt:

$$0 \leq f(\lambda_0) = \alpha\left(-\frac{\beta}{\alpha}\right)^2 + 2\beta\left(-\frac{\beta}{\alpha}\right) + \mathbf{E}(Y^2) = -\frac{\beta^2}{\alpha} + \mathbf{E}(Y^2),$$

also $\mathbf{E}(XY)^2 = \beta^2 \leq \alpha\mathbf{E}(Y^2) = \mathbf{E}(X^2)\mathbf{E}(Y^2)$. □

Nicht ganz ideal an der Chebychev-Ungleichung (Fakt 2.3.4) ist, dass sie nur für $t > \sqrt{\mathbf{Var}(X)}$ nützliche Information liefert. Für kleinere t ist die Schranke $\mathbf{Var}(X)/t^2$ größer oder gleich 1, also trivial. Oft hilft die folgende Variante.

Proposition 2.7.2 (*Chebychev-Cantelli-Ungleichung*)
Es sei X eine Zufallsvariable mit $\mathbf{E}(X^2) < \infty$. Dann gilt für alle $t > 0$:

$$\mathbf{Pr}(X \geq \mathbf{E}(X) + t) \leq \frac{\mathbf{Var}(X)}{\mathbf{Var}(X) + t^2} \text{ und } \mathbf{Pr}(X \leq \mathbf{E}(X) - t) \leq \frac{\mathbf{Var}(X)}{\mathbf{Var}(X) + t^2}.$$

Beweis:[9] Die zweite Ungleichung folgt, indem man die erste auf die Zufallsvariable $X' = \mathbf{E}(X) - X$ anwendet, die dieselbe Varianz hat wie X. Wir zeigen die erste Ungleichung. Wir können o. B. d. A. annehmen, dass $\mathbf{E}(X) = 0$ ist (sonst betrachte

[9] Wir geben hier einen Beweis mit der Cauchy-Schwarz-Ungleichung an. In der Übung wird die Ungleichung auf direktem Weg bewiesen.

$X' = X - \mathbf{E}(X)$); dann ist $\mathbf{Var}(X) = \mathbf{E}(X^2)$. Man erinnere sich an die Iverson-Notation: $[X \leq t]$ ist die charakteristische Funktion des Ereignisses $\{X \leq t\}$, usw. Für alle $t \in \mathbb{R}$ gilt offenbar: $t - X \leq (t - X) \cdot [X < t]$, also

$$t = \mathbf{E}(t - X) \leq \mathbf{E}((t - X) \cdot [X < t]).$$

Für $t > 0$ können wir dann mit der Cauchy-Schwarz-Ungleichung wie folgt weiterrechnen:

$$\begin{aligned} t^2 &\leq \mathbf{E}((t-X)^2)\mathbf{E}([X<t]^2) \\ &= \mathbf{E}((t-X)^2)\mathbf{Pr}(X<t) \\ &= (\mathbf{Var}(X) + t^2)\mathbf{Pr}(X<t). \end{aligned}$$

Umstellen ergibt:

$$\mathbf{Pr}(X \geq t) = 1 - \mathbf{Pr}(X < t) \leq 1 - \frac{t^2}{\mathbf{Var}(X) + t^2} = \frac{\mathbf{Var}(X)}{\mathbf{Var}(X) + t^2},$$

wie gewünscht. □

Bemerkung: Wir vergleichen Prop. 2.7.2 mit der Chebychev-Ungleichung (Fakt 2.3.4). Für die Wahrscheinlichkeit einer beidseitigen Abweichung liefert die Chebychev-Ungleichung engere Schranken; sie wirkt aber nur für $t > \sqrt{\mathbf{Var}(X)}$. Die Chebychev-Cantelli-Ungleichung ist geeignet, wenn man die Wahrscheinlichkeit der Abweichung nur nach einer Seite begrenzen will; sie wirkt für alle $t > 0$.

Wir leiten noch eine obere und eine untere Schranke für $\mathbf{Pr}(X > 0)$ her, falls $X \not\equiv 0$ eine Zufallsvariable mit Werten in $\mathbb{N}$ ist.

Proposition 2.7.3

Für eine Zufallsvariable X mit Werten in $\mathbb{N}$, die nicht konstant 0 ist und deren Erwartungswert und Varianz definiert ist, gilt:

$$\frac{\mathbf{E}(X)^2}{\mathbf{E}(X^2)} \leq \mathbf{Pr}(X > 0) \leq \mathbf{E}(X).$$

Beweis: Die zweite Ungleichung folgt direkt aus der Markov-Ungleichung, da wegen der Ganzzahligkeit von X die Gleichung $\mathbf{Pr}(X > 0) = \mathbf{Pr}(X \geq 1)$ gilt. Weiter

beobachten wir, mit Prop. 2.7.2 (Chebychev-Cantelli-Ungleichung):

$$1 - \mathbf{Pr}(X > 0) = \mathbf{Pr}(X = 0) = \mathbf{Pr}(X \leq \mathbf{E}(X) - \mathbf{E}(X)) \leq \frac{\mathbf{Var}(X)}{\mathbf{Var}(X) + \mathbf{E}(X)^2}$$
$$= \frac{\mathbf{E}(X^2) - \mathbf{E}(X)^2}{\mathbf{E}(X^2)} = 1 - \frac{\mathbf{E}(X)^2}{\mathbf{E}(X^2)}.$$

Umstellen liefert die erste Ungleichung. □

Wenn X Summe von 0-1-wertigen Zufallsvariablen ist, kann man alternativ mit folgender Ungleichung die Wahrscheinlichkeit für $\mathbf{Pr}(X > 0)$ nach unten abschätzen.

Proposition 2.7.4 (*Conditional Expectation Inequality, CEI*)
Für beliebige Zufallsvariablen $X_1, X_2, \ldots, X_n$ mit Werten in $\{0, 1\}$ gilt:

$$\mathbf{Pr}(X_1 + \cdots + X_n > 0) \geq \sum_{1 \leq i \leq n} \frac{\mathbf{Pr}(X_i = 1)}{\mathbf{E}(X \mid X_i = 1)}.$$

Beweis: Sei $X = X_1 + \cdots + X_n$. Wir wählen die Zufallsvariable Y so, dass $X \cdot Y = [X > 0]$; sei dazu $Y(\omega) = 1/X(\omega)$, falls $X(\omega) > 0$ und $Y(\omega) = 0$, falls $X(\omega) = 0$. Dann gilt:

$$\begin{aligned}
\mathbf{Pr}(X > 0) &= \mathbf{E}(X \cdot Y) \qquad \text{(Wahl von } Y\text{)} \\
&= \sum_{1 \leq i \leq n} \mathbf{E}(X_i \cdot Y) \\
&\overset{(1)}{=} \sum_{1 \leq i \leq n} \mathbf{Pr}(X_i = 1) \cdot \mathbf{E}(\tfrac{1}{X} \mid X_i = 1) \\
&\overset{(2)}{\geq} \sum_{1 \leq i \leq n} \frac{\mathbf{Pr}(X_i = 1)}{\mathbf{E}(X \mid X_i = 1)}.
\end{aligned}$$

Für (1) benutzt man, dass $\mathbf{E}(X_i \cdot Y \mid X_i = 1) = \mathbf{E}(Y \mid X_i = 1)$ und $\mathbf{E}(X_i \cdot Y \mid X_i = 0) = 0$ gilt. Für (2) wendet man die Jensensche Ungleichung (Prop. 2.3.8(a)) auf die für $x > 0$ konvexe Funktion $x \mapsto \frac{1}{x}$ und die Zufallsvariable X mit dem auf $\{X_i = 1\}$ bedingten Wahrscheinlichkeitsraum an. Dies liefert $\mathbf{E}(\frac{1}{X} \mid X_i = 1) \geq 1/\mathbf{E}(X \mid X_i = 1)$. □

A Anhang

A.1 Ungleichungen aus der Analysis und der Kombinatorik

Proposition A.1.1

Für alle $x \in \mathbb{R}$ gilt $1 + x \leq e^x$, mit Gleichheit genau für $x = 0$.

Beweis: Die Funktion $f(x) = e^x - (1+x)$ besitzt die Ableitung $f'(x) = e^x - 1$ und die zweite Ableitung $f''(x) = e^x > 0$. Die Ableitung ist also strikt monoton wachsend; sie hat bei $x = 0$ ihre einzige Nullstelle. Daraus folgt, dass f an der Stelle $x = 0$ ein globales Minimum hat, d. h., es gilt $e^x - (1+x) \geq f(0) = 0$ für alle x, mit Gleichheit genau für $x = 0$. □

Diese Behauptung wird oft für den Fall (kleiner) negativer Werte angewendet; sie liest sich dann $1 - x \leq e^{-x}$, für $x > 0$ beliebig.

Proposition A.1.2

(a) Für alle $y > 0$ und alle $x \geq -1$ gilt: $(1 + x)^y \leq e^{xy}$.

(b) Für alle $y > 0$ und alle $x \geq -y$ gilt: $\left(1 + \frac{x}{y}\right)^y \leq e^x$.

Beweis. (a) Es gilt $0 \leq 1 + x \leq e^x$ (nach Prop. A.1.1). Weil $y > 0$ ist, ist die Funktion $u \mapsto u^y$ in $[0, \infty)$ monoton wachsend. Damit: $(1 + x)^y \leq (e^x)^y = e^{xy}$, also (a). Aussage (b) folgt aus (a), indem man $x' = x/y \geq -1$ betrachtet. □

Proposition A.1.3

Für alle $x \in \mathbb{R}$ mit $|x| < 1$ gilt $e^x \leq \frac{1}{1-x}$, mit Gleichheit genau für $x = 0$.

Beweis: Sei $g(x) := \ln(e^x(1 - x)) = x + \ln(1 - x)$. Wir zeigen: $g(x) \leq 0$ für $x \in \mathbb{R}, |x| < 1$, mit Gleichheit genau für $x = 0$. Differenzieren liefert $g'(x) = 1 - \frac{1}{1-x}$, eine Funktion, die in $(-1, 1)$ strikt monoton fallend ist und in $x = 0$ eine Nullstelle hat. Daraus folgt, dass $g(x)$ für $x < 0$ strikt monoton wächst und für $x > 0$ strikt monoton fällt. Weil zudem $g(0) = 0$ gilt, folgt die Behauptung. □

Proposition A.1.4

Für alle $x > 0$ gilt $\ln x \leq x - 1$, mit Gleichheit genau für $x = 1$.

Beweis: Setze $y := x - 1$. Dann lautet die Behauptung: $\ln(1+y) \leq y$ für alle $y > -1$, mit Gleichheit genau für $y = 0$. Wir wenden die Exponentialfunktion an, und erhalten, dass Folgendes äquivalent zur Behauptung ist: $1 + y \leq e^y$ für alle $y > -1$, mit Gleichheit genau für $y = 0$. Diese Aussage folgt aber aus Prop. A.1.1. □

Proposition A.1.5

Für alle $n, k \in \mathbb{N}$, $0 \leq k \leq n$, gilt:

$$\binom{n}{k} \leq \frac{n^n}{k^k (n-k)^{n-k}} = \frac{1}{(\alpha^\alpha (1-\alpha)^{1-\alpha})^n},$$

wobei $\alpha = \frac{k}{n}$. Weiterhin:

$$\binom{n}{k} \leq \left(\frac{en}{k}\right)^k.$$

Beweis: Für $k = 0$ und $k = n$ ist nichts zu zeigen – linke und rechte Seite sind beide gleich 1. Sonst gilt nach der binomischen Formel:

$$n^n = (k + (n-k))^n = \sum_{0 \leq i \leq n} \binom{n}{i} k^i (n-k)^{n-i} \geq \binom{n}{k} k^k (n-k)^{n-k},$$

und daher $\binom{n}{k} \leq \frac{n^n}{k^k (n-k)^{n-k}}$. Die zweite Ungleichung folgt, weil

$$\left(\frac{n}{n-k}\right)^{n-k} = \left(1 + \frac{k}{n-k}\right)^{n-k} < e^k,$$

mit Prop. A.1.2(b). □

A.2 Summierbarkeit

Bei der Diskussion von abzählbar unendlichen Wahrscheinlichkeitsräumen ist es bequem, „Summen“ von unendlich vielen Summanden zu betrachten, ohne dass eine

bestimmte Reihenfolge festgelegt ist. Wir notieren grundlegende Definitionen, Fakten und Schreibweisen.

Erinnerung: In der Analysis diskutiert man das Konzept einer absolut konvergenten Reihe. Dabei heißt eine Reihe $\sum_{i=1}^{\infty} a_i$ mit reellen Summanden $a_1, a_2, \ldots$ *absolut konvergent*, wenn $\sum_{i=1}^{\infty} |a_i| < \infty$, das heißt, wenn die Zahlenmenge $\{\sum_{1\le i\le n} |a_i| \mid n \ge 1\}$ eine obere Schranke in $\mathbb{R}$ hat. Zunächst ist dann $a := \sum_{i=1}^{\infty} a_i$ definiert, aber zusätzlich gilt noch, dass eine Umordnung der Summationsreihenfolge den Wert der Reihe nicht ändert, das heißt: Wenn $\pi\colon \mathbb{N} \to \mathbb{N}$ eine Bijektion ist, dann gilt auch $\sum_{i=1}^{\infty} a_{\pi(i)} = a$.

Beispiel: Die Reihe $\sum_{i=0}^{\infty} x^i$ ist absolut konvergent für jedes x mit $|x| < 1$. (Das sieht man am besten mit dem Quotientenkriterium ein.) Hingegen ist die Reihe $\sum_{i=1}^{\infty}(-1)^i/i$ zwar konvergent, aber nicht absolut konvergent (denn $\sum_{i=1}^{\infty} 1/i$ ist divergent).

In der Wahrscheinlichkeitsrechnung (mit diskreten W-Räumen) sind Diskussionen über Summationsreihenfolgen sehr unbequem. Man vermeidet dies mit einem geeigneten Konzept.[10] Im Folgenden betrachten wir Indexmengen I ohne bestimmte Reihenfolge. Wenn jedem $i \in I$ eine (reelle) Zahl a_i zugeordnet ist, möchten wir nach der „Summe“ $\sum_{i\in I} a_i$ der „Familie“ $(a_i)_{i\in I}$ fragen. Was soll das sein? Wenn I endlich ist, gibt es kein Problem: Beim Addieren der endlich vielen Zahlen in beliebiger Reihenfolge kommt immer dasselbe Ergebnis heraus, wegen der Kommutativität der Addition, und mit $\sum_{i\in I} a_i$ bezeichnen wir die Summe. Aber was ist, wenn I unendlich ist?

Definition A.2.1
Die Familie $(a_i)_{i\in I}$ heißt *summierbar*, wenn $\left\{\sum_{i\in J} |a_i| \;\middle|\; J \subseteq I \text{ endlich}\right\}$ nach oben beschränkt ist.

Lemma A.2.2
$(a_i)_{i\in I}$ ist summierbar $\Leftrightarrow$ es gibt eine eindeutig bestimmte Zahl a mit folgender Eigenschaft:

$$\forall \varepsilon > 0\ \exists J \subseteq I \text{ endlich}\ \ \forall K\colon J \subseteq K \subseteq I, K \text{ endlich}\ \Rightarrow \left|\sum_{i\in K} a_i - a\right| \le \varepsilon. \tag{A.2.21}$$

[10] In allgemeinen, nicht diskreten Wahrscheinlichkeitsräumen verschwinden die Summen, und ihre Rolle wird von Integralen übernommen.

Im Fall der Summierbarkeit heißt die Zahl a aus dem Lemma die *Summe* von $(a_i)_{i\in I}$, geschrieben $\sum_{i\in I} a_i$.

Beweis. „⇒“: Die Eindeutigkeit ist nicht schwer einzusehen. Wir zeigen nur die Existenz. Setze $I^+ := \{i \in I \mid a_i > 0\}$ und $I^- := \{i \in I \mid a_i < 0\}$. Weil $(a_i)_{i\in I}$ summierbar ist, sind $\left\{\sum_{i\in J} a_i \;\middle|\; J \subseteq I^+ \text{ endlich}\right\}$ und $\left\{\sum_{i\in J}(-a_i) \;\middle|\; J \subseteq I^- \text{ endlich}\right\}$ beide nach oben beschränkt. (Man lässt $J = \emptyset$ zu, daher enthalten die beiden Mengen mindestens die Zahl 0.) Es sei s^+ das Supremum der ersten Menge (zu I^+) und s^- das Supremum der zweiten Menge. Wir definieren $a := s^+ - s^-$ und zeigen, dass dieses a die Aussage des Lemmas erfüllt. Sei $\varepsilon > 0$ gegeben. Wähle eine endliche Menge $J^+ \subseteq I^+$ mit $\sum_{i\in J^+} a_i \geq s^+ - \frac{1}{2}\varepsilon$, und eine endliche Menge $J^- \subseteq I^-$ mit $\sum_{i\in J^-}(-a_i) \geq s^- - \frac{1}{2}\varepsilon$. Definiere $J := J^+ \cup J^-$. Wenn K eine endliche Menge mit $J \subseteq K \subseteq I$ ist, dann kann sich $\sum_{i\in K} a_i = \sum_{i\in K\cap I^+} a_i - \sum_{i\in K\cap I^-}(-a_i)$ um nicht mehr als $2 \cdot \frac{1}{2}\varepsilon = \varepsilon$ von a unterscheiden, weil die erste Summe zwischen $s^+ - \frac{1}{2}\varepsilon$ und s^+ und die zweite Summe zwischen $s^- - \frac{1}{2}\varepsilon$ und s^- liegt.

„⇐“: Wenn die Menge $\{\sum_{i\in J} |a_i| \mid J \subseteq I \text{ endlich}\}$ nicht nach oben beschränkt ist, dann ist, mit I^+ und I^- wie im Teil „⇒“, die Menge $\{\sum_{i\in K} a_i \mid K \subseteq I^+ \text{ endlich}\}$ oder $\{\sum_{i\in K}(-a_i) \mid K \subseteq I^- \text{ endlich}\}$ nicht nach oben beschränkt. Wir nehmen z. B. den ersten Fall an. Sei nun $J \subseteq I$ endlich. Weil $\{\sum_{i\in K} a_i \mid K \subseteq I^+ \text{ endlich}\}$ nicht nach oben beschränkt ist, gibt es endliche Teilmengen $K \subseteq I^+$ mit $\sum_{i\in K} a_i$ beliebig groß, also ist $\{\sum_{i\in J\cup K} a_i \mid K \subseteq I^+ \text{ endlich}\}$ nicht nach oben beschränkt. Daher kann (A.2.21) für kein a erfüllt sein. □

Wir werden die Theorie der summierbaren Familien hier nicht weiter ausarbeiten, sondern belassen es dabei, einige Rechenregeln zu notieren, ohne Beweis. Diese besagen, dass sich solche Summen durch die „eingebaute“ absolute Konvergenz gegenüber Operationen sehr gutmütig verhalten, und man mit ihnen im Wesentlichen wie mit endlichen Summen rechnen kann.

Proposition A.2.3

(a) **(Majorisierung und Auswahl)** Wenn $(a_i)_{i\in I}$ summierbar ist und $(b_i)_{i\in I}$ erfüllt $|b_i| \leq |a_i|$ für jedes i, dann ist auch $(b_i)_{i\in I}$ summierbar. Insbesondere: Wenn $(a_i)_{i\in I}$ summierbar und $J \subseteq I$ beliebig ist, dann ist auch die Teilfamilie $(a_i)_{i\in J}$ summierbar.

(b) **(Monotonie)** Wenn $(a_i)_{i\in I}$ und $(b_i)_{i\in I}$ summierbar sind und es gilt $a_i \leq b_i$ für jedes i, dann gilt

$$\sum_{i\in I} a_i \leq \sum_{i\in I} b_i.$$

(c) **(Linearität)** Wenn $(a_i)_{i\in I}$ und $(b_i)_{i\in I}$ summierbar sind, und $\alpha, \beta \in \mathbb{R}$, dann ist auch $(\alpha a_i + \beta b_i)_{i\in I}$ summierbar, und es gilt

$$\sum_{i\in I}(\alpha a_i + \beta b_i) = \alpha\Big(\sum_{i\in I} a_i\Big) + \beta\Big(\sum_{i\in I} b_i\Big).$$

(d) **(Distributivität)** Wenn $(a_i)_{i\in I}$ und $(b_j)_{j\in J}$ summierbar sind, dann ist auch $(a_i \cdot b_j)_{(i,j)\in I\times J}$ summierbar, und es gilt

$$\sum_{(i,j)\in I\times J} (a_i \cdot b_j) = \Big(\sum_{i\in I} a_i\Big) \cdot \Big(\sum_{j\in J} b_j\Big).$$

(e) **(Assoziativität)** Wenn $(a_i)_{i\in I}$ summierbar ist und $(I_k)_{k\in K}$ eine beliebige disjunkte Zerlegung von I ist (d. h. $I_k \cap I_{k'} = \emptyset$ für $k \neq k'$ und $\bigcup_{k\in K} I_k = I$), dann gilt

$$\sum_{i\in I} a_i = \sum_{k\in K}\Big(\sum_{i\in I_k} a_i\Big).$$

(Insbesondere sind alle vorkommenden Summen definiert.)

In unserer Diskussion haben wir nie gesagt, dass die Indexmenge I abzählbar unendlich sein muss. Es könnte also zum Beispiel auch $I = \mathbb{R}$ sein. Die folgende Tatsache zeigt aber, dass diese Verallgemeinerung gegenüber absolut konvergenten Reihen mit Indexmenge $\mathbb{N}$ nur scheinbar ist.

Lemma A.2.4
Wenn $(a_i)_{i\in I}$ summierbar ist, dann ist die Menge $J_{\neq 0} = \{i \in I \mid a_i \neq 0\}$ endlich oder abzählbar unendlich.

Beweis. Für $n \geq 1$ setze $I_n := \{i \in I \mid |a_i| \geq \frac{1}{n}\}$. Weil $(a_i)_{i\in I}$ summierbar ist, ist jede der Mengen I_n endlich, und daher ist $J_{\neq 0}$ als Teilmenge von $\bigcup_{n\geq 1} I_n$ entweder selbst endlich oder abzählbar unendlich. □

3 Modellierung und Transformationen

In diesem Abschnitt überlegen wir zunächst allgemein, wie man die Semantik von randomisierten Algorithmen beschreiben kann. Hierzu benutzen wir eine randomisierte Version der Registermaschine. Diese Modellierung erlaubt es, den Wahrscheinlichkeitsraum präzise zu spezifizieren, auf dem unsere Aussagen über Fehlerwahrscheinlichkeiten und Erwartungswerte für Laufzeiten u. ä. beruhen. Wir stellen weiter allgemeine Techniken bereit, um die Fehlerwahrscheinlichkeit zu verringern und im Fall von fehlerbehafteten randomisierten Algorithmen worst-case-Laufzeitschranken (anstelle von Erwartungswerten) herzustellen. Zuletzt diskutieren wir den Zusammenhang zwischen fehlerfreien und „selbst-verifizierenden“ randomisierten Algorithmen.

3.1 Modell, Grundbegriffe

Wir gehen vom Modell der *Registermaschine* (RAM) aus, wie es in Vorlesungen zu „Berechenbarkeit und Komplexitätstheorie“ besprochen wird. (Zur Auffrischung: Anhang A.) Eine solche Registermaschine wird durch ein Programm $(B_i)_{0\le i<l}$ spezifiziert, wobei die Befehlszeilen B_i aus dem Befehlsvorrat des RAM-Modells stammen (Lade- und Speicherbefehle, arithmetische Befehle, Sprungbefehle). Wenn wir eine Eingabe in passender Form in einige Register schreiben und M mit Befehlszeile 0 starten, ergibt sich in eindeutiger Weise eine Berechnung. Falls diese hält, kann das Resultat aus den Registerinhalten abgelesen werden.

Wir wollen dies noch etwas formaler fassen. Die Menge $\mathcal{K}$ der Konfigurationen (die gar nicht von M abhängt) hat als Elemente Paare (z,α), wobei $z \in \mathbb{N}$ (Befehlszählerstand) und $\alpha\colon \mathbb{N} \to \mathbb{N}$ eine Funktion ist, die zu i den Registerinhalt $\alpha(i) = \langle R_i\rangle$ angibt. Wir verlangen, dass $\alpha(i) = 0$ für „fast alle i“, d. h. dass $\alpha(i) \neq 0$ nur für endlich viele i gilt.

Eine *Startkonfiguration* $k = (z,\alpha)$ ist eine Konfiguration mit $z = 0$ und $\alpha(i) = 0$ für alle $i > 0$ außer ungeraden $i < 2 \cdot \alpha(0)$. Wenn $k = (z,\alpha) \in \mathcal{K}$ eine Konfiguration ist, dann ist k entweder eine *Haltekonfiguration* (nämlich wenn $z \ge l$ gilt, also kein nächster Schritt definiert ist) oder es kann Befehlszeile B_z ausgeführt werden, was zu einer neuen Konfiguration k' führt. Wir schreiben in diesem Fall: $k \vdash_M k'$, und sagen,

dass k' die eindeutig bestimmte Nachfolgekonfiguration von k ist. Wenn man in einer Startkonfiguration k_0 beginnt, ergibt sich so eine Konfigurationenfolge

$$k_0 \vdash_M k_1 \vdash_M k_2 \vdash_M \cdots ,$$

die in einer Haltekonfiguration enden oder unendlich lange weiterlaufen kann.

Wir wollen die Ein-/Ausgabekonventionen im Vergleich zum Anhang, wo nur n-Tupel von Zahlen als Inputs zugelassen sind, ein wenig verändern: Ein *Algorithmus* $\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$ besteht aus einer Registermaschine $M = M_{\mathcal{A}}$ (also einem Programm) und zusätzlich noch einer Inputmenge $I = I_{\mathcal{A}}$ und einer Outputmenge $Z = Z_{\mathcal{A}}$, sowie einer Inputfunktion $\text{IN} = \text{IN}_{\mathcal{A}}$ mit $\text{IN}\colon I \to \mathcal{K}$, die einen Input $x \in I$ in eine Startkonfiguration $\text{IN}(x) \in \mathcal{K}$ transformiert, und einer Outputfunktion $\text{OUT} = \text{OUT}_{\mathcal{A}}$ mit $\text{OUT}\colon \{k \in \mathcal{K} \mid k \text{ Haltekonfiguration}\} \to Z$, die aus einer Haltekonfiguration k ein Ergebnis $\text{OUT}(k) \in Z$ extrahiert. (Die Funktionen IN und OUT sollten „sehr einfache“ Kodierungs- und Dekodierungsfunktionen sein und keine Berechnung „verstecken“.) Um $\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$ auf x auszuführen, startet man M auf $k_0 = \text{IN}(x)$, und betrachtet die davon erzeugte Konfigurationenfolge $k_0 \vdash_M k_1 \vdash_M k_2 \vdash_M \cdots$. Falls diese mit einer Haltekonfiguration k_t endet, ist

$$\mathcal{A}(x) := \text{OUT}(k_t) \in Z$$

das Resultat.

Die Churchsche These (siehe „Automaten, Sprachen und Komplexität (ASK)“) besagt unter anderem, dass jeder Algorithmus in ein RAM-Programm transformiert werden kann; in diesem Sinn ist es also gerechtfertigt, RAM-Programme zusammen mit Ein-/Ausgabekonventionen mit Algorithmen gleichzusetzen. Unter gewissen Einschränkungen an die Bitlänge der in den Registern zu speichernden Zahlen kann man auch sagen, dass die Rechenzeit eines RAM-Algorithmus dem Standard-Rechenzeitbegriff aus der Algorithmik ziemlich genau entspricht.

Wir wollen nun das RAM-Modell um den Aspekt „Randomisierung“ erweitern. Hierzu führen wir einen weiteren Befehl ein:

RANDOM

Dieser Befehl soll (intuitiv gesehen) folgenden Effekt haben: Wenn in R_0 eine Zahl $r = \langle R_0 \rangle \geq 2$ steht, so wird eine Zahl s aus der Menge $\{0, 1, \ldots, r-1\}$ zufällig (gemäß

der uniformen Verteilung) gewählt und nach R_0 geschrieben. Falls in R_0 die Zahl 0 oder 1 steht, wird 0 nach R_0 geschrieben. Der Befehlszähler wird um 1 erhöht.

Registermaschinen, die diesen neuen Befehl zur Verfügung haben, heißen „***randomisierte Registermaschinen* (*RRAM*)**“. In Anlehnung an die Churchsche These wollen wir eine RRAM M zusammen mit Ein-/Ausgabekonventionen einen **(*sequentiellen*) *randomisierten Algorithmus*** nennen.

Beispiel 3.1.1
Das folgende Programm in Pseudocode hat keinen Input und kann nur die Ausgabe „1“ erzeugen:

$R_0 \leftarrow 2$
RANDOM
while $R_0 = 0$ **do**
    $R_0 \leftarrow 2$; RANDOM
return R_0

Als RRAM-Programm:

Zeile	Befehl	Kommentar
0	$R_0 \leftarrow 2$	
1	RANDOM	0 oder 1 in R_0
2	if $R_0 > 0$ goto 6	Ende mit $\langle R_0 \rangle = 1$
3	$R_0 \leftarrow 2$	
4	RANDOM	0 oder 1 in R_0
5	goto 2	
6	$R_1 \leftarrow R_0$	Resultatformat
7	$R_0 \leftarrow 1$	herstellen

Beispiel 3.1.2
Das folgende Programm in Pseudocode berechnet wenigstens eine eventuell interessante Ausgabe. Der Input ist $s \geq 0$, was die Anzahl der Seiten eines fairen „Würfels“ sein soll. Die Seiten heißen $0, 1, \ldots, s-1$. Der Algorithmus soll wiederholt würfeln, bis eine 0 erscheint, und die Anzahl der ausgeführten Würfe ausgeben. Bei Eingabe 0 wird 0 ausgegeben.

Input: $s \in \mathbb{N}$.
if $s = 0$ **then return** 0
count $\leftarrow 1$
$R_0 \leftarrow s$
RANDOM
while $R_0 > 0$ **do**
    count $\leftarrow$ count $+ 1$
    $R_0 \leftarrow s$
    RANDOM
return count.

Die Ausgabe ist $\mathcal{A}(s)$, eine Zufallsvariable. Der Zufall „steckt“ dabei nicht in s (das ist durch die Eingabe festgelegt), sondern in den Zufallsexperimenten, die der Algorithmus ausführt.

Wenn wir diesen Algorithmus als RRAM-Programm darstellen wollen, sorgen wir dafür, dass die IN-Funktion aus einer Eingabe $x = s$ das richtige Eingabeformat herstellt (Befehlszähler auf 0, 1 nach R_0, s nach R_1, sonstige Register auf 0) und dass die OUT-Funktion die Ausgabe, die am Ende in R_1 bzw. $\alpha(1)$ steht, ausgibt.

Zeile	Befehl	Kommentar
0	if $R_1 = 0$ goto 12	$s = 0$ liefert Ausgabe 0
1	$R_4 \leftarrow 1$	Konstante 1
2	$R_2 \leftarrow 1$	Schrittzähler
3	$R_0 \leftarrow R_1$	s nach R_0 kopieren
4	RANDOM	würfeln
5	if $R_0 = 0$ goto 10	Ende wenn Ergebnis 0
6	$R_2 \leftarrow R_2 + R_4$	Zähler erhöhen
7	$R_0 \leftarrow R_1$	s nach R_0 kopieren
8	RANDOM	würfeln
9	goto 5	Schleifenanfang
10	$R_1 \leftarrow R_2$	Zählerstand nach R_1
11	$R_0 \leftarrow 1$	

Bis auf nicht so wesentliche technische Einzelheiten (irrationale Wahrscheinlichkeiten oder rationale Wahrscheinlichkeiten p, die nur mit sehr großem Zähler und sehr großem Nenner darstellbar sind) lässt sich mit dieser Modellierung jedes Zufallsexperiment nachbauen, das man im Bereich der randomisierten Algorithmen braucht.

(Übung: Wie realisiert man auf einer RRAM den Wurf einer Münze mit Wahrscheinlichkeit p für „Kopf“ und $1-p$ für „Zahl“, wobei $0 \leq p \leq 1$ rational ist?)

Wenn $k = (z, \alpha)$ eine Konfiguration einer RRAM M ist, wo B_z der RANDOM-Befehl ist, und $\alpha(0) \geq 2$, so hat k genau $\alpha(0)$ viele verschiedene legale Nachfolgekonfigurationen $k' = (z+1, \alpha')$, für jeden neuen Wert $0, 1, \ldots, \alpha(0)-1$ als $\alpha'(0)$ eine. Wir schreiben $k \vdash_M k'$ für jede dieser Konfigurationen und definieren eine Übergangswahrscheinlichkeit:

$$p_{k,k'} := \frac{1}{\alpha(0)}, \text{ für jedes solche } k'.$$

(Wenn der RANDOM-Befehl auszuführen ist und $\alpha(0) \in \{0, 1\}$, wird wie bei gewöhnlichen Rechenschritten eine eindeutige Nachfolgekonfiguration festgelegt.) Falls k nur eine Nachfolgekonfiguration k' besitzt, setzen wir $p_{k,k'} := 1$, auch in dem Fall, wo die Ausführung eines gewöhnlichen RAM-Befehls von k zu k' führt.

Nun können wir jeder (Teil-)Berechnung eine Wahrscheinlichkeit zuordnen: Wenn $\varphi = (k_0, k_1, \ldots, k_s)$ eine Konfigurationenfolge mit $k_0 \vdash_M k_1 \vdash_M \cdots \vdash_M k_s$ ist, so setzen wir

$$p_\varphi := p_{k_0 k_1} \cdot p_{k_1 k_2} \cdot p_{k_2 k_3} \cdots p_{k_{s-1} k_s}.$$

Diese Festlegung (Multiplikation der Wahrscheinlichkeiten) modelliert die Unabhängigkeit der von den RANDOM-Befehlen ausgelösten Zufallsexperimente. Wenn in φ genau u RANDOM-Befehle vorkommen, wobei in R_0 die Inhalte $r_1, \ldots, r_u \geq 2$ stehen, dann ist $p_\varphi = \frac{1}{r_1} \cdots \frac{1}{r_u}$.

Zu einem gegebenen randomisierten Algorithmus $\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$ betrachten wir für jeden Input x die Menge der Berechnungen, die möglicherweise zu interessanten Ergebnissen führen. Dies sind natürlich nur terminierende Berechnungen:

$$\Omega_{\mathcal{A},x} := \{\varphi \mid \varphi = (k_0, k_1, \ldots, k_t), \text{IN}(x) = k_0 \vdash_M \cdots \vdash_M k_t,\ k_t \text{ Haltekonfiguration}\}$$

Eine andere (anschauliche) Sicht auf diese Menge ist durch den Begriff des *Berechnungsbaums* $\text{CT}_{\mathcal{A},x}$ gegeben. Dies ist ein (meist unendlicher) Baum, der folgendermaßen induktiv aufgebaut wird: Die Wurzel ist mit der Konfiguration $k_0 = \text{IN}(x)$ beschriftet. Knoten, die mit einer Haltekonfiguration beschriftet sind, haben keine Nachfolger, sind also Blätter. Wenn Knoten v mit Konfiguration $k = (z, \alpha)$ beschriftet ist und in Befehlszeile z der Befehl „RANDOM“ ist, und wenn $r = \alpha(0) \geq 2$ ist, dann hat v genau r Nachfolgeknoten $v_0, \ldots, v_{r-1}$, die mit den r Nachfolgekonfigurationen von k beschriftet sind. Die Kante von v nach v_i wird mit „$\frac{1}{r}$“ markiert, für $0 \leq i < r$. In allen anderen Fällen hat v einen Nachfolgeknoten v', der mit der

eindeutig bestimmten Nachfolgekonfiguration k' von k beschriftet ist. Die Kante von v nach v' wird mit „1“ markiert.

Man sieht sofort, dass die Wege in $\mathrm{CT}_{\mathcal{A},x}$ genau die Berechnungen von M auf x sind, wobei es möglicherweise auch unendliche Wege, also nichtterminierende Berechnungen gibt. Der Menge $\Omega_{\mathcal{A},x}$ entsprechen Wege, die an Blättern enden. Die Wahrscheinlichkeit, in einem Knoten v „vorbeizukommen“, ist genau das Produkt der Kantenbeschriftungen auf dem Weg von der Wurzel nach v.

Wir würden gerne die Menge $\Omega_{\mathcal{A},x}$ als Trägermenge eines Wahrscheinlichkeitsraumes benutzen, um die Berechnungen von $\mathcal{A}$ auf x zu modellieren. Geht das? Wir beobachten:

Lemma 3.1.3

$$\sum_{\varphi\in\Omega_{\mathcal{A},x}} p_\varphi \leq 1.$$

Man beachte, dass es sich um eine Summe von eventuell unendlich vielen positiven Summanden handelt. Da $\mathcal{K}$ abzählbar unendlich ist, ist auch $\Omega_{\mathcal{A},x}$ höchstens abzählbar unendlich. Der Wert solcher Summen ist immer wohldefiniert.

Beweis des Lemmas:[1] Wir sagen, dass $\varphi = (k_0, \ldots, k_s)$ eine partielle Berechnung von $\mathcal{A}$ auf x ist, wenn $k_0 \vdash_M k_1 \vdash_M \cdots \vdash_M k_s$ und $k_0 = \mathrm{IN}(x)$ ist. (Wir verlangen nicht, dass k_t Haltekonfiguration ist.)
Für $t \geq 0$ definiere

$$S_t := \sum\{p_\varphi \mid \varphi = (k_0, \ldots, k_s),\ \varphi \text{ ist partielle Berechnung auf } x \text{ mit } s = t \text{ oder } (s < t \text{ und } \varphi \in \Omega_{\mathcal{A},x})\ \}.$$

Man zeigt durch Induktion über t, dass $S_t = 1$ für alle t gilt. Für $t = 0$ ist dies klar; der Induktionsschritt von $t-1$ auf t benutzt, dass sich die Summe nicht ändert, wenn man einen Rechenschritt ausführt und manche partiellen Berechnungen durch einen RANDOM-Befehl in Schritt t in mehrere partielle Berechnungen aufgespaltet werden.
Damit gilt für die Teilsumme

$$S'_t = \sum\{p_\varphi \mid \varphi = (k_0, \ldots, k_s), s \leq t \text{ und } \varphi \in \Omega_{\mathcal{A},x}\ \},$$

[1] Für Interessierte (in der Vorlesung weggelassen).

die die bis Schritt t beendeten Berechnungen erfasst, erst recht $S'_t \leq 1$. Daher ist $\sum_{\varphi\in\Omega_{\mathcal{A},x}} p_\varphi = \sup\{S'_t \mid t \geq 0\} \leq 1$, und das Lemma ist bewiesen. □

Könnte es sein, dass $\sum_{\varphi\in\Omega_{\mathcal{A},x}} p_\varphi < 1$ ist? Dann gäbe es ein $\delta > 0$ derart, dass $S'_t \leq 1-\delta$ ist für alle $t \geq 0$. Daraus folgt, dass für jedes t die Werte p_φ für die bis zum Schritt t nicht beendeten partiellen Berechnungen φ eine Summe von mindestens δ ergeben; informal kann man das so interpretieren, dass die zufallsgesteuerte Berechnung von $\mathcal{A}$ auf x mit Wahrscheinlichkeit mindestens δ nie anhält. Mit Berechnungen, die mit positiver Wahrscheinlichkeit nicht anhalten, wollen wir uns in dieser Vorlesung nicht beschäftigen. Wir schließen daher randomisierte Algorithmen $\mathcal{A}$, die $\sum_{\varphi\in\Omega_{\mathcal{A},x}} p_\varphi < 1$ für ein $x \in I$ erfüllen, von der weiteren Betrachtung aus.

Bemerkung 3.1.4
Wenn man es doch einmal mit einem solchen Algorithmus zu tun bekommt, kann man ihn wie folgt modifizieren. Aus x berechnet man eine Zeitschranke $b(x) \in \mathbb{N}$ derart, dass die Eigenschaften des Algorithmus nicht entscheidend verändert werden, wenn man ihn nach $b(x)$ Schritten abbricht. Dann ändert man das RRAM-Programm so, dass die Schritte gezählt und laufend mit der Schranke $b(x)$ verglichen werden. Wird diese Zeitschranke überschritten, wird die Berechnung abgebrochen und das Ergebnis „nicht fertig“ ausgegeben. Diese Änderung verlangsamt den Algorithmus um einen konstanten Faktor, erzwingt aber, dass er immer hält.

Festlegung.[2]
Wir betrachten nur randomisierte Algorithmen $\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$, die $\sum_{\varphi\in\Omega_{\mathcal{A},x}} p_\varphi = 1$ erfüllen, für alle $x \in I$.

Wir haben schon bemerkt, dass $\Omega_{\mathcal{A},x}$ abzählbar ist. Daher bildet nach der „Festlegung“ das Paar

$$(\Omega_{\mathcal{A},x}, (p_\varphi)_{\varphi\in\Omega_{\mathcal{A},x}})$$

einen *Wahrscheinlichkeitsraum*. Dieser Wahrscheinlichkeitsraum ist die Basis für die Untersuchung des Verhaltens von $\mathcal{A}$ auf x. Wenn wir uns explizit hierauf beziehen wollen, schreiben wir

$$\mathbf{Pr}_{\mathcal{A},x}(\ldots), \mathbf{E}_{\mathcal{A},x}(\ldots), \mathbf{Var}_{\mathcal{A},x}(\ldots), \text{ usw.}$$

Man beachte, dass jede Eingabe $x \in I$ ihren eigenen Wahrscheinlichkeitsraum erzeugt. Wir lassen die Indizes weg, wenn aus dem Zusammenhang klar ist, welcher Wahrscheinlichkeitsraum gemeint ist.

[2] Man beachte dabei, dass die Menge der Algorithmen mit dieser Eigenschaft *unentscheidbar* ist. Weder sie selbst noch ihr Komplement ist semientscheidbar (rekursiv aufzählbar).

Definition 3.1.5
Für einen Algorithmus $\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$, $x \in I$ und $\varphi \in \Omega_{\mathcal{A},x}$ definieren wir:

$$t_{\mathcal{A},x}(\varphi) := t, \text{ wenn } \varphi = (k_0, \ldots, k_t).$$

Damit ist $t_{\mathcal{A},x} \colon \Omega_{\mathcal{A},x} \to \mathbb{N}$ eine *Zufallsvariable*, für die wir auch $t_{\mathcal{A}}(x)$ schreiben. Man sollte sich von dieser Notation nicht zu dem Eindruck verleiten lassen, dass es sich bei $t_{\mathcal{A}}(x)$ um eine Zahl handelt.

Analog definieren wir den Speicherplatz:

Definition 3.1.6
Für einen Algorithmus $\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$, $x \in I$ und $\varphi \in \Omega_{\mathcal{A},x}$ definieren wir:

$$s_{\mathcal{A},x}(\varphi) := \max\{j \mid \text{ in } \varphi = (k_0, \ldots, k_t) \text{ wird } R_j \text{ gelesen oder beschrieben}\}.$$

Wir schreiben auch $s_{\mathcal{A}}(x)$ für diese Zufallsvariable.

Wir können zu diesen Zufallsvariablen Erwartungswerte bilden:

Definition 3.1.7
Die **erwartete Rechenzeit** von $\mathcal{A}$ auf x:

$$\overline{t}_{\mathcal{A}}(x) := \mathbf{E}(t_{\mathcal{A},x}) = \sum_{\varphi \in \Omega_{\mathcal{A},x}} p_\varphi \cdot t_{\mathcal{A},x}(\varphi).$$

Der **erwartete Speicherplatz** von $\mathcal{A}$ auf x:

$$\overline{s}_{\mathcal{A}}(x) := \mathbf{E}(s_{\mathcal{A},x}).$$

Man beachte: Nach Fakt 2.2.9 gilt: $\mathbf{E}(t_{\mathcal{A},x}) = \sum_{i \geq 1} \mathbf{Pr}(t_{\mathcal{A},x} \geq i)$. (Analog für $\mathbf{E}(s_{\mathcal{A},x})$.)

Im Kontrast dazu stehen ***worst-case-Rechenzeit*** und ***worst-case-Speicherplatz***:

Definition 3.1.8
Die **Rechenzeit** von $\mathcal{A}$ auf x im schlechtesten Fall:

$$\widehat{t}_{\mathcal{A}}(x) := \sup\{t_{\mathcal{A},x}(\varphi) \mid \varphi \in \Omega_{\mathcal{A},x}\}.$$

(Dies ist ein Wert in $\mathbb{N} \cup \{\infty\}$.)
Der **Speicherplatz** von $\mathcal{A}$ auf x im schlechtesten Fall:

$$\widehat{s}_{\mathcal{A}}(x) := \sup\{s_{\mathcal{A},x}(\varphi) \mid \varphi \in \Omega_{\mathcal{A},x}\}.$$

Wie bei gewöhnlichen Algorithmen betrachtet man Größenklassen von Inputs. Hierzu nimmt man an, dass eine Abbildung size: $I \to \mathbb{N}$ gegeben ist, die jedem Input x seine ***Größe*** $\text{size}(x) = |x|$ zuordnet. Die Größenklassen sind die Mengen $I_n = \{x \in I \mid |x| = n\}$. Wir maximieren Zeitschranken über solche Größenklassen:

Definition 3.1.9
Für $n \in \mathbb{N}$ definiere:

$$\overline{T}_{\mathcal{A}}(n) := \sup\{\overline{t}_{\mathcal{A}}(x) \mid |x| = n\}.$$

Diese Größe gibt die „erwartete Rechenzeit im schlechtesten Fall über alle Inputs der Größe n“ an; auf englisch: „worst case expected time“.

Definition 3.1.10
Für $n \in \mathbb{N}$ definiere:

$$\widehat{T}_{\mathcal{A}}(n) := \sup\{\widehat{t}_{\mathcal{A}}(x) \mid |x| = n\} = \sup\{t_{\mathcal{A},x}(\varphi) \mid |x| = n, \varphi \in \Omega_{\mathcal{A},x}\}.$$

Diese Größe gibt die „Rechenzeit im schlechtesten Fall über alle Inputs der Größe n“ an.

Natürlich könnte man zu $\overline{T}_{\mathcal{A}}(n)$ und $\widehat{T}_{\mathcal{A}}(n)$ analoge Größen auch für den Speicherplatz definieren.

Zuletzt wollen wir noch definieren, was wir als Resultat eines randomisierten Algorithmus $\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$ auf einem Input x ansehen wollen.

Definition 3.1.11
Sei $\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$ ein randomisierter Algorithmus. Das **Resultat** von $\mathcal{A}$ auf $x \in I$ ist die folgende Zufallsgröße:

$$\text{res}_{\mathcal{A},x}(\varphi) := \text{OUT}(k_t), \text{ für } \varphi = (k_0, \ldots, k_t) \in \Omega_{\mathcal{A},x}.$$

Statt $\text{res}_{\mathcal{A},x}$ schreiben wir meistens $\boldsymbol{\mathcal{A}(x)}$.

Bei $\text{res}_{\mathcal{A},x} = \mathcal{A}(x)$ handelt es sich also um eine *Zufallsgröße* mit Werten in Z.

3.2 Typen randomisierter Algorithmen

Definition 3.2.1
$\mathcal{A} = (M, I, Z, \text{IN}, \text{OUT})$ sei ein randomisierter Algorithmus, und $f\colon I \to Z$ sei eine Funktion. Die Zahl

$$\text{err}_{\mathcal{A},f}(x) := \mathbf{Pr}(\mathcal{A}(x) \neq f(x))$$

heißt die **Fehlerwahrscheinlichkeit von $\mathcal{A}$ bzgl. f auf Eingabe x**. Noch genauer, wenn man den Wahrscheinlichkeitsraum explizit macht:

$$\text{err}_{\mathcal{A},f}(x) = \sum_{\substack{\varphi \in \Omega_{\mathcal{A},x} \\ \text{res}_{\mathcal{A},x}(\varphi) \neq f(x)}} p_\varphi.$$

Klar: Mit Wahrscheinlichkeit $1 - \text{err}_{\mathcal{A},f}(x)$ berechnet $\mathcal{A}$ auf Input x den Wert $f(x)$.

Definition 3.2.2
$\mathcal{A}$ und f seien wie eben. Wir sagen, dass $\mathcal{A}$ die Funktion f „mit unbeschränktem Fehler“[3] berechnet, falls für jedes $x \in I$ gilt: $\text{err}_{\mathcal{A},f}(x) < \frac{1}{2}$.

Bemerkung 3.2.3
Wenn es eine Funktion f gibt, die von $\mathcal{A}$ mit unbeschränktem Fehler berechnet wird, dann ist f eindeutig bestimmt. (*Beweis*: Sei $x \in I$. Dann gibt es maximal ein $z \in Z$ mit $\mathbf{Pr}(\mathcal{A}(x) \neq z) < \frac{1}{2}$. Dies sieht man wie folgt: Seien $z, z' \in Z$ verschieden. Dann sind $\{\mathcal{A}(x) = z\}$ und $\{\mathcal{A}(x) = z'\}$ zwei disjunkte Ereignisse, die also

[3] Korrekter müsste man „mit unter $\frac{1}{2}$ nicht beschränkter Fehler*wahrscheinlichkeit*“ oder noch besser „... *Irrtumswahrscheinlichkeit*“ sagen, aber das ist zu umständlich und daher nicht üblich.

nicht beide Wahrscheinlichkeit $> \frac{1}{2}$ haben können. Also können die Ungleichungen $\mathbf{Pr}(\mathcal{A}(x) \neq z) < \frac{1}{2}$ und $\mathbf{Pr}(\mathcal{A}(x) \neq z') < \frac{1}{2}$ nicht gleichzeitig gelten.)
Die Fehlerwahrscheinlichkeit $\frac{1}{2}$ ist hier die Grenze: Der Algorithmus, der auf Input x einfach zufällig einen der Werte 0 und 1 wählt und ausgibt, berechnet *jede* Funktion $f\colon I \to \{0,1\}$ „mit Fehlerwahrscheinlichkeit $\leq \frac{1}{2}$“. Solche Algorithmen können natürlich zur Berechnung von Funktionen nicht geeignet sein.

Um den Überblick zu behalten, werden wir oft Fehlerschranken angeben, die für alle Inputs einer Größe gelten. Uns interessiert also die Situation, wo es für jedes n eine Schranke $\varepsilon_n < \frac{1}{2}$ für die Werte $\mathrm{err}_{\mathcal{A},f}(x)$ für alle x mit $|x| = n$ gibt.

Definition 3.2.4
$\mathcal{A}$ und f seien wie eben.
Sei $(\varepsilon_n)_{n\geq 0}$ eine Folge von Zahlen mit $0 \leq \varepsilon_n < \frac{1}{2}$. Wir sagen, dass $\mathcal{A}$ die Funktion f **mit Fehlerschranke $(\varepsilon_n)_{n\geq 0}$ berechnet**, falls für jedes $x \in I$ mit $|x| = n$ gilt: $\mathrm{err}_{\mathcal{A},f}(x) \leq \varepsilon_n$.
$\mathcal{A}$ berechnet die Funktion f **mit beschränktem Fehler**, falls ein $\varepsilon < \frac{1}{2}$ existiert, so dass $\mathrm{err}_{\mathcal{A},f}(x) \leq \varepsilon$ für alle $x \in I$. In diesem Fall heißt $\mathcal{A}$ ein **Monte-Carlo-Algorithmus**.

Beispiel: Es sei $\mathcal{A}$ der Algorithmus MinCut aus Kapitel 1, in der Variante mit $n^2/2$ Wiederholungen, also Fehlerwahrscheinlichkeit e^{-1}, so modifiziert, dass er nicht den ermittelten Schnitt C, sondern nur die Kardinalität $|C|$ als Resultat ausgibt. Dieser Algorithmus berechnet die Funktion f, die einem Graphen G die Größe eines minimalen Schnitts in G zuordnet, mit Fehlerschranke e^{-1}. Wenn man $n^2(\ln n)/2$ Wiederholungen ansetzt, erhält man einen Algorithmus mit Fehlerschranke $(1/n)_{n\geq 1}$. (Hierbei ist die Eingabegröße n, die Anzahl der Knoten.)

Wir betrachten nun noch den äußerst häufigen und wichtigen Spezialfall von *Entscheidungsproblemen*, also Funktionen mit nur zwei Werten 0 („nein“) und 1 („ja“), mit einem Algorithmus, der nur bei einem Funktionswert einen Fehler machen kann.

Definition 3.2.5
Sei $\mathcal{A}$ ein Algorithmus für Inputs aus einer Menge I, derart dass $\mathrm{res}_{\mathcal{A},x}$ nur Werte in $\{0,1\}$ annimmt, und sei $f\colon I \to \{0,1\}$ eine Funktion. Wir sagen, dass $\mathcal{A}$ die Funktion f „mit einseitigem Fehler“ berechnet, falls für jedes $x \in I$ gilt:

$$\begin{aligned} f(x) = 0 &\Rightarrow \mathbf{Pr}(\mathcal{A}(x) = 0) = 1; \\ f(x) = 1 &\Rightarrow \mathbf{Pr}(\mathcal{A}(x) = 0) < 1. \end{aligned}$$

Das heißt also Folgendes:

- Wenn $\mathcal{A}$ auf Input x als Resultat 1 ausgibt (auf mindestens einem Berechnungsweg), dann ist garantiert $f(x) = 1$.
- Wenn $\mathcal{A}$ auf Input x als Resultat 0 ausgibt, ist $f(x) = 0$ und $f(x) = 1$ möglich.

Der Zusammenhang zwischen Algorithmus und Funktion hat Ähnlichkeit mit der Situation bei nichtdeterministischen Maschinenmodellen. Man sieht sofort, dass ein solcher Algorithmus die Funktion f eindeutig bestimmt. Wieder ist die Situation besonders interessant, wo die Fehlerwahrscheinlichkeit durch eine Schranke kleiner als 1 beschränkt ist, für alle Inputs (einer Größe) gleichmäßig.

Definition 3.2.6
$\mathcal{A}$ und f seien wie eben, mit Werten 0 und 1. Sei weiter $(\varepsilon_n)_{n\geq 0}$ eine Folge von Zahlen mit $0 \leq \varepsilon_n < 1$. Wir sagen, dass $\mathcal{A}$ die Funktion f mit einseitigem Fehler und Fehlerschranke $(\varepsilon_n)_{n\geq 0}$ berechnet, wenn für jedes $x \in I$ mit $|x| = n$ gilt:

$$\begin{aligned} f(x) = 0 &\Rightarrow \mathbf{Pr}(\mathcal{A}(x) = 0) = 1; \\ f(x) = 1 &\Rightarrow \mathbf{Pr}(\mathcal{A}(x) = 0) \leq \varepsilon_n. \end{aligned}$$

Gängige, etwas ungenaue Abkürzung: „$\mathcal{A}$ berechnet f mit einseitigem Fehler ε_n."
$\mathcal{A}$ berechnet die Funktion f **mit beschränktem einseitigen Fehler**, falls ein $\varepsilon < 1$ existiert, so dass $\mathcal{A}$ die Funktion f mit einseitigem Fehler und Fehlerschranke $(\varepsilon_n)_{n\geq 0}$ berechnet, mit $\varepsilon_n = \varepsilon$ für alle $n \geq 0$. In diesem Fall heißt $\mathcal{A}$ ein **Monte-Carlo-Algorithmus mit einseitigem Fehler**.

Beispiel: Die Algorithmen in Abschnitt 1.2 (Vergleich von Polynomprodukten) und 1.3 (Verifikation von Matrixprodukten) sind Monte-Carlo-Algorithmen mit einseitigem Fehler.

Schließlich sind natürlich Algorithmen interessant, die stets (mit Wahrscheinlichkeit 1) das richtige Resultat liefern.

Definition 3.2.7
$\mathcal{A}$ sei ein Algorithmus für Inputs aus einer Menge I, $f\colon I \to Z$ sei eine Funktion. Wir sagen, dass $\mathcal{A}$ die Funktion f „ohne Fehler“ berechnet, falls für jedes $x \in I$ gilt:

$$\mathrm{err}_{\mathcal{A},f}(x) = 0 \text{ , d. h. } \mathbf{Pr}(\mathcal{A}(x) \neq f(x)) = 0.$$

In diesem Fall heißt $\mathcal{A}$ ein **Las-Vegas-Algorithmus**.

Beispielsweise ist Randomisiertes Quicksort (Abschnitt 1.4) ein Las-Vegas-Algorithmus. Man verwendet Las-Vegas-Algorithmen, wenn sie einfacher (zu programmieren) sind als deterministische Varianten, wenn die erwartete Laufzeit besser ist als die worst-case-Laufzeit, und um worst-case-Inputs auszuschließen.

3.3 Wahrscheinlichkeitsverbesserung

Was sollen wir mit einem Algorithmus $\mathcal{A}$ anfangen, der eine 0-1-wertige Funktion f mit einseitigem Fehler 0,99 berechnet? Wenn dieser Algorithmus auf Eingabe x die Ausgabe 1 liefert, wissen wir, dass $f(x) = 1$ ist – gut. Aber wenn die Ausgabe 0 ist, wissen wir ja eigentlich gar nichts.

Genauso kann man fragen, was ein Algorithmus $\mathcal{A}$ nützt, der eine Funktion f mit Fehlerschranke 0,49 berechnet. Wenn auf Input x die Ausgabe y geliefert wird, ist es sehr gut möglich, dass $f(x) \neq y$ ist. Was sollen wir mit einem solchen Resultat anfangen?

Der Ansatz der „Wahrscheinlichkeitsverbesserung“ (engl.: *probability amplification*) führt hier weiter. Man benutzt den Algorithmus $\mathcal{A}$ mit der schlechten Fehlerschranke, um einen neuen Algorithmus $\mathcal{A}'$ mit kleinerer Fehlerwahrscheinlichkeit zu erhalten.

3.3.1 Einseitiger Fehler

Wir beginnen mit MC-Algorithmen mit einseitigem Fehler.

Sei $\mathcal{A}$ ein Algorithmus, der f mit einseitigem Fehler mit Schranke $(\varepsilon_n)_{n\geq 0}$ berechnet.

Wir betrachten den folgenden neuen Algorithmus $\mathcal{A}'$ (in Pseudocode-Notation), der im Wesentlichen $\mathcal{A}$ auf Eingabe x mit $|x| = n$ c-mal wiederholt (wobei $c = c_n$ eine zu wählende Zahl ist) und dabei beobachtet, ob einmal das Resultat 1 auftritt.

Algorithmus 3.3.1 ***Wiederholung einseitig***

INPUT: x mit $|x| = n$.
METHODE:
1 Bestimme Wiederholungszahl $c = c_n$
2 **for** i := 1 **to** c **do**
3 r := $\mathcal{A}(x)$;
4 **if** r = 1 **then return** 1;
5 **return** 0. // Habe c-mal Resultat 0 erhalten.

Algorithmus $\mathcal{A}'$ ruft also c-mal Algorithmus $\mathcal{A}$ auf demselben Input x auf, für $c = c_n$. Wichtig ist, dass bei jedem Aufruf neue Zufallsexperimente durchgeführt werden, so dass die Resultate $r_1, \ldots, r_c$ (stochastisch) unabhängig sind.

Analyse: 1. Zeitbedarf. Zur Organisation der Wiederholungen ist in jedem Schleifendurchlauf Aufwand $O(1)$ nötig. Ein Durchlauf durch $\mathcal{A}$ kostet erwartete Zeit $\bar{t}_{\mathcal{A}}(x)$. Wegen der Linearität des Erwartungswertes ist $\bar{t}_{\mathcal{A}'}(x) = O(c_n) + c_n \cdot \bar{t}_{\mathcal{A}}(x)$.
2. Fehlerwahrscheinlichkeit. Wenn $f(x) = 0$ ist, dann liefern (mit Wahrscheinlichkeit 1) alle Aufrufe von $\mathcal{A}$ das Resultat 0, also gibt auch $\mathcal{A}'$ den Wert 0 aus. Wenn $f(x) = 1$ ist, sehen wir, nach dem Aufbau des Algorithmus:

$$\mathbf{Pr}(\mathcal{A}'(x){=}0) = \mathbf{Pr}(r_1{=}0 \wedge \ldots \wedge r_{c_n}{=}0) = \prod_{1 \le i \le c_n} \mathbf{Pr}(r_i = 0) = \mathbf{Pr}(\mathcal{A}(x) = 0)^{c_n} \le \varepsilon_n^{c_n}.$$

Dabei haben wir die Unabhängigkeit der Ergebnisse $r_1, \ldots, r_{c_n}$ benutzt, sowie die Tatsache, dass die Fehlerwahrscheinlichkeit in jedem Aufruf dieselbe ist.

Wenn $\varepsilon_n \le \frac{1}{2}$, dann gilt also $\mathbf{Pr}(\mathcal{A}'(x) = 0) \le 2^{-c_n}$, eine Zahl, die man klein machen kann, indem man c_n genügend groß wählt.

Wir sehen uns den Fall $\frac{1}{2} < \varepsilon_n < 1$ noch etwas genauer an.[4] Setze $\delta_n := 1 - \varepsilon_n$. Dann gilt $0 < \delta_n < \frac{1}{2}$. Wir haben

$$\mathbf{Pr}(\mathcal{A}'(x) = 0) \le (1 - \delta_n)^{c_n} \overset{\text{Prop. A.1.2}}{\le} e^{-c_n \delta_n}.$$

Wenn wir uns eine Fehlerschranke $\varepsilon'_n < \varepsilon_n$ wünschen, können wir vorab bestimmen,

[4] Man stelle sich zur Illustration die Schranke $\varepsilon_n = 1 - \frac{2}{n^2}$ vor, wie sie beim MinCut-Algortihmus in 1.1 aufgetreten ist.

welche Wiederholungszahl $c_n = c(\varepsilon_n, \varepsilon'_n)$ ausreichend ist:

$$\mathbf{Pr}(\mathcal{A}'(x) = 0) \leq \varepsilon'_n \Leftarrow e^{-c_n\delta_n} \leq \varepsilon'_n \Leftarrow e^{c_n\delta_n} \geq \frac{1}{\varepsilon'_n} \Leftarrow c_n\delta_n \geq \ln(1/\varepsilon'_n) \Leftarrow c_n \geq \frac{\ln(1/\varepsilon'_n)}{1-\varepsilon_n}.$$

Wir erhalten:

Satz 3.3.2
Wenn $\mathcal{A}$ die Funktion $f\colon I \to \{0,1\}$ mit einseitigem Fehler $(\varepsilon_n)_{n\geq 0}$ berechnet und $(\varepsilon'_n)_{n\geq 0}$ eine Folge von Wunsch-Fehlerschranken ist, so berechnet jede Version des Algorithmus 3.3.1, die $c_n \geq (\ln 2)\log(1/\varepsilon'_n)/(1-\varepsilon_n)$ Wiederholungen benutzt, die Funktion f mit einseitigem Fehler $(\varepsilon'_n)_{n\geq 0}$. Der erwartete Zeitbedarf ist $\bar{t}_{\mathcal{A}'}(x) = c_n \cdot (\bar{t}_{\mathcal{A}}(x) + O(1))$.

Es ist noch interessant zu überlegen, wie sich verschiedene Werte ε_n und ε'_n auf die Wiederholungszahl $c(\varepsilon_n, \varepsilon'_n) \approx \log(1/\varepsilon'_n)/(1-\varepsilon_n)$ auswirken. Der Einfluss von ε_n auf die Wiederholungszahl ist im Wesentlichen proportional zu $\frac{1}{1-\varepsilon_n}$. Beispielsweise könnte $\varepsilon_n = 1 - n^{-k}$ sein. In diesem Fall wäre der Beitrag des Nenners zur Wiederholungszahl ein Faktor von n^k. Der Einfluss von ε'_n ist im Vergleich viel geringer. Um die Fehlerschranke zu halbieren, muss man den Zähler $\log(1/\varepsilon'_n)$ nur um 1 erhöhen. Zum Beispiel führt die ziemlich kleine Wunsch-Fehlerschranke $\varepsilon'_n = 2^{-20} \approx 10^{-6}$ zu einem Zähler $20\ln 2$ in $c(\varepsilon_n, \varepsilon'_n)$. Soll die Fehlerschranke $\varepsilon'_n = 1/n^2$ sein, ist ein Zähler von $(2\ln 2)\log n$ ausreichend. Mit einem Zähler von $(\ln 2)n^\ell$ erreicht man sogar eine Fehlerschranke von $\varepsilon'_n = 2^{-n^\ell}$.

Beispielsweise können wir mit einem Aufwand, der nur einem zusätzlichen Faktor $n = |V|$ in der Laufzeit entspricht, die Fehlerwahrscheinlichkeit beim MinCut-Algorithmus auf e^{-n} drücken.

3.3.2 Allgemeine Algorithmen mit beschränktem Fehler

Hier betrachten wir Algorithmen, die Funktionen $f\colon I \to Z$ berechnen, mit Fehlerschranke $(\varepsilon_n)_{n\geq 0}$, wobei $\varepsilon_n < \frac{1}{2}$ gilt.

Die Grundidee zur Verringerung der Fehlerwahrscheinlichkeit ist dieselbe wie vorher: *Wiederholung*. Wir lassen also den gegebenen Algorithmus $\mathcal{A}$ auf Eingabe x mit $|x| = n$ c-mal ablaufen, für eine (noch zu bestimmende) Wiederholungszahl $c = c_n$. Es entstehen Ergebnisse $r_1, \ldots, r_c$ aus Z. Beispielsweise könnte $Z = \mathbb{N}$, $c = 15$ und

$$(r_1, \ldots, r_c) = (2, 1, 3, 1, 1, 2, 1, 1, 4, 2, 1, 1, 1, 3, 1)$$

sein. Es liegt nahe, hier das Ergebnis 1 auszugeben, da es ziemlich oft vorkommt. Allgemeiner gesagt: Wenn es einen Wert r gibt, der in $(r_1, \ldots, r_c)$ mehr als $\frac{1}{2}c$-mal vorkommt, sollte man diesen Wert ausgeben. Er ist dann eindeutig bestimmt. Wenn es keinen solchen Wert gibt (z.B. bei der Ergebnisfolge $(r_1, \ldots, r_c) = (2, 1, 3, 1, 2, 2, 1, 3, 4, 2, 1, 2, 1, 3, 1)$), dann ist es eigentlich egal, welchen Wert wir ausgeben.

Wir betrachten also einen neuen Algorithmus $\mathcal{A}'$ (Algorithmus 3.3.3), der im Wesentlichen $\mathcal{A}$ c-mal wiederholt und dabei beobachtet, ob ein Resultat r in der (absoluten) Mehrheit der Fälle auftritt.

Algorithmus 3.3.3 ***Mehrheitsverfahren einfach***
INPUT: $x \in I$ mit $|x| = n$.
METHODE:
1 Bestimme $c = c_n$;
2 **for** `i` $:= 1$ **to** c **do**
3 `r[i]` $:= \mathcal{A}(x)$;
4 **if** in `r[1..`c`]` kommt Wert r häufiger als $\frac{1}{2}c$-mal vor
5 **then return** r
6 **else return** einen beliebigen Wert aus Z.

Analyse: 1. Zeitbedarf. Wir nehmen der Einfachheit halber an, dass die Resultate von $\mathcal{A}$ Objekte sind, die in konstanter Zeit zu speichern und zu vergleichen sind. Die erwartete Zeit für die Wiederholungen ist wie vorher $c_n \cdot (\bar{t}_{\mathcal{A}}(x) + O(1))$. Wir werden weiter unten noch sehen, dass Aufwand $O(c_n)$ genügt, um ein Ergebnis zu identifizieren, das häufiger als $\frac{1}{2}c_n$-mal vorkommt (falls es existiert); daher ist der zusätzliche Aufwand für die Auswahl der Ausgabe ebenfalls $O(c_n)$.

2. Fehlerwahrscheinlichkeit. Sei $x \in I$ ein fester Input mit $n = |x|$. Sei $\varepsilon := \varepsilon(x) := \mathrm{err}_{\mathcal{A},f}(x)$ die Fehlerwahrscheinlichkeit von $\mathcal{A}$ auf x. Sei $c = c_n$. Die Zufallswerte $r_1, \ldots, r_c$ sollen die Resultate der c Aufrufe von $\mathcal{A}$ im Verlauf des Ablaufs von $\mathcal{A}'$ auf x sein. Nach der Formulierung des Algorithmus $\mathcal{A}'$ gilt:

$$\mathcal{A}'(x) \neq f(x) \;\Rightarrow\; \text{mindestens } c/2 \text{ der } r_i \text{ sind } \neq f(x).$$

Wir müssen also eine obere Schranke für die Wahrscheinlichkeit finden, dass dies passiert. Wir beobachten: Wenn $r_i \neq f(x)$ für mindestens $c/2$ der $i \in \{1, \ldots, c\}$, dann gibt es eine Teilmenge $D \subseteq \{1, \ldots, c\}$ mit Kardinalität $|D| \geq c/2$, so dass

$r_i \neq f(x)$ für alle $i \in D$ und $r_i = f(x)$ für alle $i \notin D$ gilt. Wir definieren Ereignisse

$$A_D := \{\forall i \in D : r_i \neq f(x) \wedge \forall i \notin D : r_i = f(x)\}.$$

Dann gilt, wie eben beobachtet:

$$\{\mathcal{A}'(x) \neq f(x)\} \subseteq \bigcup_{\substack{D \subseteq \{1,\ldots,c\} \\ |D| \geq c/2}} A_D.$$

Für jedes D gilt wegen der Unabhängigkeit der c Wiederholungen:

$$\mathbf{Pr}(A_D) = \varepsilon(x)^{|D|}(1-\varepsilon(x))^{c-|D|} = \varepsilon^{|D|}(1-\varepsilon)^{c-|D|}.$$

Mit Monotonie (Fakt 2.1.6(d)) und der Vereinigungsschranke (Fakt 2.1.6(c)) ergibt sich

$$\mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \leq \sum_{\substack{D \subseteq \{1,\ldots,c\} \\ |D| \geq c/2}} \mathbf{Pr}(A_D) = \sum_{\substack{D \subseteq \{1,\ldots,c\} \\ |D| \geq c/2}} \varepsilon^{|D|}(1-\varepsilon)^{c-|D|}.$$

Für jedes j gibt es genau $\binom{c}{j}$ viele Teilmengen $D \subseteq \{1,\ldots,c\}$ mit Kardinalität j. Wir erhalten:

$$\mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \leq \sum_{c/2 \leq j \leq c} \binom{c}{j} \varepsilon^j (1-\varepsilon)^{c-j}.$$

Weil $\varepsilon < \frac{1}{2}$ und $j \geq c/2$, gilt $\varepsilon^{j-c/2} \leq (1-\varepsilon)^{j-c/2}$, und wir können folgern:

$$\mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \leq \sum_{c/2 \leq j \leq c} \binom{c}{j} (\varepsilon(1-\varepsilon))^{c/2} \leq \sum_{0 \leq j \leq c} \binom{c}{j} (\varepsilon(1-\varepsilon))^{c/2}.$$

Mit $\sum_{0 \leq j \leq c} \binom{c}{j} = 2^c$ erhalten wir schließlich:

$$\mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \leq (4\varepsilon(1-\varepsilon))^{c/2}.$$

Wir beobachten, dass die Funktion $t \mapsto t(1-t)$ im Intervall $[0, \frac{1}{2}]$ strikt monoton wächst, und daher $4\varepsilon(1-\varepsilon) < 4 \cdot \frac{1}{2} \cdot \frac{1}{2} = 1$ gilt. Damit sieht unser Ergebnis schon ganz gut aus: Wir erhalten eine Schranke für die Fehlerwahrscheinlichkeit von $\mathcal{A}'$, die in der Wiederholungszahl c exponentiell fällt (mit Basis $\sqrt{4\varepsilon(1-\varepsilon)} < 1$). Weil $\varepsilon_n < \frac{1}{2}$ eine obere Schranke für $\varepsilon = \varepsilon(x)$ ist, gilt wegen der Monotonie von $t \mapsto t(1-t)$ auch $\mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \leq (4\varepsilon_n(1-\varepsilon_n))^{c_n/2}$.

Wie groß muss man $c_n = c(\varepsilon_n, \varepsilon'_n)$ wählen, um eine vorgegebene Fehlerwahrscheinlichkeit ε'_n zu erreichen?

Wenn $\varepsilon_n \leq \frac{2-\sqrt{3}}{4} \approx 0{,}067$, dann gilt $\varepsilon_n(1-\varepsilon_n) \leq \frac{1}{16}$, also $\sqrt{4\varepsilon_n(1-\varepsilon_n)} \leq \frac{1}{2}$, und

$$\mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \leq \sqrt{4\varepsilon_n(1-\varepsilon_n)}^{c_n} \leq 2^{-c_n}.$$

Damit genügt in diesem Fall eine Wiederholungszahl von $c_n \geq \log(1/\varepsilon'_n)$. Wir kümmern uns von hier an nur noch um den Fall $\frac{2-\sqrt{3}}{4} < \varepsilon_n < \frac{1}{2}$. Wir setzen $\delta_n := \frac{1}{2} - \varepsilon_n$. Dann gilt $4\varepsilon_n(1-\varepsilon_n) = 4(\frac{1}{2} - \delta_n)(\frac{1}{2} + \delta_n) = 1 - 4\delta_n^2$. Damit, wie vorher:

$$\mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \leq (1-4\delta_n^2)^{c_n/2} \overset{\text{Prop. A.1.2}}{\leq} e^{-2\delta_n^2 \cdot c_n}.$$

Damit $\mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \leq \varepsilon'_n$ ist, genügt es,

$$e^{-2\delta_n^2 \cdot c_n} \leq \varepsilon'_n$$

zu haben, oder

$$2\delta_n^2 \cdot c_n \geq \ln(1/\varepsilon'_n) = (\ln 2)\log(1/\varepsilon'_n).$$

Hierfür hinreichend:

$$c_n \geq \frac{(\ln 2)\log(1/\varepsilon'_n)}{2\delta_n^2} = \frac{(2\ln 2)\log(1/\varepsilon'_n)}{(1-2\varepsilon_n)^2}. \tag{3.3.1}$$

Wir erhalten:

Satz 3.3.4
Wenn $\mathcal{A}$ die Funktion f mit Fehlerschranke $(\varepsilon_n)_{n\geq 0}$ berechnet, wobei $0 \leq \varepsilon_n < \frac{1}{2}$ ist, und $(\varepsilon'_n)_{n\geq 0}$ mit $\varepsilon'_n < \frac{1}{2}$ gegeben ist, dann berechnet jede Version von Algorithmus 3.3.3, die $c_n \geq (2\ln 2)\cdot\log(1/\varepsilon'_n)/(1-2\varepsilon_n)^2$ Wiederholungen benutzt, die Funktion f mit Fehlerschranke $(\varepsilon'_n)_{n\geq 0}$. Der erwartete Zeitbedarf ist $\bar{t}_{\mathcal{A}'}(x) = O(c_n \cdot \bar{t}_{\mathcal{A}}(x))$, für x mit $|x| = n$.

Ebenso wie im Fall des einseitigen Fehlers kann man auch hier überlegen, welcher Aufwand nötig ist, um mit schlechten Fehlerwahrscheinlichkeiten ε_n (nahe an $\frac{1}{2}$) zurechtzukommen. Wenn $\varepsilon_n = \frac{1}{2} - \delta_n$, sind $c_n \geq (2\ln 2)\cdot 2/(2\delta_n)^2 = (\ln 2)/\delta_n^2$ Wiederholungen ausreichend, um auf Fehlerwahrscheinlichkeit $\varepsilon'_n = \frac{1}{4}$ zu kommen. Für $\varepsilon_n = 0{,}49$ sind dies z. B. etwa $(\ln 2)/\frac{1}{100^2} \approx 6932$ Wiederholungen. Bei Fehlerwahrscheinlichkeiten $\varepsilon_n \leq \frac{1}{2} - n^{-k}$ kommt man unter polynomiellem Aufwand ($\Theta(n^{2k})$ Wiederholungen)

Algorithmus 3.3.5 *Mehrheitsverfahren*

INPUT: x
METHODE:
```
1   count := 0;
2   for i := 1 to c do
3     r := A(x);
4     if count = 0
5       then count := 1; R := r
6       else
7         if r = R
8           then count := count + 1
9           else count := count − 1
10  return R.
```

zu einer Fehlerwahrscheinlichkeit $\frac{1}{4}$. Wenn wir von einem Algorithmus $\mathcal{A}$ mit Fehlerwahrscheinlichkeit $\varepsilon_n = \varepsilon = \frac{1}{4}$ ausgehen, ist es weit weniger teuer, zu recht kleinen Fehlerwahrscheinlichkeiten zu kommen: $1{,}39 \cdot 20/(1/2)^2 \approx 111$ Wiederholungen genügen, um Fehlerwahrscheinlichkeit 2^{-20} zu erreichen, und 221 Wiederholungen für eine Fehlerwahrscheinlichkeit von $2^{-40} < 10^{-12}$.

Ermittlung des Mehrheitswertes in Linearzeit. Wenn man Algorithmus 3.3.3 naiv implementiert, muss man die Resultate $r_1, \ldots, r_c$ aufbewahren und dann ein Ergebnis ermitteln, das häufiger als $\frac{c}{2}$-mal vorkommt – falls ein solches existiert. Auf den ersten Blick vermutet man, dass hier ein komplizierterer Zähl- oder Sortieraufwand nötig ist. Mit einem kleinen Trick lässt sich dies vermeiden: man muss nur ein Resultat speichern, und $c - 1$ Identitätsvergleiche zwischen Resultaten durchführen. Wir benötigen ein Register `R`, das Werte aus Z speichern kann, und einen Zähler `count`. Das Verfahren ist als Algorithmus 3.3.5 angegeben.

Solange `count` einen positiven Wert enthält, beharrt der Algorithmus auf dem gegenwärtigen Inhalt des Registers `R`. Falls dieses Resultat erneut eintrifft, gilt es als noch besser bestätigt, und `count` wird erhöht. Falls ein Ergebnis eintrifft, das vom Inhalt von `R` verschieden ist, gilt der gespeicherte Wert als weniger stark bestätigt, und `count` wird heruntergezählt. Eine Änderung von `R` erfolgt nur, wenn `count` durch

solche abweichenden Werte wieder auf 0 gefallen ist. Dann beginnt man mit einem neuen Wert von vorn.

Beispiel: Wenn die Resultate der Aufrufe von $\mathcal{A}$ auf x die Werte

$$(22, 22, 11, 33, 11, 11, 22, 33, 44, 11, 11, 22, 11, 11, 11, 33, 11)$$

sind, dann sind die Inhalte von `R` und `count` nach den Runden $i = 0, 1, \ldots, 17$:

–	22	22	22	22	11	11	11	11	44	44	11	11	11	11	11	11	11
0	1	2	1	0	1	2	1	0	1	0	1	0	1	2	3	2	3

Die Ausgabe ist 11, wie gewünscht. Würde man nach $c' = 10$ Runden abbrechen, wäre die Ausgabe 44, die keineswegs besonders häufig ist. Allerdings gibt es unter den Resultaten $r_1, \ldots, r_{10}$ auch keines, das häufiger als 5-mal auftritt.

Proposition 3.3.6
Wenn in $r_1, \ldots, r_c$ ein Wert r^* mehr als $\frac{1}{2}c$-mal vorkommt, dann gibt Algorithmus 3.3.5 diesen Wert r^* aus. (Damit gilt die Analyse aus Satz 3.3.4 ebenfalls für Algorithmus 3.3.5.)

Beweis: Es sei c_i der Stand von `count` ($0 \leq i \leq c$) und s_i der Inhalt von `R` ($1 \leq i \leq c$) nach dem i-ten Schleifendurchlauf. Wir definieren einen ganzzahligen „Pegelstand“ p_i, $i = 0, \ldots, c$: $p_0 = 0$ und

$$p_i := \begin{cases} c_i, & \text{falls } s_i = r^*, \\ -c_i, & \text{falls } s_i \neq r^*, \end{cases}$$

für $1 \leq i \leq c$. (Wenn in `R` der richtige Wert steht, zählen wir den Inhalt von `count` positiv, sonst negativ.)

Wir beobachten nun (vgl. Abb. 3.3.1):

- In jedem Schleifendurchlauf erhöht oder erniedrigt sich p_i um 1.
  (Dies folgt daraus, dass Algorithmus 3.3.5 in Zeilen 5, 8 und 9 den Wert c_i um 1 erhöht oder erniedrigt.)
- Wenn $r_i = r^*$, dann gilt $p_i = p_{i-1} + 1$.
  (1. Fall: $p_{i-1} > 0$. Dann ist $s_{i-1} = r^*$, also wird in Zeile 8 `count` um 1 erhöht.

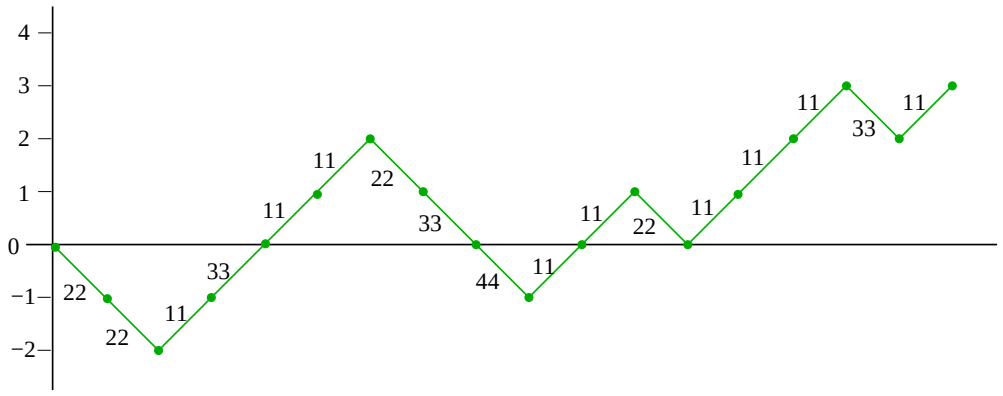


Abbildung 3.3.1: Die Pegelstände p_i, $i = 0, \ldots, 17$, zu den Ergebnissen $r_1, \ldots, r_{17}$ im Beispiel.

2. Fall: $p_{i-1} = 0$. Dann ist $c_i = 1$ und $s_i = r^*$, nach Zeile 5, also $p_i = 1$.
3. Fall: $p_{i-1} < 0$. Dann ist $s_{i-1} \neq r^*$, aber $r_i = r^*$, also ist aktuelles Ergebnis und Inhalt von `R` verschieden. Auch nach Runde i gilt $s_i \neq r^*$. Nach Zeile 9 gilt $c_i = c_{i-1} - 1$, also $p_i = -(c_{i-1} - 1) = p_{i-1} + 1$.

(Man beachte, dass ein negativer Pegelstand auch steigen kann, wenn ein falsches Resultat kommt. Der Zähler/Pegel zählt also keineswegs exakt die Anzahl der korrekten und falschen Ergebnisse oder deren Differenz.) Wir starten mit $p_0 = 0$; in mehr als $\frac{c}{2}$ Runden wächst p_i um 1; p_i fällt um 1 in weniger als $\frac{c}{2}$ Runden. Also ist $p_c > 0$, und das bedeutet, dass $s_c = r^*$ ist und der Wert r^* ausgegeben wird. □

3.3.3 Diskussion: Was bedeuten Fehlerwahrscheinlichkeiten?

Es sei $f\colon I \to Z$ eine Funktion und $\mathcal{A}$ sei ein randomisierter Algorithmus, der f mit einer Fehlerschranke $(\varepsilon_n)_{n\geq 0}$ berechnet. Es sei n fest. Wie sollen wir diese Situation interpretieren?

1. Fehlerrate tolerierbar: Wenn wir $\mathcal{A}$ häufig (auf verschiedenen Eingaben) ausführen, dann erwarten wir, dass im Durchschnitt über diese vielen Aufrufe in nicht viel mehr als etwa $100\varepsilon_n$ Prozent der Fälle ein falsches Ergebnis ausgegeben wird. (Nach dem „Gesetz der großen Zahlen“ wird diese Vorstellung um so sicherer eintreffen, je mehr Wiederholungen stattfinden.) Man kann sich dann überlegen, welcher Prozentsatz

von falschen Resultaten toleriert werden kann, und man kann gegebenenfalls die Fehlerwahrscheinlichkeit durch Wiederholung auf demselben Input (Algorithmus 3.3.5) auf einen akzeptablen Wert senken.

2. Keine Fehler tolerierbar: Es gibt Algorithmen, bei denen das Auftreten eines Fehlers prinzipiell nicht akzeptabel ist. Manche Algorithmen werden auch nur ein einziges Mal, wenn überhaupt, ausgeführt, und es ist entscheidend, dass sie in dieser Situation korrekt funktionieren. Beispielsweise sollte ein Algorithmus, der bei einem seltenen Störfall in einem Kernkraftwerk benutzt wird, sicher funktionieren und nicht eine spürbare Versagenswahrscheinlichkeit haben. Ebenso sollten Algorithmen, die (auch wiederholt) an kritischer Stelle zur Steuerung von Flugzeugen oder bei Operationsrobotern eingesetzt werden, keine Versagenswahrscheinlichkeit einer Größe aufweisen, die erwarten lässt, dass ein solcher Fehler mitunter auftritt. Hier muss man also so vorgehen, dass man durch Wiederholung auf demselben Input (Algorithmus 3.3.5) die Fehlerwahrscheinlichkeit ε_n auf einen extrem kleinen Wert senkt. Zum Beispiel kann man ein Monte-Carlo-Verfahren mit einseitigem Fehler, das eigentlich Fehlerwahrscheinlichkeit $\varepsilon \leq \frac{1}{4}$ hat, durch nur 40-fache Wiederholung auf eine Fehlerwahrscheinlichkeit von $\varepsilon' = 2^{-80} \approx 10^{-24}$ bringen. Diese Fehlerwahrscheinlichkeit ist viel kleiner als die Wahrscheinlichkeit eines Hardwareversagens oder eines Fehlers in der Berechnung, der durch andere Einflüsse verursacht wird; sie kann im Zusammenhang mit dem Einsatz realer Rechner und realer technischer Geräte, die vielfältige andere Fehlerquellen haben, auf jeden Fall vernachlässigt werden.

Allerdings ist hier eine **Warnung** angebracht: Bei Verwendung randomisierter Algorithmen in kritischen Bereichen ist immer zu bedenken, dass die verwendeten Zufallszahlen normalerweise der Annahme der idealen Zufälligkeit nicht entsprechen, sondern von „Pseudo-Zufallszahlen-Generatoren“ bereitgestellt werden. Die Annahme, dass sich solche Zahlen wie Zufallszahlen verhalten, muss also ebenfalls immer kritisch hinterfragt werden. In dieser Vorlesung vernachlässigen wir diesen Aspekt.

3.4 Laufzeitkappung

Ziel dieses Abschnittes ist klarzumachen, dass man sich bei der Betrachtung von fehlerbehafteten Algorithmen (insbesondere Monte-Carlo-Algorithmen) im Wesentlichen auf worst-case-Laufzeitschranken beschränken kann – sie sind nicht viel schlechter als erwartete Laufzeiten. Einige technische Details sind dabei aber zu beachten.

Gegeben sei also ein randomisierter Algorithmus $\mathcal{A}$ für eine Funktion f. Die Idee

ist, die Berechnung von $\mathcal{A}$ auf einer Eingabe x abzubrechen, sobald eine gewisse Zeitschranke $s(x)$ überschritten wird. Dann wird eine beliebige Ausgabe ausgegeben. Dieses Abbrechen führt zu einer neuen Quelle für falsche Ausgaben, zusätzlich zu der, die schon in $\mathcal{A}$ liegt. Wir müssen darauf achten, dass diese neue Fehlerquelle die Fehlerwahrscheinlichkeit nicht allzu sehr erhöht.

Eine sehr naheliegende Abbruchschranke wäre $c \cdot \bar{t}_{\mathcal{A}}(x)$, für eine Konstante c. Eine kleine technische Schwierigkeit ist, dass diese Zahl normalerweise nicht bekannt und nicht leicht zu berechnen ist.

Definition 3.4.1
Eine Funktion $s\colon I \to \mathbb{N}$ heißt eine *konstruierbare Schätzfunktion* für $\bar{t}_{\mathcal{A}}(x)$, wenn $s(x)$ in Zeit $t_s(x) = O(s(x))$ berechenbar ist und $\bar{t}_{\mathcal{A}}(x) \leq s(x)$ ist für alle $x \in I$.

Die worst-case-Laufzeitschranken hängen direkt von der Schätzfunktion ab; wir sollten uns also bemühen, möglichst genaue Schätzfunktionen zu entwickeln. Dies bedeutet, dass wir $\mathcal{A}$ möglichst genau analysieren sollten. Der neue Algorithmus $\mathcal{A}'$ benutzt eine (noch zu diskutierende) Konstante $c \geq 1$.

Algorithmus 3.4.2 ***Laufzeitkappung allgemein***

INPUT: x

METHODE:
1 berechne $t := s(x)$;
2 Lasse $\mathcal{A}$ auf x für maximal $c \cdot t$ RRAM-Schritte laufen;
3 **falls** Berechnung endet: Ausgabe $\mathcal{A}(x)$;
4 **falls** Berechnung nicht endet: Ausgabe 0.

Analyse: 1. Laufzeit: Die Berechnung von $s(x)$ kostet Zeit $O(t_s(x))$; eine durch einen Schrittzähler kontrollierte Berechnung der RRAM, die spätestens nach $c \cdot s(x)$ Schritten abgebrochen wird, kostet Zeit $O(s(x))$. Als worst-case-Zeitschranke erhalten wir $O(t_s(x) + s(x)) = O(s(x))$, als Schranke für die erwartete Laufzeit: $\bar{t}_{\mathcal{A}'}(x) = O(t_s(x) + \bar{t}_{\mathcal{A}}(x))$.

2. Fehlerwahrscheinlichkeit: Wenn $\mathcal{A}'$ auf x das falsche Resultat liefert, dann hat entweder $\mathcal{A}$ bis zum Ende gerechnet, aber mit dem falschen Ergebnis, oder $\mathcal{A}$ ist nicht fertig geworden. Mit der Vereinigungsschranke und der Markov-Ungleichung erhalten

wir:

$$\begin{aligned}
\mathrm{err}_{\mathcal{A}',f}(x) &= \mathbf{Pr}(\mathcal{A}'(x) \neq f(x)) \\
&\leq \mathbf{Pr}(\mathcal{A}(x) \neq f(x)) + \mathbf{Pr}(t_{\mathcal{A}}(x) \geq c \cdot s(x)) \\
&\leq \mathrm{err}_{\mathcal{A},f}(x) + \frac{\mathbf{E}(t_{\mathcal{A}}(x))}{c \cdot s(x)} \\
&\leq \mathrm{err}_{\mathcal{A},f}(x) + \frac{\mathbf{E}(t_{\mathcal{A}}(x))}{c \cdot \mathbf{E}(t_{\mathcal{A}}(x))} \\
&= \mathrm{err}_{\mathcal{A},f}(x) + c^{-1}.
\end{aligned}$$

Im Fall eines Monte-Carlo-Algorithmus $\mathcal{A}$ mit Fehlerschranke $(\varepsilon_n)_{n\geq 0}$ und einer „Wunsch-Fehlerschranke“ $(\varepsilon'_n)_{n\geq 0}$ mit $\varepsilon_n < \varepsilon'_n < \frac{1}{2}$ sollten wir $c_n = (\varepsilon'_n - \varepsilon_n)^{-1}$ wählen; dann ergibt sich $\mathrm{err}_{\mathcal{A}',f}(x) \leq \varepsilon'_n$. Damit die Laufzeit nicht zu groß wird, sollte zwischen ε_n und $\frac{1}{2}$ kein zu kleiner Abstand sein – gegebenenfalls sollte man also *vor* dem Schritt der Laufzeitkappung eine Wahrscheinlichkeitsverbesserung durchführen.

Im Fall eines Monte-Carlo-Algorithmus $\mathcal{A}$ mit einseitigem Fehler und konstanter Fehlerschranke $\varepsilon < 1$ kann man analog vorgehen. Wichtig ist hier, dass die Ausgabe im Abbruchfall 0 sein muss, so dass für Inputs x mit $f(x) = 0$ das richtige Resultat erscheint, selbst wenn die Berechnung abgebrochen wird.

3.5 Las-Vegas-Algorithmen und selbstverifizierende Algorithmen

Grundsätzlich betrachtet man bei randomisierten Algorithmen, die keine Fehler machen, also Las-Vegas-Algorithmen, die erwartete Rechenzeit. Es gibt auch eine Variante von fehlerfreien Algorithmen, die eine worst-case-Laufzeitschranke haben.

Definition 3.5.1
Ein Algorithmus $\mathcal{A} = (M, I, Z \cup \{\text{„?“}\}, \text{IN}, \text{OUT})$ heißt ein **selbstverifizierender (s.v.) Algorithmus** für $f\colon I \to Z$, wenn $\mathbf{Pr}(\mathcal{A}(x) \in \{f(x), \text{„?“}\}) = 1$ gilt; dabei ist „?“ $\notin Z$.

Ein selbstverifizierender Algorithmus gibt entweder das richtige Resultat aus oder teilt über die Ausgabe „?“ mit, wenn es ihm nicht gelungen ist, das korrekte Ergebnis zu berechnen. (Die Ausgabe „?“ kann man als „Ich weiß nicht“ lesen.)

Ziel dieses Abschnitts ist es zu zeigen, dass s.v. Algorithmen mit Versagenswahrscheinlichkeit < 1 und Las-Vegas-Algorithmen praktisch dasselbe sind. Laufzeitkappung führt uns von Las-Vegas-Algorithmen zu s.v. Algorithmen; Wiederholung macht aus s.v. Algorithmen wieder Las-Vegas-Algorithmen. Sei zunächst $\mathcal{A}$ ein Las-Vegas-Algorithmus mit einer konstruierbaren Schätzfunktion $s(x)$ für die erwartete Laufzeit $\bar{t}_{\mathcal{A}}(x)$. Die Zahl $c \geq 1$ sei eine Konstante. Folgendes ist Algorithmus $\mathcal{A}'$.

Algorithmus 3.5.2 ***Laufzeitkappung Las Vegas***

INPUT: x

METHODE:
1 berechne $t := s(x)$;
2 Lasse $\mathcal{A}$ auf x für maximal $c \cdot t$ RRAM-Schritte laufen;
3 **falls** Berechnung endet: Ausgabe $\mathcal{A}(x)$;
4 **falls** Berechnung nicht endet: Ausgabe „?“.

Analyse: 1. Laufzeit: Wie bei den Monte-Carlo-Algorithmen erhalten wir als worst-case-Zeitschranke $O(t_s(x) + s(x)) = O(s(x))$, als Schranke für die erwartete Laufzeit: $\bar{t}_{\mathcal{A}'}(x) = O(t_s(x) + \bar{t}_{\mathcal{A}}(x))$.

2. Fehlerwahrscheinlichkeit: Wenn $\mathcal{A}'$ auf x das falsche Resultat liefert, dann ist $\mathcal{A}$ nicht fertig geworden. Mit der Markov-Ungleichung erhalten wir:

$$\mathrm{err}_{\mathcal{A}',f}(x) = \mathbf{Pr}(t_{\mathcal{A}}(x) \geq c \cdot s(x)) \leq \frac{\mathbf{E}(t_{\mathcal{A}}(x))}{c \cdot s(x)} \leq c^{-1}.$$

Wenn wir also bei $\mathcal{A}'$ eine Fehlerwahrscheinlichkeit $\varepsilon < 1$ tolerieren wollen, sollten wir $c \geq 1/\varepsilon$ wählen.

Nun kommen wir zur Umkehrung. Gegeben sei also ein s.v. Algorithmus für die Funktion f. Wir nehmen an, die Laufzeit sei $\leq t(x)$ im schlechtesten Fall. Es liegt nahe, im Fall des Ergebnisses „?“ einfach den Algorithmus erneut zu starten. Dies liefert den folgenden Algorithmus $\mathcal{A}'$.

Algorithmus 3.5.3 ***Wiederholung s.v. Algorithmus***

INPUT: x

METHODE:

```
1   repeat
2     r := A(x)
3   until r ≠ „?“;
4   return r.
```

Analyse: 1. Laufzeit (Version A): Wir betrachten einen Input x und untersuchen die Anzahl Y der Schleifendurchläufe auf x. Definiere („Versagenswahrscheinlichkeit“):

$$\varepsilon_x := \mathbf{Pr}(\mathcal{A} \text{ auf } x \text{ liefert „?“}).$$

Dann gilt für $j \geq 1$:

$$\mathbf{Pr}(Y \geq j) = \mathbf{Pr}(\text{die ersten } j-1 \text{ Aufrufe liefern „?“}) = \varepsilon_x^{j-1}.$$

Mit Fakt 2.2.9 erhalten wir:

$$\mathbf{E}(Y) = \sum_{j \geq 1} \mathbf{Pr}(Y \geq j) = \sum_{j \geq 1} \varepsilon_x^{j-1} = \sum_{j \geq 0} \varepsilon_x^{j} = \frac{1}{1-\varepsilon_x}.$$

Da jeder Schleifendurchlauf Zeit $t(x) + O(1)$ kostet, ist die erwartete Laufzeit

$$\bar{t}_{\mathcal{A}'}(x) = O\Big(\frac{t(x)}{1-\varepsilon_x}\Big).$$

Diese Zeitschranke gilt, ohne dass man $t(x)$ oder ε_x kennen muss. Ein spezieller Fall liegt vor, wenn es ein $\varepsilon < 1$ gibt derart, dass $\varepsilon_x \leq \varepsilon$ für alle $x \in I$ gilt. In diesem Fall erhalten wir die Laufzeitschranke $\bar{t}_{\mathcal{A}'}(x) = O(t(x)/(1-\varepsilon))$: die erwartete Laufzeit von $\mathcal{A}'$ ist nur um einen konstanten Faktor schlechter als die (worst-case-)Laufzeit von $\mathcal{A}$.

1. Laufzeit (Version B): Eine noch präzisere (und auf keinen Fall schlechtere) Schranke liefert die folgende elegante Analyse. Wir betrachten zwei *bedingte* erwartete Zeitschranken, je eine für den Fall, dass $\mathcal{A}$ erfolgreich ist und den Fall, dass $\mathcal{A}$ das Ergebnis „?“ liefert:

- $\bar{t}(x) = \mathbf{E}(t_{\mathcal{A}}(x) \mid \mathcal{A}(x) = f(x))$;

- $\overline{s}(x) = \mathbf{E}(t_{\mathcal{A}}(x) \mid \mathcal{A}(x) = \text{„?“})$.

Wir nehmen dabei an, dass der Zusatzaufwand für die Schleifenorganisation in $\mathcal{A}'$ schon in die Laufzeit für einen Durchlauf von $\mathcal{A}$, erfolgreich oder erfolglos, mit einberechnet wurde. Wir berechnen nun $\mathbf{E}(t_{\mathcal{A}'}(x))$ wie folgt: Beim ersten Aufruf von $\mathcal{A}$ gibt es zwei Möglichkeiten: entweder $\mathcal{A}(x) = f(x)$ (Wahrscheinlichkeit $1 - \varepsilon_x$) – dann kostet der Aufruf erwartete Zeit $\overline{t}(x)$; oder $\mathcal{A}(x) = \text{„?“}$ (Wahrscheinlichkeit ε_x) – dann haben wir erwartete Zeit $\overline{s}(x)$ für den ersten Aufruf von $\mathcal{A}$ und zusätzlich die erwarteten Kosten für $\mathcal{A}'$, weil wir ja einfach die Schleife erneut starten. Das liefert:

$$\mathbf{E}(t_{\mathcal{A}'}(x)) = (1 - \varepsilon_x)\overline{t}(x) + \varepsilon_x(\overline{s}(x) + \mathbf{E}(t_{\mathcal{A}'}(x))).$$

Interessant ist, dass die unbekannte erwartete Zeit rechts wieder auftritt, aber das macht nichts, da wir die Gleichung nach $\mathbf{E}(t_{\mathcal{A}'}(x))$ auflösen können. Dies ergibt:

$$\mathbf{E}(t_{\mathcal{A}'}(x)) = \overline{t}(x) + \frac{\varepsilon_x \overline{s}(x)}{1 - \varepsilon_x}.$$

Auch diese Gleichung gilt unabhängig davon, ob wir die erwarteten Laufzeiten bei $\mathcal{A}$ kennen. Sie gibt ein recht natürliches Ergebnis: Wie wir in der ersten Laufzeitanalyse (Version A) berechnet haben, erwarten wir insgesamt $\frac{1}{1-\varepsilon_x}$ Ausführungen von $\mathcal{A}$. Davon ist eine (die letzte) erfolgreich (Ergebnis $\neq \text{„?“}$), die anderen nicht (Ergebnis „?“), und von diesen erwarten wir $\frac{1}{1-\varepsilon_x} - 1 = \frac{\varepsilon_x}{1-\varepsilon_x}$ viele.

2. Fehlerwahrscheinlichkeit: Es sei B_j das Ereignis, dass $\mathcal{A}'$ nach genau j Schleifendurchläufen anhält und das korrekte Ergebnis $f(x)$ liefert. Dann gilt:

$$\mathbf{Pr}(B_j) = \mathbf{Pr}(\text{die ersten } j-1 \text{ Versuche liefern „?“} \wedge \text{der } j\text{-te Versuch liefert } f(x)).$$

Wegen der Unabhängigkeit der Versuche und der Definition von selbstverifizierenden Algorithmen ist $\mathbf{Pr}(B_j) = \varepsilon_x^{j-1}(1-\varepsilon_x)$. Die Ereignisse B_j sind disjunkt, daher können wir addieren:

$$\mathbf{Pr}(\mathcal{A}' \text{ liefert Resultat } f(x)) = \mathbf{Pr}\Big(\bigcup_{j\geq 1} B_j\Big) = \sum_{j\geq 1} \mathbf{Pr}(B_j) = \sum_{j\geq 1} \varepsilon_x^{j-1}(1-\varepsilon_x) = 1.$$

Das heißt, dass $\mathcal{A}'$ ein Las-Vegas-Algorithmus ist.

A Registermaschinen

In diesem Abschnitt stellen wir das Modell der Registermaschine (RAM – **r**andom **a**ccess **m**achine – Maschine mit wahlfreiem Speicherzugriff) vor.

Registermaschinen operieren auf natürlichen Zahlen bzw. endlichen Folgen von Zahlen. Ihre Struktur stellt sie in die Nähe des von-Neumann-Rechenmodells. Sie haben Programme mit Speicher-, Rechen- und Sprungbefehlen und einen Speicher mit Speicherzellen, auf die mit direkter und indirekter Adressierung zugegriffen werden kann. Registermaschinen unterscheiden sich von GOTO-Programmen (Vorlesung „Automaten, Sprachen und Komplexität (ASK)“) hauptsächlich dadurch, dass es potenziell unendlich viele Speicherzellen und indirekte Adressierung gibt.

Eine *Registermaschine* (RAM) hat einerseits einen Speicher, der aus unendlich vielen *Speicherzellen* R_0, R_1, R_2, ... besteht, die man traditionell *Register* nennt. Jedes Register kann eine natürliche Zahl speichern. $\langle R_i \rangle$ bezeichnet die in R_i gespeicherte Zahl. Wir interessieren uns nur für Speicherzustände, in denen alle bis auf endlich viele Register den Inhalt 0 haben. Andererseits hat eine RAM ein Programm, das einem Maschinenprogramm in einer simplen Maschinensprache gleicht. Formal ist das Programm eine Folge $B_0, B_1, \ldots, B_{l-1}$ von l *Befehlen*, die aus einem Befehlsvorrat stammen, der Speicher- oder Kopierbefehle, arithmetische Befehle und bedingte und unbedingte Sprungbefehle enthält. B_k heißt „die k-te Programmzeile“; man stellt sich die Befehle also in l Zeilen angeordnet vor. Um die Arbeitsweise der RAM zu beschreiben, geben wir ihr noch ein zusätzliches Register, den *Befehlszähler* BZ, das ebenfalls natürliche Zahlen speichern kann. Der Inhalt von BZ wird mit $\langle \mathrm{BZ} \rangle$ bezeichnet. In Abb. A.0.2 ist der Aufbau einer RAM schematisch wiedergegeben.

RAMs mit verschiedenen Programmen werden als verschiedene Maschinen angesehen. Hat man eine RAM M (d. h. ein Programm) und stehen in BZ und $R_0, \ldots, R_m$ irgendwelche Zahlen, und in $R_{m+1}, R_{m+2}, \ldots$ Nullen, kann M einen Schritt ausführen:

> Falls $0 \leq \langle \mathrm{BZ} \rangle < l$, wird Befehl $B_{\langle \mathrm{BZ} \rangle}$ ausgeführt (dies verändert $\langle \mathrm{BZ} \rangle$ und eventuell den Inhalt eines Registers). Ist $\langle \mathrm{BZ} \rangle \geq l$, passiert nichts: die RAM hält.

Dieser Vorgang kann natürlich iteriert werden, bis möglicherweise schließlich ein Befehlszählerinhalt $\geq l$ erreicht wird. In der Tabelle auf Seite 30 beschreiben wir den

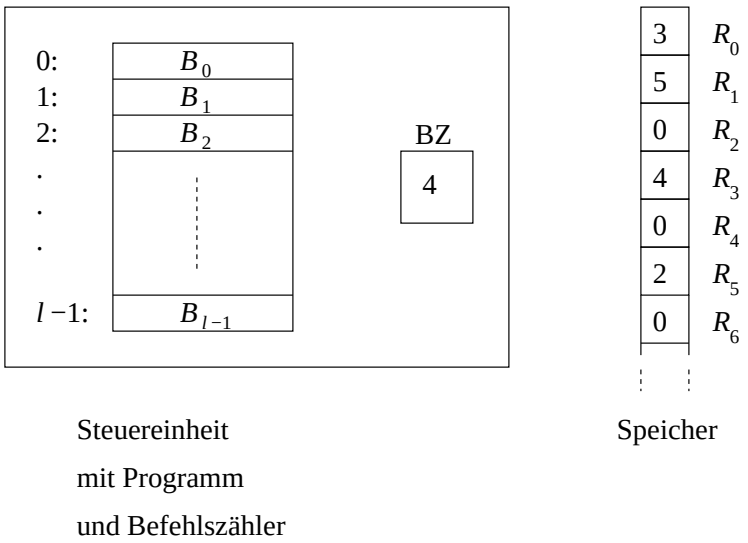


Abbildung A.0.2: Schema einer Registermaschine

Befehlssatz einer RAM. Es handelt sich um sogenannte Dreiadressbefehle: Operanden werden aus zwei Registern geholt, das Resultat einer Operation in einem dritten Register gespeichert. Nicht-Sprungbefehle führen dazu, dass der Befehlszähler um 1 erhöht wird. In der Tabelle steht das Zeichen „:=“ für eine Zuweisung: „$R_i := p$“ für einen Index $i \in \mathbb{N}$ und eine Zahl $p \in \mathbb{N}$ bewirkt, dass nachher das Register R_i die Zahl p enthält. Analog sind Zuweisungen „BZ $:= m$“ zu interpretieren. Bei der Subtraktion werden negative Resultate durch 0 ersetzt; die Division ist die ganzzahlige Division ohne Rest. Division durch 0 führt zum Anhalten.

Mitunter betrachtet man RAMs mit eingeschränktem Befehlsvorrat, z. B. $\{+, -\}$-RAMs, bei denen die $*$- und $\div$-Befehle fehlen. Da man Multiplikation und Division durch Teilprogramme ersetzen kann, die nur Addition und Subtraktion benutzen, bedeutet dies keine prinzipielle Einschränkung; allerdings kann sich die Anzahl der für eine Berechnung nötigen Schritte erhöhen.

Befehlszeile	**Einschränkungen, Beschreibung**	**Wirkung**
Speicherbefehle:		
$R_i \leftarrow R_j$	$i, j \in \mathbb{N}$ ($*$ Register kopieren mit direkter Adressierung $*$)	$R_i := \langle R_j \rangle$; BZ $:= \langle$BZ$\rangle + 1$;
$R_{R_i} \leftarrow R_j$	$i, j \in \mathbb{N}$ ($*$ Register kopieren; Ziel indirekt adressiert $*$)	$R_{\langle R_i \rangle} := \langle R_j \rangle$; BZ $:= \langle$BZ$\rangle + 1$;
$R_i \leftarrow R_{R_j}$	$i, j \in \mathbb{N}$ ($*$ Register kopieren; Quelle indirekt adressiert $*$)	$R_i := \langle R_{\langle R_j \rangle} \rangle$; BZ $:= \langle$BZ$\rangle + 1$;
Arithmetische Befehle:		
$R_i \leftarrow k$	$i, k \in \mathbb{N}$ ($*$ Konstante laden $*$)	$R_i := k$; BZ $:= \langle$BZ$\rangle + 1$;
$R_i \leftarrow R_j + R_k$	$i, j, k \in \mathbb{N}$ ($*$ holen, addieren, speichern $*$)	$R_i := \langle R_j \rangle + \langle R_k \rangle$; BZ $:= \langle$BZ$\rangle + 1$;
$R_i \leftarrow R_j - R_k$	$i, j, k \in \mathbb{N}$ ($*$ holen, subtrahieren, speichern; negatives Resultat durch 0 ersetzen $*$)	$R_i := \max\{0, \langle R_j \rangle - \langle R_k \rangle\}$; BZ $:= \langle$BZ$\rangle + 1$;
$R_i \leftarrow R_j * R_k$	$i, j, k \in \mathbb{N}$ ($*$ holen, multiplizieren, speichern $*$)	$R_i := \langle R_j \rangle \cdot \langle R_k \rangle$; BZ $:= \langle$BZ$\rangle + 1$;
$R_i \leftarrow R_j \div R_k$	$i, j, k \in \mathbb{N}$ ($*$ holen, dividieren, speichern $*$) ($*$ Fehlerhalt $*$)	if $\langle R_k \rangle > 0$ then $\{R_i := \langle R_j \rangle$ div $\langle R_k \rangle$; BZ $:= \langle$BZ$\rangle + 1\}$ else BZ $:= l$
Sprungbefehle:		
goto m	$m \in \mathbb{N}$ ($*$ unbedingter Sprung $*$)	BZ $:= m$
if $(R_i = 0)$ goto m	$i, m \in \mathbb{N}$ ($*$ bedingter Sprung $*$)	BZ $:= \begin{cases} m, \text{ falls } \langle R_i \rangle = 0; \\ \langle \text{BZ} \rangle + 1, \text{ sonst.} \end{cases}$
if $(R_i > 0)$ goto m	$i, m \in \mathbb{N}$ ($*$ bedingter Sprung $*$)	BZ $:= \begin{cases} m, \text{ falls } \langle R_i \rangle > 0; \\ \langle \text{BZ} \rangle + 1, \text{sonst.} \end{cases}$
STOP:		
($*$ bedingte oder unbedingte Sprünge „goto m“ mit Sprungziel $m \geq l$ wirken wie bedingte oder unbedingte STOP-Befehle. Die RAM hält auch, wenn der Befehl in Zeile $l - 1$ ausgeführt wird und es sich nicht um einen Sprungbefehl handelt. $*$)		

Konfigurationen: Der „innere Zustand“ einer Registermaschine M lässt sich durch die Angabe des Befehlszählerstandes und der Registerinhalte vollständig angeben. Wir definieren: Eine *Konfiguration* k ist ein Paar (z, α) aus einer natürlichen Zahl z und einer Funktion $\alpha\colon \mathbb{N} \to \mathbb{N}$, die fast überall den Wert 0 hat, d. h., es existiert n_α mit der Eigenschaft, dass für $i > n_\alpha$ stets $\alpha(i) = 0$ gilt. (Die Menge $\mathcal{K}$ aller Konfigurationen ist abzählbar unendlich. Sie hängt nicht von M ab.)

Schritt, Nachfolgekonfiguration, Haltekonfiguration: Sei eine RAM M (d. h. ein Programm) gegeben. Eine Konfiguration k' heißt *Nachfolgekonfiguration* von k, in Zeichen $k \vdash_M k'$ oder $k \vdash k'$, wenn ein Schritt von M in der oben beschriebenen Weise von k zu k' führt. Jede Konfiguration hat höchstens eine Nachfolgekonfiguration. Es kann sein, dass es keine Nachfolgekonfiguration von k gibt, weil in k eine Division durch 0 auszuführen wäre („Fehlerkonfiguration“). Die andere, „normale“ Möglichkeit ist, dass $k = (z, \alpha)$ für ein $z \geq l$ ist, wo l die Zeilenanzahl von M ist. Solche Konfigurationen heißen *Haltekonfigurationen*.

Startkonfiguration: Eingaben für Registermaschinen sind grundsätzlich n-Tupel $(a_0, \ldots, a_{n-1})$ von natürlichen Zahlen. Einem solchen n-Tupel entspricht die folgende *Startkonfiguration* (andere Konventionen sind möglich): Ist die Eingabe $(a_0, \ldots, a_{n-1}) \in \mathbb{N}^n$, so ist die Startkonfiguration gleich $(0, \alpha)$ mit

$\alpha(0) = n$,
$\alpha(2i + 1) = a_i$, für $0 \leq i \leq n - 1$,
$\alpha(j) = 0$ für alle anderen j.

Damit hat man die Register R_2, R_4, ... zur freien Verfügung. Der Befehlszähler hat anfangs den Inhalt 0. Für die Ausgabe wird die entsprechende (umgekehrte) Konvention benutzt: Als Resultat nach dem Anhalten gilt das Zahlentupel

$$(\alpha(1), \alpha(3), \ldots, \alpha(2\alpha(0) - 1)).$$

Gegeben eine Startkonfiguration $k_0 = (0, \alpha)$ und eine RAM M, gibt es eine eindeutig bestimmte Konfigurationenfolge

$$k_0 \vdash_M k_1 \vdash_M k_2 \vdash_M \cdots,$$

die entweder endlich oder unendlich ist. Diese stellt die *Berechnung* von M auf der durch die Startkonfiguration gegebenen Eingabe dar. Diese Berechnung ist die formale Entsprechung der folgenden informalen Beschreibung der Arbeitsweise einer RAM:

① man schreibt $a_0, \ldots, a_{n-1}$ gemäß der Eingabekonvention in die Register von M und setzt BZ auf 0.

② while $0 \leq \langle \mathrm{BZ} \rangle < l$ do
„führe Befehl $B_{\langle \mathrm{BZ} \rangle}$ aus“
(mit der Wirkung wie in der Tabelle angegeben)

③ falls und sobald in ② eine Situation mit $\langle \mathrm{BZ} \rangle \geq l$ erreicht wird, wird aus den Registerinhalten gemäß der Ausgabekonvention das Resultat abgelesen.

Beispiel A.0.1
Wir wollen aus $a_0, \ldots, a_{n-1}$ die Summe $a_0 + \cdots + a_{n-1}$ und das Produkt $a_0 \cdots a_{n-1}$ berechnen. Die Register mit geraden Indizes werden verwendet wie folgt:
– in Register R_2 wird von n nach 0 heruntergezählt,
– in R_4 wird die Summe und in R_6 das Produkt akkumuliert,
– R_8 enthält den Index des nächsten zu verarbeitenden Inputregisters,
– in R_{10} wird der Inhalt dieses Registers zwischengespeichert,
– R_{12} enthält die Konstante 1,
– R_{14} die Konstante 2.

Zeile	Befehl	Kommentar
0	$R_{12} \leftarrow 1$	Konstante
1	$R_{14} \leftarrow 2$	laden
2	$R_2 \leftarrow R_0$	n ins Zählregister
3	$R_4 \leftarrow 0$	Initialisiere Teilsumme
4	$R_6 \leftarrow 1$	und Teilprodukt
5	$R_8 \leftarrow 1$	Indexregister auf R_1 stellen
6	if $(R_2 = 0)$ goto 13	Schleife: Zeilen 6–12
7	$R_{10} \leftarrow R_{R_8}$	Operanden holen
8	$R_4 \leftarrow R_4 + R_{10}$	addieren
9	$R_6 \leftarrow R_6 * R_{10}$	multiplizieren
10	$R_8 \leftarrow R_8 + R_{14}$	Indexregister um 2 erhöhen
11	$R_2 \leftarrow R_2 - R_{12}$	Zähler dekrementieren
12	goto 6	zum Schleifentest
13	$R_1 \leftarrow R_4$	Ausgabeformat
14	$R_3 \leftarrow R_6$	herstellen:
15	$R_0 \leftarrow 2$	2 Ausgabewerte

Beispiel A.0.2
Berechnung von $a_0^{a_1}$. Dabei ist nur das Verhalten auf zweistelligen Eingaben (a_0, a_1) interessant.

(Naive) Idee: a_1-faches Multiplizieren von a_0 mit sich selbst. Realisiert wird dies durch eine Schleife, in der R_3 in Einerschritten von a_1 bis 0 heruntergezählt wird. Das Programm für M sieht folgendermaßen aus:

Zeile	Befehl	Kommentar
0	$R_2 \leftarrow 1$	Konstante 1
1	$R_4 \leftarrow 1$	a_0^0
2	if $(R_3 = 0)$ goto 6	Zeilen 2–5:
3	$R_4 \leftarrow R_4 * R_1$	Schleife
4	$R_3 \leftarrow R_3 - R_2$	
5	goto 2	
6	$R_1 \leftarrow R_4$	Resultatformat
7	$R_0 \leftarrow 1$	herstellen

Auf der Eingabe $(a_0, a_1) = (6, 3)$ läuft das Programm wie folgt ab. Die Zeile zu Schritt t repräsentiert dabei den gesamten Zustand (die „Konfiguration“ der RAM) nach Schritt $t = 0, 1, 2, \ldots$. Schritt 0 entspricht dem Anfangszustand, nach der Initialisierung.

Schritt-Nr.	R_0	R_1	R_2	R_3	R_4	$\langle \mathrm{BZ} \rangle$
0	2	6	0	3	0	0
1	2	6	1	3	0	1
2	2	6	1	3	1	2
3	2	6	1	3	1	3
4	2	6	1	3	6	4
5	2	6	1	2	6	5
6	2	6	1	2	6	2
7	2	6	1	2	6	3
8	2	6	1	2	36	4
9	2	6	1	1	36	5
10	2	6	1	1	36	2
11	2	6	1	1	36	3
12	2	6	1	1	216	4
13	2	6	1	0	216	5
14	2	6	1	0	216	2
15	2	6	1	0	216	6
16	2	216	1	0	216	7
17	1	216	1	0	216	8

Als *Rechenzeit* der RAM M auf Eingabe $(a_0, \ldots, a_{n-1})$ sehen wir die Anzahl der Schritte an, die M auf dieser Eingabe ausführt. Diese Rechenzeit kann auch „unendlich“ sein, wenn die Berechnung nicht anhält. Man sollte bemerken, dass dieses „uniforme Kostenmaß“ nicht ganz fair ist, wenn durch Multiplikationen in wenigen Rechenschritten sehr große Zahlen erzeugt werden. Man beschränkt daher oft die erlaubte Anzahl der Bits der in den Registern gespeicherten Zahlen auf $O(\log(|x|))$, wobei $|x|$ die „(Darstellungs-)Größe“ von Input x ist.

Bemerkung A.0.3
Wir betrachten (idealisierte) Programme in einer Standard-Programmiersprache – der Eindeutigkeit halber nehmen wir etwa Java. Wir können uns alle Zahltypen wie ganze Zahlen oder reelle Zahlen durch Tupel von natürlichen Zahlen repräsentiert denken, Characters ebenfalls durch Zahlen, Zeiger durch Arrayindizes. Übersetzen wir das Programm mittels eines Compilers in Maschinensprache, ergibt sich ein Programm, das relativ leicht in ein RAM-Programm zu transformieren ist. Wir stellen daher fest: Mit Standard-Programmiersprachen berechenbare Funktionen sind auch auf einer RAM berechenbar. – Umgekehrt kann man sich fragen, ob alle RAM-berechenbaren Funktionen von Java-Programmen berechnet werden können. Diese Frage ist dann

zu verneinen, wenn man die Endlichkeit von Rechnern in Betracht zieht, die dazu führt, dass für jedes auf einer festen Maschine ausführbare Java-Programm eine größte darstellbare Zahl und eine Maximalzahl von verwendbaren Registern gegeben ist. Betrachtet man „idealisierte“ Programme und Computer ohne solche Einschränkungen, lässt sich tatsächlich die Äquivalenz beweisen. Wir geben uns damit zufrieden, im RAM-Modell eine Abstraktion zu erkennen, die auf jeden Fall die Fähigkeiten konkreter Rechner und Programmiersprachen umfasst, einschließlich von Rechenzeitbetrachtungen.

4 Randomisierte Suche

Mit diesem Kapitel wenden wir uns dem Hauptteil der Vorlesung zu: Beispielalgorithmen und Analysetechniken. Hier geht es um ein Grundproblem: Das Suchen von Objekten anhand bestimmter Kriterien. Die Suchprobleme sind daher ganz unterschiedlich, und ebenso unterschiedlich sind die algorithmischen Ansätze.

Abschnitt 4.1 betrachtet ein sehr einfach aussehendes Problem: Gegeben ist eine endliche Grundmenge A mit einer Teilmenge B, die aber nur über Tests „ist x in B?“ zugänglich ist. Man soll ein Element von B finden. Der offensichtliche randomisierte Algorithmus, nämlich so lange zufällige Elemente von A zu wählen, bis man auf ein Element von B stößt, benötigt möglicherweise sehr viele Zufallszahlen. Wir finden einen Algorithmus, der fast ebenso schnell ist, aber im Wesentlichen nur konstant viele Elemente von A zufällig wählt.

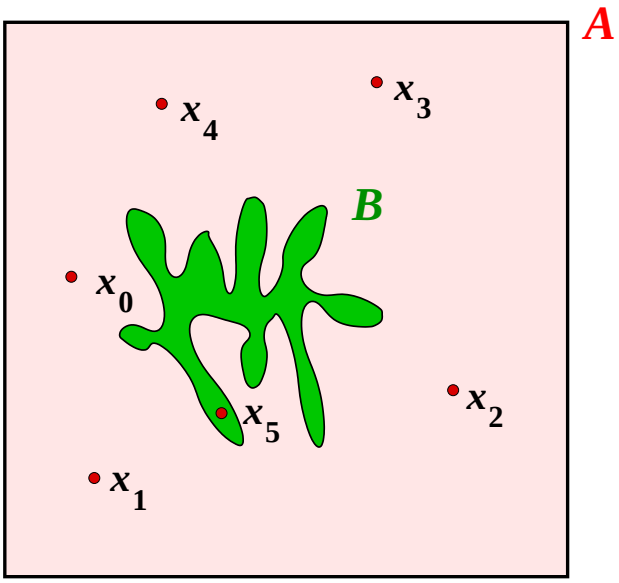
In Abschnitt 4.2 geht es um ein klassisches Problem, nämlich die Implementierung von Wörterbüchern mit linearen Listen. Normalerweise ist die Zeit für eine Suche, eine Einfügung, und eine Löschung $\Theta(n)$, wenn n Elemente gespeichert sind. Wir werden sehen, dass man in einer angeordneten Liste viel schneller suchen kann, wenn man Listenelemente zufällig auswählen kann.

In Abschnitt 4.3 betrachten wir den Algorithmus „Quickselect“, der schon aus der Vorlesung „Algorithmen und Datenstrukturen“ bekannt sein sollte. Wir betrachten hier eine alternative Analyse, die sich an der Analyse von Quicksort aus dem ersten Kapitel anlehnt. (Eine gute Gelegenheit, die Quicksort-Analyse nochmals anzusehen!)

In Abschnitt 4.4 schließlich wird ein anderer Algorithmus zur Ermittlung des Medians einer Folge präsentiert. Dieser verallgemeinert die Idee, ein Pivot zufällig zu wählen, darauf, mehrere zufällige Elemente zu wählen und sie zur sehr effizienten Reduzierung der Problemgröße zu benutzen. Das verbleibende Problem kann dann unter Einhaltung der linearen Zeitschranke einfach durch Sortieren gelöst werden. Das Ergebnis ist ein Algorithmus mit der bestmöglichen Vergleichsanzahl für das Medianproblem.

4.1 Suche mit wenigen Zufallsbits

Wir stellen uns folgendes Grundproblem vor: Gegeben ist eine endliche Menge A der Größe n sowie eine nichtleere Teilmenge $B \subseteq A$.



Dabei haben wir auf die Menge B keinen direkten Zugriff, wie etwa durch eine Auflistung, sondern wir haben nur die folgenden Operationen zur Verfügung:

- für gegebenes $x \in A$ können wir einen Test „ist $x \in B$?“ durchführen;
- wir können alle Elemente der Menge A systematisch aufzählen;
- wir können ein Element von A zufällig wählen.

Die Aufgabe ist, ein Element von B zu finden. Die *Kosten* hierfür ist die Anzahl der durchgeführten Tests „ist $x \in B$?“.

Algorithmus 4.1.1 ***Zufällige Suche***

INPUT: Endliche Menge A mit unbekannter nichtleerer Teilmenge B
METHODE:
1 **repeat**
2 $\quad$ x $\leftarrow$ zufälliges Element von A
3 **until** x $\in B$;
4 **return** x.

Beispiel: Wir wollen im Bereich der ℓ-Bit-Zahlen eine Primzahl finden.

$$A = \{x \in \mathbb{N} \mid 2^{\ell-1} \leq x < 2^{\ell}\}.$$
$$B = \{p \in A \mid p \text{ ist Primzahl}\}.$$

Wie in Kapitel 5 im Detail besprochen wird, gibt es einen sehr effizienten (randomisierten) *Primzahltest*, der es erlaubt, Zahlen $x \in A$ darauf zu testen, ob sie in B sind. Ein Verfahren, effizient (d. h., in Rechenzeit $O(\ell^c)$ für eine Konstante c) eine Primzahl in A zu konstruieren, kennt man dagegen nicht.

Wenn wir die Elemente von A aufzählen und nacheinander testen, könnte es passieren, dass wir zuerst alle Elemente von $A - B$ sehen, dann erst ein Element von B. Dies ist in eigentlich allen Situationen unbefriedigend. Sogar in der angenehmen Situation, in der etwa jedes zweite Element von A zu B gehört, würde man im schlimmsten Fall $\frac{1}{2}|A| + 1$ Tests durchführen, bis man das erste Element von B findet. Attraktiver ist ein randomisiertes Vorgehen, das wir als Nächstes beschreiben (Algorithmus 4.1.1). Man wählt wiederholt ein Element x aus A zufällig; wenn und sobald dieses x in B liegt, stoppt man und gibt x aus. Dieses Verfahren würde in der Situation $|B| \geq \frac{1}{2}|A|$ im Schnitt nur höchstens zwei Versuche brauchen. Wir erkennen aber auch, dass dieser Algorithmus endlos lange läuft, wenn $B = \emptyset$ ist; diesen Fall müssen wir also ausschließen.

Wir analysieren die erwartete Laufzeit, die hier im Wesentlichen durch die Anzahl der Schleifendurchläufe bestimmt ist. Mit

$$\varrho := \frac{|B|}{|A|}$$

(gesprochen: „rho“) bezeichnen wir die *Dichte* von B in A. Seien $X_0, X_1, X_2, \ldots$ die gewählten Elemente (Zufallsobjekte in A), und sei R die Anzahl der Schleifendurchläufe. Die Zufallsvariable R ist offensichtlich geometrisch verteilt mit Parameter ϱ

(siehe Abschnitt 2.5.1). Wir haben also:

$$\mathbf{E}(R) = \frac{1}{\varrho}. \tag{4.1.1}$$

Wir wollen in diesem Abschnitt auch Zufallsexperimente als Ressource betrachten. Um fair zu messen, benutzen wir die Anzahl der Zufallsbits als Maß (nicht die Anzahl der Zufallszahlen, die eine Vielzahl von Bits auf einen Schlag liefern). Um eine Zahl aus A zufällig zu wählen, müssen wir eine $\lceil \log |A| \rceil$-Bit-Zahl (eine Zahl in $\{0, 1, \ldots, |A| - 1\}$) wählen.

Proposition 4.1.2
Zufällige Suche wie in Algorithmus 4.1.1 beschrieben kostet im erwarteten Fall $1/\varrho$ Tests „ist $x \in B$?“; die erwartete Zahl von erzeugten Zufallsbits ist $\lceil \log |A| \rceil / \varrho$.

Wenn die Dichte ϱ klein ist, kann die Zahl der Zufallsbits erheblich sein. Beispielsweise ist im Fall der Suche nach einer ℓ-ziffrigen Primzahl die Dichte $\Theta(1/\ell)$ und damit die erwartete Anzahl von Zufallsbits $\Theta(\ell^2)$.

Wir überlegen im Folgenden, wie wir mit relativ wenigen Zufallsbits fast ebenso effizient ein x in B finden können. Eine etwas allgemeinere Interpretation der Situation ist die eines „*special purpose*-Pseudozufallszahlengenerators“. Eigentlich benötigen wir eine lange Folge $X_0, X_1, \ldots$, von zufälligen Elementen von A. Um Zufallsbits zu sparen, wählen wir einen kurzen zufälligen „*seed*“ (das deutsche „Samenkorn“ ist ungebräuchlich) und erzeugen daraus algorithmisch eine ganze Folge von Elementen von A. Diese Folge ist nicht mehr vollständig zufällig, sondern nur „pseudozufällig“. Im konkreten Fall der Suche nach einem Element von B in A können wir das Verhalten unseres Verfahrens genau analysieren.

Für unsere Konstruktion nehmen wir an, dass A eine algebraische Struktur hat, nämlich dass A ein endlicher Körper ist, zum Beispiel $A = \mathbb{Z}_p$ für eine Primzahl p.[1] Die Idee ist, zwei Elemente r und s in $\mathbb{Z}_p$ zufällig zu wählen und dann die Folge

$$X_0 = r \cdot 0 + s, X_1 = r \cdot 1 + s, X_2 = r \cdot 2 + s, X_3 = r \cdot 3 + s, \ldots \tag{4.1.2}$$

[1] Sollte dies nicht der Fall sein, nummerieren wir die Elemente von A durch, stellen uns also $A = \{0, \ldots, |A| - 1\}$ vor, suchen eine Primzahl $p \in [|A|, 2|A|]$ (eine solche existiert nach einem Satz der Zahlentheorie) und arbeiten mit $A' = \mathbb{Z}_p$. Die neue Dichte $\varrho' = \frac{|B|}{|A'|}$ unterscheidet sich höchstens um den Faktor 2 von ϱ.

(Arithmetik in $\mathbb{Z}_p$) in A zu testen. Eine kleine Komplikation liegt darin, dass $r = 0$ sein könnte. Dann würde man immer wieder nur den Eintrag s testen. Dies fangen wir mit einer Sonderbehandlung ab, die die Elemente von A in der Reihenfolge $s, s+1, s+2, \ldots, p-1, 0, 1, \ldots, s-1$ betrachtet. Ansonsten erhalten wir eine Folge von mehr oder weniger zufälligen Elementen in A. Wenn wir in $\mathbb{Z}_p$ arbeiten, können wir programmiertechnisch die Elemente $X_0, X_1, \ldots$ ohne Multiplikation, nur durch iterierte Addition, erzeugen. Wir erhalten Algorithmus 4.1.3.

Algorithmus 4.1.3 ***Paarweise unabhängige Suche***

INPUT: Menge $A = \mathbb{Z}_p$ (Körper) mit unbekannter, nichtleerer Teilmenge B
METHODE:
1 Wähle r und s aus $\mathbb{Z}_p$ zufällig;
2 Falls $r = 0$, setze $r \leftarrow 1$; // erzwingt kompletten Durchlauf durch A
3 $\mathtt{x} \leftarrow (s - r) \bmod p$;
4 **repeat**
5 $\quad \mathtt{x} \leftarrow (\mathtt{x} + r) \bmod p$;
6 **until** $\mathtt{x} \in B$;
7 **return** x.

Analyse: Korrektheit. Wir stellen zunächst fest, dass beim Schleifendurchlauf r auf jeden Fall einen Wert $\neq 0$ hat. Dies führt dazu, dass die Folge $(X_i)_{i=0,\ldots,p-1}$ mit $X_i = r \cdot i + s$ jedes Element von $A = \mathbb{Z}_p$ genau einmal trifft. (Man muss nur überlegen, dass für jedes $a \in A$ die Gleichung $r \cdot i + s = a$ genau eine Lösung $i = (a - s) \cdot r^{-1}$ hat.) Also wird ein Element von B gefunden, und die Schleife endet erfolgreich. Für die Analyse stellen wir uns zunächst vor, dass der Wert $r = 0$ einfach beibehalten wird (damit die Mathematik glatter funktioniert). Wenn dieser (mathematisch) ideale Algorithmus $r = 0$ und $s \notin B$ wählt, findet er niemals ein Element von B, wohingegen Algorithmus 4.1.3 nach höchstens p Versuchen erfolgreich ist. Wenn der ideale Algorithmus $r = 0$ und $s \in B$ wählt, finden beide Algorithmen in der ersten Runde ein $x \in B$.

Anzahl der Zufallsbits: Wir wählen zwei Elemente von A zufällig; es werden also $2\lceil \log |A| \rceil$ Zufallsbits benötigt.

Rechenzeit: Wir analysieren, was wir über die Anzahl der Schleifendurchläufe sagen können. Wir definieren: $X_i := r \cdot i + s$, $i = 0, 1, \ldots, p-1$, und

$$Y_i := [X_i \in B] = \begin{cases} 1, & \text{falls } X_i \in B, \\ 0, & \text{andernfalls.} \end{cases}$$

Proposition 4.1.4
Jede Zufallsvariable $X_i, 0 \leq i \leq p-1$, ist in A uniform verteilt, und $X_i, 0 \leq i \leq p-1$, sind **paarweise unabhängig**, d. h.: $i \neq j \Rightarrow X_i, X_j$ unabhängig.

Beweis: Für festes i haben wir $\mathbf{Pr}(X_i = a) = p^{-1}$, weil es für jedes r genau ein s mit $r \cdot i + s = a$ gibt, nämlich $s = a - r \cdot i$. Es gilt

$$\mathbf{Pr}(X_i = a \wedge X_j = b) = \mathbf{Pr}(r \cdot i + s = a \wedge r \cdot j + s = b) = \frac{\left|\{(r,s) \mid \binom{i\ 1}{j\ 1} \cdot \binom{r}{s} = \binom{a}{b}\}\right|}{p^2}.$$

Nun hat das Gleichungssystem

$$\begin{pmatrix} i & 1 \\ j & 1 \end{pmatrix} \cdot \begin{pmatrix} r \\ s \end{pmatrix} = \begin{pmatrix} a \\ b \end{pmatrix}$$

genau eine Lösung $\binom{r}{s}$ über dem Körper $\mathbb{Z}_p$, weil die Matrix $\binom{i\ 1}{j\ 1}$ Determinante $i-j \neq 0$ hat. Daher wird jeder Wert $\binom{a}{b}$ mit derselben Wahrscheinlichkeit p^{-2} angenommen, und wir erhalten:

$$\begin{aligned} \mathbf{Pr}(X_i = a \wedge X_j = b) &= \mathbf{Pr}(r \cdot i + s = a \wedge r \cdot j + s = b) = p^{-2} \\ &= \mathbf{Pr}(X_i = a) \cdot \mathbf{Pr}(X_j = b), \end{aligned}$$

was die Unabhängigkeit beweist. □

Korollar 4.1.5
Die Zufallsvariablen Y_i, $0 \leq i \leq p-1$, haben Erwartungswert $\mathbf{E}(Y_i) = \varrho$ und sind ebenfalls paarweise unabhängig.

Beweis: Weil X_i uniform verteilt ist, ist $\mathbf{E}(Y_i) = \mathbf{Pr}(Y_i = 1) = \mathbf{Pr}(X_i \in B) = \varrho$. Wenn $i \neq j$, dann sind X_i und X_j unabhängig, also auch Y_i und Y_j als Funktionen von X_i bzw. X_j. □

Nun definieren wir, für $0 \leq k \leq p$:

$$Z_k := Y_0 + \cdots + Y_{k-1}.$$

Das ist die Anzahl der Erfolge in den ersten k Runden (unter der Annahme, dass man auch nach einem Treffer weitermacht). Beachte: $Z_0 = 0$.

Lemma 4.1.6
$\mathbf{E}(Z_k) = k\varrho$ und $\mathbf{Var}(Z_k) = k\varrho(1-\varrho)$.

Beweis: Wegen der Linearität des Erwartungswertes gilt:

$$\mathbf{E}(Z_k) = \mathbf{E}(Y_0) + \cdots + \mathbf{E}(Y_{k-1}) = k\varrho.$$

Weiter folgt aus der paarweisen Unabhängigkeit (Fakt 2.5.7(b) und die folgende Bemerkung): $\mathbf{Var}(Z_k) = \sum_{0 \le i < k} \mathbf{Var}(Y_i)$. Nun ist Y_i 0-1-wertig, also

$$\mathbf{Var}(Y_i) = \mathbf{Pr}(Y_i = 1) \cdot (1-\varrho)^2 + \mathbf{Pr}(Y_i = 0) \cdot (-\varrho)^2 = \varrho(1-\varrho),$$

und daher $\mathbf{Var}(Z_k) = k\varrho(1-\varrho)$. □

Wir wenden nun die Chebychev-Cantelli-Ungleichung (Proposition 2.7.2) an.[2] Dies liefert:

$$\begin{aligned} \mathbf{Pr}(Z_k = 0) = \mathbf{Pr}(Z_k \le \mathbf{E}(Z_k) - \mathbf{E}(Z_k)) &\le \frac{\mathbf{Var}(Z_k)}{\mathbf{Var}(Z_k) + (\mathbf{E}(Z_k))^2} \\ &= \frac{k\varrho(1-\varrho)}{k\varrho(1-\varrho) + (k\varrho)^2} = \frac{1-\varrho}{1-\varrho+k\varrho} < \frac{1}{1+k\varrho}. \end{aligned} \tag{4.1.3}$$

Wir betrachten nun wieder den (realen) Algorithmus 4.1.3, mit $r \neq 0$, und bezeichnen mit R die Rundenzahl in diesem Algorithmus.

Lemma 4.1.7
Für Algorithmus 4.1.3 gilt:

$$\mathbf{Pr}(R \ge k+1) \le \frac{1-\varrho}{1-\varrho+k\varrho} < \frac{1}{1+k\varrho},$$

für $0 \le k < p$, und $\mathbf{Pr}(R \ge p+1) = 0$.

Beweis: Die Modifikation vom idealen Algorithmus (der zur Zufallsvariablen Z_k führt) zum realen Algorithmus kann Rundenanzahlen nur verringern. Daher gilt $\mathbf{Pr}(R \ge k+1) \le \mathbf{Pr}(Z_k = 0)$. Wegen $r \neq 0$ ist $Z_p \ge 1$ garantiert. □

[2] In der Originalarbeit von Chor und Goldreich wurde die Chebychev-Ungleichung benutzt. Diese führt zu einer ähnlichen Abschätzung, die aber für kleine k nicht „zieht“. Insbesondere ist die Abschätzung $\mathbf{Pr}(Z_k = 0) \le \frac{1}{2}$ mit der Chebychev-Ungleichung erst mit $k \ge 2/\varrho$ zu erreichen.

Für $k \geq \frac{1-\varrho}{\varrho} = \frac{1}{\varrho} - 1$ liefert dies die Schranke $\mathbf{Pr}(R \geq k+1) \leq \frac{1}{2}$. (Dies ist für $k \geq \lfloor 1/\varrho \rfloor$ sicher erfüllt.) Es ist bemerkenswert, dass selbst unter Aufwendung von beliebig vielen Zufallsbits wie in Algorithmus 4.1.1 $1/\varrho$ viele Tests nur zu einer konstanten Erfolgswahrscheinlichkeit führen: Für $k = \lceil 1/\varrho \rceil$ haben wir (s. Prop. A.1.2 im Anhang von Kap. 2):

$$\mathbf{Pr}(\text{die ersten } k \text{ Tests sind erfolglos}) \leq (1-\varrho)^{\lceil 1/\varrho \rceil} < e^{-1} \approx 0{,}368\,;$$

dabei ist für kleine ϱ die Abschätzung mit 0,368 recht genau. Fazit also: Bei der Suche in der beschriebenen Form ist es bei kleinen Dichten ϱ praktisch unschädlich, zum Erreichen einer konstanten Erfolgswahrscheinlichkeit paarweise unabhängige Suche zu benutzen und dadurch viele Zufallsbits einzusparen.

Wir können nun Lemma 4.1.7 benutzen, um die *erwartete Rundenzahl* nach oben abzuschätzen. Dabei verwenden wir unser Wissen, dass es im realen Algorithmus (mit $r \neq 0$) mehr als p Runden auf keinen Fall geben kann. Mit dem Lemma und mit Fakt 2.2.9 sehen wir:

$$\mathbf{E}(R) = \sum_{k \geq 0} \mathbf{Pr}(R \geq k+1) \leq \sum_{0 \leq k < p} \frac{1}{1+k\varrho}. \tag{4.1.4}$$

Wir schätzen ab:

$$\frac{1}{1+k\varrho} \leq \int_{k-1}^{k} \frac{dx}{1+x\varrho} \text{ , für } k \geq 1.$$

Mit (4.1.4) folgt:

$$\mathbf{E}(R) \leq 1 + \int_{0}^{p-1} \frac{dx}{1+x\varrho} < 1 + \int_{0}^{p} \frac{dx}{1+x\varrho}.$$

Da $\int \frac{dx}{1+x\varrho} = \ln(1+x\varrho)/\varrho + C$, folgt

$$\mathbf{E}(R) < 1 + \Big[\ln(1+x\varrho)/\varrho\Big]_0^p = 1 + \frac{\ln(1+p\varrho)}{\varrho} = 1 + \frac{1}{\varrho} \cdot \ln(|B|+1).$$

(Zur allgemeinen Technik der Abschätzung von Summen durch Integrale siehe Abschnitt A unten.)

Satz 4.1.8
Sei $\emptyset \neq B \subseteq A = \mathbb{Z}_p$. Die erwartete Anzahl von Tests „$x \in B$?“ bei der paarweise unabhängigen Suche nach einem Element von B ist $\leq 1 + \ln(1+|B|)/\varrho$, für $\varrho = |B| \,/\, |A|$. Es werden $2\lceil \log p \rceil$ Zufallsbits benötigt (entsprechend zwei Zufallselementen von A).

Das Ergebnis sollte man mit dem Wert $1/\varrho$ aus (4.1.1) für die vollständig zufällige Suche vergleichen. Einerseits ist es einleuchtend, dass der Faktor $1/\varrho$ vorkommt; weniger schön ist der logarithmische Faktor $\ln(1+p\varrho) = \ln(|B|+1)$ im Zähler. (Im Fall der Suche nach einer Primzahl in $[2^{\ell-1}, 2^\ell)$ weiß man nach dem Primzahlsatz (s. Kap. 5), dass $|B| = \Theta(2^\ell/\ell)$; damit erhält man als erwartete Rundenzahl $\Theta(\ell^2)$ gegenüber von $\Theta(\ell)$ für die vollständig zufällige Suche.)

Wir können Algorithmus 4.1.3 so modifizieren, dass immer noch wenige Zufallsbits benötigt werden, dass aber eine erwartete Testzahl von $O(1/\varrho)$ erreicht wird, ohne Abhängigkeit von $|B|$. Die Idee ist dabei, zwei voneinander unabhängige Suchpunktfolgen zu generieren, die (quasi)parallel abgearbeitet werden; in jeder Runde werden also *zwei* Punkte aus A getestet. Hierfür wählen wir zwei Paare (r_1, s_1) und (r_2, s_2) von zufälligen Koeffizienten. Dies liefert Algorithmus 4.1.9.

Algorithmus 4.1.9 ***Verdoppelte paarweise unabhängige Suche***

INPUT: Menge $A = \mathbb{Z}_p$ (Körper) mit unbekannter, nichtleerer Teilmenge B
METHODE:

```
1   Wähle r_1, r_2, s_1, s_2 aus Z_p zufällig;
2   Setze r_1 ← max{1, r_1} und r_2 ← max{1, r_2};
3   x1 ← (s_1 − r_1) mod p;
4   x2 ← (s_2 − r_2) mod p;
5   repeat
6       x1 ← (x1 + r_1) mod p;
7       if x1 ∈ B then return x1;
8       x2 ← (x2 + r_2) mod p;
9       if x2 ∈ B then return x2.
```

Analyse: Korrektheit. Man sollte sich von der scheinbaren Endlosschleife nicht irritieren lassen: Da auf jeden Fall ein Element von B gefunden wird, wird die Schleife über eine der **return**-Anweisungen verlassen.

Anzahl der Zufallsbits: Wir wählen vier Elemente von A zufällig; es werden also $4\lceil\log|A|\rceil$ Zufallsbits benötigt.

Rechenzeit: Sei R die Anzahl der Runden bis zum Erfolg (mit jeweils zwei Tests „$x \in B$?“). Wir schätzen $\mathbf{E}(R)$ ab. – Wir definieren:

$$X_i^{(1)} := r_1 \cdot i + s_1 \text{ und } X_i^{(2)} := r_2 \cdot i + s_2, \text{ für } i = 0, 1, \ldots, p-1,$$

sowie

$$Y_i^{(1)} := [X_i^{(1)} \in B] \quad \text{und} \quad Y_i^{(2)} := [X_i^{(2)} \in B],$$

und schließlich, für $1 \le k \le p$:

$$Z_k^{(1)} := Y_0^{(1)} + \cdots + Y_{k-1}^{(1)} \quad \text{und} \quad Z_k^{(2)} := Y_0^{(2)} + \cdots + Y_{k-1}^{(2)}.$$

Die Analyse für die Folge Y_i, $0 \le i \le p-1$, und ihre Partialsummen Z_k von oben gilt natürlich für jede der beiden Folgen $Y_i^{(h)}$, $0 \le i \le p-1$, und $Z_k^{(h)}$ (mit $h = 1, 2$) gleichermaßen. Zudem sind $Z_k^{(1)}$ und $Z_k^{(2)}$ unabhängig. Wir benutzen wieder (4.1.3) und erhalten:

$$\begin{aligned}\mathbf{Pr}(\text{die ersten } k \text{ Runden sind erfolglos}) &= \mathbf{Pr}(Z_k^{(1)} = 0 \wedge Z_k^{(2)} = 0) \\ &= \mathbf{Pr}(Z_k^{(1)} = 0) \cdot \mathbf{Pr}(Z_k^{(2)} = 0) \le \frac{1}{(1+k\varrho)^2}\,. \qquad (4.1.5)\end{aligned}$$

Wie oben können wir mit Fakt 2.2.9 die erwartete Rundenzahl in der **repeat**-Schleife abschätzen (zur Abschätzung der Reihe durch das Integral vgl. wieder Abschnitt A):

$$\begin{aligned}\mathbf{E}(R) &= \sum_{k\ge 0} \mathbf{Pr}(R \ge k+1) = \sum_{k\ge 0} \mathbf{Pr}(\text{die ersten } k \text{ Runden sind erfolglos}) \\ &\overset{(4.1.5)}{\le} \sum_{k\ge 0} \frac{1}{(1+k\varrho)^2} = 1 + \sum_{k\ge 1} \frac{1}{(1+k\varrho)^2} \\ &< 1 + \int_0^\infty \frac{dx}{(1+x\varrho)^2} = 1 + \Big[-\frac{1/\varrho}{1+x\varrho}\Big]_0^\infty = 1 + \frac{1}{\varrho}.\end{aligned}$$

Satz 4.1.10
Sei $\emptyset \ne B \subseteq A = \mathbb{Z}_p$. Die erwartete Anzahl von Tests „$x \in B$?“ bei der **verdoppelten** zweifach unabhängigen Suche nach einem Element von B ist $\le 2(1 + 1/\varrho)$, für $\varrho = |B| \,/\, |A|$. Es werden $4\lceil \log p \rceil$ Zufallsbits benötigt.

Dieses Ergebnis macht sich neben $1/\varrho$ für die völlig zufällige Suche sehr gut!

4.2 Suche in sortierten einfach verketteten linearen Listen

In diesem Abschnitt untersuchen wir ein elementares Suchproblem: Gegeben ist eine *angeordnete* (einfach verkettete) lineare Liste mit Einträgen $x_1 < x_2 < \cdots < x_n$

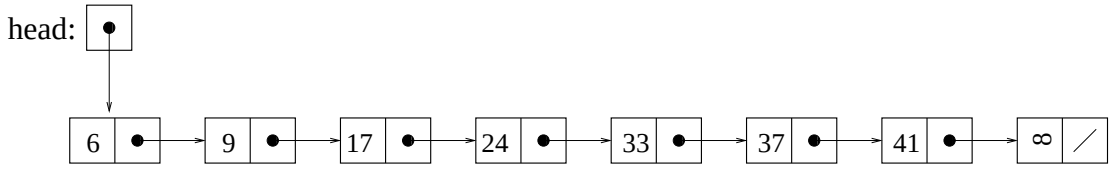


Abbildung 4.2.1: Geordnete lineare Liste, $n = 8$.

aus einem geordneten Universum $(U, <)$. (Man stelle sich als Einträge Zahlen vor.) Wir wollen in dieser Liste nach einem Eintrag x suchen. Normalerweise bleibt nichts anderes übrig, als die Liste linear von vorne bis hinten zu durchmustern, bis das Element x gefunden wurde *oder* bis das Ende der Liste erreicht wird *oder* (wegen der Sortierung!) bis ein Eintrag $y > x$ gefunden wird. Der Aufwand für diese Suche ist proportional zur Anzahl der durchgeführten Schlüsselvergleiche. Diese beträgt k, wobei $k \leq n$ minimal ist mit $x_k \geq x$. (Die Möglichkeit, dass x größer ist als alle Listeneinträge, umgehen wir, indem wir ein letztes Listenelement mit einem künstlichen Schlüssel „∞“ anhängen. Ab hier sei also $x < x_n$ vorausgesetzt.) Im schlechtesten und im mittleren Fall ist der Suchaufwand $\Theta(n)$.

Diese lineare Schranke wollen wir schlagen. Hierfür muss eine besondere Voraussetzung erfüllt sein: Es muss möglich sein, ein Element der linearen Liste uniform zufällig auszuwählen. Dies ist bei den gewöhnlichen Listenimplementierungen, wie sie von den Programmiersprachen bereitgestellt werden, nicht der Fall. Allerdings ist es leicht, diese Voraussetzung herzustellen. Wir nennen drei Möglichkeiten.

- In Abbildung 4.2.2 ist die erste Möglichkeit dargestellt. Man könnte zur Implementierung der Liste vom Programm aus ein Array `A[1..`m`] of listelem` von Listenelementen kreieren und beim Einfügen in die Liste durch geeignete Verwaltung dafür sorgen, dass die *benutzten* Listenelemente immer einen vollen Anfangsabschnitt der Länge n dieses Arrays bilden. (Wenn auch Löschungen durchgeführt werden sollen, ist entweder eine doppelte Verzeigerung der Liste nötig oder man muss für eine Löschung zweimal suchen.)

- Abbildung 4.2.3 zeigt eine zweite Möglichkeit. Eine lineare Liste lässt sich auch „zu Fuß“ realisieren, mit Hilfe eines Arrays `value[1..`m`] of data`, das die Daten enthält (im Beispielbild sind dies auch Zahlen), und eines zweiten Arrays `next[1..`m`] of int`, das die Zeiger als explizite Zahlen in $\{1, \ldots, n\}$ enthält.

Der Listenbeginn ist durch einen Wert `head` gegeben. (`free` enthält die Nummer der ersten freien Zelle, `maxlength` die Länge m des Arrays.) Um die Liste zu durchlaufen, beginnt man mit dem Wert j_1 in `head` – im Beispiel ist dies $j_1 = 4$ – und verfolgt iterativ die `next`-Werte, mt $j_{r+1} :=$ `next[`j_r`]` bis zum Listenende, das durch den Wert `next[`j_n`]`$= 0$ angezeigt wird.) Dabei müssen die benutzten Arrayzellen in beiden Arrays stets einen zusammenhängenden Anfangsabschnitt bilden. Die Implementierung von Einfüge- und Lösch-Operationen in einer solchen Struktur ist eine Übungsaufgabe für die ersten Semester. (Beim Löschen muss man zweimal suchen, um die kompakte Darstellung als Anfangsabschnitt aufrechtzuerhalten.)

- Eine dritte Möglichkeit ist es, zusätzlich zur Liste ein Array `p[1..`m`]` mitzuführen, das im Abschnitt `p[1..`n`]` einen Zeiger auf jedes Listenelement enthält. (Diese Lösung kostet natürlich zusätzlichen Platz. Sie hat aber den Vorteil, dass man beim Löschen in der linearen Liste nur einmal suchen muss.)

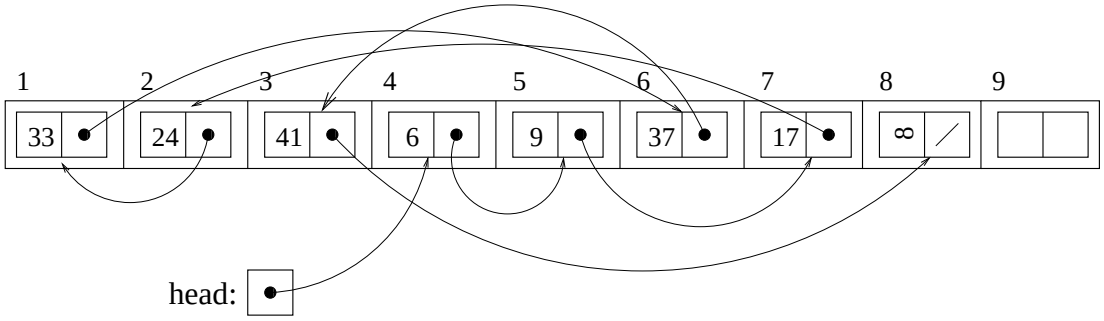


Abbildung 4.2.2: Geordnete lineare Liste, Listenelemente in Array, $m = 9$, $n = 8$.

Wir stellen uns für das Folgende vor, dass (Zeiger auf) Listenelemente zufällig gewählt werden können. Die Idee für den Algorithmus ist einfach: Man wählt zufällig eine Reihe (L viele) von Listenelementen $x_{i_1}, \ldots, x_{i_L}$ (Wiederholungen sind erlaubt, man kümmert sich also nicht darum) und sucht unter diesen den größten Eintrag x_{i_0} heraus, der $< x$ ist. Mit dem Nachfolger dieses Listenelements startend sucht man das erste Element in der Liste mit einem Schlüssel, der $\geq x$ ist. Leichte Komplikationen werden dadurch verursacht, dass x kleiner als alle Einträge in der Liste sein könnte (dann ist das erste Listenelement die Ausgabe) oder $x > x_1$ gilt und alle zufällig gewählten Elemente mindestens so groß wie x sind (dann sucht man in der Liste

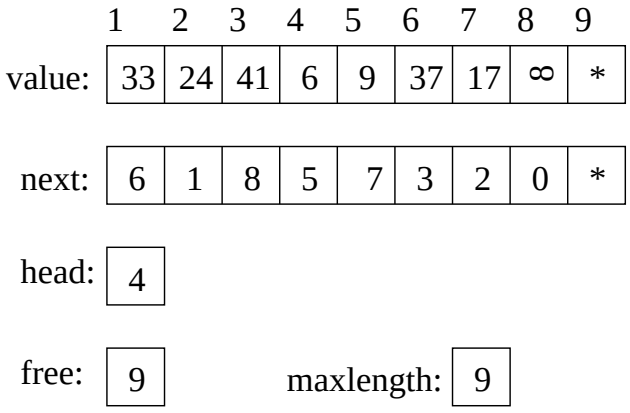


Abbildung 4.2.3: Geordnete lineare Liste in 2 Arrays, $m = 9$, $n = 8$.

startend mit dem zweiten Element x_2). Algorithmus 4.2.1 stellt dies systematisch dar. Die Zufallsauswahl ist Phase 1, die lineare Suche (inklusive Sonderfall) Phase 2.

Algorithmus 4.2.1 ***Randomisierte Suche in linearer Liste***

INPUT: Liste (Start bei `head`) mit Einträgen $x_1 < \cdots < x_n$, Suchschlüssel $x < x_n$.

METHODE:

```
   // Trivialitätstest und Initialisierung:
1    if x ≤ head.key then return head else best ← head; bestkey ← head.key;
   // Phase 1: Wähle L Einträge zufällig, bestimme den größten, der < x ist:
2    repeat L times
3        rand ← (Zeiger auf) zufälliges Listenelement;
4        if rand.key < x and bestkey < rand.key
5        then best ← rand; bestkey ← rand.key;
   // Phase 2: Lineare Suche ab dem auf best folgenden Listenelement:
6    iter ← best;
7    repeat iter ← iter.next until iter.key ≥ x;  // endet wegen x_n = ∞
8    return iter.
```

Analyse: Korrektheit. Zeile 1 testet, ob das erste Listenelement die korrekte Ausgabe ist. Falls dies nicht so ist, ist der erste Listeneintrag kleiner als x, und x_k steht

weiter hinten in der Liste. *Phase 1*: In Zeilen 2–5 werden Listeneinträge zufällig gewählt. Unter ihnen wird der größte Eintrag ausgewählt, der $\leq x$ ist („`best`“) – wenn es unter den zufällig gewählten Einträgen keinen gibt, der kleiner als x ist, bleibt es beim ersten Listeneintrag x_1. Der so bestimmte Listeneintrag steht sicher *echt vor* x_k, dem gesuchten Eintrag. Es folgt *Phase 2*. Ab dem auf `best` folgenden Listeneintrag wird linear der erste gesucht, der einen Schlüssel $\geq x$ hat. Ein solcher existiert (weil $x_n = \infty$), und er ist das korrekte Ergebnis.

Analyse: Zeitbedarf. Die Initialisierung benötigt einen Schlüsselvergleich. Phase 1 benötigt höchstens $2L$ Schlüsselvergleiche. Die Anzahl der Schlüsselvergleiche in Phase 2 ist eine Zufallsvariable. Wenn x_k der kleinste Schlüssel ist, der $\geq x$ ist, dann wird auf jeden Fall x mit x_k verglichen. Die Anzahl der weiteren Vergleiche nennen wir Y_2. Wir haben (siehe Abbildung 4.2.4):

$$\begin{aligned} Y_2 \geq j &\Rightarrow \text{in Phase 2 werden } x_{k-j}, \ldots, x_{k-1} \text{ mit } x \text{ verglichen} \\ &\Rightarrow \text{in Phase 1 werden } x_{k-j}, \ldots, x_{k-1} \text{ nicht gewählt.} \end{aligned}$$

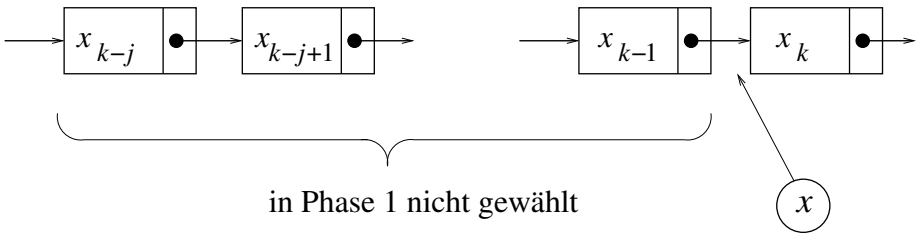


Abbildung 4.2.4: Listeneinträge, die nicht gewählt wurden, wenn $Y_2 \geq j$ ist.

Die Wahrscheinlichkeit, dass $x_{k-j}, \ldots, x_{k-1}$ in keinem der L Versuche in Phase 1 gewählt werden, ist

$$\left(\frac{n-j}{n}\right)^L.$$

Damit (mit Fakt 2.2.9):

$$\mathbf{E}(Y_2) = \sum_{j \geq 1} \mathbf{Pr}(Y_2 \geq j) \leq \sum_{1 \leq j \leq n} \left(1 - \frac{j}{n}\right)^L.$$

Die Summe schätzen wir nach oben durch ein Integral ab, wie in Abschnitt A beschrieben. Die Funktion $f\colon t \mapsto (1-\frac{t}{n})^L$ ist im Intervall $[0,n]$ monoton fallend. Lemma A.0.1(b) mit $a=0$ und $b=n$ liefert

$$\sum_{1\le j\le n}\left(1-\frac{j}{n}\right)^L \le \int_0^n \left(1-\frac{t}{n}\right)^L dt = \Big[-\frac{n}{L+1}\Big(1-\frac{t}{n}\Big)^{L+1}\Big]_0^n = \frac{n}{L+1}.$$

Wenn Y die Anzahl aller Vergleiche im Algorithmus bezeichnet, erhalten wir:

$$\mathbf{E}(Y) \le 1 + 2L + 1 + \frac{n}{L+1}.$$

Wir setzen nun L so fest, dass dieser Ausdruck möglichst klein wird. Standardmethoden (setze Ableitung von $2x + n/(x+1)$ gleich 0, was $x = \sqrt{n/2}-1$ liefert) ergeben als fast optimale Wahl $L = \lfloor\sqrt{n/2}\rfloor$. Damit wird

$$\mathbf{E}(Y) \le 2 + 2\Big\lfloor\sqrt{n/2}\Big\rfloor + \frac{n}{\Big\lfloor\sqrt{n/2}\Big\rfloor+1} \le 2 + \sqrt{2n} + \frac{n}{\sqrt{n/2}} = 2 + 2\sqrt{2n} = O(\sqrt{n}).$$

Satz 4.2.2
Suchen in einer angeordneten linearen Liste mit n Einträgen, bei der zufälliges Wählen eines Listenelements möglich ist, benötigt bei Verwendung von Algorithmus 4.2.1 im erwarteten Fall Zeit $O(\sqrt{n})$.

Bemerkung: Wenn man anstelle der in einem Array organisierten Liste direkt ein sortiertes Array benutzt, kann man mit $\lceil \log n \rceil$ Vergleichen suchen (binäre Suche). In unserer Listenstruktur sind jedoch auch *Einfügungen* und *Löschungen* in erwarteter Zeit $O(\sqrt{n})$ durchführbar, was in einem sortierten Array nicht der Fall ist.

Eine durch die Zufallsauswahl von Listenelementen inspirierte Datenstruktur ist die *Skipliste* [William Pugh: Skip Lists: A Probabilistic Alternative to Balanced Trees. Commun. ACM 33(6): 668-676 (1990)]. Dabei wird diese Zufallsauswahl iteriert: $p \in (0,1]$ sei konstant gewählt, zum Beispiel $p = \frac{1}{2}$. Aus der Originalliste wird jedes Element mit Wahrscheinlichkeit p gewählt, und Zeiger auf die gewählten Elemente bilden die Liste der Stufe 1. Aus der Liste der Stufe i wird auf dieselbe Weise die Liste der Stufe $i+1$ gebildet, für $i = 1, 2, 3, \ldots$. Im Erwartungswert hat die Liste der Stufe i etwa $p^i n$ Einträge, und mit hoher Wahrscheinlichkeit gibt es nur $O(\log(1/p))$ viele Stufen. Suchen erfolgen erst in der Liste der höchsten Stufe, dann in der darunterliegenden, usw. Es ergeben sich erwartete Zeiten für Suchen, Einfügen und Löschen von $O(\log n)$. Details zur Datenstruktur *Skipliste* findet man zum Beispiel in dem Buch „Algorithmen und Datenstrukturen“ von Ottmann und Widmayer.

4.3 Quickselect

Für diesen Abschnitt sei U eine durch die Relation $<$ totalgeordnete Menge (z.B. $U = \mathbb{N}$).

Definition 4.3.1
Wenn $S = (a_1, \ldots, a_n)$ eine Folge von n Elementen von U ist (Wiederholungen sind erlaubt!) und $1 \leq k \leq n$ ist, dann heißt $a \in \{a_1, \ldots, a_n\}$ **das Element von Rang k** in S, wenn

$$|\{i \mid 1 \leq i \leq n, a_i < a\}| < k \leq |\{i \mid 1 \leq i \leq n, a_i \leq a\}|.$$

Das Element a von Rang $\lceil n/2 \rceil$ heißt der **Median** von S.

Das Selektionsproblem besteht darin, zu einem gegebenen Array $A[1\,..\,n]$ mit Einträgen $a_1, \ldots, a_n$ den Eintrag von Rang k zu finden. Die „einfachste“ Möglichkeit hierfür ist, die Folge zu sortieren und das Element an Position k auszugeben. (Rechenzeit: $O(n \log n)$.)

Beispiel: Sortieren von $S = (4, 1, 27, 4, 15, 11, 7, 30)$ liefert $(1, 4, 4, 7, 11, 15, 27, 30)$. Also ist 4 das Element vom Rang 2 (und vom Rang 3) und 7 ist der Median. Wenn man $S = (1, 30, 4, 11, 15, 27, 7)$ sortiert, ergibt sich $(1, 4, 7, 11, 15, 27, 30)$. Also ist 30 das Element vom Rang 7, und 11 ist der Median. *Beachte*: Wenn n ungerade ist, ist der Median das „mittlere“ Element, wenn man S sortiert anordnet; wenn n gerade ist, sitzt der Median unmittelbar links neben der Mitte.

Eine weitere Möglichkeit zu demonstrieren, dass a das Element von Rang k ist, ist, A so umzusortieren, dass erst Einträge $\leq a$ kommen, dann a in Position k, dann Einträge $\geq a$. Im ersten Beispiel zeigt die Anordnung $(4, 1, 4, 11, 7, 27, 30, 15)$, dass 4 das Element von Rang 3 ist, die Anordnung $(4, 1, 4, 7, 27, 30, 11, 15)$, dass 7 der Median ist. Wir bemerken: Wenn eine solche Anordnung gegeben ist, genügen $n - 1$ Vergleiche, um die Behauptung zu verifizieren, dass an Stelle k der Eintrag von Rang k steht. Wir betrachten nun einen Algorithmus, der in (erwarteter) Linearzeit eine solche Anordnung herstellt. Für die Zwecke dieser theoretischen Diskussion nehmen wir an, dass alle Einträge verschieden sind.

Gegeben sei also ein Array `A[1..n]`, in dem die verschiedenen Einträge $a_1 < \cdots < a_n$ aus einem totalgeordneten Bereich $(U, <)$ in **irgendeiner Reihenfolge** gespeichert sind.

Aufgabe: „Selection(k)“: Arrangiere die Einträge in `A[1 . . n]` so um, dass in `A[`k`]` das Element a_k von Rang k in $S = \{a_1, \ldots, a_n\}$ steht und dass die Einträge in `A[`$1 \ldots k-1$`]` kleiner als a_k und die in `A[`$k+1 \ldots n$`]` größer als a_k sind.

Der „Quickselect-Algorithmus“ ist ein randomisierter Algorithmus, der das Selektionsproblem in erwarteter Zeit $O(n)$ und mit erwarteter Vergleichsanzahl $\leq 4n$ löst.

Algorithmus und Analyseprinzip finden sich separat auf Folien aus der Vorlesung „Algorithmen und Datenstrukturen“ vom SS 2016. (Bitte an diese Stelle einfügen!) Das zentrale Ergebnis ist Folgendes.

Satz 4.3.2
Der Algorithmus **QuickSelect** macht auf Eingaben der Länge n im erwarteten Fall weniger als $4n$ Vergleiche.

Auf den Folien wird die Summation der Terme der letzten Summe

$$S = \sum_{1 \leq i < k < j \leq n} \frac{1}{j - i + 1}$$

nur sehr summarisch durchgeführt, mit dem Ergebnis, dass diese Summe kleiner als $n - 2$ ist. Damit beträgt der Beitrag der Paare (i, j) mit $i < k < j$ zur erwarteten Vergleichsanzahl weniger als $2n$.

Hier wollen wir für Interessierte eine noch schärfere Schranke herleiten.

(Beginn nicht prüfungsrelevanter Abschnitt.)

Wir analysieren dazu die Summe genauer. Wie auf den Folien werden die Summanden in die nachfolgende $(k-1) \times (n-k)$-Matrix geschrieben (für $k \leq n/2$):

$$\begin{pmatrix}
\frac{1}{k+1} & \frac{1}{k+2} & \cdots & \frac{1}{n-k} & \frac{1}{n-k+1} & \frac{1}{n-k+2} & \cdots & \cdots & \frac{1}{n-1} & \frac{1}{n} \\
\frac{1}{k} & \frac{1}{k+1} & \frac{1}{k+2} & \cdots & \frac{1}{n-k} & \frac{1}{n-k+1} & \frac{1}{n-k+2} & \cdots & \frac{1}{n-2} & \frac{1}{n-1} \\
\vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\
\frac{1}{4} & \frac{1}{5} & \cdots & \frac{1}{k+1} & \frac{1}{k+2} & \cdots & \frac{1}{n-k} & \frac{1}{n-k+1} & \frac{1}{n-k+2} & \cdots \\
\frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{k} & \frac{1}{k+1} & \frac{1}{k+2} & \cdots & \frac{1}{n-k} & \frac{1}{n-k+1} & \frac{1}{n-k+2}
\end{pmatrix}$$

Auf den Folien wurde gesagt, dass die Summe aller Einträge kleiner als n sein muss, weil die (konstanten) Einträge in jeder Diagonalen sich höchstens zu 1 summieren und es exakt $n-2$ Diagonalen gibt.

Wir analysieren die Summe der Matrixeinträge nun genauer. Wir teilen sie dazu in drei Teile ein:

Teil I, Summe B_1: Einträge strikt unterhalb der Diagonalen mit den $\frac{1}{k+1}$-Einträgen,

Teil II, Summe B_2: Einträge ab der Diagonalen mit den $\frac{1}{k+1}$-Einträgen (startet in linker oberer Ecke) bis zur Diagonalen mit den $\frac{1}{n-k}$-Einträgen (endet zwei Positionen links von rechter unterer Ecke),

Teil III, Summe B_3: Einträge strikt oberhalb der Diagonalen mit den $\frac{1}{n-k}$-Einträgen.

Setze $\alpha := k/n$. Dann gilt $0 < \alpha \leq \frac{1}{2}$. Offenbar gilt:

$$B_1 = \frac{k-2}{k} + \frac{k-3}{k-1} + \frac{k-4}{k-2} + \cdots + \frac{1}{3} < k-2.$$

Nach Beispiel A.0.2(ii) im Anhang haben wir $\frac{1}{u+1} + \frac{1}{u+2} + \cdots + \frac{1}{v} \leq \ln\left(\frac{v}{u}\right)$. Damit:

$$\begin{aligned} B_3 &= \frac{1}{n} + \frac{2}{n-1} + \frac{3}{n-2} + \cdots + \frac{k-1}{n-k+2} + \frac{k-1}{n-k+1} \\ &\leq \left(\frac{n+1}{n} - 1\right) + \left(\frac{n+1}{n-1} - 1\right) + \cdots + \left(\frac{n+1}{n-k+1} - 1\right) \\ &= (n+1)\left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{n-k+1}\right) - k \\ &\leq (n+1)\ln\left(\frac{n}{n-k}\right) - k. \end{aligned}$$

Wir erhalten, weil $\ln(1/(1-\alpha)) \leq \ln 2 < 1$:

$$B_1 + B_3 < (n+1)\ln\left(\frac{n}{n-k}\right) - 2 = (n+1)\ln\left(\frac{1}{1-\alpha}\right) - 2 < -n\ln(1-\alpha).$$

(Der Logarithmus ist natürlich negativ!) Schließlich, nochmals mit Beispiel A.0.2(ii):

$$\begin{aligned} B_2 &= (k-1)\left(\frac{1}{k+1} + \frac{1}{k+2} + \frac{1}{k+3} + \cdots + \frac{1}{n-k}\right) \\ &\leq (k-1)\ln\left(\frac{n-k}{k}\right) < \alpha n \ln\left(\frac{n-k}{k}\right) \\ &= \alpha n \ln\left(\frac{1-\alpha}{\alpha}\right) = n \cdot (\alpha\ln(1-\alpha) - \alpha\ln(\alpha)). \end{aligned}$$

Wir fassen zusammen:

$$\begin{aligned} S &= B_1 + B_2 + B_3 \\ &\leq n \cdot (-\ln(1-\alpha) + \alpha\ln(1-\alpha) - \alpha\ln(\alpha)) \\ &= n \cdot (-(1-\alpha)\ln(1-\alpha) - \alpha\ln\alpha) \\ &= n \cdot H_e(\alpha), \end{aligned}$$

wobei $H_e(\alpha)$ die Entropie der Verteilung $(\alpha, 1-\alpha)$ zur Basis e ist. Bezüglich der in der Informatik üblichen binären Entropie $H_2(\alpha) = -(1-\alpha)\log_2(1-\alpha) - \alpha\log_2(\alpha)$ ergibt sich mit Hilfe der Beziehung $H_e(\alpha) = (\ln 2)H_2(\alpha)$:

$$S \leq n \cdot (\ln 2)H_2(\alpha).$$

Insgesamt, weil $\alpha \leq \frac{1}{2}$:

$$\mathbf{E}(C_k) \leq n \cdot (2 + (2 \ln 2) H_2(\alpha)) .$$

Der schlechteste Fall tritt für $\alpha = \frac{1}{2}$, also $k = n/2$, auf. Dies ist der Fall des Medians. Dann ist $H_2(\frac{1}{2}) = 1$, und $\mathbf{E}(C_{n/2}) \leq (2 + (2 \ln 2))n \approx 3.3863n$.

(Ende nicht prüfungsrelevanter Abschnitt.)

Satz 4.3.3
Quickselect mit $k = \alpha n$ hat erwartete Vergleichsanzahl $\leq (2 + (2 \ln 2) H_2(\alpha)) \cdot n$.

Der nächste Abschnitt 4.4 zeigt, dass (mit einem anderen, etwas komplizierteren Algorithmus) die Vergleichsanzahl noch auf etwa $\frac{3}{2}n$ verbessert werden kann.

4.4 Mediansuche mit „Random Sampling“

In diesem Abschnitt betrachten wir nochmals das Selektionsproblem, und zwar den wichtigen Spezialfall $k = \lceil n/2 \rceil$: das ***Medianproblem***.

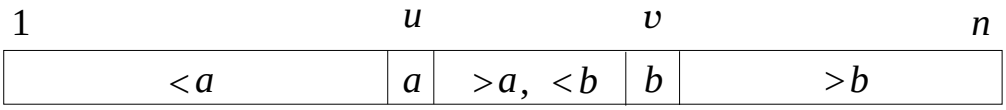
Zur Vereinfachung der Analyse nehmen wir an, dass $n/2$ und $n^{1/4}$ ganze Zahlen sind. (Andernfalls arbeitet man mit geeigneten gerundeten Werten.)

Die Idee ist, mit einem Zufallsverfahren zwei Elemente a und b in S zu finden derart, dass mit großer Wahrscheinlichkeit a kleiner als der Median und b größer als der Median ist. Dann kann man durch einmaliges Durchmustern aller Einträge in `A` und Vergleichen mit a und b die Einträge von `A` in drei Klassen einteilen und umarrangieren: In `A[`$1 \,..\, u-1$`]` stehen die Einträge, die kleiner als a sind; in `A[`u`]` steht a; in `A[`$u+1 \,..\, v-1$`]` stehen die Einträge, die zwischen a und b liegen; in `A[`v`]` steht b; in `A[`$v+1 \,..\, n$`]` schließlich stehen die Einträge, die größer als b sind. Nun muss man nur noch die Einträge in `A[`$u+1 \,..\, v-1$`]` sortieren, um den Median an seinen Platz zu schaffen und die anderen Elemente wie verlangt anzuordnen.

Wenn nur $v - u = o(n/\log n)$ gilt, wird der Aufwand für das Sortieren $o(n)$ sein.

Aber wie finden wir die gewünschten Einträge a und b? Die Intuition hinter dem Algorithmus ist folgende. Wir wählen

$$L = n^{3/4}$$



viele Einträge aus S rein zufällig, der – algorithmischen – Einfachheit halber mit Wiederholung. Das ist im Durchschnitt jeder $n^{1/4}$-te Eintrag. (Diese Zufallsauswahl nennt man „*random sampling*“.) Man kann sich vorstellen, dass diese zufällig gewählten Elemente $b_1 \leq \cdots \leq b_L$ einigermaßen gleichmäßig über den Bereich $\{1, \ldots, n\}$ der Ränge in S verstreut sind. Wir wählen nun aus der (Multi-)Menge $B = \{b_1, \ldots, b_L\}$ zwei Elemente a und b, die wir in der Nähe der Ränge $n/2 - n^{3/4}$ bzw. $n/2 + n^{3/4}$ vermuten. Da B um den Faktor $n^{1/4}$ „dünner“ als S ist, sollten wir hierfür die Elemente aus B vom Rang $(n/2 - n^{3/4})/n^{1/4} = \frac{1}{2}n^{3/4} - \sqrt{n}$ und $(n/2 + n^{3/4})/n^{1/4} = \frac{1}{2}n^{3/4} + \sqrt{n}$ nehmen. Um diese Elemente zu identifizieren, wird B einfach sortiert. Der Aufwand hierfür ist $O(n^{3/4} \log n) = o(n)$.

Es kann natürlich passieren, dass bei der Zufallsauswahl etwas schief geht. Es kann sein, dass a und b den Median nicht „einrahmen“ oder dass der Abstand $v - u$ zu groß ist. Mit Hilfe der Hoeffding-Schranke werden wir aber zeigen, dass dies nur mit verschwindend geringer Wahrscheinlichkeit passiert. Zudem kann diese Situation erkannt werden, so dass wir einen selbstverifizierenden Algorithmus mit sehr kleiner Versagenswahrscheinlichkeit erhalten (Algorithmus 4.4.1).

Algorithmus 4.4.1 ***Random Sample Median***
Input: Array `A[1..n]` (ungeordnet)
Methode:
(1) $L \leftarrow n^{3/4}$;
(2) Wähle **zufällig** $j_1, \ldots, j_L$ aus $\{1, \ldots, n\}$;
(3) Sortiere $\texttt{A}[j_1], \ldots, \texttt{A}[j_L]$ aufsteigend; Resultat ist $b_1 \leq b_2 \leq \cdots \leq b_L$.
(4) $a \leftarrow b_{L/2-\sqrt{n}}$;
(5) $b \leftarrow b_{L/2+\sqrt{n}}$;
(6) **partitioniere**(a, b, u, v), mit Ausgabewerten u und v, das bewirkt:
für geeignete u und v arrangiere `A[1..n]` so um, dass gilt:
Einträge $< a$ stehen in $\texttt{A}[1\,..\,u-1]$;
Einträge $> b$ stehen in $\texttt{A}[v+1\,..\,n]$;
a steht in $\texttt{A}[u]$, b steht in $\texttt{A}[v]$;
Einträge zwischen a und b stehen in $\texttt{A}[u+1\,..\,v-1]$;
(7) **Fall 1:** $u > n/2$ oder $v < n/2$: **return** „?“;
(8) **Fall 2:** $v - u > 2D \cdot n^{3/4}$: **return** „?“; // D: eine noch festzulegende Konstante
(9) **Fall 3:** Sonst. Sortiere $\texttt{A}[u+1\,..\,v-1]$, z.B. mit Mergesort; **return** $\texttt{A}[n/2]$.

Es ist klar, dass der Algorithmus, wenn er den Fall 3 erreicht und den mittleren Abschnitt sortiert hat, die verlangte Umordnung „Median in die mittlere Position“ korrekt vorgenommen hat. Wir haben also einen selbstverifizierenden Algorithmus, den man durch Wiederholung in einen Las-Vegas-Algorithmus umbauen kann, wenn die Wahrscheinlichkeit für die Ausgabe „?“ genügend klein ist.

Laufzeitanalyse: Entscheidend ist die Zahl der Vergleiche. Für das Sortieren der $n^{3/4}$ Zufallselemente genügen $O(n^{3/4} \log n) = o(n)$ Vergleiche. Für die Separierung in die drei Abteilungen „$< a$“, „zwischen a und b (einschließlich)“ und „$> b$“ werden im schlechtesten Fall $2n$ Vergleiche benötigt. Wenn man für jeden Eintrag x *zufällig* entscheidet, ob man x zuerst mit a oder mit b vergleicht, ist mit hoher Wahrscheinlichkeit die Zahl von Vergleichen für Zeile (6) sogar nur etwa $\frac{3}{2}n + o(n)$ (Übungsaufgabe!). Das Sortieren des mittleren Teilarrays in Fall 3 kostet wieder $O(n^{3/4} \log n) = o(n)$ Vergleiche.

Wahrscheinlichkeitsanalyse: Wir schätzen die Wahrscheinlichkeit dafür ab, dass nicht Fall 3 eintritt.

Mit welcher Wahrscheinlichkeit tritt **Fall 1** ein, d. h. ist $u > n/2$ oder $v < n/2$? Wir

fragen also nach der Wahrscheinlichkeit $\mathbf{Pr}(A_1 \cup A_2)$ für die Ereignisse

$$A_1 := \{u > n/2\} \text{ und } A_2 := \{v < n/2\}.$$

Wir betrachten nur A_1. Wir können uns o. B. d. A. vorstellen, dass $a_1 < \cdots < a_n$ gilt. (Diese Anordnung wird vom Algorithmus natürlich nicht hergestellt. Da über die zufälligen Indizes $j_1, \ldots, j_L$ Einträge zufällig gewählt werden, ist die Anordnung im Array `A` unerheblich.) Wenn A_1 eintritt, dann trifft die Zufallsauswahl $\texttt{A}[j_1], \ldots, \texttt{A}[j_L]$ die Menge $\{a_1, \ldots, a_{n/2}\}$ (erste Hälfte) strikt weniger häufig als $(L/2 - \sqrt{n})$-mal. Andererseits erwarten wir genau $L/2$ Treffer. Diese Unterschreitung des Erwartungswertes ist sehr unwahrscheinlich, wie man durch Anwendung der Hoeffding-Schranke (Abschnitt 2.6) sieht. Wir definieren:

$$\begin{aligned} X_i &:= \left\{ \begin{array}{ll} 1, & \text{falls } \texttt{A}[j_i] \in \{a_1, \ldots, a_{n/2}\}, \\ 0, & \text{sonst,} \end{array} \right\}, \text{ für } 1 \leq i \leq L; \\ X &:= X_1 + \cdots + X_L, \\ m &:= \mathbf{E}(X) = L/2, \\ \delta &:= \sqrt{n}/m = 2n^{-1/4}. \end{aligned}$$

Dann haben wir, nach einer Form der Hoeffding-Schranke (Korollar 2.6.4, Formel (2.6.18) in Abschnitt 2.6):

$$\begin{aligned} \mathbf{Pr}(A_1) &= \mathbf{Pr}(X < (1-\delta)m) \\ &\leq e^{-\delta^2 m/2} \\ &= e^{-n^{-1/2} \cdot L} \\ &= e^{-n^{1/4}}. \end{aligned}$$

Da man $\mathbf{Pr}(A_2)$ analog abschätzen kann, ergibt sich als Schranke für die Wahrscheinlichkeit, dass Fall 1 eintritt, $2e^{-n^{1/4}}$, eine mit wachsendem n sehr rasch verschwindende Zahl.

Mit welcher Wahrscheinlichkeit tritt **Fall 2** ein, d. h. es ist $v - u > 2Dn^{3/4}$? B sei das Ereignis, dass Fall 2 eintritt, *nicht aber* Fall 1.

Wenn Fall 2 eintritt, aber nicht Fall 1, dann ist $u \leq n/2 \leq v$ und es tritt mindestens einer der beiden folgenden Fälle ein:
(i) $v - n/2 > D \cdot n^{3/4}$ (Ereignis B_1),
(ii) $n/2 - u > D \cdot n^{3/4}$ (Ereignis B_2).

(Wenn $v - n/2$ und $n/2 - u$ beide $\leq D \cdot n^{3/4}$ wären, hätten wir $v - u \leq 2D \cdot n^{3/4}$.)

Wir zeigen, dass B_1 exponentiell kleine Wahrscheinlichkeit hat; für B_2 zeigt man dies analog.

Dass $v > n/2 + D \cdot n^{3/4}$ ist, bedeutet, dass $b = b_{L/2+\sqrt{n}}$ nicht in $\{a_1, \ldots, a_{n/2+D\cdot n^{3/4}}\}$ sitzt, d. h. dass die Zufallsauswahl $\texttt{A}[j_1], \ldots, \texttt{A}[j_L]$ weniger als $L/2+\sqrt{n}$ oft die Menge $S_{\text{unten}} := \{a_1, \ldots, a_{n/2+D\cdot n^{3/4}}\}$ trifft. In S_{unten} erwarten wir aber $(n/2 + Dn^{3/4}) \cdot (L/n) = L/2 + D\sqrt{n}$ Treffer. Wir rechnen nun wieder mit Hilfe der Hoeffding-Ungleichung aus, dass eine Trefferzahl von $\leq L/2 + \sqrt{n}$ für nicht zu kleine D extrem unwahrscheinlich ist.

Definiere:

$$
\begin{aligned}
X_i &:= \left\{ \begin{array}{ll} 1, & \text{falls } \texttt{A}[j_i] \in S_{\text{unten}}, \\ 0, & \text{sonst,} \end{array} \right\}, \text{ für } 1 \leq i \leq L; \\
X &:= X_1 + \cdots + X_L, \\
m &:= \mathbf{E}(X) = L \cdot \frac{|S_{\text{unten}}|}{n} = \frac{L}{2} + D\sqrt{n}, \\
\delta &:= \frac{m - (L/2 + \sqrt{n})}{m} = \frac{(D-1)\sqrt{n}}{L/2 + D\sqrt{n}}.
\end{aligned}
$$

Nun haben wir, wieder mit der Hoeffding-Schranke (Korollar 2.6.4 in Kap. 2):

$$
\begin{aligned}
\mathbf{Pr}(B_1) &\leq \mathbf{Pr}(X \leq m(1-\delta)) \\
&\leq e^{-\delta^2 m/2} \\
&= e^{-(D-1)^2 \cdot n/(L+2D\sqrt{n})} \\
&< e^{-(D-1)^2 \cdot n^{1/4}/(1+2Dn^{-1/4})}.
\end{aligned}
$$

Für jedes feste $D > 1$ geht auch diese Wahrscheinlichkeit mit wachsendem n exponentiell rasch gegen 0. Man wählt z.B. $D = \frac{5}{2}$ und hat damit einen zu sortierenden „mittleren Bereich“ von maximal $5n^{3/4}$ Elementen und eine Wahrscheinlichkeit von höchstens $2e^{-2.25\cdot n^{1/4}/(1+5n^{-1/4})} < e^{-n^{1/4}}$ für Fall 2 (für n genügend groß).

Wir fassen zusammen:

Satz 4.4.2
Algorithmus 4.4.1 ist ein selbstverifizierender Algorithmus für das Medianproblem (einschließlich Umarrangieren). Die Anzahl der durchgeführten Vergleiche ist $\frac{3}{2}n + O(n^{3/4} \log n)$, die Versagenswahrscheinlichkeit ist $O(e^{-n^{1/4}})$.

Bemerkung 1: Beim Umbau zu einem Las-Vegas-Algorithmus gemäß Kapitel 3 ergibt sich eine *erwartete* Vergleichsanzahl von $\frac{3}{2}n + O(n^{3/4} \log n)$, da auch mit den zusätzlichen Faktoren $1/(1 - O(e^{-n^{1/4}})) = 1 + O(e^{-n^{1/4}})$ die Vergleichsanzahl in $\frac{3}{2}n + O(n^{3/4} \log n)$ bleibt.

Bemerkung 2: Unter den bekannten Median-Algorithmen gibt es keinen, der die Vergleichsanzahl des Random-Sampling-Algorithmus (im Wesentlichen $\frac{3}{2}n$) schlägt.

Bemerkung 3: Eine Variante der Random-Sampling-Methode führt zu effizienten parallelen Sortieralgorithmen.

A Ergänzung: Abschätzung von Summen durch Integrale

Im Abschnitt über zweifach unabhängige Suche (4.1) haben wir zweimal eine unendliche Reihe durch ein Integral abgeschätzt. Auch bei der genauen Analyse von Quickselect (Abschnitt 4.3) ergab sich die Notwendigkeit für eine solche Abschätzung. Dahinter steckt eine einfache Technik, die es erlaubt, Summen mit monoton fallenden oder monoton wachsenden Summanden einfach und recht genau abzuschätzen. Aus den Ungleichungen lässt sich auch ablesen, wie groß der Schätzfehler maximal ist.

Lemma A.0.1
Sei $f\colon [a,b] \to \mathbb{R}_0^+$ eine Funktion, wobei $a \in \mathbb{N}$ und $b \in \mathbb{N} \cup \{\infty\}$. Dann gilt:

(a) Wenn f monoton wachsend ist und $a < b$ ganze Zahlen sind, gilt:

$$\int_a^b f(x)\,dx \le \sum_{a<i\le b} f(i) \text{ und } \sum_{a\le i<b} f(i) \le \int_a^b f(x)\,dx.$$

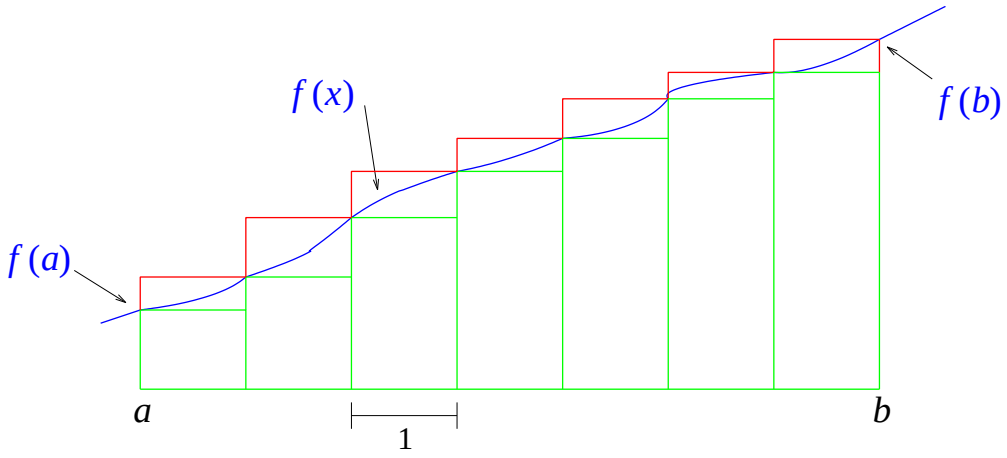
(b) Wenn f monoton fallend ist und $a < b$ ganze Zahlen sind, gilt:

$$\int_a^b f(x)\,dx \le \sum_{a\le i<b} f(i) \text{ und } \sum_{a<i\le b} f(i) \le \int_a^b f(x)\,dx.$$

(c) Wenn f monoton fallend ist, gilt für die unendliche Reihe:

$$\int_a^\infty f(x)\,dx \le \sum_{i\ge a} f(i) \le \int_a^\infty f(x)\,dx + f(a).$$

Beweis: (a) Die Situation ist wie in der folgenden Skizze. Die roten Rechtecke deuten eine Obersumme für das Integral $\int_a^b f(x)\,dx$ an, die grünen Rechtecke eine Untersumme.



Weil f monoton wachsend ist, gilt:

$$\int_{i-1}^{i} f(x)\,dx \le f(i), \text{ für } a < i \le b, \text{ und } f(i) \le \int_{i}^{i+1} f(x)\,dx, \text{ für } a \le i < b. \quad \text{(A.0.6)}$$

Wir summieren die erste Ungleichung in (A.0.6) über $a < i \le b$ und erhalten $\int_a^b f(x)\,dx \le \sum_{a<i\le b} f(i)$. Die zweite Ungleichung summieren wir über $a \le i < b$ und erhalten $\sum_{a\le i<b} f(i) \le \int_a^b f(x)\,dx$.

(b) Weil f monoton fallend ist, gilt:

$$\int_{i}^{i+1} f(x)\,dx \le f(i), \text{ für } a \le i < b, \text{ und } f(i) \le \int_{i-1}^{i} f(x)\,dx, \text{ für } a < i \le b. \quad \text{(A.0.7)}$$

Wir summieren die erste Ungleichung in (A.0.7) über $a \le i < b$ und erhalten $\int_a^b f(x)\,dx \le \sum_{a\le i<b} f(i)$. Die zweite Ungleichung summieren wir über $a < i \le b$ und erhalten $\sum_{a<i\le b} f(i) \le \int_a^b f(x)\,dx$.

(c) Wenn $b = \infty$, summieren wir die erste Ungleichung in (A.0.7) über alle $i \ge a$ und die zweite über alle $i > a$, um die behaupteten Ungleichungen zu erhalten. □

Beispiele A.0.2
(i) Mit $f(x) = 1/x$ und $a = 1$ und $b = n + 1$ erhalten wir mit Lemma A.0.1(b) obere und untere Schranken für $H_n = \sum_{1\le i\le n} \frac{1}{i}$:

$$\boxed{\ln n < \ln(n+1) = \int_1^{n+1} \frac{dx}{x} \le H_n \le 1 + \ln n.}$$

(ii) Mit $f(x) = 1/x$ und $0 \le a < b$ beliebig gilt, nach Lemma A.0.1(b):

$$\sum_{a<i\le b} \frac{1}{i} \le \int_a^b \frac{dx}{x} = \int_1^b \frac{dx}{x} - \int_1^a \frac{dx}{x} = \ln b - \ln a = \ln\left(\frac{b}{a}\right).$$

(iii) Mit $f(x) = 1/x^2$ und $b = \infty$ erhalten wir mit Lemma A.0.1(c):

$$\int_a^\infty f(x)\,dx = \left[-1/x\right]_a^\infty = \frac{1}{a} \le \sum_{i\ge a} \frac{1}{i^2} \le \int_a^\infty f(x)\,dx + f(a) = \frac{1}{a} + \frac{1}{a^2}.$$

(iv) Mit $f(x) = \ln(x)$ und $a = 1$ und $b = n$ können wir Lemma A.0.1(a) anwenden. Beachte, dass $\int \ln x \, dx = x(\ln x - 1) + C$, also $\int_1^n \ln(x)\, dx = n(\ln n - 1) + 1$. Wir erhalten:

$$n(\ln n - 1) + 1 + \ln 1 \leq \sum_{1 \leq i \leq n} \ln i \text{ , d.h. } n(\ln n - 1) + 1 \leq \sum_{1 \leq i \leq n} \ln i = \ln(n!).$$

Daraus schließen wir:

$$\boxed{n! \geq e^{n(\ln n - 1)+1} = e \cdot \left(\frac{n}{e}\right)^n .}$$

Weiter ergibt sich (zweite Ungleichung in Lemma A.0.1(a)):

$$\sum_{1 \leq i \leq n} \ln i \leq n(\ln n - 1) + 1 + \ln n \text{ , d.h. } \ln(n!) \leq n(\ln n - 1) + 1 + \ln n.$$

Daraus schließen wir:

$$\boxed{n! \leq e^{n(\ln n - 1)+1+\ln n} = (en) \cdot \left(\frac{n}{e}\right)^n .}$$

Dies sind schon (für den geringen Aufwand) recht ordentliche Abschätzungen für $n!$.

Eine viel schärfere Abschätzung bietet die *Stirling'sche Formel*, grob $n! \sim \sqrt{2\pi n}(n/e)^n$, die zeigt, dass der wahre Wert von $n!$ (geometrisch) zwischen $e(n/e)^n$ und $(en)(n/e)^n$ liegt, und dass die wirkliche Konstante $\sqrt{2\pi}$ ist. Abschätzungen, die auf etwa $1 + \frac{1}{144n^2}$ genau sind, bieten die folgenden Ungleichungen, die für alle $n \geq 1$ gelten:

$$\boxed{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n+1}} \;\leq\; n! \;\leq\; \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}.}$$

5 Randomisierte Algorithmen für Probleme aus der Zahlentheorie

Zu Aussagen, die mit $(*)$ markiert sind, gibt es Beweise oder Anmerkungen im Anhang A.

In diesem Kapitel betrachten wir randomisierte Algorithmen für Probleme, die mit dem Rechnen mit ganzen Zahlen zu tun haben. Dies sind Primzahltests, ein darauf aufbauendes Verfahren zur Erzeugung von zufälligen Primzahlen großer Bitlänge sowie ein Verfahren zum Ziehen von Quadratwurzeln modulo p, wo p eine Primzahl ist. Insbesondere wegen ihrer Anwendung in der Kryptographie haben diese Verfahren sehr große praktische Bedeutung. Zunächst diskutieren wir die nötigen Grundkonzepte aus der Zahlentheorie und grundlegende effiziente Algorithmen für natürliche und ganze Zahlen.

5.1 Fakten aus der Zahlentheorie und grundlegende Algorithmen

5.1.1 Teiler, Reste, Euklidischer Algorithmus

Die in diesem Kapitel relevanten Zahlenbereiche sind:

- $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$, die natürlichen Zahlen, und
- $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, 3, \ldots\}$, die ganzen Zahlen.

In den Diskussionen in diesem Kapitel sehen wir ganze Zahlen unter zwei Blickwinkeln an: Erstens ist es die unendliche Menge $\mathbb{Z}$, die mit ihren abstrakten Operationen $+$, $-$, $\cdot$ und $/$ und den neutralen Elementen 0 und 1 einen *Ring mit* 1 bildet. Über diese Struktur und Eigenschaften von ganzen Zahlen können wir *mathematische Aussagen* machen und beweisen. Andererseits interessieren uns *Algorithmen* für Zahlen. Für Betrachtungen aus dieser Perspektive stellen wir uns die Zahlen immer als zu

einer passenden Basis b dargestellt vor: Binärdarstellung, Dezimaldarstellung, Hexadezimaldarstellung, Darstellung zur Basis 256 (eine Ziffer ist ein Byte) oder 2^{32} (eine Ziffer ist ein 32-Bit-Wort, passend für die Darstellung in einem Rechner). Eine Zahl ist dann als ein String über einem endlichen Alphabet gegeben. Für Rechenzeitbetrachtungen ist es nicht wichtig, ob negative Zahlen als Paar aus Vorzeichen und Absolutbetrag oder in Zweierkomplementdarstellung dargestellt werden. Die Anzahl der Ziffern in der Darstellung von $a \in \mathbb{N}$ zur Basis b ist $\lceil \log_b(a+1) \rceil$.

Wir nehmen an, dass Algorithmen für die folgenden Grundoperationen auf ganzen Zahlen gegeben sind: Addition, Subtraktion, Multiplikation, Division (mit Rest). Zwei einziffrige Zahlen können in Zeit $O(1)$ addiert und subtrahiert werden („von der Hardware“). Addition und Subtraktion zweier n-ziffriger Zahlen kosten Zeit $O(n)$, Multiplikation einer n-ziffrigen und einer ℓ-ziffrigen Zahl (mit der Schulmethode) kostet Zeit $O(n\ell)$, ebenso Division einer n-ziffrigen durch eine ℓ-ziffrige Zahl. Dabei entspricht „Rechenzeit“ einfach der Anzahl der ausgeführten Operationen auf einzelnen Ziffern.[1]

Eine Zahl $x \in \mathbb{Z}$ heißt **Teiler** von $y \in \mathbb{Z}$, wenn es ein $z \in \mathbb{Z}$ mit $y = x \cdot z$ gibt. Notation: $x \mid y$. Wir sagen dann auch, dass y von x geteilt wird oder dass y ein Vielfaches von x ist. Wenn x nicht Teiler von y ist, schreibt man $x \nmid y$.

Beachte: Die Teiler von 1 sind 1 und -1. Bezüglich der Teilbarkeitsrelation $x \mid y$ gibt es keinen Unterschied zwischen x und $-x$ und zwischen y und $-y$. Die Teilbarkeitsrelation ist reflexiv und transitiv: $x \mid x$ und $(x \mid y \wedge y \mid z) \Rightarrow x \mid z$. Weiter gilt: Wenn x Teiler von $y \neq 0$ ist, dann gilt $|x| \leq |y|$. Die Teilbarkeitsrelation wird in Abb. 5.1.1 anschaulich dargestellt.

Eine immer wieder benutzte Eigenschaft ist, dass die Menge $V_x = \{y \in \mathbb{Z} \mid x \text{ teilt } y\}$ der Vielfachen von x unter Addition und Multiplikation mit beliebigen Zahlen abgeschlossen ist, d. h.:[2]

$$x \mid y \ \wedge \ x \mid z \quad \Rightarrow \quad x \mid (sy + tz), \text{ für beliebige } s, t \in \mathbb{Z}. \tag{5.1.1}$$

[1] Es gibt asymptotisch schnellere Verfahren: Der Karatsuba-Algorithmus (s. Vorlesung „Algorithmen und Datenstrukturen“) mit Rechenzeit $O(n^{1,59})$ für zwei n-ziffrige Zahlen, der Algorithmus von Schönhage und Strassen sogar mit Rechenzeit $O(n \log n \log\log n)$. Im Jahr 2019 erschien eine Arbeit [David Harvey and Joris van der Hoeven, Integer multiplication in time $O(n \log n)$, Proceedings of the Thirteenth Algorithmic Number Theory Symposium, S. 293–310, 2019. `http://dx.doi.org/10.2140/obs.2019.2.293`], die zeigt, dass man zwei n-Bit-Zahlen in Zeit $O(n \log n)$ multiplizieren kann. Nach aktuellem Stand ergeben sich Vorteile aber erst für unrealistisch lange Zahlen. Wir werden diese „schnellen“ Algorithmen nicht berücksichtigen.

[2] Man sagt: V_x ist ein *Ideal* in $\mathbb{Z}$.

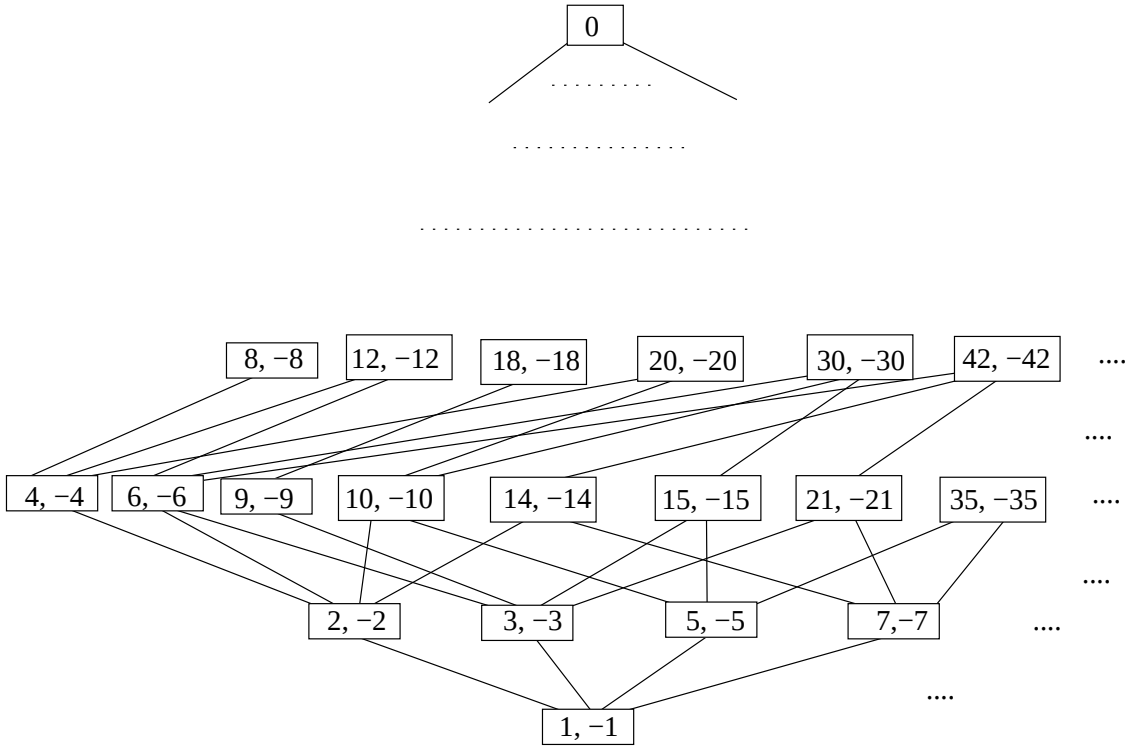


Abbildung 5.1.1: Teilbarkeitsrelation, Ausschnitt. Eine Linie verbindet x (unten) mit y (weiter oben), wenn $x \mid y$ und $|x| \neq |y|$ gilt und wenn es kein z gibt, das $|z| < |y|$ und $x \mid z$ und $z \mid y$ erfüllt.

Fakt 5.1.1 ***Division mit Rest***
Zu $x \in \mathbb{Z}$ und $m \geq 2$ gibt es ein r mit $0 \leq r < m$ und ein q mit $x = qm + r$. Die Zahlen q und r sind eindeutig bestimmt.

Beispiele für $m = 4$: $23 = 5 \cdot 4 + 3$, $-12 = (-3) \cdot 4 + 0$, $-23 = (-6) \cdot 4 + 1$.

Die Zahl $r = x - qm$ („**Rest**“) bezeichnen wir mit $\boldsymbol{x}$ **mod** $\boldsymbol{m}$. Beachte, dass $x - r$ ein Vielfaches von m ist. Der „**Quotient**“ $q = \lfloor x/m \rfloor$ wird auch mit $\boldsymbol{x}$ **div** $\boldsymbol{m}$ bezeichnet. Der Divisionsalgorithmus für Zahlen in Dezimaldarstellung, wie er in der Schule gelehrt wird, berechnet aus Eingabe x und m Quotient q und Rest r, in Rechenzeit $O((\log x)(\log m))$.

Wir betrachten nun die Menge $\mathbb{Z}_m = [m] = \{0, 1, \ldots, m-1\}$ und Operationen

- Addition modulo m: $+_m \colon \mathbb{Z}_m^2 \ni (x, y) \mapsto x +_m y := (x + y) \bmod m \in \mathbb{Z}_m$, und
- Multiplikation modulo m: $\cdot_m \colon \mathbb{Z}_m^2 \ni (x, y) \mapsto x \cdot_m y := xy \bmod m \in \mathbb{Z}_m$.

Fakt 5.1.2 $(*)$
Für jedes $m \geq 2$ bildet $\mathbb{Z}_m$ mit $+_m$ und $\cdot_m$ einen *kommutativen Ring mit* 1. Die neutralen Elemente für $+_m$ und $\cdot_m$ sind 0 und 1.

Das heißt im Detail: Die Operationen $+_m$ und $\cdot_m$ führen nicht aus dem Bereich $\mathbb{Z}_m$ heraus. Die Addition erfüllt alle Rechenregeln für abelsche Gruppen, mit neutralem Element 0. Insbesondere hat jedes Element $x \in \mathbb{Z}_m$ ein additives Inverses $-x$ (beachte $-x = m - x$ für $0 < x < m$ und $-0 = 0$). Die Multiplikation ist assoziativ und kommutativ. Die 1 ist neutrales Element. Für Addition und Multiplikation gelten die Distributivgesetze.

Lemma 5.1.3 $(*)$
Für jedes $m \geq 2$ ist die Abbildung $\mathbb{Z} \to \mathbb{Z}_m$, $x \mapsto x \bmod m$, ein *Homomorphismus*; d. h. für $x, y \in \mathbb{Z}$ gilt:

(i) $(x + y) \bmod m = (x \bmod m) +_m (y \bmod m)$,

(ii) $xy \bmod m = (x \bmod m) \cdot_m (y \bmod m)$.

Dieses Lemma wird hauptsächlich benutzt, um Rechnungen zu vereinfachen. Um einen arithmetischen Ausdruck modulo m auszuwerten (inklusive Potenzen), kann man auf beliebige Zwischenergebnisse die (mod m)-Operation anwenden.

Beispiel: Um $13^7 \bmod 11$ zu berechnen, rechnet man

$$\begin{aligned}(13 \bmod 11)^7 \bmod 11 = 2^7 \bmod 11 &= ((2^5 \bmod 11) \cdot 4) \bmod 11 \\ &= 10 \cdot 4 \bmod 11 = 40 \bmod 11 = 7.\end{aligned}$$

Fakt 5.1.4 $(*)$
Für jedes $m \geq 2$ und alle $x, y \in \mathbb{Z}$ gilt:
$x \bmod m = y \bmod m$ genau dann wenn $m \mid x - y$.

Beispiel: $29 \bmod 12 = 53 \bmod 12 = 5$, und $53 - 29 = 24$ ist durch 12 teilbar.

Schreibweise: $x \equiv y \pmod m$ (oder „$x \equiv_m y$“ oder „$x \equiv y\,(m)$“) bedeutet, dass $x - y$ durch m teilbar ist. Man sagt: „x ist *kongruent* y modulo m.“ Aus Fakt 5.1.4 folgt sofort, dass $\equiv_m$ eine Äquivalenzrelation ist. (Sie ist reflexiv, symmetrisch und transitiv.)

Fakt 5.1.5
Für jedes $m \geq 2$ und alle ganzen Zahlen x_1, x_2, y_1, y_2 gilt:

$$x_1 \equiv_m y_1 \text{ und } x_2 \equiv_m y_2 \quad \Rightarrow \quad x_1 + x_2 \equiv_m y_1 + y_2 \text{ und } x_1 \cdot x_2 \equiv_m y_1 \cdot y_2.$$

Dies hat den Effekt, dass man bei mit „$\equiv$“ geschriebenen Transformationen von $\{+, -, \cdot\}$-Ausdrücken stets Zahlen oder Unterausdrücke durch kongruente Zahlen bzw. Unterausdrücke ersetzen kann. Dies führt zu größerer Flexibilität und daher u. U. einfacheren Rechnungen als nur die Rechnung mit Resten in $[m]$, wie von Lemma 5.1.3 erlaubt.

Beispiel: $13^7 \equiv 2^7 \equiv 2^5 \cdot 2^2 = 32 \cdot 4 \equiv (-1) \cdot 4 = -4 \equiv 7 \pmod{11}$.

Definition 5.1.6 ***Größter gemeinsamer Teiler***
Für $x, y \in \mathbb{Z}$ ist $\mathrm{ggT}(x, y)$, der *größte gemeinsame Teiler* von x und y, die (eindeutig bestimmte) *nichtnegative* Zahl d mit:

$$\text{(i) } d \mid x \text{ und } d \mid y; \quad \text{(ii) wenn } c \mid x \text{ und } c \mid y \text{ gilt, dann folgt } c \mid d.$$

Zahlen d, die nur (i) erfüllen, heißen *gemeinsame Teiler* von x und y. Wenn x und y nicht beide 0 sind, beschreibt die Definition den größten gemeinsamen Teiler im Standardsinn, d. h. die größte ganze Zahl, die sowohl x als auch y teilt. Weiter folgt aus der Definition[3]: $\mathrm{ggT}(0, 0) = 0$. Zwei Zahlen x und y mit $\mathrm{ggT}(x, y) = 1$ heißen ***teilerfremd***. Der größte gemeinsame Teiler zweier Zahlen ist eindeutig bestimmt.

Beispiele: Die gemeinsamen Teiler von 0 und -3 sind $-3, -1, 1, 3$, und es gilt $\mathrm{ggT}(0, -3) = 3$.

[3]Die Teilbarkeitsrelation $\mid$ ist reflexiv und transitiv; man nennt so etwas (partielle) *Quasiordnung* oder *Präordnung*. Bezüglich dieser partiellen Quasiordnung sind 1 und -1 kleinste Elemente (sie teilen jede ganze Zahl) und 0 ist größtes Element (jede ganze Zahl teilt die 0), vgl. Abb. 5.1.1. Im Sinn dieser partiellen Quasiordnung ist 0 also das größte Element, das $x = 0$ und $y = 0$ teilt.

Die gemeinsamen Teiler von -30 und 24 sind $-6, -3, -2, -1, 1, 2, 3, 6$, und es gilt $\mathrm{ggT}(-30, 24) = 6$.
Die gemeinsamen Teiler von -12 und 35 sind $-1, 1$, also gilt $\mathrm{ggT}(-12, 35) = 1$; die beiden Zahlen sind teilerfremd.

Es gibt einen wohlbekannten effizienten Algorithmus zur Ermittlung des größten gemeinsamen Teilers von zwei Zahlen. (Dieser beweist auch die Existenz.) Weil Teilbarkeit nicht vom Vorzeichen der Argumente abhängt, gilt $\mathrm{ggT}(x, y) = \mathrm{ggT}(|x|, |y|)$; daher kann man sich auf den Fall $x, y \geq 0$ beschränken. Weiter gelten die Gleichungen

$$\begin{aligned}
&\mathrm{ggT}(a, 0) = a \text{ , für beliebige } a \geq 0,\\
&\mathrm{ggT}(a, b) = \mathrm{ggT}(b, a) \text{ und}\\
&\mathrm{ggT}(a, b) = \mathrm{ggT}(b, a \bmod b) \text{ , für } b > 0. \qquad (\text{Und: } a \bmod b < b.)
\end{aligned}$$

(Die beiden ersten Gleichungen sind offensichtlich. Für die dritte überlegt man sich Folgendes. Wir haben $a \bmod b = a - qb$ für $q = \lfloor a/b \rfloor$. Daraus folgt mit (5.1.1), dass jeder gemeinsame Teiler von a und b auch $a \bmod b = a - qb$ teilt und dass jeder gemeinsame Teiler von b und $a \bmod b$ auch $a = qb + (a \bmod b)$ teilt. Also haben die Zahlenpaare (a, b) und $(b, a \bmod b)$ dieselbe Menge gemeinsamer Teiler, und damit auch denselben größten gemeinsamen Teiler.)

Diese Gleichungen liefern sofort einen rekursiven Algorithmus für den ggT, aber sie führen auch leicht zu einem iterativen Verfahren.

Algorithmus 5.1.7 ***Euklidischer Algorithmus***
Input: Zwei natürliche Zahlen x und y.
Methode:

```
1   a, b: integer; a ← x; b ← y;
2   while b > 0 repeat
3       (a, b) ← (b, a mod b);      // simultane Zuweisung
4   return a.
```

Fakt 5.1.8 $(*)$
Algorithmus 5.1.7 gibt $\mathrm{ggT}(x, y)$ aus. Die Gesamtzahl von ausgeführten Ziffernoperationen ist $O((\log x)(\log y))$.

Man beachte, dass $\lceil \log(x + 1) \rceil \approx \log x$ die Anzahl der Bits ist, die man braucht, um x aufzuschreiben. Damit hat der Euklidische Algorithmus bis auf einen konstan-

i	a_i	b_i
0	10534	12742
1	12742	10534
2	10534	2208
3	2208	1702
4	1702	506
5	506	184
6	184	138
7	138	46
8	46	0

Tabelle 1: Ablauf des Euklidischen Algorithmus auf Eingabe $x = 10534$, $y = 12742$.

ten Faktor denselben Aufwand wie die Multiplikation von x und y, wenn man die Schulmethode benutzt.

Beispiel: Auf Eingabe $x = 10534$, $y = 12742$ ergibt sich der in Tab. 1 angegebene Ablauf. Die Zahlen a_i und b_i bezeichnen den Inhalt der Variablen `a` und `b`, nachdem die Schleife in Zeilen 2–3 i-mal ausgeführt worden ist. Die Ausgabe ist $46 = \mathrm{ggT}(10534, 12742)$.

Beispiel: (a) 21 und 25 sind teilerfremd. Es gilt $31 \cdot 21 + (-26) \cdot 25 = 651 - 650 = 1$.
(b) Es gilt $\mathrm{ggT}(21, 35) = 7$, und $2 \cdot 35 - 3 \cdot 21 = 7$.

Die folgende sehr nützliche Aussage verallgemeinert diese Beobachtung:

Lemma 5.1.9 ***... von Bezout*** $(*)$
(a) Wenn $x, y \in \mathbb{Z}$ teilerfremd sind, gibt es $s, t \in \mathbb{Z}$ mit $sx + ty = 1$.
(b) Für $x, y \in \mathbb{Z}$ gibt es s, t mit $sx + ty = \mathrm{ggT}(x, y)$.

Beweis: (a) folgt direkt aus (b). Also müssen wir nur (b) beweisen. Wenn $x = y = 0$, gilt $sx + ty = 0 = \mathrm{ggT}(x, y)$ für alle $s, t \in \mathbb{Z}$. Seien also ab hier nicht x und y beide 0. Wir definieren $I_{x,y} := \{sx + ty \mid s, t \in \mathbb{Z}\}$. Diese Menge enthält die Zahlen $|x| \geq 0$ (setze $s = \mathrm{sign}(x)$ und $t = 0$) und $|y| \geq 0$ (setze $s = 0$ und $t = \mathrm{sign}(y)$), und sie hat folgende „Abschlusseigenschaft“:

Wenn $a \geq b > 0$ und $a, b \in I_{x,y}$, dann gilt auch $(a \bmod b) \in I_{x,y}$.

(Wir haben $a = s_a x + t_a y$ und $b = s_b x + t_b y$ für passende Koeffizienten und $a \bmod b = a - qb$ für den Quotienten $\lfloor a/b \rfloor$. Dann gilt $a \bmod b = (s_a - qs_b)x + (t_a - qt_b)y$, also $(a \bmod b) \in I_{x,y}$.) Man betrachtet nun den Ablauf des Euklidischen Algorithmus 5.1.7, gestartet mit $a = |x| \in I_{x,y}$ und $b = |y| \in I_{x,y}$. Nach der Abschlusseigenschaft sind sämtliche dabei entstehenden Zwischenwerte Elemente von $I_{x,y}$, also auch der schließlich entstehende Wert $\mathrm{ggT}(x, y)$. □

Die Überlegungen in diesem Beweis führen direkt zu einem sehr effizienten Algorithmus, dem Erweiterten Euklidischen Algorithmus, der zu x und y die Koeffizienten s und t aus dem Lemma von Bezout berechnet. Die Rechenzeiten des folgenden Algorithmus sind (in O-Notation) ebenso groß wie die des gewöhnlichen Euklidischen Algorithmus.[4]

Algorithmus 5.1.10 ***Erweiterter Euklidischer Algorithmus***
EINGABE: Zahlen $x, y \geq 0$.
METHODE:

```
0    a, b, s_a, t_a, s_b, t_b, q: integer;
1    (a, b) ← (x, y);
2    (s_a, t_a, s_b, t_b) ← (1, 0, 0, 1);
3    while b > 0 repeat
4        q ← a div b;
5        (a, b) ← (b, a − q · b);
6        (s_a, t_a, s_b, t_b) ← (s_b, t_b, s_a − q · s_b, t_a − q · t_b);
7    return (a, s_a, t_a);
```

Genau wie im (einfachen) Euklidischen Algorithmus findet die eigentliche Arbeit in der **while**-Schleife statt (4 Zeilen). Wie dort werden in den Variablen `a` und `b` Zahlen a, $b \geq 0$ mitgeführt, die stets $\mathrm{ggT}(a, b) = d = \mathrm{ggT}(x, y)$ erfüllen. Die Variablen $\mathtt{s_a}$, $\mathtt{t_a}$, $\mathtt{s_b}$, $\mathtt{t_b}$ enthalten stets Zahlen s_a, t_a, s_b, t_b, die folgende Gleichungen erfüllen:

$$a = s_a \cdot x + t_a \cdot y \quad \text{und} \quad b = s_b \cdot x + t_b \cdot y.$$

Diese Gleichung wird durch die Initialisierung hergestellt. In einem Schleifendurchlauf wird a durch b ersetzt und (s_a, t_a) durch (s_b, t_b), und es wird b durch $a - q \cdot b$ ersetzt

[4] Nur um die Notation etwas zu vereinfachen, nehmen wir $x, y \geq 0$ an. Es ist kein Problem, den Algorithmus so zu modifizieren, dass beliebige ganze Zahlen x, y verarbeitet werden.

sowie (s_b, t_b) durch $(s_a - q \cdot s_b, t_a - q \cdot t_b)$. Dadurch bleiben die Gleichungen gültig. Wenn schließlich $b = 0$ geworden ist, gilt $d = \mathrm{ggT}(x, y) = a = s_a \cdot x + t_a \cdot y$. Das bedeutet, dass die Ausgabe das gewünschte Ergebnis darstellt. Als Beispiel betrachten wir den Ablauf des Algorithmus auf der Eingabe $(x, y) = (10534, 12742)$. Die Zahlen a_i ,b_i, $s_{a,i}$, $t_{a,i}$, $s_{b,i}$, $t_{b,i}$ bezeichnen den Inhalt von `a`, `b`, $\mathtt{s_a}$, $\mathtt{t_a}$, $\mathtt{s_b}$, $\mathtt{t_b}$ nach dem i-ten Schleifendurchlauf. Die Zahl q_i ist der Quotient im i-ten Durchlauf.

i	a_i	b_i	$s_{a,i}$	$t_{a,i}$	$s_{b,i}$	$t_{b,i}$	q_i
0	10534	12742	1	0	0	1	–
1	12742	10534	0	1	1	0	0
2	10534	2208	1	0	−1	1	1
3	2208	1702	−1	1	5	−4	4
4	1702	506	5	−4	−6	5	1
5	506	184	−6	5	23	−19	3
6	184	138	23	−19	−52	43	2
7	138	46	−52	43	75	−62	1
8	46	0	75	−62	−277	229	3

Die Ausgabe ist $(46, 75, -62)$. Man überprüft leicht, dass $46 = \mathrm{ggT}(10534, 12742) = 75 \cdot 10534 - 62 \cdot 12742$ gilt.

Fakt 5.1.11

(a) Wenn Algorithmus 5.1.10 auf Eingabe (x, y) die Ausgabe (d, s, t) liefert, dann gilt $d = \mathrm{ggT}(x, y) = sx + ty$.

(b) Die Anzahl der Schleifendurchläufe ist dieselbe wie beim gewöhnlichen Euklidischen Algorithmus.

(c) Die Anzahl von Ziffernoperationen für Algorithmus 5.1.10 ist $O((\log x)(\log y))$.

Wir notieren noch eine wichtige Folgerung aus dem Lemma von Bezout.[5]

[5] In der Schule begründet man diese Tatsache mit Hilfe der Primzahlzerlegung. Dieser Umweg ist aber gar nicht nötig.

Fakt 5.1.12
Wenn x und y teilerfremd sind und a sowohl durch x als auch durch y teilbar ist, dann ist a auch durch xy teilbar.

Beweis: Weil x und y Teiler von a sind, kann man $a = ux$ und $a = vy$ schreiben, für ganze Zahlen u, v. Weil x und y teilerfremd sind, folgt aus Lemma 5.1.9, dass man $1 = \mathrm{ggT}(x, y) = sx + ty$ schreiben kann, für ganze Zahlen s, t. Dann ist $a = asx + aty = vysx + uxty = (vs + ut)xy$, also ist xy Teiler von a. □

5.1.2 Eigenschaften der Multiplikation in $\mathbb{Z}_m$

In $\mathbb{Z}_m$ spielen die Elemente, die bezüglich $\cdot_m$ ein *multiplikatives Inverses* haben, eine spezielle Rolle.

Lemma 5.1.13
Für jedes $m \geq 2$ und jedes $a \in \mathbb{Z}$ gilt:
Es gibt ein b mit $ab \bmod m = 1$ genau dann wenn $\mathrm{ggT}(a, m) = 1$.

Beweis: „⇒“: Weil $ab \bmod m = 1$ gilt, können wir $ab = qm + 1$ schreiben, für ein $q \in \mathbb{Z}$. Wenn nun eine Zahl d Teiler von a und von m ist, dann teilt d auch $ab - qm = 1$, also ist $d \in \{-1, 1\}$. Daraus folgt $\mathrm{ggT}(a, m) = 1$.
„⇐“: Weil $\mathrm{ggT}(a, m) = 1$ gilt, können wir nach dem Lemma von Bezout $sa + tm = 1$ schreiben, für passende ganze Zahlen s und t. Setze $b = s \bmod m$. Dann gilt $ab = a(s - \lfloor s/m \rfloor m) = 1 - (t + a\lfloor s/m \rfloor)m$, also gilt $ab \bmod m = 1$. □

Wegen der Homomorphieeigenschaft (Lemma 5.1.3) kommt es für die Frage, ob $ab \bmod m = 1$ gilt, nur auf $a \bmod m$ und $b \bmod m$ an. Wir können unsere Überlegungen also auf Elemente von $\mathbb{Z}_m$ beschränken. Wenn $a, b \in \mathbb{Z}_m$ und $ab \bmod m = 1$, nennen wir b ein *multiplikatives Inverses* zu a modulo m.

Beispiel: Aus $6 \cdot 21 + (-5) \cdot 25 = 1$ folgt, dass 6 ein multiplikatives Inverses zu 21 modulo 25 ist.

Definition 5.1.14
Für $m \geq 2$ sei $\mathbb{Z}_m^* := \{a \in \mathbb{Z}_m \mid \mathrm{ggT}(a, m) = 1\}$.

Fakt 5.1.15 $(*)$
Für jedes $m \geq 2$ gilt:
$\mathbb{Z}_m^*$ mit der Multiplikation modulo m als Operation ist eine abelsche Gruppe.

Beispiel: $\mathbb{Z}_{21}^* = \{1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20\}$ und $\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$.

Am schönsten ist die Situation, wenn alle Zahlen aus $\mathbb{Z}_m$ außer der 0 in $\mathbb{Z}_m^*$ liegen. Dies ist genau dann der Fall, wenn m Primzahl ist.

Definition 5.1.16

(a) Eine Zahl $p \geq 1$ heißt **Primzahl**, wenn p genau zwei positive Teiler hat.[a] (Diese Teiler sind dann 1 und p.)

(b) Eine Zahl $x \geq 1$ heißt **zusammengesetzt**, wenn sie einen Teiler y mit $1 < y < x$ besitzt.[b]

[a] Die Zahl 1 hat nur einen positiven Teiler, nämlich 1. Also ist 1 keine Primzahl.
[b] Die Zahl 1 besitzt keinen Teiler y mit $1 < y < 1$. Also ist 1 auch nicht zusammengesetzt.

Proposition 5.1.17 $(*)$
Für jedes $m \geq 2$ gilt:
m ist Primzahl $\Leftrightarrow$ $\mathbb{Z}_m^* = \{1, \ldots, m-1\}$ $\Leftrightarrow$ $\mathbb{Z}_m$ ist ein Körper.

Die zweite Äquivalenz ist dabei klar: Der Ring $\mathbb{Z}_m$ ist nach Definition ein Körper genau dann wenn jedes Element von $\mathbb{Z}_m - \{0\}$ ein multiplikatives Inverses besitzt. Für die erste Äquivalenz (die auch einfach einzusehen ist) siehe Anhang A.

Beispiel: Wir rechnen modulo 13 und geben für jedes $a \in \mathbb{Z}_{13}^*$ das Inverse b sowie das Produkt $a \cdot b$ an (das natürlich bei der Division durch 13 Rest 1 lassen muss).

a	1	2	3	4	5	6	7	8	9	10	11	12
b	1	7	9	10	8	11	2	5	3	4	6	12
$a \cdot b$	1	14	27	40	40	66	14	40	27	40	66	144

14 ist keine Primzahl, und es gibt keine Zahl b mit $2 \cdot b \bmod 14 = 1$; das heißt, dass $2 \notin \mathbb{Z}_{14}^*$ und daher, dass $\mathbb{Z}_{14}$ kein Körper ist.

Basis für randomisierte Primzahltests ist das folgende grundlegende Ergebnis. Wir beginnen mit einem Beispiel, für die Primzahlen $p = 7$ und $p = 257$.

$$\begin{aligned} 5^6 \equiv (-2)^6 \equiv 64 \equiv 1 \qquad & (\text{mod } 7); \\ 3^6 \equiv 9^3 \equiv 2^3 \equiv 8 \equiv 1 \qquad & (\text{mod } 7); \\ 2^{256} \equiv (2^8)^{32} \equiv 256^{32} \equiv (-1)^{32} \equiv 1 \qquad & (\text{mod } 257). \end{aligned}$$

Dass hier stets das Ergebnis 1 entsteht, ist kein Zufall.

Fakt 5.1.18 ***Kleiner Satz von Fermat***
Wenn p eine Primzahl ist, dann gilt:

$$a^{p-1} \bmod p = 1, \text{ für jedes } a \in \mathbb{Z} \text{ mit } \mathrm{ggT}(a,p) = 1 \text{ (d. h. } p \not| \, a).$$

Beweis: Da sich a und $a \bmod p$ bezüglich Rechnungen modulo p gleich verhalten, können wir $a \in \{1, \ldots, p-1\}$ annehmen. (Es gilt $a \bmod p \neq 0$, weil a nicht durch p teilbar ist.) Betrachte die Abbildung

$$g_a \colon \mathbb{Z}_p^* \ni x \mapsto ax \bmod p \in \mathbb{Z}_p^*.$$

(Beispiel: Wenn man mit $a = 4$ modulo $p = 7$ rechnet, ist $(g_4(1), \ldots, g_4(6)) = (4, 1, 5, 2, 6, 3)$. Man sieht, dass alle Zahlen in $\{1, \ldots, 6\}$ im Bild von g_4 vorkommen.) Die Abbildung g_a ist injektiv, da gilt: $a^{-1} \cdot g_a(x) \bmod p = a^{-1}ax \bmod p = x$. Also gilt

$$\{1, \ldots, p-1\} = \{a \cdot_p 1, \ldots, a \cdot_p (p-1)\}.$$

Wir multiplizieren, in $\mathbb{Z}_p$, alle Elemente der Menge $\{1, \ldots, p-1\}$, in zwei verschiedenen Reihenfolgen, und gruppieren dann um:

$$1 \cdot_p \ldots \cdot_p (p-1) = (a \cdot_p 1) \cdot_p \ldots \cdot_p (a \cdot_p (p-1)) = a^{p-1} \cdot_p (1 \cdot_p \ldots \cdot_p (p-1)).$$

Weil p eine Primzahl ist, ist nach Fakt 5.1.15 die Zahl $P := 1 \cdot_p \ldots \cdot_p (p-1)$ ein Element von $\mathbb{Z}_p^*$. Daher existiert P^{-1}, das Inverse in dieser Gruppe. Wenn wir beide Seiten der letzten Gleichung mit P^{-1} multiplizieren, erhalten wir $1 = a^{p-1} \bmod p$. □

Wir bemerken, dass auch die Umkehrung des kleinen Satzes von Fermat gilt. Wenn m keine Primzahl ist, dann gibt es ein $a \in \{1, \ldots, m-1\}$ mit $\mathrm{ggT}(a,m) \neq 1$. Für jede solche Zahl gilt $a^{m-1} \bmod m \neq 1$. (Wäre $a^{m-1} \bmod m = 1$, hätten wir $a \cdot a^{m-2} - qm = 1$ für ein q, also $\mathrm{ggT}(a,m) = 1$.)

Ab hier soll p immer eine Primzahl bezeichnen. Normalerweise ist dabei $p > 2$, also p ungerade.[6]

Oft kommt es darauf an, Potenzen $x^y \bmod m$ zu berechnen, für ein $y \geq 0$. Dabei kann y eine sehr große Zahl sein. *Beispiel* für eine solche Aufgabe: Berechne

$$3^{1384788374932954500363985403554603584759089} \bmod 28374610730064817362847320073310723412 34.$$

Der Exponent hat 43 Dezimalziffern und seine Binärdarstellung hat 140 Bits. Ein Computeralgebraprogramm berechnet das Ergebnis[7] in Sekundenbruchteilen. Wie kann das gehen? Sicherlich nicht durch y-faches Multiplizieren! Es gibt eine einleuchtende rekursive Formel, die formal auf Lemma 5.1.3 beruht und die den Weg zu einer effizienten Berechnung weist:[8]

$$x^y \bmod m = \begin{cases} 1, & \text{wenn } y = 0, \\ x \bmod m, & \text{wenn } y = 1, \\ ((x^2 \bmod m)^{y/2}) \bmod m, & \text{wenn } y \geq 2 \text{ gerade ist}, \\ ((x^2 \bmod m)^{(y-1)/2} \bmod m) \cdot x \bmod m, & \text{wenn } y \geq 2 \text{ ungerade ist}. \end{cases}$$

Diese Formel führt unmittelbar zu folgender rekursiver Prozedur.

Algorithmus 5.1.19 ***Schnelle modulare Exponentiation***
function modexp(x, y, m) // berechnet $z = x^y \bmod m$
EINGABE: Ganze Zahlen x, $y \geq 0$, und $m \geq 2$, mit $0 \leq x < m$.
METHODE:

```
0   if y = 0 then return 1;
1   if y = 1 then return x;
2   z ← modexp(x · x mod m, ⌊y/2⌋, m);  // rekursiver Aufruf
3   if y ist ungerade then z ← z · x mod m
4   return z.
```

Um $x^y \bmod m$ für beliebige x zu berechnen, ruft man modexp$(x \bmod m, y, m)$ auf.

Was macht dieser Algorithmus anschaulich? Wir betrachten ein Beispiel. Sei $x = 2$, $y = 25$, $m = 7$. Es entstehen rekursive Aufrufe mit $y = 25, 12, 6, 3, 1$, wobei das erste

[6] „All primes are odd, except 2, which is the oddest of all." (D. E. Knuth) – Ein Wortspiel damit, dass „odd" sowohl „ungerade" als auch „merkwürdig" bedeuten kann. Das zahlentheoretische Verhalten der Potenzen von 2 ist ganz anders als das der Potenzen einer Primzahl $p > 2$.

[7] 74590552468911142187725098562763415853 5.

[8] Den Fall $y = 1$ könnte man im Programmtext einsparen.

Argument die Werte $x = 2$, $x^2 = 2^2 = 4$, $x^4 = 4^2 \bmod 7 = 2$, $x^8 = 2^2 = 4$, $x^{16} = 4^2 \bmod 7 = 2$ annimmt. Beim Zurückgehen in der Rekursion werden die Argumente aufmultipliziert, bei denen y ungerade ist, dies sind 2, ~~4~~, ~~2~~, 4, 2, mit Produkt 16 mod $7 = 2$. Diese Auswahl entspricht gerade der Binärdarstellung 11001 von 25, von rechts nach links gelesen. Allgemein berechnet die Rekursion iterativ die Potenzen $x^{2^i} \bmod m$, für $i = 0, 1, \ldots, \lfloor \log y \rfloor$, und die Prüfung, ob y gerade oder ungerade ist, wählt die Faktoren $x^{2^i} \bmod m$ aus, bei denen Bit b_i in der Binärdarstellung $b_k \ldots b_1 b_0$ von y Wert 1 hat.

Man erkennt sofort, dass bei jedem rekursiven Aufruf die Bitanzahl des Exponenten y um 1 sinkt, dass also die Anzahl der Rekursionsebenen etwa $\log y$ beträgt. In jeder Rekursionsebene ist eine oder sind zwei Multiplikationen modulo m auszuführen, was $O((\log m)^2)$ Ziffernoperationen erfordert (Schulmethode für Multiplikation und Division). Damit kommt man bei der schnellen Exponentiation selbst bei der einfachsten Implementierung der Multiplikation mit $O((\log y)(\log m)^2)$ Ziffernoperationen zum Ergebnis.

Lemma 5.1.20
Die Berechnung von $x^y \bmod m$ benötigt $O(\log y)$ Multiplikationen und Divisionen modulo m von Zahlen aus $\{0, \ldots, m^2 - 1\}$ sowie $O((\log y)(\log m)^2)$ Bitoperationen.

Bemerkung: Man kann denselben Algorithmus in einem beliebigen *Monoid* (Bereiche mit einer assoziativen Operation und einem neutralen Element) benutzen. Einen Monoid bilden zum Beispiel die natürlichen Zahlen mit der Multiplikation oder die Menge Σ^* über einem Alphabet Σ mit der Konkatenation. In Abschnitt 5.4 benutzen wir die schnelle Exponentiation für einen etwas ungewöhnlichen Monoid. Wenn man Zeile 1 weglässt und $y \geq 1$ fordert, genügt auch eine assoziative Operation ohne neutrales Element, um das Verfahren anwendbar zu machen.

5.1.3 Der Chinesische Restsatz und die Eulersche φ-Funktion

Der „Chinesische Restsatz“ besagt im Wesentlichen, dass für teilerfremde Zahlen m und n die Strukturen $\mathbb{Z}_m \times \mathbb{Z}_n$ (mit komponentenweisen Operationen) und $\mathbb{Z}_{mn}$ isomorph sind.

Wir beginnen mit einem Beispiel, nämlich $m = 3$, $n = 8$, also $mn = 24$. Die folgende Tabelle gibt die Reste der Zahlen $x \in \{0, 1, \ldots, 23\}$ modulo 3 und modulo 8 an. Die Restepaare wiederholen sich zyklisch für die anderen Elemente $x \in \mathbb{Z}$.

x	0	1	2	3	4	5	6	7	8	9	10	11
$x \bmod 3$	0	1	2	0	1	2	0	1	2	0	1	2
$x \bmod 8$	0	1	2	3	4	5	6	7	0	1	2	3

x	12	13	14	15	16	17	18	19	20	21	22	23
$x \bmod 3$	0	1	2	0	1	2	0	1	2	0	1	2
$x \bmod 8$	4	5	6	7	0	1	2	3	4	5	6	7

Wenn wir die Einträge in Zeilen 2 und 3 als 24 Paare in $\mathbb{Z}_3 \times \mathbb{Z}_8$ ansehen, erkennen wir, dass sie alle verschieden sind, also auch alle Möglichkeiten in $\{0,1,2\} \times \{0,1,\ldots,7\}$ abdecken. D. h.: Die Abbildung $x \mapsto (x \bmod 3, x \bmod 8)$ ist eine Bijektion zwischen $\mathbb{Z}_{24}$ und $\mathbb{Z}_3 \times \mathbb{Z}_8$. Zudem spiegeln sich arithmetische Operationen auf den Elementen von $\mathbb{Z}_{24}$ in den Resten modulo 3 und 8 wider. Beispielsweise liefert die komponentenweise Addition von $(2,7)$ und $(2,1)$ das Resultat $(1,0)$, was der Addition von 23 und 17 (modulo 24) mit dem Resultat $40 \bmod 24 = 16$ entspricht. Genauso ist $(2^5 \bmod 3, 3^5 \bmod 8) = (2,3)$, was der Gleichung $11^5 \bmod 24 = 11$ entspricht.

Der Chinesische Restsatz besagt im Wesentlichen, dass eine solche strukturelle Entsprechung zwischen den Resten modulo mn und Paaren von Resten modulo m bzw. n immer gilt, wenn m und n teilerfremd sind.

Fakt 5.1.21 ***Chinesischer Restsatz***
m und n seien teilerfremd. Dann ist die Abbildung[a]

$$\Phi\colon \mathbb{Z}_{mn} \ni x \mapsto (x \bmod m, x \bmod n) \in \mathbb{Z}_m \times \mathbb{Z}_n$$

bijektiv. Zudem: Wenn $\Phi(x) = (x_1, x_2)$ und $\Phi(y) = (y_1, y_2)$, dann gilt:

(a) $\Phi(x +_{mn} y) = (x_1 +_m y_1, x_2 +_n y_2)$;

(b) $\Phi(x \cdot_{mn} y) = (x_1 \cdot_m y_1, x_2 \cdot_n y_2)$;

(c) $\Phi(0) = (0,0)$, $\Phi(1) = (1,1)$, $\Phi(nm-1) = (n-1, m-1)$.

[a] Φ ist der griechische Großbuchstabe *Phi*, gesprochen „(groß-)Fi“.

Für mathematisch-strukturell orientierte Leser/innen: Die Gleichungen (a) und (b)

kann man etwas abstrakter auch so fassen: Die Abbildung Φ ist ein Ring-mit-1-Isomorphismus zwischen $\mathbb{Z}_{mn}$ und $\mathbb{Z}_m \times \mathbb{Z}_n$.

Wir bemerken noch Folgendes, obgleich es für die in dieser Vorlesung betrachteten Situationen nicht direkt wichtig ist. Die Funktion Φ lässt sich selbstverständlich durch zwei Divisionen sehr effizient berechnen. Wie sieht es aber mit der Umkehrung aus? Kann man aus $y \in \mathbb{Z}_m$ und $z \in \mathbb{Z}_n$ *effizient* ein $x \in \mathbb{Z}_{mn}$ mit $\Phi(x) = (y, z)$ berechnen? Dies ist tatsächlich der Fall. Man berechnet mit dem erweiterten Euklidischen Algorithmus das multiplikative Inverse $u = n^{-1} \bmod m$ und das multiplikative Inverse $v = m^{-1} \bmod n$. (Dies ist möglich, weil $\mathrm{ggT}(n, m) = 1$, also $n \in \mathbb{Z}_m^*$ und $m \in \mathbb{Z}_n^*$.) Dann setzt man $x := (yun + zvm) \bmod mn$. Dann ist $x \bmod m = yun \bmod m = y$ (weil $un \bmod m = 1$ gilt) und analog $x \bmod n = zvm \bmod n = z$. Also gilt $\Phi(x) = (y, z)$.

Wir halten noch fest, wie sich Zahlen, die zu m und n teilerfremd sind, in der Sichtweise des Chinesischen Restsatzes verhalten.

Proposition 5.1.22
Wenn man die Abbildung Φ aus dem Chinesischen Restsatz auf $\mathbb{Z}_{mn}^*$ einschränkt, ergibt sich eine Bijektion zwischen $\mathbb{Z}_{mn}^*$ und $\mathbb{Z}_m^* \times \mathbb{Z}_n^*$.

In der Beispieltabelle für $m = 3$, $n = 8$, $mn = 24$ sieht das so aus, für $\mathbb{Z}_3^* = \{1, 2\}$, $\mathbb{Z}_8^* = \{1, 3, 5, 7\}$, $\mathbb{Z}_{24}^* = \{1, 5, 7, 11, 13, 17, 19, 23\}$:

$x \in \mathbb{Z}_{24}^*$	1	5	7	11	13	17	19	23
$x \bmod 3$	1	2	1	2	1	2	1	2
$x \bmod 8$	1	3	5	7	1	3	5	7

Der Chinesische Restsatz und die nachfolgenden Bemerkungen und Behauptungen lassen sich leicht auf $r > 2$ paarweise teilerfremde Faktoren $n_1, \ldots, n_r$ verallgemeinern. Die Aussagen lassen sich durch vollständige Induktion über r beweisen.

Prop. 5.1.22 liefert auch eine Formel für die Kardinalitäten der Mengen $\mathbb{Z}_m^*$ für beliebige $m \geq 2$.

Definition 5.1.23 ***Eulersche φ-Funktion***
Für $m \geq 2$ sei[a]

$$\varphi(m) := |\mathbb{Z}_m^*| = |\{x \mid 0 \leq x < m, \mathrm{ggT}(x, m) = 1\}|.$$

[a] φ ist der griechische Kleinbuchstabe *phi*, gesprochen „(klein-)fi“.

Einige Beispielwerte sind in Tab. 2 angegeben.

m	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\varphi(m)$	1	2	2	4	2	6	4	6	4	10	4	12	6	8	8

Tabelle 2: Eulersche φ-Funktion für kleine m

Folgendes ist eine unmittelbare Konsequenz aus Proposition 5.1.22:

Lemma 5.1.24
Für teilerfremde Zahlen n und m gilt $\varphi(mn) = \varphi(m) \cdot \varphi(n)$.

Uns interessiert (später) besonders der Fall $\varphi(p) = p - 1$ für Primzahlen p und $\varphi(pq) = \varphi(p) \cdot \varphi(q) = (p-1)(q-1)$ für verschiedene Primzahlen p und q. Man kann aber auch eine allgemeine Formel für $\varphi(n)$ gewinnen.

Lemma 5.1.25
Für $m \geq 2$ gilt

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

Beweis: Wenn m eine Primzahlpotenz p^t ist, dann besteht $\mathbb{Z}_m^*$ aus den Zahlen in $\mathbb{Z}_m = \{0, 1, \ldots, p^t - 1\}$, die nicht durch p teilbar sind. Da es in $\mathbb{Z}_m$ insgesamt p^t Zahlen gibt und p^{t-1} Vielfache von p, gilt $\varphi(m) = p^t - p^{t-1} = m - m/p = m(1 - 1/p)$. Nun nehmen wir an, dass

$$m = p_1^{t_1} \cdots p_s^{t_s}$$

gilt, für verschiedene Primzahlen $p_1, \ldots, p_s$ und $t_1, \ldots, t_s \geq 1$. Die Faktoren $p_1^{t_1}, \ldots, p_s^{t_s}$ sind teilerfremd. Mit Lemma 5.1.24, $(s-1)$-mal angewendet, erhalten wir

$$\varphi(m) = \prod_{i=1}^{s} \varphi(p_i^{t_i}) = \prod_{i=1}^{s} p_i^{t_i}(1 - 1/p_i) = m \cdot \prod_{i=1}^{s} \left(1 - \frac{1}{p_i}\right). \qquad \square$$

Mit dieser Formel lassen sich die Werte in Tabelle 2 schnell verifizieren. (*Beispiel*: $\varphi(12) = 12(1-1/2)(1-1/3) = 12 \cdot (1/2) \cdot (2/3) = 4$.)

Bemerkung: Die simple Formel in Lemma 5.1.25 könnte zu dem Schluss verleiten, dass sich $\varphi(m)$ zu gegebenem m immer leicht berechnen lässt. Aber Achtung: Man muss dazu die Menge der Primfaktoren von m kennen. Im allgemeinen Fall läuft das darauf hinaus, dass man das Faktorisierungsproblem für m lösen muss, und hierfür kennt man keine effizienten Algorithmen (also Algorithmen mit Rechenzeit $O((\log m)^c)$ für eine Konstante c). Tatsächlich ist kein effizienter Algorithmus bekannt, der es erlaubt, $\varphi(m)$ aus m zu berechnen.

5.1.4 Primzahlen

Wir erinnern an Definition 5.1.16, die Definition der Konzepte „Primzahl“ und „zusammengesetzte Zahl“.

Fakt 5.1.26
Wenn p eine Primzahl ist und $p \mid xy$ gilt, dann gilt $p \mid x$ oder $p \mid y$.

Beweis: Wenn $p \mid x$, sind wir fertig. Also können wir $p \nmid x$ annehmen, das heißt $\mathrm{ggT}(p, x) = 1$. Nach dem Lemma von Bezout (Lemma 5.1.9) können wir $1 = sp + tx$ schreiben, für ganze Zahlen s, t. Daraus folgt: $y = spy + txy$. Weil xy durch p teilbar ist, folgt $p \mid y$. □

Satz 5.1.27 ***Fundamentalsatz der Arithmetik***
Jede Zahl $N \geq 1$ kann als Produkt von Primzahlen geschrieben werden. Die Faktoren sind dabei bis auf die Reihenfolge eindeutig bestimmt.

Fakt 5.1.28
Jede zusammengesetzte Zahl $N \geq 2$ besitzt einen Primfaktor p mit $p \leq \sqrt{N}$.

Bemerkung: Wir betrachten das resultierende naive Faktorisierungsverfahren: Teste die Zahlen in $\{2, \ldots, \lfloor\sqrt{N}\rfloor\}$ nacheinander darauf, ob sie N teilen; wenn ein Faktor p gefunden wurde (dieses p muss eine Primzahl sein), suche mit demselben Verfahren (startend bei p) Teiler von $N' = N/p$. Dieses Verfahren hat im schlechtesten Fall Rechenzeit mindestens $\Theta(\sqrt{N}) = \Theta(2^{(\log N)/2})$, also exponentiell in der Bitlänge von

N. Wie wir später genauer diskutieren werden, sind für das Auffinden der Primzahlzerlegung einer gegebenen Zahl N überhaupt keine effizienten Algorithmen bekannt (also Algorithmen mit Laufzeiten $O((\log N)^c)$ für konstantes c). Aber es gibt effiziente Algorithmen, mit denen man feststellen kann, ob eine Zahl N eine Primzahl ist oder nicht. Dieser Unterschied in der Schwierigkeit des Faktorisierungsproblems und des Primzahlproblems liegt einer ganzen Reihe von kryptographischen Verfahren zugrunde.

Satz 5.1.29 ***Satz von Euklid***
Es gibt unendlich viele Primzahlen.

Über die Verteilung der Primzahlen (ihre „Dichte“) in $\mathbb{N}$ gibt der berühmte Primzahlsatz[9] Auskunft. Mit $\pi(x)$ bezeichnen wir die Anzahl der Primzahlen, die nicht größer als x sind.

Satz 5.1.30 ***Primzahlsatz***

$$\lim_{x\to\infty} \frac{\pi(x)}{x/\ln x} = 1.$$

Das heißt, dass für große x in $(x, 2x]$ etwa $\frac{2x}{\ln(2x)} - \frac{x}{\ln x} \approx \frac{x}{\ln x}$ Primzahlen zu erwarten sind. Die μ-ziffrigen Zahlen bilden das Intervall $[2^{\mu-1}, 2^{\mu})$. Der Anteil der Primzahlen in diesem Intervall ist näherungsweise

$$\frac{2^{\mu-1}/\ln(2^{\mu-1})}{2^{\mu-1}} \approx \frac{1}{(\ln 2)(\mu-1)} \approx 1{,}44/\mu.$$

Für $\mu \approx 2000$ ist der relative Anteil von Primzahlen im interessanten Zahlenbereich also etwa $\approx 1{,}44/2000 \approx 1/1400$. Er sinkt umgekehrt proportional zur Ziffernzahl.

Eine schärfere Form des Primzahlsatzes ist folgende Aussage[10] (wobei der Beweis Nichtspezialisten nicht zugänglich ist):

$$\frac{x}{\ln x}\left(1+\frac{1}{\ln x}\right) \overset{(x\geq 599)}{\leq} \pi(x) \overset{(x\geq 2)}{\leq} \frac{x}{\ln x}\left(1+\frac{1.2762}{\ln x}\right).$$

[9] 1793 von Gauß und unabhängig 1798 von Legendre vermutet; 1896 unabhängig von Hadamard und de La Vallée Poussin bewiesen.

[10] Corollary 5.2 in: Dusart, Pierre. Explicit estimates of some functions over primes. Ramanujan J. 45, 227–251 (2018). https://doi.org/10.1007/s11139-016-9839-4.

Für $x \geq 500\,000$ folgt daraus: $\frac{x}{\ln x} < \pi(x) < 1{,}1\frac{x}{\ln x}$.

Daraus folgt, dass für $n \geq 20$ der Anteil der Primzahlen unter den n-Bit-Zahlen folgende Ungleichung erfüllt:

$$\begin{aligned}\frac{|\{p \in [2^{n-1}, 2^n) \mid p \text{ is prime}\}|}{2^{n-1}} &= \frac{\pi(2^n) - \pi(2^{n-1})}{2^{n-1}} \\ &\geq \frac{2^n/(n\ln 2) - 1{,}1 \cdot 2^{n-1}/((n-1)\ln 2)}{2^{n-1}} \\ &\geq \frac{2}{n\ln 2} - \frac{1{,}1}{n\ln 2} \cdot \frac{n}{n-1} \\ &\geq \frac{6}{5n}.\end{aligned}$$

Mit Hilfe eines Computeralgebraprogramms findet man heraus, dass die Ungleichung $|\{p \in [2^{n-1}, 2^n) \mid p \text{ is prime}\}|/2^{n-1} \geq 6/(5n)$ auch für $9 \leq n \leq 20$ gilt. (Für $n = 8$ ist sie falsch.) Man kann sich also merken:

Für $n \geq 9$ ist der Anteil der Primzahlen an den n-Bit-Zahlen mindestens $\dfrac{6}{5n}$.

Ziffernzahl n	Dusart-Schranke für $(\pi(2^n) - \pi(2^{n-1}))/2^{n-1}$	numerische untere Schranke
256	$\frac{6}{5\cdot 256}$	$\geq \frac{1}{214}$
512	$\frac{6}{5\cdot 512}$	$\geq \frac{1}{427}$
1024	$\frac{6}{5\cdot 1024}$	$\geq \frac{1}{854}$
2048	$\frac{6}{5\cdot 2048}$	$\geq \frac{1}{1707}$

Eine leichter zu beweisende Aussage der Art $\pi(2m) - \pi(m) = \Theta(m/\log m)$ ist die folgende:

Satz 5.1.31 ***Ungleichung von Finsler***
Für jede ganze Zahl $m \geq 2$ liegen im Intervall $(m, 2m]$ mindestens $m/(3\ln(2m))$ Primzahlen*:*

$$\pi(2m) - \pi(m) \geq \frac{m}{3\ln(2m)}.$$

Ein vollständiger, vergleichsweise einfacher *Beweis* für Satz 5.1.31 findet sich zum Beispiel in dem Lehrbuch „Elemente der Diskreten Mathematik: Zahlen und Zählen, Graphen und Verbände“ von Diekert, Kufleitner, Rosenberger (De Gruyter 2013).

5.2 Der Miller-Rabin-Primzahltest

In diesem Abschnitt lernen wir einen randomisierten Algorithmus kennen, der es erlaubt, zu einer gegebenen Zahl N zu entscheiden, ob N eine Primzahl ist oder nicht.

Ein idealer Primzahltest sieht so aus:

Eingabe: Eine natürliche Zahl $N \geq 3$.

Ausgabe: 0, falls N eine Primzahl ist; 1, falls N zusammengesetzt ist.

Wozu braucht man Primzahltests? Zunächst ist die Frage „Ist N eine Primzahl?“ eine grundlegende mathematisch interessante Fragestellung. Spätestens mit dem Siegeszug des RSA-Kryptosystems ab den späten 1970er Jahren hat sich die Situation jedoch dahin entwickelt, dass man Algorithmen benötigt, die immer wieder neue vielziffrige Primzahlen (etwa mit 1000 oder 1500 Bits[11] bzw. 301 oder 452 Dezimalziffern) bereitstellen können. Den Kern dieser Primzahlerzeugungs-Verfahren (siehe Abschnitt 5.3) bildet ein Verfahren, das eine gegebene Zahl N darauf testet, ob sie prim ist.

Der naive Primzahltest („trial division“), der dem *brute-force*-Paradigma folgt und oben schon beschrieben wurde, findet durch direkte Division der Zahl N durch $2, 3, 4, \ldots, \lfloor\sqrt{N}\rfloor$ heraus, ob N einen nichttrivialen Teiler hat. Man kann dieses Verfahren durch einige Tricks beschleunigen, aber die Rechenzeit wächst dennoch mit $\Theta(\sqrt{N})$. Dies macht es für Zahlen N mit mehr als 40 Dezimalstellen praktisch undurchführbar, von Zahlen mit mehr als 100 Dezimalstellen ganz zu schweigen. (Achtung: Damit wird nichts über die Qualität anderer Faktorisierungsalgorithmen gesagt. Es gibt an-

[11] `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?__blob=publicationFile&v=12` Stand 24.03.2020. BSI TR-02102-1 Kryptographische Verfahren: Empfehlungen und Schlüssellängen, S. 28: „Für einen Einsatzzeitraum über das Jahr 2022 hinaus wird empfohlen, RSA/DLIES-Schlüssel von 3000 Bits Länge zuverwenden, um ein gleichmäßiges Sicherheitsniveau in allen empfohlenen asymmetrischen Verschlüsselungsverfahren zu erzielen. Die Schlüssellänge von 2000 Bit bleibt für DLIES-Schlüssel bis 2022 zur vorliegenden Richtlinie konform, außerdem übergangsweise für RSA-Schlüssel bis Ende 2023.“ Dabei bedeutet *Schlüssellänge* die Bitanzahl eines Produkts $n = p \cdot q$ für Primzahlen p und q mit jeweils der halben Länge.

dere, sehr fortgeschrittene Faktorisierungsalgorithmen, die bei entsprechendem Zeitaufwand und mit sehr leistungsstarken Rechnern auch noch mit 200-stelligen Zahlen zurechtkommen. Für Information zu früheren und aktuelleren Faktorisierungserfolgen siehe z. B. `https://en.wikipedia.org/wiki/RSA_numbers`.)

In diesem Abschnitt beschreiben wir den randomisierten Primzahltest von Miller und Rabin. Dabei handelt es sich um einen Monte-Carlo-Algorithmus mit einseitigem Fehler. Das heißt: Auf Eingaben N, die Primzahlen sind, wird immer 0 ausgegeben; auf Eingaben N, die zusammengesetzt sind, gibt es eine gewisse (von N abhängige) Wahrscheinlichkeit, dass die Ausgabe 0, also falsch ist. Für kein zusammengesetztes N ist diese Wahrscheinlichkeit größer als die „Fehlerschranke“ $\frac{1}{4}$. Wir beweisen nur die Fehlerschranke $\frac{1}{2}$. Im Beweis benutzen wir einfache zahlentheoretische Überlegungen. Eine herausragende Eigenschaft des Miller-Rabin-Tests ist seine Effizienz. Wir werden sehen, dass selbst bei Verwendung der Schulmethoden für Multiplikation und Division die Bitkomplexität des Primzahltests nur $O((\log N)^3)$ ist, also $O(n^3)$, wenn $n = \log N$ die Bitlänge der Eingabezahl ist.

Bemerkung: Der Miller-Rabin-Algorithmus stammt aus dem Jahr 1977; er folgte einem kurz vorher vorgestellten anderen randomisierten Primzahltest (Solovay-Strassen-Test). Für diesen und andere randomisierte Primzahltests (z. B. den „Strong Lucas Probable Prime Test“ oder den „Quadratic Frobenius Test“ von Grantham) sei auf die Literatur verwiesen. Im Jahr 2002 stellten Agrawal, Kayal und Saxena einen deterministischen Primzahltest mit polynomieller Rechenzeit vor. Die Rechenzeit ist z. B. durch $O((\log N)^{7,5})$ beschränkt. Dieser Algorithmus stellte insofern einen gewaltigen Durchbruch dar, als er ein Jahrhunderte altes offenes Problem löste, nämlich die Frage nach einem effizienten[12] *deterministischen* Verfahren für das Entscheidungsproblem „ist N Primzahl oder zusammengesetzte Zahl“? Andererseits ist seine Laufzeit im Vergleich etwa zu dem hier diskutierten randomisierten Verfahren so hoch, dass nach wie vor die randomisierten Algorithmen benutzt werden, um für kryptographische Anwendungen Primzahlen zu erzeugen.

Da gerade Zahlen leicht zu erkennen sind, beschränken wir im Folgenden unsere Überlegungen auf ungerade Zahlen $N \geq 3$.

5.2.1 Der Fermat-Test

Wir erinnern uns an den kleinen Satz von Fermat (Fakt 5.1.18):

[12] d. h. mit Rechenzeit polynomiell in der Bitlänge $\log N$.

$$p \text{ ist Primzahl und } 1 \leq a < p \quad \Rightarrow \quad a^{p-1} \bmod p = 1.$$

Wir können diese Aussage dazu benutzen, um „Belege“ oder „Zertifikate“ oder „Zeugen“ dafür anzugeben, dass eine Zahl N zusammengesetzt ist: Wenn wir eine Zahl a mit $1 \leq a < N$ finden, für die $a^{N-1} \bmod N \neq 1$ gilt, dann ist N definitiv keine Primzahl.

Definition 5.2.1
Sei $N \geq 3$ ungerade und zusammengesetzt.
Eine Zahl $a \in \{1, \ldots, N-1\}$ heißt **F-Zeuge** für N, wenn $a^{N-1} \bmod N \neq 1$ gilt.
Eine Zahl $a \in \{1, \ldots, N-1\}$ heißt **F-Lügner** für N, wenn $a^{N-1} \bmod N = 1$ gilt.
Die Menge der F-Lügner nennen wir L_N^{F}.

Wir bemerken, dass ein F-Zeuge zwar belegt, dass es Faktoren $k, \ell > 1$ mit $N = k \cdot \ell$ *gibt*, dass aber ein F-Zeuge nicht auf solche Faktoren hinweist oder sie beinhaltet. Das *Finden* von Faktoren wird von Primzahltests auch nicht verlangt und normalerweise auch nicht geleistet.

Man sieht sofort, dass 1 und $N-1$ immer F-Lügner sind: Es gilt $1^{N-1} \bmod N = 1$ und $(N-1)^{N-1} \equiv (-1)^{N-1} \equiv 1 \pmod{N}$, weil $N-1$ gerade ist.

Für jede zusammengesetzte Zahl N gibt es mindestens einen F-Zeugen. Wir erinnern uns, dass nach Proposition 5.1.17 die Zahl N genau dann zusammengesetzt ist, wenn $\{1, \ldots, N-1\} - \mathbb{Z}_N^* \neq \emptyset$ gilt.

Lemma 5.2.2
Wenn N keine Primzahl ist, ist jedes $a \in \{1, \ldots, N-1\} - \mathbb{Z}_N^*$ ein F-Zeuge.

Beweis: Sei $d = \mathrm{ggT}(a, N) > 1$. Dann ist auch a^{N-1} durch d teilbar, also auch $a^{N-1} \bmod N = a^{N-1} - \lfloor a^{N-1}/N \rfloor \cdot N$. Daher ist $a^{N-1} \bmod N \neq 1$. □

Leider ist für manche zusammengesetzten Zahlen N die Menge $\{1, \ldots, N-1\} - \mathbb{Z}_N^*$ äußerst dünn. Wenn zum Beispiel $N = pq$ für zwei Primzahlen p und q ist, dann gilt $\mathrm{ggT}(a, N) > 1$ genau dann wenn p oder q ein Teiler von a ist. Es gibt genau $p + q - 2$ solche Zahlen a in $\{1, \ldots, N-1\}$, was gegenüber N sehr klein ist, wenn p und q annähernd gleich groß sind. Um eine gute Chance zu haben, F-Zeugen zu finden, sollte es also mehr als nur die in $\{1, \ldots, N-1\} - \mathbb{Z}_N^*$ geben.

Beispiel: $N = 91 = 7 \cdot 13$. Tabelle 3 zeigt, dass es 18 Vielfache von 7 und 13 gibt (für größere p und q wird der Anteil dieser offensichtlichen F-Zeugen noch kleiner

F-Zeugen in $\{1,\ldots,90\} - \mathbb{Z}_{91}^*$:	
7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84; 13, 26, 39, 52, 65, 78	
F-Lügner:	F-Zeugen in $\mathbb{Z}_{91}^*$:
1, 3, 4, 9, 10, 12, 16, 17, 22,	2, 5, 6, 8, 11, 15, 18, 19, 20,
23, 25, 27, 29, 30, 36, 38, 40, 43,	24, 31, 32, 33, 34, 37, 41, 44, 45,
48, 51, 53, 55, 61, 62, 64, 66, 68,	46, 47, 50, 54, 57, 58, 59, 60, 67,
69, 74, 75, 79, 81, 82, 87, 88, 90	71, 72, 73, 76, 80, 83, 85, 86, 89

Tabelle 3: F-Zeugen und F-Lügner für $N = 91 = 7 \cdot 13$. Es gibt 36 F-Lügner und 36 F-Zeugen in $\mathbb{Z}_{91}^*$. Wir wissen nach Lemma 5.2.2, dass alle 18 Vielfachen von 7 und 13 F-Zeugen sind.

sein), und daneben weitere 36 F-Zeugen und 36 F-Lügner in $\{1, 2, \ldots, 90\}$. In diesem Beispiel gibt es um einiges mehr F-Zeugen als F-Lügner. Wenn dies für alle zusammengesetzten Zahlen N der Fall wäre, wäre es eine elegante randomisierte Strategie, einfach zufällig nach F-Zeugen zu suchen.

Dies führt zu unserem ersten Versuch für einen randomisierten Primzahltest.

Algorithmus 5.2.3 ***Fermat-Test***
EINGABE: Ungerade Zahl $N \geq 3$.
METHODE:
1 Wähle a zufällig aus $\{1, \ldots, N-1\}$;
2 **if** $a^{N-1} \bmod N \neq 1$ **then return** 1 **else return** 0.

Die Laufzeitanalyse liegt auf der Hand: Der teuerste Teil ist die Berechnung der Potenz $a^{N-1} \bmod N$ durch schnelle Exponentiation, die nach den Ergebnissen von Lemma 5.1.20 $O(\log N)$ arithmetische Operationen und $O((\log N)^3)$ Ziffernoperationen benötigt. Weiter ist es klar, dass der Algorithmus einen F-Zeugen gefunden hat, wenn er „1“ ausgibt, dass in diesem Fall also N zusammengesetzt sein muss. Umgekehrt ausgedrückt: Wenn N eine Primzahl ist, gibt der Fermat-Test garantiert „0“ aus.

Für $N = 91$ wird das falsche Ergebnis 0 ausgegeben, wenn als a einer der 36 F-Lügner

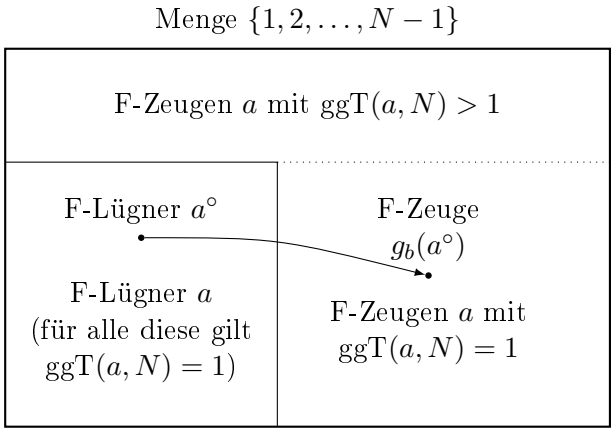


Abbildung 5.2.2: Möglichkeiten für a im Fermat-Test. N ist zusammengesetzt; L_N^{F} ist die Menge der F-Lügner; es gibt einen F-Zeugen $b \in \mathbb{Z}_N^*$. Die Abbildung g_b bildet L_N^{F} injektiv in die Menge der F-Zeugen in $\mathbb{Z}_N^*$ ab. Also gibt es mehr F-Zeugen als F-Lügner.

gewählt wird. Die Wahrscheinlichkeit hierfür ist $\frac{36}{90} = \frac{2}{5} = 0{,}4$.

Für viele zusammengesetzte Zahlen N gibt es reichlich F-Zeugen, so dass der Fermat-Test für diese N mit konstanter Wahrscheinlichkeit das korrekte Ergebnis liefert. Wir analysieren das Verhalten des Fermat-Tests für solche „gutmütigen“ Eingabezahlen N (für die $N = 91$ ein typisches Beispiel ist).

Satz 5.2.4
Sei $N \geq 3$ eine ungerade zusammengesetzte Zahl. (Dann ist $N \geq 9$.) Wenn es mindestens einen F-Zeugen $b \in \mathbb{Z}_N^*$ gibt, dann liefert der Fermat-Test auf Eingabe N mit Wahrscheinlichkeit größer als $\frac{1}{2}$ die korrekte Antwort „1“.

Beweis: Sei $b \in \mathbb{Z}_N^*$ ein F-Zeuge. Betrachte die Funktion $g_b \colon L_N^{\mathrm{F}} \to \mathbb{Z}_N^*$, die den F-Lügner a auf $g_b(a) = ba \bmod N$ abbildet. Wie im Beweis von Fakt 5.1.18 sieht man, dass g_b injektiv ist. Weiter ist $g_b(a)$ für jedes $a \in L_N^{\mathrm{F}}$ ein F-Zeuge:

$$(ba \bmod N)^{N-1} \bmod N = (b^{N-1} \bmod N)(\underbrace{(a^{N-1}) \bmod N}_{=1}) = b^{N-1} \bmod N \neq 1.$$

Wir können also jedem F-Lügner a einen „Zwilling“ $g_b(a)$ in der Menge der F-Zeugen zuordnen. Daraus folgt, dass es in $\mathbb{Z}_N^*$ mindestens so viele F-Zeugen wie F-Lügner

gibt. Mit Lemma 5.2.2 ergibt sich, dass $\{1, \ldots, N-1\}$ mehr F-Zeugen als F-Lügner enthält. Daher ist die Wahrscheinlichkeit, dass die im Fermat-Test gewählte Zahl a ein F-Lügner ist, kleiner als $\frac{1}{2}$. □

Die Fehlerwahrscheinlichkeit $\frac{1}{2}$ ist natürlich viel zu groß. Wir verringern sie durch wiederholte Ausführung des Fermat-Tests.

Algorithmus 5.2.5 ***Iterierter Fermat-Test***
EINGABE: Ungerade Zahl $N \geq 3$, eine Zahl $\ell \geq 1$.
METHODE:

```
1    repeat ℓ times
2        a ← ein zufälliges Element von {1, ..., N − 1};
3        if a^{N−1} mod N ≠ 1 then return 1;
4    return 0.
```

Wenn die Ausgabe 1 ist, hat der Algorithmus einen F-Zeugen für N gefunden, also ist N zusammengesetzt. D. h.: Wenn N eine Primzahl ist, ist die Ausgabe 0. Andererseits: Wenn N zusammengesetzt ist, und es mindestens einen F-Zeugen $a \in \mathbb{Z}_N^*$ gibt, dann ist nach Satz 5.2.4 die Wahrscheinlichkeit für die falsche Ausgabe „0“ höchstens $\left(\frac{1}{2}\right)^\ell = 2^{-\ell}$. Indem wir ℓ genügend groß wählen, können wir die Fehlerwahrscheinlichkeit so klein wie gewünscht einstellen.

Wenn es darum geht, aus einem genügend großen Bereich zufällig gewählte Zahlen darauf zu testen, ob es sich um eine Primzahl handelt, dann ist der Fermat-Test eine sehr effiziente und zuverlässige Methode. Wir kommen im folgenden Abschnitt über Primzahlerzeugung nochmals darauf zurück.

Wenn man allerdings über die Herkunft der zu testenden Zahl N keine Information hat und eventuell damit rechnen muss, dass jemand (ein „Gegenspieler“) absichtlich eine besonders schwierige Eingabe vorlegt, dann stößt der Fermat-Test an eine Grenze. Es gibt nämlich „widerspenstige“ zusammengesetzte Zahlen, denen man mit diesem Test nicht beikommen kann, weil alle Elemente von $\mathbb{Z}_N^*$ F-Lügner sind. Mit diesen befasst sich der folgende Abschnitt.

5.2.2 Carmichael-Zahlen

Definition 5.2.6
Eine ungerade zusammengesetzte Zahl N heißt eine **Carmichael-Zahl**, wenn für alle $a \in \mathbb{Z}_N^*$ die Gleichung $a^{N-1} \bmod N = 1$ gilt.

Die kleinste Carmichael-Zahl ist $561 = 3 \cdot 11 \cdot 17$. Weitere kleine Carmichael-Zahlen sind $1105 = 5 \cdot 13 \cdot 17$ und $1729 = 7 \cdot 13 \cdot 19$. Erst im Jahr 1994 wurde bewiesen, dass es unendlich viele Carmichael-Zahlen gibt, genauer: Wenn x genügend groß ist, dann gibt es in $\{N \mid N \leq x\}$ mehr als $x^{2/7}$ Carmichael-Zahlen. Die aktuell beste bekannte untere Schranke für $|\{N \leq x \mid N \text{ Carmichael-Zahl}\}|$ ist $x^{1/3}$. Von Erdős (1956) stammt die obere Schranke $x \cdot \exp(\frac{-c \ln x \ln\ln\ln x}{\ln\ln x})$ für die Größe dieser Menge, für eine Konstante $c > 0$, die zeigt, dass Carmichael-Zahlen *viel* seltener als Primzahlen sind. (Mehr Details: `https://en.wikipedia.org/wiki/Carmichael_number`.)

Wenn wir dem Fermat-Test eine Carmichael-Zahl N als Eingabe geben, ist die Wahrscheinlichkeit für die falsche Antwort 0 nach Lemma 5.1.25 genau

$$\frac{\varphi(N)}{N-1} > \frac{\varphi(N)}{N} = \prod_{\substack{p \text{ prim} \\ p \text{ teilt } N}} \left(1 - \frac{1}{p}\right) > 1 - \sum_{\substack{p \text{ prim} \\ p \text{ teilt } N}} \frac{1}{p}.$$

Diese Wahrscheinlichkeit liegt nahe an 1, wenn N nur wenige und relativ große Primfaktoren hat. An solchen Carmichael-Zahlen besteht etwa im Bereich der Zahlen im Bereich $[10^{15}, 10^{16}]$ kein Mangel, wie ein Blick in entsprechende Tabellen zeigt. Zum Beispiel ist $N = 651693055693681 = 72931 \cdot 87517 \cdot 102103$ eine Carmichael-Zahl mit $\varphi(N)/N > 0.99996$.

Der Wiederholungstrick zur Wahrscheinlichkeitsverbesserung hilft hier leider auch nicht, denn wenn etwa p_0 der kleinste Primfaktor von N ist, und N nur 3 oder 4 Faktoren hat, dann sind $\Omega(p_0)$ Wiederholungen nötig, um die Fehlerwahrscheinlichkeit auf $\frac{1}{2}$ zu drücken. Sobald p_0 mehr als 30 Dezimalstellen hat, ist dies undurchführbar.

Für einen zuverlässigen, effizienten Primzahltest, der für *alle* Eingabezahlen N funktioniert, müssen wir über den Fermat-Test hinausgehen. Interessanterweise ist dies praktisch ohne Effizienzverlust möglich.

Für spätere Benutzung stellen wir noch eine Hilfsaussage über Carmichael-Zahlen bereit.

Lemma 5.2.7
Wenn N eine Carmichael-Zahl ist, dann ist N keine Primzahlpotenz.

Beweis: Wir beweisen die Kontraposition: Wenn $N = p^\ell$ für eine ungerade Primzahl p und einen Exponenten $\ell \geq 2$ ist, dann ist N keine Carmichael-Zahl.

Dazu genügt es, eine Zahl $a \in \mathbb{Z}_N^*$ anzugeben, so dass $a^{N-1} \bmod N \neq 1$ ist. Wir definieren:

$$a := p^{\ell-1} + 1.$$

(Wenn z. B. $p = 7$ und $\ell = 3$ ist, ist $N = 343$ und $a = 49 + 1 = 50$.) Man sieht sofort, dass $a < p^\ell = N$ ist, und dass a nicht von p geteilt wird, also a und N teilerfremd sind; also ist $a \in \mathbb{Z}_N^*$. Nun rechnen wir modulo N, mit der binomischen Formel:

$$\begin{aligned} a^{N-1} &\equiv (p^{\ell-1} + 1)^{N-1} \\ &\equiv \sum_{0 \leq j \leq N-1} \binom{N-1}{j} (p^{\ell-1})^j \\ &\equiv 1 + (p^\ell - 1) \cdot p^{\ell-1} \pmod{N}. \end{aligned} \tag{5.2.2}$$

(Die letzte Äquivalenz ergibt sich daraus, dass für $j \geq 2$ gilt, dass $(\ell-1)j \geq \ell$ ist, also der Faktor $(p^{\ell-1})^j = p^{(\ell-1)j}$ durch $N = p^\ell$ teilbar ist, der entsprechende Summand also modulo N wegfällt.) Nun ist $p^\ell - 1$ nicht durch p teilbar, also ist $(p^\ell - 1) \cdot p^{\ell-1}$ nicht durch $N = p^\ell$ teilbar. Damit folgt aus (5.2.2), dass $a^{N-1} \not\equiv 1 \pmod{N}$ ist, also $a^{N-1} \bmod N \neq 1$. □

Folgerung: Jede Carmichael-Zahl N lässt sich als $N = N_1 \cdot N_2$ schreiben, wo N_1 und N_2 teilerfremde ungerade Zahlen ≥ 3 sind.

Bemerkung. Ganz ähnlich kann man sogar zeigen, dass die Primfaktoren einer Carmichael-Zahl N alle verschieden sein müssen. Weiter haben Carmichael-Zahlen mindestens drei Primfaktoren.[13] Schon aus diesen Tatsachen kann man entnehmen, dass Carmichael-Zahlen wohl eher selten sind.

5.2.3 Nichttriviale Quadratwurzeln der 1

Lemma 5.2.8
Wenn p eine ungerade Primzahl ist, dann gilt $b^2 \bmod p = 1$, $b \in \mathbb{Z}_p$, genau für $b \in \{1, p-1\}$.

[13] Für an solchen Details interessierte Leser/innen finden sich in Anhang B nicht prüfungsrelevante nähere Informationen zu Carmichael-Zahlen, u. a. ein mathematisches Kriterium und ein randomisierter Algorithmus, um sie zu erkennen.

Beweis: Offensichtlich gilt für jedes beliebige $m \geq 2$, dass $1^2 \bmod m = 1$ und $(m-1)^2 \bmod m = (m(m-2)+1) \bmod m = 1$ ist. Nun sei $b \in \{0, \ldots, p-1\}$ beliebig mit $b^2 \equiv 1 \pmod{p}$. Dann gilt $b^2 - 1 \equiv 0 \pmod{p}$, also ist p ein Teiler von $b^2 - 1 = (b+1)(b-1)$. Nach Fakt 5.1.26 ist p Teiler von $b+1$ oder von $b-1$. Im ersten Fall ist $b \equiv -1 \pmod{p}$, im zweiten Fall ist $b \equiv 1 \pmod{p}$. □

Die im Lemma angegebene Eigenschaft lässt sich in ein weiteres Zertifikat dafür ummünzen, dass eine Zahl N zusammengesetzt ist:

> Wenn es ein $b \in \{2, \ldots, N-2\}$ gibt, das $b^2 \bmod N = 1$ erfüllt,
> dann ist N zusammengesetzt.

Zahlen $b \in \{2, \ldots, N-2\}$ mit $b^2 \bmod N = 1$ heißen ***nichttriviale Quadratwurzeln der* 1 modulo *N***. Solche Zahlen gibt es nur, wenn N zusammengesetzt ist. Beispielsweise sind genau die Zahlen 1, 27, 64 und 90 die Quadratwurzeln der 1 modulo 91; davon sind 27 und $64 = 91 - 27$ nichttrivial. Wir beobachten: $27^2 \bmod 91 = 729 \bmod 91 = 1$. Beachte, dass $27 \equiv -1 \pmod{7}$ und $27 \equiv 1 \pmod{13}$. Allgemeiner sieht man mit der Verallgemeinerung von Fakt 5.1.21 (Chinesischer Restsatz) auf r Faktoren leicht ein, dass es für ein Produkt $N = p_1 \cdots p_r$ aus verschiedenen ungeraden Primzahlen $p_1, \ldots, p_r$ genau 2^r Quadratwurzeln der 1 modulo N gibt, nämlich die Zahlen b, $0 \leq b < N$, die $b \bmod p_j \in \{1, p_j - 1\}$, $1 \leq j \leq r$, erfüllen. Wenn N nicht sehr viele verschiedene Primfaktoren hat, ist es also aussichtslos, einfach zufällig gewählte b's darauf zu testen, ob sie vielleicht nichttriviale Quadratwurzeln der 1 sind. Dennoch wird uns dieser Begriff bei der Formulierung eines effizienten Primzahltests helfen.

5.2.4 Der Miller-Rabin-Test

Wir kehren nochmals zum Fermat-Test zurück und sehen uns die dort durchgeführte Exponentiation $a^{N-1} \bmod N$ etwas genauer an. Die Zahl $N-1$ ist gerade, daher kann man sie als $N - 1 = u \cdot 2^k$ schreiben, für eine ungerade Zahl u und ein $k \geq 1$. (In der Binärdarstellung von $N-1$ ist k die Anzahl der Nullen am Ende, u ist die Zahl, die durch Weglassen dieser Nullen entsteht.) Dann gilt $a^{N-1} \equiv (a^u \bmod N)^{2^k} \bmod N$,

a	$b_0 = a^{81}$	$b_1 = a^{162}$	$b_2 = a^{324}$	F-Z.?	MR-Z.?
2	252	129	66	×	×
7	307	324	1		
32	57	324	1		
49	324	1	1		
65	0	0	0	×	×
126	1	1	1		
201	226	51	1		×
224	274	1	1		×

Tabelle 4: Potenzen $a^{N-1} \bmod N$ mit Zwischenschritten, $N = 325$

und wir können $a^{N-1} \bmod N$ mit $k+1$ Zwischenschritten berechnen: Mit

$$\begin{aligned}
b_0 &= a^u \bmod N \\
b_1 &= b_0^2 \bmod N = a^{u\cdot 2} \bmod N, \\
b_2 &= b_1^2 \bmod N = a^{u\cdot 2^2} \bmod N, \\
&\vdots \\
b_i &= b_{i-1}^2 \bmod N = a^{u\cdot 2^i} \bmod N, \\
&\vdots \\
b_k &= b_{k-1}^2 \bmod N = a^{u\cdot 2^k} \bmod N
\end{aligned}$$

ist $b_k = a^{N-1} \bmod N$. Beispielsweise erhalten wir für $N = 325 = 5^2 \cdot 13$ den Wert $N - 1 = 324 = 81 \cdot 2^2$. (Man bemerkt, dass 325 die Binärdarstellung 101000100 und 81 die Binärdarstellung 1010001 hat.) In Tabelle 4 berechnen wir a^{81}, a^{162} und a^{324}, alle modulo 325, für verschiedene a.

Die Grundidee des Miller-Rabin-Tests ist nun, diese verlangsamte Berechnung der Potenz $a^{N-1} \bmod N$ auszuführen und dabei nach nichttrivialen Quadratwurzeln der 1 Ausschau zu halten.

b_0	b_1	$\cdots$				$\cdots$	b_{k-1}	b_k	Fall	F-Z.?	MR-Z.?
1	1	$\cdots$	1	1	1	$\cdots$	1	1	1		
$N-1$	1	$\cdots$	1	1	1	$\cdots$	1	1	2a		
$*$	$*$	$\cdots$	$*$	$N-1$	1	$\cdots$	1	1	2b		
$*$	$*$	$\cdots$	$*$	$*$	$*$	$\cdots$	$*$	$\neq 1$	3	$\times$	$\times$
$*$	$*$	$\cdots$	$*$	1	1	$\cdots$	1	1	4a		$\times$
$*$	$*$	$\cdots$	$*$	$*$	$*$	$\cdots$	$*$	1	4b		$\times$

Tabelle 5: Potenzen a^{N-1} mod N berechnet mit Zwischenschritten, mögliche Fälle.

Im Beispiel in Tabelle 4 sehen wir, dass 2 ein F-Zeuge für 325 ist, der in $\mathbb{Z}_{325}^*$ liegt, und dass 65 ein F-Zeuge ist, der nicht in $\mathbb{Z}_{325}^*$ liegt. Dagegen sind 7, 32, 49, 126, 201 und 224 alles F-Lügner für 325. Wenn wir aber 201^{324} mod 325 mit zwei Zwischenschritten berechnen, dann entdecken wir, dass 51 eine nichttriviale Quadratwurzel der 1 ist, was beweist, dass 325 keine Primzahl ist. Ähnlich liefert die Berechnung mit Basis 224, dass 274 eine nichttriviale Quadratwurzel der 1 ist. Die entsprechenden Berechnungen mit 7, 32 und 49 dagegen liefern keine weiteren Informationen, weil $7^{162} \equiv 32^{162} \equiv -1 \pmod{325}$ und $49^{81} \equiv -1 \pmod{325}$ gilt. Auch die Berechnung der Potenzen von 126 liefert keine nichttriviale Quadratwurzel der 1, weil $126^{81} \bmod 325 = 1$ gilt.

Wie kann die Folge $b_0, \ldots, b_k$ überhaupt aussehen? Wenn $b_i = 1$ oder $b_i = N-1$ gilt, dann sind $b_{i+1}, \ldots, b_k$ alle gleich 1. Daher beginnt die Folge $b_0, \ldots, b_k$ mit einer (eventuell leeren) Folge von Elementen $\notin \{1, N-1\}$ und endet mit einer (eventuell leeren) Folge von Einsen. Diese beiden Teile können von einem Eintrag $N-1$ getrennt sein; dies muss aber nicht so sein. Die möglichen Muster sind in Tabelle 5 angegeben. Dabei steht „$*$“ für ein Element $\notin \{1, N-1\}$. Wir unterscheiden vier Fälle, mit einigen Unterfällen:

Fall 1: $b_0 = 1$.

Fall 2: In $b_0, \ldots, b_{k-1}$ kommt der Wert $N-1$ vor.
(Fall 2a: $b_0 = N-1$; Fall 2b: $b_i = N-1$ für ein $i \in \{1, \ldots, k-1\}$.)

Fall 3: $b_k \neq 1$. – Dann ist N zusammengesetzt, weil a ein F-Zeuge für N ist.

(Damit dies passieren kann, müssen alle Werte $b_0, \ldots, b_{k-1}$ von 1 und $N-1$ verschieden sein.)

Fall 4: Es gibt ein i mit $0 < i \leq k$ mit $b_{i-1} \notin \{1, N-1\}$ und $b_i = 1$. – Dann ist N zusammengesetzt, weil b_{i-1} eine nichttriviale Quadratwurzel der 1 ist. (Wenn dies passiert, sind alle Werte $b_0, \ldots, b_{i-1}$ von 1 und $N-1$ verschieden.)

Wir beobachten: Wenn N eine Primzahl ist, dann gilt nach Fakt 5.1.18 (Kleiner Satz von Fermat) für jedes a die Gleichung $b_k = a^{N-1} \bmod N = 1$. Weiter kann es nicht passieren, dass $b_{i-1} \notin \{1, N-1\}$ und $b_i = 1$ ist. Daraus folgt, dass für eine Primzahl N *nur* die Fälle 1 und 2 vorkommen können. Wenn wir bei Eingaben N, die Primzahlen sind, keinen Fehler machen wollen, müssen wir also in Fällen 1 und 2 die Ausgabe 0 erzeugen. (Bei einer zusammengesetzten Zahl N als Eingabe tritt Fall 1 etwa für $a = 1$ auf und Fall 2a für $a = N-1$, weil u ungerade ist. Aber auch Fall 2b kann vorkommen; in Tab. 4 ist etwa $a = 32$ eine solche Zahl.)

In den Fällen 3 und 4 stellt die Zahl a (mit ihren Potenzen $b_0, \ldots, b_k$) hingegen einen Beleg dafür dar, dass N keine Primzahl ist: Im Fall 3 ist a ein F-Zeuge, im Fall 4 ist b_{i-1} eine nichttriviale Quadratwurzel der 1, und das kann nur passieren, wenn N zusammengesetzt ist. Hier sollten wir also 1 ausgeben.

Das ist auch schon der ganze Algorithmus von Miller-Rabin, abstrakt formuliert: Wähle a aus $\{1, \ldots, N-1\}$ zufällig. Finde heraus, ob Fall 1 oder 2 eintritt (dann ist die Ausgabe 0) oder Fall 3 oder 4 eintritt (dann ist die Ausgabe 1).

Um einen besonders effizienten Algorithmus zu bekommen, wollen wir die Fallunterscheidung noch etwas stromlinienförmiger gestalten. Betrachten von Tab. 5 zeigt, dass Fall 1 oder 2 genau dann eintritt, wenn Folgendes gilt:

$$b_0 = 1 \quad \text{oder} \quad \text{in der Folge } b_0, \ldots, b_{k-1} \text{ zu } a \text{ erscheint der Wert } N-1. \tag{5.2.3}$$

(Interessanterweise spielt das letzte Folgenglied b_k gar keine Rolle für die Unterscheidung!) Dies führt zu folgender Definition in Analogie zu der der F-Zeugen und F-Lügner.

Definition 5.2.9
Sei $N \geq 3$ ungerade und zusammengesetzt.
Schreibe $N - 1 = u \cdot 2^k$, für u ungerade, $k \geq 1$.
Eine Zahl a, $1 \leq a < N$, heißt ein **MR-Lügner für N**, wenn (5.2.3) gilt.
Die Menge der MR-Lügner für N nennen wir L_N^{MR}.
Eine Zahl a, $1 \leq a < N$, heißt ein **MR-Zeuge für N**, wenn (5.2.3) nicht gilt.
(D. h.: $a^u \notin \{1, N-1\}$ und in der Folge $b_1, \ldots, b_{k-1}$ kommt $N-1$ nicht vor.)

Aus der obigen Diskussion der möglichen Fälle folgt sofort:

Lemma 5.2.10
Sei $N \geq 3$ ungerade.

(a) a ist F-Zeuge für $N \quad \Rightarrow \quad a$ ist MR-Zeuge für N (Fall 3).

(b) Wenn N eine Primzahl ist, dann gilt (5.2.3) für alle $a < N$.

Der Test selbst ist leicht und sehr effizient durchführbar: Man wählt a zufällig, berechnet zunächst $b_0 = a^u \bmod N$ und testet, ob $b_0 \in \{1, N-1\}$ ist. Falls ja, trifft Fall 1 bzw. Fall 2a zu, die Ausgabe ist 0. Falls nein, berechnet man durch iteriertes Quadrieren nacheinander die Werte $b_1, b_2, \ldots$, bis

- entweder der Wert $N-1$ auftaucht (Fall 2b, Ausgabe 0)
- oder der Wert 1 auftaucht (Fall 4a, a ist MR-Zeuge, Ausgabe 1)
- oder $b_1, \ldots, b_{k-1}$ berechnet worden sind, ohne dass Werte $N-1$ oder 1 vorgekommen sind (Fall 3 oder 4b, a ist MR-Zeuge, Ausgabe 1).

In Pseudocode sieht das Verfahren dann folgendermaßen aus. Im Unterschied zur bisherigen Beschreibung wird nur eine Variable `b` benutzt, die nacheinander die Werte $b_0, b_1, \ldots$ aufnimmt.

Algorithmus 5.2.11 ***Miller-Rabin-Primzahltest***
EINGABE: Ungerade Zahl $N \geq 3$.
METHODE:

```
1    Bestimme u ungerade und k ≥ 1 mit N − 1 = u · 2^k;
2    wähle zufällig ein a aus {1, ..., N − 1};
3    b ← a^u mod N;                          // mit schneller Exponentiation
4    if b ∈ {1, N − 1} then return 0;        // Fall 1 oder 2a
5    for j from 1 to k − 1 do                // „wiederhole (k − 1)-mal“
6        b ← b^2 mod N;
7        if b = N − 1 then return 0;         // Fall 2b
8        if b = 1 then return 1;             // Fall 4a
9    return 1.                               // Fall 3 oder 4b
```

Wie steht es mit der Laufzeit des Algorithmus? Man benötigt höchstens $\log N$ Divisionen durch 2, um u und k zu finden (Zeile 1). (Wenn man benutzt, dass N normalerweise in Binärdarstellung gegeben sein wird, ist die Sache noch einfacher: k ist die Zahl der Nullen, mit der die Binärdarstellung von $N-1$ aufhört, u ist die Zahl, die durch den Binärstring ohne diese letzten Nullen dargestellt wird.) Die Berechnung von $a^u \bmod N$ mit schneller Exponentiation in Zeile 3 benötigt $O(\log N)$ arithmetische Operationen und $O((\log N)^3)$ Ziffernoperationen (mit der Schulmethode für Multiplikation und Division). Die Schleife in Zeilen 5–8 wird weniger als k-mal durchlaufen; offenbar ist $k \leq \log N$. In jedem Durchlauf ist die Multiplikation modulo N die teuerste Operation. Insgesamt benutzt der Algorithmus $O(\log N)$ arithmetische Operationen auf Zahlen, die kleiner als N^2 sind, und $O((\log N)^3)$ Ziffernoperationen. Dies ist auch die Rechenzeit des Fermat-Tests. Tatsächlich verursacht der Miller-Rabin-Test im Vergleich zum Fermat-Test kaum Mehraufwand.

Nun wenden wir uns dem Ein-/Ausgabeverhalten des Miller-Rabin-Tests zu.

Lemma 5.2.12
Wenn die Eingabe N für den Miller-Rabin-Test eine Primzahl ist, dann wird 0 ausgegeben.

Beweis: Wir haben oben schon festgestellt, dass Ausgabe 1 genau dann erfolgt, wenn die zufällig gewählte Zahl a ein MR-Zeuge ist. Nach Lemma 5.2.10(b) gibt es nur dann MR-Zeugen, wenn N zusammengesetzt ist. □

Historische Notiz. Im Jahr 1976 stellte Gary L. Miller[14] einen deterministischen Algorithmus vor, der auf der Idee beruhte, die Folge $b_0, \ldots, b_k$ zu betrachten. Dieser testete die kleinsten $O((\log N)^2)$ Elemente $a \in \{2, 3, \ldots, N-1\}$ auf die Eigenschaft, MR-Zeugen zu sein. Der Algorithmus hat polynomielle Laufzeit und liefert das korrekte Ergebnis, wenn die „erweiterte Riemannsche Vermutung (ERH)“ stimmt, eine berühmte Vermutung aus der Zahlentheorie, die bis heute unbewiesen ist. Um 1976 schlug Michael Rabin[15] vor, Millers Algorithmus so zu modifizieren, dass die zu testende Zahl a zufällig gewählt wird. Er und (unabhängig) Louis Monier[16] bewiesen dann, dass dieser Algorithmus konstante Fehlerwahrscheinlichkeit kleiner als $\frac{1}{4}$ hat. Der Algorithmus heißt heute allgemein der ***Miller-Rabin-Test***. Es sollte noch erwähnt werden, dass der ***Solovay-Strassen-Test***[17], der ein ähnliches Verhalten wie der Miller-Rabin-Test hat, aber nicht ganz so effizient ist, (1974 eingereicht und) 1977 publiziert wurde.

5.2.5 Fehlerschranke für den Miller-Rabin-Test

Wir müssen nun noch die Fehlerwahrscheinlichkeit des Miller-Rabin-Tests für den Fall analysieren, dass N eine zusammengesetzte (ungerade) Zahl ist. Dies wird also für diesen Abschnitt angenommen.

Wir beweisen, dass die Wahrscheinlichkeit, dass der Miller-Rabin-Test die (unerwünschte) Antwort 0 gibt, kleiner als $\frac{1}{2}$ ist. Dazu wollen wir ähnlich vorgehen wie bei der Analyse des Fermat-Tests für Nicht-Carmichael-Zahlen (Beweis von Satz 5.2.4). Dort haben wir gezeigt, dass die F-Lügner für solche N höchstens die Hälfte von $\mathbb{Z}_N^*$ ausmachen.

Wenn N keine Carmichael-Zahl ist, ist dies leicht: Nach Lemma 5.2.10(a) gilt $L_N^{\mathrm{MR}} \subseteq L_N^{\mathrm{F}}$, und wir können den Beweis von Satz 5.2.4 verwenden.

Es bleibt der Fall, dass N eine Carmichael-Zahl ist. Unser Plan ist, eine Menge B_N mit $L_N^{\mathrm{MR}} \subseteq B_N \subseteq \mathbb{Z}_N^*$ zu definieren, die höchstens die Hälfte der Elemente von $\mathbb{Z}_N^*$ enthält.

[14] Gary Lee Miller, amer. Informatiker, Paris-Kanellakis-Preis 2003.

[15] Michael O. Rabin (*1931), israelischer Mathematiker und Informatiker, Turing Award 1976.

[16] Louis Monier, frz.-am. Informatiker, Mitbegründer der AltaVista-Suchmaschine

[17] Robert M. Solovay (*1938), amer. Mathematiker; und Volker Strassen (*1936), deuscher Mathematiker, Paris-Kanellakis-Preis 2003, Knuth Prize 2008.

a	b_0	b_1	$\cdots$		b_{i_0}		$\cdots$	b_{k-1}	b_k
$a_1 = 1$	1	1	$\cdots$	1	1	1	$\cdots$	1	1
$a_2 = N-1$	$N-1$	1	$\cdots$	1	1	1	$\cdots$	1	1
a_3	$*$	$*$	$\cdots$	$*$	$N-1$	1	$\cdots$	1	1
a_4	$*$	$*$	$\cdots$	$N-1$	1	1	$\cdots$	1	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	1	1
a	a^u	$a^{u\cdot 2^1}$	$\cdots$	$a^{u\cdot 2^{i_0-1}}$	$a^{u\cdot 2^{i_0}}$	$a^{u\cdot 2^{i_0+1}}$	$\cdots$	$a^{u\cdot 2^{k-1}}$	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$a_\#$	$*$	$*$	$\cdots$	$*$	$N-1$	1	$\cdots$	1	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$a_?$	$*?$	$*?$	$\cdots$	$*?$	$*?$	$\cdots$	$\cdots$	$\cdots$	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$a_{\varphi(N)}$	?	?	$\cdots$	?	?	?	$\cdots$	?	1

Tabelle 6: Sicht auf B_N als Teilmenge von $\mathbb{Z}_N^*$

Weil u ungerade ist, gilt

$$(N-1)^{u\cdot 2^0} \equiv (N-1)^u \equiv (-1)^u \equiv -1 \equiv N-1 \pmod{N}.$$

Sei i_0 das größte $i \in \{0, 1, \ldots, k\}$ mit der Eigenschaft, dass es einen MR-Lügner $a_\#$ mit $a_\#^{u\cdot 2^i} \bmod N = N-1$ gibt. (Wir werden dieses $a_\#$ weiter unten nochmals benutzen.) Weil N eine Carmichael-Zahl ist, gilt $a^{u\cdot 2^k} \bmod N = a^{N-1} \bmod N = 1$ für alle $a \in \mathbb{Z}_N^*$, und daher $0 \le i_0 < k$. Wir definieren:

$$B_N := \{a \in \mathbb{Z}_N^* \mid a^{u\cdot 2^{i_0}} \bmod N \in \{1, N-1\}\}. \tag{5.2.4}$$

Zur Veranschaulichung betrachte man Tabelle 6. In der dort angegebenen Matrix entspricht jede Zeile einem der $\varphi(N)$ Elemente von $\mathbb{Z}_N^*$, beginnend mit $a_1 = 1$ und $a_2 = -1 \equiv N-1$. In der Zeile für a ist die von a erzeugte Folge $b_0, \ldots, b_k$ eingetragen. Dass N eine Carmichael-Zahl ist, drückt sich dadurch aus, dass alle Einträge in der

b_k-Spalte gleich 1 sind. Der Eintrag für $a_2 = -1$ in der b_0-Spalte ist $N-1$. Spalte i_0 ist die am weitesten rechts stehende Spalte, in der es einen Eintrag $-1 \equiv N-1$ gibt, und $a_\#$ ist ein Element von $\mathbb{Z}_N^*$, das zu diesem Eintrag führt. Die Menge B_N besteht aus den Elementen von $\mathbb{Z}_N^*$, die in Spalte i_0 Eintrag 1 oder $N-1$ haben. – Wir werden nun zeigen, dass diese Menge B_N die gewünschten Eigenschaften hat.

Lemma 5.2.13

(a) $L_N^{\mathrm{MR}} \subseteq B_N$.

(b) $B_N \subsetneq \mathbb{Z}_N^*$.

(c) $|B_N| \leq \frac{1}{2}|\mathbb{Z}_N^*|$, also $\mathbf{Pr}_{a\in\{1,\ldots,N-1\}}(a \text{ ist MR-Lügner}) < \frac{1}{2}$.

Beweis: (Teil (a) ist nur die Überprüfung, dass die Definition von B_N technisch sinnvoll ist. Der eigentlich interessante Teil ist (b). Teil (c) ist dann Routine.)

(a) Sei a ein beliebiger MR-Lügner.
Fall 1: $a^u \bmod N = 1$. – Dann ist auch $a^{u\cdot 2^{i_0}} \bmod N = 1$, und daher gilt $a \in B_N$. (Die Zeile zu a in Tab. 6 sieht aus wie die von $a_1 = 1$.)
Fall 2: $a^{u\cdot 2^i} \bmod N = N-1$ für ein $i < k$. – Dann ist $0 \leq i \leq i_0$ nach der Definition von i_0. Wenn $i = i_0$ ist, haben wir direkt $a \in B_N$ (Beispiel in Tab. 6: a_3 oder $a_\#$); wenn $i < i_0$ ist, dann gilt

$$a^{u\cdot 2^{i_0}} \bmod N = (a^{u\cdot 2^i} \bmod N)^{2^{i_0-i}} \bmod N = (-1)^{2^{i_0-i}} \bmod N = 1,$$

also ebenfalls $a \in B_N$ (Beispiel in Tab. 6: a_2 oder a_4).

(b) Wir benutzen die in Lemma 5.2.7 beobachtete Eigenschaft von Carmichael-Zahlen, keine Primzahlpotenz zu sein. Nach diesem Lemma können wir $N = N_1 \cdot N_2$ schreiben, für teilerfremde ungerade Zahlen N_1, N_2. Im Folgenden werden wir diese Zerlegung und die Eigenschaften aus dem Chinesischen Restsatz (Fakt 5.1.21) benutzen.

Die grobe Idee der nun folgenden Konstruktion ist folgende: Wir haben das Element 1 mit $1^{u\cdot 2^{i_0}} \bmod N = 1$ und das Element $a_\#$ mit $a_\#^{u\cdot 2^{i_0}} \equiv -1 \pmod N$. Aus diesen beiden Elementen „basteln“ wir mit Hilfe des Chinesischen Restsatzes ein $b \in \mathbb{Z}_N^*$, das sich modulo N_1 wie 1 und modulo N_2 wie $a_\#$ verhält. Es wird sich zeigen, dass $b^{u\cdot 2^{i_0}} \bmod N \notin \{1, N-1\}$ gilt, also $b \notin B_N$ ist.

Wir führen diese Idee jetzt formal durch. Sei $x_2 = a_\# \bmod N_2$. Nach dem Chinesischen Restsatz (Fakt 5.1.21) gibt es eine eindeutig bestimmte Zahl $b \in \{0, 1, \ldots, N-1\}$,

die

$$b \equiv 1 \pmod{N_1} \quad \text{und} \quad b \equiv x_2 \pmod{N_2}$$

erfüllt. (Es folgt $b \equiv a_\# \pmod{N_2}$.) Wir zeigen, dass b in $\mathbb{Z}_N^* - B_N$ liegt.

Wir notieren, dass für beliebige $x, x' \in \mathbb{Z}$ gilt:

$$x \equiv x' \pmod{N} \qquad \Rightarrow \qquad x \equiv x' \pmod{N_i}, \quad \text{für } i = 1, 2. \tag{5.2.5}$$

(Wenn $x - x'$ durch N teilbar ist, dann auch durch N_1 und N_2.)

Beh. 1: $b \in \mathbb{Z}_N^*$.
Bew.: Offensichtlich gilt $b^{N-1} \equiv 1^{N-1} \equiv 1 \pmod{N_1}$. Weiter haben wir, modulo N_2 gerechnet:

$$b^{N-1} \equiv a_\#^{N-1} \overset{(5.2.5)}{\equiv} (a_\#^{N-1} \bmod N) \equiv 1 \pmod{N_2}.$$

Wegen der Eindeutigkeitsaussage im Chinesischen Restsatz folgt daraus $b^{N-1} \equiv 1 \pmod{N}$. Nach Lemma 5.2.2 folgt $b \in \mathbb{Z}_N^*$.

Beh. 2: $b \notin B_N$.
Bew.: Indirekt. Annahme: $b \in B_N$, d. h. $b^{u \cdot 2^{i_0}} \equiv 1 \pmod{N}$ oder $b^{u \cdot 2^{i_0}} \equiv -1 \pmod{N}$.

1. *Fall*: $b^{u \cdot 2^{i_0}} \equiv 1 \pmod{N}$. – Mit (5.2.5) folgt $b^{u \cdot 2^{i_0}} \equiv 1 \pmod{N_2}$. Andererseits gilt

$$b^{u \cdot 2^{i_0}} \equiv x_2^{u \cdot 2^{i_0}} \equiv a_\#^{u \cdot 2^{i_0}} \overset{(5.2.5)}{\equiv} (a_\#^{u \cdot 2^{i_0}} \bmod N) \equiv N - 1 \equiv -1 \pmod{N_2},$$

ein Widerspruch, weil N_2 ungerade ist.

2. *Fall*: $b^{u \cdot 2^{i_0}} \equiv -1 \pmod{N}$. – Mit (5.2.5) folgt $b^{u \cdot 2^{i_0}} \equiv -1 \pmod{N_1}$. Andererseits gilt

$$b^{u \cdot 2^{i_0}} \equiv 1^{u \cdot 2^{i_0}} \equiv 1 \pmod{N_1},$$

ebenfalls ein Widerspruch, weil N_1 ungerade ist.

(c) Wir verwenden die in (b) konstruierte Zahl $b \in \mathbb{Z}_N^* - B_N$. Wie im Beweis von Satz 5.2.4 betrachtet man die injektive Funktion $g_b \colon B_N \ni a \mapsto ba \bmod N \in \mathbb{Z}_N^*$. Es gilt $g_b(a) \notin B_N$ für jedes $a \in B_N$, denn

$$g_b(a)^{u \cdot 2^{i_0}} \bmod N = (b^{u \cdot 2^{i_0}} \bmod N)(a^{u \cdot 2^{i_0}}) \bmod N \in \{b^{u \cdot 2^{i_0}} \bmod N, (N - b^{u \cdot 2^{i_0}}) \bmod N\}.$$

Daraus ergibt sich sofort $|B_N| \leq \frac{1}{2}|\mathbb{Z}_N^*|$. □

Wir haben also eine Schranke von $\frac{1}{2}$ für die Irrtumswahrscheinlichkeit im Miller-Rabin-Algorithmus bewiesen. Eine etwas kompliziertere Analyse zeigt, dass sogar die Fehlerschranke $\frac{1}{4}$ gilt; man kann auch zeigen, dass es zusammengesetzte Zahlen N gibt (zum Beispiel $703 = 19 \cdot 37$), bei denen eine Fehlerwahrscheinlichkeit von fast $\frac{1}{4}$ tatsächlich auftritt. Durch ℓ-fache Wiederholung, ebenso wie in Algorithmus 5.2.5, kann man die Fehlerschranke auf $4^{-\ell}$ reduzieren. Wir kürzen den Miller-Rabin-Test mit „MiRa-Test“ ab. Man beachte, dass bei jedem neuen Aufruf eine neue Zufallszahl a gewählt wird.

Algorithmus 5.2.14 ***Iterierter Miller-Rabin-Test***
EINGABE: Ungerade Zahl $N \geq 3$, eine Zahl $\ell \geq 1$.
METHODE:

```
1    repeat ℓ times
2        if MiRa-Test(N) = 1 then return 1;
3    return 0;
```

Diesen Test wollen wir kurz IterMiRa(N, ℓ) nennen. – Wir erhalten zusammenfassend:

Proposition 5.2.15
Algorithmus 5.2.14 benötigt $O(\ell \log N)$ arithmetische Operationen auf Zahlen, die kleiner als N^2 sind, und $O(\ell(\log N)^3)$ Bitoperationen. Wenn N eine Primzahl ist, ist die Ausgabe 0, wenn N zusammengesetzt ist, ist die Wahrscheinlichkeit, dass 0 ausgegeben wird, kleiner als $4^{-\ell}$.

5.3 Die Erzeugung von Primzahlen

Für kryptographische Anwendungen (zum Beispiel für die Erzeugung von Schlüsselpaaren für das RSA-Public-Key-Kryptosystem) werden vielziffrige Primzahlen benötigt. Eine typische Aufgabe in diesem Zusammenhang könnte lauten:

Finde eine zufällige Primzahl mit μ Bits!

Dabei hängt μ von der Anwendung ab; wir können uns z. B. $\mu = 512$ oder $\mu = 2048$ vorstellen.

Man erinnere sich an den Primzahlsatz und an den Satz von Finsler, die angeben, wie viele Primzahlen es in $[m, 2m)$ gibt.

Ein naheliegender Ansatz zur Erzeugung einer zufälligen Primzahl in $[m, 2m)$ ist folgender: Man wählt wiederholt eine (ungerade) Zahl aus $[m, 2m)$ zufällig und wendet auf sie den Miller-Rabin-Test an. Dies wiederholt man, bis eine Zahl gefunden wurde, die die Ausgabe 0 liefert.

Algorithmus 5.3.1 ***GetPrime – Randomisierte Primzahlerzeugung***
EINGABE: Zahl $m \geq 2$, Zahl $\ell \geq 1$ // Zuverlässigkeitsparameter
METHODE:
1 **repeat**
2 $\mathtt{N} \leftarrow$ zufällige ungerade Zahl in $[m, 2m)$;
3 **until** IterMiRa$(\mathtt{N}, \ell) = 0$; // Algorithmus 5.2.14
4 **return** N.

Die Ausgabe ist korrekt, wenn er eine Primzahl zurückgibt, ein Fehler tritt auf, wenn eine zusammengesetzte Zahl zurückgegeben wird. Auf den ersten Blick könnte man meinen (aufgrund von Proposition 5.2.15), dass die Fehlerwahrscheinlichkeit höchstens $1/4^\ell$ ist – eben die Fehlerwahrscheinlichkeit des iterierten MiRa-Tests. Wir werden sehen, dass wir auf der Basis der Analyse des Miller-Rabin-Tests nur ein etwas schwächeres Ergebnis bekommen.[18]

Wir definieren: $\text{Prim}_m := \{p \in [m, 2m) \mid p \text{ ist Primzahl}\}$.

[18] Tatsächlich ist die Fehlerwahrscheinlichkeit durch $1/4^\ell$ beschränkt, für genügend große m. Dies kann man aber nur durch fortgeschrittene zahlentheoretische Untersuchungen über das Verhalten einer zufälligen ungeraden Zahl beim Miller-Rabin-Test beweisen [P. Beauchemin, G. Brassard, C. Crépeau, C. Goutier, C. Pomerance: The generation of random numbers that are probably prime. *J. Cryptology* **1**(1): 53-64 (1988)].

Satz 5.3.2
Bei der Anwendung von Algorithmus 5.3.1 auf eine gerade Zahl m gilt:

(a) Für jedes $p \in \mathrm{Prim}_m$ gilt:

$$\mathbf{Pr}(\mathrm{GetPrime}(m,\ell) = p \mid \mathrm{GetPrime}(m,\ell) \in \mathrm{Prim}_m) = \frac{1}{|\mathrm{Prim}_m|}.$$

In Worten: Jede Primzahl in $[m, 2m)$ hat dieselbe Wahrscheinlichkeit, als Ergebnis zu erscheinen.

(b)

$$\mathbf{Pr}(\mathrm{GetPrime}(m,\ell) \notin \mathrm{Prim}_m) \leq \frac{3\ln(2m)}{2\cdot 4^\ell} = O\left(\frac{\log m}{4^\ell}\right).$$

((!) Nicht $1/4^\ell$, wie naiv vermutet.)

(c) Die erwartete Rundenzahl ist $\leq \frac{3}{2}\ln(2m)$, der erwartete Rechenaufwand ist $O(\ell(\log m)^2)$ arithmetische Operationen und $O(\ell(\log m)^4)$ Bitoperationen.

Beweis: (a) Eine Primzahl wird als Resultat genau dann geliefert, wenn keine zusammengesetzte Zahl fälschlicherweise vom Miller-Rabin-Test akzeptiert wird, bevor (in Zeile 2) die erste echte Primzahl gewählt wird. Jede der Primzahlen in $[m, 2m)$ hat dieselbe Wahrscheinlichkeit, diese erste gewählte Primzahl zu sein.

(b)

$$\begin{aligned}
&\mathbf{Pr}(\mathrm{GetPrime}(m,\ell) \notin \mathrm{Prim}_m)\\
&= \mathbf{Pr}\Big(\exists i \geq 1\text{: in Runden } j = 1,\dots,i-1 \text{ wird eine}\\
&\qquad\qquad \text{zusammengesetzte Zahl gewählt } \textit{und erkannt } \wedge\\
&\qquad\qquad \text{in Runde } i \text{ wird zusammengesetzte Zahl } N \text{ gewählt}\\
&\qquad\qquad \text{und der iterierte MiRa-Test auf } N \text{ liefert } 0\Big)\\
&\leq \sum_{i\geq 1}\left(1 - \frac{|\mathrm{Prim}_m|}{m/2}\right)^{i-1}\cdot\frac{1}{4^\ell}\\
&= \frac{m/2}{|\mathrm{Prim}_m|}\cdot\frac{1}{4^\ell}\\
&\leq \frac{3\ln(2m)}{2\cdot 4^\ell}.
\end{aligned}$$

Wir haben benutzt, dass es in $[m, 2m)$ genau $m/2$ ungerade Zahlen gibt (m ist gerade). Die letzte Ungleichung folgt aus der Ungleichung von Finsler (Satz 5.1.31).

(c) Da man in jeder Runde mit Wahrscheinlichkeit mindestens $\frac{|\mathrm{Prim}_m|}{m/2}$ eine Primzahl wählt, ist die erwartete Rundenzahl nicht größer als $\frac{m/2}{|\mathrm{Prim}_m|} = O(\log m)$. □

Bemerkung: Bei der Erzeugung zufälliger Primzahlen mit μ Bits für kryptographische Zwecke wird man aus Effizienzgründen nicht Algorithmus 5.3.1 mit $\ell = \Theta(\log \mu)$ anwenden, der $O(\ell\mu^2)$ Multiplikationen benötigt, sondern jede gewählte Zahl N zunächst auf Teilbarkeit durch Primzahlen kleiner als die Bitlänge μ testen (da viele zusammengesetzte Zahlen N kleine Teiler haben, werden diese sehr rasch als zusammengesetzt erkannt) und nachher eine Kombination zweier verschiedener Primzahltests (z. B. Miller-Rabin und „Lucas Strong Probable Prime Test“) jeweils mit nur einer oder zwei Iterationen anwenden. Dies erfordert nur $O(\mu)$ „volle“ Multiplikationen. Die Dichte der zusammengesetzten Zahlen, die von einem solchen Test nicht erkannt werden, ist als sehr gering einzuschätzen. Über eine klassische experimentelle Untersuchung hierzu berichtet die kurze Notiz

`people.csail.mit.edu/rivest/Rivest-FindingFourMillionLargeRandomPrimes.ps`

von Ron Rivest (bekannt von „RSA“ und von „Cormen, Leiserson, Rivest und Stein“).

5.4 Quadratische Reste und Quadratwurzeln modulo p

Wir betrachten hier den Begriff des quadratischen Rests und der Quadratwurzeln modulo p, für Primzahlen p.

Beispiel: Für $p = 13$ und $p = 19$ betrachten wir die Quadrate der Zahlen in $\mathbb{Z}_p^*$.

a	1	2	3	4	5	6	7	8	9	10	11	12
a^2	1	4	9	16	25	36	49	64	81	100	121	144
$a^2 \bmod 13$	1	4	9	3	12	10	10	12	3	9	4	1

a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$a^2 \bmod 19$	1	4	9	16	6	17	11	7	5	5	7	11	17	6	16	9	4	1

Wir bemerken, dass in beiden Fällen jede Zahl entweder keinmal oder zweimal als Quadrat vorkommt, dass infolgedessen genau $\frac{1}{2}(p-1)$ Zahlen in $\mathbb{Z}_p^*$ als Quadratzahlen

auftauchen. Im Fall $p = 13$ ist $12 = -1 \bmod 13$ eine Quadratzahl, im Fall $p = 19$ ist $18 = -1 \bmod 19$ keine Quadratzahl. Es stellt sich die Frage, wie man zu gegebenem p und a ermitteln kann, ob a in $\mathbb{Z}_p$ ein Quadrat ist (einfach, wenn man weiß, wie es geht) und wie man gegebenenfalls ein c mit $c^2 \bmod p = a$ finden kann (eventuell nicht ganz so einfach).

Bemerkung: In $\mathbb{Z}_p$ spielt $p-1$ die Rolle von -1, dem additiven Inversen von 1.

Definition 5.4.1
Für eine ungerade Primzahl p sei

$$\mathrm{QR}_p := \{a \in \mathbb{Z}_p^* \mid \exists b \in \mathbb{Z}_p^* \colon b^2 \bmod p = a\}$$

die Menge der „quadratischen Reste“ in $\mathbb{Z}_p^*$.

Wir stellen einige auch für sich wichtige Hilfsaussagen bereit.

Fakt 5.4.2
Wenn p eine ungerade Primzahl ist, dann gilt:

(a) Wenn $d \geq 1$ ist, dann hat ein Polynom $f(X) = X^d + a_{d-1}X^{d-1} + \cdots + a_1 X + a_0$ in $\mathbb{Z}_p$ höchstens d Nullstellen.

(b) Für $s \geq 1$ gibt es in $\mathbb{Z}_p^*$ höchstens s Elemente b mit $b^s \bmod p = 1$.

Beweis: (a) Dies ist eine Tatsache, die in Körpern allgemein gilt. Man kann sich recht leicht überlegen, dass man für Nullstellen $b_1, \ldots, b_k$ aus $f(X)$ den Faktor $(X - b_1) \cdots (X - b_k)$ ausklammern kann, d. h. $f(X) = (X - b_1) \cdots (X - b_k) \cdot g(X)$ schreiben kann. Dann kann k nicht größer als der Grad d von $f(X)$ sein.
(b) Wenn $b_1, \ldots, b_{s+1}$ verschiedene Lösungen wären, hätte das Polynom $X^s - 1$ Grad s, aber $> s$ Nullstellen, was (a) widerspricht. □

Wir verallgemeinern Lemma 5.2.8 wie folgt:

Lemma 5.4.3
Wenn p eine ungerade Primzahl ist und $a \in \mathrm{QR}_p$, dann gibt es genau zwei Zahlen $b \in \mathbb{Z}_p$ mit $b^2 \bmod p = a$. Daher gilt: $|\mathrm{QR}_p| = \frac{1}{2}(p-1)$.

Beweis: Nach Definition von QR_p gibt es ein solches b. Dann gilt auch $(p-b)^2 \bmod p = (p^2 - 2pb + b^2) \bmod p = b^2 \bmod p = a$. Weil p ungerade ist, gilt $b - (p-b) \bmod p = 2b \bmod p \neq 0$, also $b \neq p - b$. Weitere Quadratwurzeln von a kann es nicht geben, da sonst das Polynom $X^2 - a$ vom Grad 2 mehr als zwei Nullstellen hätte, was nach Fakt 5.4.2(a) nicht möglich ist. □

Die folgende Behauptung erlaubt es, effizient zu testen, ob a ein quadratischer Rest modulo p ist.

Proposition 5.4.4 ***Eulersches Kriterium***
Für jede ungerade Primzahl p und jedes $a \in \mathbb{Z}_p^*$ gilt:

(a) $a^{(p-1)/2} \bmod p \in \{1, -1\}$.

(b) $a \in \mathrm{QR}_p \;\Leftrightarrow\; a^{(p-1)/2} \bmod p = 1$.

Beweis von Proposition 5.4.4:

(a) Nach dem kleinen Satz von Fermat (Fakt 5.1.18) gilt $a^{p-1} \bmod p = 1$ für jedes $a \in \mathbb{Z}_p^*$. Daraus folgt

$$\left(a^{(p-1)/2} \bmod p\right)^2 \bmod p = 1 \text{ für jedes } a \in \mathbb{Z}_p^*.$$

Mit Lemma 5.4.3 folgt, dass $a^{(p-1)/2} \bmod p$ eine der beiden Quadratwurzeln von 1 ist, also gleich 1 oder gleich -1 sein muss.

(b) Wir definieren $A := \{a \in \mathbb{Z}_p^* \mid a^{(p-1)/2} \bmod p = 1\}$. Unser Ziel ist zu zeigen, dass $\mathrm{QR}_p = A$ ist. Eine Inklusion ist leicht einzusehen:

Beh. 1: $\mathrm{QR}_p \subseteq A$. (Wenn $a \in \mathrm{QR}_p$, also $a = b^2 \bmod p$ für ein b, dann gilt

$$a^{(p-1)/2} \bmod p = (b^2 \bmod p)^{(p-1)/2} \bmod p = b^{p-1} \bmod p = 1,$$

nach dem kleinen Satz von Fermat. Also ist $a \in A$.)

Um die Gleichheit zu zeigen, benutzen wir ein Kardinalitätsargument.

Beh. 2: $|\mathrm{QR}_p| = \frac{1}{2}(p-1)$. Dies gilt, weil nach Lemma 5.4.3 jeder quadratische Rest genau zwei verschiedene Quadratwurzeln hat und natürlich jede der $p-1$ Zahlen b in $\mathbb{Z}_p^*$ Quadratwurzel von $b^2 \bmod p$ ist.

Beh. 3: $|A| \leq \frac{1}{2}(p-1)$. (Nach Fakt 5.4.2 (mit $s = \frac{1}{2}(p-1)$) gilt, dass höchstens $\frac{1}{2}(p-1)$ viele Elemente von $\mathbb{Z}_p^*$ die Gleichung $a^{(p-1)/2} \bmod p = 1$ erfüllen.)

Aus Beh. 1, 2 und 3 folgt $\mathrm{QR}_p = A$, wie gewünscht. □

Mit diesem Kriterium kann man mit einer schnellen Exponentiation testen, ob eine Zahl a quadratischer Rest modulo p ist oder nicht.

Algorithmus 5.4.5 ***Quadratzahltest modulo p***
Input: Primzahl p, Zahl a mit $1 < a < p$
Methode:
(1) Berechne $z = a^{(p-1)/2} \bmod p$; // Schnelle Exponentiation
(2) **if** $z = 1$ **then return** „Quadrat“ **else return** „Nichtquadrat“.

Wir rechnen versuchsweise in $\mathbb{Z}_{19}$, für $18 \notin \mathrm{QR}_{19}$ und $17 \in \mathrm{QR}_{19}$:

$18^9 \bmod 19 = (((-1)^8 \bmod 19) \cdot 18) \bmod 19 = 18 \neq 1$;
$17^9 \bmod 19 = (((-2)^8 \bmod 19) \cdot (-2)) \bmod 19 = (256 \cdot (-2)) \bmod 19 = (9 \cdot (-2)) \bmod 19 = 1$.

Wir notieren eine leichte Folgerung aus dem Eulerschen Kriterium:

Korollar 5.4.6
Sei p eine ungerade Primzahl. Dann gilt:

(a) Für $a, b \in \mathbb{Z}_p^*$ und das Produkt $c = ab \bmod p$ gilt:

$$\begin{aligned} a, b \in \mathrm{QR}_p &\Rightarrow c \in \mathrm{QR}_p; \\ a, b \notin \mathrm{QR}_p &\Rightarrow c \in \mathrm{QR}_p; \\ a \in \mathrm{QR}_p, b \notin \mathrm{QR}_p &\Rightarrow c \notin \mathrm{QR}_p. \end{aligned}$$

(b) Für $a \in \mathbb{Z}_p^*$ und a^{-1} in $\mathbb{Z}_p^*$, das multiplikative Inverse von a, gilt:

$$a \in \mathrm{QR}_p \quad \Leftrightarrow \quad a^{-1} \in \mathrm{QR}_p.$$

((a) folgt durch Anwendung von Exponentiationsregeln und Proposition 5.4.4; (b) folgt aus $a \cdot a^{-1} = 1 \in \mathrm{QR}_p$ und (a).)

Wenn das so einfach geht – kann man dann zu einer Quadratzahl a auch leicht eine Quadratwurzel b modulo p berechnen?

Erstaunlicherweise gibt es hier zwei sehr verschiedene Fälle. Wir betrachten das Beispiel $p = 19$. Die Quadrate in $\mathbb{Z}_p^*$ sind die neun Zahlen $1, 4, 5, 6, 7, 9, 11, 16, 17$. Wir berechnen $a^5 \bmod 19$ für diese Quadratzahlen und quadrieren das Resultat:

a	1	4	5	6	7	9	11	16	17
$b = a^5 \bmod 19$	1	17	9	5	11	16	7	4	6
$b^2 \bmod 19$	1	4	5	6	7	9	11	16	17

Wir sehen: $a^5 \bmod 19$ ist für jede der Quadratzahlen a eine Quadratwurzel. (Die andere ist dann natürlich $(19 - a^5) \bmod 19$.) Dies ist kein Zufall. Die Zahl 5 ergibt sich aus $p = 19$ als $\frac{1}{4}(p + 1)$. Diese Operation ist für alle Primzahlen möglich, die um 1 unter einem Vielfachen von 4 liegen.

Proposition 5.4.7
Wenn p eine Primzahl ist derart, dass $p + 1$ durch 4 teilbar ist, dann ist für jedes $a \in \mathrm{QR}_p$ die Zahl $a^{(p+1)/4} \bmod p$ eine Quadratwurzel von a.

Beweis:

$$((a^{(p+1)/4}) \bmod p)^2 \bmod p = (a^{(p+1)/2}) \bmod p = (((a^{(p-1)/2}) \bmod p) \cdot a) \bmod p = a,$$

wobei wir Proposition 5.4.4(b) benutzt haben. □

Mit anderen Worten: Für die Primzahlen $p \in \{3, 7, 11, 19, 23, 31, 43, 47, 59, \dots\}$ läuft das Ziehen von Quadratwurzeln auf eine schnelle Exponentiation mit Exponent $\frac{1}{4}(p + 1)$ hinaus. (Bitkomplexität $O((\log p)^3)$.)

Wie steht es mit den anderen Primzahlen $5, 13, 17, 29, 37, 41, 53, \dots$? Gibt es einen ähnlich effizienten Algorithmus zum Finden von Quadratwurzeln? Hier gibt es nochmals zwei Unterfälle. Wenn man leicht einen beliebigen nichtquadratischen Rest modulo p finden kann, dann kann man mit Hilfe eines deterministischen Algorithmus, der in Anhang C beschrieben ist, mit $O((\log p)^2)$ arithmetischen Operationen Quadratwurzeln finden. Dies ist der Fall, wenn $p \equiv 5 \pmod 8$, also für $5, 13, 29, 37, 53, \dots$, weil dann stets $2 \notin \mathrm{QR}_p$ gilt (Ergebnis der Zahlentheorie im Zusammenhang mit

dem „Quadratischen Reziprozitätsgesetz“). Es bleiben die Primzahlen p mit $p \equiv 1 \pmod 8$, beispielsweise $17, 41, 73, \ldots, 257, \ldots$. Interessanterweise kennt man für solche Primzahlen keinen *deterministischen* Algorithmus, der Quadratwurzeln modulo p berechnet und dazu Zeit $(\log p)^{O(1)}$ braucht. Wir betrachten hier einen ***randomisierten*** Algorithmus[19] für diese Aufgabe. Er funktioniert für alle Primzahlen $p \geq 3$; angesichts Prop. 5.4.7 wird man ihn aber nur verwenden, wenn $p \equiv 1 \pmod 4$ ist.

Idee: Gegeben ein $a \in \mathrm{QR}_p$, bezeichne eine Quadratwurzel von a mit $\diamond$. Die andere Quadratwurzel von a ist dann $-\diamond$. Rechne mit $\diamond$ (in $\mathbb{Z}_p$) wie mit einer Unbekannten. Es entstehen Ausdrücke $\alpha + \beta \cdot \diamond$, die man addieren, subtrahieren und multiplizieren kann. Höhere Potenzen von $\diamond$ entstehen nicht, wenn $\diamond^2$ durch a ersetzt wird. Es gelten folgende Regeln:

$$\begin{aligned} (\alpha + \beta \cdot \diamond) \pm (\gamma + \delta \cdot \diamond) &= (\alpha \pm \gamma) + (\beta \pm \delta) \cdot \diamond; \\ (\alpha + \beta \cdot \diamond) \cdot (\gamma + \delta \cdot \diamond) &= (\alpha\gamma + \beta\delta \cdot a) + (\alpha\delta + \beta\gamma) \cdot \diamond. \end{aligned} \tag{5.4.6}$$

Wenn man diese Operationen *implementieren* will, stellt man ein Element $\alpha + \beta \cdot \diamond$ als (α, β) dar und rechnet wie in (5.4.6). Mit den in (5.4.6) angegebenen Operationen bildet die Menge der Paare (a, b) in $\mathbb{Z}_p \times \mathbb{Z}_p$ einen Ring mit 1. Insbesondere besitzt die Multiplikation ein neutrales Element, nämlich $(1, 0)$, und die Multiplikation ist kommutativ und assoziativ (prüft man leicht nach). Daher kann man das schnelle Exponentiationsverfahren aus Algorithmus 5.1.19 anwenden, um „symbolisch“ die Potenz $(\alpha + \beta \cdot \diamond)^i$ in $\log i$ Runden mit jeweils höchstens 10 Multiplikationen modulo p zu berechnen.

Wesentlich ist die folgende Grundeigenschaft. Wenn man in der abstrakten Notation eine Gleichheit ausgerechnet hat, dann gilt sie für beide Quadratwurzeln von a:

Wenn sich mit den Regeln von (5.4.6) $(\alpha+\beta\cdot\diamond)^i$ zu $\gamma+\delta\cdot\diamond$ berechnet und wenn $w^2 \bmod p = a$ ist, dann gilt $(\alpha + \beta \cdot w)^i = \gamma + \delta \cdot w$.

(Man setzt w für $\diamond$ ein und verfolgt die symbolische Rechnung, aber nun in $\mathbb{Z}_p$.)

[19] Der Algorithmus wurde 1907 von dem italienischen Mathematiker Michele Cipolla (1880–1947) gefunden.

Algorithmus 5.4.8 ***Algorithmus von Cipolla: Quadratwurzeln***
Input: Primzahl $p \geq 3$, $a \in \{1, \ldots, p-1\}$ mit $a^{(p-1)/2} \bmod p = 1$.
Methode:

```
        //  1–4: Finde b mit b^2 = a oder b^2 - a ∉ QR_p:
1       repeat
2         Wähle b uniform zufällig aus {1,...,p-1};
3         if b^2 mod p = a then return {b,-b};  //  großes Glück gehabt!
4       until (b^2-a)^((p-1)/2) mod p = p-1;  //  b^2-a ist nichtquadratischer Rest
5       (c+d·⋄) ← (b+1·⋄)^((p-1)/2);
6             //  Schnelle Exponentiation mit Ausdrücken α+β·⋄, Regeln (5.4.6)
7       u ← d^(-1) mod p  //  erweiterter Euklidischer Algorithmus
8       return {u,-u}.
```

Korrektheit: w sei eine Quadratwurzel von a; die andere ist dann $-w$. Nach der obigen Vorüberlegung gilt (in $\mathbb{Z}_p$ gerechnet), weil sowohl $w^2 = a$ als auch $(-w)^2 = a$ gilt:

$$\begin{aligned} c + dw &= (b+w)^{(p-1)/2} \in \{1, -1\} \text{ und} \\ c - dw &= (b-w)^{(p-1)/2} \in \{1, -1\}. \end{aligned} \tag{5.4.7}$$

Wir addieren diese beiden Gleichungen und erhalten:

$$2 \cdot c = (b+w)^{(p-1)/2} + (b-w)^{(p-1)/2}. \tag{5.4.8}$$

Andererseits ist $(b^2 - a)^{(p-1)/2} = -1$ (wegen Zeile 4 im Algorithmus). Wir setzen $a = w^2$ ein und erhalten

$$-1 = (b^2 - w^2)^{(p-1)/2} = (b+w)^{(p-1)/2} \cdot (b-w)^{(p-1)/2}.$$

Wegen (5.4.7) muss einer der Faktoren gleich 1 und der andere gleich -1 sein:

$$\{(b+w)^{(p-1)/2}, (b-w)^{(p-1)/2}\} = \{1, -1\}.$$

Mit (5.4.8) folgt $2 \cdot c = 0$, und damit $c = 0$, weil in $\mathbb{Z}_p$ gilt, dass $2 = 1 + 1 \neq 0$ ist. Subtraktion der Gleichungen in (5.4.7) ergibt

$$2dw = (b+w)^{(p-1)/2} - (b-w)^{(p-1)/2}.$$

Die beiden Terme in der Differenz sind 1 und -1 (in irgendeiner Reihenfolge), also ist $2dw \in \{2, -2\}$, oder (wieder weil $2 \neq 0$):

$$dw \in \{1, -1\}, \text{ d.h.: } w \in \{d^{-1}, -d^{-1}\},$$

wobei das multiplikative Inverse in $\mathbb{Z}_p$ berechnet wird. Weil nach der Ausgangssituation w und $-w$ die Quadratwurzeln von a sind, sind auf jeden Fall $u = d^{-1}$ und $-u = -d^{-1}$ diese Wurzeln.

Rechenzeit: In der Laufzeit des Algorithmus gibt es drei entscheidende Komponenten: (i) Das Finden eines Elements b derart, dass b Quadratwurzel von a (sehr unwahrscheinlich) oder $b^2 - a \notin \mathrm{QR}_p$ ist; (ii) die Exponentiation zum Finden von $(b + 1 \cdot \diamond)^{(p-1)/2}$ (die höchstens $\log p$ Schleifendurchläufe mit je höchstens 10 modularen Multiplikationen erfordert); (iii) der Aufwand für den erweiterten Euklidischen Algorithmus. Teile (ii) und (iii) haben Kosten, die einer festen Anzahl modularer Exponentiationen entsprechen, also $O(\log p)$ Multiplikationen und Additionen. Für Teil (i) ist zu überlegen, was die Dichte

$$\frac{|G|}{p-1} = \frac{|\{b \in \mathbb{Z}_p^* \mid b^2 = a \ \vee \ b^2 - a \notin \mathrm{QR}_p\}|}{p-1}$$

der Menge G der „guten“ Elemente in $\mathbb{Z}_p^*$ ist.

Behauptung: $|G| \geq \frac{1}{2}(p+1)$.
Beweis der Behauptung: Die Elemente w und $-w$ sind verschiedene und jedenfalls gute Elemente. Es geht also um die Anzahl der guten Elemente in der Menge („*Nicht-Wurzeln*“)

$$NW := \mathbb{Z}_p^* - \{w, -w\}.$$

Wir definieren eine Funktion $f\colon NW \to \mathbb{Z}_p$ durch

$$f(b) := \frac{b+w}{b-w} \quad \text{(Arithmetik in } \mathbb{Z}_p\text{)}.$$

Diese Funktion ist wohldefiniert, da $b \neq w$, und sie nimmt den Wert 0 nicht an, da $b \neq -w$. Weiterhin wird der Wert 1 nicht angenommen (wenn $b + w = b - w$, wäre $2w = 0$, also $w = 0$), und der Wert -1 auch nicht (wenn $b + w = -(b - w)$, wäre $2b = 0$, also $b = 0$, was $b \in \mathbb{Z}_p^*$ widerspricht). Also gilt $f\colon NW \to \{2, \ldots, p-2\}$. Man sieht leicht, dass f injektiv ist:

$$\begin{aligned}\frac{b_1+w}{b_1-w} = \frac{b_2+w}{b_2-w} \quad &\Rightarrow \quad (b_1+w)(b_2-w) = (b_2+w)(b_1-w)\\ &\Rightarrow \quad b_1b_2 - wb_1 + wb_2 - w^2 = b_1b_2 + wb_1 - wb_2 - w^2\\ &\Rightarrow \quad 2w(b_2-b_1) = 0\\ &\Rightarrow \quad b_1 = b_2,\end{aligned}$$

wobei wir $2 \neq 0$ und $w \neq 0$ benutzt haben. Aus der Injektivität folgt, dass f eine Bijektion zwischen NW und $\{2,\ldots,p-2\}$ ist. In der letzteren Menge gibt es höchstens $(p-1)/2-1 = \frac{1}{2}(p-3)$ quadratische Reste (weil 1 ein quadratischer Rest ist). Wir haben:

$$b^2 - a \in \mathrm{QR}_p \quad \Leftrightarrow \quad \frac{b+w}{b-w} \in \mathrm{QR}_p.$$

(Das gilt, weil $b^2-a = (b+w)(b-w)$ ist, also $(b+w)/(b-w) = (b^2-a)/(b-w)^2$, und nach Korollar 5.4.6 die Menge der (nicht-)quadratischen Reste in $\mathbb{Z}_p$ unter Division mit quadratischen Resten (hier $(b-w)^2$) abgeschlossen ist.) Wegen dieser Eigenschaft von f ist

$$|G| = 2 + |\{u \in NW \mid f(u) \notin \mathrm{QR}_p\}| \geq 2 + (p-3) - \frac{1}{2}(p-3) = \frac{1}{2}(p+1).$$

Damit ist die Behauptung bewiesen. – Die Wahrscheinlichkeit, in Zeile 2 ein Element von $|G|$ zu wählen, ist also mindestens $(p+1)/(2(p-1)) > \frac{1}{2}$. Damit ist die erwartete Anzahl von Versuchen, bis ein geeignetes b gefunden wird, kleiner als 2. □

Wir fassen unsere Ergebnisse im folgenden Satz zusammen:

Satz 5.4.9
Für eine Primzahl p und einen quadratischen Rest a modulo p berechnet Algorithmus 5.4.8 die beiden Quadratwurzeln von a. Die erwartete Anzahl von Runden im ersten Teil ist $O(1)$; der erwartete Aufwand ist $O((\log p)^3)$ Bitoperationen.

Bemerkung: Für Zahlen der Form $N = p \cdot q$ für verschiedene Primzahlen p, q ist kein effizienter Algorithmus zur Ermittlung von Quadratwurzeln modulo N bekannt (auch kein randomisierter!), wenn nur N und die Quadratzahl a (modulo N), nicht aber die Faktoren p und q Teil der Eingabe sind. Diese Tatsache ist die Grundlage der Sicherheit des *Rabin-Kryptosystems*.

Dieses „Public-Key-Kryptosystem“ funktioniert wie folgt. Wir nehmen vereinfachend an, dass Nachrichten Bitstrings sind, die wir als natürliche Zahlen auffassen. Alice ist der Absender, Bob der Empfänger. Bob bereitet das System vor, indem er zwei verschiedene (große) Primzahlen p und q wählt, zum Beispiel mit einem Algorithmus wie 5.3.1. Er hält p und q geheim und stellt $N = p \cdot q$ als „öffentlichen Schlüssel“ zur Verfügung. Alice kennt also N. Wenn sie Bob eine Nachricht $x < N$ schicken möchte, berechnet sie den „Kryptotext“ $y = x^2 \bmod N$ und sendet ihn (als Bitstring) an Bob. Zum Entschlüsseln von y geht Bob wie folgt vor: Er benutzt einen Algorithmus zum Ziehen von Quadratwurzeln modulo einer Primzahl, um Zahlen $u < p$ mit $u^2 \equiv y \pmod{p}$ und $v < q$ mit $v^2 \equiv y \pmod{q}$ zu berechnen. Dann berechnet er mit Hilfe der effizienten Version des Chinesischen Restsatzes die vier Zahlen $x_1, x_2, x_3, x_4 < N$, die folgende Kongruenzen erfüllen:

$$\begin{aligned}
x_1 &\equiv u \pmod{p} \text{ und } x_1 \equiv v \pmod{q},\\
x_2 &\equiv u \pmod{p} \text{ und } x_2 \equiv -v \pmod{q},\\
x_3 &\equiv -u \pmod{p} \text{ und } x_3 \equiv v \pmod{q},\\
x_4 &\equiv -u \pmod{p} \text{ und } x_4 \equiv -v \pmod{q}.
\end{aligned}$$

Eine dieser vier Zahlen (Bitstrings) ist die Nachricht x, die Alice geschickt hat. Man geht davon aus, dass Bob aufgrund von eingebauten Redundanzen die richtige Nachricht aussuchen kann.

Das Rabin-Kryptosystem ist praktisch nicht besonders wichtig, aber es hat eine interessante theoretische Eigenschaft. Wenn ein Angreifer einen effizienten Algorithmus hat (also einen mit Rechenzeit polynomiell in $\log N$), der es ihm erlaubt, aus beliebigen Kryptotexten y den zugehörigen Klartext zu ermitteln, dann kann dieser Angreifer effizient die Faktoren p und q berechnen. Da die letzte Aufgabe nach heutigem Stand des Wissens für genügend große Zahlen p und q als nicht effizient lösbar gilt (es ist kein Polynomialzeitalgorithmus für dieses Problem bekannt), nimmt man an, dass es keinen effizienten Algorithmus gibt, der das Rabin-Kyptosystem brechen kann. (Details zu diesen Überlegungen: Vorlesung „Kryptographie“.)

A Beweise und Bemerkungen für Abschnitt 5.1

Bemerkung zu Fakt 5.1.2: Wenn man es genau nimmt, sind die Elemente von $\mathbb{Z}_m$ nicht Zahlen, sondern Äquivalenzklassen

$$[a] = \{qm + a \mid q \in \mathbb{Z}\}, \text{ für } 0 \leq a < m,$$

zur Äquivalenzrelation

$$x \equiv y \pmod{m}, \quad (\text{auch: } x \equiv y\ (m) \ \text{ oder } \ x \equiv_m y\ ),$$

die durch „m ist Teiler von $x - y$“ definiert ist (vgl. Fakt. 5.1.4). Die Operationen werden auf den Repräsentanten ausgeführt, zum Beispiel gilt in $\mathbb{Z}_{11}$, dass $[4] + [7] = [11] = [0]$ ist, also sind $[4]$ und $[7]$ zueinander invers. Allgemeiner gilt in jedem $\mathbb{Z}_m$ die Gleichheit $[a] + [m - a] = [m] = [0]$, d. h., $[m - a]$ ist das additive Inverse $-[a]$ zu $[a]$. Multiplikativ gilt in $\mathbb{Z}_m$, dass $[1]$ neutrales Element ist, und dass $[1] \cdot [1] = [1]$ und $[m - 1] \cdot [m - 1] = [m(m - 2) + 1] = [1]$, also $[1]$ und $-[1] = [m - 1]$ (triviale) Quadratwurzeln der 1 sind.

Es ist bequem, die eckigen Klammern wegzulassen und für $[i]$ einfach i zu schreiben. Man muss dann nur aufpassen, dass man Ausdrücke wie „-1“ richtig interpretiert, nämlich als das additive Inverse von 1 in $\mathbb{Z}_m$, also $[m - 1]$.

Zum *Beweis* von Lemma 5.1.3: Diese Aussagen lassen sich mathematisch besser in der Sprechweise der Äquivalenzklassen verstehen. Dazu muss man zunächst ordentlich definieren: $[x] + [y] := [x + y]$ und $[x] \cdot [y] := [x \cdot y]$. (Die Summe von $[x]$ und $[y]$ in $\mathbb{Z}_m$ ist die Äquivalenzklasse von $x + y$; das Produkt von $[x]$ und $[y]$ in $\mathbb{Z}_m$ ist die Äquivalenzklasse von $x \cdot y$.) Damit diese Definition funktioniert, muss man zunächst überprüfen, dass das Ergebnis nicht von den irgendwie gewählten Repräsentanten x und y abhängt, sondern tatsächlich nur von den Klassen $[x]$ und $[y]$. (Wenn wir zum Beispiel $m = 7$ nehmen, dann sollte $[10 \cdot 17]$ dasselbe sein wie $[(-4) \cdot 3]$. Dies ist der Fall: $170 \equiv 2 \equiv -12 \pmod{7}$.) Diese Eigenschaft nennt man die *Wohldefiniertheit*:

$$x \equiv x' \wedge y \equiv y' \pmod{m} \ \Rightarrow \ x + y \equiv x' + y' \pmod{m}.$$

Diese Überprüfung ist ganz leicht: Wenn $x' = x + km$ und $y' = y + \ell m$, dann ist $x' + y' = x + y + (k + \ell)m$ und $x' \cdot y' = x \cdot y + (ky + x\ell + k\ell m)m$. Unser Lemma ist nun im Wesentlichen eine spezielle Formulierung dieser Wohldefiniertheit: Man kann beliebige Repräsentanten x und y aus $\mathbb{Z}$ oder die speziellen Repräsentanten

$x' = (x \bmod m)$ und $y' = (y \bmod m)$ aus $[m]$ benutzen. Wegen der allgemeinen Wohldefiniertheit gilt dann $[x+y] = [x'+y']$ sowie $[x \cdot y] = [x' \cdot y']$, und das heißt $x + y \equiv x' + y' \pmod m$ sowie $x \cdot y \equiv x' \cdot y' \pmod m$. Mit dem folgenden Fakt 5.1.4 ergibt sich

(i) $(x+y) \bmod m = (x'+y') \bmod m$,

(ii) $(x \cdot y) \bmod m = (x' \cdot y') \bmod m$,

wie gewünscht. □

Beweis von Fakt 5.1.4: Wenn $x \bmod m = y \bmod m$ gilt, gibt es Quotienten q und q' mit $x = qm + r$ und $y = q'm + r$, für den gemeinsamen Rest r. Damit gilt $x - y = (q - q')m$. Wenn umgekehrt $x - y = mz$ für eine ganze Zahl z und $x = qm + r$ und $y = q'm + r'$ für Quotienten q, q' und Reste r, r', dann ist $r - r' = (x-y) + (q'-q)m = (z + q' - q)m$ durch m teilbar, und $|r - r'| < m$. Daher muss $r - r' = 0$ gelten. □

Beweis von Fakt 5.1.8:

Die Korrektheit folgt aus den Vorüberlegungen. Wir diskutieren also nur die Rechenzeit. Wenn $|x| < |y|$, hat der erste Schleifendurchlauf nur den Effekt, die beiden Zahlen zu vertauschen. Wir ignorieren diesen trivialen Schleifendurchlauf. Nachher nimmt die Zahl b in `b` in jeder Runde strikt ab, also terminiert der Algorithmus. Um einzusehen, dass er sogar sehr schnell terminiert, bemerken wir Folgendes. Betrachte den Beginn eines Schleifendurchlaufs. Der Inhalt von `a` sei a, der Inhalt von `b` sei b, mit $a \geq b > 0$. Nach einem Schleifendurchlauf enthält `a` den Wert $a' = b$ und `b` den Wert $b' = a \bmod b$. Falls $b' = 0$, endet der Algorithmus. Sonst wird noch ein Durchlauf ausgeführt, an dessen Ende `a` den Wert $b' = a \bmod b$ enthält. Wir behaupten: $b' < \frac{1}{2}a$. Um dies zu beweisen, betrachten wir zwei Fälle: Wenn $b > \frac{1}{2}a$ ist, gilt $b' = a \bmod b = a - b < \frac{1}{2}a$. Wenn $b \leq \frac{1}{2}a$ ist, gilt $b' = a \bmod b < b \leq \frac{1}{2}a$. – Also wird der Wert in `a` in jeweils zwei Durchläufen mindestens halbiert. Nach dem ersten Schleifendurchlauf enthält `a` den Wert $\min\{x, y\}$. Also kann es höchstens $\log(\min\{x, y\})$ weitere Schleifendurchläufe geben; danach ist der Wert in `b` auf jeden Fall 0. □

Bemerkungen zu Lemma 5.1.9: In (b), also auch im Spezialfall (a), ergibt sich die Existenz von s und t direkt aus dem erweiterten Euklidischen Algorithmus. Interessant ist noch die folgende Umkehrung von (a): Wenn wir $1 = sx + ty$ schreiben

können, für ganze Zahlen s, t, dann sind x und y teilerfremd. Der Grund ist einfach: Wenn $c \mid x$ und $c \mid y$ gilt, dann folgt $c \mid sx + ty$, also $c \mid 1$. Daher ist $c \in \{-1, 1\}$.

Beweis von Fakt 5.1.15: (i) Assoziativgesetz und Kommutativgesetz für Multiplikation modulo m gelten in $\mathbb{Z}_m$ allgemein. (ii) 1 ist neutrales Element sogar in $\mathbb{Z}_m$. (iii) Wir zeigen, dass $\mathbb{Z}_m^*$ unter Multiplikation abgeschlossen ist. Seien also $a, b \in \mathbb{Z}_m^*$, d. h. $\mathrm{ggT}(a, m) = \mathrm{ggT}(b, m) = 1$. Nach Lemma 5.1.9(a) kann man $1 = xa + um = yb + vm$ schreiben, für ganze Zahlen x, u, y, v. Dies ergibt

$$1 = (xa + um)(yb + vm) = (xy)(ab) + (xav + uyb + umv)m.$$

Daraus folgt, dass ab und m teilerfremd sind, also $ab \bmod m \in \mathbb{Z}_m^*$. (iv) Nach Lemma 5.1.13 hat jedes Element von $\mathbb{Z}_m^*$ ein multiplikatives Inverses $b \in \mathbb{Z}_m$. Weil natürlich a auch das Inverse von b ist, folgt nach Lemma 5.1.13 auch, dass $b \in \mathbb{Z}_m^*$ ist. □

Beweis von Prop. 5.1.17: Die zweite Äquivalenz ist klar: Der Ring $\mathbb{Z}_m$ ist nach Definition ein Körper genau dann wenn jedes Element von $\mathbb{Z}_m - \{0\}$ ein multiplikatives Inverses besitzt.

Erste Äquivalenz:

„⇒“: Sei m eine Primzahl. Dann ist jedes $a \in \{1, \ldots, m-1\} = \mathbb{Z}_m - \{0\}$ zu m teilerfremd; nach der Definition folgt $\mathbb{Z}_m^* = \mathbb{Z}_m - \{0\}$.

„⇐“: Sei m eine zusammengesetzte Zahl, etwa durch k mit $2 \leq k < m$ teilbar. Dann ist $k = \mathrm{ggT}(k, m) > 1$, also hat k nach Lemma 5.1.13 kein multiplikatives Inverses modulo m. (Man sieht auch direkt, dass $kb \bmod m = kb - qm$ durch k teilbar ist, also für kein b gleich 1 sein kann.) □

B Mehr über Carmichaelzahlen

Wie kann man einsehen, dass $561 = 3 \cdot 11 \cdot 17$, $1105 = 5 \cdot 13 \cdot 17$ und $1729 = 7 \cdot 13 \cdot 19$ Carmichaelzahlen sind? In diesem Abschnitt stellen wir (nur für Interessierte) ein einfaches Kriterium bereit, das es erlaubt, Zahlen auf die Eigenschaft „Carmichaelzahl“ zu testen, *wenn ihre Primfaktorzerlegung gegeben ist*. Das folgende Lemma (mitsamt Beweis) ist eine leichte Verallgemeinerung von Lemma 5.2.7.

Lemma B.0.1
Es sei N eine Carmichaelzahl und p ein Primfaktor von N. Wir schreiben $N = p^\ell \cdot M$ für eine zu p teilerfremde Zahl $M \geq 1$. Dann gilt $\ell = 1$. (Man sagt: N ist *quadratfrei*; kein Primfaktor kommt in N mit einer Vielfachheit größer als 1 vor.)

Beweis: Indirekt. Angenommen, $\ell \geq 2$. Wir geben eine Zahl $a \in \mathbb{Z}_N^*$ an, die $a^{N-1} \bmod N \neq 1$ erfüllt. Dazu benutzen wir den Chinesischen Restsatz. Weil p^ℓ und M teilerfremd sind, gibt es nach dem CRS eine Zahl $a \in \mathbb{Z}_N$ mit $a \bmod p^\ell = p^{\ell-1} + 1$ und $a \bmod M = 1$. Weil a teilerfremd zu p^ℓ und teilerfremd zu M ist, gilt $a \in \mathbb{Z}_N^*$, nach Prop. 5.1.22.

Wir zeigen gleich, dass $a^{N-1} \bmod p^\ell \neq 1$ gilt. Nach dem Chinesischen Restsatz folgt daraus, dass $a^{N-1} \bmod N \neq 1$ gilt, wie gewünscht. Wir rechnen modulo p^ℓ, mit der binomischen Formel:

$$\begin{aligned} a^{N-1} &\equiv (p^{\ell-1}+1)^{N-1} \\ &\equiv \sum_{0 \leq j \leq N-1} \binom{N-1}{j} (p^{\ell-1})^j \\ &\equiv 1 + (N-1) \cdot p^{\ell-1} \pmod{p^\ell}. \end{aligned} \tag{B.0.9}$$

(Die letzte Äquivalenz ergibt sich daraus, dass für $j \geq 2$ gilt, dass $(\ell-1)j \geq \ell$ ist, also der Faktor $(p^{\ell-1})^j = p^{(\ell-1)j}$ durch p^ℓ teilbar ist, der entsprechende Summand also modulo p^ℓ wegfällt.) Weil $N-1 = p^\ell \cdot M - 1$ nicht durch p teilbar ist, ist $(N-1) \cdot p^{\ell-1}$ nicht durch p^ℓ teilbar. Damit folgt aus (B.0.9), dass $a^{N-1} \not\equiv 1 \pmod{p^\ell}$ ist, also $a^{N-1} \bmod p^\ell \neq 1$. □

Das Lemma bedeutet also, dass in der Primfaktorzerlegung einer Carmichaelzahl kein Primfaktor mehrfach vorkommt.

Lemma B.0.2
Es sei N eine Carmichaelzahl und p ein Primfaktor von N. Dann gilt $p - 1 \mid N - 1$.

Beweis: Hier benötigen wir etwas Gruppentheorie und Zahlentheorie. Wir schreiben $N = p \cdot M$ für $M = N/p > 1$. Nach Lemma B.0.1 sind dann M und p teilerfremd. Nach dem Chinesischen Restsatz sind $\mathbb{Z}_N^*$ und $\mathbb{Z}_p^* \times \mathbb{Z}_M^*$ isomorph. Weil p eine Primzahl ist, ist (nach einem einfachen Satz der Zahlentheorie) $\mathbb{Z}_p^* = \{1, \ldots, p-1\}$ eine *zyklische* Gruppe. Das bedeutet, dass es in $\mathbb{Z}_p^*$ ein Element g (ein „erzeugendes Element“) mit

$\mathbb{Z}_p^* = \{1, g, g^2, \ldots, g^{p-2}\}$ gibt (Rechnung in $\mathbb{Z}_p^*$, also modulo p). Für dieses Element gilt:

$$g^j = 1 \;\Leftrightarrow\; p-1 \mid j, \text{ für alle ganzen Zahlen } j. \qquad (*)$$

Der Chinesische Restsatz liefert uns ein Element $a \in \mathbb{Z}_N^*$ mit $a \bmod p = g$ und $a \bmod M = 1$. Weil N Carmichaelzahl ist, gilt $a^{N-1} \bmod N = 1$, also nach Wahl von a und dem Chinesischen Restsatz $g^{N-1} \bmod p = a^{N-1} \bmod p = 1$. Nach $(*)$ folgt $p-1 \mid N-1$. □

Man sieht nun ganz leicht, dass Produkte von zwei Primzahlen nie Carmichaelzahlen sein können.

Lemma B.0.3
(a) Alle Primfaktoren einer Carmichaelzahl N sind kleiner als $\sqrt{N}$.
(b) Jede Carmichaelzahl hat mindestens drei Primfaktoren.

Beweis: (b) folgt direkt aus (a). Der Beweis von (a) ist indirekt. Annahme: N ist Carmichaelzahl und N hat einen Primfaktor $p > \sqrt{N}$. (Der Fall $p = \sqrt{N}$ ist nach Lemma B.0.1 ausgeschlossen.) Dann ist $p > N/p$. Offensichtlich gilt $p-1 \mid (p-1)(N/p)$, also $p-1 \mid N - N/p$. Nach Lemma B.0.2 gilt $p-1 \mid N-1$. Es folgt $p-1 \mid N/p - 1$, also $p-1 \leq N/p - 1$, ein Widerspruch. □

Die Situation bei der Carmichaelzahl $561 = 3 \cdot 11 \cdot 17$ ist also typisch: Es gibt mindestens drei verschiedene Primfaktoren, hier 3, 11 und 17, und es gilt, dass $2 = 3-1$, $10 = 11-1$ und $16 = 17-1$ Teiler von $560 = 561-1$ sind. (Man prüfe die entsprechenden Gleichungen auch für $1105 = 5 \cdot 13 \cdot 17$ und $1729 = 7 \cdot 13 \cdot 19$ nach!) Es stellt sich heraus, dass diese Bedingung Carmichaelzahlen vollständig charakterisiert.

Proposition B.0.4
N ist eine Carmichaelzahl genau dann wenn N mindestens drei Primfaktoren hat, die alle verschieden sind, und wenn für jeden Primfaktor p von N gilt: $p-1 \mid N-1$.

Beweis: Dass jede Carmichaelzahl die genannten Eigenschaften hat, haben wir schon gesehen. Sei nun $N = p_1 \ldots p_r$ eine Zahl mit verschiedenen Primfaktoren $p_1, \ldots, p_r$, so dass $p_i - 1 \mid N-1$ für $1 \leq i \leq r$. Es sei $a \in \mathbb{Z}_N^*$ beliebig. Dann gilt für jedes i:

$$a^{N-1} \bmod p_i = (a^{p_i-1} \bmod p_i)^{(N-1)/(p_i-1)} \bmod p_i.$$

Nach dem kleinen Satz von Fermat gilt $a^{p_i-1} \bmod p_i = 1$, und damit $a^{N-1} \bmod p_i = 1$. Die verallgemeinerte Version des Chinesischen Restsatzes für r teilerfremde Faktoren besagt, dass $\mathbb{Z}_N$ und $\mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_r}$ isomorph sind, über die Funktion $\Phi(b) = (b \bmod p_1, \ldots, b \bmod p_r)$, die 1 auf $(1, \ldots, 1)$ abbildet. Es folgt $a^{N-1} \bmod N = 1$. □

Proposition B.0.4 führt dazu, dass für Zahlen N *mit gegebener Primfaktorzerlegung* effizient entschieden werden kann, ob N Carmichaelzahl ist. Wir müssen nur für jeden Primfaktor p von N prüfen, ob p in der Primfaktorzerlegung nur einfach vorkommt und ob $p-1$ die Zahl $N-1$ teilt. Aber Achtung: Wenn die Primfaktoren von N nicht bekannt sind, funktioniert dieser Test nicht.

In diesem Fall können wir wie folgt vorgehen, für eine gegebene ungerade Zahl $N \geq 3$ und eine Wiederholungszahl ℓ:

- Für zufällig gewählte Zahlen $a_1, a_2, \ldots$ aus $\{1, \ldots, N-1\}$ tue Folgendes:
  - Teste, ob $\mathrm{ggT}(a_j, N) = 1$ gilt. Falls nein: Notiere, dass N zusammengesetzt ist. Falls ja:
  - Lasse den Miller-Rabin-Test auf N mit a_j ablaufen, mit folgender Modifikation: Alle Zahlen $b_i = a_j^{u \cdot 2^i} \bmod N$, $i = 0, 1, \ldots, k$, werden berechnet, und es wird am Ende auch geprüft, ob a_j F-Zeuge ist.
  - Falls a_j F-Zeuge ist, wird Ausgabe „2“ ausgegeben und angehalten.
  - Falls a_j MR-Zeuge ist, wird notiert, dass N zusammengesetzt ist.
- Wenn nie mit Ausgabe „2“ abgebrochen wird, wird die Schleife so lange wiederholt, bis ℓ viele a_j mit $\mathrm{ggT}(a_j, N) = 1$ untersucht worden sind.
- Wenn nie festgestellt wurde, dass N zusammengesetzt ist, ist die Ausgabe „0“.
- Wenn festgestellt wurde, dass N zusammengesetzt ist, aber in jeder Berechnung mit einem a_j, das teilerfremd zu N ist, das Ergebnis $b_k = a_j^{N-1} \bmod N = 1$ ist (also a_j kein F-Zeuge ist), ist die Ausgabe „1“.

Dieser Algorithmus verhält sich wie folgt:

(a) Wenn N eine Primzahl ist, ist die Ausgabe 0.

(b) Wenn N eine Carmichaelzahl ist, dann ist die Ausgabe 0 oder 1, und es gilt $\mathbf{Pr}(\text{Ausgabe ist } \neq 1) \leq 4^{-\ell}$.

(c) Wenn N eine zusammengesetzte Zahl, aber keine Carmichael-Zahl ist, sind Ausgaben 0, 1 und 2 möglich, und es gilt $\mathbf{Pr}(\text{Ausgabe ist } \neq 2) \leq 2^{-\ell}$.

Zu (a): Ausgabe 1 wird nur erzeugt, wenn ein MR-Zeuge gefunden wurde. Ausgabe 2 wird nur erzeugt, wenn ein F-Zeuge gefunden wurde. In beiden Fällen ist N keine Primzahl.
(b) Ausgabe 2 kann nur für Zahlen vorkommen, die zusammengesetzt sind, aber keine Carmichaelzahl. Die Ausgabe 0 entsteht nur dann, wenn bei den ℓ Versuchen mit a_j, die teilerfremd zu N sind, niemals ein MR-Zeuge gefunden wird. Wir hatten in Abschnitt 5.2.5 gesagt, dass die Wahrscheinlichkeit hierfür in einem Versuch höchstens $\frac{1}{4}$ ist, für alle ℓ Versuche zusammen also $4^{-\ell}$.
(c) Ausgabe 0 oder 1 kann nur entstehen, wenn jedes der getesteten a_j mit $\mathrm{ggT}(a_j, N) = 1$ ein F-Lügner ist. Nach Satz 5.2.4 ist dies für jedes neu gewählte a_j mit Wahrscheinlichkeit höchstens $\frac{1}{2}$ der Fall. Die Wahrscheinlichkeit, dass in ℓ Versuchen jedesmal ein F-Lügner gewählt wird, ist höchstens $2^{-\ell}$.

Dieser Test zur Klassifizierung von Zahlen als Primzahlen, Carmichaelzahlen und sonstige zusammengesetzte Zahlen ist also ein Monte-Carlo-Algorithmus mit allgemeiner Ausgabe, wie in Definition 3.2.4 betrachtet.

Ein effizienter deterministischer Test für die Eigenschaft „N ist Carmichaelzahl“ ist nicht bekannt.

C Quadratwurzeln modulo Primzahl p, alternativer Algorithmus

Wir betrachten hier nur den Fall $p \equiv 1 \pmod 4$. Nach Prop. 5.4.7 ist das Problem des Quadratwurzelziehens für die anderen Primzahlen p leicht deterministisch zu lösen.[20]

Algorithmus C.0.1 ***Quadratwurzeln, randomisiert***
Input: Primzahl p mit $p-1$ durch 4 teilbar;
$a \in \{1, \ldots, p-1\}$ mit $a^{(p-1)/2} \bmod p = 1$.
Methode:
// Finde $b \in \mathbb{Z}_p^* - \mathrm{QR}_p$:
1 **repeat**
2 $b \leftarrow$ Zufallszahl in $\{2, \ldots, p-2\}$
3 **until** $b^{(p-1)/2} \bmod p = p-1$;
4 Schreibe $\frac{1}{2}(p-1) = u \cdot 2^k$, u ungerade, $k \geq 1$;
// Berechne $s_0, \ldots, s_k$ gerade mit $a^{u \cdot 2^{k-i}} \cdot b^{s_i} \bmod p = 1$:
5 $\mathtt{s} \leftarrow 0$;
6 **for** $\mathtt{i}$ **from** 1 **to** k **do**
7 **if** $a^{u \cdot 2^{k-\mathtt{i}}} \cdot b^{\mathtt{s}/2} \bmod p = 1$
8 **then** $\mathtt{s} \leftarrow \mathtt{s}/2$
9 **else** $\mathtt{s} \leftarrow \mathtt{s}/2 + (p-1)/2$;
10 **return** $a^{(u+1)/2} \cdot b^{\mathtt{s}/2} \bmod p$.

Bemerkung: Wenn $p \equiv 5 \pmod 8$, dann ist 2 ein nichtquadratischer Rest modulo p (Zahlentheorie, quadratisches Reziprozitätsgesetz). Man kann dann Zeilen 1–3 durch $b \leftarrow 2$ ersetzen und erhält einen deterministischen Algorithmus. Für $p \equiv 1 \pmod 8$ ist keine einfache, direkte Methode zum Finden eines nichtquadratischen Restes bekannt.

Erklärung, Korrektheitsbeweis, Zeitanalyse: Wir setzen voraus, dass ein vorab durchgeführter Quadratzahltest (Algorithmus 5.4.5) ergeben hat, dass

$$a^{(p-1)/2} \bmod p = 1 \tag{C.0.10}$$

[20] Der hier angegebene Algorithmus wurde 1891 von Alberto Tonelli (1849–1920), einem italienischen Mathematiker, gefunden. Daniel Shanks hat ihn 1972 erneut gefunden. Er wird daher oft als Algorithmus von Tonelli und Shanks bezeichnet.

ist und daher a in $\mathbb{Z}_p$ eine Quadratwurzel besitzt. In Zeilen 1–3 wird mit randomisierter Suche eine Zahl $b \in \mathbb{Z}_p^*$ gesucht, die kein quadratischer Rest ist. Weil $(p-1)/2$ gerade ist, ist $(-1)^{(p-1)/2} \bmod p = 1$, also -1 ein quadratischer Rest; wir müssen also 1 und $p-1$ gar nicht testen. Unter den $p-3$ Zahlen in $\{2, \ldots, p-2\}$ sind genau $\frac{1}{2}(p-1)$ viele Nichtquadrate; das ist mehr als die Hälfte, also wird nach einer erwarteten Rundenzahl von < 2 ein passendes b gefunden. (Da der Rest des Algorithmus deterministisch ist, handelt es sich insgesamt um einen lupenreinen Las-Vegas-Algorithmus.)

Bemerkung: In Kapitel 4.1 haben wir ein randomisiertes Suchverfahren betrachtet, das in einer Menge A mit einer nichtleeren Teilmenge B, die nur indirekt, durch einen Test „ist $x \in B$?“ gegeben ist, ein Element von B sucht.

Man beachte, dass das in Zeilen 1–3 gelöste Problem, ein Element der Menge $\mathbb{Z}_p^* - \mathrm{QR}_p$ zu finden, genau dieser Aufgabenstellung entspricht:

- A ist $\mathbb{Z}_p^*$;
- B ist $\mathbb{Z}_p^* - \mathrm{QR}_p$;
- der Test „ist $b \in \mathbb{Z}_p^* - \mathrm{QR}_p$?“ wird in Zeile 3 durchgeführt (Eulersches Kriterium).

Die Menge B hat Dichte etwas größer als $\frac{1}{2}$ in $\mathbb{Z}_p^*$. Man beachte, dass tatsächlich, wie in Kapitel 4 abstrakt angenommen, keine explizite Darstellung von $\mathbb{Z}_p^* - \mathrm{QR}_p$ bekannt ist.

Zu Zeile 4: Weil $p-1$ durch 4 teilbar ist, ist $\frac{1}{2}(p-1)$ gerade, hat also eine Darstellung wie angegeben. – Startend in Zeile 5 berechnen wir in der Variablen `s` eine Folge $s_0, s_1, \ldots, s_k$ mit folgender Eigenschaft:

$$a^{u \cdot 2^{k-i}} \cdot b^{s_i} \bmod p = 1 \quad \text{und} \quad s_i \text{ ist gerade} \quad . \tag{C.0.11}$$

Die Eigenschaft (C.0.11) beweisen wir durch Induktion über die Schleifendurchläufe $i = 0, 1, \ldots, k$.
$i = 0$: $s_0 = 0$ ist gerade und $a^{u \cdot 2^k} \cdot b^0 \bmod p = a^{(p-1)/2} \bmod p = 1$, wegen (C.0.10).
$0 \le i < k$, „$i \to i+1$“: Nach Induktionsvoraussetzung gilt (C.0.11) für i. Wir betrachten Schleifendurchlauf $i+1$. In Zeile 7 wird berechnet:

$$f := a^{u \cdot 2^{k-(i+1)}} \cdot b^{s_i/2} \bmod p.$$

Die Zahl f ist eine Quadratwurzel von 1, denn $f^2 \bmod p = a^{u\cdot 2^{k-i}} \cdot b^{s_i} \bmod p = 1$ (nach (C.0.11) für i). Daher gilt $f \in \{1, -1\}$. Diese beiden Fälle werden in Zeilen 7–9 betrachtet.

1. Fall: $f = 1$. – Dann wird $s_{i+1} = s_i/2$, und $a^{u\cdot 2^{k-(i+1)}} \cdot b^{s_{i+1}} \bmod p = f = 1$.

2. Fall: $f = -1$. – Dann wird $s_{i+1} = s_i/2 + (p-1)/2$, und

$$a^{u\cdot 2^{k-(i+1)}} \cdot b^{s_{i+1}} \bmod p = f \cdot b^{(p-1)/2} \bmod p = (-1)\cdot(-1) \bmod p = 1.$$

(Wir benutzen hier, dass b kein quadratischer Rest ist, also $b^{(p-1)/2} \bmod p = -1$ ist.)

Es fehlt nur noch der Nachweis, dass s_{i+1} gerade ist. Annahme: s_{i+1} ungerade. Dann gilt:

$$a^{u\cdot 2^{k-(i+1)}} \cdot b^{s_{i+1}} \bmod p = 1.$$

Dabei ist die rechte Seite (die Zahl 1) ein quadratischer Rest, ebenso wie $a^{u\cdot 2^{k-(i+1)}} \bmod p$. Der Faktor $b^{s_{i+1}} \bmod p$ hingegen ist als ungerade Potenz von $b \notin \mathrm{QR}_p$ nicht in QR_p. Dies ist ein Widerspruch zu Korollar 5.4.6.

In Zeile 10 des Algorithmus wird die Zahl $t = a^{(u+1)/2} \cdot b^{s_k/2} \bmod p$ als Ergebnis ausgegeben. Wir berechnen:

$$t^2 \bmod p = a^{u+1} \cdot b^{s_k} \bmod p = a \cdot (a^u \cdot b^{s_k}) \bmod p = a,$$

wobei wir (C.0.11) (für $i = k$) angewendet haben. Daher ist t eine Quadratwurzel von a.

Laufzeit: Es werden maximal $\log p$ Runden durchgeführt; in jeder Runde ist eine schnelle Exponentiation mit $O(\log p)$ Multiplikationen modulo p durchzuführen. Die Anzahl der Multiplikationen und Divisionen ist also $O((\log p)^2)$, die Anzahl der Bitoperationen $O((\log p)^4)$.

∗ Abschnitte 6.6 und 6.7 sind **im SS 2021 nicht prüfungsrelevant**.

Das Pflichtprogramm endet auf Seite 13.

6 Algebraische Methoden

6.1 Der Satz von Schwartz-Zippel

In diesem Abschnitt betrachten wir Polynome mit mehreren Variablen $X_1, \ldots, X_n$ über einem Körper K. Ein solches Polynom f kann man als endliche Summe von Termen der Form

$$a_{\ell_1,\ldots,\ell_n} \cdot X_1^{\ell_1} X_2^{\ell_2} \cdots X_n^{\ell_n}$$

schreiben, wobei $a_{\ell_1,\ldots,\ell_n}$ ein Element von K ist, und in verschiedenen Termen die Exponentenfolge $\ell_1, \ldots, \ell_n$ verschieden ist. Der *Grad* eines solchen Terms ist $\ell_1 + \cdots + \ell_n$, der Grad $\deg(f)$ des Polynoms f ist das Maximum der Grade seiner Terme. Die Polynome von Grad 0 sind die Terme $a = aX_1^0 X_2^0 \cdots X_n^0$ mit $a \in K - \{0\}$. Das Nullpolynom hat keinen Term, sein Grad ist $-\infty$.

Beispiele für Polynome über dem Körper $K = \mathbb{Z}_5$:

$$X_1^2 X_2^2 + 4X_1 X_4^3 + 2X_2^3 X_3^3, \quad X_1 X_2 + 2X_2 + X_3, \quad X_1^3 + 3X_1^2 X_2, \quad 3, \quad 0$$

mit den Graden 6, 2, 3, 0 und $-\infty$. Polynome kann man addieren, subtrahieren und multiplizieren, und vereinfachen, indem man Terme, die zu derselben Exponentenfolge $\ell_1, \ldots, \ell_n$ gehören, zusammenfasst.

Ein Vektor $(a_1, \ldots, a_n) \in K^n$ heißt eine Nullstelle von f, wenn sich beim Einsetzen von a_i für X_i in f und Ausrechnen der Wert $f(a_1, \ldots, a_n) = 0$ ergibt. Beispielsweise ist $(1, 3)$ eine Nullstelle von $X_1^3 + 3X_1^2 X_2$ (über $K = \mathbb{Z}_5$).

Folgendes ist leicht zu beweisen (und zumindest für den Körper der reellen Zahlen aus der Schule bekannt, siehe auch Fakt 5.4.2(a)):

Lemma 6.1
Wenn f ein Polynom mit einer Variablen $X = X_1$ über einem Körper K ist, mit Grad $d \geq 0$, so besitzt f höchstens d Nullstellen.

Wir können dies auf Polynome mit n Variablen verallgemeinern:

Satz 6.2 ***Schwartz-Zippel***
Wenn f ein Polynom mit n Variablen über einem Körper K ist, mit Grad $d \geq 0$, und $S \subseteq K$ eine endliche Menge ist, dann besitzt f in S^n höchstens $d|S|^{n-1}$ Nullstellen.

Diese Aussage kann nicht verbessert werden: Betrachte den Körper $K = \mathbb{Z}_p$ für eine Primzahl p. Seien $1 \leq d \leq s \leq p$. Setze $S = \{0, \ldots, s-1\}$. Das Polynom $f = (X_1 - 0)(X_1 - 1)(X_1 - 2) \cdots (X_1 - (d-1))$ (mit n möglichen Variablen $X_1, \ldots, X_n$, von denen aber $X_2, \ldots, X_n$ in f nicht vorkommen) hat in S^n die ds^{n-1} Nullstellen $(i, a_2, \ldots, a_n)$, mit $0 \leq i < d$ und $a_2, \ldots, a_n \in S$ beliebig.

Beweis von Satz 6.2: Induktion über $d \geq 0$. Der Induktionsanfang $d = 0$ ist trivial, weil dann $f(X_1, \ldots, X_n)$ ein konstantes Polynom, aber nicht das Nullpolynom ist, also überhaupt keine Nullstelle hat.

Sei also jetzt $d \geq 1$, und sei als Induktionsvoraussetzung angenommen, dass die Aussage für Polynome mit Graden zwischen 0 und $d-1$ richtig ist.

Weil $d \geq 1$, enthält f mindestens eine Variable. Wir nehmen an, dass X_1 in f vorkommt. (Sonst benennt man die Variablen passend um.) Wir wählen $\delta \geq 1$ so, dass X_1^δ die höchste Potenz ist, mit der X_1 in f vorkommt, und klammern X_1^δ aus allen Termen aus, in denen dieser Faktor vorkommt. Dies liefert eine Zerlegung

$$f(X_1, \ldots, X_n) = X_1^\delta \cdot q(X_2, \ldots, X_n) + r(X_1, \ldots, X_n),$$

wobei q nicht das Nullpolynom ist, in q die Variable X_1 nicht vorkommt, der Grad von $q(X_2, \ldots, X_n)$ maximal $d - \delta < d$ ist, und in $r(X_1, \ldots, X_n)$ der Faktor X_1 nur mit Exponenten $< \delta$ auftaucht.

Wir definieren

$$A := \{(a_1, \ldots, a_n) \in S^n \mid q(a_2, \ldots, a_n) = 0\}$$

und

$$B := \{(a_1, \ldots, a_n) \in S^n \mid q(a_2, \ldots, a_n) \neq 0 \wedge f(a_1, \ldots, a_n) = 0\}.$$

Man sieht sofort, dass

$$f(a_1, \ldots, a_n) = 0 \;\Rightarrow\; (a_1, \ldots, a_n) \in A \cup B.$$

Es genügt also zu zeigen, dass $|A \cup B| \leq |A| + |B| \leq d|S|^{n-1}$ ist.

Dazu beobachten wir: Nach Induktionsvoraussetzung ist

$$|A| \leq |S| \cdot (d - \delta)|S|^{n-2} = (d - \delta)|S|^{n-1}.$$

(Die erste Komponente a_1 ist in S frei wählbar; für $(a_2, \ldots, a_n)$ benutzt man die Induktionsvoraussetzung.) Die Zahl $|B|$ schätzen wir folgendermaßen ab: Für jeden Vektor

$(a_2, \ldots, a_n) \in S^{n-1}$ mit $q(a_2, \ldots, a_n) \neq 0$ ist $f(X_1, a_2, \ldots, a_n) = q(a_2, \ldots, a_n)X_1^\delta + r(X_1, a_2, \ldots, a_n)$ ein Polynom in der einen Variablen X_1, vom Grad genau $\delta \geq 1$. Insbesondere ist dieses Polynom nicht das Nullpolynom, es hat also (nach Lemma 6.1) maximal δ Nullstellen in K. Wir summieren über alle $(a_2, \ldots, a_n) \in S^{n-1}$ und erhalten, dass $|B| \leq |S|^{n-1} \cdot \delta$ gilt. Wenn wir die Abschätzungen für A und B addieren, folgt $|A| + |B| \leq d|S|^{n-1}$, also hat f maximal $d|S|^{n-1}$ Nullstellen in S^n. Das ist die Induktionsbehauptung. □

6.2 Vergleich von Polynomprodukten

Im folgenden sei K irgendein (endlicher oder unendlicher) Körper.

Problemstellung: Gegeben seien Polynome $f_1, \ldots, f_r$ und $g_1, \ldots, g_s$ über K mit Variablen $X_1, \ldots, X_n$. Sei $f = f_1 \cdot \ldots \cdot f_r$ und $g = g_1 \cdot \ldots \cdot g_s$. Gilt $f = g$?

Um die Schwierigkeit der Fragestellung zu illustrieren, betrachten wir ein Beispiel. Die Polynome $f_1, \ldots, f_{2n}$ seien

$$X_1^s + 1, \ldots, X_n^s + 1, X_1^s - 1, \ldots, X_n^s - 1,$$

die Polynome $g_1, \ldots, g_n$ seien

$$X_1^{2s} - 1, \ldots, X_n^{2s} - 1.$$

Dabei kann s eine Zahl mit $O(n)$ Bits sein.[1] Wir „sehen“ (aufgrund der Formel $(X_i^s + 1)(X_i^s - 1) = X_i^{2s} - 1$), dass die Produkte der beiden Folgen gleich sind. Algorithmisch ist dies aber nicht ganz so einfach zu behandeln. (Die Terme könnten anders angeordnet sein, und zudem mit weniger offensichtlichen anderen Faktoren multipliziert sein.) Der nächstliegende deterministische Algorithmus wird die beiden Seiten ausmultiplizieren und dann vergleichen. Im Beispiel entstehen dann aber exponentiell viele Terme, der Aufwand ist also immens.

Tatsächlich ist für das Problem „Vergleich von Polynomprodukten“ kein deterministischer Algorithmus bekannt, der polynomielle Laufzeit hat. Wir geben einen sehr einfachen randomisierten Algorithmus an.

[1] Die rechnerinterne Darstellung von Polynomen für dieses Berechnungsproblem sollte man sich folgendermaßen vorstellen: Die Terme $a_{\ell_1,\ldots,\ell_n} \cdot X_1^{\ell_1} X_2^{\ell_2} \cdots X_n^{\ell_n}$, die nicht 0 sind, werden als $(\ell_1, \ell_2, \ldots, \ell_n, a_{\ell_1,\ldots,\ell_n})$ geschrieben, wobei die Zahlen binär dargestellt sind. Alle diese Terme sind als Liste gegeben. Auf diese Weise lassen sich auch Polynome mit sehr großen Graden sehr kompakt darstellen. Zum Beispiel hat das Polynom $X_1^{2^n-1} + X_2 + \cdots + X_n$ Darstellungsgröße $\Theta(n^2)$. Wenn man einen Term mit großen Exponenten für einen gegebenen Input $(a_1, \ldots, a_n) \in K$ auswerten möchte, benutzt man schnelle Exponentiation wie in Algorithmus 5.1.19. Die Größe $\text{size}(f_1, \ldots, f_r; g_1, \ldots, g_s)$ der Eingabe ist die gesamte (Bit-)Länge der Darstellung der beiden Listen von Polynomen.

Es sei

$$d = \max\{\deg(f_1) + \cdots + \deg(f_r), \deg(g_1) + \cdots + \deg(g_s)\}.$$

(Dann ist offenbar $\deg(f-g) \leq d$.) Wir legen eine beliebige endliche Menge[2,3] $S \subseteq K$ mit $|S| \geq 2d$ fest. Nun wählen wir $\boldsymbol{a} = (a_1, \ldots, a_n) \in S^n$ uniform zufällig und berechnen

$$b = f_1(\boldsymbol{a}) \cdot \ldots \cdot f_r(\boldsymbol{a}) \quad \text{und} \quad c = g_1(\boldsymbol{a}) \cdot \ldots \cdot g_s(\boldsymbol{a}).$$

(Es wird in die einzelnen Faktoren eingesetzt und ausgewertet, dann in K muliplziert.)

Ausgabe: $[b \neq c]$ (d. h. 1, falls $b \neq c$, und 0, falls $b = c$).

Algorithmus 6.1 ***Polynomprodukt***
Input: Polynome $f_1 \ldots, f_r$ und $g_1, \ldots, g_s$ über K mit Variablen $X_1, \ldots, X_n$.
Methode:
1 $d \leftarrow \max\{\deg(f_1) + \cdots + \deg(f_r), \deg(g_1) + \cdots + \deg(g_s)\}$;
2 Wähle endliche Menge $S \subseteq K$ mit $|S| \geq 2d$;
3 Wähle $\boldsymbol{a} = (a_1, \ldots, a_n) \in S^n$ zufällig;
4 **return** $[f_1(\boldsymbol{a}) \cdot \ldots \cdot f_r(\boldsymbol{a}) \neq g_1(\boldsymbol{a}) \cdot \ldots \cdot g_s(\boldsymbol{a})]$.

Es ist leicht zu sehen, dass Algorithmus 6.1 nur eine polynomielle Anzahl von Körperoperationen benötigt. Sogar wenn die Exponenten in den Polynomen groß sind, ist – mit schneller Exponentiation, siehe Algorithmus 5.1.19 – die Anzahl der K-Operationen nur linear in der Gesamtbitlänge dieser Exponenten.

Wir analysieren die Fehlerwahrscheinlichkeit.

1. Fall: $f = g$. – Dann ist $f(\boldsymbol{a}) = g(\boldsymbol{a})$ für jedes $\boldsymbol{a}$, also ist die Ausgabe immer 0.

2. Fall: $f \neq g$. – Dann gilt:

$$\text{Ausgabe ist } 0 \;\Leftrightarrow\; (f-g)(\boldsymbol{a}) = 0.$$

Das heißt: Die Ausgabe ist fehlerhaft genau dann wenn der zufällig gewählte Vektor $\boldsymbol{a}$ eine Nullstelle von $f-g$ ist. Nun ist $f-g$ ein Polynom von einem Grad ≥ 0 (weil $f \neq g$) und höchstens d. Nach dem Satz von Schwartz-Zippel (Satz 6.2) hat $f-g$ also höchstens $d|S|^{n-1}$ Nullstellen in S^n. Die Wahrscheinlichkeit, die falsche Ausgabe 0 zu erhalten, ist demnach höchstens

$$\frac{d|S|^{n-1}}{|S^n|} = \frac{d}{|S|} \leq \frac{1}{2}.$$

[2] Falls $|K| < 2d$, muss man mit aus der Algebra bekannten Methoden eine Körpererweiterung $K' \supseteq K$ mit $|K'| \geq 2d$ konstruieren und in K' rechnen.

[3] Es ist nicht nötig, die Elemente von S aufzulisten. Beispiel: Wenn $K = \mathbb{Z}_p$, könnte $S = \{0, \ldots, t-1\}$ sein, für eine Zahl $t < p$. Deshalb sind sehr große Grade d auch hier kein Hindernis.

Es handelt sich bei Algorithmus 6.1 also um einen Monte-Carlo-Algorithmus mit einseitigem Fehler, Fehlerschranke $d/|S|$. Wenn wir ihn ℓ-mal mit unabhängig gewählten Vektoren $\boldsymbol{a}$ durchführen, erhalten wir eine Fehlerschranke von $(d/|S|)^\ell$.

6.3 Polynomdeterminanten

Ganz ähnlich wie beim Vergleich von Polynomprodukten ist es bei Determinanten von Matrizen, deren Einträge Polynome sind. Gegeben sei also eine $m \times m$-Matrix

$$A(X_1, \ldots, X_n) = \begin{pmatrix} f_{11} & \cdots & f_{1m} \\ \vdots & & \vdots \\ f_{m1} & \cdots & f_{mm} \end{pmatrix}$$

mit Polynomeinträgen $f_{ij} = f_{ij}(X_1, \ldots, X_n)$. Wir nehmen an, dass $d_0 \geq \deg(f_{ij})$ für alle i, j gilt.[4] Aus der linearen Algebra kennt man die (Leibniz-Formel für die) Determinante:

$$\det(A) = \sum_{\sigma \in S_m} \operatorname{sign}(\sigma) \cdot f_{1,\sigma(1)} \cdot \ldots \cdot f_{m,\sigma(m)}.$$

Dabei ist S_m die Menge aller Permutationen σ der Menge $\{1, \ldots, m\}$, und $\operatorname{sign}(\sigma) \in \{-1, +1\}$ das Vorzeichen der Permutation σ.

Problem 1: Gegeben eine Polynommatrix A. Ist $\det(A) = 0$, also das Nullpolynom?

Problem 2: Gegeben eine Polynommatrix A und eine Folge $g_1, \ldots, g_s$ von Polynomen. Ist $\det(A) = g_1 \cdot \ldots \cdot g_s$?

Die Summe in der Definition der Determinante hat $n!$ Terme. Daher ist es normalerweise nicht möglich, das Polynom $\det(A)$ explizit zu berechnen. Auch hier hilft Randomisierung.

Sei $d = d_0 m$ oder eine andere obere Schranke für $\deg(\det(A))$. Wähle eine endliche Menge $S \subseteq K$ mit $|S| \geq 2d$. Nun wähle zufällig eine Folge $\boldsymbol{a} = (a_1, \ldots, a_n) \in S^n$. Berechne

$$\det(A(\boldsymbol{a})) = \det \begin{pmatrix} f_{11}(\boldsymbol{a}) & \cdots & f_{1m}(\boldsymbol{a}) \\ \vdots & & \vdots \\ f_{m1}(\boldsymbol{a}) & \cdots & f_{mm}(\boldsymbol{a}) \end{pmatrix}$$

wie folgt: Erst wird $\boldsymbol{a}$ in jede Komponente eingesetzt, und es wird jede für sich ausgewertet, dann wird die Determinante in K berechnet (Gauss-Elimination, $O(m^3)$ Körper-Operationen). Die Ausgabe ist $[\det(A(\boldsymbol{a})) \neq 0] \in \{0, 1\}$. Ganz analog zur Analyse von Algorithmus 6.1 sieht man folgendes: Wenn $\det(A)$ das Nullpolynom ist, dann ist die Ausgabe 0, und wenn $\det(A) \neq 0$, dann gilt $\mathbf{Pr}(\text{Ausgabe ist } 0) \leq d/|S| \leq \frac{1}{2}$.

[4] In Anwendungen sind die Einträge oft lineare Polynome, also $d_0 = 1$.

Ein analog gebauter Algorithmus löst auch Problem 2.

Eine graphentheoretische Anwendung der Polynomdeterminanten werden wir in Abschnitt 6.4 kennenlernen.

6.4 Perfektes Matching in bipartiten Graphen

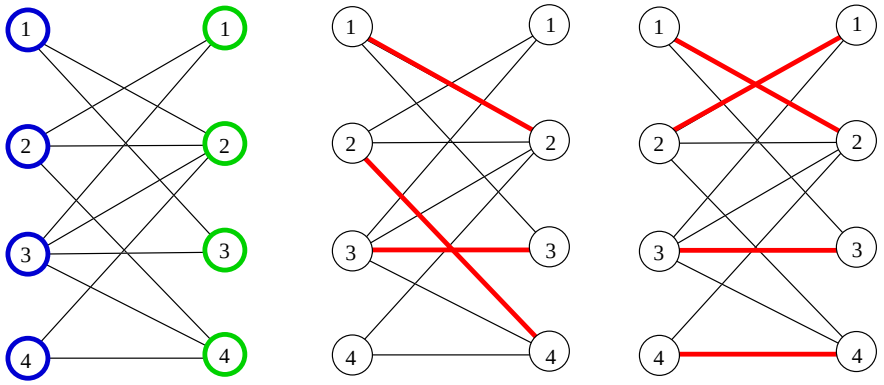


Abbildung 1: Links: Ein bipartiter Graph, $n = 4$ Knoten auf jeder Seite, $m = 11$ Kanten. Mitte: Ein nicht perfektes Matching. Rechts: Ein perfektes Matching.

In diesem Abschnitt betrachten wir bipartite Graphen $G = (U, V, E)$. Dabei ist $U = V = \{1, \ldots, n\}$ und $E \subseteq U \times V$. Eine Menge $M \subseteq E$ heißt ein *Matching* („paarweise Zuordnung“) in G, wenn für $(i, j), (i', j') \in M$ gilt: $i = i' \Leftrightarrow j = j'$. In Worten: Kanten in M sind entweder knotendisjunkt oder gleich. Ein Matching M heißt *perfekt*, wenn M aus n Kanten besteht – dann ist jeder Knoten in U genau einem Knoten in V zugeordnet.

Problem 1: Gegeben sei ein bipartiter Graph G.
Frage: Besitzt G ein perfektes Matching?

Problem 2: Gegeben sei ein bipartiter Graph G.
Aufgabe: Berechne ein perfektes Matching, wenn dies möglich ist.

Es gibt Standardalgorithmen, die diese Probleme lösen. Sie beruhen auf Flussberechnungen oder verwandten Verfahren. (Siehe Vorlesung „Effiziente Algorithmen“ im Master Informatik.) Wir stellen einen randomisierten Algorithmus für das Entscheidungsproblem vor. Dieser hat allerdings schlechtere Laufzeiten als die entsprechenden Flussalgorithmen. Der Vorteil liegt darin, dass er parallelisierbar ist, was für die Flussalgorithmen (vermutlich) nicht gilt.

Definition 6.3
$G = (U, V, E)$ sei ein bipartiter Graph auf 2 mal n Knoten. Die **Edmonds-Matrix** $\boldsymbol{A_G}$ ist eine $n \times n$-Matrix mit Einträgen

$$f_{ij} = \left\{ \begin{array}{ll} X_{ij}, & \text{falls } (i,j) \in E, \\ 0, & \text{falls } (i,j) \notin E, \end{array} \right\}, \quad \text{für } 1 \leq i,j \leq n.$$

Dabei sind die Variablen X_{ij}, $1 \leq i,j \leq n$, alle verschieden.[5]

Beispiel: Der bipartite Graph G aus Abb. 1 hat die Adjazenzmatrix bzw. Edmonds-Matrix

$$M_G = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \quad \text{bzw.} \quad A_G = \begin{pmatrix} 0 & X_{12} & X_{13} & 0 \\ X_{21} & X_{22} & 0 & X_{24} \\ X_{31} & X_{32} & X_{33} & X_{34} \\ 0 & X_{42} & 0 & X_{44} \end{pmatrix}.$$

Satz 6.4 ***Edmonds***
G hat ein perfektes Matching $\Leftrightarrow$ $\det(A_G) \neq 0$.

Die Polynomdeterminante $\det(A_G)$ wird dabei über einem beliebigen Körper K berechnet. Der Satz sagt also, dass man nur testen muss, ob $\det(A_G)$ das Nullpolynom ist, um die Existenz eines perfekten Matchings zu testen.

Beweis von Satz 6.4: Aus der Leibniz-Formel für die Determinante folgt:

$$\det(A_G) = \sum_{\sigma \in S_n} \underbrace{\operatorname{sign}(\sigma) \cdot f_{1,\sigma(1)} \cdots f_{n,\sigma(n)}}_{=:t_\sigma}. \tag{1}$$

Alle Terme in dieser Summe, die einen Faktor $f_{i,\sigma(i)} = 0$ haben, fallen heraus. Ein gegenseitiges Auslöschen anderer Terme ist unmöglich. Mit anderen Worten: Ein Summand $\operatorname{sign}(\sigma) \cdot f_{1,\sigma(1)} \cdots f_{n,\sigma(n)}$ ist genau dann in der Summe $\det(A_G)$ enthalten, wenn $f_{i,\sigma(i)} = X_{i,\sigma(i)}$ für alle i ist, und das heißt, dass E alle Kanten $(i, \sigma(i))$ enthält.

Daraus folgt: Die Summe in (1) ist genau dann nicht leer, wenn es eine Permutation σ gibt, für die E alle Kanten $(i, \sigma(i))$ enthält. Dies ist äquivalent zur Existenz eines perfekten Matchings. □

Beispiel: Beim bipartiten Graph G aus Abb. 1 hat $\det(A_G)$ den Term $X_{13}X_{21}X_{34}X_{42}$, entsprechend dem perfekten Matching $M = \{(1,3), (2,1), (3,4), (4,2)\}$.

[5] Achtung: Wir haben die Notation geändert. n bezeichnet die Knotenanzahl. Die Anzahl der Variablen ist n^2. In A_G kommen aber nur $m = |E|$ viele Variablen X_{ij} tatsächlich vor.

Algorithmus 6.2 ***Perfektes Matching, bipartit***
Input: Bipartiter Graph $G = (U, V, E)$ mit n Knoten auf beiden Seiten.
Methode:
1 Wähle Primzahl $p > 2n$; // Körper $\mathbb{Z}_p$, $S = \mathbb{Z}_p$
2 Bilde die Edmonds-Matrix A_G (eine $n \times n$-Matrix);
3 Wähle $\boldsymbol{a} = (a_{11}, \ldots, a_{1n}, \ldots, a_{n1}, \ldots, a_{nn}) \in \mathbb{Z}_p^{n \times n}$ zufällig;
4 $B \leftarrow A_G(\boldsymbol{a})$; // bilde $A_G(\boldsymbol{a})$ durch Einsetzen von $\boldsymbol{a}$ in A_G
5 **return** $[\det(B) \neq 0]$. // Gauss-Elimination über $\mathbb{Z}_p$

Wie vorher sehen wir: Wenn G kein perfektes Matching hat, also $\det(A_G)$ das Nullpolynom ist, dann gilt $\det(A_G(\boldsymbol{a})) = 0$ für alle $\boldsymbol{a}$, also ist die Antwort auf jeden Fall 0. Wenn G ein perfektes Matching hat, also $\det(A_G)$ nicht das Nullpolynom ist, dann gilt nach dem Satz von Schwartz-Zippel (Satz 6.2):

$$\mathbf{Pr}(\text{Antwort ist } 0) \leq \frac{\deg(\det(A_G))}{|\mathbb{Z}_p|} = \frac{n}{|\mathbb{Z}_p|} < \frac{1}{2}.$$

Bemerkung: (1) Das Bilden der Edmonds-Matrix im Algorithmus ist überflüssig; man erhält die Matrix $A_G(\boldsymbol{a})$ direkt aus der Adjazenzmatrix M_G, indem man jeden 1-Eintrag in M_G durch ein Zufallselement aus $\mathbb{Z}_p$ ersetzt (und die 0-Einträge stehen lässt).
(2) Anstatt in $\mathbb{Z}_p$ könnte man auch in $\mathbb{Q}$ rechnen; man setzt dann z. B. $S = \{1, \ldots, 2n\}$. Der Nachteil hierbei ist, dass die Zahlen in Zwischenergebnissen zwischen $\Omega(n \cdot \log n)$ und $O((n \cdot \log n)^2)$ Bits haben, so dass die einzelnen arithmetischen Operationen sehr teuer sind.
(3) Man kann zeigen, dass man mit polynomiell vielen Prozessoren die Determinante einer $n \times n$-Matrix mit $O((\log n)^2)$ Runden von parallel ausgeführten Ringoperationen berechnen kann (*Algorithmus von Samuelson/Berkowitz*). Damit kann man auch die Existenz eines perfekten Matchings in paralleler Zeit $O((\log n)^2)$ [Körperoperationen] testen – aber nur randomisiert. Für dieses Problem war bis 2016 kein schneller deterministischer paralleller Algorithmus bekannt.[6]

Problem 2, die Konstruktion eines perfekten Matchings, lässt sich wie folgt lösen: Zunächst testet man, ob G ein perfektes Matching besitzt. Nur im positiven Falle fährt man wir folgt fort. Setze $G' = (U, V, E') = G$. Man behandelt die Kanten $(i, j) \in E'$

[6] Aktualisierung 2018: Die Arbeit „Stephen A. Fenner, Rohit Gurjar, Thomas Thierauf: Bipartite perfect matching is in quasi-NC. STOC 2016. S. 754–763“ präsentiert einen deterministischen parallelen Algorithmus, der die Existenz eines perfekten Matchings mit einem „fast effizienten“ parallelen Algorithmus testet. Als effizient gelten normalerweise Algorithmen, die mit $n^{O(1)}$ („polynomiell vielen“) Prozessoren und paralleler Rechenzeit $(\log n)^{O(1)}$ auskommen. Der neue Algorithmus benutzt $n^{O(\log n)}$ („quasipolynomiell viele“) Prozessoren und $O((\log n)^2)$ parallele Zeit.

nacheinander, und verändert dabei eventuell den Graphen G'. Die Untersuchung von Kante $e = (i, j) \in E'$ verläuft so: Teste, ob $G'' = (U, V, E' - \{e\})$ ein perfektes Matching besitzt. Wenn dies der Fall ist, streiche e aus E'. Am Ende teste, ob E' ein perfektes Matching ist. Falls ja, wird E' ausgegeben, falls nein, beginne von vorne (wie bei Las-Vegas-Algorithmen üblich). Die Fehlerwahrscheinlichkeit bei einem Versuch kann man auf $1/n$ senken, indem man für jeden Test Algorithmus 6.2 $\log(|E| \cdot n)$-mal wiederholt.

Leider ist diese Methode zum Finden von perfekten Matchings wieder iterativ ($m = |E|$ Runden), also nicht leicht parallelisierbar. Effiziente parallele Algorithmen benötigen weitere Techniken, die in Abschnitt 6.6 vorgestellt werden, im Kontext von Matchings in beliebigen ungerichteten Graphen.

6.5 Textsuche (String-Matching) mit Fingerprinting

6.5.1 Textvergleich

In diesem Abschnitt geht es um Algorithmen für Texte. Wir nehmen dabei o. B. d. A. stets an, dass wir ein Alphabet $\Sigma = \{0, \ldots, u-1\}$ benutzen, dessen Buchstaben natürliche Zahlen sind.

Zunächst betrachten wir ein Verfahren zum Vergleich zweier gleich langer Wörter $a = (a_1, \ldots, a_n)$ und $b = (b_1, \ldots, b_n)$. Wir wählen eine Primzahl $p > u$ und betrachten die Polynome

$$f_a = a_1 + a_2 X + a_3 X^2 + \cdots + a_n X^{n-1} \text{ und } f_b = b_1 + b_2 X + b_3 X^2 + \cdots + b_n X^{n-1},$$

jeweils über dem Körper $\mathbb{Z}_p$. Offenbar gilt: $f_a = f_b$ genau dann wenn $a = b$. Für $h = f_a - f_b$ gibt es zwei Fälle:

1. Fall: $a = b$. – Dann ist h das Nullpolynom; für jedes $r \in \mathbb{Z}_p$ gilt $f_a(r) = f_b(r)$.
2. Fall: $a \neq b$. – Dann ist h nicht das Nullpolynom. Da $\deg(h) \leq n - 1$, hat h nach Lemma 6.1 nicht mehr als $n - 1$ Nullstellen. Also gibt es höchstens $n - 1$ Elemente $r \in \mathbb{Z}_p$ mit $f_a(r) = f_b(r)$.

Die Idee ist also, ein r aus $\mathbb{Z}_p$ zufällig zu wählen und zu testen, ob $f_a(r) = f_b(r)$ gilt. Die Werte $f_a(r)$ bzw. $f_b(r)$ können als (kurze) „Fingerabdrücke“ von $a = (a_1, \ldots, a_n)$ und $b = (b_1, \ldots, b_n)$ aufgefasst werden, die es gestattet, diese Wörter zu unterscheiden (wenn sie verschieden sind).

Algorithmus 6.3 ***Textvergleich mit Fingerprinting***
Input: $a = (a_1, \ldots, a_n)$ und $b = (b_1, \ldots, b_n)$ aus Σ^n, mit $\Sigma = \{0, \ldots, u-1\}$.
Methode:
1 Wähle eine Primzahl $p > \max\{u, 2n\}$;
2 Wähle zufällig ein r aus $\{0, \ldots, p-1\}$;
3 $\mathrm{fp}_a := (a_1 + a_2 \cdot r + a_3 \cdot r^2 + \cdots + a_n \cdot r^{n-1}) \bmod p$;
4 $\mathrm{fp}_b := (b_1 + b_2 \cdot r + b_3 \cdot r^2 + \cdots + b_n \cdot r^{n-1}) \bmod p$;
5 **if** $\mathrm{fp}_a = \mathrm{fp}_b$ **then return** 0 **else return** 1.

Die Polynomauswertungen in Zeilen 3 und 4 bewerkstelligt man in linearer Zeit mit dem bekannten Horner-Schema. Insgesamt kostet dieser Algorithmus also Zeit $O(n)$. Wenn $a = b$ ist, wird auf jeden Fall 0 ausgegeben. Wenn $a \neq b$ ist, gilt

$$\mathbf{Pr}(\text{Ausgabe ist } 0) = \frac{\#(\text{Nullstellen von } h = f_a - f_b)}{p} \leq \frac{n-1}{p} < \frac{1}{2}.$$

Interessant ist noch die folgende Beobachtung: Der Algorithmus ist durchführbar, selbst wenn sich die Daten a und b nicht am selben Ort (im selben Computer) befinden. Es genügt, am jeweiligen Ort die Fingerprints fp_a und fp_b zu berechnen und einen der beiden zu übermitteln. Dabei ist für eine Fehlerwahrscheinlichkeit von $1/2^\ell$ die Übermittlung von nur $\ell \cdot 2\log(\max\{u, 2n\})$ Bits nötig – viel weniger als wenn man die $n \log u$ Bits des gesamten Textes schicken würde.

6.5.2 Textsuche (Pattern matching, Algorithmus von Rabin-Karp)

Im Fall der Textsuche hat man es mit folgender Problemstellung zu tun:

Gegeben sind:

- „Text“ $t = (t_1, \ldots, t_n) \in \Sigma^n$ und
- „Muster“ $a = (a_1, \ldots, a_m) \in \Sigma^m$.

Die Frage ist, ob a als Teilwort $(t_i, \ldots, t_{i+m-1})$ in t vorkommt, und falls ja, an welcher Stelle bzw. welchen Stellen i.

Beispiel: Das Muster `bra` kommt im Text `abrakadabra` an den Stellen $i = 2$ und $i = 9$ vor.

Es muss gesagt werden, dass es für dieses sehr wichtige Problem auch hocheffiziente deterministische Algorithmen gibt. Dennoch ist die Betrachtung randomisierter Algorithmen interessant. Diese können auch leicht adaptiert werden, wenn es um

mehrdimensionale „Texte“ und „Muster“ geht oder wenn das Muster unbestimmte Buchstabenpositionen hat („wildcards“).

Um (für $1 \leq i \leq n - m + 1$) a mit $(t_i, \ldots, t_{i+m-1})$ zu vergleichen, wollen wir die „Fingerabdruckspolynome“

$$f_i = f_i(X) = t_i X^{m-1} + t_{i+1} X^{m-2} + \cdots + t_{i+m-2} X + t_{i+m-1}$$

und

$$g = g(X) = a_1 X^{m-1} + a_2 X^{m-2} + \cdots + a_{m-1} X + a_m$$

über $\mathbb{Z}_p$ benutzen. Wie in Abschnitt 6.5.1 sieht man: Wenn $a \neq (t_i, \ldots, t_{i+m-1})$ ist, sind die Polynome f_i und g nicht identisch, also ist das Differenzpolynom $h_i = f_i - g$ nicht das Nullpolynom. Weil der Grad von h_i nicht größer als $m - 1$ ist, hat h_i nach Lemma 6.1 höchstens $m - 1$ Nullstellen in $\mathbb{Z}_p$. Wir folgern:

$$|\{r \mid 0 \leq r < p \wedge h_i(r) = 0\}| \leq m - 1. \tag{2}$$

Wir wählen hier $p > knm$, für ein beliebig festzusetzendes k (das auch von n, m abhängen darf). Wenn wir nun r aus $\{0, \ldots, p-1\}$ zufällig wählen, folgt aus (2):

$$\mathbf{Pr}_r(\exists i \leq n - m + 1 \colon a \neq (t_i, \ldots, t_{i+m-1}) \wedge f_i(r) = g(r)) \leq \frac{(n-m)m}{p} < \frac{1}{k}. \tag{3}$$

Was soll nun der Vorteil dieses Verfahrens sein? Wir müssen $n - m + 2$ Polynomauswertungen vornehmen. Auf naive Weise implementiert kostet dies Zeit $O(nm)$, und das ist ebenso langsam wie der naive Textsuchalgorithmus.

Der Trick ist, dass sich die Berechnung der Fingerabdrücke beschleunigen lässt. Aus

$$\begin{aligned} f_i(r) &= t_i \cdot r^{m-1} + \quad t_{i+1} \cdot r^{m-2} + \cdots + t_{i+m-2} \cdot r \quad + t_{i+m-1} \quad \text{und} \\ f_{i+1}(r) &= \quad\quad t_{i+1} \cdot r^{m-1} + \cdots + t_{i+m-2} \cdot r^2 \quad + t_{i+m-1} \cdot r + t_{i+m} \end{aligned} \tag{4}$$

erhalten wir

$$f_{i+1}(r) = (f_i(r) - t_i \cdot r^{m-1}) \cdot r + t_{i+m}.$$

Das Körperelement r^{m-1} können wir in Zeit $O(\log m)$ vorab berechnen. Dann kann man aus $f_i(r)$ in Zeit $O(1)$ den nächsten Wert $f_{i+1}(r)$ berechnen. Als Algorithmus ergibt sich zusammengefasst folgendes:

Algorithmus 6.4 ***Textsuche mit Fingerprinting, Rabin-Karp***
Input: $a = (a_1, \ldots, a_m) \in \Sigma^m$ und $t = (t_1, \ldots, t_n) \in \Sigma^n$, $n \geq m$; Zahl $k \geq 2$.
Aufgabe: Finde die Positionen $i \in \{1, \ldots, n-m+1\}$ mit $a = (t_i, \ldots, t_{i+m-1})$.
Methode:

```
1    Wähle eine Primzahl p > nmk;
2    Wähle zufällig ein r aus {0,...,p-1};
3    f ← (t_1 · r^{m-1} + t_2 · r^{m-2} + ··· + t_{m-1} · r + t_m) mod p;   // Horner-Schema
4    g ← (a_1 · r^{m-1} + a_2 · r^{m-2} + ··· + a_{m-1} · r + a_m) mod p;   // Horner-Schema
5    i ← 1;
6    j ← m;
7    z ← r^{m-1} mod p;
8    if f = g then print(i);
9    while j < n do
10     j ← j + 1;
11     f ← ((f − t_i · z) · r + t_j) mod p
12     i ← i + 1;
13     if f = g then print(i);
14   enddo
```

Wir benennen die Eigenschaften dieses Algorithmus.

- Der Rechenaufwand ist $O(m)$ am Anfang und $O(1)$ in jedem der $n-m$ Schleifendurchläufe, insgesamt also $O(n)$.
- Mit (4) zeigt man Folgendes (durch Induktion über $i = 2, \ldots, n-m+1$): In Schleifendurchlauf für $(t_i, \ldots, t_{i+m-1})$ (Inhalt von `i` zu Beginn: $i-1$, nach Zeile 12: i) erhält die Variable `f` in Zeile 11 den Wert $f_i(r)$; dieser wird dann in Zeile 13 auf Gleichheit mit `g` getestet, das den Fingerprint g von a enthält. Wenn $a = (t_i, \ldots, t_{i+m-1})$ ist, muss sich hier Gleichheit ergeben und i wird ausgegeben; wenn $a \neq (t_i, \ldots, t_{i+m-1})$ ist, ist die Wahrscheinlichkeit, dass i ausgegeben wird, höchstens $(m-1)/(knm) < 1/(kn)$.
- Die Wahrscheinlichkeit, dass es ein i mit $a \neq (t_i, \ldots, t_{i+m-1})$ gibt, so dass i ausgegeben wird, ist höchstens $1/k$.

Wenn man nur das erste Vorkommen von a in t finden möchte, kann man den Algorithmus folgendermaßen variieren: Wenn i ausgegeben wird, wird die Schleife unterbrochen und es wird direkt geprüft, ob $a = (t_i, \ldots, t_{i+m-1})$ ist. Falls dies so ist, ist man fertig, falls nicht, wird die Bearbeitung der Schleife wieder aufgenommen. Der modifizierte Algorithmus kann niemals ein falsches Resultat liefern (es handelt sich also um einen Las-Vegas-Algorithmus), der erwartete zusätzliche Aufwand für diese

Tests ist $m \cdot (1 + \frac{1}{k})$ Vergleiche zwischen Buchstaben. – Der direkte Kontrollvergleich ist problematisch, wenn man *alle* Vorkommen des Musters finden will. Das Muster könnte $\omega(n/m)$-mal in t vorkommen; dann wäre der Vergleichsaufwand $\omega(n)$, also nicht linear.

Bemerkung: (a) Die beschriebene Technik lässt sich auf höhere Dimensionen verallgemeinern. Für $d = 2$ wäre zum Beispiel der „Text“ eine Bitmap aus $m_1 \times m_2$ Pixeln, das Muster eine Bitmap aus $n_1 \times n_2$ Pixeln. Man soll herausfinden, ob das Muster im Text irgendwo auftaucht. Dies lässt sich mit Aufwand $O(m_1 \cdot m_2)$ bewerkstelligen. (Dies ist eine interessante Übungsaufgabe. Wenn man das Muster horizontal verschiebt, kommen ja etwa n_1 neue Bildpunkte hinzu und n_1 viele verschwinden. Wie kann man die Aktualisierung der entsprechenden Polynomwerte in $O(1)$ Zeit realisieren?)

(b) Wenn das Muster „wildcards“ oder „don't care“-Buchstaben, also unbestimmte Positionen, enthält, kann man eine Variante von Algorithmus 6.4 benutzen. Dabei ist für jede unbestimmte Buchstabenposition pro Runde eine konstante Anzahl von arithmetischen Operationen nötig.

6.6 $*$ Perfekte Matchings in beliebigen ungerichteten Graphen

$*$ Dieser Abschnitt ist **im Sommersemester 2021 nicht prüfungsrelevant**.

Wir betrachten ungerichtete Graphen $G = (V, E)$ mit $V = \{1, \ldots, n\}$. Die Adjazenzmatrix A_G von G ist symmetrisch und hat Nullen auf der Hauptdiagonale. Es geht zunächst um den Test, ob G ein perfektes Matching $M \subseteq E$ besitzt, d. h. eine Kantenmenge M, in der jeder Knoten genau einmal vorkommt. Offensichtlich ist es hierfür notwendig, dass $n = |V|$ eine gerade Zahl ist. Die Größe $|M|$ eines perfekten Matchings muss $n/2$ sein.[7]

Definition 6.5
Die **Tutte-Matrix T_G** zu einem Graphen G mit Knoten $1, \ldots, n$ ist eine $n \times n$-Matrix mit Einträgen

$$f_{ij} = \left\{ \begin{array}{ll} X_{ij}, & \textit{falls } i < j \textit{ und } (i,j) \in E, \\ -X_{ji}, & \textit{falls } j < i \textit{ und } (i,j) \in E, \\ 0, & \textit{sonst.} \end{array} \right\}, \quad \textit{für } 1 \leq i, j \leq n.$$

Beispiel: Zum Graphen $G = (\{1,2,3,4\}, E)$ mit $E = \{(1,2),(1,3),(1,4),(2,4)\}$ gehört die Tutte-Matrix

$$T_G = \begin{pmatrix} f_{11} & f_{12} & f_{13} & f_{14} \\ f_{21} & f_{22} & f_{23} & f_{24} \\ f_{31} & f_{32} & f_{33} & f_{34} \\ f_{41} & f_{42} & f_{43} & f_{44} \end{pmatrix} = \begin{pmatrix} 0 & X_{12} & X_{13} & X_{14} \\ -X_{12} & 0 & 0 & X_{24} \\ -X_{13} & 0 & 0 & 0 \\ -X_{13} & -X_{24} & 0 & 0 \end{pmatrix}.$$

Offensichtlich gilt ${T_G}^{\mathsf{T}} = -T_G$, die Tutte-Matrix ist also *schiefsymmetrisch*. Die Einträge f_{ii} auf der Diagonalen sind 0. Auch für Tutte-Matrizen gibt es einen Zusammenhang zwischen Determinantenpolynom und der Existenz von perfekten Matchings in G.

Satz 6.6 ***(Tutte)***
Sei G ein beliebiger ungerichteter Graph. Dann gilt:
G hat ein perfektes Matching $\Leftrightarrow$ $\det(T_G) \neq 0$.

[7] William T. „Bill“ Tutte (1917–2002) war ein bedeutender britisch-kanadischer Mathematiker. Im 2. Weltkrieg arbeitete er als Kryptanalytiker in Bletchley Park, UK. Nach dem Krieg wandte er sich der Kombinatorik und Graphentheorie zu und erzielte dort bahnbrechende Ergebnisse. Er lehrte von 1962 bis 1985 an der University of Waterloo in Kanada.

Beweis von Satz 6.6: Man schreibt

$$\det(T_G) = \sum_{\sigma \in S_n} \operatorname{sign}(\sigma) \cdot f_{1,\sigma(1)} \cdots f_{n,\sigma(n)}. \tag{5}$$

„⇒“: Sei M ein perfektes Matching in G. Dann betrachtet man die Permutation σ_M, die i auf j und j auf i abbildet genau dann wenn $(i,j) \in M$ ist. In der Summe für die Determinante kommt der Term $t_{\sigma_M} = \operatorname{sign}(\sigma_M) \cdot f_{1,\sigma_M(1)} \cdots f_{n,\sigma_M(n)}$ vor, und keiner der Faktoren ist 0, weil jedes Paar $(i, \sigma_M(i))$ einer Kante in E entspricht. Weil $\sigma^{-1} = \sigma$ gilt und die Permutation σ aus den Faktoren des Terms $f_{1,\sigma(1)} \cdots f_{n,\sigma(n)}$ abgelesen werden kann, kann es auch keinen anderen Term geben, der t_{σ_M} auslöscht, also ist $\det(T_G)$ nicht das Nullpolynom.

„⇐“: Nun nehmen wir an, dass $\det(T_G)$ nicht das Nullpolynom ist. Wir betrachten die Summe in (5).

Jeder Term $t_\sigma = \operatorname{sign}(\sigma) \cdot f_{1,\sigma(1)} \cdots f_{n,\sigma(n)}$, der nicht 0 ist, definiert eine „*Spur*“

$$\operatorname{tr}_\sigma = \{(i, \sigma(i)) \mid 1 \le i \le n\},$$

eine Menge von geordneten Paaren, die wir als gerichtete Kanten auffassen. Jede dieser Kanten läuft entlang einer (ungerichteten) Kante in G. Weil σ Permutation ist, hat jeder Knoten i Eingangsgrad und Ausgangsgrad 1, also bildet tr_σ eine Menge von Kreisen, die Zyklen von σ entsprechen. Weil $f_{ii} = 0$ ist, haben diese Kreise eine Mindestlänge von 2.

Wir betrachten zunächst Permutationen σ, in deren Spuren ein Kreis ungerader Länge vorkommt. Wenn ein solches σ vorliegt, wählen wir den Kreis K_σ ungerader Länge in tr_σ, der den Knoten mit kleinster Nummer enthält. Wir drehen in tr_σ die Umlaufrichtung des Kreises K_σ herum. Dadurch entsteht die Spur $\operatorname{tr}_{\sigma'}$ einer eindeutig bestimmten Permutation σ'. Auf diese Weise entsteht eine eineindeutige Zuordnung $\sigma \mapsto \sigma'$ unter den Permutationen mit Spuren, die einen ungeraden Kreis haben. Diese Zuordnung ist *involutorisch*, d. h., sie ist gleich ihrer eigenen Umkehrung. Auf diese Weise werden die Permutationen σ mit einem ungeraden Kreis in tr_σ einander paarweise zugeordnet. – Es gilt dabei $\operatorname{sign}(\sigma) = \operatorname{sign}(\sigma')$, da das Vorzeichen einer Permutation nur von der Anzahl der geraden Kreise abhängt[8] und diese bei σ und σ' gleich ist.

[8] Genauer gesagt ist $\operatorname{sign}(\sigma)$ gleich 1 bzw. -1 je nachdem, ob tr_σ eine gerade oder ungerade Anzahl gerader Kreise hat.

Beispiel 6.7
Betrachte die folgende Permutation σ:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$\sigma(i)$	16	18	17	2	7	11	1	13	12	8	3	9	10	6	4	5	14	15

Diese Permutation hat die folgenden Kreise:

$k_1 = (9, 12), k_2 = (2, 18, 15, 4), k_3 = (7, 1, 16, 5), k_4 = (8, 13, 10), k_5 = (6, 11, 3, 17, 14).$

Kreise k_1, k_2, k_3 haben gerade Länge, Kreise k_4 und k_5 haben ungerade Länge. Damit ist $\text{sign}(\sigma) = -1$. Unter den beiden Kreisen ungerader Länge ist $K_\sigma = k_5$ derjenige, der den Knoten mit der kleinsten Nummer enthält, nämlich 3. (Knoten 1 und 2 liegen auf geraden Kreisen.) Damit erhalten wir σ' durch Umdrehen von $K_\sigma = k_5$ zu $K_{\sigma'} = k_5' = (14, 17, 3, 11, 6)$. Die Tabelle von σ' (Änderungen gegenüber σ in Fettdruck):

i	1	2	**3**	4	5	**6**	7	8	9	10	**11**	12	13	**14**	15	16	**17**	18
$\sigma'(i)$	16	18	**11**	2	7	**14**	1	13	12	8	**6**	9	10	**17**	4	5	**3**	15

Dann gilt:

$$\begin{aligned}
t_\sigma + t_{\sigma'} &= \text{sign}(\sigma) \cdot f_{1,\sigma(1)} \cdots f_{n,\sigma(n)} + \text{sign}(\sigma') \cdot f_{1,\sigma'(1)} \cdots f_{n,\sigma'(n)} \\
&= \text{sign}(\sigma) \cdot \prod_{(i,\sigma(i)) \notin K_\sigma} f_{i,\sigma(i)} \cdot \Big(\prod_{(i,\sigma(i)) \in K_\sigma} f_{i,\sigma(i)} + \prod_{(i,\sigma(i)) \in K_\sigma} f_{\sigma(i),i} \Big) \\
&\overset{(*)}{=} \text{sign}(\sigma) \cdot \prod_{(i,\sigma(i)) \notin K_\sigma} f_{i,\sigma(i)} \cdot \Big(\prod_{(i,\sigma(i)) \in K_\sigma} f_{i,\sigma(i)} + \prod_{(i,\sigma(i)) \in K_\sigma} (-f_{i,\sigma(i)}) \Big) \\
&= \text{sign}(\sigma) \cdot f_{1,\sigma(1)} \cdots f_{n,\sigma(n)} \cdot \Big(1 + (-1)^{\#(\text{Kanten in } K_\sigma)} \Big) = 0.
\end{aligned}$$

Die Gleichheit $(*)$ folgt dabei aus der Definition der Tutte-Matrix, die letzte Gleichheit aus der Tatsache, dass K_σ ungerade Länge hat. Das heißt, dass die Permutationen σ, deren Spuren einen ungeraden Kreis enthalten, insgesamt nichts zum Polynom $\det(T_G)$ beitragen.

Wir hatten angenommen, dass $\det(T_G)$ nicht das Nullpolynom ist. Nach dem eben erreichten Zwischenergebnis muss es dann eine Permutation σ mit $t_\sigma \neq 0$ geben, deren Spur tr_σ nur aus Kreisen gerader Länge besteht. Aus einer solchen Spur lässt sich aber leicht ein perfektes Matching M_σ konstruieren: Man nimmt aus jedem Kreis gerader Länge jede zweite Kante und ignoriert dann die Richtung. □

Der Algorithmus für den Test, ob $G = (V, E)$ ein perfektes Matching enthält, ist nun (bis auf die Sache mit den Vorzeichen) derselbe wie bei den bipartiten Graphen. In der

folgenden Formulierung lassen wir den Umweg über die Tutte-Matrix von vornherein weg.

Algorithmus 6.5 ***Test auf perfektes Matching, ungerichteter Graph***
Input: (Ungerichteter) Graph $G = (V, E)$ mit $V = \{1, \ldots, n\}$.
Problem: Teste, ob G ein perfektes Matching hat.
Methode:
0 Wenn n ungerade ist: **return** 0.
1 Wähle Primzahl $p > 2n$; // Körper $\mathbb{Z}_p$, $S = \mathbb{Z}_p$
2 Für $(i, j) \in E$ mit $i < j$ wähle a_{ij} aus $\mathbb{Z}_p$ zufällig;
3 Für $(i, j) \in E$ mit $i > j$ setze $a_{ij} \leftarrow -a_{ji}$;
4 Für $(i, j) \notin E$ setze $a_{ij} \leftarrow 0$;
5 Bilde Matrix $A = (a_{ij})_{1 \le i,j, \le n}$;
6 **return** $[\det(A) \neq 0]$. // z. B. Gauss-Elimination über $\mathbb{Z}_p$

Zeitbedarf und Wahrscheinlichkeitsanalyse sind praktisch identisch zu dem bei Algorithmus 6.2. Ebenso gelten hier auch die Bemerkungen zur Parallelisierung.

Es sei bemerkt, dass *deterministische* Algorithmen zum Test auf die Existenz von perfekten Matchings in allgemeinen ungerichteten Graphen zwar existieren, aber ungleich komplizierter sind als die für bipartite Graphen.

Nehmen wir nun an, wir haben mit Algorithmus 6.5, eventuell nach mehreren Wiederholungen, festgestellt, dass Graph G ein perfektes Matching besitzt. Nun wollen wir ein solches perfektes Matching *berechnen*. Man könnte dazu vorgehen wie im Fall von bipartiten Graphen (s. Ende von Abschnitt 6.4), d. h. die Kanten (i, j) nacheinander darauf testen, ob der Graph ohne (i, j) immer noch ein perfektes Matching besitzt und (i, j) zu streichen, falls dies so ist.

Wir betrachten einen alternativen Algorithmus, der auch gut parallelisierbar ist. (Literatur für das Folgende: K. Mulmuley, U. V. Vazirani, V. V. Vazirani, Matching is as easy as matrix inversion, *Combinatorica* 7 (1): 105113. 1987.) Wir benötigen eine Tatsache aus der Algorithmik für lineare Algebra.

Fakt 6.8 ***(Samuelson 1942, Berkowitz 1985)***
$O(n^3)$ Prozessoren können die Determinante einer Matrix $A \in K^{n \times n}$ in $O((\log n)^2)$ Zeit (d. h. Additionen und Multiplikationen) berechnen.

Die Idee für den Matchingalgorithmus ist dann im Prinzip einfach: Für jede Kante $(i, j) \in E$ testet eine Gruppe von $O(n^3)$ Prozessoren (die jeweils eine Addition oder Multiplikation in K in einem Schritt ausführen können), z. B. durch Berechnen einer

passenden Determinante, ob $(i,j) \in E$ Element eines „gesuchten“ perfekten Matchings ist.

Dabei stellt sich aber folgendes Problem: Ein Graph G kann viele verschiedene perfekte Matchings haben, und man müsste die Prozessoren dazu bringen, dass alle ihren Test bezüglich eines festen perfekten Matchings durchführen. Um eine solche Auswahl zu treffen, verwendet man wieder Randomisierung. Man versieht die Kanten in E mit zufällig gewählten Gewichten, und zwar so, dass es mit einer gewissen Wahrscheinlichkeit nur *ein* perfektes Matching M_0 mit minimalem Gesamtgewicht gibt. Dann zeigt man, dass eine Determinantenberechnung pro Kante (i,j) genügt, um zu testen, ob $(i,j) \in M_0$ gilt.

Definition 6.9
Ein (endliches) Mengensystem $(X, \mathcal{F})$ besteht aus einer Menge X mit $|X| = m \geq 1$ und einer Menge $\mathcal{F}$ von Teilmengen von X. Wenn eine Gewichtsfunktion

$$w\colon X \ni x \mapsto w(x) \in \mathbb{N}$$

gegeben ist, definieren wir $w(S)$ als $\sum_{x \in S} w(x)$, für $S \in \mathcal{F}$.

In unserer Anwendung wird $X = E$ sein und $\mathcal{F}$ die Menge aller perfekten Matchings in G. Gewichte für die Kanten werden zufällig gewählt.

Lemma 6.10 ***Isolationslemma***
Wenn man $w(x)$, für $x \in X$ aus $\{1, \ldots, 2m\}$ zufällig wählt, dann gilt:

$$\mathbf{Pr}(\text{in } \mathcal{F} \text{ gibt es eine eindeutig bestimmte Menge mit minimalem Gewicht}) \geq \frac{1}{2}.$$

(Anstelle des Faktors 2 kann man auch einen beliebigen anderen Faktor $r \geq 2$ wählen und erhält Wahrscheinlichkeit $\geq 1 - 1/r$.)

Beweis: Wir können o. B. d. A. annehmen, dass jedes $x \in X$ in einigen der $S \in \mathcal{F}$ vorkommt, in anderen nicht. (Wenn ein x in allen S vorkommt oder in keinem, dann hat $w(x)$ auf die Anordung der Gewichte $w(S)$, $S \in \mathcal{F}$ keinen Einfluss, und wir können dieses x für die Zwecke des Beweises ignorieren.)

Betrachte ein $x \in X$ und definiere:

$$\begin{aligned} W_x &:= \text{minimales Gewicht } w(S - \{x\}) \text{ über alle } S \in \mathcal{F} \text{ mit } x \in S; \\ \bar{W}_x &:= \text{minimales Gewicht } w(S) \text{ über alle } S \in \mathcal{F} \text{ mit } x \notin S. \\ \alpha_x &:= \bar{W}_x - W_x. \end{aligned}$$

Man beachte, dass die Zufallsvariablen W_x, $\bar{W}_x$ und α_x nur von $w(x')$ mit $x' \neq x$ abhängen, nicht von $w(x)$. Also sind α_x und $w(x)$ unabhängig.

Wir beobachten:

(i) Wenn $w(x) < \alpha_x$, dann muss *jede* Menge $S \in \mathcal{F}$, die minimales Gewicht hat, x enthalten:
Sei S_0 die Menge mit kleinstem Gewicht in $\mathcal{F}$, die x enthält. Dann ist

$$w(S_0) = w(S_0 - \{x\}) + w(x) = W_x + w(x).$$

Für jede Menge S', die x nicht enthält, gilt dann:
$w(S') \geq \bar{W}_x = \bar{W}_x - (W_x + w(x)) + w(S_0) = \alpha_x - w(x) + w(S_0) > w(S_0)$,
also kann $w(S')$ nicht minimal sein.

(ii) Wenn $w(x) > \alpha_x$, dann kann *keine* Menge S, die minimales Gewicht hat, x enthalten:
Sei S_0 die Menge mit kleinstem Gewicht in $\mathcal{F}$, die x enthält. Wie in (i) gilt $w(S_0) = W_x + w(x)$. Weil $0 > \alpha_x - w(x) = \bar{W}_x - (W_x + w(x)) = \bar{W}_x - w(S_0)$, gibt es eine Menge S' ohne x mit $w(S') < w(S_0)$. Die minimale Menge ist also unter denen zu suchen, die x nicht enthalten.

Wir nennen $x \in X$ *unentschieden*, wenn $w(x) = \alpha_x$.

Es gilt $\mathbf{Pr}(x \text{ unentschieden}) \leq \frac{1}{2m}$, weil α_x und $w(x)$ unabhängig sind und $w(x)$ zufällig in $\{1, \ldots, 2m\}$ ist.

Mit der Vereinigungsschranke folgt: $\mathbf{Pr}(\exists x \in X : x \text{ unentschieden}) \leq \frac{|X|}{2m} \leq \frac{1}{2}$.

Nehmen wir nun an, kein x ist *nicht* unentschieden, d. h. alle x erfüllen $w(x) \neq \alpha_x$. Wir haben Folgendes gesehen:

(i) Wenn $w(x) < \alpha_x$, dann ist $x \in S$ für *jede* Menge S mit minimalem Gewicht.

(ii) $w(x) > \alpha_x$, dann ist $x \notin S$ für *jede* Menge S mit minimalem Gewicht.

Daraus ergibt sich: Es gibt genau eine Menge mit minimalem Gewicht, nämlich $S_0 = \{x \mid w(x) < \alpha_x\}$. □

Wir wenden das Isolationslemma auf die Menge $X = E$ und $\mathcal{F} = \{M \subseteq E \mid M \text{ ist perfektes Matching}\}$ an. Wir wählen also für jede Kante (i, j) in M ein zufälliges Gewicht w_{ij} aus $\{1, \ldots, 2m\}$, mit $m = |E|$. Dadurch bekommt jedes Matching M ein Gewicht $w(M) = \sum_{(i,j) \in M} w_{ij}$.

Nach dem Isolationslemma gibt es mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ nur ein einziges Matching M_0 mit minimalem Gewicht.

Nun wollen wir für jede Kante $(i,j) \in E$ eine Prozessorengruppe (unabhängig von den anderen) testen lassen, ob $(i,j) \in E$ gilt. Hierzu bilden wir zunächst aus T_G eine neue Matrix $B = (B_{ij})_{1 \le i,j \le n}$, indem wir X_{ij} in T_G durch $2^{w_{ij}}$ ersetzen. Das heißt:

$$B_{ij} = \left\{ \begin{array}{ll} 2^{w_{ij}}, & \text{falls } i < j \text{ und } (i,j) \in E, \\ -2^{w_{ji}}, & \text{falls } j < i \text{ und } (i,j) \in E, \\ 0, & \text{sonst.} \end{array} \right\}, \quad \text{für } 1 \le i,j \le n.$$

Wir betrachten $\det(B)$.

Lemma 6.11
Wenn der Gewichtssatz $(w_{ij})_{1 \le i,j \le n}$ ein eindeutig bestimmtes minimales Matching M_0 bestimmt, dann ist $\det(B) \neq 0$. Genauer: Für $w_0 = w(M_0)$ gilt: 2^{2w_0} ist Teiler von $\det(B)$, aber 2^{2w_0+1} ist kein Teiler von $\det(B)$.

(Aus dem Lemma ergibt sich, dass man aus $\det(B)$ den Wert w_0 ablesen kann.)

Beweis: Wir argumentieren mit denselben Strukturen wie im Beweis des Satzes von Tutte. Es sei $B = (B_{ij})_{1 \le i,j \le n}$. Dann gilt

$$\det(B) = \sum_{\sigma \in S_n} \operatorname{sign}(\sigma) \cdot B_{1,\sigma(1)} \cdots B_{n,\sigma(n)}. \tag{6}$$

Im Polynom $\det(T_G)$ heben sich die Terme t_σ, deren Spuren tr_σ mindestens einen ungeraden Kreis haben, gegenseitig weg. Das gilt dann auch für die Terme in (6), deren Permutationen σ Zyklen ungerader Länge enthalten.

Wir betrachten jetzt einen beliebigen Summanden $\operatorname{sign}(\sigma) \cdot B_{1,\sigma(1)} \cdots B_{n,\sigma(n)}$, für dessen Permutation σ die Spur tr_σ nur Kreise gerader Länge besitzt. Diese Kreise kann man stets in zwei (nicht notwendig disjunkte) Matchings M_1 und M_2 zerlegen (jede zweite Kante eines Kreises kommt in M_1, die anderen in M_2). Aus der Wahl der Faktoren B_{ij} folgt

$$\begin{aligned} &|B_{1,\sigma(1)} \cdots B_{n,\sigma(n)}| \\ &\quad = \prod_{(i,j) \in M_1} 2^{w_{ij}} \cdot \prod_{(i,j) \in M_2} 2^{w_{ij}} = 2^{\sum_{(i,j) \in M_1} w_{ij} + \sum_{(i,j) \in M_2} w_{ij}} = 2^{w(M_1)+w(M_2)}. \end{aligned}$$

Wenn $M_1 \neq M_0$ oder $M_2 \neq M_0$, dann ist $w(M_1) + w(M_2) > 2w_0$, also ist der Term zu σ durch 2^{2w_0+1} teilbar.

Es gibt in (6) nur einen Summanden mit geraden Kreisen, bei dem die Zerlegung der Kreise in zwei Matchings zu zwei Kopien von M_0 führt, und zwar ist das der Term zu der schon erwähnten Permutation σ_0 mit $\sigma_0(i) = j$ genau dann wenn $(i, j) \in M_0$ gilt. Der Term zu σ_0 hat Betrag 2^{2w_0}. Daher ist $\det(B)$ Summe von $\pm 2^{2w_0}$ und anderer Terme, die durch 2^{2w_0+1} teilbar sind. Die Behauptung des Lemmas folgt. □

Um zu entscheiden, ob eine Kante zu M_0 gehört oder nicht, muss man noch einen weiteren Trick anwenden. Zu Indexpaar (i, j), $i \neq j$, betrachten wir die „doppelte Streichungsmatrix" $B^{(i,j)}$, die aus B entsteht, indem man Zeilen i und j und Spalten i und j durch Nullen ersetzt; nur an der Kreuzung von Zeile i mit Spalte j bleibt B_{ij} stehen, an der Kreuzung von Zeile j mit Spalte i bleibt B_{ji} stehen.

Bei der Bildung der Determinante $\det(B^{(i,j)})$ bleiben dann nur diejenigen Terme t_σ stehen, für die $\sigma(i) = j$ und $\sigma(j) = i$ gilt. Das heißt:

$$\det(B^{(i,j)}) = \sum_{\substack{\sigma \in S_n \\ \sigma(i)=j \wedge \sigma(j)=i}} \operatorname{sign}(\sigma) \cdot B_{1,\sigma(1)} \cdots B_{n,\sigma(n)}.$$

Wie vorher argumentiert man, dass Beiträge von Permutationen mit Spuren, die ungerade Kreise enthalten, sich gegenseitig aufheben. Beiträge zu $\det(B^{(i,j)})$ kommen also nur von Summanden, die $\sigma(i) = j$ und $\sigma(j) = i$ erfüllen und daneben nur gerade Kreise enthalten. Wieder kann man jeden dieser Terme als $2^{w(M_1)+w(M_2)}$ schreiben, wobei sowohl M_1 als auch M_2 die Kante (i, j) enthalten. Nun gibt es zwei Fälle.

1. Fall: $(i, j) \notin M_0$. – Dann ist $\det(B^{(i,j)})$ Summe von Termen $2^{w(M_1)+w(M_2)}$ mit $M_1 \neq M_0$ oder $M_2 \neq M_0$, so dass $w(M_1) + w(M_2) > 2w_0$ gilt. Also ist $\det(B^{(i,j)})$ durch 2^{2w_0+1} teilbar.

2. Fall: $(i, j) \in M_0$. Dann ist $\det(B^{(i,j)})$ Summe von $\pm 2^{2w_0}$ und anderen Termen, die alle durch 2^{2w_0+1} teilbar sind. Also ist $\det(B^{(i,j)})$ nicht durch 2^{2w_0+1} teilbar.

Wir haben gezeigt:

Lemma 6.12
Sei M_0 eindeutig bestimmtes Matching mit minimalem Gewicht w_0. Dann gilt:

$$(i, j) \in M_0 \qquad \Leftrightarrow \qquad \frac{\det(B^{(i,j)})}{2^{2w_0}} \text{ ist ungerade.}$$

Es ergibt sich der folgende Algorithmus für die Ermittlung eines perfekten Matchings.

Algorithmus 6.6 ***Finde perfektes Matching in ungerichtetem Graphen***
Input: Graph $G = (V, E)$ mit n Knoten und m Kanten; G hat perfektes Matching
Aufgabe: Finde ein perfektes Matching in G.
Methode:

```
1   Für Kante (i,j), i < j, wähle zufällige Zahl w_ij aus {1,...,2m}; B_ij := 2^{w_ij};
2   Für Kante (i,j), i > j, setze B_ij := -B_ji;
3   Für (i,j) ∉ E setze B_ij := 0;
4   Bilde Matrix B = (B_ij)_{1≤i,j,≤n};
5   b := det(B);
6   ermittle maximales k so dass 2^k Teiler von b ist;
7   if k ungerade then return „Fehlschlag“;
9   für jede Kante (i,j) ∈ E tue folgendes: // parallele Ausführung möglich
10    B^(i,j) := doppelte Streichungsmatrix von B (wie beschrieben);
11    q_ij := det(B^(i,j))/2^k;
12  M := {(i,j) | q_ij ungerade};
13  if |M| = n/2 und jedes i ∈ V kommt in M vor
14    then return M else return „Fehlschlag“
```

Satz 6.13
Algorithmus 6.6 ist ein Las-Vegas-Algorithmus mit polynomieller Laufzeit, der zu gegebenem Graphen G mit perfektem Matching entweder ein perfektes Matching oder „Fehlschlag“ ausgibt. Die Wahrscheinlichkeit für „Fehlschlag“ ist durch $\frac{1}{2}$ beschränkt. Der Algorithmus ist parallelisierbar (polynomiell viele Prozessoren, $O((\log n)^3)$ Zeit).

Beweis: Zur Korrektheit: Der Test in Zeilen 13–14 führt dazu, dass auf keinen Fall eine Menge M ausgegeben wird, die kein perfektes Matching in G ist. In allen anderen Fällen wird „Fehlschlag“ ausgegeben. Es handelt sich also um einen Las-Vegas-Algorithmus. Wir müssen die Wahrscheinlichkeit für die Ausgabe „Fehlschlag“ abschätzen. Nach dem Isolationslemma 6.10 erzeugt der Gewichtssatz w_{ij} mit Wahrscheinlichkeit höchstens $\frac{1}{2}$ mehrere perfekte Matchings in G mit minimalem Gewicht. Dies trägt also Wahrscheinlichkeit höchstens $\frac{1}{2}$ zur Fehlerwahrscheinlichkeit bei. (Zum Beispiel könnte k ungerade sein, oder Zeilen 9–12 berechnen eine Menge, die kein perfektes Matching ist.) Nun nehmen wir an, dass es nur ein perfektes Matching M_0 mit minimalem Gewicht gibt. Nach Lemma 6.11 ist dann k gerade und $w_0 = w(M_0) = k/2$. Nach Lemma 6.12 landet Kante (i, j) genau dann in der Menge M, wenn $(i, j) \in M_0$ ist. Es wird also ein perfektes Matching ausgegeben, nämlich M_0.

Zur Rechenzeit: Die Berechnung einer Determinante benötigt $O(n^3)$ Additionen und Multiplikationen in $\mathbb{Z}$. Es müssen $m + 1$ viele Determinanten berechnet werden. Durch die Berechnungen können allerdings Zahlen bis zu einer Ziffernanzahl von

$O(n^2 m \log n)$ entstehen, daher ist eine einzelne Multiplikation bis zu $O((n^2 m \log n)^2)$ teuer. Die gesamte Rechenzeit ist groß, aber immer noch polynomiell.

Zur parallelen Berechnung: Determinanten von $n \times n$-Matrizen können in paralleler „Zeit“ $O((\log n)^2)$ berechnet werden; dabei wird eine Addition oder Multiplikation in $\mathbb{Z}$ als Elementaroperation gezählt. Mit den schon erwähnten sehr langen Zahlen muss man auch für diese Elementaroperationen parallele Verfahren einsetzen. Mit $O(\ell^2)$ Prozessoren lassen sich zwei ℓ-ziffrige Zahlen in Zeit $O(\log \ell)$ addieren und multiplizieren. Damit kann man das gesamte Verfahren aus Algorithmus 6.6 parallel implementieren, mit polynomiell vielen Prozessoren und Rechenzeit $O((\log n)^3)$. □

6.7 ∗ Äquivalenztest für Read-Once-Branchingprogramme

∗ Dieser Abschnitt ist **im Sommersemester 2021 nicht prüfungsrelevant**.

Wir schicken einige Bemerkungen zu azyklischen gerichteten Graphen („directed acyclic graphs“, „DAGs“) $G = (V, E)$ voraus. Knoten mit Ausgangsgrad 0 heißen *Senken*. Jeder DAG besitzt eine *topologische Sortierung* der Knoten, das ist eine lineare Anordnung der Knoten als $v_1, \ldots, v_N$, so dass für alle Kanten (v_k, v_ℓ) gilt: $k > \ell$. (Eine solche Anordnung erhält man, indem man Tiefensuche auf G ausführt und die Knoten nach *aufsteigenden fin-Nummern* sortiert, ähnlich wie in der AuD-Vorlesung im Kapitel „Tiefensuche“.) Diese Anordnung macht Konstruktionen und Beweise „durch Induktion über die Knotenanordnung“ möglich: Um eine Behauptung $\varphi(v)$ für alle Knoten $v \in V$ zu beweisen, beweist man $\varphi(v)$ für die Senken (Induktionsanfang) und dann beweist man, dass aus $\varphi(w)$ für alle Nachfolger w von v auch $\varphi(v)$ folgt. Entsprechend sehen Definitionen „durch Induktion über die Knotenanordnung“ aus.

Definition 6.14
Ein **Branchingprogramm** („Verzweigungsprogramm“, „Binäres Entscheidungsdiagramm“) $G = (V, E, s)$ für n Boolesche Variablen $x_1, \ldots, x_n$ ist ein azyklischer gerichteter Graph $G = (V, E)$, mit einem ausgezeichneten (Start-)Knoten $s \in V$ und folgenden weiteren Spezifikationen: (i) Jeder Knoten, der keine Senke ist („innerer Knoten“), ist mit einer Variablen x_i beschriftet und hat zwei Ausgangskanten, eine 0-Kante und eine 1-Kante. (ii) Jede Senke ist mit 0 oder mit 1 beschriftet. (Man kann auch, ohne das Modell zu ändern, alle 0-Senken identifizieren und alle 1-Senken identifizieren, so dass es nur eine oder zwei Senken gibt. Dann heißt die 0-Senke t_0 und die 1-Senke t_1.)

Bemerkung: Bei manchen (von Hand entworfenen) Branchingprogrammen wird s die einzige Quelle des Graphen (V, E) sein, also der einzige Knoten mit Eingangsgrad 0.

Dies wird in der Definition aber nicht gefordert. Man beachte auch, dass keineswegs alle Variablen $x_1, \ldots, x_n$ als Knotenmarkierungen in G vorkommen müssen.

Beispiel:

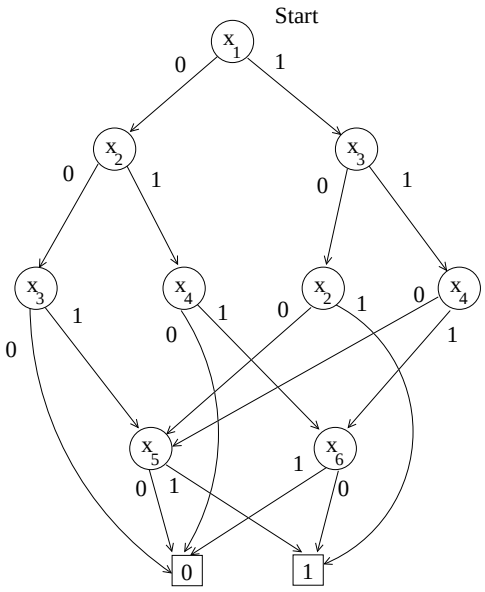


Abbildung 2: Ein Branchingprogramm mit Variablen $x_1, \ldots, x_6$. Der Startknoten ist markiert. Die Senken t_0 und t_1 sind als entsprechend markierte Quadrate dargestellt.

Definition 6.15
Ein Branchingprogramm $G = (V, E, s)$ über den Booleschen Variablen $x_1, \ldots, x_n$ definiert eine n-stellige Boolesche Funktion $f_G \colon \{0,1\}^n \to \{0,1\}$. Gegeben $a = (a_1, \ldots, a_n)$, wird $f(a)$ wie folgt durch das Ablaufen eines Wegs in G ermittelt: Starte in Knoten s. Wenn in Knoten v angekommen, wobei v ein innerer Knoten mit Beschriftung x_i ist, folge der a_i-Kante aus v zum nächsten Knoten. Wenn eine Senke t_b mit $b \in \{0,1\}$ erreicht wurde, ist $f(a) = b$.

Bemerkung: Im Extremfall besteht G nur aus einer Senke t_b. Diese ist dann auch der Startknoten, und die berechnete Funktion f_G ist die Konstante b.

Eine (offensichtlich äquivalente) einfache Beschreibung der berechneten Funktion f_G ist die folgende: Gegeben sei Input a. Wenn der innere Knoten v mit der Variablen x_i beschriftet ist, dann lösche die $\overline{a}_i$-Kante aus v. Auf diese Weise bleibt ein einziger Weg mit Startknoten s übrig. Dieser endet in einer Senke t_b. Dann ist $f_G(a) = b$.

Beispiel:

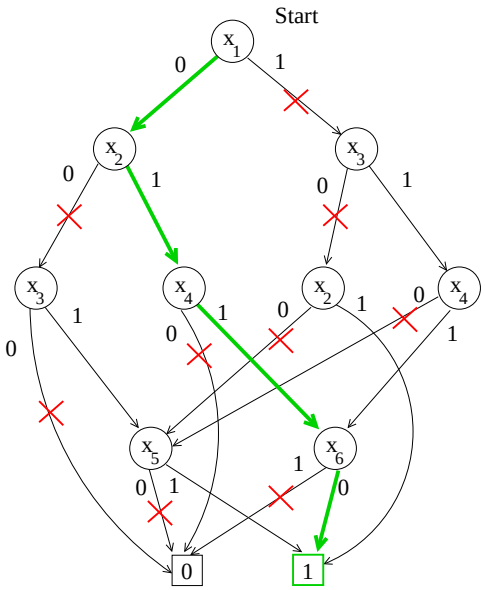


Abbildung 3: Im Branchingprogramm aus Abb. 2 wurden passend zum Input $a = (0, 1, 1, 1, 1, 0)$ Kanten gestrichen; ein Weg vom Startknoten zur 1-Senke bleibt übrig (grün). Dieser Weg würde auch zur Eingabe $a' = (0, 1, 0, 1, 0, 0)$ passen, da die Werte a_3 und a_5 gar nicht abgefragt werden. Es gilt also $f_G(a) = f_G(a') = 1$.

Hilfreich ist aber noch eine andere Beschreibung von f_G. Hierzu definieren wir eine ganze Menge von Funktionen, eine für jeden Knoten $v \in V$: f_v ist diejenige Boolesche Funktion, die von G berechnet wird, wenn man v zum Startknoten erklärt. Für einen festen Input a hängen die Funktionswerte dann auf die folgende Weise zusammen:

(i) Wenn $v = t_b$ eine b-Senke ist, dann ist $f_v(a) = b$.

(ii) Wenn v ein innerer Knoten ist, mit 0-Nachfolger v_0 (also entlang der 0-Kante) und 1-Nachfolger v_1 (entlang der 1-Kante), und wenn v mit x_i markiert ist, dann ist

$$f_v(a) = (a_i \wedge f_{v_1}(a)) \vee (\overline{a}_i \wedge f_{v_0}(a)).$$

(Ein Beweis durch Induktion über die Knotenanordnung zeigt die Korrektheit dieser Beschreibung. Festlegung (i) ist offensichtlich korrekt. Wenn die Rechnung im inneren x_i-Knoten v startet und $a_i = 0$ ist, geht man zum 0-Nachfolger v_0 und wertet G von

dort aus weiter mit a aus. Dies liefert nach I.V. gerade den Wert $f_{v_0}(a)$. Wenn $a_i = 1$ ist, erhält man den Wert $f_{v_1}(a)$. Genau dies wird durch die Formel in (ii) ausgedrückt, die „*if* a_i *then* $f_{v_1}(a)$ *else* $f_{v_0}(a)$“ bedeutet.)

Es ergibt sich die folgende etwas abstraktere Beschreibung von f_G, die mit ganzen Booleschen Funktionen operiert. Durch Induktion über die Knotenanordnung können wir für jeden Knoten $v \in V$ die Funktion $f_v \colon \{0,1\}^n \to \{0,1\}$ kompakt beschreiben.

(i) Wenn $v = t_b$ eine b-Senke ist, dann ist f_v die konstante b-Funktion.

(ii) Wenn v ein innerer Knoten ist, mit 0-Nachfolger v_0 und 1-Nachfolger v_1, und wenn v mit x_i markiert ist, dann ist

$$f_v = (x_i \wedge f_{v_1}) \vee (\overline{x}_i \wedge f_{v_0}).$$

Dann ist $f_G = f_s$ für den Startknoten s.

Beispiel: In Abb. 4 ist unser Beispiel-BP mit Knotennamen A, B, C, D, E, F, H,

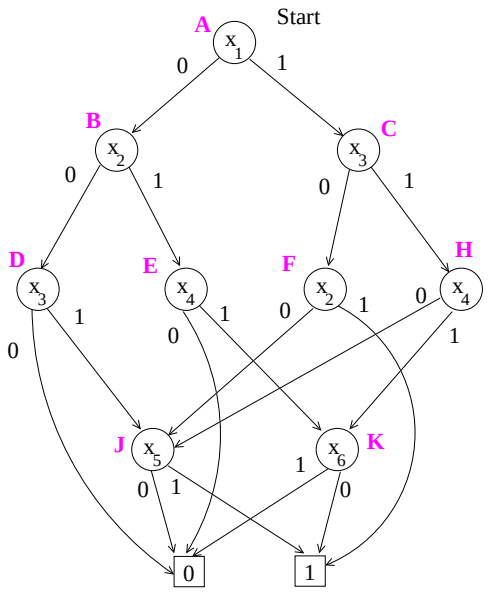


Abbildung 4: Das Branchingprogramm aus Abb. 2 mit Knotennamen A, B, C, D, E, F, H, J, K.

J, K dargestellt. Wir haben, aufgeschrieben in der Reihenfolge einer topologischen

Anordnung:[9]

$$
\begin{aligned}
f_{\mathrm{K}} &= \overline{x}_6 \\
f_{\mathrm{J}} &= x_5 \\
f_{\mathrm{H}} &= x_4\overline{x}_6 \vee \overline{x}_4 x_5 \\
f_{\mathrm{F}} &= (x_2 \wedge 1) \vee \overline{x}_2 x_5 = x_2 \vee \overline{x}_2 x_5 \\
f_{\mathrm{E}} &= x_4\overline{x}_6 \vee (\overline{x}_4 \wedge 0) = x_4\overline{x}_6 \\
f_{\mathrm{D}} &= x_3 x_5 \vee (\overline{x}_3 \wedge 0) = x_3 x_5 \\
f_{\mathrm{C}} &= x_3 f_{\mathrm{H}} \vee \overline{x}_3 f_{\mathrm{F}} = x_3(x_4\overline{x}_6 \vee \overline{x}_4 x_5) \vee \overline{x}_3(x_2 \vee \overline{x}_2 x_5) = \ldots \\
f_{\mathrm{B}} &= x_2 f_{\mathrm{E}} \vee \overline{x}_2 f_{\mathrm{D}} = \ldots \\
f_{\mathrm{A}} &= f_G = x_1 f_{\mathrm{C}} \vee \overline{x}_1 f_{\mathrm{B}} = \ldots
\end{aligned}
$$

Fakt 6.16
Zu jeder Booleschen Funktion $g\colon \{0,1\}^n \to \{0,1\}$ gibt es ein Branchingprogramm G_g mit $g = f_{G_g}$. Man kann sogar zusätzlich fordern, dass die Variablen in der Reihenfolge $x_1, \ldots, x_n$ gelesen werden.

Der *Beweis* ist nicht schwer. Man legt einen vollständigen Binärbaum mit n Levels von inneren Knoten an, wobei die 2^{i-1} Knoten auf Level i mit x_i markiert sind. Der x_1-Knoten auf Level 1 ist der Startknoten. Für $1 \leq i < n$ hat jeder x_i-Knoten ein 0-Kind und ein 1-Kind auf Level $i+1$. Die x_n-Knoten haben die Senken t_0 und t_1 als mögliche Nachfolger. Diese Nachfolger können so eingestellt werden, dass Eingabe $a \in \{0,1\}^n$ zum Erreichen der $g(a)$-Senke $t_{g(a)}$ führt. Das Branchingprogramm hat $2^n + 1$ Knoten ($2^n - 1$ innere Knoten und zwei Senken.)

Der Nachteil des beschriebenen allgemeinen Branchingprogramms ist natürlich, dass es sehr groß ist. Manche n-stellige Funktionen haben auch nicht allzu große Branchingprogramme. Eine interessante Eigenschaft dieser Branchingprogramme ist, dass auf jedem Berechnungsweg jede Variable höchstens einmal vorkommt. Wir interessieren uns besonders für solche.

Definition 6.17
Ein **Read-Once-Branchingprogramm** (Read-Once-BP) $G = (V, E, s)$ ist ein Branchingprogramm, das bei der Berechnung des Wertes für eine beliebige Eingabe a das Bit a_i höchstens einmal liest. (Äquivalent ist: Auf jedem Weg vom Startknoten zu einer der Senken kommt jede Variable maximal einmal als Beschriftung vor.)

[9] Der Konvention folgend, schreiben wir das Zeichen $\wedge$ meist nicht. „$x_2 f_{\mathrm{E}}$“ bedeutet also „$x_2 \wedge f_{\mathrm{E}}$“.

In diesem Abschnitt betrachten wir einen (Nicht-)Äquivalenztest für Read-Once-BPe. Das bedeutet, dass zu zwei gegebenen Read-Once-BPen G und G' zu entscheiden ist, ob sie dieselbe Funktion darstellen. Wieso ist das interessant? Man stelle sich eine Situation vor, in der zwei Darstellungen für eine Funktion gegeben sind: eine als Spezifikation, die andere als (neu entworfener) Schaltkreis, oder zwei Darstellungen als Schaltkreis, einer alt, klassisch erprobt, und der andere ein neuer Entwurf. Angenommen, wir haben Algorithmen, die solche Darstellungen in Branchingprogramme transformieren, sogar in Read-Once-BPe. Um zu testen, ob der neue Entwurf korrekt ist, müssen wir herausfinden, ob die beiden so erzeugten Read-Once-BPe dieselbe Funktion darstellen. (Ansätze dieser Art sind die einfachsten Bausteine in der Hardwareverifikation.)

Definition 6.18
Zwei Branchingprogramme G und G' für dieselbe Variablenmenge $x_1, \ldots, x_n$ heißen *äquivalent*, in Zeichen $G \sim G'$, wenn $f_G = f_{G'}$ gilt.

Das *Äquivalenzproblem für Read-Once-Branchingprogramme* besteht darin, zu zwei vorgelegten Read-Once-BPen G und G' zu entscheiden, ob $G \sim G'$ gilt oder nicht. Wir präsentieren im Folgenden einen Monte-Carlo-Algorithmus mit einseitigem Fehler für das (Nicht-)Äquivalenzproblem. (Es sei angemerkt, dass man keinen deterministischen Polynomialzeitalgorithmus für dieses Problem kennt.)

Die Grundidee ist „Arithmetisierung der Logik“. Gegeben sein ein beliebiger Körper $K = (K, +, \cdot, 0, 1)$. Wie kann man die logischen Operationen $\wedge$, $\vee$ und $\neg$ durch Operationen über dem Körper K so darstellen, dass auf dem Definitionsbereich $\{0,1\} \subseteq K$ derselbe Effekt entsteht? Klar: $b \wedge c = b \cdot c$ und $\neg b = 1 - b$, für $b, c \in \{0,1\}$. Weiter erhält man mit den deMorgan-Regeln: $b \vee c = 1 - (1-b) \cdot (1-c) = 1 - b - c + b \cdot c$, für $b, c \in \{0,1\}$. Für unsere Zwecke wichtig ist noch die *Fallunterscheidung* mit 3 Bits:

$$\textit{if } b \textit{ then } c \textit{ else } d \text{ oder } (b \wedge c) \vee (\overline{b} \wedge d),$$

die sich für $b, c, d \in \{0,1\}$ durch $b \cdot c + (1-b) \cdot d$ darstellen lässt.

Bemerkung: Allgemeiner kann man jede Boolesche Funktion auf n Variablen $x_1, \ldots, x_n$ als Polynom über K mit Variablen $X_1, \ldots, X_n$ darstellen. Beispiele sind:

- $x_1 \wedge x_2$ wird durch $X_1 X_2$ dargestellt.
- $\neg x_1$ wird durch $1 - X_1$ dargestellt.
- $x_1 \vee x_2$ wird durch $X_1 + X_2 - X_1 X_2$ dargestellt.

- $x_1 \oplus x_2$ (exklusives oder) wird durch $X_1(1-X_2)+(1-X_1)X_2 = X_1+X_2-2X_1X_2$ dargestellt.
- $x_1 \oplus x_2 \oplus x_3$ wird durch

  $$X_1 + X_2 + X_3 - 2X_1X_2 - 2X_1X_3 - 2X_2X_3 + 4X_1X_2X_3$$

  dargestellt. (Wenn keines oder zwei der Eingabebits 1 sind, ergibt sich 0, wenn genau ein Eingabebit 1 ist, ergibt sich 1, wenn alle drei 1 sind, ergibt sich $1 + 1 + 1 - 2 - 2 - 2 + 4 = 6$. (Überlegen Sie: Was ist die Darstellung von $x_1 \oplus \cdots \oplus x_5$, von $x_1 \oplus \cdots \oplus x_n$?)
- *if* x_1 *then* x_2 *else* x_3 wird durch $X_1X_2 + (1 - X_1)X_3 = X_3 + X_1X_2 - X_1X_3$ dargestellt.

Was sollen die Koeffizienten 2 und 4 in den Polynomen in einem beliebigen Körper bedeuten? Nun, ganze Zahlen $k \in \mathbb{Z}$ können in K auf natürliche Weise interpretiert werden, als $\hat{k} \in K$:

$$\hat{k} = \underbrace{1 + \cdots + 1}_{k\text{-mal, Addition in } K} \quad \text{für } k \geq 1;$$

dann ist $\hat{k}$ als additives Inverses von $\widehat{-k}$ auch für negative $k \in \mathbb{Z}$ definiert.

Zu einem gegebenen BP G mit n Booleschen Variablen definieren wir nun, durch Induktion über die Knotenanordnung, n-stellige Polynome p_v über Variablen $X_1, \ldots, X_n$ mit Koeffizienten aus K, die sich auf $\{0,1\}^n$ wie die oben diskutierten Funktionen f_v verhalten sollen. Ein BP G mit Senken t_0 und t_1 und Startknoten s sei gegeben.

(i) p_{t_0} ist das Nullpolynom; p_{t_1} ist das konstante Polynom 1.

(ii) Wenn v ein innerer Knoten ist, mit Beschriftung x_i, mit 0-Nachfolger v_0 und 1-Nachfolger v_1, dann ist

$$p_v := X_i \cdot p_{v_1} + (1 - X_i) \cdot p_{v_0}.$$

(iii) p_G ist p_s, für den Startknoten s von G.

Lemma 6.19
Für $a \in \{0,1\}^n$ gilt $f_G(a) = p_G(a)$.

(Man beachte, dass in dem Ausdruck $f_G(a)$ die Eingabebits eigentlich Wahrheitswerte darstellen, anhand derer Entscheidungen gefällt werden, in dem Ausdruck $p_G(a)$ dagegen die Elemente 0 und 1 aus K, mit denen gerechnet wird.)

Beweis (Idee): Man zeigt durch Induktion über die Knotenanordnung, dass $p_v(a) = f_v(a)$ gilt, für alle $a \in \{0,1\}^n$. Dies ist mit der oben diskutierten induktiven Definition von f_v reine Routine. □

Nun ist der Plan für unseren Äquivalenztest klar: Wenn G und G' gegeben sind, testen wir mit dem schon bekannten Ansatz aus den vorherigen Abschnitten, ob $p_G = p_{G'}$ gilt oder nicht, und schließen dann auf f_G und $f_{G'}$. Es sind allerdings noch zwei Dinge zu klären:

- Nach Lemma 6.19 gilt: $p_G = p_{G'} \Rightarrow f_G = f_{G'}$. Aber gilt auch die Umkehrung $f_G = f_{G'} \Rightarrow p_G = p_{G'}$? Wenn es vorkommen könnte, dass $f_G = f_{G'}$ gilt, aber $p_G \neq p_{G'}$, würde der Plan sofort fehlschlagen: Wir würden die Polynome auf Identität testen und mit großer Wahrscheinlichkeit Ungleichheit feststellen, obwohl die Booleschen Funktionen gleich sind. Man denke etwa an die (verschiedenen) Polynome $X_1 - X_1^2 X_2$ und $X_1^2 - X_1 X_2$, die beide die Boolesche Funktion $x_1 \overline{x}_2$ berechnen.
- Wie kann man p_G effizient auswerten?

Wir arbeiten diese Punkte nacheinander ab. Zuerst stellen wir fest, dass die Read-Once-Eigenschaft dazu führt, dass Polynome wie $X_1 - X_1^2 X_2$ nicht als p_G auftreten können.

Definition 6.20
Ein Polynom $p(X_1, \ldots, X_n)$ heißt *multilinear*, wenn in jedem Term $a_{\ell_1,\ldots,\ell_n} X_1^{\ell_1} \cdots X_n^{\ell_n}$ von p die Exponenten $\ell_1, \ldots, \ell_n$ aus der Menge $\{0,1\}$ kommen.

Beispiel: $5X_1X_2X_4 + 2X_3 + 1$ ist multilinear, $X_1^2 + X_1X_2$ nicht.

Lemma 6.21
Wenn G ein Read-Once-BP mit Variablen $x_1, \ldots, x_n$ ist, dann ist p_G *multilinear* (und hat daher Grad höchstens n).

Beweis: Wir zeigen durch Induktion über die Knotenreihenfolge, dass alle p_v multilinear sind. (Daraus folgt, dass $p_G = p_s$ multilinear ist.) Die Polynome $p_{t_0} = 0$ und $p_{t_1} = 1$ enthalten überhaupt keine Variable. Nun sei v ein innerer Knoten mit 0-Nachfolger v_0 und 1-Nachfolger v_1, der mit x_i beschriftet ist. Aus der Read-Once-Eigenschaft folgt, dass in dem Bereich des Graphen G, der von v_0 und v_1 aus erreichbar ist, die Boolesche Variable x_i nirgends als Beschriftung vorkommen kann. Daher enhalten p_{v_0} und p_{v_1} die Variable X_i nicht. Nach Induktionsvoraussetzung sind p_{v_0} und p_{v_1}

multilinear. Daraus folgt durch Ausmultiplizieren, dass auch $p_v = (1 - X_i)p_{v_0} + X_i p_{v_1}$ keine Variable mit einem Exponenten größer als 1 enthält, also multilinear ist. □

Die nächste Feststellung ist dann zentral: Für *multilineare* Polynome p und p' über einem beliebigen Körper K folgt aus gleichem Werteverlauf auf Inputs in $\{0,1\}^n \subseteq K$, dass sie identisch sind. Dies ergibt sich sofort aus dem folgenden Lemma (anzuwenden auf das Polynom $p - p'$).

Lemma 6.22
Es sei K ein Körper und p ein **multilineares** Polynom mit Variablen $X_1, \ldots, X_n$ und Koeffizienten aus K. Dann gilt: Wenn $p(a_1, \ldots, a_n) = 0$ für alle $a_1, \ldots, a_n \in \{0,1\}$, dann ist p das Nullpolynom.

Beweis: Wir zeigen durch Induktion über die Anzahl m der Variablen aus $X_1, \ldots, X_n$, die in p tatsächlich vorkommen: Wenn p nicht das Nullpolynom ist, dann gibt es ein $a \in \{0,1\}^n$ mit $p(a) \neq 0$.

I.A.: Sei $m = 0$. Dann ist p eine Konstante $c \in K$. Weil p nicht das Nullpolynom ist, gilt $c \neq 0$. Daraus folgt $p(a) = c \neq 0$ für alle $a \in \{0,1\}^n$.

I.V.: $m \geq 1$ und die Behauptung stimmt für alle $m' < m$.

I.Schritt: Angenommen, in p kommen $m \geq 1$ viele Variablen vor. Wir wählen eine davon, etwa X_1. Wir klammern X_1 aus allen Termen aus, in denen es vorkommt, und erhalten die Darstellung

$$p(X_1, \ldots, X_n) = X_1 \cdot q(X_2, \ldots, X_n) + r(X_2, \ldots, X_n).$$

(Höhere Potenzen von X_1 gibt es nicht, weil p multilinear ist.) Dabei ist q ein multilineares Polynom mit Variablen $X_2, \ldots, X_n$, das nicht das Nullpolynom ist, und in dem strikt weniger Variablen tatsächlich vorkommen als in p. Nach I.V. gibt es $(a_2, \ldots, a_n) \in \{0,1\}^{n-1}$ mit $q(a_2, \ldots, a_n) \neq 0$. Die Werte $p(0, a_2, \ldots, a_n) = r(a_2, \ldots, a_n)$ und $p(1, a_2, \ldots, a_n) = q(a_2, \ldots, a_n) + r(a_2, \ldots, a_n)$ sind also verschieden, und wir können $a_1 \in \{0,1\}$ mit $p(a_1, a_2, \ldots, a_n) \neq 0$ wählen. □

Der randomisierte Algorithmus zum Vergleich von f_G und $f_{G'}$, mit Variablen $x_1, \ldots, x_n$, verläuft nun im Prinzip wie folgt:

1. Wähle Primzahl $p > 2n$.
2. Wähle zufällig $a_1, \ldots, a_n$ aus $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, setze $a = (a_1, \ldots, a_n)$.
3. Werte über $\mathbb{Z}_p$ aus: $y \leftarrow p_G(a)$ und $z \leftarrow p_{G'}(a)$.
4. Die Ausgabe ist $[y \neq z]$.

Das Verhalten dieses Algorithmus ist wie folgt:

1. Fall: $f_G \neq f_{G'}$. – Nach Lemma 6.19 gilt dann $p_G \neq p_{G'}$. Nach dem Satz von Schwartz-Zippel hat $h = p_G - p_{G'}$ in $\mathbb{Z}_p$ maximal $\deg(p_G - p_{G'}) \cdot p^{n-1}$ Nullstellen. Da der Grad von p_G und $p_{G'}$ höchstens n ist, sind dies höchstens $n \cdot p^{n-1}$ viele. Also ist die Wahrscheinlichkeit, dass in Zeile 2 zufällig ein Element a mit $p_G(a) = p_{G'}(a)$, also eine Nullstelle von h, gewählt wird, höchstens $n/p < \frac{1}{2}$. Damit: $\mathbf{Pr}(\text{Ausgabe ist } 0) < \frac{1}{2}$.

2. Fall: $f_G = f_{G'}$. Nach Lemma 6.21 sind p_G und $p_{G'}$ multilineare Polynome. Mit Lemma 6.22 folgt $p_G = p_{G'}$. Daher liefert der Algorithmus immer den Wert 0 zurück.

Wir haben es also mit einem Monte-Carlo-Algorithmus mit einseitigem Fehler zu tun, mit Fehlerschranke $\frac{1}{2}$.

Die einzige verbleibende Frage ist, wie man zu einem gegebenen Branchingprogramm G und einer Eingabe $a \in \mathbb{Z}_p^n$ den Wert $p_G(a)$ effizient berechnet. Es ist nicht möglich, das Polynom explizit aufzuschreiben, da es im Normalfall viel zu viele Terme haben wird. (Zum Beispiel hat die Funktion $x_1 \oplus \cdots \oplus x_n$ ein sehr kleines Branchingprogramm, aber sein Polynom hat $2^n - 1$ Terme.) Die Idee ist, die Werte $r_v = p_v(a)$ durch Induktion über die Knotenreihenfolge in G auszurechnen. Es sei $a = (a_1, \ldots, a_n) \in \mathbb{Z}_p^n$ der Input. Das Branchingprogramm G sei gegeben. Wir rechnen:

(i) $r_{t_0} \leftarrow 0$ und $r_{t_1} \leftarrow 1$ (in $\mathbb{Z}_p$).

(ii) Wenn v ein innerer Knoten ist, mit Beschriftung x_i, mit 0-Nachfolger v_0 und 1-Nachfolger v_1, dann setze

$$r_v \leftarrow (a_i \cdot r_{v_1} + (1 - a_i) \cdot r_{v_0}) \bmod p.$$

(iii) Die Ausgabe ist r_s.

Man zeigt leicht durch Induktion über die Knotenreihenfolge, dass $r_v = p_v(a)$ gilt, und dass daher die Ausgabe r_s tatsächlich gleich $p_s(a) = p_G(a)$ ist. Der Aufwand ist $O(|V|)$ Operationen in $\mathbb{Z}_p$, für die Knotenmenge V von G.

Bemerkung: Man nimmt mit Erstaunen zur Kenntnis, dass der Äquivalenztest eine Berechnung in $\mathbb{Z}_p$ durchführt, obwohl die Problemstellung keine Zahlen erwähnt und G überhaupt nicht für die Benutzung mit Zahlen „gedacht“ ist, sondern als Darstellung einer Booleschen Funktion.