2 Erste Beispiele

2.1 Ein randomisierter Identitätstest

Im folgenden geht es um die Aufgabe, die Gleichheit zweier „großer“ Objekte zu überprüfen. Dabei werden die Objekte nicht in ihrer gesamten Struktur verglichen, sondern es werden eine Art „Fingerabdrücke“ der Objekte berechnet und miteinander verglichen.

Im Unterschied zur Anwendung in der Kriminalistik kann man aber nur sicher sein, dass verschiedene Fingerabdrücke von verschiedenen Objekten stammen. Gleichheit von Fingerabdrücken wird zwar auch zum Anlass genommen, die Gleichheit der Objekte zu vermuten, dies kann aber (zum Beispiel mit einer Wahrscheinlichkeit von $1/4$) falsch sein.

2.1 Gegeben seien drei $n \times n$ Matrizen $\mathbf{A}$, $\mathbf{B}$ und $\mathbf{C}$. Die Aufgabe bestehe darin, festzustellen, ob $\mathbf{AB} = \mathbf{C}$ ist.

Die Einträge der Matrizen mögen aus einem Körper $\mathbb{F}$ stammen. Dessen neutrale Elemente bzgl. Addition bzw. Multiplikation seien wie üblich mit 0 resp. 1 bezeichnet.

2.2 Die Aufgabe ist natürlich ganz einfach zu lösen, indem man $\mathbf{AB}$ ausmultipliziert und das Ergebnis mit $\mathbf{C}$ vergleicht. Dafür benötigt man bei Benutzung des besten derzeit bekannten Multiplikationsalgorithmus $\Theta(n^{2.376\cdots})$ Schritte (Coppersmith und Winograd 1990).

Wir wollen im folgenden eine probabilitische Methode von Freivalds (1977) betrachten, die in Zeit $O(n^2)$ mit Wahrscheinlichkeit $1 - 2^{-k}$ die richtige Antwort liefert.

2.3 Algorithmus.

⟨*Mit einem „Bit“ ist einer der Werte* 0 *oder* 1 *gemeint.*⟩
$\mathbf{r} \leftarrow$ ⟨*Vektor von* n *unabhängigen Zufallsbits*⟩
$\mathbf{x} \leftarrow \mathbf{Br}$
$\mathbf{y} \leftarrow \mathbf{Ax}$
$\mathbf{z} \leftarrow \mathbf{Cr}$
if $(\mathbf{y} \neq \mathbf{z})$ **then**
    return NO
else
    return YES
fi

Der Zeitbedarf dieses Algorithmus ist offensichtlich in $O(n^2)$.

2.4 Freivalds' Algorithmus überprüft also statt der Gleichheit $\mathbf{AB} = \mathbf{C}$ nur, ob $\mathbf{A}(\mathbf{Br}) = \mathbf{Cr}$ für einen zufällig gewählten Vektor $\mathbf{r}$ gilt. Die Werte $\mathbf{y} = \mathbf{A}(\mathbf{Br})$ bzw. $\mathbf{z} = \mathbf{Cr}$ werden sozusagen als *Fingerabdrücke* von $\mathbf{AB}$ resp. $\mathbf{C}$ benutzt.

2.5 Lemma. Es seien $\mathbf{A}$, $\mathbf{B}$ und $\mathbf{C}$ drei $n \times n$ Matrizen mit $\mathbf{AB} \neq \mathbf{C}$ und $\mathbf{r}$ sei ein Vektor von n Bits, die unabhängig und gleichverteilt gewählt wurden. Dann ist $\mathbf{P}(\mathbf{ABr} = \mathbf{Cr}) \leqslant 1/2$.

2.6 Beweis. Es sei $\mathbf{D} = \mathbf{AB} - \mathbf{C}$, also $\mathbf{D} \neq \mathbf{0}$. Außerdem setzen wir $\mathbf{y} = \mathbf{ABr}$ und $\mathbf{z} = \mathbf{Cr}$. Dann gilt $\mathbf{y} = \mathbf{z}$ genau dann, wenn $\mathbf{Dr} = \mathbf{0}$.

Es bezeichne $\mathbf{d} = (d_1, \ldots, d_n)$ eine Zeile von $\mathbf{D}$, die nicht nicht der Nullvektor ist, und es sei k der größte Index, für den $d_k \neq 0$. Eine untere Schranke für die Wahrscheinlichkeit, dass $\mathbf{dr} \neq 0$ ist, ist auch eine untere Schranke für die Wahrscheinlichkeit, dass $\mathbf{Dr} \neq \mathbf{0}$ ist.

Wir suchen eine obere Schranke für die Wahrscheinlichkeit, dass $\mathbf{dr} = 0$ ist. Das ist genau dann der Fall, wenn $\sum_{i=1}^{n} d_i r_i = 0$, also genau dann, wenn $r_k = (-\sum_{i \neq k} d_i r_i)/d_k$. Um einzusehen, wie unwahrscheinlich dies ist, stelle man sich vor, $r_1, \ldots, r_{k-1}$ seien bereits gewählt. Die Wahl der r_i mit $k < i \leqslant n$ hat keinen Einfluss, da die entsprechenden d_i alle gleich 0 sind. Dann kann offensichtlich für höchstens *einen* der beiden möglichen Werte für r_k $\mathbf{dr} = 0$ sein. Also ist diese Wahrscheinlichkeit höchstens $1/2$.

Mit einer Wahrscheinlichkeit größer gleich $1/2$ ist folglich $\mathbf{dr} \neq 0$ und damit auch $\mathbf{Dr} \neq \mathbf{0}$. ∎

2.7 Korollar. Wenn $\mathbf{AB} = \mathbf{C}$ ist, liefert Algorithmus 2.3 stets die richtige Antwort. Wenn $\mathbf{AB} \neq \mathbf{C}$ ist, liefert Algorithmus 2.3 mit einer Wahrscheinlichkeit größer gleich $1/2$ die richtige Antwort.

2.8 Eine Fehlerwahrscheinlichkeit von $1/2$ ist sehr groß. Wie kann man sie kleiner machen?

Eine Möglichkeit findet man beim nochmaligen genauen Lesen des vorletzten Abschnittes von Beweis 2.6. Der Nenner 2 ergab sich nämlich aus der Tatsache, dass für die Komponenten von $\mathbf{r}$ jeweils zufällig einer von *zwei* Werten gewählt wurde. Enthält der Körper $\mathbb{F}$ aber mindestens 2^k Elemente, aus denen man gleichverteilt und unabhängig wählt, dann sinkt entsprechend die Fehlerwahrscheinlichkeit auf 2^{-k}.

Eine andere Möglichkeit, die Fehlerwahrscheinlichkeit zu drücken ist allgemeiner und auch bei anderen Gelegenheiten anwendbar: Man führt k unabhängige Wiederholung des Algorithmus durch und produziert nur dann am Ende die Antwort yes, wenn jeder Einzelversuch diese Antwort geliefert hat. Damit kann die Fehlerwahrscheinlichkeit von $1/2$ auf 2^{-k} (allgemein von p auf p^k) gedrückt werden.

Warum ist das so?

Es seien $Y_1, Y_2, \ldots, Y_k$ die Zufallsvariablen für die Antworten bei den einzelnen Versuchen und es sei $\mathbf{AB} \neq \mathbf{C}$. Dann ist die Wahrscheinlichkeit für eine falsche Antwort des Algorithmus

$$\mathbf{P}(Y_1 = \text{yes} \wedge Y_2 = \text{yes} \wedge \cdots \wedge Y_k = \text{yes}) \quad \text{und das ist gleich} \quad \prod_{i=1}^{k} \mathbf{P}(Y_i = \text{yes})$$

wegen der Unabhängigkeit der Y_i.

Wir merken noch an, dass bei beiden eben diskutierten Möglichkeiten zur Senkung der Fehlerwahrscheinlichkeit die gleiche Anzahl von Zufallsbits benötigt wird. Diese Zahl wächst proportional mit k. Alternativen, wie man das gleiche Ergebnisse erreichen kann, aber dabei unter Umständen deutlich weniger Zufallsbits benötigt, sind Gegenstand eines späteren Kapitels der Vorlesung.

2.2 Vergleich von Wörtern

Wir wollen noch zwei ähnlich gelagerte Probleme betrachten, bei denen es darauf ankommt, Wörter miteinander zu vergleichen. Der bequemeren Notation wegen werden wir uns auf Bitfolgen beschränken. Man mache sich aber als Übungsaufgabe klar, dass die Algorithmen und ihre Analysen für den Fall größerer Alphabete angepasst werden können.

Zunächst betrachten wir die wohl einfachste Aufgabe, den Test zweier Folgen auf Gleichheit.

2.9 Gegeben seien zwei „Datenbestände“, die als gleich lange Bitfolgen (also Wörter) $a_1 \cdots a_n$ und $b_1 \cdots b_n$ repräsentiert sind.

Die Aufgabe besteht darin herauszufinden, ob die beiden Bitfolgen gleich sind. Interpretiert man sie auf die naheliegende Weise als Zahlen $a = \sum_{i=1}^{n} a_i 2^{i-1}$ und $b = \sum_{i=1}^{n} b_i 2^{i-1}$, muss also festgestellt werden, ob $a = b$ ist.

2.10 Man stelle sich zum Beispiel vor, dass a und b an verschiedenen Orten A und B vorliegen. Die Aufgabe bestehe darin, Gleichheit festzustellen, und dabei möglichst wenige Bits zu übertragen.

Es ist klar, dass man diese Aufgabe deterministisch lösen kann, indem man n Bits überträgt. Im folgenden soll ein probabilistischer Algorithmus analysiert werden, bei dem nur $O(\log n)$ Bits übertragen werden und der mit einer Fehlerwahrscheinlichkeit kleiner gleich $1/n$ die Aufgabe löst.

2.11 Algorithmus.

⟨*Überprüfung, ob Bitfolgen* $a_1 \cdots a_n$ *und* $b_1 \cdots b_n$ *gleich sind*⟩
$p \leftarrow$ ⟨*Primzahl; zufällig gleichverteilt aus denen kleiner oder gleich* $n^2 \ln n^2$ *ausgewählt.*⟩
$a \leftarrow \sum_{i=1}^{n} a_i 2^{i-1}$
$b \leftarrow \sum_{i=1}^{n} b_i 2^{i-1}$
if $(a \bmod p = b \bmod p)$ **then**
    return YES
else
    return NO
fi

Bevor wir den Algorithmus analysieren, seien zwei Dinge angemerkt. Die Aufgabe, ein geeignetes p auszuwählen, ist ebenfalls nicht ganz leicht. Überlegen Sie sich etwas!

Für eine Primzahl p bezeichne $F_p(x) : \mathbb{Z} \to \mathbb{Z}_p$ die Abbildung $x \mapsto x \bmod p$. Algorithmus 2.11 überprüft, ob $F_p(a) = F_p(b)$ ist, und liefert genau dann eine falsche Antwort, wenn das der Fall ist obwohl $a \neq b$ ist. Das ist genau dann der Fall, wenn p ein Teiler von $c = a - b$ ist.

Für die Anwendung des Algorithmus in dem in Punkt 2.10 skizzierten Szenario wird man z. B. am Ort A sowohl p „würfeln“ als auch den Wert $F_p(a)$ berechnen und beides zu B übertragen, um dort $F_p(b)$ zu berechnen und mit $F_p(a)$ zu vergleichen. Da beide Werte kleiner oder gleich p sind, also kleiner als $n^2 \ln n^2$ ist, genügen dafür $O(\log n)$ Bits.

2.12 Satz. *Bei zufälliger gleichverteilter Wahl einer Primzahl* p *kleiner oder gleich* $n^2 \log n^2$ *ist*

$$\mathbf{P}(F_p(a) = F_p(b) \mid a \neq b) \in O(\frac{1}{n}) .$$

Für den Beweis benötigen wir ein schwieriges und ein leicht herzuleitendes Resultat.

2.13 Satz. (Chebyshev) *Für die Anzahl $\pi(n)$ der Primzahlen kleiner oder gleich n gilt:*

$$\frac{7}{8}\frac{n}{\ln n} \leqslant \pi(n) \leqslant \frac{9}{8}\frac{n}{\ln n}$$

Es ist also $\pi(n) \in \Theta(n/\ln n)$.

2.14 Lemma. Eine Zahl kleiner oder gleich 2^n hat höchstens n verschiedene Primteiler.

Der „Beweis“ besteht in der Überlegung, dass jeder Primteiler größer oder gleich 2 ist, das Produkt von $n+1$ Primteilern also größer oder gleich 2^{n+1}.

2.15 Beweis. (von Satz 2.12) Algorithmus 2.11 liefert genau dann eine falsche Antwort, wenn $c \neq 0$ ist und von p geteilt wird. Da $c \leqslant 2^n$ ist, hat es nach dem vorangegangenen Lemma höchstens n verschiedene Primteiler.

Es sei t die obere Schranke des Intervalls, so dass der Algorithmus gleichverteilt eine der Primzahlen im Bereich $[2;t]$ auswählt. Nach Satz 2.13 gibt es dort $\pi(t) \in \Theta(t/\ln t)$ Primzahlen.

Die Wahrscheinlichkeit $\mathbf{P}(F_p(a) = F_p(b) \mid a \neq b)$, ein p zu wählen, das zu einer falschen Antwort führt, ist also höchstens $O(\frac{n}{t/\ln t})$.

Wählt man nun wie im Algorithmus $t = n^2 \ln n^2$, so ergibt sich eine obere Schranke in $O(1/n)$. ∎

Wir kommen nun zu der zweiten Aufgabenstellung.

2.16 Beim sogenannten *pattern matching* sind ein *Text* $x = x_1 \cdots x_n$ und ein kürzeres *Suchmuster* $y = y_1 \cdots y_m$ gegeben. Die Aufgabe besteht darin, herauszufinden, ob für ein $1 \leqslant j \leqslant n - m + 1$ gilt: Für alle $1 \leqslant i \leqslant m$ ist $y_i = x_{j+i-1}$.

Bezeichnet $x(j)$ das Teilwort $x_j \cdots x_{j+m-1}$ der Länge m von x, dann gilt es also herauszufinden, ob für ein $1 \leqslant j \leqslant n - m + 1$ das zugehörige $x(j) = y$ ist.

Die letzte Formulierung lässt es schon naheliegend erscheinen, die zuvor untersuchte Technik erneut zu verwenden.

Zuvor sei darauf hingewiesen, dass man das Problem deterministisch in Zeit $O(m+n)$ lösen kann, etwa mit den Algorithmen von Boyer und Moore (1977) oder Knuth, Morris und Pratt (1977). Das ist besser als der triviale deterministische Algorithmus, der für jedes j alle Bits von $x(j)$ und y vergleicht und daher $O(nm)$ Zeit benötigt.

Im folgenden werden zwei randomisierte Algorithmen vorgestellt. Beim ersten handelt es sich um ein Monte-Carlo-Verfahren, beim zweiten um ein Las-Vegas-Verfahren, die Fehlerwahrscheinlichkeit ist dort also stets 0.

2.17 Die grundlegende Idee besteht wieder darin, statt y und ein $x(j)$ direkt zu vergleichen, dies mit

den Fingerabdrücken $F_p(y)$ und $F_p(x(j))$ zu tun. Eine wesentliche Beobachtung ist dabei:

$$\begin{aligned} x(j+1) &= \sum_{i=1}^{m} x_{j+1+i-1} 2^{i-1} \\ &= \frac{1}{2}(x_j - x_j) + \frac{1}{2} \sum_{i=1}^{m-1} x_{j+1+i-1} 2^i + x_{j+1+m-1} 2^{m-1} \\ &= \frac{1}{2} \left(\sum_{i=0}^{m-1} x_{j+i} 2^i - x_j \right) + x_{j+m} 2^{m-1} \\ &= \frac{1}{2} \left(\sum_{i=1}^{m} x_{j+i-1} 2^{i-1} - x_j \right) + x_{j+m} 2^{m-1} \\ &= \frac{1}{2} \left(x(j) - x_j \right) + x_{j+m} 2^{m-1} \, . \end{aligned}$$

Man kann also mit einer *konstanten* Anzahl von Operationen $x(j+1)$ aus $x(j)$ berechnen und folglich auch $F_p(x(j+1))$ aus $F_p(x(j))$. Der folgende Algorithmus benötigt daher $O(m+n)$ Schritte.

2.18 Algorithmus.

⟨*Überprüfung, ob Bitfolge* $y_1 \cdots y_m$ *in* $x_1 \cdots x_n$ *vorkommt*⟩
⟨*Ausgabe: erstes* j*, wo das der Fall ist, oder* -1 *sonst.*⟩
$p \leftarrow$ ⟨*Primzahl; zufällig gleichverteilt aus denen kleiner oder gleich* $n^2 m \ln n^2 m$ *ausgewählt.*⟩
$y \leftarrow \sum_{i=1}^{m} y_i 2^{i-1} \bmod p$
$z \leftarrow \sum_{i=1}^{m} x_i 2^{i-1} \bmod p$
for $j \leftarrow 1$ **to** $n - m$ **do**
  if $(y = z)$ **then**
    return j ⟨*die erste Stelle, an der eine „Übereinstimmung" gefunden wurde*⟩
  fi
  $z \leftarrow (z - x_j)/2 + x_{j+m} 2^{m-1} \bmod p$
od
return -1 ⟨y *kommt sicher nicht in* x *vor*⟩

2.19 Satz. *Algorithmus 2.18 liefert höchstens mit Wahrscheinlichkeit* $O(1/n)$ *eine falsche Antwort.*

2.20 Beweis. Ähnlich wie in Beweis 2.15 sei t die obere Schranke des Intervalls, so dass der Algorithmus gleichverteilt eine der Primzahlen im Bereich $[2; t]$ auswählt. Nach Satz 2.13 gibt es dort $\pi(t) \in \Theta(t/\ln t)$ Primzahlen.

Die Wahrscheinlichkeit $\mathbf{P}(F_p(y) = F_p(x(j)) \mid y \neq x(j))$, ein p zu wählen, das an der Stelle j zu einer falschen Antwort führt, ist höchstens $O(\frac{m}{t/\ln t})$ (da $|y - x(j)|$ höchstens m verschiedene Primteiler besitzt). Die Wahrscheinlichkeit, dass eine falsche Antwort gegeben wird, weil an irgendeiner der $O(n)$ Stellen der Test versagt, ist daher höchstens $O(\frac{nm}{t/\ln t})$.

Wählt man nun wie im Algorithmus $t = n^2 m \ln n^2 m$, so ergibt sich eine obere Schranke in $O(1/n)$. ∎

2.3 Ein randomisierter Quicksortalgorithmus

Der folgende Algorithmus ist eine naheliegende randomisierte Variante eines einfachen Quicksort-Algorithmus. Der Einfachheit halber nehmen wir an, dass alle Eingaben paarweise verschieden sind.

2.21 Übung. Wie kann man jeden Algorithmus, der voraussetzt, dass alle Eingaben paarweise verschieden sind, so modifizieren, dass er auch dann richtig arbeitet, wenn mehrere Elemente mit gleichem Sortierschlüssel auftreten? (Und zwar sogar so, dass die relative Reihenfolge solcher Elemente erhalten bleibt!)

2.22 Algorithmus.
proc $R[1 \dots n] \leftarrow \text{RandQuickSort}(S[1:n])$
⟨*Eingabe: ein Feld* $S[1:n]$ *paarweise verschiedener Zahlen*⟩
⟨*Ausgabe: ein Feld* $R[1:n]$ *der Zahlen aus* S *in sortierter Reihenfolge*⟩
⟨*Zwischenablage in Feldern* S_1 *und* S_2⟩
$i \leftarrow$ **random**$(1,n)$ ⟨*liefere gleichverteilt eine natürliche Zahl zwischen* 1 *und* n⟩
$y \leftarrow S[i]$
$j_1 \leftarrow 1; j_2 \leftarrow 1;$
for $i \leftarrow 1$ **to** n **do**
  if $S[i] < y$ **then** $S_1[j_1] \leftarrow S[i]; j_1 \leftarrow j_1 + 1$ **fi**
  if $S[i] > y$ **then** $S_2[j_2] \leftarrow S[i]; j_2 \leftarrow j_2 + 1$ **fi**
od
return $\text{RandQuickSort}(S_1[1:j_1-1]) \cdot y \cdot \text{RandQuickSort}(S_2[1:j_2-1])$ ⟨$\cdot$ *sei Konkatenation*⟩

2.23 Der obige Algorithmus liefert unabhängig davon, welcher Wert auf einer Rekursionsstufe als Pivotelement gewählt wird, stets die korrekte Ausgabe.

Allerdings haben die Auswahlen der y einen Einfluß auf die Laufzeit des Algorithmus (selbst bei festgehaltener Eingabefolge). Wird zum Beispiel immer der kleinste vorhandene Wert als Pivotelement gewählt, so ergibt sich eine quadratische Laufzeit. Wird dagegen immer der Median gewählt, ist die Laufzeit $O(n \log n)$.

Die Laufzeit dieses Algorithmus ist also eine Zufallsvariable. Im folgenden soll ihr *Erwartungswert* bestimmt werden.

Als erstes sieht man, dass er nur von der Anzahl n zu sortierender Elemente abhängt, aber nicht von den konkreten Werten. Des weiteren wollen wir den Erwartungswert der Laufzeit nur bis auf einen konstanten Faktor bestimmen. Deswegen genügt es, festzustellen, wie oft je zwei Feldelemente miteinander verglichen werden, denn zwei Feldelemente werden höchstens einmal miteinander verglichen.

2.24 Es bezeichne X_{ij} die Zufallsvariable, die angibt, ob die Elemente $R[i]$ und $R[j]$ miteinander verglichen wurden ($X_{ij} = 1$) oder nicht ($X_{ij} = 0$). Gesucht ist dann also

$$\mathbf{E}\left[\sum_{i=1}^{n}\sum_{j>i} X_{ij}\right] = \sum_{i=1}^{n}\sum_{j>i} \mathbf{E}\left[X_{ij}\right]$$

Ist p_{ij} die Wahrscheinlichkeit für den Vergleich von $R[i]$ und $R[j]$, so ist $\mathbf{E}\left[X_{ij}\right] = 1 \cdot p_{ij} + 0 \cdot (1 - p_{ij}) = p_{ij}$. Das Problem besteht also im wesentlichen darin, die p_{ij} zu bestimmen.

2.25 Man beachte, dass diese Wahrscheinlichkeiten *nicht* alle gleich sind, und überlege sich, warum das so ist!

2.26 Satz. *Der Erwartungswert der Laufzeit von* RandQuickSort *für Eingaben der Länge* n *ist in* $O(n \log n)$.

2.27 Beweis. Es seien nun ein i und ein j ($1 \leqslant i < j \leqslant n$) beliebig aber fest vorgegeben. Zur Bestimmung von p_{ij} betrachtet man binäre Bäume mit den zu sortierenden Zahlen als Knoten, die durch je eine Ausführung von RandQuickSort wie folgt rekursiv festgelegt sind: Die Wurzel des Baumes ist das zufällig gewählte Pivotelement y. Der linke Teilbaum ergibt sich rekursiv nach der gleichen Regel gemäß der Ausführung beim Aufruf RandQuickSort($S_1[1 : j_1 - 1]$) und analog der rechte gemäß der Ausführung beim Aufruf RandQuickSort($S_2[1 : j_2 - 1]$). Bei einem In-Order-Durchlauf des Baumes ergäbe sich also die sortierte Reihenfolge der Werte.

Von Interesse ist im Folgenden aber eine andere Reihenfolge: Beginnend bei der Wurzel werden nacheinander absteigend auf jedem Niveau jeweils von links nach rechts alle Knoten besucht.

(Auch) bei dieser Besuchsreihenfolge wird irgendwann zu ersten Mal ein Element $R[k]$ angetroffen, für das $i \leqslant k \leqslant j$ ist. Es ist bei den Vergleichen in diesem Rekursionsschritt, dass tatsächlich entschieden wird, dass $R[i]$ *vor* $R[j]$ einzusortieren ist.

Dabei gibt es zwei Fälle:

1. Es ist $k = i$ oder $k = j$, d. h. eines der Elemente $R[i]$ und $R[j]$ ist das Pivotelement und die beiden werden miteinander verglichen.

2. Es ist $i < k < j$, d. h. ein anderes Element ist Pivot, $R[i]$ und $R[j]$ werden nicht miteinander verglichen, kommen in verschiedene Teilbäume (da auch $R[i] < R[k] < R[j]$ ist) und werden folglich auch später nie mehr miteinander verglichen.

Nun wird im Algorithmus jedes (noch) zur Verfügung stehende Element gleichwahrscheinlich als Pivotelement ausgewählt. Außerdem wissen wir, dass offensichtlich eines der $j - i + 1$ Elemente $R[i], \ldots, R[j]$ ausgewählt wurde.

Im betrachteten Rekursionsschritt wird also eines von $j - i + 1$ Elementen gleichwahrscheinlich ausgewählt, und in genau zwei dieser Fälle ($k = i$ und $k = j$) werden $R[i]$ und $R[j]$ miteinander verglichen.

Also ist die Wahrscheinlichkeit dafür, dass $R[i]$ und $R[j]$ überhaupt miteinander verglichen werden gerade $p_{ij} = 2/(j - i + 1)$.

Damit können wir nun abschätzen:

$$\mathbf{E}\left[\sum_{i=1}^{n-1}\sum_{j>i} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j>i}\mathbf{E}[X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1} = \sum_{i=1}^{n-1}\sum_{k=2}^{n-i+1}\frac{2}{k} \leqslant 2\sum_{i=1}^{n}\sum_{k=1}^{n}\frac{1}{k}\,.$$

Nun ist $H_n = \sum_{k=1}^{n}\frac{1}{k}$ die sogenannte n-te harmonische Zahl, für die man weiß: $H_n = \ln n + \Theta(1)$. Hieraus folgt die Behauptung. ∎

2.28 Es sei auf einen Unterschied zwischen den beiden in diesem Kapitel behandelten Algorithmen hingewiesen: Der randomisierte Quicksortalgorithmus liefert *immer* das richtige Ergebnis. Beim Identitätstest nimmt man dagegen in Kauf, dass die Antwort manchmal nicht zutreffend ist, allerdings so „selten“, dass man die Fehlerwahrscheinlichkeit leicht beliebig weit drücken kann.

Außerdem ist letzterer ein Entscheidungsproblem, für das nur zwei Antworten in Frage kommen.

2.29 Arbeiten über randomisierte Primzahltests (Rabin 1976; Solovay und Strassen 1977; Solovay und Strassen 1978) gehören zu den Klassikern. Auch diese Algorithmen liefern manchmal die falsche Antwort, aber immer nur für zusammengesetzte Zahlen die Behauptung „prim“; für Primzahlen ist die Antwort stets richtig.

Schwieriger war es für Adleman und Huang (1987), einen randomisierten Algorithmus anzugeben, der für Primzahlen manchmal fälschlicherweise behauptet sie sei zusammengesetzt, aber für zusammengesetzte Zahlen stets die richtige Antwort liefert.

Vor einigen Jahren haben schließlich Agrawal, Kayal und Saxena (2002) bewiesen, dass das Problem sogar deterministisch in Polynomialzeit gelöst werden kann.

Zusammenfassung

1. Randomisierte Algorithmen enthalten eine Zufallskomponente.
2. Das führt im Allgemeinen dazu, dass — *bei festgehaltener Eingabe* — z. B. die Laufzeit eine randomisierten Algorithmus eine Zufallsvariable ist.
3. Manche randomisierten Algorithmen liefern immer das richtige Ergebnis.
4. Manche randomisierten Algorithmen liefern unter Umständen ein falsches Ergebnis. Dann ist man im Allgemeinen an kleinen Fehlerwahrscheinlichkeiten interessiert.

Literatur

Adleman, L. M. und A. M.-D. Huang (1987). „Recognizing Primes in Random Polynomial Time“. In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*. S. 462–469 (siehe S. 18).

Agrawal, Manindra, Neeraj Kayal und Nitin Saxena (2002). *PRIMES is in P*. Report. Kanpur-208016, India: Department of Computer Science und Engineering, Indian Institute of Technology Kanpur. URL: http://www.cse.iitk.ac.in/news/primality.pdf (siehe S. 18).

Boyer, Robert S. und J. Strother Moore (1977). „A fast string search algorithm“. In: *Communications of the ACM* 20.10, S. 762–772 (siehe S. 14).

Coppersmith, Don und Shmuel Winograd (1990). „Matrix Multiplication via Arithmetic Progressions“. In: *Journal of Symbolic Computation* 9.3, S. 251–280 (siehe S. 11).

Freivalds, Rusins (1977). „Probabilistic machines can use less running time“. In: *Information Processing 77, Proceedings of the IFIP Congress 1977*. Hrsg. von B. Gilchrist. Amsterdam: North Holland, S. 839–842 (siehe S. 11).

Knuth, D. E., J. H. Morris und V. R. Pratt (1977). „Fast pattern matching in strings“. In: *SIAM Journal on Computing* 6.2, S. 323–350 (siehe S. 14).

Rabin, Michael O. (1976). „Probabilistic Algorithms“. In: *Algorithms and Complexity*. Hrsg. von J. F. Traub. Academic Press, S. 21–39 (siehe S. 18).

Solovay, R. und V. Strassen (1977). „A Fast Monte-Carlo Test for Primality“. In: *SIAM Journal on Computing* 6.1, S. 84–85 (siehe S. 18).

Solovay, R. und V. Strassen (1978). „Erratum: A Fast Monte-Carlo Test for Primality“. In: *SIAM Journal on Computing* 7.1, S. 118 (siehe S. 18).

