Bonner Mathematikturnier 2017
Vorbereitungsmaterial

Radboud Universiteit

KU LEUVEN

UNIVERSITÄT BONN

Vorwort

Das Bonner Mathematikturnier gliedert sich in zwei Runden: die **Staffel**, die vormittags stattfindet, und den Nachmittagswettbewerb „**Sum of Us**“. Im Nachmittagsprogramm dreht sich alles um die praktische Anwendung der Mathematik. Um sich auf die „Sum of Us“ vorzubereiten, stellen wir die folgenden Unterlagen zur Verfügung. Wir empfehlen wärmstens, diese durchzuarbeiten. Das Material darf auch während der Nachmittagsrunde benutzt werden. Es beinhaltet einige Aufgaben, deren Lösungen circa eine Woche vor dem Turniertag veröffentlicht werden.

Das diesjährige Thema ist **fehlererkennende und -korrigierende Codes**. Wenn wir zum Beipsiel einen Text lesen, dann vrebessern wir unbeuwsst kleinere Fehler. Immehrin ist in der Sprache, in der der Text geschrbeien ist, nicht jede beleibige Konbimation von Buchstaben ein Wort und beianhe automatsich krriogiert man dann eine fehlerhafte Buchstabeknombination zu einem bestheenden Begriff, der sich daovn nicht zu sehr unterscehidet. Das ist auch der Grnud, dass du die leztten paar Säzte ohen gorße Pbroleme lseen konnetst.

Bei digitaler Kommunikation, wenn man beispielsweise von seinem Handy aus oder übers Internet eine Nachricht verschickt oder eine Mediendatei abspielt, werden Fehler mathematisch aufgespürt und verbessert. Über diesen Anwendungsbereich der Mathematik wollen wir euch hier eine Einleitung geben.

Die Aufgaben und das Vorbereitungsmaterial haben Klaas Parmentier, Pieter Senden, Floris Vermeulen und Art Waeterschoot (Mathematikstudenten an der KU Leuven) unter der Begleitung von Joeri Van der Veken und Bart Dioos erstellt. Die Übersetzung ins Deutsche erfolgte durch Leoni Winschermann.

Die Organisatoren,

Wieb Bosma (Radboud Universiteit Nijmegen),
Rainer Kaenders (Universität Bonn),
Joeri Van der Veken (KU Leuven).

Inhaltsverzeichnis

1 Einleitung **3**

2 Einfache Beispiele von Codes **4**

2.1 Vom Text zu Bits und zurück 4

2.2 Der Wiederholungscode 6

2.3 Der Paritätscode 7

3 Fehler detektieren und korrigieren **8**

4 Interludium: Modulo rechnen **9**

5 Aus der Praxis: Hammingcodes **11**

6 Aus der Praxis: Barcodes **13**

1 Einleitung

Gegenwärtig werden Informationen digital gespeichert und versendet: digitale Fotos, SMS-Nachrichten, dein Lieblingsfilm als Mediendatei oder auf Blu-ray,... das sind nichts anderes als Gruppen von Nullen und Einsen, auch **Bits** genannt.

Sobald man digitale Informationen versendet, zum Beispiel über ein mobiles Netzwerk oder das Internet, oder wenn eine Mediendatei oder Blu-ray gelesen wird, lässt es sich nicht verhindern, dass Fehler auftreten, wodurch Teile der Daten verändert werden: eine Null kann eine Eins geworden sein und umgekehrt. Was wir wollen, ist, dass der Empfänger die ursprüngliche Information aus der Information, die ankommt, rekonstruieren kann. Darum wird noch vor dem Versenden zusätzliche Information hinzugefügt. Diesen Prozess nennen wir **codieren** und die zusätzliche Information nennen wir **redundante Information**. Wenn wir diese redundante Information schlau genug wählen, sollte der Empfänger im Stande sein, die Fehler sowohl zu finden, als auch zu verbessern. Das Rekonstruieren der ursprünglichen Botschaft nennen wir **decodieren**.

Ein einfaches Beispiel: nimm an, dass Frank über eine schlechte Radio- oder Telefonverbindung seinen Namen buchstabieren will. Dann kann er „Friedrich-Richard-Anton-Nordpol-Kaufmann“ sagen statt „F-R-A-N-K“. Seine Nachricht besteht dann aus 36 Buchstaben, von denen 5 die ursprüngliche Botschaft formen, während die anderen 31 redundante Information sind. Anstelle von „Friedrich“ könnte der Empfänger das Wort „Friedlich“ verstehen, aber dann kann er noch immer den Namen von Frank rekonstruieren. Er könnte auch „Richahd“ statt „Richard“ verstehen, aber das erste Wort gehört nicht zu unserer Sprache und sicher auch nicht zum deutschen Funkalphabet. Er weiß also, dass „Richard“ gesagt wurde und kann dementsprechend wieder Franks Namen rekonstruieren. Auch wenn dieser Code sehr gut funktioniert, ist er nicht besonders **effizient**: durch das Codieren wird die Menge an Information immerhin versiebenfacht! Wenn wir eine WhatsApp-Nachricht verschicken, wollen wir aber lieber nicht, dass nur ein Sechstel der Mobilen Daten, für die wir bezahlen, die eigentliche Nachricht beinhaltet. Wir wollen also nicht nur, dass unser fehlerkorrigierende Code funktioniert, sondern er soll außerdem so effizient wie möglich sein. Und damit stehen wir auch schon vor einem ziemlich schwierigen mathematischen Problem...

2 Einfache Beispiele von Codes

2.1 Vom Text zu Bits und zurück

Bevor wir zwei simpele Codes besprechen, setzen wir uns erst noch mit einer geläufigen Technik auseinander, mit der man einen Text in eine Bitfolge umzusetzen kann und andersrum, nämlich dem ASCII[1]-Protokoll, vom dem eine vereinfachte Version auf Seite 5 zu finden ist.

Mit Hilfe dieser Tabelle werden Buchstaben und Symbole umgesetzt in eine Folge von jeweils 7 Bits. So können wir zum Beispiel das Wort „Code“ schreiben als

`1000011110111111001001100101`.

Andererseits sehen wir an Hand der Tabelle, dass die Bitreihe

`1000011110111111011001100001`

für das Wort „Cola“ steht.

[1] ASCII steht für *American Standard Code for Information Interchange.*

Symbol	Bits	Symbol	Bits	Symbol	Bits
a	1100001	A	1000001	0	0110000
b	1100010	B	1000010	1	0110001
c	1100011	C	1000011	2	0110010
d	1100100	D	1000100	3	0110011
e	1100101	E	1000101	4	0110100
f	1100110	F	1000110	5	0110101
g	1100111	G	1000111	6	0110110
h	1101000	H	1001000	7	0110111
i	1101001	I	1001001	8	0111000
j	1101010	J	1001010	9	0111001
k	1101011	K	1001011	(Leerzeichen)	0100000
l	1101100	L	1001100	.	0101111
m	1101101	M	1001101	,	0101100
n	1101110	N	1001110	:	0111010
o	1101111	O	1001111	;	0111011
p	1110000	P	1010000	?	0111111
q	1110001	Q	1010001	!	0100001
r	1110010	R	1010010	(	0101000
s	1110011	S	1010011	)	0101001
t	1110100	T	1010100	[	1011011
u	1110101	U	1010101	]	1011101
v	1110110	V	1010110	*	0101010
w	1110111	W	1010111	/	0101111
x	1111000	X	1011000	+	0101011
y	1111001	Y	1011001	–	0101101
z	1111010	Z	1011010	=	0111101

Tabelle 1: Vereinfachte Version der ASCII-Umrechnungstabelle

Aufgabe 2.1. Setze die Worte „Math“ und „Tech“ in Bitfolgen um.

Aufgabe 2.2. Was steht hier?

- `10110011001111100110010011110100000011101001010 01`
- `10100111100011110100111001011101110110001111001 01`

2.2 Der Wiederholungscode

Einer der simpelsten fehlerkorrigierenden Codes ist der **Wiederholungscode**. Hierbei codieren wir eine Botschaft dadurch, dass wir jedes Bit n mal wiederholen.

Nimm an, dass wir die Nachricht `1101` versenden wollen. Wir benutzten den Wiederholungscode mit $n = 3$. Die codierte Nachricht ist dann

`111111000111`,

konstruiert durch die dreimalige Wiederholung jedes einzelnen Bits. Anstelle der ursprünglichen Botschaft von 4 Bits, versenden wir die codierte Nachricht von 12 Bits.

Nimm nun an, dass unterwegs etwas schief gelaufen ist und dass die empfangene Nachricht

`101111000111`

lautet. Das zweite Bit ist also falsch angekommen. Um die Nachricht zu decodieren, teilt der Empfänger die Nachricht in 3-Bit-lange Stücke: `101`, `111`, `000` und `111`. Diese Stücke nennen wir die **Worte** des Codes. Der Code gesteht nur zwei Worte zu, nämlich `000` und `111`, immerhin kann ein anderes Wort kein Teil der verschickten Nachricht gewesen sein. Der Empfänger sieht also, dass das erste Wort, das er bekommen hat, falsch ist. Er kann einen Fehler *detektieren* (aufspüren). Mehr noch, wenn er davon ausgeht, dass in dem Wort nur ein einziger Fehler ist, dann kann er den Fehler *korrigieren*. Das ursprüngliche erste Wort müsste dann schließlich `111` gewesen sein und nicht `000`. Der Empfänger kann die empfangenen Bitfolgen also decodieren und findet die korrekte ursprüngliche Botschaft `1101`. Mit diesem Code hat er die Möglichkeit pro Wort genau einen Fehler zu detektieren und zu korrigieren.

Aufgabe 2.3. Wie viele Fehler können wir dann pro Wort (das jetzt aus n Bits besteht) detektieren und korrigieren, wenn wir den Wiederholungscode für beliebige n anwenden?

Aufgabe 2.4. Jemand schickt dir eine Botschaft, indem er die ASCII-Tabelle und den Wiederholungscode mit $n = 3$ benutzt. Die empfangene Nachricht, die wahrscheinlich fehlerhaft ist, lautet

`001111110011000101101111110110000010001100`.

Was war die ursprüngliche Botschaft?

2.3 Der Paritätscode

Ein anderes Beispiel für einen einfachen Code ist der **Paritätscode**. Um eine Botschaft zu codieren, teilen wir sie erst in Gruppen von 4 Bits auf. (Wenn die Menge an Bits in der Nachricht kein Vielfaches von vier ist, ergänzen wir ganz zum Schluss einige Nullen.) An jede dieser Gruppen wird dann hinten ein redundantes Bit angehängt nach den folgenden Regeln:

- wir fügen eine `0` hinzu, falls die Menge der Einsen in der Gruppe gerade ist.
- wir fügen eine `1` hinzu, falls die Menge der Einsen in der Gruppe ungerade ist.

Ein Wort besteht in diesem Code also aus fünf Bits, von denen eine gerade Anzahl Bits eine `1` ist. So ist beispielsweise `00101` ein zugelassenes Wort, `10101` aber nicht.

Wenn wir die Botschaft `11010011` versenden wollen, dann sehen wir, dass die erste Gruppe von vier Bits drei Einsen hat, also fügen wir hier eine `1` hinzu. Die zweite Gruppe hat zwei Einsen, also hängen wir hinten eine `0` dran. Die codierte Botschaft lautet dementsprechend `1101100110`.

Nimm jetzt an, dass die empfangene Nachricht `1001100110` ist. Das zweite Bit ist also falsch. Der Empfänger macht nun das Folgende: er unterteilt die Nachricht in Wörter von fünf Bits und findet so `10011` und `00110`. Da das erste Wort eine ungerade Menge Einsen hat, weiß er, dass sich ein Fehler eingeschlichen hat. Allerdings weiß er nicht, was genau der Fehler ist. Während der Wiederholungscode mit $n = 3$ einen Fehler sowohl detektieren als auch korrigieren kann, kann der Paritätscode zwar auch einen Fehler erkennen, nicht aber verbessern. Andererseits ist die Menge der überflüssigen Information bei Ersterem viel größer als bei Letzterem.

Aufgabe 2.5. Codiere die ursprüngliche Nachricht aus Aufgabe 2.4 mit dem Paritätscode.

3 Fehler detektieren und korrigieren

Zusammen mit den zwei einfachen Codes zuvor haben wie den Begriff **Wort** eingeführt. Anhand der Beispiele scheint es, dass auch die Struktur der Menge aller von einem Code zugelassener Worte, die man auch als die **Sprache** eines Codes bezeichnet, festlegt, wie viele Fehler ein Code pro Wort detektieren und eventuell korrigieren kann. Lasst uns das eben mit gesprochener Sprache vergleichen.

Beispiel 3.1. Wenn wir uns den Satz „Gedtern haben wir uns hier verapredet“ ansehen, kostet es uns kaum Mühe um abzuleiten, dass wahrscheinlich „Gestern haben wir uns hier verabredet“ gemeint war. Jedes Wort hier, dass nicht zur deutschen Sprache gehört, hat nämlich starke Ähnlichkeit mit genau einem Wort, das wir wohl erkennen.

Beispiel 3.2. Wenn du das Word „Kild“ liest, dann siehst du sofort, dass es falsch ist, es hat nämlich keinerlei Bedeutung im Deutschen. Allerdings können wir nicht sagen, was gemeint war. Das Wort „Kild“ unterscheidet sich mit nur einem Buchstaben von sowohl „Kind“, als auch „mild“, als auch „wild“.

Beispiel 3.3. Manchmal kannst du die Fehler nichtmal detektieren. Wenn du eine SMS mit „Ich will nur noch ins Fett“ einem Freund schickst, dann kann er bestenfalls vermuten, dass du „Ich will nur noch ins Bett“ meinst, aber ganz sicher weiß er es doch nicht...

Ein Vorteil von Codierungs-Sprache gegenüber gesprochener Sprache ist, dass in fehlerkorrigierenden Codes alle Wörter gleichlang sind, zum Beispiel n Bits beim Wiederholungscode für beliebige n und 5 Bits beim Paritätscode.

Aufgabe 3.1. Aus wie vielen Worten besteht die Sprache des

- Wiederholungscodes für beliebige n,
- Paritätscodes?

Im Allgemeinen gelten die nachfolgenden Behauptungen über das Detektieren und Korrigieren von Fehlern mit Hilfe eines Codes.

- Um einen oder mehrere Fehler in einem Wort **detektieren** zu können, darf das empfangene Wort nicht Teil der Sprache des Codes sein.
- Um einen Fehler in einem Wort **korrigieren** zu können, darf es nur ein einziges Wort in der Sprache geben, das sich an genau einer Stelle vom empfangenen Wort unterscheidet. Mit anderen Worten: alle Wörter in der Sprache müssen sich in mindestens 3 Stellen voneinander unterscheiden.

Die Menge der Stellen in denen sich zwei Codewörter einer bestimmten Sprache unterscheiden, nennt man den **Hammingabstand** zwischen diesen Wörtern, benannt nach dem Mathematiker Richard Hamming, einem Pionier im Entwickeln von fehlerkorrigierenden Codes. Einen seiner Codes werden wir im nächsten Kapitel kennen lernen. Wir notieren den Hammingabstand mit d_H (d vom Englischen "distance" und H von "Hamming"). So ist zum Beispiel $d_H(\texttt{1101}, \texttt{1011}) = 2$ und $d_H(\texttt{10101010}, \texttt{01010101}) = 8$.

Aufgabe 3.2. Bestimme die folgenden Hammmingabstände:

- $d_H(\texttt{101001}, \texttt{101001})$,
- $d_H(\texttt{01101}, \texttt{01101})$,
- $d_H(\texttt{01101110}, \texttt{01101010})$.

Aufgabe 3.3. Wir wissen bereits, dass, um einen Fehler pro Wort korrigieren zu können, der Hammingabstand zwischen jedem Paar Codeworten mindestens 3 sein muss. Was muss der minimale Hammingabstand sein um k Fehler pro Codewort verbessern zu können?

Aufgabe 3.4. Zeige, dass der Hammingabstand die Dreiecksungleichung erfüllt, also dass für drei Wörter w_1, w_2 und w_3, die alle gleich lang sind, gilt, dass

$$d_H(w_1, w_2) + d_H(w_2, w_3) \geq d_H(w_1, w_3).$$

Das Verbessern eines Fehlers mit Hilfe eines fehlerkorrigierenden Codes funktioniert, indem man ein empfangenes Wort, das nicht zur Sprache des Codes gehört, durch das der Sprache zugehörige Wort ersetzt, das "am nächsten dran liegt" im Hinblick auf den Hammingabstand.

4 Interludium: Modulo rechnen

Ein mathematisches Konzept, dass wir brauchen werden, um fortgeschrittenere fehlerkorrigierende Codes zu begreifen, ist das sogenannte *Modulo rechnen*. Die einfachste Methode um es zu erklären, ist wahrscheinlich anhand einer Uhr. Ein Tag beginnt um Mitternacht, es ist dann 0 Uhr. Wir fangen an die Stunden zu zählen als 1 Uhr, 2 Uhr, 3 Uhr usw.. Mittags ist es 12 Uhr. Und danach zählen wir einfach weiter: 13 Uhr, 14 Uhr,... bis 23 Uhr. Danach sagen wir nicht, dass es 24

Uhr wird, sondern wieder 0 Uhr. Also 24 wird 0, 25 wird 1, 26 wird 2 usw.. Wir nennen das **Modulo 24 rechnen** und schreiben zum Beispiel

$$24 \equiv 0 \bmod 24,\ 25 \equiv 1 \bmod 24,\ 26 \equiv 2 \bmod 24, \ldots, 65 \equiv 17 \bmod 24, \ldots$$

Eigentlich ist $a \bmod 24$ nichts anderes, als der Rest bei Division der natürlichen Zahl a durch 24. Man könnte sagen, dass wir für jede natürliche Zahl a, mit $a \bmod 24$ die eindeutige Zahl $b \in \{0, 1, \ldots, 23\}$ notieren, sodass $a - b$ ein Vielfachens ist von 24. Diese Definition können wir auch für ganze Zahlen a ausbreiten. Dann ist beispielsweise $(-1) \equiv 23 \bmod 24$, was auch unseren Erwartungen im Beispiel mit der Uhr entspricht: -1 Uhr ist eigentlich 23 Uhr am Tag zuvor.

Wir müssen natürlich nicht Modulo 24 rechnen, sondern können 24 durch jede ganze Zahl ersetzen, die größer oder gleich 2 ist.

Definition 4.1. Seien a und n ganze Zahlen mit $n \geq 2$. Dann ist $a \bmod n$ die eindeutige Zahl $b \in \{0, 1, \ldots, n-1\}$, sodass $a - b$ ein Vielfaches von n ist.

Beispiel 4.1.

- $19 \equiv 1 \bmod 9$, denn $19 - 1 = 18 = 2 \cdot 9$,
- $-19 \equiv 8 \bmod 9$, denn $-19 - 8 = -27 = -3 \cdot 9$,
- $291 \equiv 1 \bmod 2$, denn $291 - 1 = 290 = 145 \cdot 2$,
- $a \equiv 0 \bmod 2$ wenn a gerade ist und $a \equiv 1 \bmod 2$ wenn a ungerade ist.

Aufgabe 4.1. Berechne

- $29 \bmod 12$,
- $83 \bmod 25$,
- $38273172 \bmod 10$,
- $38273172 \bmod 3$.

Für kompliziertere Berechnungen sind die folgenden Resultate sehr praktisch:

$$((a \bmod n) + (b \bmod n)) \bmod n \equiv (a + b) \bmod n,$$
$$((a \bmod n) \cdot (b \bmod n)) \bmod n \equiv (a \cdot b) \bmod n.$$

Lass uns die erste Formel kontrollieren für beispielsweise $a = 3$, $b = 7$ und $n = 4$. Der linke Teil gibt uns $((3 \bmod 4) + (7 \bmod 4)) \equiv (3 + 3) \equiv 6 \equiv 2 \bmod 4$, während der rechte Teil $(3 + 7) \equiv 10 \equiv 2 \bmod 4$ ist.

Aufgabe 4.2. Berechne

- $10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \bmod 5$,
- $8^{43} \bmod 9$,
- $32103210321 \bmod 16$,
- $5^2 \cdot 3^7 \cdot 2^{37} \bmod 97$.

5 Aus der Praxis: Hammingcodes

Die Wiederholungs- und Paritätscodes sind zu einfach, als dass sie effizient Fehler detektieren und korrigieren könnten. Darum werden wir jetzt eine Familie von komplizierteren Codes einführen: die Hammingcodes. Sie wurden von dem Mathematiker Richard Hamming bereits rund 1950 erarbeitet, werden heute aber immer noch genutzt, obwohl inzwischen neue Codes entwickelt wurden. Wir beschränken uns hier auf die (7,4)-Hammingcodes, was bedeutet, dass wir an jede Gruppe von 4 Bits einer Botschaft 3 redundante Bits hinzufügen. Codewörter haben also 7 Bits.

Beispiel 5.1. Wir teilen, genau wie beim Paritätscode, die Botschaft wieder in Blöcke von 4 Bits auf (eventuell nach Anfüllen der Nachricht mit Nullen) und fügen an jeden Block von 4 Bits nach dem in der folgenden Tabelle erläuterten Schema 3 redundante Bits hinzu.

0000 000	0100 101	1000 011	1100 110
0001 111	0101 010	1001 100	1101 001
0010 110	0110 011	1010 101	1110 000
0011 001	0111 100	1011 010	1111 111

Tabelle 2: Ein $(7, 4)$-Hammingcode

Die Sprache des Codes besteht demnach aus 16 Wörtern von jeweils 7 Bits. Außerdem ist der Code so konstruiert, dass alle Codewörter einen Hammingabstand von mindestens 3 voneinander haben, sodass genau ein Fehler detektiert und korrigiert werden kann. Nimm an, dass eine Botschaft mit diesem Code verschlüsselt wird. Wenn der Empfänger das Wort `0111010` bekommt, dann sieht er, dass ein Fehler passiert ist, denn das ist kein Codewort. Wenn man davon ausgeht, dass

nur ein einziger Fehler im Wort ist, dann kann man diesen auch korrigieren, denn nur das Wort `0101010` unterscheidet sich an genau einer Stelle von `0111010`. Die ursprüngliche Botschaft war also `0101`.

Aufgabe 5.1. Eine Nachricht wird mit dem Hammingcode aus der Tabelle oben codiert. Wenn der Empfänger die Nachricht

`00001111100110110001`

bekommt, die eventuell Fehler beinhaltet, was ist dann wahrscheinlich die ursprüngliche Nachricht?

Wir geben jetzt eine detailliertere Erläuterung darüber, wie der Hammingcode in Beispiel 5.1 konstruiert wird. Gegeben 4 Bits $\mathtt{b}_1\mathtt{b}_2\mathtt{b}_3\mathtt{b}_4$, wird ein Codewort $\mathtt{c}_1\mathtt{c}_2\mathtt{c}_3\mathtt{c}_4\mathtt{c}_5\mathtt{c}_6\mathtt{c}_7$ von 7 Bits anhand der folgenden Regeln erzeugt:

$$\begin{aligned}(\mathtt{c}_1, \mathtt{c}_2, \mathtt{c}_3, \mathtt{c}_4) &= (\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_3, \mathtt{b}_4)\\ \mathtt{c}_5 &\equiv (\mathtt{b}_2 + \mathtt{b}_3 + \mathtt{b}_4) \bmod 2,\\ \mathtt{c}_6 &\equiv (\mathtt{b}_1 + \mathtt{b}_3 + \mathtt{b}_4) \bmod 2,\\ \mathtt{c}_7 &\equiv (\mathtt{b}_1 + \mathtt{b}_2 + \mathtt{b}_4) \bmod 2.\end{aligned}$$

Weil alle Formeln linear sind in b_1, b_2, b_3 und b_4 (modulo 2), ist dies ein Beispiel eines **linearen Codes**.

Aufgabe 5.2. Zeige, dass der Wiederholungscode für beliebige n und der Paritätscode ebenfalls linear sind, indem du derartige Formeln auch für diese beiden Codes aufstellst.

Wenn man $\mathtt{c}_5$, $\mathtt{c}_6$ und $\mathtt{c}_7$ mit anderen Linearkombinationen berechnet, kann man andere (7,4)-Hammingcodes finden, die genauso gut funktionieren. Allerdings sind nicht alle lineare Kombinationen zugelassen. Wir geben zwei Beispiele zur Verdeutlichung:

Beispiel 5.2. Der Code mit den redundanten Bits

$$\begin{aligned}\mathtt{c}_5 &\equiv (\mathtt{b}_1 + \mathtt{b}_2 + \mathtt{b}_4) \bmod 2,\\ \mathtt{c}_6 &\equiv (\mathtt{b}_1 + \mathtt{b}_2 + \mathtt{b}_3) \bmod 2,\\ \mathtt{c}_7 &\equiv (\mathtt{b}_1 + \mathtt{b}_3 + \mathtt{b}_4) \bmod 2,\end{aligned}$$

ist ein anderer $(7, 4)$-Hammingcode.

Aufgabe 5.3. Schreibe alle Codewörter von diesem Code auf und kontrolliere, dass der Hammingabstand untereinander immer mindestens 3 ist.

Beispiel 5.3. Der Code mit den redundanten Bits

$$\begin{aligned} \mathtt{c}_5 &\equiv (\mathtt{b}_1 + \mathtt{b}_2) \bmod 2, \\ \mathtt{c}_6 &\equiv (\mathtt{b}_3 + \mathtt{b}_4) \bmod 2, \\ \mathtt{c}_7 &\equiv (\mathtt{b}_1 + \mathtt{b}_2 + \mathtt{b}_3 + \mathtt{b}_4) \bmod 2, \end{aligned}$$

ist *kein* $(7, 4)$-Hammingcode. Weil $c_7 \equiv (c_5 + c_6) \bmod 2$ gilt, enthält das letzte redundante Bit keinerlei neue Informationen.

Aufgabe 5.4. Überprüfe, dass es für den letzten Code zwei Codewörter gibt, die sich nur an zwei Stellen voneinander unterscheiden.

Aufgabe 5.5. Benutze die ASCII-Umwandlungstabelle auf Seite 5 und den $(7, 4)$-Hammingcode aus Beispiel 5.1 um das Wort „Math“ zu codieren. (Diese Übung hast du teilweise schon in Aufgabe 2.1 gelöst.)

Aufgabe 5.6. Du bekommst die Nachricht

`10101011111100010110100101100100000110011001010 01`

geschickt, inklusive eventueller Fehler. Die Nachricht wurde mit der ASCII-Tabelle in Bits umgesetzt und mit dem $(7, 4)$-Hammingcode aus Beispiel 5.1 codiert. Was war die ursprüngliche Botschaft?

6 Aus der Praxis: Barcodes

Ein letztes Beispiel für die Verwendung von Fehlerkorrektur, das wir hier diskutieren wollen, sind Barcodes. Die meisten Produkte, die verkauft werden, sind mit den wohlbekannten „Strichcodes“ versehen. Gegenwärtig codieren diese die sogenannte EAN[2]-13-Nummer des Produktes, eine Zahl bestehend aus 13 Ziffern, die auch unter dem Barcode selbst zu finden ist. Wie genau diese Zahl in Striche umgesetzt wird, ist an sich schon eine interessante Sache, aber hier wollen wir vor allem nach dem Aufbau der Zahl selbst schauen. Die ersten drei Ziffern stehen für das Land in dem die EAN-13-Nummer zugewiesen wurde (nicht unbedingt das Land in dem das Produkt auch produziert wurde). So stehen 540–549 für Belgien und Luxemburg, 400–440 für Deutschland und 870–879 für die Niederlande. Dann

[2]EAN steht für *European Article Number*

folgen 9 Ziffern, die den Fabrikanten und das Produkt identifizieren und als Letztes wird eine Kontrollziffer angehangen, die aus den zwölf vorherigen Ziffern errechnet wird. Wenn beim Scannen an der Kasse etwas schief läuft, weil der Barcode zum Beispiel beschädigt ist, dann wird die Kontrollziffer wahrscheinlich nicht mit dem Rest der Zahl übereinstimmen und das System kann den Fehler detektieren.

Die Berechnung der Kontrollziffer geht wie folgt: nimm die ersten 12 Ziffern der EAN-13-Nummer

$$a_1a_2a_3a_4a_5a_6a_7a_8a_9a_{10}a_{11}a_{12}$$

und berechne die Summe der Ziffern auf den geraden Stellen und die Summe der Ziffern auf den ungeraden Stellen:

$$\begin{aligned} s_0 &= a_2 + a_4 + a_6 + a_8 + a_{10} + a_{12}, \\ s_1 &= a_1 + a_3 + a_5 + a_7 + a_9 + a_{11}. \end{aligned}$$

Berechne dann

$$s = 3s_0 + s_1.$$

Die Kontrollziffer wird gegeben durch

$$c_{13} \equiv (-s) \bmod 10.$$

Beispiel 6.1. Die Zahl 5400112347034 ist eine gültige EAN-13-Nummer. Wenn wir die oben genannten Berechnungen durchführen, finden wir nämlich

$$\begin{aligned} s_0 &= a_2 + a_4 + a_6 + a_8 + a_{10} + a_{12} = 4 + 0 + 1 + 3 + 7 + 3 = 18, \\ s_1 &= a_1 + a_3 + a_5 + a_7 + a_9 + a_{11} = 5 + 0 + 1 + 2 + 4 + 0 = 12 \end{aligned}$$

und darum $s = 3s_0 + s_1 = 3 \cdot 18 + 12 = 66$. Es gilt $4 \equiv (-66) \bmod 10$.

Aufgabe 6.1. Bestimme die fehlende Ziffer in den folgenden EAN-13-Nummern.

Eine Variation von Barcodes, die aktuell viel benutzt wird, sind die quadratischen QR-Codes. Auch hier wird sehr viel redundante Information hinzugefügt, sodass die Botschaft noch immer decodiert werden kann, wenn beim Scannen etwas schief geht.

Aufgabe 6.2. Scanne den QR-Code auf dem Deckblatt des Vorbereitungsmaterials mit deinem Smartphone und stelle fest, dass es immer noch funktioniert, auch wenn die Information teilweise durch den Text "Vorbereitungsmaterial" überdeckt wird.

Bemerkung: Zum Scannen von QR-codes (und Barcodes) eignet sich die App "i-nigma", die auf allen Betriebssystemen gut funktioniert. Am Tag des Turnieres muss jedes Team im Besitz eines Smartphones sein, auf dem diese oder eine ähnliche App installiert ist.