Fehlerkorrektur

Digitale AV Technik, MIB 5

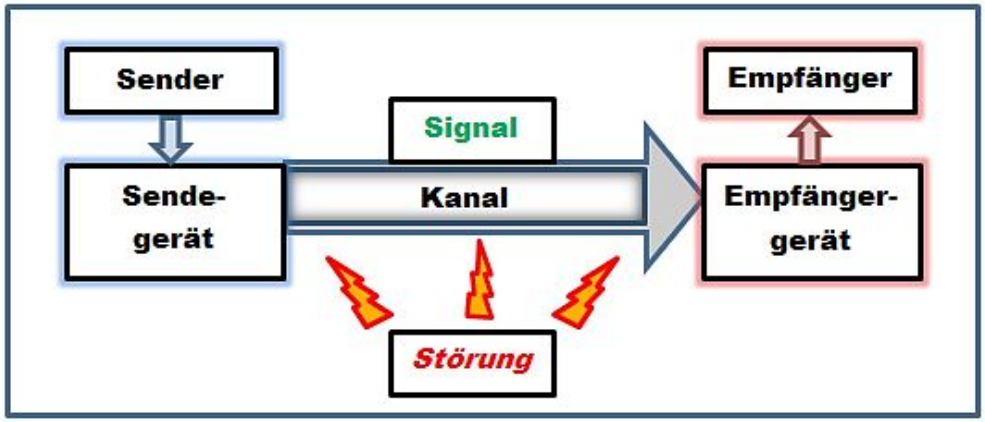
Aus Sicht der Informationstheorie

Problem	Kompression	**Fehlerkorrektur**
Ziel	Effizienz	Verlässlichkeit
Anwendung	Quellencodierung	**Kanalcodierung**

Algorithmische Perspektive

Problem	**Fehlerkorrektur**
Algorithmen	**Hamming code**
	Reed-Solomon Code
	Turbo-Code

Fehlererkennung



Störungen sind unvermeidlich. Zunächst ist es schon mal hilfreich zu erkennen, ob es einen Fehler bei der Datenübertragung gab oder nicht.

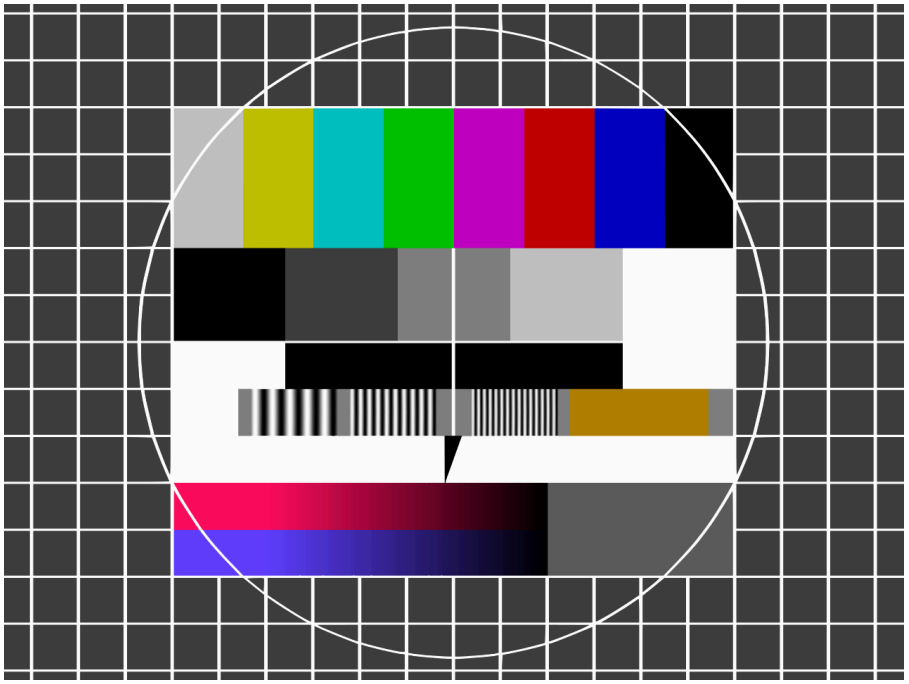
😉 Der erste Bitfehler der Geschichte

Aigeus, König von Athen, wartete besorgt auf die Rückkehr seines Sohnes Theseus, der nach Kreta gereist war, um gegen den Minotaurus zu kämpfen. Vor der Abreise hatten sie vereinbart, dass Theseus bei einer erfolgreichen Rückkehr weiße Segel setzen würde. Doch auf der Heimfahrt vergaß die Besatzung vor lauter Freude, die schwarzen Segel gegen weiße auszutauschen. Als Aigeus die schwarzen Segel sah, nahm er an, sein Sohn sei tot, und stürzte sich aus Verzweiflung ins Meer.

Fehler erkennen

- Wann sind Fehler überhaupt kritisch?
- Wie kann überprüft werden, ob eine Nachricht fehlerfrei übertragen wurde?

Testsignale



Bildquelle: Rotkaeppchen68 (in de.wikipedia), CC BY-SA 3.0, via Wikimedia Commons

Man sendet ein vereinbartes Signal, dass der Empfänger bereits kennt. Dadurch lassen sich systematische (also immer wieder gleich auftretende) Fehler erkennen.

Blockcodes

Ein Code, also eine Bitkette wird um eine Anzahl Bits erweitert, die keinen zusätzlichen Inhalt beitragen, sondern zum Schutz der Daten hinzugefügt werden.

Paritätsbits

Man prüft die Parität der Daten eines Blocks, also ob sie eine gerade (oder ungerade) Anzahl an **1en** enthalten.
Man verwendet im einfachen Fall nur ein Schutzbit und setzt dieses so, dass immer eine gerade Anzahl an **1en** entsteht.
Der Empfänger prüft einfach die Parität und weiß, dass ein Fehler dabei war, wenn keine gerade Anzahl an **1en** vorliegt.

Beispiel Parität

Beispiele:

100111 -> Parität 0

1101 -> Parität 1

10101 -> Parität 1

Berechnung im Rechner mit **XOR**:

```
def compute_parity(binary_string):
    parity = 0
    for bit in binary_string:
        parity ^= int(bit)
    return parity
```

Prüfsummen (checksums)

Bei vielen Anwendungen wird eine Prüfsumme berechnet und mit übertragen. Der Empfänger berechnet die selbe Summe und vergleicht das Ergebnis.

Beispiel: EAN

Zyklische Redundanzüberprüfung (CRC)

CRC sind weit verbreitet bei der Datenübertragung auch innerhalb einer Festplatte.

Die Idee ist es den Binärcode als Polynom zu interpretieren. Dieses Polynom wird dann zyklisch durch ein allen bekanntes Prüfpolynom dividiert und der Rest wird als Schutzbits angehängt.

Sehr gute Video-Quelle: Ben Eater

CRC Übersicht

Wir benötigen folgende Konzepte:

- Fehlererkennung mit Modulo-Division
- Nachrichtendaten als Polynom
- Polynomdivision
- Endliche Körper

Diese ermöglichen das Versenden und Überprüfen von Nachrichten mit CRC sehr effizient bei Wahl eines geeigneten Generatorpolynoms.

Fehlererkennung mit Modulo-Division (01)

Wir wollen den Text "Hi!" übertragen. Dieser wird in ASCII umgewandelt und als Binärzahl interpretiert:

```
H: 72 --> 01001000
i: 105 --> 01101001
!: 33 --> 00100001

Hi! --> 010010000110100100100001
```

Was ist das als Zehnerzahl?

Fehlererkennung mit Modulo-Division (02)

Wir hängen an die Binärzahl noch 16 Nullen an:

```
01001000 01101001 00100001 00000000 00000000
```

Was ist das als Zehnerzahl?

Wir verwenden im Weiteren kleinere Zahlen zur Veranschaulichung des Prinzips

Fehlererkennung mit Modulo-Division (03)

M = 123 sei die Nachricht, die wir übertragen wollen, G = 7 sei die Prüfzahl. Wir erweitern die Nachricht M um eine Null (Länge der Prüfzahl): M' = 1230

Wir berechnen M' mod G = 1230 mod 7 = 5 und übertragen den kombinierten Wert T = M' + G - (M' mod G) = 1230 + 7 - 5 = 1232

Der Empfänger erhält also 1232 und kann nun prüfen, ob die Nachricht korrekt übertragen wurde. Da 1232 mod 7 = 0, also ist die Nachricht korrekt.

Wenn der Empfänger 1332 erhält, kann er feststellen, dass die Nachricht fehlerhaft ist, da 1332 mod 7 = 2.

Nachrichtendaten als Polynom

Binärzahlen stellen ja immer die Zweierpotenzen dar:

$$1001 = 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$$

Das können wir auch als Polynom interpretieren:

$$f(x) = 1 * x^3 + 0 * x^2 + 0 * x^1 + 1 * x^0 = x^3 + 1$$

Analog: $1010 = 1 * x^3 + 0 * x^2 + 1 * x^1 + 0 * x^0 = x^3 + x$

Polynomdivision

Polynome kann man dividieren wie Zahlen.

Bsp:

$f(x) = x^3 + 1$ und $g(x) = x + 1$

$\frac{f(x)}{g(x)} = \frac{x^3+1}{x+1} = x^2 - x + 1$

Wie kann man die Lösung binär darstellen?

Endliche Körper (engl.: finite fields) (1)

Wir arbeiten mit dem endlichen Körper $\mathbb{F}_2$ oder auch GF(2) genannt.

Weitere algebraische Strukturen
Körper

Definition:

1. Ein Ring $(R, +, 0, \cdot, 1)$ heißt **Körper** (engl. *field*), wenn $(R \setminus \{0\}, \cdot, 1)$ eine Gruppe ist.

Folie aus dem ersten Semester!

Endliche Körper (engl.: finite fields) (2)

$\mathbb{F}_2$ enthält nur die Elemente 0 und 1, sowie die Operationen Addition und Multiplikation.

Für einen endlichen Körper gilt:

- Es gibt eine Addition und eine Multiplikation, die beide abgeschlossen sind.
- Es gibt neutrale und inverse Elemente für Addition und Multiplikation.
- Addition und Multiplikation sind kommutativ und assoziativ.
- Es gilt das Distributivgesetz: $a * (b + c) = a * b + a * c$.

Endliche Körper (engl.: finite fields) (3)

- Addition: „normal Addieren dann mod. 2 rechnen“

  $0 + 0 = 0$

  $0 + 1 = 1$

  $1 + 0 = 1$

  $1 + 1 = 0$

  entspricht **XOR**

Endliche Körper (engl.: finite fields) (4)

- Multiplikation: „normal Multiplizieren, dann mod. 2 rechnen“

  $0 * 0 = 0$

  $0 * 1 = 0$

  $1 * 0 = 0$

  $1 * 1 = 1$

  entspricht **AND**

Endliche Körper (engl.: finite fields) (5)

Prüfe:

- Gibt es neutrale und inverse Elemente für Addition und Multiplikation? Und wenn ja, welche sind es?
- Sind Addition und Multiplikation kommutativ und assoziativ?
- Gilt das Distributivgesetz: $a * (b + c) = a * b + a * c$.

Polynomdivision im $\mathbb{F}_2$

$f(x) = x^3 + 1$ und $g(x) = x + 1$

$\Longrightarrow$ Polynome als Binärzahlen darstellen:

$f(x) = 1001$ und $g(x) = 11$

```
1001
11
--
010
 11
 --
 011
  11
  --
   0
```

Kein Rest, also wäre es eine richtig übertragene Nachricht.

CRC Algorithmus

1. Stelle Polynome als Binärzahlen dar und hänge an die Nachricht N Nullen, wobei N die Länge des Prüfpolynoms minus 1 ist.
2. Verschiebe den Divisor soweit nach links bis die führenden Stellen übereinstimmen
3. Berechne ein XOR zwischen Dividend und dem verschobenen Divisor
4. Falls das Ergebnis des XOR einen geringeren Grad als der Divisor hat ist dies der Rest der Division, andernfalls weiter mit Schritt 2 wobei das Ergebnis des XOR den neuen Dividend bildet.
5. Hänge den Rest an die Nachricht an.

CRC Anwendung

Der Empfänger kann nun die Nachricht mit dem Prüfpolynom dividieren und prüfen, ob der Rest 0 ist. Falls ja, ist die Nachricht korrekt übertragen worden.
Das Prüfpolynom muss also bekannt sein, damit der Empfänger die Nachricht überprüfen kann.
In der Praxis werden Prüfpolynome verwendet, die eine hohe Fehlererkennungsrate haben. Diese sind in der Regel standardisiert, z.B. CRC32.

Beispiel (Schritt 1)

Nachricht sei "Hi" als Binärzahl:

```
H: 72 --> 01001000
i: 105 --> 01101001

Hi --> 0100100001101001
```

Nachricht um 2 Nullen erweitern (Länge des Prüfpolynoms minus 1):

```
010010000110100100
```

Prüfpolynom sei 101

Beispielrechnung (Schritt 2 + 3)

Verschiebe den Divisor soweit nach links bis die führenden Stellen übereinstimmen:

```
01001000110100100
 101
```

Berechne ein XOR zwischen Dividend und dem verschobenen Divisor

```
01001000110100100
 101
 ------
 00101000110100100
```

und das ganze solange, bis...

Beispielrechnung (Schritt 4)

...der Grad des Ergebnisses kleiner ist als der Grad des Divisors:

```
00101000011010010 0
...
                0110
                 101
               ------
                  11
```

Beispielrechnung (Schritt 5)

Hänge den Rest an die Nachricht an:

```
01001000011010010011
```

Der Empfänger kann nun die Nachricht mit dem Prüfpolynom dividieren und prüfen, ob der Rest 0 ist. Falls ja, ist die Nachricht korrekt übertragen worden.

Fazit Fehlererkennung

Einfache Paritätschecks und Prüfsummen wie EAN können nur erkennen, ob es einen Fehler bei der Übertragung gab, aber nicht wo. Daher lässt sich der Fehler auch nicht korrigieren.

CRC kann Fehler erkennen und auch korrigieren, aber nur wenn die Anzahl der Fehler begrenzt ist. Je nach Länge und Wert des Prüfpolynoms kann CRC auch mehrere Fehler erkennen und korrigieren.

Fehlerkorrektur

Wie könnte man mit Hilfe der Prüfzeichen oder Prüfbits erkennen, wo genau der Fehler liegt?

Mehrdimensionale Paritätsbits

Grundidee: Verwende ein Paritätsbit für Zeilen (also für die einzelnen Codeworte) und eins für die Spalten (also für die einzelnen Bits in den Codeworten).

Beispiel:

```
1 0 0 1 0 1 1
0 1 1 0 1 1 0
1 0 1 0 1 1 0
1 0 0 0 0 0 1
1 0 1 1 1 1 1
1 1 1 0 1 0 0
1 0 0 0 0 0 1
```

- **Overhead:** 2n + 1 zusätzliche Bits für n^2 Datenbits
- 3-fehlererkennend
- 1-fehlerkorrigierend

Was sind Hamming-Codes?

- Hamming-Codes sind eine Form von **fehlerkorrigierenden Codes**, die in digitalen Kommunikationssystemen verwendet werden.
- Sie können **Einzelfehler korrigieren** und **mehrere Fehler erkennen**.

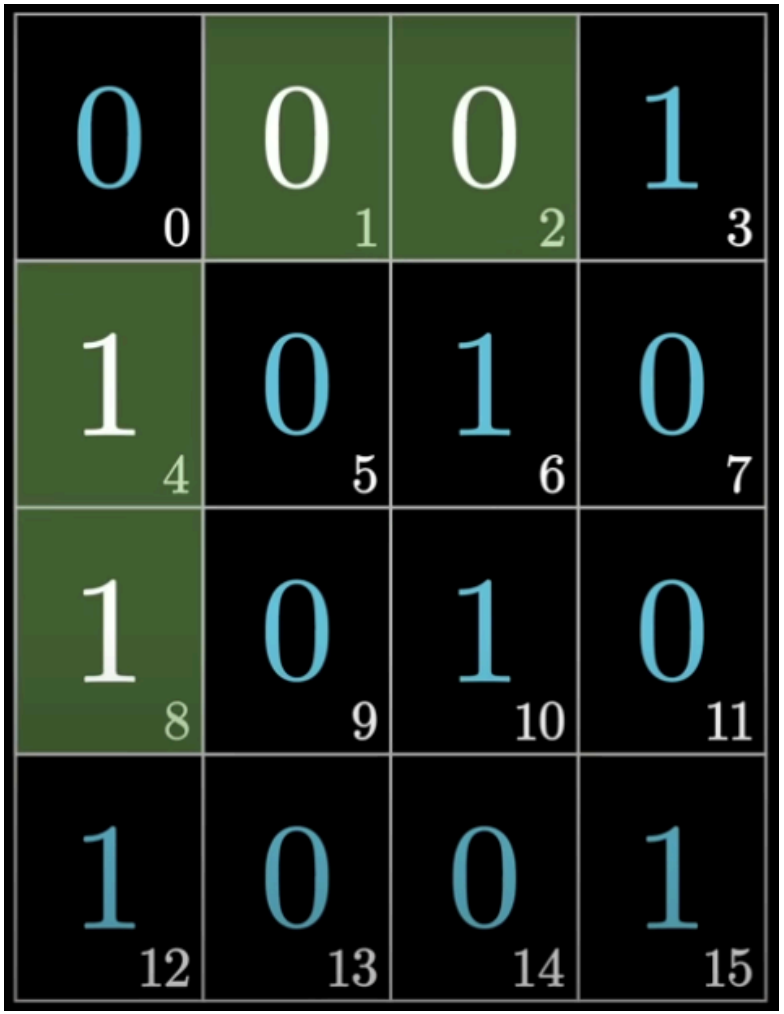
Im folgenden sind einige Screenshots aus dem sehr zu empfehlenden Video von Grant Sanderson: 3Blue1Brown - Hamming Codes

Wie funktionieren Hamming-Codes? (1)

1. **Datenbits und Paritätsbits:**

- Datenbits werden durch **zusätzliche Paritätsbits** ergänzt (grün hinterlegt im Bild rechts).

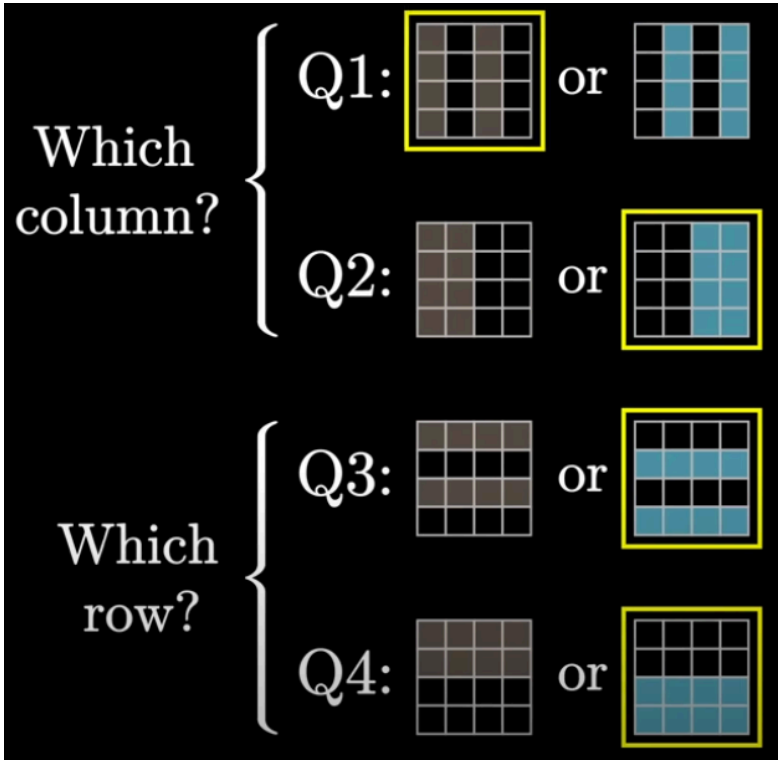
- Die Positionen der Paritätsbits folgen einer **2er-Potenz-Reihe**: 1, 2, 4, 8, usw.



Wie funktionieren Hamming-Codes? (2)

2. **Berechnung der Paritätsbits:**

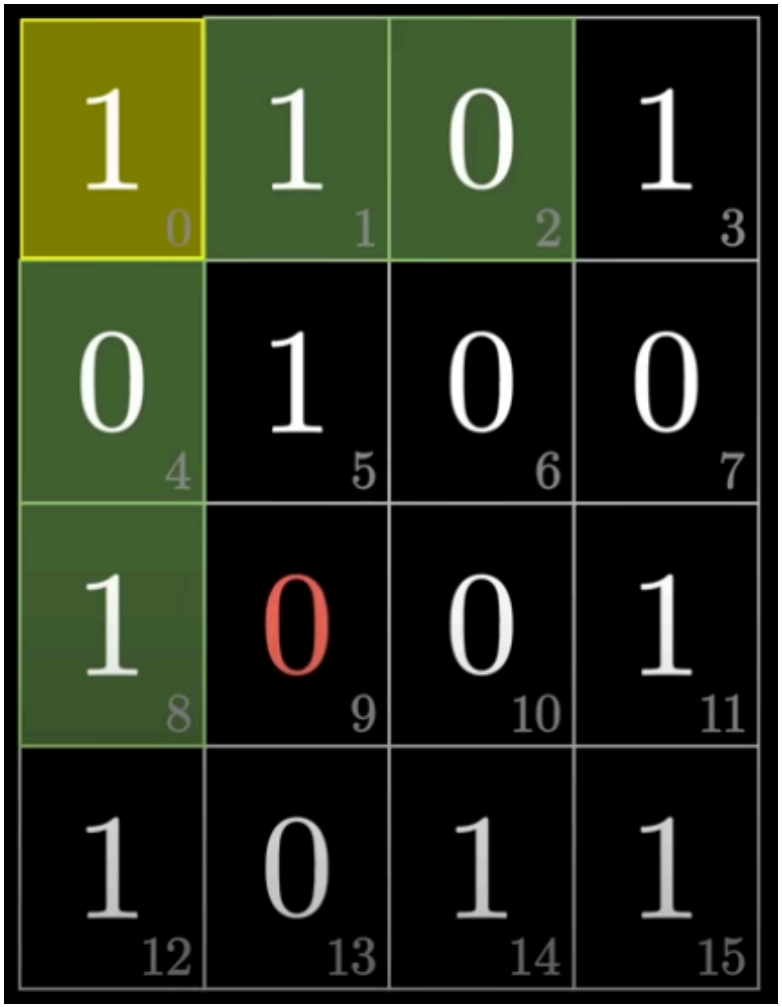
- Jedes Paritätsbit prüft eine bestimmte Menge von Datenbits.
- Das Ziel: Sicherstellen, dass die Anzahl der **1en** in einer bestimmten Gruppe von Bits **gerade** (oder ungerade) ist.



Wie funktionieren Hamming-Codes? (3)

3. **Fehlerkorrektur**

- Wenn ein Fehler auftritt, kann der **Ort des Fehlers** durch die **kombinierte Ausgabe der Paritätsbits** genau bestimmt werden.
- Das fehlerhafte Bit wird dann einfach **umgekippt** um den Fehler zu korrigieren.



🔑 Wichtige Konzepte des Verfahrens:

- **Redundanz:** Zusätzliche Bits werden hinzugefügt, um Fehler erkennen zu können.
- **Hamming-Distanz:** Die minimale Anzahl von **Bitflips**, die erforderlich ist, um einen gültigen Code in einen anderen zu verwandeln. Hamming-Codes haben eine Distanz von **3**, was bedeutet, dass sie **einen Fehler korrigieren** und **zwei Fehler erkennen** können.
- **Syndromberechnung:** Die Ausgabe der Paritätsbits ergibt ein sogenanntes **Syndrom**, das direkt den fehlerhaften Bitindex angibt.

📽️ Warum ist das Video so gut?

- **3Blue1Brown** verwendet **anschauliche Animationen**, um das komplexe Thema leicht verständlich zu machen.
- Er erklärt die **logischen Grundlagen** auf eine intuitive Weise, die sowohl für Anfänger als auch für Fortgeschrittene interessant ist.

Hier nochmal der Link zum Video:

🌐 3Blue1Brown - Hamming Codes

Ausblick

Reed-Solomon Codes

Turbocodes werden nicht in der Vorlesung behandelt, aber sind ein sehr interessantes Thema. Sie werden in der Praxis verwendet, z.B. in Mobilfunknetzen.