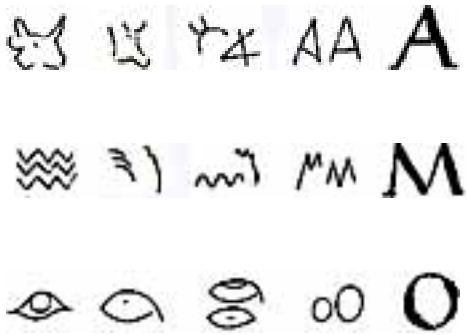
Leitprogramm

Codes



Fach: **Mathematik, Informatik**

Schule: **Gymnasium**

Alter: **11. und 12. Schuljahr**

Dauer: **4 Stunden**

Autor: **Heidi Gebauer**

Betreuer: **Prof. Dr. Juraj Hromkovic**

Fassung vom: **29.9.2005**

Schulerprobung: **Bisher noch keine**

Motivation

Stell dir vor, Du möchtest eine Musik-CD anfertigen. Nun kannst Du die entsprechenden Musikdaten in eine Bitfolge (= Sequenz von 0 und 1) umwandeln und diese auf der CD speichern (wie man das genau anstellt, ist eine Wissenschaft für sich). Der Player liest dann die gespeicherten Bits, rekonstruiert die entsprechenden Töne und spielt die Musik ab. Diese Aktionen nennt man auch **Codierung** (Umwandlung in eine Bitfolge), bzw. **Decodierung** (Rekonstruktion).
Klappt also alles bestens – aber nur solange die CD unversehrt bleibt.
Denn schon kleine Verunreinigungen oder Kratzer können dazu führen, dass der Player die entsprechenden Bits falsch einliest und somit die falschen Töne abspielt (und damit das ganze Stück ungeniessbar werden lässt).
Um diesem Szenario entgegenzuwirken, könnte man doch – sicherheitshalber – mehr Information speichern als (für die Reproduktion) unbedingt nötig. Eine Variante wäre, jedes Bit mehrmals abspeichern. So kann der Player – so lange nicht allzu viele Fehler passiert sind - Lesefehler bemerken und gegebenenfalls auch korrigieren.
Solche Codierungen nennt man **'fehlerkorrigierende Codes'**. Ein Code entspricht einer Regel, wie man codiert bzw. decodiert.
Da Fehler ärgerlich oder sogar katastrophal sind, hätte man am liebsten einen Code, der möglichst viele Fehler korrigieren kann. Auf der andern Seite kostet Speicherung viel Platz. Deshalb möchte man nicht allzu viele "Sicherheitsbits" verschwenden. Doch diese beiden Ziele widersprechen sich (je grösser die Zuverlässigkeit, desto mehr Sicherheitsbits werden benötigt). Ein klassisches Dilemma.
Die Bedeutung von **fehlerkorrigierenden Codes** geht weit über die CD-Herstellung hinaus; eingesetzt werden sie auch bei der Internet-Kommunikation, der Uebermittlung von Radiosignalen und überhaupt überall dort wo Daten übertragen werden sollten und Fehler passieren können.

Zielsetzung

Der Ablauf dieses Leitprogrammes sieht nun folgendermassen aus:
Zuerst betrachten wir zwei fehlerkorrigierende Codes, danach verallgemeinern wir das Konzept der 'Codes' und zu guter Letzt beschäftigen wir uns mit den Grenzen des Machbaren.
A propos Mathematik: Lange hielt ich fehlerkorrigierenden Codes für eine hübsche, mathematische Spielerei – bis mir aufging wie unentbehrlich und hilfreich sie in unserem – fehlerbehafteten – täglichen Leben sind.
Nach der Durcharbeitung dieses Leitprogrammes solltest Du das Konzept der fehlerkorrigierenden Codes verstehen, zwei solche Codes kennen und wissen, wofür sie gut sind.

Lernziele

1. Ein Beispiel für einen Code kennen
2. Ein Gefühl für die Struktur von Codes bekommen
3. Mehr Uebung im Umgang mit Beweisen
4. Das Dilemma 'Kürze vs Zuverlässigkeit' in eigenen Worten formulieren

Begriffs-Erklärungen

Hier findest Du Erklärungen zu den verwendeten Begriffen. Du kannst diese Seite ruhig überspringen und gleich mit dem ersten Kapitel beginnen. Bei Bedarf ist zurückblättern dann jederzeit möglich.

- **Bit**
  Das Wort 'Bit' ist eine Wortkreuzung aus 'binary' und 'digit'. Es ist die kleinste Einheit eines Computers und kann nur die Werte '0' und '1' annehmen.

- **Bitfolge, Bitsequenz**
  Das ist nichts anderes als aneinandergereihte Bits. (Bsp: 0100)

- **Code, Codewörter**
  Wir verwenden den Begriff 'Code' im ersten Kapitel mit einer leicht anderen Bedeutung als in Kapitel 2 & 3.

  Kapitel 1: Der **Code** entspricht einer Regel, wie man codiert bzw. decodiert.
  Diejenigen Bitfolgen, welche die Uebersetzung einer Nachricht sind, nennen wir **Codewörter**.

  Kapitel 2: Ein **Code** entspricht einer Menge von gleichlangen Bitsequenzen. Diejenigen Bitsequenzen, welche im Code vorkommen, nennen wir **Codewörter**.

  Ganz allgemein kann man sagen: Codewörter sind diejenigen Bitfolgen, die vom Sender verschickt werden.

- **Codeblock**
  Ab und zu verwenden wir auch den Ausdruck **Codeblock** für Codewort.

- **Codieren**
  Unter **Codieren** verstehen wir: Die Uebersetzung einer Nachricht in eine Bitsequenz – sprich in ein Codewort.

- **Decodieren**
  Wir verwenden den Begriff 'Decodieren' im ersten Kapitel mit einer leicht anderen Bedeutung als in Kapitel 2 & 3.
  Kapitel 1: Decodieren bedeutet: "Zurückübersetzen" einer Bitsequenz in eine Nachricht.
  Kapitel 2: Decodieren bedeutet: Umwandlung des erhaltenen Wortes in ein Codewort (nach festgelegten Regeln).

- **Invertieren:**
  Dieser Begriff bezieht sich auf Bits. Ein **Bit invertieren** bedeutet: Seinen Wert ändern – sprich: Aus '0' wird '1' und aus '1' wird '0'.

1. Ein konkretes Beispiel

In diesem Kapitel werden zwei ausgewählte fehlerkorrigierende Codes vorgestellt.
Wir werden zuerst – an konkreten Beispielen – veranschaulichen wie sie funktionieren.
Danach werden wir sie nach folgenden Kriterien beurteilen

- Wie gut können Fehler korrigiert werden?
- Wie viele (Zusatz-) Bits werden dabei benötigt?

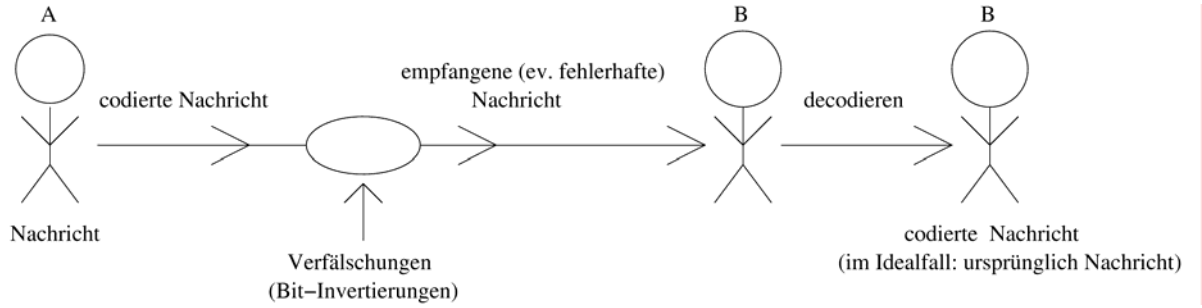
Schlussendlich zeigen wir dann auf, wie ein Code konkret 'Fehler korrigieren' kann.
Zu guter Letzt seid ihr dran:
Ihr werdet in die Rolle des Uebermittlers schlüpfen und selbständig ein paar (kleine!) Nachrichten codieren bzw. decodieren.

Lernziele

1. Die zwei vorgestellten Codes kennen
2. Nachrichten codieren bzw. decodieren können
3. Wissen was es bedeutet 'ein Code kann einen Fehler korrigieren'

1.1 Allgemeine Infos

Durchs ganze Leitprogramm hindurch werden wir das folgende allgemeine Szenario betrachten:



A möchte *B* eine Nachricht übermitteln. Dazu codiert er sie in eine Bitfolge und "schickt" diese an *B*. Auf dem Weg kann es jedoch zu Verfälschungen kommen. *B* empfängt die (u.U. verfälschte) Bitfolge und decodiert sie. Für das Musik-Beispiel heisst das: *A* ist derjenige, der die CD brennt, *B* ist der CD-Player. Die Nachricht ist die Musik, die Bitfolge ist die auf der CD gespeicherte Codierung. "Verschicken" bedeutet "auf der CD speichern". Die Daten werden sozusagen auf der CD "transportiert", Verfälschungen kommen durch Kratzer zustande.)
Wie das mit der Decodierung genau funktioniert wird später erklärt.

Wie sehen nun diese "Verfälschungen" konkret aus?
In der Praxis ist fast alles möglich (vertauschte Bits, verlorengegangene Bits oder sogar Verdoppelungen von Bits). In diesem Leitprogramm betrachten wir jedoch nur die folgende Fehlermöglichkeit: Ein Bit kann seinen Wert ändern, sprich: aus '0' kann '1' werden oder umgekehrt. Diesen Vorgang nennt man auch **Invertierung**.
Alle andern Verfälschungen schliessen wir aus.

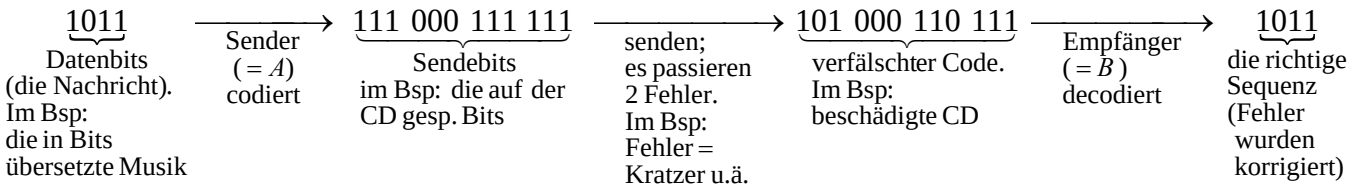
Und zum Schluss noch dies: Solche Codes machen überhaupt nur dann Sinn, wenn nicht allzu viele Fehler passieren; würde beispielsweise jedes Bit mit Wahrscheinlichkeit ½ invertiert, dann ist die ankommende Bit-Folge – trotz ausgeklügelster Codierung – völlig chaotisch.
Wir nehmen also im Folgenden immer an, dass nicht allzu viele Fehler passieren. (D.h. die Wahrscheinlichkeit, dass ein Bit invertiert wird ist wesentlich kleiner als ½ .)

1.2 Repetitionscodes

In diesem Kapitel nehmen wir an, dass die Nachrichten immer Bitfolgen sind. Dies ist keine Einschränkung, denn wir können unser Musikstück (oder was auch immer wir verschicken wollen) einfach in eine Bitsequenz "übersetzen".
Wie wollen wir unsere Nachricht nun codieren?
Die wohl simpelste Möglichkeit wäre es nun, sie einfach so zu lassen wie sie ist. Denn unter Codierung verstehen wir 'Umwandlung in eine Bitfolge' und unsere Nachricht ist bereits eine Bitfolge.
Aber ganz so einfach können wir es uns nicht machen, denn wir wollen ja einen fehlerkorrigierenden Code und müssen deshalb noch etwas Zusatzinformation "reincodieren".

Eine naheliegende Idee wäre, jedes Bit mehrmals – sagen wir 3 mal – zu senden; für das Bit '0' wird also '000' gesendet, für das Bit '1' wird '111' übermittelt.
Der Empfänger decodiert nun jeden 3-er Block einzeln und macht dabei einen sogenannten "Mehrheitsentscheid", das heisst wenn in einem Block mehr '1' als '0' vorkommen, decodiert er ihn zu '1', andernfalls decodiert er ihn zu '0'.
(Damit dieser Entscheid eindeutig gefällt werden kann, sendet man bei diesem Verfahren jedes Bit eine ungerade Anzahl Male.)
In diesem Fall verschicken wir pro '**Datenbit**' drei '**Sendebits**'.

Beispiel:



Wir haben nun gesehen, dass dieser Code einen Fehler pro 3-Bit-Block korrigieren kann. Treten jedoch zwei oder mehr Fehler pro 3-Bit-Block auf, können diese nicht mehr korrigiert werden. Beispiel: *A* möchte das Bit '1' übertragen. Beim Senden passieren jedoch zwei Fehler und aus '111' wird '001'. *B* wird dann zu '0' decodieren.

Aufgabe: Wie sieht es aus, wenn nun jedes Bit 5 mal wiederholt wird?
Wie Viele Fehler (pro 5-Bit-Block) kann der Code nun korrigieren?

(Die Antwort geht aus untenstehendem Text hervor.)

Betrachten wir noch kurz den allgemeinen Fall: Wir senden jedes Bit *n* mal (wobei *n* eine ungerade Zahl ist), oder anders gesagt, für jedes Bit verschicken wir einen Block von *n* (gleichen) Bits.

Definition: Wir sagen "Ein Code kann pro gesendeten Block *k* Fehler **korrigieren"**, falls gilt:
Wenn bei der Uebertragung eines Blocks höchstens *k* Fehler passiert sind, dann decodiert ihn der Empfänger korrekt (das heisst er decodiert den Block zum ursprünglichen Datenbit)

Der ***n*-Bit-Repetitionscode (=** der Code, der jedes Bit *n* mal sendet) kann also pro gesendeten *n*-Bit-Block $\frac{n-1}{2}$ Fehler korrigieren (beachte: *n* ist eine ungerade Zahl).

CHECK POINT I

Nun könnte man auch auf die Idee kommen, jedes Bit eine gerade Anzahl Male zu senden. Wir betrachten also den *n*-Bit-Repetitionscode für **gerade** *n*. Um Komplikationen zu vermeiden legen wir fest: Erhält der Empfänger einen Block, der gleich viele '1' und '0' erhält, kann er – nach Belieben – zu '0' **oder** '1' decodieren.

Wie viele Fehler kann der *n*-Bit-Repetitionscode korrigieren, wenn *n* gerade ist?

Hast Du eine Lösung? Dann schau nach auf Seite 34.
Weisst Du nicht weiter? Dann lies den letzten Abschnitt nochmals genau durch.

Mit entsprechend vielen Wiederholungen kann ein Repetitionscode also beliebig zuverlässig gemacht werden. Die Sache hat allerdings einen Haken: Je öfter man die einzelnen Bits wiederholt, desto länger dauert die Uebertragung (bzw. im Fall von CD's: desto mehr Speicher wird benötigt). Und man kann nun wirklich nicht Stunden brauchen, um ein einziges Bit zu übermitteln. Es besteht also ein klares Dilemma zwischen Aufwand und Zuverlässigkeit.

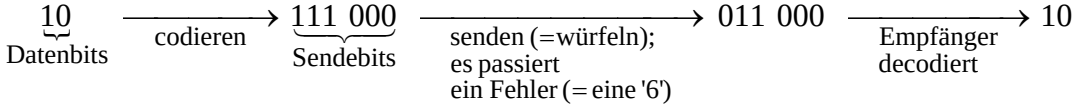
Wenn nun sehr wenig Fehler passieren (viel, viel weniger als ein Fehler pro 3 Bits), dann ist sogar die Verwendung des 3-Bit-Repetitioncodes ein 'mit Kanonen auf Spatzen schiessen'. Könnte man für solche Fälle nicht einen Code konstruieren, der zwar nicht ganz so viele Fehler korrigieren kann, aber dafür mit viel weniger Sendeaufwand auskommt?
Man kann. Und zwar (unter anderem) mit dem im nächsten Unterkapitel beschriebenen Code.

Experiment: Für einmal wollen wir nicht nur von fehlerhaften Uebertragungen reden, sondern sie auch mal konkret simulieren. Alles was Du dazu brauchst ist ein Würfel und Papier & Bleistift. Schreib Dir nun eine beliebige Bitsequenz auf (die Länge sollte etwa 10 Bits betragen). Codiere sie dann in den 3-Bit-Repetitionscode. Die Uebertragung wird dann wie folgt simuliert:
Würfle für jedes Bit einmal. Fällt eine '6' bedeutet dies 'es ist ein Fehler passiert'. Invertiere in diesem Fall das entsprechende Bit. Wirfst Du eine andere Zahl als '6', bedeutet dies 'alles okay', belasse also das entsprechende Bit wie es ist. Decodiere dann zum Schluss die erhaltene Bitsequenz nach der weiter oben beschriebenen Regel (zur Erinnerung: Falls mehr '1' als '0' vorkommen, decodiere zu '1', andernfalls decodiere zu '0').

Beispiel: Wir möchten die Bitsequenz 10 mitteilen.

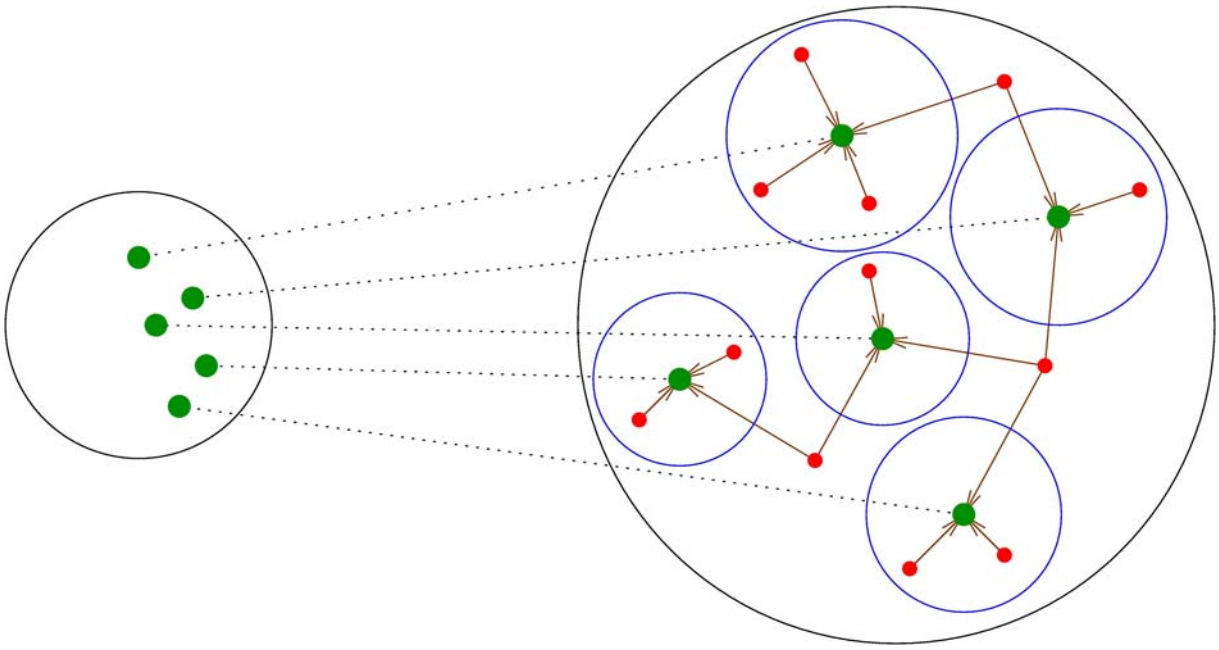
1. Codierung: 111 000
2. Würfle für jedes Bit. **Annahme:** Für die erste '1' wird eine '6' geworfen, für alle andern Bits wird eine andere Zahl geworfen. Die empfangene Sequenz ist 011 000

3. Der Empfänger decodiert die empfangene Sequenz zu 10



1.3 Der 3-Kreis-Hamming-Code

Motivation: Wie versprochen, konstruieren wir nun einen Code, der etwas sparsamer mit den zu sendenden Bits umgeht. (Im Gegenzug müssen wir dafür in Kauf nehmen, dass nicht mehr ganz so viele Fehler korrigiert werden können.)
Konkret: Statt für jedes Bit 3 oder mehr Sendebits aufzuwenden, fassen wir nun 4 aufeinanderfolgende Bits zu einem Codeblock zusammen. Anschaulich sieht das so aus: Wir "übersetzen" eine 4-Bit-Sequenz in eine 7-Bit-Sequenz. Eine Möglichkeit, diese Uebersetzung zu realisieren ist, jedem 4-Bit-Block noch 3 'Kontrollbits' anzuhängen.



Raum der 7-Bit-Sequenzen
(grün: Codewort
rot: kein Codewort
die Pfeile geben an, zu welchem Codewort decodiert wird)
Jedes Codewort hat sozusagen einen "Umkreis" (blau), in welchem alle Sequenzen zu ihm decodiert werden.

Beachte: Es gibt viel mehr 7-Bit-Sequenzen als 4-Bit-Sequenzen.
Aufgabe 1: Wie viele 4-Bit-Sequenzen gibt es? Und wie viele 7-Bit-Sequenzen?
(Lösung: s. Seite 41)

Beim Decodieren wird dann wieder "zurückübersetzt".
Empfangen wir eine (7-Bit-)Sequenz, welche kein Codewort (d.h. keine Uebersetzung einer 4-Bit-Sequenz) ist, decodieren wir zu demjenigen Codewort, welches den kleinsten "Abstand" (= die kleinste Anzahl unterschiedlicher Stellen) zur empfangenen Sequenz aufweist. Haben mehrere Codeworte den gleichen (kleinsten) Abstand, so können wir davon ein beliebiges auswählen. Die Decodierung ist also nicht immer eindeutig!

Unser Ziel ist deshalb: Zwei Codeworte sollen sich jeweils möglichst stark voneinander unterscheiden. So stark, dass folgendes gilt: Passiert bei der

Uebertragung eines Codewortes ein Fehler (= eine Bit-Invertierung), dann hat das ursprünglich gesendete Codewort den eindeutig kleinsten Abstand vom empfangen Wort. Der Empfänger decodiert somit also immer richtig.

Aufgabe 2: Nehmen wir mal an, wir möchten jeweils 4-Bit-Blöcke in 7-Bit-Blöcke übersetzen. Wie viele Zusatzbits braucht man dann pro Bit (im Schnitt)?
Die Lösung steht auf Seite 41.

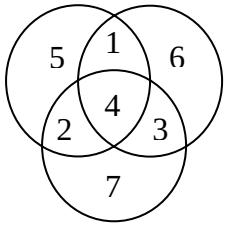
Weshalb nimmt man eigentlich genau 4-Bit resp. 7-Bit Blöcke?
Könnte man stattdessen nicht 3-Bit-Blöcke in 5-Bit-Blöcke übersetzen?
Man bräuchte dann im Schnitt nur
Oder wie wäre es mit 2 und 4?

Aufgabe 3: Wir möchten einen Code konstruieren, der wie folgt funktioniert: Der Sender nimmt jeweils 2-Bit-Blöcke , hängt ihnen **ein** Kontrollbit an und schickt dann das ganze über die Leitung. Der Empfänger decodiert im ersten Schritt zu demjenigen Codewort (= Codierung eines 2-Bit-Blockes), welches vom empfangen Wort den kleinsten Abstand hat. Falls dabei mehrere den gleichen kleinsten Abstand haben, wählt er von ihnen ein beliebiges aus. Im zweiten Decodierungsschritt streicht er das Prüferbit.
Gibt es so einen Code?
Konstruiere ein Beispiel oder zeige, dass es keinen solchen geben kann!
Die Lösung findest Du auf Seite 41.

Aufgabe 4: Diesmal soll der Code 2-Bit-Blöcke nehmen, zwei Kontrollbits anhängen und sie in 4-Bit-Blöcke übersetzen. Ausserdem soll er (pro Block) einen Fehler korrigieren können. Gibt es so einen Code?
Lösung: s. Seite 41.

Man kann auf ähnliche Art zeigen, dass es keinen Code gibt, der 2-Bit-Blöcke auf irgendeine Art in 4-Bit-Blöcke übersetzt und dabei einen Fehler korrigiert. Auch mit der Uebersetzung von 3-Bit Blöcken in 5-Bit-Blöcken funktioniert es nicht. So ist die Idee, es mit 4 und 7 zu probieren gar nicht mal so exotisch.

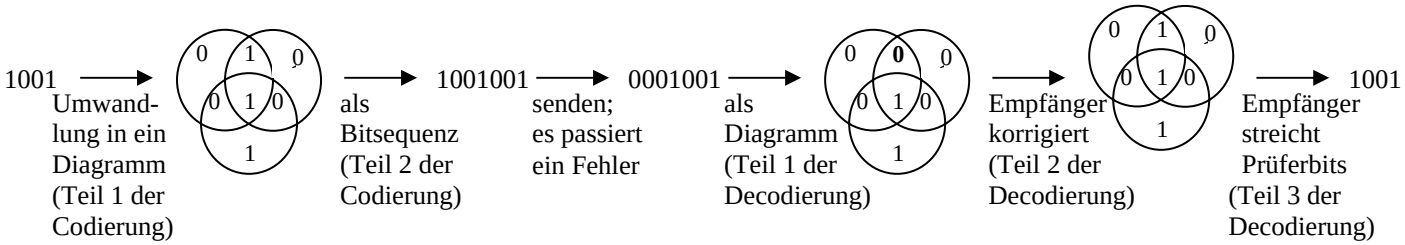
Realisierung: Der 3-Kreis-Hamming-Code lässt sich mit folgendem Diagramm illustrieren:



Der Sender: Er trägt die 4 Datenbits nacheinander in die Felder 1 bis 4 ein, wobei das erste Bit ins Feld 1 und das letzte ins Feld 4 kommt. Die **Prüferbits** (Bit 5 bis 7) wählt er so, dass die Summe aller Zahlen in jedem Kreis gerade ist. Danach schickt er den so konstruierten Block zum Empfänger.

Der Empfänger: Sobald er einen Block erhält, schreibt er die 7 Bits in die entsprechenden Felder des Diagramms, wobei das erste Bit ins Feld 1 und das letzte ins Feld 7 kommt. Dann prüft er, ob die Summe aller Zahlen in jedem Kreis gerade ist. Falls ja, nimmt er an, dass kein Fehler passiert ist, und decodiert den Block zu seinen ersten 4 Bits, d.h. er streicht die Prüferbits.
Falls nein, nimmt er – optimistischerweise – an, dass nur ein Fehler passiert ist. Dann sucht er dasjenige Bit, das er ändern muss, um in allen Kreisen ein gerade Summe zu erhalten. (Dass dies immer möglich ist, werden wir gleich zeigen.) Mit andern Worten: Der Empfänger sucht dasjenige "gültige" Diagramm, welches sich vom erhaltenen Diagramm an genau einer Stelle unterscheidet (gültig = die Summe aller Zahlen in jedem Kreis ist gerade).

Beispiel:



In diesem Fall konnte ein Fehler korrigiert werden.
Der Einfachheit halber benutzen wir im Folgenden die Formulierung 'ein Kreis hat gerade Summe'. Damit meinen wir 'die Summe aller Zahlen in diesem Kreis ist gerade'.

CHECK POINT II

Nun bist Du (wie wohl kaum zu übersehen) am sogenannten Checkpoint angelangt. Versuche, die untenstehenden Fragen zu beantworten.

1. Codiere die Bitfolge 1011 in den 3-Kreis-Hamming-Code

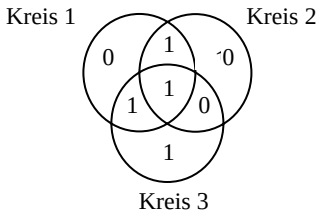
Weisst Du nicht weiter? Dann lies noch mal den Abschnitt 'Der Sender' genau durch.

2. Decodiere die Bitfolge 1110111 (im 3-Kreis-Hamming-Code)

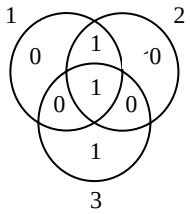
Weisst Du nicht weiter? Dann lies noch mal den Abschnitt 'Der Empfänger' genau durch

Hast Du je eine Lösung? Dann schaue nach auf Seite 34.

Zurück zum eigentlichen Thema:
Wie findet der Empfänger nun das Bit, welches er ändern muss, um alle Kreis-Summen gerade werden zu lassen? Nehmen wir mal an, das Diagramm des Empfängers sieht folgendermassen aus:



In diesem Fall haben die Kreise 1 und 3 eine ungerade Summe, Kreis 2 hat eine gerade Summe. Um alle Kreis-Summen gerade werden zu lassen, müssen wir in Kreis 1 und Kreis 3 je ein Bit ändern, in Kreis 2 sollte aber alles beim alten bleiben. Die einzige Möglichkeit dies zu erreichen (wenn wir nur genau ein Bit ändern wollen) ist, das Bit in der Schnittmenge von Kreis 1 und Kreis 3 (aber nicht Kreis 2!) zu verändern
Also



Allgemein gesagt: Wir müssen dasjenige Bit ändern, welches in der Schnittmenge von allen Kreisen mit ungerader Summe enthalten ist, aber in keinem Kreis mit gerader Summe vorkommt.

Aufgabe 5:

a) Der Empfänger erhält die Sequenz 0010110. Wie decodiert er?
b) Der Empfänger erhält die Sequenz 1100110. Wie decodiert er?
c) Der Empfänger erhält die Sequenz 0001111. Wie decodiert er?
d) Der Empfänger erhält die Sequenz 1001100. Wie decodiert er?

Lösung: Siehe Seite 41

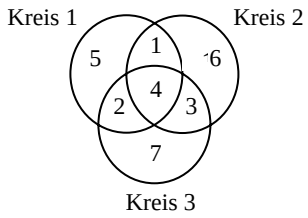
Wir haben also jeweils genau eine Möglichkeit, das zu ändernde Bit zu wählen.
Mit andern Worten:
Für jedes "ungültige" Diagramm (ungültig = nicht alle Kreise haben gerade Summe) gilt:
Es gibt genau ein gültiges Diagramm, welches sich von ihm an genau einer Stelle unterscheidet.
Nun wollen wir noch zeigen, dass der 3-Kreis-Hamming-Code **einen Fehler korrigieren** kann.
Erinnerung: Ein Code kann einen Fehler korrigieren, wenn gilt:
Falls genau ein Fehler aufgetreten ist, wird der Empfänger den entsprechenden Block zu der richtigen (= ursprünglichen) Bitsequenz decodieren.
Nehmen wir mal an, der Sender *A* schickt dem Empfänger *B* eine Nachricht. Dazu übersetzt er diese Nachricht in ein Diagramm und schickt die entsprechende Bitfolge an *B*. Nehmen wir weiter an, dass beim Uebermitteln genau 1 Fehler passiert. *B* erhält somit eine fehlerhafte Bitfolge und damit ein fehlerhaftes Diagramm. Taufen wir es D_{empf}. D_{empf} ist nun kein gültiges Diagramm. Der Grund: Wenn man in einem gültigen Diagramm genau ein Bit ändert, dann hat mindestens ein Kreis ungerade Summe. (Ueberlege, warum dem so ist.)
Da beim Senden genau ein Fehler passiert ist, unterscheidet sich D_{Send} an genau einer Stelle von $D_{empf.}$
Vom letzten Abschnitt wissen wir: Es gibt genau ein gültiges Diagramm, welches sich von D_{empf} an genau einer Stelle unterscheidet.
Dieses Diagramm kann also nur D_{Send} sein! *B* wird somit richtig decodieren, der Fehler wird behoben.
Also wird *B* das Bit *b* ändern und so zum D_{send} erhalten. Der Fehler wird also korrigiert!
Beachte dazu auch die Tabelle zu Aufgabe 2 auf Seite 15.

Was ziehen wir für eine **Bilanz**?
Ganz am Anfang dieses Unterkapitels suchten wir einen Code mit der folgenden Eigenschaft: 'Die einzelnen Codewörter unterscheiden sich so stark voneinander, dass ein Fehler korrigiert werden kann'.
Wie wir gerade eben gezeigt haben, erfüllt der 3-Kreis-Hamming-Code diese Bedingung – und ist damit in unserem Kontext ein 'guter Code'.

1.4 Zeig was Du kannst

Nun kannst Du testen, wie gut Du das Gelernte im Griff hast. Löse dazu die Aufgaben 1 bis 5.
Die Lösungen stehen auf Seite 36.
Falls Du die ersten fünf Aufgaben richtig gelöst hast, dann bist Du bereit für den Kapiteltest (melde dich dazu beim Tutor).
Sonst empfiehlt es sich, den jeweiligen Knackpunkt nochmals durchzugehen.
Die Aufgaben 6 und 7 sind **freiwillige Knobelaufgaben**.

Zur Erinnerung: Das Diagramm für den 3-Kreis-Hamming-Code sieht folgendermassen aus:



1. Der Sender möchte die Sequenz 1000 übermitteln und codiert sie (korrekt) zu 1000110. Beim Uebermitteln passieren zwei Fehler, so dass der Empfänger die Sequenz 1110110 erhält. Wie wird er decodieren?

2. Du weisst ja bereits wie man einen 7-Bit Block decodieren kann (im 3-Kreis-Hamming-Code). Nun ist es etwas mühsam, jedes mal das Diagramm aufzuzeichnen und herauszufinden, welches Bit man ändern muss. Das Problem kann man umgehen, indem man eine Tabelle aufstellt und dort genau notiert, welches Bit man in welchem Fall ändern muss.
   Deine Aufgabe: Fülle die folgende Tabelle aus!

Kreissumme ?	Kreis 1	Kreis 2	Kreis 3	zu änderndes Bit
	gerade	gerade	gerade	keines!
	gerade	gerade	ungerade	7
	gerade	ungerade	gerade	
	gerade	ungerade	ungerade	
	ungerade	gerade	gerade	
	ungerade	gerade	ungerade	
	ungerade	ungerade	gerade	
	ungerade	ungerade	ungerade	

3. <u>Fehlerwahrscheinlichkeiten und Zuverlässigkeit</u>

Als Aufwärmung folgt eine **kleine** Auffrischung der Wahrscheinlichkeitstheorie.
Falls Du Dich in diesem Bereich sicher & sattelfest fühlst, kannst Du diese Repetition überspringen.

Auffrischung Wahrscheinlichkeitstheorie

Beim Ausrechnen von Wahrscheinlichkeiten kann es hilfreich sein, sogenannte "Wahrscheinlichkeitsräume" zum Modellieren von Zufallsexperimenten zu betrachten. Ein Wahrscheinlichkeitsraum ist ein Paar $\Omega = (S,Prob)$, wobei S den **Ereignisraum** und $Prob$ die **Wahrscheinlichkeitsverteilung** bezeichnet. Zur Erinnerung: Der Ereignisraum S entspricht der Menge aller möglichen Ergebnisse des Experimentes, die Wahrscheinlichkeitsverteilung $Prob$ ordnet jedem solchen Ergebnis aus S eine Wahrscheinlichkeit zu.
Die möglichen Ergebnisse des Experimentes heissen auch **Elementarereignisse**.

Beispiel: Unfairer Würfelwurf, bei dem die "6" mit W'keit (=Wahrscheinlichkeit) 0.5 fällt und alle andern Zahlen mit W'keit 0.1 fallen. In diesem Fall ist
S = {"1","2","3","4","5","6"} und
$Prob("6") = 0.5$, $Prob("1") = Prob("2") = Prob("3") = Prob("4") = Prob("5") = 0.1$

Wir wissen:

- Für jedes Ereignis $A \subseteq S$ gilt: Prob(A) ist die Summe der Wahrscheinlichkeiten aller Elementarereignisse aus A.
  (Beispiel: Das Ereignis, dass eine gerade Zahl fällt, entspricht der Menge {2,4,6}. Die Wahrscheinlichkeit, dass dieses Ereignis eintritt ist demnach gleich
  $Prob(\{2,4,6\}) = Prob("2") + Prob("4") + Prob("6") = 0.1 + 0.1 + 0.5 = 0.7$)

- Die Summe der Wahrscheinlichkeiten aller Elementarereignisse in S ergibt 1 (da $Prob(S) = 1$)

- Für jedes Ereignis $A \subseteq S$ gilt: $Prob(A) = 1 - Prob(\bar{A})$, wobei $\bar{A}$ (= S\A) das Komplementärereignis von A ist.

- Die mehrfache (sagen wir k-fache) Wiederholung eines Experimentes $\Omega = (S,Prob)$ ist delbst wieder ein Experiment. Nennen wir es $\Omega_k = (S_k,Prob_k)$. Es gilt:

  - S_k besteht aus allen k-Tupeln der Form $(a_1,a_2, \ldots ,a_k)$, wobei a_i das Ergebnis der i'ten Durchführung von Ω ist.

  - $Prob_k(a_1,a_2, \ldots ,a_k) = Prob(a_1) \cdot Prob(a_2) \cdots Prob(a_k)$
    (in Worten: die Wahrscheinlichkeit eines Ergebnisses $(a_1,a_2, \ldots ,a_k)$ des k – stufigen Experimentes ist das Produkt der Wahrscheinlichkeiten der Resultate $a_1,a_2, \ldots ,a_k$ des Basisexperimentes.

Jetzt solltest Du fit genug sein, um die Aufgabe anzupacken!

Für die folgende Teil-Uebungen nehmen wir an, dass jedes Bit mit der Wahrscheinlichkeit von 0.1 invertiert (= beim Senden verändert) wird.

a) Angelina möchte Brad ein Bit übermitteln. Dazu verwendet sie den 3-Bit-Repetitions-Code.

Wie gross ist die Wahrscheinlichkeit, dass Brad die Nachricht zum richtigen (= von Angelina codierten) Bit decodiert?

Tipp: Eine (recht nützliche) Möglichkeit den Wahrscheinlichkeitsraum zu modellieren ist die folgende: Wie die gesendeten (und empfangenen) Bits genau aussehen, spielt keine Rolle. Uns interessiert nur, welche Bits bei der Uebertragung verfälscht (= invertiert) wurden.
Mit *R* bezeichnen wir die **r**ichtig übertragenen Bits, mit *F* bezeichnen wir die **f**ehlerhaft übertragenen Bits. *RFF* bedeutet dann beispielsweise: Bei Bit 2 und 3 (nicht aber bei Bit 1) ist ein Uebertragungsfehler passiert.
Der Ereignisraum beinhaltet dann alle 3-Tuppel bestehend aus *R* und *F*.

b) Brad hat sich sehr über die kleine Nachricht gefreut und möchte nun Angelina zurückschreiben. Seine Antwort umfasst 4 Bits. Auch er verwendet den 3-Bit-Repetionscode.
Wie gross ist die Wahrscheinlichkeit, dass Angelina (nach dem Decodieren) die richtige Nachricht in den Händen hält?

Tipp: Wir könnten theoretisch vorgehen wie in Aufgabe a). Dies würde allerdings die Rechnung unnötigerweise sehr verkomplizieren. Ob die einzelnen Bits richtig oder falsch übertragen werden, kann uns jetzt egal sein. Uns interessiert nur noch die Wahrscheinlichkeit dafür, dass eine (1-Bit-)Nachricht richtig decodiert wird. Diese haben wir bereits in a) berechnet. Und nun haben wir 4 solche 1-Bit-Nachrichten...

c) Angelina lässt nicht lumpen und denkt sich ebenfalls ein Antwort-Textchen aus. Es umfasst auch 4 Bits. Allerdings probiert sie jetzt den 3-Kreis-Hamming-Code aus.
Wie gross ist jetzt die Wahrscheinlichkeit, dass die von Brad decodierte Nachricht die richtige ist?

Hinweis: Für diese Aufgabe kannst Du benutzen (ohne dies zu beweisen) dass beim Auftreten von 2 oder mehr Fehlern ein Block **in jedem Fall** falsch decodiert wird. (Siehe dazu auch Aufgabe 6.)

4. In dieser Aufgabe wird jedes Bit mit der Wahrscheinlichkeit 0.2 invertiert.

a) Betrachten wir den 5-Bit-Repetitionscode. Wie gross ist die Wahrscheinlichkeit, dass eine 1-Bit-Nachricht richtig decodiert wird?

b) **A**lice möchte eine Nachricht bestehend aus 10 Bits an ihre Freunde **Ba**sil, **Be**rnd, **Bi**ll, **Bo**b und **Bu**ddy schicken. Um die Sache nicht allzu langweilig zu gestalten, wählt sie für jeden eine eigene Codierung.

1. Für **Ba**sil versendet sie den 3-Bit-Repetitionscode.
Wie gross ist die Wahrscheinlichkeit, dass er am Schluss die richtige Nachricht in den Händen hält?
Tipp: Berechne zuerst die Wahrscheinlichkeit, dass eine 1-Bit-Nachricht richtig decodiert wird. Hier haben wir dann 10 solche Nachrichten...

2. **Be**rnd bekommt die Nachricht im 5-Bit-Repetitionscode zugeschickt.
Wie gross ist die Wahrscheinlichkeit, dass er zur richtigen Nachricht decodiert?

3. Bei **Bi**ll entscheidet sie sich für ein Mittelding: Sie codiert abwechslungsweise ein Bit im 3-Bit-Repetitionscode und ein Bit im 5-Bit-Repetitionscode.
   Konkret: Das erste Bit wiederholt sie 3 mal, das zweite 5 mal, das dritte wieder 3 mal etc...
   Wie gross ist jetzt die Erfolgswahrscheinlichkeit?
   (Erfolg = Beim Decodieren kommt die richtige Nachricht heraus.)

4. Für **Bo**ris verwendet sie den 3-Kreis-Hamming-Code – allerdings geht die Aufteilung in 4-er Blöcke nicht auf. So codiert sie die ersten zwei 4-er Blöcke (sprich: die ersten 8 Bits) mit dem 3-Kreis-Hamming-Code. Die restlichen 2 Bits schickt sie dann 'einfach so', ohne Codierung, über die Leitung.
   Wie gross ist jetzt die Erfolgswahrscheinlichkeit?
   Hinweis: Für diese Aufgabe kannst Du benutzen (ohne dies zu beweisen) dass beim Auftreten von 2 oder mehr Fehlern ein Block **in jedem Fall** falsch decodiert wird.

5. Für **Bu**ddy lässt sie sich schliesslich noch etwas kompliziertes einfallen:
   Die ersten drei Bits codiert sie mit dem 3-Bit-Repetitionscode, die nächsten 3 Bits mit dem 5-Bit-Repetitionscode und die letzten 4 Bits mit dem 3-Kreis-Hamming-Code.
   Wie siehts jetzt mit der Erfolgswahrscheinlichkeit aus?
   (Der Hinweis aus der obigen Teil-Aufgabe gilt hier auch)

5. Betrachten wir nun einen neuen Code. Der Empfänger sendet dabei jedes Bit zweimal. Die Decodierungsregeln sind wie folgt: Erhält der Empfänger eine Sequenz mit mehr '0' als '1' oder umgekehrt, dann wendet er den üblichen Mehrheitsentscheid an. Enthält die Sequenz jedoch genau gleich viele '0' und '1', dann wirft er eine Münze. Fällt Kopf, dann decodiert er zu '1', andernfalls decodiert er zu '0'.
   Jedes Bit wird mit der Wahrscheinlichkeit 0.1 invertiert
   Wie gross ist die Wahrscheinlichkeit, dass eine 1-Bit-Nachricht richtig decodiert wird?

6. Knobelaufgabe

Wir betrachten nun den 3-Kreis-Hamming-Code.
Zeige: Falls bei der Uebertragung eines Blocks 2 oder mehr Fehler auftreten, wird er in jedem Fall falsch decodiert.

Hinweis: Unterscheide die folgenden 3 Fälle:

1. Mindestens zwei der Datenbits (= Bits 1 bis 4) werden invertiert
2. Nur Prüferbits (= Bits 5 bis 7) werden invertiert
3. Genau ein Datenbit und (mindestens) ein Prüferbit werden invertiert

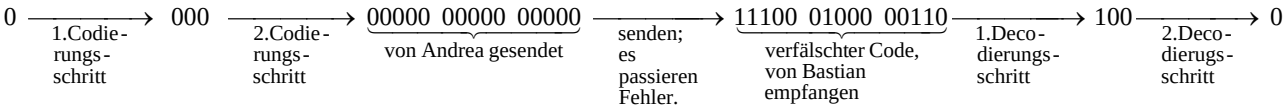
7. Knobelaufgabe

Andrea möchte Bastian Nachrichten schicken. Dazu basteln die beiden ein neues Codierungssystem. Codierung und Decodierung benötigen dabei nun je zwei Schritte.
Codierung: Im ersten Schritt codiert Andrea die Nachricht mit dem 3-Bit-Repetitionscode.

Im zweiten Schritt nimmt sie das Ergebnis und codiert es mit dem 5-Bit-Repetitionscode.

Decodierung: Im ersten Schritt decodiert Bastian gemäss dem 5-Bit-Repetitionscode.
Im zweiten Schritt nimmt er das Ergebnis und decodiert es gemäss dem 3-Bit-Repetitionscode.

Bsp:



Deine Aufgabe:

a) Finde ein Beispiel für eine fehlerhafte Uebertagung, die in diesem Code, nicht aber im 15-Bit-Repetitionscode, zu einem richtigen Resultat führt.

b) Finde ein Beispiel für eine fehlerhafte Uebertagung, die im 15-Bit-Repetitionscode, nicht aber in diesem Code, zu einem richtigen Resultat führt

2. Allgemeine Aussagen

Im letzten Kapitel wurden zwei exemplarische Codes vorgestellt. Nun möchten wir den Begriff 'Code' verallgemeinern und ein wenig formalisieren.
Zunächst werden wir das verallgemeinerte Konzept vorstellen und zeigen wie sich die beiden Codes vom letzten Kapitel darin einbetten lassen.
Danach geht es um die Frage: "Wie viele Fehler kann ein (gegebener) Code korrigieren?" Als Antwort werden wir eine Formel herleiten.
Zu guter Letzt betrachten wir eine spezielle Klasse von Codes. Sie besitzen eine spezielle Struktur und verfügen deshalb über besondere Eigenschaften. Als Clou werden wir zum Schluss eine verblüffend einfache Formel herleiten, um die Anzahl korrigierbarer Fehler in solchen Codes zu bestimmen.

Lernziele

1. Die Formel für die Anzahl korrigierbaren Fehler kennen und verstehen
2. Lineare Codes kennen
3. Etwas mehr Uebung im Umgang mit Beweisen

2.1 Verallgemeinertes Konzept

Im letzten Kapitel haben wir nur Codes betrachtet, welche Bitsequenzen in längere Bitsequenzen "übersetzen".
In diesem Kapitel werden wir das nicht mehr so eng sehen. Wir definieren einen Code als **Menge von gleichlangen Bitsequenzen.** Diejenigen Bitfolgen, die im Code vorkommen, nennen wir **Codewörter**.

Beispiel: C = {00011, 00100, 11010} ist ein Code mit 3 Codewörtern.
Man kann sich vorstellen, dass jedes dieser Codewörter eine Bedeutung hat, so steht vielleicht 00011 für 'rot', 00100 für 'gelb' und 11010 für 'blau'. Man kann auch sagen: Eine Nachricht (bestehend aus 'rot', 'gelb', 'blau') wird in Codewörter übersetzt.
(Natürlich könnte man die 3 Codewörter auch mit jeweils 2 Bits darstellen, aber eine grössere Länge hat – wie wir später sehen werden – durchaus ihre Vorteile.)

Eine Decodierungsstrategie legen wir auch noch fest: Der Empfänger decodiert jeweils zu dem Codewort, welches sich vom empfangenen Wort an möglichst wenigen Stellen unterscheidet.
Nun möchten wir noch kurz die beiden Codes vom letzten Kapitel aufgreifen und zeigen, dass sie sich ins verallgemeinerte Konzept einbetten lassen.
Der ***n*-Bit Repetitionscode** besteht – formal gesprochen – aus den beiden Codewörtern 00...0 und 11....1. Erhält der Empfänger ein Wort, vergleicht er die Anzahl '0' und '1' und fällt einen 'Mehrheitsentscheid' (sprich: falls mehr '1' als '0' vorkommen, interpretiert er es als '11...1' , andernfalls interpretiert er es als '00...0'). Die obige Decodierungs-Strategie wird also eingehalten.
Der **3-Kreis-Hammingcode** besitzt als Codewörter alle 7-Bit-Sequenzen, die ein gültiges Diagramm repräsentieren.
Erhält der Empfänger eine Bit-Sequenz, die nicht als Codewort figuriert, ändert er ein (bestimmtes) Bit und erhält so ein gültiges Codewort.
Oder anders gesagt: Er sucht dasjenige Codewort, welches sich vom empfangenen Wort an genau einer Stelle unterscheidet. Dies ist automatisch dasjenige Codewort welches sich vom empfangenen Wort an den wenigsten Stellen unterscheidet. Auch hier wird also die obige Decodierungsstrategie angewendet.

2.2 Wie viele Fehler kann ein gegebener Code korrigieren?

Im folgenden werden wir eine Formel herleiten, die uns verrät, wie viele Fehler ein Code korrigieren kann.
Um unsere Formulierungen klar und verständlich zu halten (und uns vor umständlichen Bandwurmsätzen zu verschonen), werden wir in Zukunft die folgenden nützlichen Begriffe verwenden.

Definition: Der **Abstand** zwischen zwei Codewörtern bezeichnet die Anzahl Stellen an denen sie sich unterscheiden. (Beispiel: 00111 und 10101 haben voneinander Abstand 2.)

Definition: Der **Minimalabstand** eines Codes entspricht dem kleinstmöglichen Abstand, die zwei Codewörter voneinander haben. Oder formal ausgedrückt:
Minimalabstand = min{Abstand von *a* und *b* : *a, b* sind versch. Codewörter}
(Beispiel: der Code C = {0001, 0011, 1100} hat Minimalabstand 1, da die ersten beiden Wörter Abstand 1 haben.)

Erinnerung: Wir sagen 'ein Code kann *k* Fehler korrigieren' und meinen: Wenn bei der Uebertragung eines Codewortes höchstens *k* Fehler passiert sind, dann wird der Empfänger richtig decodieren.
(Achtung: es kann durchaus vorkommen, dass mehr als *k* Fehler passieren und trotzdem richtig decodiert wird. Massgebend ist jedoch einzig, wie viele Fehler immer – also auch im kritischsten Fall – korrigiert werden können.)

CHECK POINT I

1. Berechne den Minimalabstand der folgenden Codes

a). C_1 = {00001, 00110, 11000}

b). C_2 = {01100, 10011, 11001, 10101}

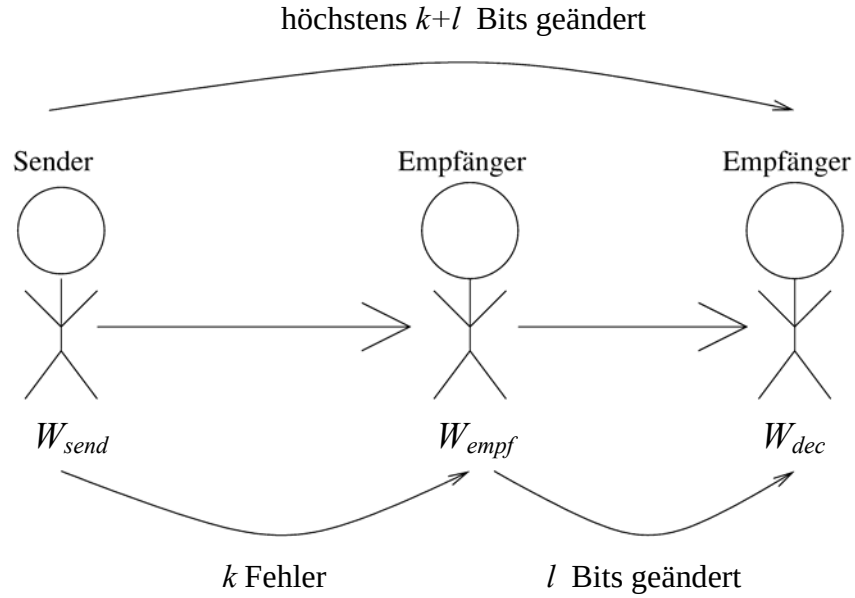
2. Zeige, dass der folgende Code keinen Fehler korrigieren kann
C_3 = {0111, 0100, 1001}

Hinweis: Finde ein Szenario, bei dem ein Fehler passiert und falsch decodiert wird.

Hast Du je eine Lösung? Dann schau nach auf Seite 45.

Wenn wir nun berechnen möchten wie viele Fehler ein gegebener Code korrigieren kann, könnten wir alle Codeworte mitsamt den möglichen Fehlerkombinationen durchgehen (und dann austesten, bis zu wie vielen Fehlern richtig decodiert wird). Das würde sicher funktionieren, wäre aber sehr mühsam und aufwendig.
Einfacher wäre es doch, wenn wir lediglich den Minimalabstand (sprich: den kleinstmöglichen Abstand, den zwei Codewörter voneinander haben) zu betrachten bräuchten

und daraus die Anzahl korrigierbare Fehler bestimmen könnten. Das funktioniert auch tatsächlich – mit Hilfe der folgenden Ueberlegungen:
Betrachten wir den Uebermittlungsvorgang mal etwas genauer:



Der **Sender** verschickt ein Codewort (bezeichnet mit W_{ges}).

Bei der **Uebertragung** passieren k Fehler. Das resultierende (=empfangene) Wort bezeichnen wir mit W_{empf} (möglicherweise ist dies kein Codewort).

Der **Empfänger** erhält W_{empf} und decodiert es zu dem Codewort mit dem kleinsten Abstand (bezeichnet mit W_{dec}). Die Anzahl Bits, die er dabei verändert bezeichnen wir mit l.
Insgesamt (= beim Uebertragen und beim Decodieren) werden also maximal $k+l$ Bits verändert.

Wir wissen: $l \le k$. (Denn W_{empf} kann man durch Invertierung der k 'Fehlerbits' wieder zu W_{ges} zurückcodieren. Somit ist das nächste Codewort höchstens k Bits entfernt.)

Gilt nun $\mathbf{2k < m}$ (bzw. $k < \frac{m}{2}$), dann können wir sagen: $k + l \le k + k = 2k < m$.
Somit wird sicher richtig decodiert, denn:
Um (von W_{ges} aus) zu einem anderen Codewort zu kommen, müsste man mindestens m Bits invertieren. Der Abstand zwischen W_{ges} und W_{dec} ($= l+k$) ist jedoch kürzer als m.
Also kommt bei der Decodierung wieder das richtige heraus.
Ist andererseits $2k \ge m$, dann ist eine falsche Decodierung durchaus möglich.

Zusammengefasst: Ein Code mit Minimaldistanz m kann k Fehler korrigieren $\Leftrightarrow k < \frac{m}{2}$.

CHECK POINT II

Es ist wieder einmal Zeit für einen Checkpoint. Versuche die untenstehenden Fragen zu beantworten.

1. Wie viele Fehler kann der folgende Code korrigieren?
   C = {000011, 001100, 11000, 101010}

   Weisst Du nicht weiter? Dann lies noch mal den letzten Abschnitt genau durch.

2. Konstruiere einen Code mit 3 Wörtern der Länge 5 (Bits), der einen Fehler korrigiert

   Weisst Du nicht weiter? Dann trau Dich, einfach mal zu 'pröbeln' (und lies gegebenenfalls noch mal den letzten Abschnitt genau durch).

Hast Du je eine Lösung? Dann schaue nach auf Seite 45.

2.3 Lineare Codes

In diesem Unterkapitel wollen wir – in einem gewissen Sinne – mit Codes 'rechnen'. Dazu benötigen wir ein Verfahren, um zwei Codewörter zusammenzuzählen. Doch wie kann man nun zwei Bit-Sequenzen addieren?
Für die Addition von zwei Bits definieren wir:

- 0 + 0 = 0
- 1 + 0 = 1
- 1 + 1 = 0

Zwei Codewörter (= Bitsequenzen) kann man nun addieren, indem man die einzelnen Bits addiert (ohne Uebertrag oder dergleichen).

Beispiel:

```
   1001
+  1101
=  0100
```

Achtung: Die obige Definition ist **eine mögliche** Art, eine Summe zu definieren. In einem andern Fall wird man vielleicht ein anderes Additions-Verfahren festlegen. (Vielleicht kennst Du schon das Zusammenzählen im 2-er System. Dort bildet man die Summe anders als oben beschrieben.) Wenn wir in diesem Kapitel von 'Summe' sprechen, meinen wir damit jedoch immer die eben eingeführte Addition.

Definition: Ein Code heisst **linear**, falls die Summe von zwei Codewörtern selbst wieder ein Codewort ist.

Beispiele: Der Code $C = \{000, 011, 101, 110\}$ ist linear, der Code $D = \{000, 001, 011, 100\}$ hingegen nicht (beispielsweise ist 100+011 kein Codewort).

Aufgabe: Sind die folgenden Codes linear?

a) $C_1 = \{0000, 0001, 0111, 0110, 1101, 1011\}$

b) $C_2 = \{0000, 0111, 1110, 1001\}$

Lösung: Siehe Seite 45

Diese simple Bedingung führt dazu, dass lineare Codes einige besondere Gesetzmässigkeiten aufweisen. Zum Beispiel lässt sich die Anzahl korrigierbarer Fehler nun mit einer überraschend einfachen Formel bestimmen. Diese werden wir im Folgenden herleiten. Zur Vorbereitung benötigen wir nun noch ein paar Hilfs-Beobachtungen.

Beobachtung 1: Das Nullwort (= die Bitsequenz 00...0) kommt in jedem Code vor. (Begründung: Addiert man ein beliebiges Codewort mit sich selbst, kommt 00...0 heraus.)

Betrachten wir diese Summe mal etwas genauer.
Wir wissen: Die Summe von zwei Bits ist genau dann gleich 1, wenn die beiden Bits unterschiedliche Werte haben (denn, wie am Anfang des Unterkapitels definiert, ist 0 + 0 = 0, 1 + 1 = 0, 1 + 0 = 1). Dies führt uns gleich zur nächsten Feststellung.

Beobachtung 2: Die Summe von zwei Codewörtern (was genau den Summen der jeweiligen Bits entspricht) hat genau an den Stellen eine '1', an denen sich die beiden Summanden unterscheiden.

Beispiel:

```
  1010
+ 1111
= 0101
```

Die beiden Summanden unterscheiden sich im 2. und im 4. Bit

Die Anzahl '1' in der Summe von zwei Codewörtern entspricht also genau deren Abstand. (Zur Erinnerung: Abstand = Anzahl Stellen, an denen sich diese beiden Codewörter unterscheiden.)

Um uns nicht in komplizierten Formulierungswirren zu verstricken, werden wir in Zukunft die folgenden beiden Begriffe verwenden.

Definition: Das **Gewicht** eines Codewortes entspricht der Anzahl '1', die darin vorkommen.
Das **Minimalgewicht** eines Codes ist das kleinstmögliche Gewicht das von einem seiner Codewörter (das Nullwort mal ausgenommen) besessen wird.
(Formal:
Minimalgewicht = min{Gewicht von a : a ist ein Codewort aber nicht das Nullwort}.)

Nach all diesen Vorbereitungen präsentieren wir jetzt das versprochene Ergebnis.

Wir behaupten:

Der Minimalabstand eines linearen Codes ist gleich seinem Minimalgewicht. (Zur Erinnerung: Minimalabstand = kürzestmöglicher Abstand zwischen zwei Codewörtern.)

Wir werden diese Behauptung in zwei Schritten zeigen. Zuerst beweisen wir, dass der **Minimalabstand** gleichzeitig ein Gewicht eines Codewortes ist. Danach zeigen wir, dass das **Minimalgewicht** gleichzeitig der Abstand zwischen zwei bestimmten Codewörtern ist. (Warum beweist dies, dass der Minimalabstand gleich dem Minimalgewicht ist? Bezeichnen wir den Minimalabstand mit d_{min} und das Minimalgewicht mit w_{min}. Da d_{min} ein Gewicht ist, gilt $w_{min} \leq d_{min}$. Und da w_{min} ein Abstand ist, gilt $d_{min} \leq w_{min}$. Somit gilt $d_{min} = w_{min}$.)

1.Schritt: Das Gewicht eines Codewortes ist tatsächlich sein Abstand vom Nullwort. Somit entspricht das Minimalgewicht dem Abstand vom Nullwort zum Codewort mit dem kleinsten Gewicht. Also ist das Minimalgewicht ein Abstand zwischen zwei bestimmten Codewörtern.

2.Schritt: Der Minimalabstand bezeichnet ja bekanntlich den kleinstmöglichen Abstand zwischen zwei Codewörtern. Weiter oben haben wir schon gesehen, dass der Abstand von zwei Codewörtern genau dem Gewicht (= der Anzahl '1') ihrer Summe entspricht. Da wir es mit linearen Codes zu tun haben, ist diese Summe

wieder ein Codewort. Jeder Abstand –und damit auch der Minimalabstand - entspricht also dem Gewicht eines bestimmten Codewortes.

Nun wissen wir: Der Minimalabstand eines linearen Codes entspricht seinem Minimalgewicht.

Im letzten Unterkapitel haben wir gesehen:

Ein Code mit Minimaldistanz m kann k Fehler korrigieren $\Leftrightarrow \; k < \frac{m}{2}$.

Und deshalb kann man schlicht und einfach sagen:

Ein linearer Code mit Minimalgewicht w kann k Fehler korrigieren $\Leftrightarrow k < \frac{w}{2}$.

2.4 Zeig was Du kannst

Nun kannst Du testen, wie gut Du das gelernte im Griff hast. Löse dazu die Aufgaben 1 bis 3. Die Aufgaben 6 und 7 sind freiwillige Knobelaufgaben. Die Lösungen stehen auf Seite 46. Falls Du die ersten fünf richtig gelöst hast, dann bist Du bereit für den Kapiteltest (melde dich dazu beim Tutor).
Sonst empfiehlt es sich, den jeweiligen Knackpunkt nochmals durchzugehen.

1. Wie viele Fehler können die folgenden Codes korrigieren?

   a) C = {0101, 1110, 1001}
   b) C = {00011, 11000, 11111, 00100}
   c) C = {000'000, 110'001, 001'111, 111'110} (Die Hochkommas dienen nur der Lesbarkeit.)
   d) C = {000'011, 110'010, 001'100, 111'101}

2. Sind die folgenden Codes linear?

   a) C = {0000, 0001, 0011, 0010}
   b) C = {00000, 10001, 01101, 11100, 11101}
   c) C = {0011, 0001, 0010}
   d) C = {00000, 10111, 01110, 11001}

3. Nehmen wir mal an, a und b seien beliebige Bitfolgen gleicher Länge.
   Mit N bezeichnen wir das Nullwort. Zeige: $C = \{N,a,b,a+b\}$ ist linear.

4. Betrachte den Code C = {0001, 1100, 1111}.
   Bestimme die Intimsphäre von 0001!

5. Wir haben in diesem Kapitel das Konzept der 'Fehlerkorrektur' kennen gelernt. Parallel dazu gibt es das Konzept der 'Fehlerdetektion': Der Empfänger **bemerkt** (= detektiert) einen Fehler, falls er eine Bitsequenz erhält, die nicht als Codewort figuriert. Er ist nicht beauftragt, zu decodieren.
   Wir sagen 'ein Code kann l Fehler **detektieren**', wenn gilt:
   Falls maximal l Fehler passiert sind, bemerkt der Empfänger, dass ein Fehler aufgetreten ist.
   Deine Aufgabe: Drücke (nur mithilfe des Minimalabstandes) aus, wie viele Fehler ein Code detektieren kann.

6. Freiwillige Knobelaufgabe

Konstruiere einen **linearen Code** mit 4 Codewörtern der Länge 6, welcher einen Fehler korrigiert.

7. Freiwillige Knobelaufgabe

Zeige: Der Code, der alle n-Bit Sequenzen mit geradem Gewicht enthält, ist linear (für alle n).

3. Additum

In diesem Kapitel geht es um folgende Probleme

- Gibt es einen Code mit 4 Codewörtern der Länge 8, der 3 Fehler korrigieren kann?
- Gibt es einen Code mit 3 Codewörtern der Länge 4, der einen Fehler korrigieren kann?

oder, allgemein formuliert:

- Gibt es einen Code mit *r* Codewörtern der Länge *n*, der *k* Fehler korrigieren kann?

Das Problem in dieser Allgemeinheit ist eine sehr komplexe, anspruchsvolle, theorielastige (vielleicht sogar noch nicht mal gelöste) Aufgabe und würde den Rahmen dieses Leitprogrammes bei weitem sprengen.
Wir wollen deshalb im Folgenden "bloss" eine **notwendige Bedingung** dafür herleiten, welche ein Code mit diesen Eigenschaften (sprich Länge *n*, *r* Codewörtern, *k* korrigierbaren Fehlern) erfüllen muss.
Ist unsere Bedingung also nicht erfüllt, dann kann es keinen solchen Code geben. Wir müssen daher gar nicht erst versuchen, ihn zu konstruieren.

Lernziel: Ein besseres Gefühl für die Struktur von Codes bekommen

Drehen wir zu Beginn mal eine kleine Aufwärmrunde. Unsere Codes brauchen vorerst keine Fehler zu korrigieren.

Frage 1: Wie viele Codewörter besitzt ein Code der Länge *n* höchstens?

Antwort 1: Es gibt 2^n Bitsequenzen der Länge *n* (denn für jedes Bit hat man 2 Möglichkeiten). Also besitzt ein Code der Länge *n* höchstens 2^n Codewörter.

Frage 2a) Wie lang muss ein Code (mindestens) sein, der 5 Codewörter enthält?

Antwort 2a) Es muss gelten: $5 \le 2^n$. Diese Ungleichung ist für $n \ge 3$ erfüllt. Der Code muss also (mindestens) Länge 3 haben.

Frage 2b) Wie lange muss ein Code (mindestens) sein, um 11 Codewörter zu enthalten?

Antwort 2b) Es muss gelten: $11 \le 2^n$. Der Code muss also mindestens Länge 4 haben.

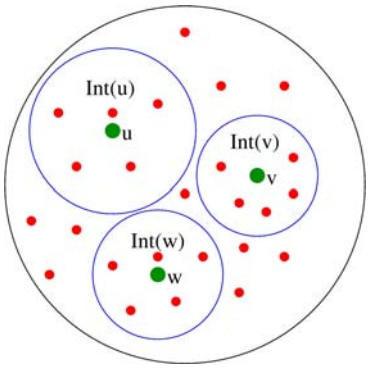
Frage 2c) Wie lange muss ein Code (mindestens) sein, um *r* Codewörter zu enthalten?

Antwort 2c) Es muss gelten $r \le 2^n$. Oder mit andern Worten: $\log_2(r) \le n$

Die Decodierungsregeln sind immer noch gleich wie in Kapitel 2: Erhält der Empfänger ein Wort *w*, dann decodiert er zu demjenigen Codewort, welches von *w* den kleinsten Abstand hat. Haben mehrere Codewörter den kleinstmöglichen Abstand zu *w*, dann kann der Empfänger davon ein beliebiges auswählen.
Noch ein Wort zur Decodierung: Betrachten wir eine Bitsequenz *w* und ein Codewort *c*. Wir sagen '*w* wird **eindeutig** zu *c* decodiert', wenn *w* zu jedem andern Codewörtern einen grösseren Abstand als zu *c* hat.
Um unsere Formulierungen verständlich und anschaulich zu halten, führen wir den folgenden (selbst kreierten) Begriff ein:

Definition: Die **Intimsphäre**[1] eines Codewortes *w* ist die Menge aller Bitsequenzen, die **eindeutig** zu *w* decodiert werden. Wir bezeichnen sie mit *Int*(*w*).
(Beispiel I: C_1 = {000,111}. Dann ist
Int(000) = {000, 001, 010, 100} und
Int(111) = {111, 011, 101, 110}.

Beispiel II: C_2 = {10, 01}. Dann ist
Int(10) = {10} und
Int(01) = {01}.
Achtung: 11 und 00 kommen in keiner Intimsphäre vor, denn ihre Decodierung ist nicht eindeutig. (11 hat von 10 und 01 den gleichen Abstand. Erhält der Empfänger 11, kann – je nach Lust und Laune – zu 10 **oder** 01 decodieren. Das gleiche gilt für 00.)



Veranschaulichung der Intimsphäre.
u,v,w sind Codewörter,
die roten Punkte representieren Bitsequenzen.

Sehr nützlich ist die Tatsache, dass alle Intimsphären voneinander disjunkt sind. (Wäre eine Bitsequenz in zwei Intimsphären vertreten, dann würde sie zu zwei verschiedenen Wörtern decodiert werden können. Ihre Decodierung wäre damit nicht eindeutig und somit dürfte sie in keiner Intimsphäre enthalten sein. Es kann also nicht sein, dass eine Bitsequenz in zwei Intimsphären vorkommt.) Was uns dieses Fakt bringt, werden wir etwas später sehen.

Aufgabe: Betrachte den Code *C* = {001, 110, 111} und bestimmt für jedes Codewort seine Intimsphäre. (Lösung: Siehe Seite 48.)

[1] Dieser Begriff ist weder eine Konvention noch ein allgemein anerkannter Ausdruck. In der Literatur wird dafür der Begriff 'Nachbarschaft' verwendet.

Wie viele Bit-Sequenzen enthält nun so eine Intimsphäre?
Betrachten wir einen Code *C*. Seine Länge bezeichnen wir mit *n*, die Anzahl Fehler, die er korrigieren kann, mit *k*.
Richten wir unser Augenmerk auf ein Codewort *w*. Wir wissen: Wenn beim Uebertragen höchstens *k* Fehler passiert sind, dann wird richtig decodiert. Also werden alle Bitsequenzen, die von *w* Abstand höchstens *k* haben, eindeutig zu *w* decodiert. Demzufolge enthält *Int*(*w*) alle Sequenzen, die von *w* höchstens Abstand *k* haben.
Die Intimsphäre eines Codewortes *w* enthält dann

- *w* selbst
- Alle Bitsequenzen, die sich von *w* an genau einer Stelle unterscheiden (= Abstand 1 haben)
- Alle Bitsequenzen, die sich von *w* an genau zwei Stellen unterscheiden (= Abstand 2 haben)

  .
  .
  .
- Alle Bitsequenzen, die sich von *w* an genau *k* Stellen unterscheiden
- eventuell noch einige Bitsequenzen mit grösserem Abstand

Nun sind wir schon fast fertig. Wir brauchen nur noch zu herauszufinden, wie viele Bitsequenzen sich von *w* an genau *l* Stelle unterscheiden (für ein gegebenes *l*).
Betrachten wir zuerst den Fall $l = 1$ (sprich: die Anzahl Bitsequenzen, die sich von *w* an genau einer Stelle unterscheiden). Es gibt *n* Möglichkeiten, die "unterschiedliche Stelle" zu wählen. Also gibt es *n* Sequenzen, die sich von *w* an genau einer Stelle unterscheiden.
Allgemein formuliert: Die Anzahl Möglichkeiten, die *l* unterschiedlichen Stellen zu wählen, entspricht genau der Anzahl Möglichkeiten, *l* Elemente aus *n* Elementen zu wählen. Und dies ist – wie aus der Kombinatorik bekannt – gleich $\binom{n}{l}$ ($= \frac{n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot (n-l+1)}{l!}$).

Und so gilt:

$$|Int(w)| \geq 1 + \binom{n}{1} + \binom{n}{2} + \ldots + \binom{n}{k}$$

Alle Intimsphären zusammen dürfen höchstens 2^n Bitsequenzen beinhalten. (Denn es gibt nur so viele Sequenzen der Länge *n*)
Weiter oben haben wir schon gesehen, dass alle Intimsphären voneinander disjunkt sind. Jedes Element kommt demnach in höchstens einer Intimsphäre vor. Daraus können wir schliessen $\sum_{w \in c} |Int(w)| \leq 2^n$.
Setzen wir nun die obige Formel (aus dem Kästchen) ein, erhalten wir:

$$\sum_{w \in c} (1 + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{k}) \leq 2^n .$$

Bezeichnen wir nun die Anzahl Codewörter mit *r*, so bekommen wir:

$$r \cdot (1 + \binom{n}{1} + \binom{n}{2} + \ldots + \binom{n}{k}) \leq 2^n .$$

Die krönende Schlussfolgerung lautet nun:
Ein Code mit *r* Codewörtern der Länge *n*, der *k* Fehler korrigiert, muss die obenstehende Bedingung erfüllen.

Beispiele: Kramen wir nochmal die Fragen vom Anfang des Kapitels hervor.

1.Gibt es einen Code mit 4 Codewörtern der Länge 8, der 3 Fehler korrigieren kann?

Antwort: Wir setzen in die Formel ein und erhalten $4 \cdot (1 + \binom{8}{1} + \binom{8}{2} + \binom{8}{3}) = 372$

Aber $2^8 = 256$.
Die Bedingung (sprich die Formel) ist also nicht erfüllt, es kann demnach keinen solchen Code geben.

2a). Gibt es einen Code mit 3 Codewörtern der Länge 4, der einen Fehler korrigieren kann?

Antwort: Wir setzen in die Formel ein und erhalten $3 \cdot (1 + \binom{4}{1}) = 15$ und $2^4 = 16$.

Die Bedingung (sprich die Formel) ist also erfüllt.
Gibt es nun aber wirklich einen Code mit diesen Eigenschaften? Dies können wir nicht aus dem Stegreif entscheiden (denn die Formel ist nur dann aussagekräftig, wenn sie nicht erfüllt ist, andernfalls tappen wir weiter im Dunkeln). Um wirklich herauszufinden, ob es solch einen Code gibt, müssen wir auf andere Methoden (z.B. pröbeln) ausweichen.

2b). Freiwillige Knobelaufgabe: Zeige, dass es keinen solchen (= in a). beschriebenen) Code gibt. Für die Antwort siehe Seite 48.

Was bringt uns nun diese hergeleitete Bedingung?
Wollen wir (für vorgegebene Länge, Anzahl Codewörter, Anzahl korrigierbare Fehler) einen Code konstruieren, dann können wir mithilfe unserer Formel schnell entscheiden, ob es überhaupt Sinn macht, solch einen Code zu suchen. Ist die Formel (sprich: die Bedingung) nicht erfüllt, dann wissen wir mit Sicherheit: Es kann keinen solchen Code geben. Wir können uns also die Anstrengung sparen, einen solchen Code zu konstruieren versuchen.
Ist die Formel allerdings erfüllt, sind wir nicht viel schlauer – es ist möglich, dass es einen solchen Code gibt, aber keineswegs sicher.
Unsere Bedingung ist also kein Zaubermittel, erspart uns aber unter Umständen viel Aufwand.

Zeig was Du kannst

1. Gibt es einen Code mit 4 Codewörtern der Länge 6, der zwei Fehler korrigieren kann?

2. Gibt es einen Code mit 5 Codewörtern der Länge 7, der zwei Fehler korrigieren kann?

3. Gibt es einen Code mit 6 Codewörtern der Länge 10, der drei Fehler korrigieren kann?

4. Gibt es einen Code mit 7 Codewörtern der Länge 12, der vier Fehler korrigieren kann?

5. **Knobelaufgabe**

   Suche einen Code kleinstmöglicher Länge, der
   - 3 Codewörter enthält und
   - zwei Fehler korrigieren kann.

6. Unsere Formel kann sehr hilfreich sein, wenn wir Codes mit vielen Codewörtern und grosser Länge betrachten. Wenn wir uns aber – aus irgendeinem Grund – auf zwei Codewörter beschränken wollen, dann geht es bedeutend einfacher.
   Deine Aufgabe:
   Finde eine Ungleichung (in der nur n und k vorkommen) die **genau dann** erfüllt ist, wenn es einen Code mit zwei Codewörtern der Länge n gibt, der k Fehler korrigiert.

7. Knobelaufgabe

Bis jetzt haben wir nur Bitfolgen betrachtet, d.h. Folgen die nur aus '0' und '1' bestehen. Betrachten wir nun mal Folgen, in denen '0', '1' **und** '2' vorkommen dürfen. Nennen wir sie **3-Folgen** (dies ist kein offizieller Begriff).
Bsp: '02100120', '0100', '022112' etc.
Analog kann man nun auch Codes betrachten, die – anstelle von Bitfolgen - aus solchen 3-Folgen bestehen. Wir bezeichnen sie mit **3-Codes**.
Der Abstand zweier solcher Folgen entspricht immer noch der Anzahl Stellen, an denen sie sich unterscheiden.

a) Formuliere die Ungleichung auf S.26 für 3-Codes um.
b) Auf eine ähnliche Art kann man 4-Folgen und 4-Codes definieren: In 4-Folgen dürfen nur '0', '1', '2', und '3' vorkommen und 4-Codes bestehen aus 4-Folgen.
   Wie lässt sich nun die Ungleichung formulieren?

Lösungen gibts auf Seite 48.

4. Lösungen

4.1 Lösungen Kapitel 1

4.1.1 CHECKPOINT I

Wir nehmen im Folgenden an, dass *n* gerade ist.

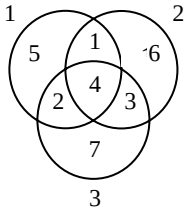
Solange weniger als $\frac{n}{2}$ Fehler passiert sind, wird der Empfänger immer richtig decodieren.

Passieren jedoch $\frac{n}{2}$ Fehler, dann wird unter Umständen falsch decodiert.

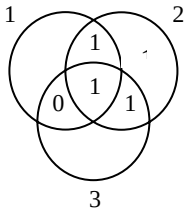
Der Code kann also $\frac{n}{2}-1$ Fehler korrigieren.

4.1.2 CHECKPOINT II

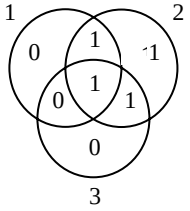
Zur Erinnerung: Das Diagramm für den 3-Kreis-Hamming-Code sieht folgendermassen aus:



1. (K2) Wir tragen zunächst die 4 Datenbits (= 1011) der Reihe nach in das Diagramm ein

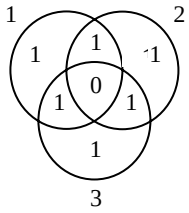


Danach wählen wir die restlichen Bits (= Prüferbits) so dass alle Kreise eine gerade Summe aufweisen.

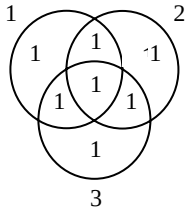


Die entsprechende Codierung ist dann 1011010

2. (K2) Wir erhalten die Sequenz 1110111 – oder als Diagramm ausgedrückt



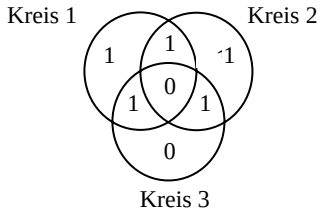
Nun bestimmen wir für jeden Kreis seine Summe. Dabei bemerken wir: Alle 3 Kreise haben eine ungerade Summe. Wenn wir nun das innerste Bit (dasjenige, das gleich 0 ist) invertieren, wird jede Kreis-Summe um 1 geändert. Damit sind nun alle Kreis-Summen gerade.



Wir decodieren also zu 1111.

4.1.3 Zeig was Du kannst

1. (K2) Der Empfänger erhält die Sequenz 1110110 – beziehungsweise das folgende Diagramm



Er berechnet alle Kreis-Summen und bemerkt, dass die Kreise 1 und 2 eine ungerade Summe aufweisen. Also muss er dasjenige Bit invertieren, welches in der Schnittmenge der Kreise 1 und 2, aber nicht in Kreis 3 enthalten ist. Dies ist Bit Nummer 1.
Der Empfänger decodiert also zu 0110. Der Uebertragungsfehler wird also **nicht** behoben.

2. (K3)

Kreissumme ?	Kreis 1	Kreis 2	Kreis 3	zu änderndes Bit
	gerade	gerade	gerade	keines!
	gerade	gerade	ungerade	7
	gerade	ungerade	gerade	6
	gerade	ungerade	ungerade	3
	ungerade	gerade	gerade	5
	ungerade	gerade	ungerade	2
	ungerade	ungerade	gerade	1
	ungerade	ungerade	ungerade	4

3a). (K3) Dieses Experiment entspricht der 3-fachen Wiederholung des Experimentes 'Sende ein Bit'. Wir arbeiten deshalb – wie im Tipp erwähnt – mit dem folgenden Wahrscheinlichkeitsraum. Der Ereignisraum S beinhaltet alle 8 (= 2^3) 3-Tupel, bestehend aus R und F. (R = das entsprechende Bit wurde richtig gesendet; F = das entsprechende Bit wurde fehlerhaft gesendet.) Konkret: S = {RRR, RRF, RFR, FRR, FRR, RFR, RRF, FFF}. Die Wahrscheinlichkeit für eines dieser Tupel richtet sich nur nach dessen Anzahl R und F. Betrachten wir ein bestimmtes Tupel t und bezeichnen die Anzahl R mit r und die Anzahl F mit f. Dann gilt:

$$Prob(t) = Prob(\text{ein Bit wird richtig gesendet})^r \cdot Prob(\text{ein Bit wird fehlerhaft gesendet})^f$$
$$= 0.9^r \cdot 0.1^f$$

Doch was sind nun die "guten Fälle" (= diejenigen, in denen richtig decodiert wird)? Genau diejenigen Uebertragungen, in denen höchstens ein Fehler passiert! (Werden zwei oder mehr Bits beim Uebertragen invertiert, dann decodiert der Empfänger zum falschen Bit.) Oder anders ausgedrückt: Die Tupel, in denen höchstens ein F vorkommt. Wir wissen:

- $Prob(RRR) = 0.9^3 = 0.729$

- Die drei andern "guten Fälle" (also RRF, RFR, FFR) treten jeweils mit W'keit $0.9^2 \cdot 0.1 = 0.081$ auf.

Die Wahrscheinlichkeit, dass ein "guter Fall" eintritt (sprich: richtig decodiert wird), beträgt demnach:
***Prob*(*richtig decodiert*) = 0.729 + 3 · 0.081 = 0.972**

3b). (K2) Die Wahrscheinlichkeit, dass eine 1-Bit-Nachricht richtig decodiert wird, haben wir schon in a). berechnet. Sie beträgt 0.972.
Brad möchte nun sozusagen 4 1-Bit-Nachrichten übermitteln. Damit die ganze Mitteilung richtig ankommt, muss jede 1-Bit-Nachricht richtig decodiert werden.
Also gilt:
***Prob*(4-Bit-Nachricht richtig decodiert) = *Prob*(*1-Bit-Nachricht richtig decodiert*)^4**
$= 0.972^4 = 0.89$

3c). (K3) Wir gehen ähnlich vor wie in a). Als Elementarereignisse betrachten wir alle 2^7 7-Tupel bestehend aus *R* und *F*. Betrachten wir ein Tupel *t* und bezeichnen die Anzahl *R* mit *r* und die Anzahl *F* mit *f*. Dann gilt:
$Prob(t) = Prob(\text{ein Bit wird richtig gesendet})^r \cdot Prob(\text{ein Bit wird fehlerhaft gesendet})^f$
$= 0.9^r \cdot 0.1^f$

Die "guten Fälle" sind nun diejenigen Uebertragungen, bei denen höchstens ein Fehler passiert - sprich: diejenigen Tupel, bei denen höchstens ein *F* vorkommt. (Bei zwei oder mehr Fehlern wird ja – wie im Hinweis angegeben – falsch decodiert).
Wir wissen:

- $Prob(\text{kein Fehler}) = Prob(RRRRRRR) = 0.9^7 = 0.48$
- Mit t_i bezeichnen wir das Tupel, welches an der Stelle *i* ein *F* und an allen andern Stellen ein *R* hat. Es gilt:
  $Prob(t_i) = 0.9^6 \cdot 0.1 = 0.05$
  Es gibt nun 7 Möglichkeiten, die Stelle für das *F* zu wählen. Also:
  $Prob(\text{es passiert genau ein Fehler}) = 7 \cdot Prob(t_i) = 7 \cdot 0.9^6 \cdot 0.1 = 0.37$

Und daraus können wir schliessen:
***Prob*(*richtig decodiert*) = *Prob*(kein Fehler) + *Prob*(genau ein Fehler)**
$= 0.48 + 0.37 = 0.85$

4a). (K3) Wir gehen gleich vor wie in Aufgabe 3. Der Ereignisraum beinhaltet nun alle 32 Tupel, bestehend aus *R* und *F*. Die "guten Fälle" sind die Uebertragungen, in denen höchstens zwei Fehler passieren – sprich diejenigen Tupel, in denen höchstens zwei *F* vorkommen. Diese lassen sich aufteilen in

- ein Tupel, das nur aus *R* besteht ($Prob(\text{kein Fehler}) = 0.8^5 = 0.36$)
- 5 Tupel, die aus vier *R* und einem *F* bestehen (haben jeweils W'keit = $0.8^4 \cdot 0.2$)
- $\binom{5}{2} = 10$ Tupel, die aus drei *R* und zwei *F* bestehen (jeweilige W'keit = $0.8^3 \cdot 0.2^2$)

***Prob*(*richtig decodiert*)**
= ***Prob*(kein Fehler) + *Prob*(genau ein Fehler) + *Prob*(genau zwei Fehler)**
= $0.8^5 + 5 \cdot 0.8^4 \cdot 0.2 + 10 \cdot 0.8^3 \cdot 0.2^2$
= 0.94208

4b.1). (K3) Die Wahrscheinlichkeit, dass eine 1-Bit-Nachricht erfolgreich übermittelt wird, lässt sich berechnen wie in Aufgabe 3a). Sie beträgt $0.8^3 + 3 \cdot 0.8^2 \cdot 0.2 = 0.896$. Alice sendet nun sozusagen zehn 1-Bit-Nachrichten. Damit die ganze Nachricht richtig ankommt, müssen alle zehn 1-Bit-Nachrichten richtig decodiert werden. Die Wahrscheinlichkeit für eine erfolgreiche Uebermittlung beträgt also:
***Prob*(10-Bit-Nachricht richtig) = *Prob*(*1-Bit-Nachricht richtig*)^{10}**
$= 0.896^{10} = 0.3335$

4b.2). (K2) Die Wahrscheinlichkeit, dass eine 1-Bit-Nachricht erfolgreich übermittelt wird, beträgt 0.94208. Dies haben wir schon in 4a). berechnet. Analog zur letzten Teilaufgabe können wir nun schliessen:
***Prob*(10-Bit-Nachricht richtig) = *Prob*(*1-Bit-Nachricht richtig*)^{10}**
$= 0.94208^{10} = 0.5507$

4b.3). (K2) Alice sendet nun zehn 1-Bit-Nachrichten – fünf davon im 3-Bit-Repetitionscode und fünf im 5-Bit-Repetitionscode. Damit die ganze Nachricht richtig ankommt, müssen alle 1-Bit-Nachrichten erfolgreich übermittelt werden. Die Wahrscheinlichkeit, dass eine 1-Bit-Nachricht erfolgreich übermittelt wird, haben wir in den vorigen Aufgaben berechnet. Sie beträgt 0.896 (3-Bit-Repetitionscode), bzw. 0.94208 (5-Bit-Repetitionscode). Also:
***Prob*(10-Bit-Nachricht richtig)**
= ***Prob*(*1-Bit richtig im 3-Rep-Code*)^5 · *Prob*(*1-Bit richtig im 3- Rep-Code*)^5**
= **$0.896^5 \cdot 0.94208^5$**
= **0.43**

4b.4). (K3) Zunächst berechnen wir die Wahrscheinlichkeit, dass ein 4-er Block im 3-Kreis-Hamming-Code erfolgreich übertragen wird. Wir tun das wie in Aufgabe 3c).
Die einzigen "guten Fälle" sind diejenigen Uebertragungen bei denen höchstens ein Fehler passiert, denn ab zwei Fehlern wird gemäss Hinweis immer falsch decodiert.
Wir wissen:

- *Prob*(kein Fehler) = $0.8^7 = 0.2097$
- Es gibt 7 Möglichkeiten, die Stelle für den Fehler auszusuchen. Es gibt also 7 Möglichkeiten für 'genau einen Fehler'.
- Alle haben jeweils W'keit = $0.8^6 \cdot 0.2 = 0.05243$

Die Wahrscheinlichkeit für eine erfolgreiche Uebertragung **eines** 4-er-Blocks beträgt
***Prob*(4-er Block in Hamming-Code richtig) = 0.2097 + 7 · 0.05242 = 0.58**
Alice übertragt nun 4 Nachrichten – zwei 4-er-Blöcke im 3-Kreis-Hamming-Code und zwei uncodierte Bits. Die W'keit, dass eine 1-Bit-Nachricht richtig ankommt, beträgt 0.8. Damit die ganze Nachricht richtig ankommt, müssen alle 4 Einzel-Nachrichten erfolgreich übertragen werden. Die Wahrscheinlichkeit dafür beträgt
***Prob*(Nachricht richtig) = *Prob*(4-er Block in Hamming-Code richtig)$^2 \cdot 0.8^2$**
$= 0.58^2 \cdot 0.8^2 = 0.22$

4b.5). (K3) Aus den vorhergehenden Teilaufgaben wissen wir

- Die Wahrscheinlichkeit, dass eine 1-Bit-Nachricht im 3-Bit Repetitionscode erfolgreich übermittelt wird, beträgt $0.8^3 + 3 \cdot 0.8^2 \cdot 0.2 = 0.896$.
- Die Wahrscheinlichkeit, dass eine 1-Bit-Nachricht im 5-Bit-Repetitionscode erfolgreich übermittelt wird, beträgt 0.94208.
- Die Wahrscheinlichkeit, dass eine 4-Bit-Nachricht im 3-Kreis-Hamming-Code erfolgreich übermittelt wird, beträgt 0.58

Die Wahrscheinlichkeit, dass die ganze Nachricht erfolgreich übertragen wird, beträgt
***Prob*(Nachricht richtig) = $0.896^3 \cdot 0.94208^3 \cdot 0.58 = 0.35$**

5. (K3) Die "guten Fälle" sind

- diejenigen Uebertagungen, bei denen kein Fehler passiert und
- diejenigen Uebertragungen, bei denen genau ein Fehler passiert und der Empfänger beim Münzwurf die 'richtige Seite' erhält.

Die Wahrscheinlichkeit, dass kein Fehler passiert, beträgt 0.9^2.
Die Wahrscheinlichkeit, dass genau ein Fehler passiert beträgt $2 \cdot 0.1 \cdot 0.9 = 0.18$.
Die Wahrscheinlichkeit, dass genau ein Fehler passiert und anschliessend die 'richtige Münzseite' erscheint, beträgt $0.5 \cdot 0.18 = 0.09$.

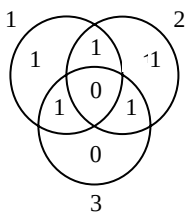
Die Wahrscheinlichkeit, dass eine 1-Bit-Nachricht erfolgreich übermittelt wird beträgt
***Prob*(Nachricht richtig) = *Prob*(kein Fehler) + *Prob*(genau ein Fehler und Münze ok)**
$= 0.9^2 + 0.09$
$= 0.9$

6. (K3) Zur Erinnerung nochmals der Hinweis:
Unterscheide die folgenden 3 Fälle:
1. Mindestens zwei der Datenbits (= Bits 1 bis 4) werden invertiert.
2. Nur Prüferbits (= Bits 5 bis 7) werden invertiert.
3. Genau ein Datenbit und (mindestens) ein Prüferbit werden invertiert.

Wir werden nun für jeden der 3 Fälle zeigen, dass falsch decodiert wird.

Fall 1: Ganz allgemein wissen wir: Der Empfänger ändert höchstens ein Bit, um ein gültiges Diagramm zu bekommen. Von den beiden invertierten Datenbits wird also höchstens eines auf den richtigen Wert "zurückgeändert".

Fall 2: Falls nur Prüferbits (und keine Datenbits) invertiert werden, dann gilt:
Ein Kreis hat ungerade Summe ⇔ sein Prüferbit wurde (beim Senden) invertiert.
(Beispiel:



)

In diesem Fall wurden die Prüferbits der Kreise 1 und 2 – sprich Bit 5 und 6 – verfälscht.)

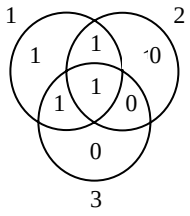
Falls also (beim Uebertragen) mindestens zwei Prüferbits verfälscht wurden, dann haben mindestens zwei Kreise eine ungerade Summe. Der Empfänger ändert also ein Bit in der Schnittmenge dieser zwei Kreise, sprich: ein Datenbit. Damit wird ein – richtig gesendetes – Datenbit invertiert. Der Empfänger decodiert also falsch (= nicht zur ursprünglichen Nachricht).

Fall 3: Wir unterscheiden zwischen den folgenden Diagrammen.

D_{Send}: Das Diagramm, welches der Sender verschickt
D_{Empf}: Das Diagramm, welches der Empfänger erhält
D_{Dec}: Das vom Empfänger decodierte Diagramm

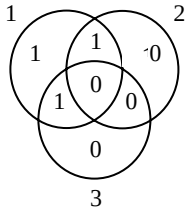
D_{Send} und D_{Dec} sind gültige Diagramme (= alle Kreise haben gerade Summe), D_{Empf} kann auch ungültig sein.

Der Sender schickt nun das Diagramm D_{Send}.
(Beispiel:



)

Beim Senden wird nun ein Datenbit invertiert (gemäss Bedingung von Fall 3). Nennen wir es Bit *i*.
(In unserem Beispiel wird Bit 1 invertiert.
Die Situation sieht zwischenzeitlich so aus:

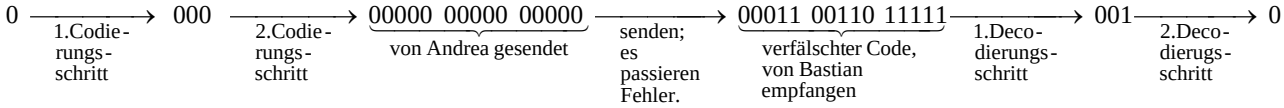


)

In dieser zwischenzeitlichen Situation haben alle Kreise, die Bit *i* beinhalten (in unserem Beispiel: die Kreise 1 und 2) ungerade Summe.
Um zu D_{Empf} zu gelangen, müssen wir noch ein Prüferbit invertieren (schliesslich lautet Fall Nr.3: Ein Datenbit und mindestens ein Prüferbit werden beim Uebertragen invertiert.)
Aendern wir ein Prüferbit in einem Kreis, der Bit *i* beinhaltet (in unserem Beispiel Kreis 1 oder 2), dann hat der entsprechende Kreis in D_{Empf} gerade Summe. Der Empfänger wird demzufolge beim Decodieren kein Bit aus diesem Kreis (und damit auch nicht Bit *i*) ändern.
Aendern wir dagegen ein Prüferbit in einem Kreis, der Bit *i* **nicht** beinhaltet (in unserem Beispiel Kreis 3), dann hat der entsprechende Kreis in D_{Empf} ungerade Summe. Der Empfänger wird also beim Decodieren ein Bit aus diesem Kreis ändern- und das kann sicher nicht Bit *i* sein.
Bit *i* wird also nicht mehr "zurückkorrigiert" und damit ist $D_{Dec} \neq D_{Send}$. Es wird also falsch decodiert.

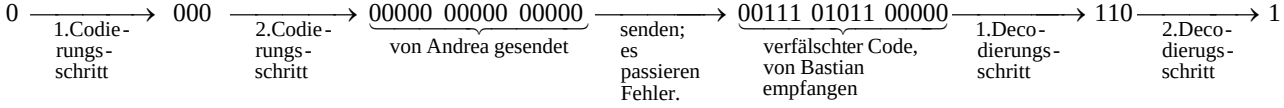
7. (K3) Wir nehmen an, dass das Bit '0' gesendet wird (andernfalls kann man die '0' und '1' einfach vertauschen).

a).



Bastian hält am Schluss das richtige Bit in den Händen. Jemand, der nach dem 15-Bit-Repetitionscode decodiert, hätte dagegen einen Mehrheitsentscheid gefällt. Da die empfangene Nachricht mehr '1' als '0' enthält, wäre am Schluss '1' – und somit das falsche Bit – herausgekommen.

b).



Bastian decodiert hier zum falschen Bit. Der Mehrheitsentscheid wäre jedoch zugunsten von '0' ausgefallen.

4.1.4 Sonstige Aufgaben

Aufgabe 1: Es gibt $2^4 = 16$ 4-Bit-Sequenzen und $2^7 = 128$ 7-Bit-Sequenzen

Aufgabe 2: Pro 4-Bit-Block braucht man 7 Sendebits. Pro 4 Datenbits benötigt man also 3 'Zusatzbits'. Pro Datenbit werden durchschnittlich ¾ Zusatzbits benutzt.

Aufgabe 3: Wir wollen zeigen, dass es keinen solchen Code geben kann.
Wenn für die 4 möglichen 2-Bit-Nachrichten nur ein Kontrollbit zur Verfügung steht, dann haben zwei solche 2-Bit-Nachrichten das gleiche Kontrollbit – sagen wir '0'. Nehmen wir an, die beiden Codewörter mit dem Kontrollbit '0' seien

- w_1 = *? ?* 0 und
- w_2 = *? ?* 0 (wobei Fragezeichen für irgendwelche Bits stehen)

Es gibt nun zwei Möglichkeiten:

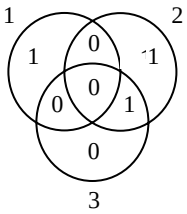
1. w_1 und w_2 unterscheiden sich an genau einer Stelle. Wenn beim Senden von w_1 nun an genau dieser Stelle ein Fehler passiert, erhält der Empfänger stattedessen w_2.

2. w_1 und w_2 unterscheiden sich an genau zwei Stellen. Wenn beim Senden von w_1 nun an einer dieser beiden Stelle ein Fehler passiert, dann erhält der Empfänger ein Wort, das sich sowohl von w_1 als auch von w_2 an genau einer Stelle unterscheidet. Dann können wir nicht mehr garantieren, dass er zu w_1 decodiert.

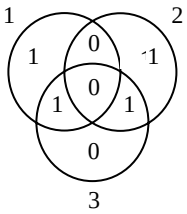
Aufgabe 4: Nun stehen zwei Kontrollbits (= 4 mögliche Kombinationen) zur Verfügung. Wir wollen nun wieder zeigen, dass es keinen solchen Code gibt. Betrachten wir nun irgendeinen Code, der jedem 2-Bit-Block ein Prüferbit anhängt. Wenn nun zwei Codewörter die gleichen beiden Kontrollbits haben, dann entsteht das gleiche Problem wie in Aufgabe 2. Der Code kann in diesem Fall keinen Fehler korrigieren. Alle vier Codewörter müssen also unterschiedliche "Kontrollbit-Kombinationen" haben (andernfalls kann der Code sowieso keinen Fehler korrigieren). Damit ein Fehler korrigiert werden kann, müssen sich je zwei Codewörter an mindestens drei Stellen unterscheiden (unterscheiden sich zwei Codewörter nur an zwei Stellen, dann muss beim Senden nur ein Fehler passieren und das empfange Wort hat von beiden den gleichen Abstand). Betrachten wir nun '00' und bezeichnen seine Kontrollbits mit *a* und *b*. Das entsprechende Codewort lautet nun 00*ab*. Wie sieht es nun mit der Codierung von '01' aus? Damit sie sich von 00*ab* an mindestens 3 Stellen unterscheidet, muss ihr erstes Kontrollbit ungleich *a* und das zweite Kontrollbit ungleich *b* sein. Bezeichnen wir mit $\bar{a}$ die Invertierung von *a*, d.h. $\bar{a} = 0$ falls $a = 1$ und $\bar{a} = 1$ falls $a = 0$. $\bar{b}$ entspricht der Invertierung von *b*. '01' muss also zu $01\bar{a}\bar{b}$ codiert werden. Doch auch die Codierung von '10' muss sich von 00*ab* an mindestens drei Stellen unterscheiden. Deshalb muss sie ebenfalls $01\bar{a}\bar{b}$ lauten. Doch nun haben zwei Codewörter die gleichen Kontrollbits. In Aufgabe 2 haben wir gesehen, dass in diesem Fall kein Fehler korrigiert werden kann. Deshalb kann auch in diesem Code kein Fehler korrigiert werden.

Aufgabe 5:

a) Der Empfänger erhält die Sequenz 0010110.
Das (zu 0010110) zugehörige Diagramm sieht so aus

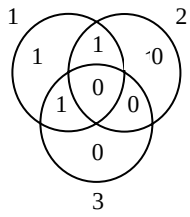


Kreis 1 und Kreis 3 haben dabei ungerade Summe.
Deshalb verändert der Empfänger dasjenige Bit, das in der Schnittmenge von Kreis1 und Kreis 3 (aber nicht in Kreis 2) liegt.
Das "korrigierte" Diagramm sieht dann so aus

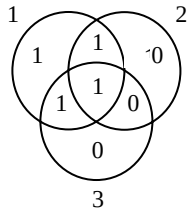


Der Empfänger decodiert also zu 0110.

b) Der Empfänger erhält die Sequenz 1100100.
Das zugehörige Diagramm sieht so aus

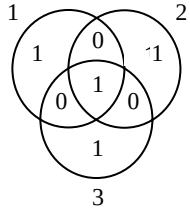


Alle Kreise haben dabei ungerade Summe.
Deshalb verändert der Empfänger dasjenige Bit, das in der Schnittmenge von Kreis 1,Kreis 2 und Kreis 3 liegt. Das korrigierte Diagramm sieht dann so aus



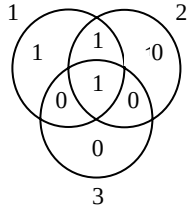
Der Empfänger decodiert also zu 1101.

c) Der Empfänger erhält die Sequenz 0001111.
Das zugehörige Diagramm sieht so aus

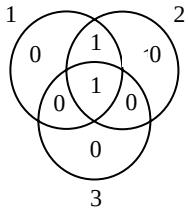


Alle Kreis haben gerade Summe! Der Empfänger ändert kein Bit und decodiert zu 0001.

d) Der Empfänger erhält die Sequenz 1100100.
Das zugehörige Diagramm sieht so aus



Kreis 1 hat als einziger eine ungerade Summe. Der Empfänger ändert deshalb dasjenige Bit, das im Kreis 1 und keinem andern liegt. Das korrigierte Diagramm sieht dann so aus



Der Empfänger decodiert somit zu 1100.

4.2 Lösungen Kapitel 2

4.2.1 CHECKPOINT I

1a). Alle Codewörter haben voneinander Abstand 3 oder mehr. Die ersten beiden Codewörter (00001 und 00110) haben voneinander Abstand 3.
Die Minimaldistanz beträgt also 3.

1b). Alle Codewörter haben voneinander Abstand 2 oder mehr. Das zweite und das dritte Codewort (10011 und 11001) haben voneinander Abstand 2.
Die Minimaldistanz beträgt also 2.

2. Ein Beispiel (es gibt mehrere Möglichkeiten):
Der Sender schickt das Codewort 0111. Es passiert ein Fehler und der Empfänger erhält 0110.
Diese Sequenz hat nun von 0111 und 0100 den gleichen Abstand. Der Empfänger kann also wählen, zu welchem Codewort er decodiert. Wenn wir Pech haben, decodiert er zu 0100.
Somit kann der Code keinen Fehler korrigieren
Anschaulich sieht das so aus

$W_{ges} = 0111 \xrightarrow[\text{es passiert ein Fehler}]{\text{senden;}} W_{empf} = 0110 \xrightarrow[\text{codiert}]{\text{Empfänger}} W_{dec} = 0100$

4.2.2 CHECKPOINT II

1. (K2) Alle Codewörter haben voneinander Abstand mindestens 3. Das vierte Codewort (101010) hat von allen andern Abstand genau 3. Die Minimaldistanz beträgt also 3 und damit kann der Code einen Fehler korrigieren.

2. (K3) Ein Beispiel: $C = \{00011, 11000, 10110\}$.

4.2.3 Sonstige Aufgaben

Lösung zur Aufgabe 'Sind die folgenden Codes linear?'

a) Der Code C_1 ist nicht linear (beispielsweise ist 0001+1101 = 1100 kein Codewort).
b) Der Code C_2 ist linear (alle Summen von Codewörtern sind selbst Codewörter).

4.2.3 Zeig was Du kannst

1. (K2)

   a) Dieser Code kann keinen Fehler korrigieren, denn das erste und das letzte Codewort (0101 und 1001) haben Abstand 2. So kann schon ein Fehler zu einer falschen Decodierung führen.

   b) Dieser Code kann einen Fehler korrigieren, denn der Minimalabstand beträgt 3 (erreicht wird er z.B. zwischen dem ersten und dem dritten Wort)

   c) Dieser Code ist linear und hat Minimalgewicht 3. Deshalb beträgt der Minimalabstand 3 und somit kann der Code einen Fehler korrigieren.

   d) Dieser Code kann einen Fehler korrigieren, denn der Minimalabstand beträgt 3. Erreicht wird der dieser Minimalabstand beispielsweise zwischen dem ersten und dem zweiten Wort.

2. (K2)

   a) Ja, der Code ist linear (die Summe von zwei Codewörtern ist wieder ein Codewort).

   b) Nein, der Code ist nicht linear (die Summe des zweiten und des fünften Wortes ist beispielsweise kein Codewort).

   c) Nein, der Code ist nicht linear, denn das Nullwort ist nicht dabei. Wenn man ein beliebiges Codewort mit sich selbst addiert, dann kommt das Nullwort heraus. Deshalb ist das Nullwort in jedem linearen Code dabei.

   d) Ja, der Code ist linear

3. (K2) Um zu beweisen, dass *C* ein linearer Code ist, brauchen wir bloss zu zeigen, dass die Summe zweier Codewörter wieder ein Codewort ist. Dazu betrachten wir alle möglichen Paare:
   - $a + b$ ist ein Codewort
   - $a + (a+b) = (a + a) + b = b$ (beachte: für jedes Codewort c gilt: $c + c = 0...0$)
   - $b + (a+b) = a + (b + b) = a$

4. (K3) Der Code hat Minimalabstand 3, kann also einen Fehler korrigieren. Aendern wir also ein Bit von 0001, dann sind wir noch immer in seiner Intimsphäre . Aendern wir drei oder mehr Bits, dann enthält das resultierende Wort mindestens zwei '1' und hat damit von 1111 Abstand höchstens 2. Damit liegt es näher bei 1111 als bei 0001 (Abstand mindestens 3, da mindestens drei Stellen geändert) und gehört somit **nicht** zur Intimsphäre von 0001.

Bleibt noch der Fall, in dem genau zwei Bits geändert werden. Liegen **beide** geänderten Bits innerhalb der ersten drei Bits, dann enthält das resultierende Wort mindestens drei '1' und gehört so zur Intimsphäre von 1111.
Betrachten wir nun die Situation, das vierte Bit und sonst noch ein Bit verändert werden. Falls eine der ersten beiden Stellen geändert wird, dann lautet das resultierende Wort 1000 oder 0100 und liegt so in der Intimsphäre von 1100.
Falls hingegen das vierte Bit und keine der ersten beiden Stellen geändert werden (sprich: Bit 3 und 4 werden geändert), dann hat lautet resultierende Wort 0011. Dieses hat von 0001 den kleineren Abstand als von den andern beiden Codewörtern und gehört somit in die Intimsphäre von 0001.
Kurz gesagt: *Int*(0001) = {000, 001, 010, 100}

5. (K3) Wir betrachten einen Code und bezeichnen seinen Minimalabstand mit *m.* Wir betrachten wiederum W_{ges} (= das vom Sender verschickte Codewort) und W_{empf} (= das vom Empfänger erhaltene Codewort). Es gilt
Der Empfänger bemerkt Fehler $\Leftrightarrow W_{empf}$ ist kein Codewort
Falls weniger als *m* Fehler passiert sind, kann W_{empf} kein Codewort sein (sonst wären W_{ges} und W_{empf} zwei Codewörter mit Abstand kleiner als *m*).
Falls jedoch *m* Fehler passiert sind, kann W_{empf} durchaus ein Codewort sein (schliesslich gibt es Codewortpaare mit Abstand *m*).
Ein Code kann also *m* –1 Fehler detektieren.

6.(K3) Ein Beispiel: *C* = {000000, 000111, 111000, 111111}.

7.(K3) Wir müssen lediglich zeigen, dass die Summe von zwei Bitsequenzen mit geradem Gewicht selbst wieder gerades Gewicht hat. Wir nehmen zwei (*n*-Bit)-Sequenzen mit geradem Gewicht und nennen sie v_1 und v_2. Mit w_1 und w_2 bezeichnen wir ihre Gewichte. Den Abstand zwischen v_1 und v_2 (sprich die Anzahl Stellen, an denen sie sich unterscheiden) bezeichnen wir mit *r.*
Es gilt:

- Es gibt $w_1 - r$ Stellen, an denen v_1 eine '1' hat, und v_2 eine '0'.
- Es gibt $w_2 - r$ Stellen, an denen v_2 eine '1' hat und v_1 eine '1'.

v_1 und v_2 unterscheiden sich also an $(w_1 - r) + (w_2 - r) = w_1 + w_2 - 2r$ Stellen.
Zur Erinnerung: das Gewicht der Summe von zwei Codewörtern entspricht genau deren Abstand.
$v_1 + v_2$ hat also Gewicht $w_1 + w_2 - 2r$ und das ist gerade!

4.3 Lösungen Additum

4.3.1 Sonstige Aufgaben:

Lösung zur Aufgabe 'Betrachte den Code $C = \{001, 110, 111\}$ und bestimme für jedes Codewort seine Intimsphäre':

- $Int(001) = \{001, 000\}$
- $Int(110) = \{110, 010, 100\}$
- $Int(111) = \{111\}$

Lösung zu Knobelaufgabe 2b):
Wir müssen zeigen, dass es keinen Code mit 3 Codewörtern der Länge 4 gibt, der einen Fehler korrigiert.
Wir werden versuchen, solch einen zu konstruieren und dann feststellen, dass es nicht gehen kann.
Zunächst wählen wir ein beliebiges Codewort, z.B. 0000 (wir verbauen uns dadurch keine mögliche Lösung).
Das zweite Codewort müssen wir so wählen, dass es sich vom ersten (sprich 0000) an mindestens 3 Stellen unterscheidet. (Schliesslich wollen wir ja einen Fehler korrigieren. Und da muss der Minimalabstand mindestens 3 betragen).
Wir unterscheiden also zwei Möglichkeiten:

1. Das zweite Wort unterscheidet sich vom ersten an 4 Stellen. Dann ist es 1111. Das dritte Wort müsste sich dann von beiden (sprich von 0000 und 1111) an drei Stellen unterscheiden. Solch ein Wort lässt sich jedoch nicht finden.

2. Das zweite Wort unterscheidet sich vom ersten an 3 Stellen. Dann besitzt es drei '1'. Das dritte Wort sollte sich nun von beiden an 3 (oder mehr) Stellen unterscheiden. Damit es sich von 0000 an drei Stellen unterscheidet, müsste es mindestens 3 '1' haben. Damit es sich von dem zweiten Wort um drei Stellen unterscheidet, müsste es mindestens zwei '0' haben. Doch ein Wort der Länge 4 kann nicht drei '1' und zwei '0' haben. Solch ein Wort lässt sich also nicht finden.

4.3.2 Zeig was Du kannst

1. (K2) Wir setzen in die Formel ein und erhalten $4 \cdot (1 + \binom{6}{1} + \binom{6}{2}) = 88$ und $2^6 = 64$

   Es gibt also keinen solchen Code.

2. (K2) Wir setzen in die Formel ein und erhalten $5 \cdot (1 + \binom{7}{1} + \binom{7}{2}) = 145$ und $2^7 = 128$

Es gibt also keinen solchen Code.

3. (K2) Wir setzen in die Formel ein und erhalten
$6 \cdot (1 + \binom{10}{1} + \binom{10}{2} + \binom{10}{3}) = 1056$ und $2^{10} = 1024$. Es gibt also keinen solchen Code.

4. (K2) Wir setzen in die Formel ein und erhalten
$7 \cdot (1 + \binom{12}{1} + \binom{12}{2} + \binom{12}{3} + \binom{12}{4}) = 5558$ und $2^{12} = 4096$. Es gibt also keinen solchen Code.

5. (K5) Zuerst suchen wir – mithilfe von pröbeln – die kleinstmögliche Codelänge, welche die (in den letzten Teilaufgaben rege angewandte) Ungleichung erfüllt.
Danach prüfen wir, ob es wirklich einen Code mit dieser Länge und den gewünschten Eigenschaften gibt. Falls nicht, erhöhen wir die Länge um 1 und prüfen weiter. Solange bis wir einen Code gefunden haben.
Wir erhalten zunächst: $3 \cdot (1 + \binom{6}{1} + \binom{6}{2}) = 66$ und $2^6 = 64$. Der Code hat mind. Länge 7.
Ausserdem: $3 \cdot (1 + \binom{7}{1} + \binom{7}{2}) = 87$ und $2^7 = 128$. Die Ungleichung ist für $n = 7$ ist erfüllt.

7 ist also die kleinstmögliche Codelänge, welche die Ungleichung erfüllt.
Nun prüfen wir, ob es einen Code der Länge 7 mit den gewünschten Eigenschaften gibt. Wir nehmen ein beliebiges Codewort, sagen wir 00...00 (wir schränken unsere Lösung damit nicht ein). Das zweite Codewort nennen wir w. Wir müssen es so wählen, dass es sich vom Nullwort an mindestens 5 Stellen unterscheidet – sprich: w beinhaltet mindestens fünf '1'. Richten wir nun das Augenmerk auf das dritte Codewort: Es muss sich vom Nullwort an mindestens 5 Stellen unterscheiden und somit mindestens fünf '1' besitzen. Auf der andern Seite muss es sich von w an mindestens 5 Stellen unterscheiden, d.h. es muss mindestens drei '0' beinhalten. Doch das ist zu viel verlangt – ein 7-Bit-Wort kann nicht gleichzeitig fünf '1' und drei '0' beinhalten. Es gibt also keinen solchen Code der Länge 7.

Nun erhöhen wir die Länge um 1. Jetzt lässt sich der gewünschte Code leicht finden:
C = {000'000'00, 000'111'11, 111'001'10}.
8 ist also die kleinstmögliche, gesuchte Codelänge

6. (K5) Wir wissen (aus dem letzten Kapitel):
Ein Code kann k Fehler korrigieren $\Leftrightarrow$ die Minimaldistanz beträgt $2k+1$ (oder mehr).
Beinhaltet ein Code der Länge n genau 2 Codewörter, kann man diese so wählen, dass sie voneinander Abstand n haben. (Zum Beispiel C = {00...0, 11..1}).
Dann ist die Minimaldistanz n und es gilt:
Ein Code (der Länge n) kann k Fehler korrigieren $\Leftrightarrow$ $n \geq 2k+1$
Oder anders formuliert:
Es gibt einen Code mit 2 Codewörtern der Länge n, der k Fehler korrigiert $\Leftrightarrow$ $n \geq 2k+1$.

7. (K5)

a) Wir gehen analog vor wie beim Beweis im letzten Unterkapitel. Betrachten wir einen Code C . Seine Länge bezeichnen wir mit n, die Anzahl Fehler, die er korrigieren kann, mit k. Zunächst kümmern wir uns um die Intimsphären. Nehmen wir dazu ein beliebiges Codewort w. Seine Intimsphäre enthält **mindestens**

- w selbst
- Alle 3-Folgen, die sich von w an genau einer Stelle unterscheiden (= Abstand 1 haben)
- Alle 3-Folgen, die sich von w an genau zwei Stellen unterscheiden (= Abstand 2 haben)

  .
  .
  .
- Alle 3-Folgen, die sich von w an genau k Stellen unterscheiden

Im letzten Unterkapitel haben wir **Bitfolgen** betrachtet und dabei gezeigt: Die Anzahl Bitfolgen, die sich von einem Wort w an genau l Stellen unterscheiden, entspricht der Anzahl Möglichkeiten, l aus n Elementen zu wählen. Dies wiederum ist gleich $\binom{n}{l}$.

Dem ist hier jedoch nicht so!
(Für 3-Folgen geht es jedoch nicht ganz so einfach – wir brauchen noch ein paar Zusatzüberlegungen)
Betrachten wir zuerst mal den Fall $l = 1$. n und die andern Variabeln lassen wir vorerst aus dem Spiel und schauen einen konkreten Fall an.
Bsp: Wie viele 3-Folgen unterscheiden sich von 102 an genau einer Stelle?
Es sind genau die folgenden Sequenzen: 002, 202, 112, 122, 100, 101.
Damit gibt es genau $2 \cdot 3 = 6$ solche 3-Folgen.

Wie sieht es nun für n aus?
Es gibt n Möglichkeiten, die "unterschiedliche Stelle" zu wählen. Für diese "unterschiedliche Stelle" stehen nun 2 verschiedene Zahlen zur Auswahl. (Im Bsp: Wenn die erste Position als "unterschiedliche Stelle" gewählt wird, dann stehen '0' und '2' zur Auswahl.) Insgesamt gibt es also genau $2 \cdot n$ 3-Folgen, die sich von w an genau einer Stelle unterscheiden.

Betrachten wir nun noch kurz den Fall '$l = 2$'. Es gibt nun $\binom{n}{2}$ Möglichkeiten, die "unterschiedliche Stelle" zu wählen. Für die zwei "unterschiedlichen Stellen" stehen nun je zwei Zahlen zur Verfügung – es gibt also $2^2 = 4$ mögliche Kombinationen, diese Stellen auszufüllen. Also gibt es insgesamt $\binom{n}{2} \cdot 4$ 3-Folgen, die sich von w an genau zwei Stellen unterscheiden.

Und nun, ganz allgemein für l: Es gibt $\binom{n}{l}$ Möglichkeiten, die "unterschiedlichen Stellen" zu wählen. Für die "unterschiedlichen Stellen" stehen nun je 2 Zahlen zur

Verfügung. Es gibt also 2^l mögliche Kombinationen, diese Stelle auszufüllen. Insgesamt gibt es nun $\binom{n}{l} \cdot 2^l$ 3-Folgen, die sich an genau *l* Stellen von *w* unterscheiden.

Daraus können wir schliessen

$$|Int(w)| \geq 1 + \binom{n}{1} \cdot 2 + \binom{n}{2} \cdot 2^2 + \binom{n}{3} \cdot 2^3 + \ldots + 2^k \cdot \binom{n}{k}$$

Es gibt nun 3^n 3-Folgen der Länge *n*. Da alle Intimsphären disjunkt sind, kommt jede 3-Folge in höchstens einer Intimsphäre vor. Somit gilt $\sum_{w \in c} |Int(w)| \leq 3^n$.

Bezeichnen wir die Anzahl Codewörter mit *r* und setzen die Formel aus dem Kästchen ein, erhalten wir

$$r \cdot (1 + 2 \cdot \binom{n}{1} + 2^2 \cdot \binom{n}{2} + 2^3 \cdot \binom{n}{3} + \ldots + 2^k \cdot \binom{n}{k}) \leq 3^n .$$

b). Auf die gleiche Art und Weise wie in a) erhalten wir

$$r \cdot (1 + 3 \cdot \binom{n}{1} + 3^2 \cdot \binom{n}{2} + 3^3 \cdot \binom{n}{3} + \ldots + 3^k \cdot \binom{n}{k}) \leq 4^n .$$