Informatietheorie

Hans Melissen

j.b.m.melissen@its.tudelft.nl

INHOUDSOPGAVE

0 Wat is informatietheorie?

Dit dictaat behandelt de grondbeginselen van het vak Informatietheorie. Het vak is rond 1947 ontstaan toen de Amerikaan Claude Shannon zijn artikel *Mathematical Theory of Communication* publiceerde. Een belangrijk resultaat uit dit artikel was dat je je opslagcapaciteit optimaal kunt benutten en de hoogst mogelijke transmissiesnelheid kunt halen als je compressie en krachtige foutcorrectie toepast.

Informatietheorie houdt zich bezig met het volgende soort vragen:

- Wat is informatie eigenlijk precies? Hoe kun je een hoeveelheid informatie getalsmatig beschrijven?
- Hoe kun je informatie op een zo compact mogelijke manier opslaan en verzenden?
- Wat is de maximale hoeveelheid informatie die over een bepaald kanaal kan worden verzonden?
- Hoe kun je informatie veilig verzenden?
- Hoe constateer je fouten in verzonden en opgeslagen informatie? Is het mogelijk om zulke fouten na afloop nog te corrigeren?
- Hoe verstuur je informatie over een onbetrouwbaar kanaal?

Het gaat er dus eigenlijk om informatie te kwantificeren (in getallen te vangen) en om de transmissie van data te optimaliseren en betrouwbaar te maken. We zullen ons daarbij beperken tot discrete informatiebronnen.

Hieronder zie je een eenvoudig model dat overdracht van data beschrijft. Je hebt daarin te maken met een bron van informatie: een zender, vervolgens een kanaal waarover de informatie wordt verstuurd en tenslotte een plek waar de informatie naar toe gaat: de ontvanger. Het kan hierbij gaan om informatie die van de ene plak naar de andere wordt verzonden (datatransmissie), maar ook om data die wordt opgeslagen om later weer te worden geraadpleegd.

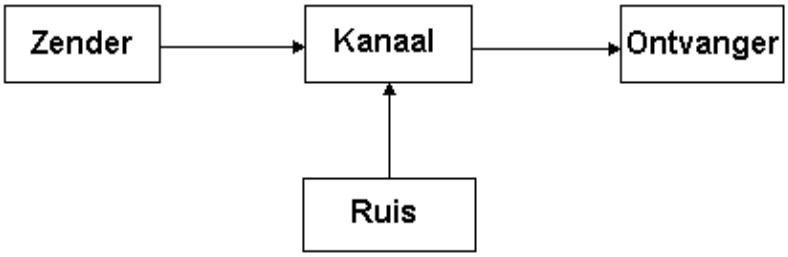


Fig. 1: Eenvoudig informatiemodel

Een kanaal kan bijvoorbeeld zijn: een telefoonlijn (elektrische signalen door een koperdraad of licht door een glasvezelkabel), de “ether” (een radiosignaal, elektromagnetische straling, maar ook akoestische transmissie (geluid)), een laser in een CD drive die een signaal leest van een CD-ROM of DVD, een magnetisch opslagmedium, zoals een harddisk. Een informatiekanaal kan onderhevig zijn aan ruis. Dat betekent dat je de ontvangen informatie niet helemaal kunt vertrouwen. Ruis kan bijvoorbeeld worden veroorzaakt door kosmische straling, bliksem, krassen en vuil op een CD, overspraak (beïnvloeding door een ander kanaal), thermische ruis uit een elektronische component. Ruis maakt dat een signaal anders kan worden ontvangen dan het wordt verstuurd. Een deel van een bericht wordt dan bijvoorbeeld niet ontvangen of onjuist geïnterpreteerd. Denk hierbij bijvoorbeeld ook aan een harde schijf met *bad sectors*.

In een ingewikkelder informatiemodel wordt de data door de zender bewerkt om de datatransmissie veiliger, efficiënter en/of betrouwbaarder te laten verlopen. Zo kunnen door de zender de volgende stappen worden afgewerkt:

1. **Datareductie** In deze stap wordt alle overbodige informatie uit het bericht gehaald. Uit een source-file voor een compiler kunnen bijvoorbeeld veel overbodige spaties worden verwijderd.
2. **Broncodering** (datacompressie) De data die overblijft wordt vervolgens zo compact mogelijk gecodeerd om zo weinig mogelijk geheugen of bandbreedte van een transmissiekanaal te gebruiken. Voorbeelden zijn ZIP voor data en MPEG voor beeld en geluid (MP3).
3. **Vercijfering** (encryptie) Om te voorkomen dat de data te eenvoudig te lezen is voor buitenstaanders kan het bericht vervolgens worden vercijferd (versleuteld, ge-encrypt) met behulp van een geheimschrift. Veelgebruikte algoritmes zijn DES en RSA. Dit soort algoritmes is erop gebaseerd dat het “kraken” van zo’n algoritme te veel rekentijd zou kosten.
4. **Kanaalcodering** (foutdetectie/-correctie) In de laatste slag wordt het bericht zodanig gecodeerd dat eventuele fouten die tijdens het verzenden optreden kunnen worden gedetecteerd, of gecorrigeerd. Dit gebeurt door middel van foutdetecterende of foutcorrigerende codes, die bepaalde extraatjes aan de data toevoegen om daarmee eventueel opgetreden fouten te kunnen constateren of zelfs te repareren. Denk hierbij aan een pariteitsbits, of aan de Reed-Solomoncodering die voor CD’s wordt gebruikt.

De volgorde in deze stappen is essentieel. Kanaalcodering moet als laatste plaatsvinden wil de foutcorrectie en –detectie zijn werk kunnen doen. Broncodering moet als eerste plaatsvinden omdat vercijfering de structuur van de data kan verstoren, waardoor efficiënte compressie vaak niet meer mogelijk is (Random data is namelijk niet te comprimeren).

Vervolgens wordt de informatie via een discreet kanaal verzonden. Dit kan bijvoorbeeld wel betekenen dat het signaal gemoduleerd wordt en als een analoog signaal door de ether gaat en vervolgens weer na AD-conversie (Analoog $\rightarrow$ Digitaal) door de ontvanger kan worden bewerkt. Tijdens het verzenden kan door ruis weer vervorming van het signaal optreden. De ontvanger moet vervolgens alle processtappen van de zender in omgekeerde volgorde doorlopen: kanaaldecodering, ontcijferen, brondecodering en datareconstructie.

In dit dictaat zullen we aandacht besteden aan informatie-inhoud (entropie), datacompressie en foutdetecterende en –corrigerende codes. Versleuteling komt aan bod in het dictaat Cryptografie.

1 Wat is informatie?

1.1 Entropie

Voor de Sahara wordt door een weerstation dagelijks een regenvoorspelling afgegeven. De informatiebron geeft twee mogelijke soorten informatie: “regen” of “droog”. Nu is de kans op regen in de Sahara minimaal, dus het bericht “droog” heeft een zeer grote kans. Hierdoor bevat het bericht meteen ook weinig informatie, want de inhoud is behoorlijk voorspelbaar. Als je gokt dat de voorspelling voor de volgende dag “droog” zal luiden, dan heb je vrijwel altijd gelijk. Het bericht “regen” heeft daarentegen een zeer kleine kans, en daarmee een grote informatie-inhoud.

Dit voorbeeldje maakt duidelijk dat informatie iets met kansen te maken heeft. Informatie geeft je antwoord op een vraag, waarvan je het antwoord nog niet (met zekerheid) kende. Informatie helpt je van bepaalde onzekerheden af. Zo’n vraag kan bijvoorbeeld ook zijn: “Wat zou het volgende bit van dit bericht zijn?”, die je stelt terwijl je een databestand aan het bekijken bent. Om precies te zeggen wat informatie is hebben we het begrip kansruimte nodig, want in de informatietheorie wordt een informatiebron gezien als een opeenvolging van gebeurtenissen die een bepaalde kansverdeling hebben. De informatie-inhoud is niet afhankelijk van de precieze inhoud van het bericht (de semantiek van de informatie) maar alleen van de kans daarop. Als op een bepaalde dag de kans op regen in de Sahara 1/50000 is en de kans op een beurscrash ook 1/50000, dan zit er evenveel informatie in de mededeling “Het gaat morgen regenen”, als in de bewering “Morgen crasht de beurs”. Een bijna zekere gebeurtenis bevat weinig informatie en uitsluitsel over een zeer onzekere uitkomst bevat veel informatie.

Een **informatiebron** zullen we opvatten als een generator van symbolen (letters) die kunnen worden gekozen uit een eindig alfabet. Zo’n alfabet is bijvoorbeeld: {“regen”, “droog”}, of {0, 1} voor een binair bericht, of de ASCII- of *unicode*tabel voor een tekst. Informatie bestaat dus uit een opeenvolging van symbolen die niet voorspelbaar zijn (anders zou de informatie-inhoud namelijk nul zijn). Elk volgend symbool in de tekst heeft een bepaalde kans. Vaak zijn die kansen gerelateerd. Als in een Nederlandse tekst een q voor komt, is de kans zeer groot dat het volgende symbool een u is, veel groter dan de kans op een u in het algemeen. Toch zullen we voorlopig even aannemen dat de kansen van de symbolen onafhankelijk van elkaar zijn. Zo’n informatiebron heet een **geheugenloze informatiebron** (*zero-memory information source*). Deze bron heeft geen geheugen, hij weet bij het verzenden van een symbool het vorige niet meer, zodat er geen correlatie is tussen opeenvolgende symbolen.

Een **kansruimte** is een verzameling van eindig veel mogelijke gebeurtenissen $X = \{x_1, x_2, \ldots, x_n\}$ met bijbehorende kansen $p_1, p_2, \ldots, p_n$ (ook wel genoteerd als $p(x_1), p(x_2), \ldots, p(x_n)$). Dit zijn getallen tussen 0 en 1: $0 \le p_j \le 1$. Er treedt altijd één van de mogelijke gebeurtenissen op, dus de som van alle kansen is gelijk aan 1: $p_1+p_2+ \ldots + p_n = 1$.
Denk bijvoorbeeld aan het gooien met twee dobbelstenen. De kans dat je 2 gooit is 1/36 (dit kan namelijk op één manier: 1 + 1), de kans op 3 (= 1 + 2 = 2 + 1) is 2/36, … de kans op 7 (= 1 + 6 = 2 + 5 = 3 + 4 = 4 + 3 = 5 + 2 = 6 + 1) is 6/36, etc. De som van alle kansen is altijd gelijk aan 1:

$$\frac{1}{36}+\frac{2}{36}+\frac{3}{36}+\frac{4}{36}+\frac{5}{36}+\frac{6}{36}+\frac{5}{36}+\frac{4}{36}+\frac{3}{26}+\frac{2}{36}+\frac{1}{36}=\frac{36}{36}=1.$$

Je kunt ook denken aan informatie in een tekstbestand. Elk symbool is gekozen uit een alfabet, een collectie van symbolen, bijvoorbeeld: {0,1} voor een binair gecodeerd bestand, of de

ASCII karakters. Je weet nooit wat het volgende symbool zal zijn, want elk symbool heeft een bepaalde kans om voor te komen.

Als we kwantitatief willen praten over informatie zullen we er een getal aan moeten toekennen. Dit getal hangt alleen af van de kans, de onzekerheid van een gebeurtenis (antwoord op een vraag) en niet van de gebeurtenis zelf. Het getal is groot voor gebeurtenissen met een kleine kans en klein als de kans in de buurt van de 1 komt (een bijna zekere gebeurtenis). We noteren die informatie-inhoud van een kans p even met I(p). Voor deze I(p) moet er een speciale eigenschap gelden. Als je namelijk twee onafhankelijke gebeurtenissen hebt die respectievelijk kans p_1 en p_2 hebben, dan is de kans dat ze allebei plaatsvinden gelijk aan $p_1 \times p_2$ (Let wel: alleen als de gebeurtenissen onafhankelijk zijn!). Dit betekent dat de informatie van deze gezamenlijke gebeurtenis gelijk is aan $I(p_1p_2)$. Echter, omdat de gebeurtenissen onafhankelijk zijn (ze beïnvloeden elkaar niet) krijg je uit het tegelijkertijd plaatsvinden van de twee gebeurtenissen evenveel informatie als wanneer de gebeurtenissen apart hadden plaatsgevonden. De totale hoeveelheid informatie is dus de som van de hoeveelheid informatie in elk van de twee gebeurtenissen: $I(p_1) + I(p_2)$. Dat betekent dat er moet gelden: $I(p_1p_2) = I(p_1) + I(p_2)$. Een functie die hieraan voldoet is de logaritme: $\log (p_1p_2) = \log p_1 + \log p_2$.

In 1927 voerde Hartley een maat in voor de hoeveelheid informatie in een gebeurtenis met uitkomst x_j en kans p_j, namelijk:

$$I(x_j) = -\,^2\log p_j.$$

Dit heet ook wel de **zelfinformatie**. Als een gebeurtenis kans 1 heeft, is de informatie-inhoud gelijk aan 0, want: $^2\log 1 = 0$. Zo'n gebeurtenis vindt zeker plaats en bevat dus geen informatie, je leert niets nieuws van de uitkomst, want die wist je al. Een gebeurtenis met een kleine kans heeft een grote informatie-inhoud: als de kans op een gebeurtenis bijvoorbeeld 1/16 is, dan is de informatie-inhoud: $-^2\log(1/16) = 4$. Voor heel kleine kansen wordt de informatie-inhoud zeer groot.
De reden voor de logaritme in de formule voor informatiedichtheid zagen we eerder al. Als twee gebeurtenissen onafhankelijk zijn, dan is de kans op de gezamenlijke gebeurtenis gelijk aan p_1p_2. Bijvoorbeeld: de kans om een 6 te gooien met een dobbelsteen (kans 1/6), en vervolgens een 3 (kans 1/6) is (1/6) x (1/6) = 1/36. De informatie-inhoud van deze twee gebeurtenissen samen is:

$$-\,^2\log (p_1p_2) = -\,^2\log p_1 \; -\,^2\log p_2,$$

en dat is dus gelijk aan de som van de twee informatie-inhouden afzonderlijk. Dit klopt met je gevoel over hoe dit met informatie moet gaan: de totale hoeveelheid informatie in twee berichten die niets met elkaar te maken hebben is gewoon de som van de twee afzonderlijke informatie-inhouden.
De reden voor het grondtal 2 in de $^2\log$ is dat de informatie-inhoud wordt uitgedrukt in bits per gebeurtenis. Als je 2^N symbolen hebt met dezelfde kans, dan heeft elk symbool N bits aan informatie. Die N bits heb je namelijk nodig om elk van de symbolen te kunnen opslaan. Een andere manier om dit te zeggen is dat je N ja-nee vragen moet stellen om achter het juiste symbool te komen. De kans op een symbool is $1/2^N$ en $^2\log (1/2^N)$ is dan precies gelijk aan N.

Nu we weten wat de informatie-inhoud van een gebeurtenis is kunnen we het ook hebben over de gemiddelde hoeveelheid informatie *in de totale kansruimte*, ook wel **entropie** genaamd. Die krijg je door de hoeveelheid informatie in gebeurtenis x_j te vermenigvuldigen met zijn kans, en dit op te tellen voor alle mogelijkheden. Een gewogen gemiddelde van de hoeveelheden informatie per gebeurtenis, gewogen met de kans daarop. Dit is in het algemeen de manier om een gemiddelde in een kansruimte te bepalen:

$$H(X) = \sum_{j=1}^{n} p_j I(x_j) = -\sum_{j=1}^{n} p_j \,{}^2\log p_j = -p_1\,{}^2\log p_1 - p_2\,{}^2\log p_2 - \ldots - p_n\,{}^2\log p_n.$$

Dit getal moet je interpreteren als het aantal bits dat gemiddeld aan informatie in deze kansruimte aanwezig is voor één gebeurtenis. Het is een maat voor de gemiddelde onzekerheid in de informatiebron, de onvoorspelbaarheid van de symbolen die de bron genereert. Het begrip entropie komt oorspronkelijk uit de thermodynamica. Daar is het een maat voor de wanorde in een systeem. Geordende systemen (bijvoorbeeld kristalstructuren) hebben een lage entropie, wanordelijke (bijvoorbeeld water) een hoge (en een hogere temperatuur dan de geordende versie).

Voorbeeld: Bij het gooien met één dobbelsteen hebben we bijvoorbeeld zes gebeurtenissen, allemaal met kans 1/6, dus

$$H = 6 \cdot \left(-\frac{1}{6}\,{}^2\log\frac{1}{6} \right) = {}^2\log 6 = \frac{\log 6}{\log 2} = 2{,}58496\ldots \text{ bits.}$$

Dit is “precies” het aantal bits dat je (gemiddeld) nodig hebt om een getallen van 1 t/m 6 te kunnen representeren.

Voorbeeld: Bij het gooien met twee dobbelstenen zijn er zesendertig gebeurtenissen, allemaal met kans 1/36 (eerst een 3 en dan een 4 is iets anders dan eerst een 4 en dan een 3, dus

$$H = 36 \cdot \left(-\frac{1}{36}\,{}^2\log\frac{1}{36} \right) = {}^2\log 36 = \frac{\log 36}{\log 2} = 5{,}16992\ldots \text{ bits} = 2 \text{ x } 2{,}58496\ldots \text{ bits.}$$

Dit is precies tweemaal de informatie van één worp. De twee worpen hebben niets met elkaar te maken, dus dat is logisch.

Voorbeeld: Als je bij het gooien met twee dobbelstenen alleen naar het totale aantal gegooide punten kijkt zijn er elf gebeurtenissen met de volgende kansen:

punten	**manieren**	**kans**
2	1+1	1/36
3	1+2=2+1	2/36
4	1+3=2+2=3+1	3/36
5	1+4=2+3=3+2=4+1	4/36
6	1+5=2+4=3+3=4+2+5+1	5/36
7	1+6=2+5=3+4=4+3=5+2=6+1	6/36
8	2+6=3+5=4+4=5+3=6+2	5/36
9	3+6=4+5=5+4=6+3	4/36
10	4+6=5+5=6+4	3/36
11	5+6=6+5	2/36
12	6+6	1/36

De entropie is nu:

$$H = -\frac{1}{36}\,{}^2\log\frac{1}{36} - \frac{2}{36}\,{}^2\log\frac{2}{36} - \frac{3}{36}\,{}^2\log\frac{3}{36} - \ldots - \frac{1}{36}\,{}^2\log\frac{1}{36} = 3{,}27440\ldots \text{bits.}$$

Je ziet dat de som van de ogen minder informatie bevat dan wanneer je de uitslag van elke dobbelsteen weet, zoals in het vorige voorbeeld.

Voorbeeld: Bij het werpen met twee dobbelstenen kun je ook alleen op de uitslagen letten, zonder de volgorde van de dobbelstenen erbij te betrekken, zoals je dat eigenlijk altijd doet als je met twee dobbelstenen gooit. Twee gelijken (zes mogelijkheden) hebben dan een kans 1/36 en twee niet-gelijke ogen (15 manieren) kun je op twee manieren gooien, die hebben dus kans 2/36. De entropie wordt dan

$$H = -6\times\frac{1}{36}\,{}^{2}\log\frac{1}{36} - 15\times\frac{2}{36}\,{}^{2}\log\frac{2}{36} = 4{,}33659\ldots \text{bits}.$$

Dit geeft minder informatie dan wanneer je van de eerste en de tweede dobbelsteen apart weet wat ermee is gegooid, maar je hebt meer informatie dan wanneer je alleen de som weet.

Voorbeeld: Een TV scherm is opgebouwd uit 576 lijnen met 720 pixels per lijn. Elke pixel kan 10 verschillende grijswaarden aannemen. Als we er vanuit gaan dat elke mogelijkheid even waarschijnlijk is, dan krijgen we $n = 10^{414720}$ verschillende beelden. De kans op één zo'n beeld is $p = 1/n$ en de entropie is

$$H = -\sum_{j=1}^{n} p\,{}^{2}\log p = -np\,{}^{2}\log p = -\,{}^{2}\log p = {}^{2}\log n = {}^{2}\log 10^{414720} = 414720\,{}^{2}\log 10 =$$

$$= 414720\,\frac{\log 10}{\log 2} \approx 1.3776\ldots\cdot 10^{6} \text{ bits} = \frac{1.2776\cdot 10^{6}}{8\cdot 1024}\ \text{kByte} = 168.17\ \text{kB}.$$

Er zit dus zo'n 168 kilobyte aan informatie in één zo'n TV-beeld. Dit is ook precies het aantal bytes dat je nodig hebt om één zo'n beeld binair te coderen.

Voorbeeld: In een 4x4 vierkant is één hokje gearceerd. Hoeveel informatie is er bevat in dit vierkant? Hoe kun je uitvinden welk vierkant gearceerd is? Door te vragen: "Is het 1?", "Is het 2?", etc. Dit is niet zo'n intelligente manier, want gemiddeld heb je 8,5 vragen nodig, en in het ergste geval moet je 16 vragen stellen. Dat betekent dat je 16 bits nodig hebt om de antwoorden te kunnen representeren.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Je kunt ook vragen: "Zit het vierkant in de bovenste helft?", "Zit het in de linkerhelft?" Met deze twee vragen weet je in welk 2x2 vierkant het kleine vierkantje zit. Door de twee vragen nog een keer te stellen kun je vervolgens het vierkantje precies lokaliseren. Hiervoor heb je dus 4 bits nodig. Een andere manier om dit te zien is als volgt: Je kunt de positie van het vierkantje in het vierkant aanduiden met een getal van 1 t/m 16. Binair kun je dit in 4 bits representeren.
De entropie is in dit voorbeeld:

$$H = -\sum_{j=1}^{16}\frac{1}{16}\,{}^{2}\log\frac{1}{16} = {}^{2}\log 16 = 4 \text{ bits}.$$

Er zit dus ook precies aan informatie in wat we hierboven hebben gevonden.

Een **binair symmetrische informatiebron** (*Binary Symmetric Source*) is een bron die twee boodschappen (0/1) kan afgeven met kansen respectievelijk: p en 1-p (Bijvoorbeeld onze regenvoorspelling in de Sahara). De entropie van deze informatiebron is:

$$H = -p\,{}^{2}\log p - (1-p)\,{}^{2}\log(1-p).$$

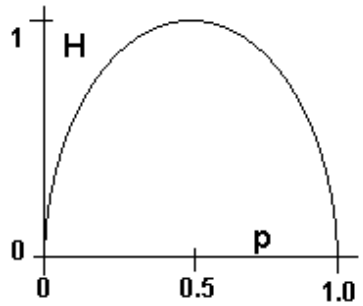


Fig. 2: Entropie van een binair symmetrische bron als functie van de kans p.

In de figuur zien we dat de entropie maximaal is als p = 0,5, dat wil zeggen, als beide antwoorden even waarschijnlijk zijn. De maximale informatie-inhoud is 1 bit per bericht. Dat is kan moeilijk anders, omdat er maar één bit per bericht wordt verstuurd. Als de kans p klein is, dan is ook de entropie klein, bijvoorbeeld voor p = 0,05 is de entropie H = 0,28639… bit per bericht. De informatie-inhoud van een 0 is groot: $-^2\log 0{,}05 = 4{,}321\ldots$, maar de kans erop is klein: 0,05. De informatie-inhoud van een 1 is klein: $-^2\log 0{,}95 = 0.074\ldots$, maar de kans erop is groot: 0,95.
Als één van de twee mogelijkheden met zekerheid optreedt (p = 0 of p = 1), dan is de entropie gelijk aan 0. Er zit geen informatie in zo'n gebeurtenis, omdat je zeker weet wat het antwoord zal worden.

Als je p=0 in de formule van de entropie invult zie je dat er een probleempje optreedt. Je krijgt dan namelijk $-0 \times {}^2\log 0$ en dat is $0 \times \infty$, dus het is niet direct duidelijk wat daar uit komt. Er komt 0 uit, en dat kun je op twee manieren zien. De eerste is door $-p\ {}^2\log p$ op de rekenmachine uit te rekenen voor steeds kleinere waarde van p. Je ziet dat er dan een steeds kleinere waarde uit komt en je kunt je er zo van overtuigen dat er wel nul uit zal komen. De tweede manier is netter en levert echt een bewijs dat het zo is:

$$\lim_{\varepsilon \downarrow 0} -\varepsilon\ {}^2\log \varepsilon = (neem\ \varepsilon = 2^{-x})\ \lim_{x\to\infty} x2^{-x} = \lim_{x\to\infty} \frac{x}{2^x} = 0\,,$$

want 2^x gaat veel sneller naar oneindig (exponentieel) dan x (lineair).

We hebben gezien dat de entropie bij twee uitkomsten maximaal is als de twee uitkomsten gelijke kansen hebben. Dit resultaat geldt algemener. Als er N mogelijke uitkomsten zijn (letters in het alfabet), dan is de entropie maximaal als alle kansen gelijk zijn, als alle gebeurtenissen (letters) even waarschijnlijk zijn. Er geldt dus:

$$0 \le H(X) \le {}^2\log N.$$

De entropie is <u>maximaal</u> (namelijk $^2\log N$) als alle kansen gelijk zijn. In dit geval is er de grootst mogelijke onduidelijkheid over het volgende symbool. Het kan elk symbool zijn met gelijke kans, geen enkel symbool heeft een voorkeurspositie. De entropie is <u>minimaal</u> (namelijk 0) als één symbool kans 1 heeft. In dat geval weet je zeker welk symbool er als volgende komt en is de informatie-inhoud dus nul.

Voorbeeld: De entropie van een Nederlandse tekst.
Als je de 26 letters in een Nederlandse tekst binair wilt opslaan, dan heb je minstens $^2\log 26 = 4{,}700\ldots$ bits per letter nodig (als we even uitgaan van allemaal kleine letters). In de praktijk wordt dit vaak 5 bits/letter, omdat je ook de spatie en wat lettertekens als punt en komma wilt kunnen representeren. Volgens het entropiebeeld is deze representatie met 4,7 bits/letter

alleen optimaal als alle letters precies even vaak voorkomen. In een gewone tekst is dat zeker niet het geval. In normaal Nederlands komt bijvoorbeeld de letter e vaak voor, en de letter q nauwelijks. Als je met de letterfrequenties van de Nederlandse taal de entropie uitrekent, kom je op 4,15 bits/letter (hoewel dit nog afhankelijk is van het type tekst, van het gebruikte vakjargon). De entropie is dus kleiner dan de waarde 4,700… die je zou krijgen als alle symbolen even waarschijnlijk zijn. Dit wordt veroorzaakt door het feit dat sommige letters veel vaker voorkomen dan andere. Als je ook nog de context van de letters meeneemt (welke combinaties, of woorden kunnen voorkomen?), dan blijkt de entropie nog veel lager te zijn, ongeveer 1,5.

2 Communicatiekanalen

Geld moet rollen en ook informatie is nutteloos als ze niet uitgewisseld kan worden. We zullen nu een aantal eigenschappen van informatiekanalen bekijken.

2.1 Wederzijdse informatie

De wederzijdse (gemeenschappelijke) informatie is zoiets als de hoeveelheid informatie (de onzekerheid) over een voorval X, als een voorval Y plaatsvindt. Je kunt in een communicatiesysteem de verzonden boodschap als X zien en de ontvangen boodschap als Y. Bij normale verzending wil je de wederzijdse informatie tussen het verzonden en het ontvangen bericht zo groot mogelijk hebben: Als je een boodschap verstuurt, wil je hem namelijk zo volledig mogelijk ontvangen. Als je een boodschap versleutelt, wil je dat de wederzijdse informatie tussen origineel bericht en versleutelde boodschap juist minimaal is: Iemand die de versleutelde boodschap ontvangt moet daaruit niets te weten kunnen komen over het origineel. De wederzijdse informatie tussen twee berichten die niets met elkaar te maken hebben moet natuurlijk ook zo klein mogelijk zijn.

De **wederzijdse informatie** van twee informatiebronnen X en Y is gedefinieerd als

$$I(X,Y) = \sum_{x\in X,\, y\in Y} p(x,y)\ {}^{2}\log \frac{p(x,y)}{p(x)\,p(y)} \text{ (bits).}$$

Je moet hierbij sommeren over alle mogelijke combinaties van een letter x uit het alfabet van X en een letter y uit het alfabet van Y. Deze letters hebben kansen p(x) en p(y). De kans p(x,y) is de kans dat x en y gelijktijdig optreden. Als de twee informatiebronnen onafhankelijk zijn, dan is de kans dat x en y tegelijk optreden gelijk aan het product van de kansen op x en y apart: p(x,y) = p(x)p(y). Dit betekent dat er in de formule telkens $^{2}\log 1 = 0$ staat, dus de wederzijdse informatie is dan gelijk aan nul.

Voorbeeld: Voor een binaire bron geldt dat $p_{in}(0) = 0{,}4$ en $p_{in}(1) = 0{,}6$. Dit signaal gaat door een kanaal, dat de informatie doorgeeft met een foutkans van 5%. Dat betekent dat p(0,0) = 0,4 x 0,95 = 0,38, p(1,1) = 0,6 x 0,95 = 0,57, p(0,1) = 0,4 x 0,05 = 0,02 en p(1,0) = 0,6 x 0,05 = 0,03 (p(0,1) is bijvoorbeeld de kans dat er een 0 in het kanaal gaat, terwijl er een 1 uitkomt). Op de uitgang krijg je dus een 0 met kans $p_{uit}(0) = p(0,0) + p(1,0) = 0{,}4 \times 0{,}95 + 0{,}6 \times 0{,}05 = 0{,}41$ (een 0 op de ingang die goed door het kanaal gaat, of een 1 die er fout doorheen gaat). De kans op een 1 is $p_{uit}(1) = 0{,}4 \times 0{,}05 + 0{,}6 \times 0{,}95 = 0{,}59$. De wederzijdse informatie tussen ingang X en uitgang Y is dan

$$I(X,Y) = p(0,0)\ {}^{2}\log \frac{p(0,0)}{p_{in}(0)\,p_{uit}(0)} + p(0,1)\ {}^{2}\log \frac{p(0,1)}{p_{in}(0)\,p_{uit}(1)} +$$
$$+ p(1,0)\ {}^{2}\log \frac{p(1,0)}{p_{in}(1)\,p_{uit}(0)} + p(1,1)\ {}^{2}\log \frac{p(1,1)}{p_{in}(1)\,p_{uit}(1)} =$$

$$= 0{,}46065\ldots - 0{,}07121\ldots - 0{,}09106\ldots + 0{,}39171\ldots = 0{,}69010\ldots \text{ bits.}$$

De wederzijdse informatie kun je zien als de informatie die van X naar Y gaat. In het bovenstaande voorbeeld betekent dat bijvoorbeeld dat van de entropie van

$$H(X) = -0{,}4\ {}^{2}\log 0{,}4 - 0{,}6\ {}^{2}\log 0{,}6 = 0{,}9709\ldots \text{ bits}$$

die het kanaal in gaat, gemiddeld 0,69010… bits worden doorgegeven doordat er een transmissiefout van 5% wordt gemaakt.
Er zijn twee extreme situaties in het bovenstaande voorbeeld die we kunnen bekijken. De eerste situatie treedt op als de input en de output van het kanaal volledig ongerelateerd zijn, onafhankelijk van elkaar. In dat geval geldt bijvoorbeeld $p(0,1) = p_{in}(0)p_{uit}(1)$, er is geen verband tussen het optreden van een 0 aan de ingang en een 1 aan de uitgang. Analoge formules gelden voor alle andere gezamenlijke kansen. De wederzijdse informatie is in dat geval gelijk aan 0, zoals je zou verwachten. Er is geen informatieoverdracht tussen ingang en uitgang.
In de tweede situatie is het kanaal perfect. Een 0 of een 1 wordt ook precies als een 0 of een 1 doorgegeven. Dat betekent dat $p(0,0) = p_{in}(0)$ en $p(1,1) = p_{in}(1)$: als er een 0 op de invoer staat wordt de 0 naar de uitvoer doorgegeven. Verder is $p(0,1) = p(1,0) = 0$: er kan geen bit fout worden doorgegeven, een bit kan in het kanaal niet van teken veranderen. Hierdoor is ook $p_{in}(0) = p_{uit}(0)$ en $p_{in}(1) = p_{uit}(1)$. Voor de wederzijdse informatie krijgen we:

$$I(X,Y) = p(0,0)\ {}^2\log\frac{p(0,0)}{p_{in}(0)\,p_{uit}(0)} + p(0,1)\ {}^2\log\frac{p(0,1)}{p_{in}(0)\,p_{uit}(1)} +$$
$$+ p(1,0)\ {}^2\log\frac{p(1,0)}{p_{in}(1)\,p_{uit}(0)} + p(1,1)\ {}^2\log\frac{p(1,1)}{p_{in}(1)\,p_{uit}(1)} =$$
$$= p_{in}(0)\ {}^2\log\frac{1}{p_{in}(0)} + p_{in}(1)\ {}^2\log\frac{1}{p_{in}(1)} = H(X)\,.$$

De wederzijdse informatie is dus voor een perfect kanaal gelijk aan de entropie die je erin stopt: Alle informatie die je erin stopt komt er doorheen.

In het algemeen geldt voor een kanaal waarin fouten worden gemaakt, dat van de informatie H(X) die erin wordt gestopt maar een deel overkomt, namelijk: I(X,Y). Er geldt:

$$0 \le I(X,Y) \le H(X) \le H(Y).$$

De eerste ongelijkheid $0 \le I(X,Y)$ zegt dat I(X,Y) inderdaad een informatiemaat kan zijn. Er gaat altijd iets positiefs van ingang naar uitgang. In het ergste geval is de overgedragen informatie gelijk aan nul. Dat kan alleen als X en Y onafhankelijk zijn.
De tweede ongelijkheid $I(X,Y) \le H(X)$ zegt dat een kanaal nooit meer informatie kan doorspelen dan erin wordt gestopt (namelijk H(X)). Het kanaal kan wel lekken: er kan minder door gaan (I(X,Y)) dan erin komt (H(X)).
De derde ongelijkheid $H(X) \le H(Y)$, zegt dat een kanaal met ruis de informatie-inhoud niet kleiner maakt. Dat wil zeggen dat de onzekerheid over het volgende symbool minstens even groot is als de onzekerheid van het symbool dat je erin stopt. Dat komt omdat de uitvoer als gevolg van ruis meer random is dan de invoer (Je herinnert je dat een volledig random signaal de maximale entropie heeft en dus ook de maximale hoeveelheid informatie). In het bovenstaande voorbeeld met een foutkans begon je met kansen 0,4 en 0,6 voor een 0 en een 1. Na het kanaal is dat 0,41 en 0,59. Dat lijkt meer op de random verdeling 0,5 en 0,5. De entropie neemt daardoor ook toe:

$$H(X) = -0{,}4\ {}^2\log 0{,}4 - 0{,}6\ {}^2\log 0{,}6 = 0{,}9709\ldots \text{ bits},$$

$$H(Y) = -0{,}41\ {}^2\log 0{,}41 - 0{,}59\ {}^2\log 0{,}59 = 0{,}8926\ldots \text{ bits}.$$

Als een voorbeeld van wat er in een kanaal aan het origineel kan veranderen is een keer het volgende Engelse gedicht in het Frans vertaald, en vervolgens door een andere vertaler weer terugvertaald in het Engels. De originele tekst was als volgt:

The turtle lives 'twixt plated decks
Which practically conceal its sex.
I think it clever of the turtle
In such a fix to be so fertile.

Na vertaling in het Frans en weer terug naar het Engels stond er:

The turtle lives in a scaled carapace
which in fact hides its sex.
I find that it is clever for the turtle
to be so fertile in such a tricky situation.

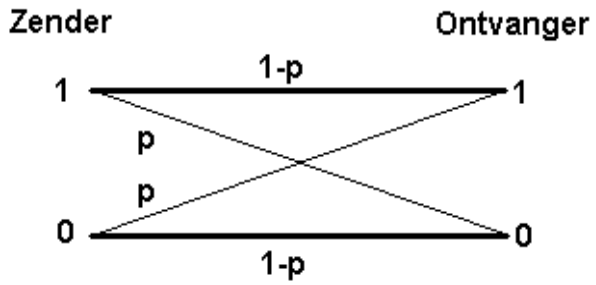
2.2 Kanaalcapaciteit

We hebben gezien dat de wederzijdse informatie I(X,Y) een maat is voor de hoeveelheid informatie die door een kanaal heen komt. Deze hoeveelheid hangt nog af van de kansen van de symbolen in X. Je kunt je voorstellen dat je deze kansen zodanig bijstelt dat je een maximale hoeveelheid informatie door het kanaal heen krijgt. De **kanaalcapaciteit** is dan ook gedefinieerd als

$$C = \max_{p(x)} I(X,Y).$$

In het algemeen is deze waarde voor een kanaal moeilijk uit te rekenen. We bekijken nu een simpele situatie waarvoor dit wel gaat:

Een eenvoudig model voor datatransmissie over een gestoord kanaal is het "Binair symmetrisch kanaal" (*Binary Symmetric Channel* BSC), zie de volgende figuur:



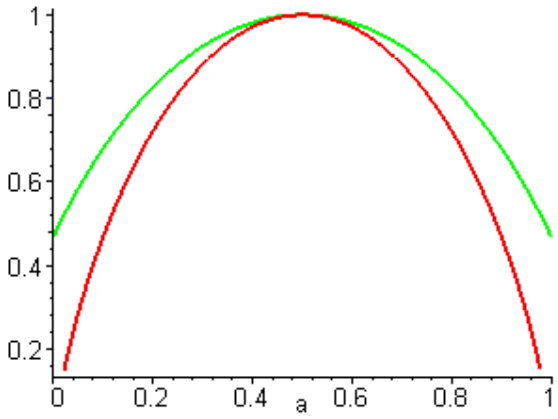
Het binair symmetrische kanaal (BSC).

Over het kanaal worden bits verstuurd, maar er is een kans p dat de transmissie een fout introduceert. Een 0 wordt dus met kans p een 1, en een 1 wordt met kans p een 0 (gelijke kansen, vandaar symmetrisch). Het kanaal is ook "geheugenloos", net als bij een dobbelsteen: de foutkansen hangen niet af van eerder verzonden bits. De kanaalcapaciteit voor dit kanaal blijkt te zijn:

$$C = 1 + p\ {}^{2}\log p + (1-p)\ {}^{2}\log (1-p).$$

Als de foutkans p gelijk is aan nul is de kanaalcapaciteit maximaal: gelijk aan 1. Alle informatie komt door het kanaal. Hetzelfde geldt als p = 1 (alleen worden 0 en 1 dan omgekeerd in het proces). Als p = 0,5, dan is de kanaalcapaciteit gelijk aan 0. Er komt geen informatie meer door het kanaal. De kans op een 0 of een 1 op de uitgang is 0,5, onafhankelijk van de invoer.

Als er fouten kunnen optreden in een kanaal zal dit de entropie doen toenemen. Dit lijkt in eerste instantie misschien tegenstrijdig. Je voegt ruis toe en krijgt daardoor meer informatie. Je moet informatie hier echter zien als onzekerheid. De onzekerheid neemt toe.



Entropie als functie van de kans a op een 0 vóór (onder) en ná (boven) een BSC met foutkans 10%.

3 Datacompressie

3.1 Inleiding

In dit hoofdstuk gaan we in op datacompressie. De bedoeling van datacompressie is om informatie zo compact mogelijk te coderen. Dit kan gebeuren om verzenden sneller en goedkoper te maken of opslaan goedkoper en efficiënter te doen. Hierbij kunnen verschillende afwegingen een rol spelen.
Informatie die gecomprimeerd wordt moet natuurlijk ook weer kunnen worden gedecomprimeerd. In het algemeen wil je dat een boodschap die je comprimeert precies zo is te decomprimeren (***lossless compression***), zonder dat er gegevens verloren gaan. Dit geldt met name voor data. Daarbij kan ieder bit van belang zijn. Als je een tekstbestand opslaat wil je niet dat er na decompressie hier en daar letters zijn verdwenen of zijn veranderd.
In andere toepassingen is het soms niet zo erg als er wat verlies van gegevens optreedt. Het kan zijn dat transmissiesnelheid of beperkte opslagcapaciteit hier veel belangrijker zijn en de doorslag geven. Zo is het bij het opslaan van audio- en videoinformatie mogelijk om een gedeelte van de informatie weg te laten zonder dat de uiteindelijke kwaliteit daar merkbaar onder lijdt. Er wordt dan gesproken van ***lossy compression***, of irreversibele compressie, compressie die je niet voor 100% kunt reconstrueren, omdat je echt iets weglaat. De DCC recorder maakt(e) bijvoorbeeld gebruik van het feit dat in harde muziekpassages het oor bepaalde frequenties toch niet meer kan onderscheiden. Die informatie kun je dus weglaten. In bewegende beelden kan de resolutie van de afzonderlijke beelden beduidend lager zijn dan bij een stilstaand beeld. Door de beweging van de beelden en de traagheid van het oog wordt er een zekere scherpte gesuggereerd en is het resultaat veel acceptabeler dan je op grond van de afzonderlijke beelden zou inschatten. In het JPEG (Joint Photographics Experts Group) algoritme, dat wordt gebruikt voor compressie van plaatjes, wordt ook een lossy compressietechniek toegepast. De mate van geheugenverlies is daarbij in te stellen. Het is duidelijk dat hoe meer verlies je toestaat, hoe beter je compressie kan zijn, maar hoe slechter de kwaliteit van het plaatje. In de ultieme *lossy* compressie comprimeer je alles in één bit. Je hebt dan weliswaar een fenomenale compressieverhouding, maar het zal duidelijk zijn dat het zelfs met zeer veel moeite niet mogelijk is om uit deze ene bit weer enige relevante informatie te decomprimeren. Wij zullen in het vervolg aandacht besteden aan verliesvrije compressie.

De bruikbaarheid van een compressiealgoritme wordt door een drietal aspecten bepaald.
1. Allereerst is er de compressiefactor. Hoeveel procent wordt het bericht kleiner?
2. Vervolgens is van belang hoeveel tijd de compressie van de data in beslag neemt.
3. Tenslotte is ook de decompressietijd belangrijk. Hoeveel tijd kost het om over de data te kunnen beschikken?

Voor veel toepassingen is de compressiefactor het belangrijkst, want dat heeft te maken met kosten en beschikbare ruimte van opslag, of benodigde bandbreedte van een kanaal. Het bereiken van een hoge compressiefactor betekent in het algemeen een ingewikkeld algoritme dat een optimale codering moet vinden, en daardoor veel compressietijd in beslag neemt, en misschien ook wel veel decompressietijd. Die tijd is er niet altijd. Als je een telefoongesprek digitaal via een ATM netwerk wilt voeren is de gemiddelde gebruikte bandbreedte van belang, dus ligt het voor de hand om compressie toe te passen. Compressie en decompressie moeten echter razendsnel kunnen gebeuren, anders wordt zo'n gesprek of film tamelijk onprettig. Soms zie je op de televisie telefonische interviews die via een satelliet lopen, waaruit je ziet dat een vertraging van enkele seconden al heel erg hinderlijk kan zijn voor het voeren van een normaal gesprek. Iets soortgelijks geldt ook in iets mindere mate voor *video-on-demand* systemen en bij fax, hoewel daar enigszins gebufferd kan worden.

Soms mag compressie best lang duren, als decompressie maar razendsnel gaat. Denk bijvoorbeeld aan een encyclopedie die gecomprimeerd op een CD-ROM of DVD is opgeslagen. Compressie mag lang duren, want dat hoeft maar één keer te gebeuren, vóór het persen van de schijf. Het resultaat moet alleen snel kunnen worden geraadpleegd, dus decompressie moet erg snel plaatsvinden. Hetzelfde geldt in het algemeen voor een database die zuinig met geheugen moet omspringen, maar weinig beperkingen op de accesstijd moet hebben. Als de data eerst rustig van tevoren mag worden gecomprimeerd kun je ook andere typen algoritmes toepassen dan wanneer je *on the fly* moet comprimeren en je niet weet wat er in het vervolg nog aan data komt. In het eerste geval kun je eerst de hele tekst analyseren om te kijken hoe je het beste kunt comprimeren, terwijl je in het tweede geval moet comprimeren terwijl je de data nog binnen moet krijgen en je dus alleen de beschikking hebt over de data die tot dan verzonden is.

De performance van een compressiealgoritme kan sterk afhankelijk zijn van het type data dat wordt gecomprimeerd. Om optimaal te functioneren moet een algoritme rekening houden met de karakteristieken van de data. Dat betekent dat er in de praktijk vaak verschillende compressiemethoden zijn voor tekst, audio, grafische data, video, etc.

3.2 Run-length encoding

Het idee achter ***run-length-encoding*** is simpel: Als een bepaald symbool een groot aantal malen achter elkaar voor komt zou je dat korter kunnen coderen door dat symbool maar éénmaal noemen, met daarbij het aantal malen dat het symbool voor komt. Bij *run-length encoding* onderscheidt men twee soorten data in de originele boodschap. Een **run** is een rijtje van n gelijke symbolen. Dit wordt gecodeerd als – (n–1) met daarbij de code voor het symbool (omdat 0 en 1 keer een letter toch niet worden gecomprimeerd). Daarnaast is er een ***sequence***, een rijtje van m symbolen waarin geen runs voorkomen. Een *sequence* wordt niet gecomprimeerd en wordt gecodeerd als (m-1), gevolgd door het rijtje, waarin m de lengte van de *sequence* is.

Voorbeeld: het rijtje XXXXXXXXXPOPIEJOPIE wordt gecodeerd als -8X9POPIEJOPIE.

Het is duidelijk dat er alleen compressie optreedt als er lange rijen van gelijke symbolen voor komen. Dit is lang niet bij elk type informatiebron het geval. Random data is het ergste dat je kunt hebben. Hierbij treedt totaal geen compressie op. In tegendeel: de tekst wordt er door de overhead langer op. *Run-length encoding* wordt wel gebruikt voor bitmap plaatjes. Daar komt het veel voor dat er grote vlakken met gelijke kleur zijn. Dat levert lange rijen van gelijke pixelwaarden op. *Run-length encoding* wordt onder andere toegepast in het Windows bitmap formaat BMP en in TIFF. De oriëntatie van een plaatje kan uitmaken voor de compressieverhouding die je kunt halen. Het kan zijn dat een plaatje dat 90^o wordt geroteerd ineens kleiner wordt, doordat er in de kolommen langere runs voor komen dan in de rijen.

3.3 Woordenboekcompressie

Bij compressie wordt er vaak vanuit gegaan dat bepaalde symbolen vaker voorkomen dan andere. Op de een of andere manier moet een compressiealgoritme hiervan gebruik maken om compressie te bewerkstelligen.

Als voorbeeld kijken we naar een taal met twee symbolen: A en B (je mag hiervoor ook lezen: 0 en 1). We nemen aan dat de A tweemaal zo vaak voor komt als de B. Om het nog eenvoudiger te maken nemen we aan dat er alleen woorden van zes symbolen worden ge-

bruikt, die elk vier A's en twee B's bevatten. Deze aanname betekent dat we niet alle mogelijke $2^6 = 64$ woorden met zes A's en/of B's hebben, maar alleen de volgende 15:

Woord	Nummer
AAAABB	0
AAABAB	1
AAABBA	2
AABAAB	3
AABABA	4
AABBAA	5
ABAAAB	6
ABAABA	7
ABABAA	8
ABBAAA	9
BAAAAB	10
BAAABA	11
BAABAA	12
BABAAA	13
BBAAAA	14

De 15 overgebleven woorden kunnen we nummeren van 0 tot en met 14, zoals dat in de tabel is te zien. Als we dit nummer gebruiken om het woord te coderen hebben we per woord 4 bits nodig.
Hoe vergelijkt dit nu met de gemiddelde hoeveelheid informatie die in deze taal aanwezig is? Hiervoor moeten we de entropie uitrekenen:

$$H = -\frac{1}{3}\log\frac{1}{3} - \frac{2}{3}\log\frac{2}{3} = \log 3 - \frac{2}{3} = 0{,}918\ldots \text{ bits/letter.}$$

Dit is minder dan de voor de hand liggende codering A→0, B→1, die 1 bit per letter oplevert. Voor woorden van zes letters komen we met de entropie op 6 x 0,918… = 5,509… bits aan informatie. Hoe kan dit nu? We hebben hier een codering gevonden die maar 4 bits nodig heeft, minder dan er aan informatie in de taal zit. Een deel van de verklaring zit in het feit dat we meer hebben verondersteld dan dat de letter A een kans 2/3 heeft en B een kans van 1/3. We hebben namelijk aangenomen dat in elk woord van zes letters lang deze kansen exact zijn. Als je dit algoritme namelijk uit wilt breiden voor langere woorden (teksten) zul je naast de codering namelijk ook de lengte van de tekst moeten opnemen. In ons geval zou bijvoorbeeld het woord BAAABA de code 110.1011 krijgen (het woord is 110 = 6 letters lang, het nummer van het woord is 1011 = 11). Als we dit aantal letters meenemen hebben we 7 bits nodig in plaats van de 4 die we eerst hadden. Dit is ineens beduidend meer dan de 5,5 die er volgens de entropie in zitten.

Toch is deze methode in het algemeen nog niet zo slecht. Het blijkt namelijk dat als je dit doet voor langere teksten dan de zes letter die we daarnet hebben bekeken, dat de compressie steeds beter wordt, en zelfs willekeurig dicht in de buurt kan komen van de waarde die door de entropie wordt gegeven. Dit zullen we nu gaan afleiden, een smulpartij voor de liefhebber!

We nemen aan dat we een tekst van $N = 3n$ letters hebben, namelijk $2n$ A's en n B's. Deze tekst wordt nu gecodeerd als [N].[nummer]. Hierin is N de lengte van de tekst en nummer is het nummer van de betreffende tekst in het woordenboek waarin alle woorden (teksten) van N lang staan die uit $2n$ A's en n B's bestaan. Dit zal in het algemeen een zeer groot woordenboek zijn. Het aantal woorden in dit woordenboek is namelijk

$$\binom{N}{n} = \frac{(3n)!}{n!(2n)!},$$

het aantal manieren waarop je n B's kunt plaatsen in een rijtje van N = 3n symbolen.

In het algemeen geldt dat de lengte (in bits) van de gecodeerde tekst gelijk is aan

$$^2\log N + {}^2\log\binom{N}{n}.$$

Om de waarde hiervan te bepalen voor grote waarden van n hebben we een formule nodig die de waarde van n! voor grote waarden van n goed benadert. Dit is de **formule van Stirling**:

$$n! \approx \sqrt{2\pi}\, n^{n+\frac{1}{2}} e^{-n}.$$

De waarde van n! wordt hierin uitgedrukt in machten. Je ziet hieruit dat n! veel harder groeit dan exponentieel. Het gaat bijna als n^n. Hierin wordt niet alleen de exponent steeds groter, zoals bij exponentiele groei, maar ook het grondtal. Voor n=12 krijgen we bijvoorbeeld:

$$12! = 479001600,$$

terwijl met de formule van Stirling:

$$\sqrt{2\pi}\, 12^{12,5} e^{-12} = 4{,}756\ldots \cdot 10^8.$$

Deze benadering is behoorlijk goed. De eerste twee cijfers en de grootteorde kloppen.

Met de formule van Stirling kunnen we de volgende benadering maken:

$$\frac{(3n)!}{(2n)!n!} \approx \frac{\sqrt{2\pi}(3n)^{3n+\frac{1}{2}} e^{-3n}}{\sqrt{2\pi}(2n)^{2n+\frac{1}{2}} e^{-2n} \sqrt{2\pi} n^{n+\frac{1}{2}} e^{-n}} = \frac{(3n)^{3n+\frac{1}{2}}}{\sqrt{2\pi}(2n)^{2n+\frac{1}{2}} n^{n+\frac{1}{2}}} = \frac{3^{3n+\frac{1}{2}} n^{3n+\frac{1}{2}}}{\sqrt{2\pi}\, 2^{2n+\frac{1}{2}} n^{2n+\frac{1}{2}} n^{n+\frac{1}{2}}} = \frac{3^{3n+\frac{1}{2}}}{\sqrt{2\pi}\, 2^{2n+\frac{1}{2}} n^{\frac{1}{2}}}.$$

Uit deze benadering kunnen we een benadering voor de lengte van onze code afleiden:

$$^2\log N + {}^2\log\binom{N}{n} = {}^2\log(3n) + {}^2\log\left(\frac{3^{3n+\frac{1}{2}}}{\sqrt{2\pi}\, 2^{2n+\frac{1}{2}} n^{\frac{1}{2}}}\right) =$$

$$^2\log 3 + {}^2\log n + \left(3n+\frac{1}{2}\right)^2\log 3 - {}^2\log\sqrt{2\pi} - \left(2n+\frac{1}{2}\right) - \frac{1}{2}{}^2\log n.$$

Om het aantal bits per symbool te bepalen moeten we dit aantal delen door N = 3n:

$$^2\log 3 - \frac{2}{3} + \frac{\frac{3}{2}\,^2\log 3 - {}^2\log\sqrt{2\pi} - \frac{1}{2}\,^2\log 2 + \frac{1}{2}\,^2\log n}{3n}.$$

Voor grote waarden van n wordt dit (de laatste term, de breuk, wordt dan steeds kleiner):

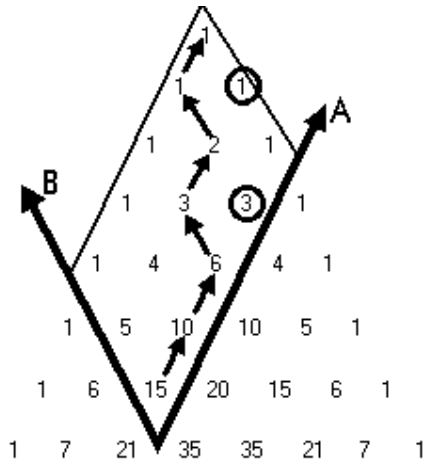
$$^2\log 3 - \frac{2}{3} = 0{,}918\ldots \text{ bits per letter.}$$

Dit is precies de waarde van de entropie, die we eerder al hadden uitgerekend. We hebben hiermee aangetoond dat als we de codering de lengte van de tekst samen met het nummer van de tekst in het woordenboek van alle teksten van N = 3n lang met 2n A's en n B's nemen, dat we met deze compressie voor lange teksten in de buurt van de entropie kunnen komen.

3.4 Het Pascal-driehoekalgoritme

Het algoritme dat we zo-even hebben bekeken codeert een tekst waarin een vast aantal A's en B's in voorkomen door een woordenboek te maken van alle mogelijke teksten met zoveel A's en B's. Deze teksten worden vervolgens genummerd en het nummer van de corresponderende tekst wordt gebruikt als codering. Nu is het in het algemeen ondoenlijk om een dergelijk woordenboek echt samen te stellen. Voor tekstjes van 300 letters met tweemaal zoveel A's als B's bevat het woordenboek bijvoorbeeld 415825146325856474478338352632640 558028046600574364870866303365730475632832400862O verschillende teksten. Dit is meer dan het geschatte aantal atomen in het heelal. Op een atoom kun je moeilijk nog een tekst van 300 letters schrijven, dus moeten we iets anders verzinnen.

Dat is het **Driehoek-van-Pascalalgoritme**. Dit algoritme vindt bij een bepaalde tekst een nummer, en omgekeerd, bij een nummer weer een tekst. Hoe dit werkt laten we zien aan de hand van de tekst AABABA, van zes letters lang, die tweemaal zoveel A's als B's bevat (we nemen even aan dat er niet meer B's dan A's zijn, anders wissel je de rol van A en B om). Allereerst maak je een driehoek van Pascal. Zo'n driehoek is opgebouwd met enen aan de rand en elk ander cijfer is telkens de som van de twee bovenstaande cijfers (bijvoorbeeld 21 = 6 + 15, zie de figuur hieronder). We hebben nu een woord met 4 A's en 2 B's. Je begint in de bovenste 1. Dan ga je 4 plaatsen naar linksonder (voor de 4 A's) langs de rand, en vervolgens 2 plaatsen naar rechtsonder (voor de twee B's), parallel aan de andere rand. Je komt dan uit in 15. Vanuit 15 maak je een pad naar de bovenste 1 door het woord AABABA te doorlopen. Voor een A ga je één plaats naar rechtsboven, voor een B één plaats naar linksboven. Als het woord op is ben je in de bovenste 1 terechtgekomen. Elke keer dat je in de B-richting gaat schrijf je telkens het getal op dat je in de A-richting zou hebben bereikt. In ons geval is dat 3 en 1. Deze getallen tel je op en je hebt het nummer van het woord AABABA: 3 + 1 = 4, ofwel binair: 0100.



Hoe vind je nu uit een nummer weer een woord terug? Je moet dan weten dat het een woord is met vier A's en twee B's. Dat brengt je weer naar het getal 15. Vanuit dit punt ga je net zolang in de A-richting totdat het getal wat je in de driehoek van Pascal krijgt kleiner is dan het nummer. Elke stap levert dan een A. Als het getal kleiner of gelijk is aan het nummer trek je het van het nummer af en ga je één stap in de B-richting. Dat levert een B. Daar doe je weer hetzelfde en probeer je weer zolang mogelijk in de A-richting te gaan. Voor het nummer 4 krijg je zo bijvoorbeeld:

$4 < 10$	$\rightarrow$	A
$4 < 6$	$\rightarrow$	A
$4 \geq 3$, nummer $4 \rightarrow 4 - 3 = 1$	$\rightarrow$	B
$1 < 2$	$\rightarrow$	A
$1 \geq 1$, nummer $1 \rightarrow 1 - 1 = 0$	$\rightarrow$	B
$0 < 1$	$\rightarrow$	A

4 is dus het nummer van AABABA.

3.5 Prefixcodes

Het coderen van informatie per symbool kan op twee manieren: De eerste manier is dat je elk symbool codeert met een codewoord van een vaste lengte. We hebben hiervan net een voorbeeld gezien. Bij het decoderen hoef je alleen maar je vaste lengte van de codewoorden te weten. Vervolgens knip je de tekst die je moet decoderen in allemaal stukjes van die lengte en kun je elk van die stukjes (de codewoorden) decoderen tot letters.
Een andere manier is om een code gebruiken waarvan de lengte van de codewoorden per symbool verschilt, om flexibeler te zijn. In dat laatste geval heb je een probleem bij het decoderen, want hoe weet je waar een code eindigt en een volgende begint? Je kunt dan niet meer tellen en groepjes maken die te decoderen zijn. Er zijn hiervoor een aantal mogelijke oplossingen. De eerste is door gebruik te maken van een zogenaamde **kommacode**. Het idee is daarbij dat je de codewoorden scheidt met behulp van een scheidingsteken, de komma. Je weet dan dat een codewoord begint na een komma en ophoudt net voor de volgende komma. Deze code levert wel enige overhead, omdat je per woord een komma moet gebruiken. Als je bijvoorbeeld een bericht binair wilt coderen dan moet je één van de tekens reserveren als komma. Je kunt bijvoorbeeld de 0 daarvoor gebruiken. Dat betekent helaas dat je maar één symbool over hebt om codewoorden te maken, namelijk de 1. Dat geeft je niet veel

vrijheid. Codewoorden zijn dan: 1, 11, 111, 1111, etc. Een tekst met deze binaire kommacode kan eruit zien als:

1101011111101111111111011111111111101111011111111111

Deze tekst bevat zeven codewoorden.

Een andere manier om ervoor te zorgen dat je weet wanneer een codewoord is afgelopen werkt als volgt: Tijdens het decoderen voeg je telkens letters toe aan een nieuw woord, totdat je een codewoord uit de lijst vindt. Vervolgens begin je aan het volgende woord. Je hebt zo geen scheidingsteken nodig, maar het geeft wel een restrictie aan de codewoorden. die je daarvoor kunt gebruiken. Een codewoord mag namelijk niet het begin vormen van een ander codewoord. Deze manier van coderen heet een prefixcode.
Een **prefixcode** is een code waarin geen enkel codewoord ook de prefix (het begin) is van een ander codewoord (de codewoorden 101 en 1011 mogen dus niet tegelijk). Een prefixcode is daarmee een code die van links naar rechts wordt gelezen en ontcijferd. De ontvanger bouwt een woord op door telkens een symbool toe te voegen en te kijken of het nieuwe woord een codewoord is. Als dat niet zo is wordt er een letter aan het woord toegevoegd en als het wel een codewoord is, wordt het vertaald en wordt er met het volgende symbool aan een nieuw woord begonnen. De codewoorden hoeven niet even lang te zijn en er wordt ook geen scheiding (komma) tussen de woorden aangegeven. Het gecodeerde bericht mag geen enkele fout bevatten, anders gaan het decoderen ontzettend mis! Een suffixcode is bijna hetzelfde als een prefixcode, alleen ga je dan van achteren naar voren coderen en decoderen.

Voorbeeld van een prefixcode:
E → 0
O → 10
I → 110
L → 1110
B → 1111
Nu is 10111011001111101110 als volgt te decoderen:
10-1110-110-0-1111-10-1110 → OLIEBOL.

Voorbeeld van een code die geen prefixcode is:
A → 0
B → 10
C → 100
Het bericht 100 kan nu worden vertaald als C, maar ook als BA.

Ook de volgende code is geen prefixcode:
A → 1
B → 10
C → 100
In het bericht 10 kun je de eerste 1 vertalen door A, maar vervolgens heb je een probleem, want de 0 erna hoort niet meer bij een codewoord, dus de ontvanger kan het bericht niet ontcijferen. Dit komt omdat er een codewoord (A en B) is waarvan de code het beginstuk vormt van een andere code (B en C).
De code is trouwens wel uniek te decoderen, alleen niet op de prefixmanier. Je moet hier net zolang letters toevoegen tot het géén codewoord meer is. Een letter er weer af geeft dan het codewoord.

Er bestaat een criterium waaraan je kunt zien of een prefixcode met bepaalde woordlengtes bestaat, namelijk de **ongelijkheid van Kraft en McMillan**: Een binaire prefixcode met woordlengtes a_1, a_2, …, a_n bestaat precies dan als

$$\frac{1}{2^{a_1}} + \frac{1}{2^{a_2}} + \cdots + \frac{1}{2^{a_n}} \leq 1.$$

Zo kun je een code maken met woordlengtes 1, 2, 3, 3, maar niet met 1, 3, 3, 3, 3, 3, want

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} = 1 \leq 1, \text{ maar } \frac{1}{2} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} = \frac{9}{8} > 1.$$

Een optimale prefixcode krijg je alleen als de lengte van een codewoord met kans p gelijk is aan $l = \lceil -^2\log p \rceil$. ($\lceil -^2\log p \rceil$ betekent: naar boven afronden op een geheel getal). Optimaal wil dan zeggen dat je een optimale compressie krijgt.
Prefixcodering wordt bijvoorbeeld gebruikt bij de Shannon-Fanocompressie en bij Huffman-codering die we hierna behandelen.
Het **broncoderingstheorema van Shannon** zegt dat je met een prefixcode een compressie kunt halen die ligt tussen de entropie H(x) en H(X) + 1.

3.6 Shannon-Fano compressie

In tekstbestanden worden karakters meestal opgeslagen als 1 byte (ASCII, EBCDIC van IBM) of 2 bytes (*unicode* ondersteund door Java). In de praktijk blijkt dat sommige karakters veel vaker voor komen dan anderen. Je kunt je voorstellen dat er winst te behalen valt als je de veel voorkomende symbolen met een kortere code representeert. Dit concept wordt gebruikt bij Shannon-Fanocompressie (1949). Elk symbool uit het alfabet wordt vervangen door een vaste prefixcode. Het idee is nu om symbolen die vaak voor komen een korte code te geven, terwijl symbolen die weinig voor komen best een langere code mogen hebben, juist omdat ze toch weinig voor komen.

Het algoritme werkt als volgt:
1. Maak een lijst met alle symbolen en bepaal hoe vaak elk symbool voor komt (frequentie).
2. Orden de symbolen naar frequentie, de meest voorkomende bovenaan.
3. Verdeel de lijst in tweeën, zodanig dat de som van de frequenties in de bovenste helft **zo dicht mogelijk** bij die van de onderste helft ligt (als er twee gelijke mogelijkheden zijn kies je die waarbij de som in de bovenste helft het grootst is).
4. De bovenste helft krijgt het symbool 0, de onderste helft een 1.
5. Pas stap 3 en 4 recursief toe op de elk van de twee helften.

Laten we dit eens toepassen op de volgende regel:

negen repelsteeltjes reden tegen een steen

Hierin komen 11 verschillende symbolen voor, in totaal 42 symbolen. We krijgen de volgende onderverdeling:

symbool	**aantal**			
e	15	**0**	**0**	
n	6		**1**	
spatie	5			**0**

t	4	**1**	**0**	**1**	**0**	
s	3				**1**	
g	2		**1**	**0**	**0**	
l	2				**1**	
r	2			**1**	**0**	**0**
d	1					**1**
j	1				**1**	**0**
p	1					**1**

De eerste scheiding deelt de aantallen precies doormidden: e+n = 15+6 = 21, spatie+t+s+g+l +r+d+j+p = 21. In de tweede helft kun je de volgende scheiding op twee manieren maken: spatie+t = 9, rest = 12, of spatie+t+s = 12, rest is 9. We kiezen de tweede. Het codewoord van een letter krijg je nu door de bijbehorende nullen en enen in de tabel van links naar rechts te lezen, bijvoorbeeld: e is 00, en g is 1100.

De code die we zo krijgen is een prefixcode. Zo wordt de vaak voorkomende ***n*** gecodeerd met de korte code 01, terwijl de ***j*** die maar één keer voor komt een langere code heeft: 11110. Het eerste woord ***negen*** wordt gecodeerd als 010011000001 (=01|00|1100|00|01).
Het totale aantal bits dat nodig is voor de hele tekst is:

$$15x2 + 6x2 + 5x3 + 4x4 + 3x4 + 2x4 + 2x4 + 2x5 + 1x5 + 1x5 + 1x5 = 126 \text{ bits.}$$

Dit is beter dan binair coderen. We hadden namelijk ook alle 11 symbolen een binaire code kunnen geven (e=0000, n=0001, etc.). Daarvoor hebben we per symbool vier bits nodig, want er zijn elf symbolen. Voor de totale tekst van 42 symbolen hebben we dus 4x42=168 bits nodig.
We kunnen ook nog kijken naar het theoretische minimum. Hiervoor hebben we de entropie nodig:

$$-\frac{15}{42}\,{}^{2}\log\frac{15}{42} - \frac{6}{42}\,{}^{2}\log\frac{6}{42} - \ldots - \frac{1}{42}\,{}^{2}\log\frac{1}{42} = 2{,}904754\ldots \text{ bit per symbool.}$$

Dit levert voor 42 symbolen: 42x2,904754… = 121,999669… bits. Er zijn dus minstens 122 bits nodig. Dat betekent dat we met 126 lang niet slecht zitten. Gemiddeld gebruiken we dan 126 / 42 = 3 bits per woord.

In werkelijkheid wordt de gecodeerde tekst langer, want om te kunnen decoderen heb je ook de binaire boom nodig, dus die moet ook worden meegestuurd.

De Shannon-Fano compressie geeft een gemiddelde lengte L(X) waarvoor geldt:

$$H(X) \leq L(X) < H(X) + 2.$$

3.7 Huffmancompressie

De **Huffmancodering** (uit 1952) lijkt erg op de Shannon-Fanocodering, maar is net even anders. Het idee is nog steeds dat elk symbool wordt vervangen door een prefixcode waarbij de veel voorkomende symbolen een kortere code krijgen dan andere. Het algoritme werkt niet top-down, zoals het Shannon-Fanoalgoritme, maar bottom-up:

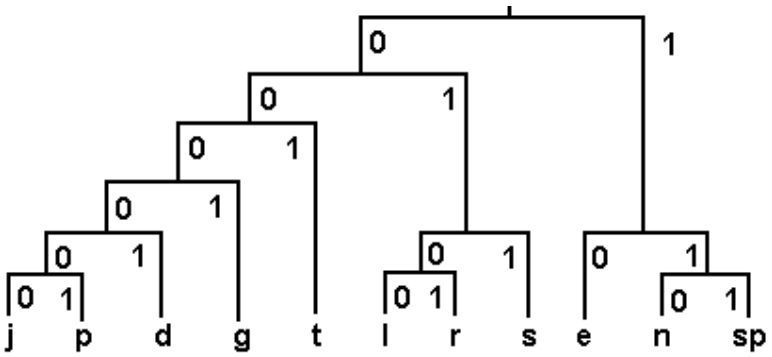
1. Maak een lijst met alle symbolen en bepaal hoe vaak elk symbool voor komt (d.w.z. de frequentie). Elk symbool wordt een knoop in een binaire boom.
2. Orden de knopen naar frequentie, de meest voorkomende bovenaan.

3. De twee knopen met de laagste frequentie worden samengevoegd tot een nieuwe knoop met als frequentie de som van de frequenties van de twee samengevoegde knopen.
4. Stap 2 en 3 worden net zolang uitgevoerd totdat de binaire boom (de **Huffmanboom**) af is, d.w.z., totdat je alles hebt samengevoegd tot één knoop.

Voor de zin

negen repelsteeltjes reden tegen een steen

levert dit de volgende binaire boom op:



Huffmanboom voor repelsteeltjes (sp=spatie).

Je ziet dat de codering iets anders uitpakt dan de Shannon-Fanocodering. Met deze Huffmancodering zijn er

$$15x2 + 6x3 + 5x3 + 4x3 + 3x3 + 2x4 + 2x4 + 2x4 + 1x5 + 1x6 + 1x6 = 125 \text{ bits}$$

nodig om de tekst te coderen, één bit minder dan bij de Shannon-Fanocodering. Dit is nog altijd drie bits groter dan het theoretisch minimum 122 dat uit de entropieberekening volgt. Het Huffmanalgoritme levert altijd een optimale (*minimum redundancy*) code, d.w.z.: Met een codelengte die per symbool wordt vastgelegd is dit het best haalbare.

Huffmancompressie geeft een gemiddelde gecodeerde lengte per symbool L(X) waarvoor geldt:

$$H(X) \leq L(X) < H(X) + 1.$$

Dit zegt dat Huffmancompressie behoorlijk optimaal is, want je kunt met de compressie heel dicht in de buurt van de entropie H(X) komen. Tevens zie je hieraan ook een beperking. Het kan zijn dat je bijna 1 van de entropie af zit. Dat is niet erg als de entropie per symbool groot is, maar als H(X) bijvoorbeeld gelijk is aan 2,3 is, dan maakt het veel uit of L(X) gelijk is aan 2,3 of aan 3,2.

3.8 Shannon's broncoderingstheorema

Shannon's broncoderingtheorema zegt het volgende: Als je de informatie uit een geheugenloze bron met entropie H (verliesloos) comprimeert door gebruik te maken van codewoorden met een vaste lengte tot R bits per letter, dan kan dit alleen als $R \geq H$.

3.9 Andere algoritmes

Tot nu toe hebben we compressiemethoden bekeken die elk symbool vervangen door een vast nieuw symbool. Soms kan het voordelig zijn om over meerdere symbolen tegelijk te kijken. In Nederlandse tekst komt de letter q nauwelijks voor, maar als hij voor komt, dan meestal in de combinatie qu. Daar zou je gebruik van kunnen maken door ook de frequentie van paren en tripels te bepalen en die in het algoritme mee te nemen. Dit worden **hogere orde methoden** genoemd.
Een nadeel van de compressietechnieken die we hiervoor hebben bekeken is dat de frequentie van de symbolen van tevoren bekend moet zijn. Als je een tekstbestand moet comprimeren kun je voordat je begint de frequenties bepalen, maar als je live audio of video moet comprimeren lukt dat niet. Je kunt dan wel de frequentiekarakteristiek van de symbolen bijhouden, terwijl je ze codeert. Dat betekent wel dat de code voor bijvoorbeeld de letter a in de loop van het coderingsproces kan veranderen. Als de letter vaker voor gaat komen zal hij een kortere code krijgen. Een algoritme dat dit doet is **adaptieve Huffmancodering**. De bijbehorende Huffmanboom kan tijdens het compressieproces veranderen. Dit algoritme wordt bijvoorbeeld gebruikt in het commando `compact` in Unix.
We hebben al gezien dat een nadeel van Hufmancompressie kan zijn dat je bijna 1 bit per symbool van de optimale entropie af kunt blijven. Dat komt omdat je voor elke letter een codewoord hebt dat een natuurlijk getal als lengte heeft. Een verbetering geeft **arithmetic coding**, waarbij de restrictie dat elke letter met een geheel aantal bits correspondeert is verdwenen.

Het **Lempel-Ziv-Welch** (LZW) algoritme (1977/84) bouwt adaptief (*on the fly*) een woordenboek op met bijbehorende codes, waarin woorden staan die vaak voor komen. De lengte van de woorden is variabel (hoewel ze bij grafische toepassingen vaak vast gekozen wordt), de lengte van de code die ze krijgen is vast. Het algoritme wordt onder andere gebruikt in het `compress` commando onder Unix, GNU gzip, ZIP en ARJ bestanden. Het wordt ook gebruikt in het GIF (Graphics Interchange Format) voor plaatjes. Daar heeft het het voordeel dat tijdens het decomprimeren alvast een beeld te tonen is met een lagere resolutie. Dat is handig, bijvoorbeeld in internetapplicaties.
Het originele algoritme LZW gebruikt een woordenboek met 4K entries, waarvan de eerste 256 de ASCII codes zijn. Voor een plaatje met 256 kleuren per pixel kan bijvoorbeeld een 12 bits compressiecode worden gebruikt. Van de twaalf bits zijn er 8 voor de pixelwaarde, dus er kunnen in het woordenboek 4096 woorden worden opgeslagen. Het algoritme begint pas na zo'n honderd bits effectief te comprimeren. Het decoderingsalgoritme dat hier bij hoort is zo slim dat het tijdens het decoderen het woordenboek kan reconstrueren, dus dat hoeft niet te worden meegestuurd. Het algoritme is snel, maar de compressiefactor is niet zo goed als bijvoorbeeld met *arithmetic coding* kan worden bereikt.

Een nieuwe compressietechniek voor plaatjes is **fractal compression**. Het idee hierbij is dat (kleine) onderdelen in een plaatje, bijvoorbeeld een licht-donker grens, ook op andere plekken weer kunnen optreden, misschien wat gedraaid en geschaald. Er is echter nog geen standaard op dit gebied.

Om de performance van compressiealgoritmes te meten zijn er een aantal standaardbibliotheken met typische bestanden in gebruik. Een voorbeeld is de Canterbury Corpus (http://www.cosc.canterbury.ac.nz/~tim/corpus/index.html), die bijvoorbeeld diverse tekstbestanden bevat (bijvoorbeeld de King James bijbel van 4 MB), plaatjes, HTML-files, source code, object files, datafiles executables, etc.

4 Foutcorrectie en detectie

4.1 Fouten

Als binaire informatie wordt opgeslagen of verzonden is er altijd een kans dat bits verkeerd worden geïnterpreteerd. Dit kan gebeuren als gevolg van defecten in opslagmedia (bijvoorbeeld een kras op een CD), elektronische ruis, defecten of thermische ruis in componenten, slechte verbinding, veroudering (bijvoorbeeld door invloed van zonlicht/warmte op floppies of zelfgebrande CD's). Als een bit verkeerd wordt gelezen, dan wordt een 0 gelezen als een 1 of een 1 als een 0, en we spreken van een bit-fout.
De foutkans is erg afhankelijk van het medium of de schakel in de keten die we bekijken. Zo heeft een magnetische (hard)disk een foutkans in de orde van 1 bit op elke 10^9 bits die gelezen worden. Een optische disk heeft een foutkans van 1 op de 100000 gelezen bits. Zonder foutcorrectie zouden veel opslagmedia te onbetrouwbaar zijn.

4.2 Foutdetecterende codes

Bij een foutdetecterende code wordt de oorspronkelijke informatie zodanig veranderd dat bepaalde fouten achteraf zijn te constateren. Er wordt bijvoorbeeld aan een groepje bits een aantal extra bits toegevoegd, redundante ("overbodige") controlebits, die niets toevoegen aan de informatie-inhoud van de gecodeerde boodschap, maar die alleen worden gebruikt om fouten te signaleren. Het toevoegen van redundantie maakt een taal robuuster. Een natuurlijke taal zoals het Nederlands is redundant, omdat maar een erg klein gedeelte van alle mogelijke woorden geldig zijn. Zo zijn er bijvoorbeeld 24 mogelijke woorden die je kunt samenstellen uit één a, één d, één n en één r:

adnr	danr	nadr	radn
adrn	darn	nard	**rand**
andr	dnar	ndar	rdan
anrd	dnra	ndra	rdna
ardn	dran	nrad	rnad
arnd	drna	nrda	rnda

Maar één van al deze woorden is een geldig woord: **rand**. Er zijn 26^7 = 8.031.810.176 mogelijke woorden van 7 letters, maar in de praktijk worden er daar maar zo'n 50.000 van gebruikt, dat is 0,0006 %. Deze redundantie maakt dat we een zin als

Dez# zin #evat a#lee# #aar onz##

toch nog redelijk goed kunnen lezen.

Het eerste idee dat misschien bij je opkomt als foutdetecterende code is om de hele boodschap tweemaal te versturen. Het bit 1 wordt dan verstuurd als het codewoord 11. Als dit wordt ontvangen als 01 of 10 is er iets mis en moet dit bit opnieuw worden verstuurd, want de ontvanger weet niet welke van de twee bits fout is, het had oorspronkelijk ook 00 kunnen zijn. De boodschap wordt door deze codering tweemaal zo lang. Het is het simpelste voorbeeld van een pariteitbit dat we hierna zullen bekijken. Veel foutdetecterende codes hebben ook een gedeeltelijk corrigerend karakter.

4.2.1 Pariteitbit

De eenvoudigste manier om te controleren of een boodschap juist is aangekomen is het gebruik van pariteitbits. Dit wordt bijvoorbeeld toegepast bij de ASCII codering. In één byte worden 128 verschillende symbolen gecodeerd. Hiervoor zijn 7 bits nodig. Het overgebleven bit is de binaire som van de andere zeven bits. Bij ontvangst kun je dit bit controleren door de eerste zeven bits weer op te tellen en het resultaat te vergelijken met het pariteitbit. Als het antwoord verschilt weet je zeker dat er iets mis is gegaan, maar niet wát er mis is. Het kan zijn dat één van de zeven informatiebits verkeerd is over gekomen. Het kan ook zijn dat er een fout in het pariteitbit is gemaakt (dat zou niet zo erg zijn, omdat de informatie er niet door wordt aangetast), maar er kunnen ook 3, 5 of 7 bits fout zijn. Als het controlebit klopt is de informatie waarschijnlijk goed overgekomen. Het kan echter ook zijn dat er twee bits fout zijn of zes, of zelfs acht. Deze simpele vorm van pariteitcontrole kun je dus alleen zinvol toepassen als de kans op twee fouten in een rijtje van 8 bits erg klein is.

4.2.2 De EAN code

Tegenwoordig staat op vrijwel elk artikel een barcode met een nummer. Deze code heet de *European Article Number* (EAN). De code werd in 1976 ingevoerd en is een variant op de Amerikaanse *Universal Product Code* (UPC). De EAN-13 code bestaat uit dertien cijfers $x_1x_2x_3x_4x_5x_6x_7x_8x_9x_{10}x_{11}x_{12}x_{13}$. De eerste twee cijfers vormen het landnummer (Nederland = 87), dan volgen vijf nummers die de fabrikant aangeven, vijf cijfers als artikelnummer, en tenslotte een controlegetal. Hierbij wordt x_{13} zo uitgerekend dat

$$x_1+3x_2+x_3+3x_4+ x_5+3x_6+x_7+3x_8+ x_9+3x_{10}+x_{11}+3x_{12}+ x_{13}$$

deelbaar is door 10. Zo is de code voor een TDK CD-R74: 4902030150655, en er geldt:
4+3x9+0+3x2+0+3x3+0+3x1+5+3x0+6+3x5+5 = 80.

Deze code kan één fout cijfer detecteren. Een fout cijfer geeft namelijk in de som een verschil van 1, 2, 3, 4, 5, 6, 7, 8 of 9. Dat verschil wordt eventueel vermenigvuldigd met 3. Dit totale verschil kan nooit deelbaar zijn door 10, dus het geeft een onjuiste check.
Verder wordt ook het omwisselen van twee cijfers die naast elkaar staan vaak gedetecteerd, maar niet altijd. Als in het bovenstaande voorbeeld de eerste twee cijfers worden verwisseld: **94**02030150655, dan is de som gelijk aan 70, ook een tienvoud, dus deze verwisseling wordt niet gedetecteerd:

In het algemeen wordt de verwisseling van twee buurcijfers niet opgemerkt als ze 5 verschillen.
Het doel van foutdetectie is hier om te voorkomen dat er tijdens het scannen van de barcode met een laser een fout wordt gemaakt (in de supermarkt geeft de scanner dan geen “Blieb!”)

Voor credit cards wordt een soortgelijk controlemechanisme gebruikt, maar dat is gebaseerd op het vermenigvuldigen met 1, 2, 1, 2, 1, 2, etc., in plaats van 1, 3, 1, 3, 1, 3, etc.

4.2.3 De ISBN codering

De ISBN (International Standard Book Number) codering, die sinds de jaren zeventig op vrijwel alle boeken staat is een zeldzaam voorbeeld van een geslaagde internationale samenwerking. De ISBN code op een boek bestaat uit tien cijfers, die vaak nog door streepje of spaties in groepen worden verdeeld, bijvoorbeeld: Het boek *Information Theory* – J. C. A. van der Lubbe, Cambridge University Press, 1997, heeft als ISBN: 0-521-46760-8.
De eerste 0 is een landcode, in dit geval een groep van Engelstalige landen als USA, Groot Brittannië, Canada en Australië. Kleinere landen hebben een langere code, zo gebruikt Nederland 90, Surinaams is 9942. De tweede groep staat voor de uitgever, hier 521 voor Cambridge University Press. Ook dit nummer is kort voor grote uitgevers en langer voor kleine uitgevers. Omdat dit nummer ook een variabele lengte heeft, en de schiedingsstreepjes tussen de nummers niet altijd worden vermeld, worden er alleen maar bepaalde combinaties uitgegeven: 00 – 19, 200 – 499, 5000 – 6999, 70000 – 79999, 800000 – 899999. Als zo’n nummer bijvoorbeeld met 5 begint weet je dat het uit 4 getallen bestaat. De derde groep is een nummer, dat de uitgever aan het boek geeft: 46760. Bij elkaar beslaan de eerste drie groepen in totaal negen cijfers. Het laatste cijfer is een check digit, een controlegetal, dat wordt gebruikt om de juistheid van het ISBN te kunnen nagaan. Dit getal wordt als volgt bepaald. Als de ISBN code het getal $x_1x_2x_3x_4x_5x_6x_7x_8x_9x_{10}$ is, dan kan het laatste cijfer als volgt worden uitgerekend:

$$x_{10} = (x_1+2x_2+3x_3+4x_4+5x_5+6x_6+7x_7+8x_8+9x_9) \pmod{11}.$$

In ons geval geldt voor het nummer 0-521-46760-8, dat het laatste cijfer als volgt wordt uitgerekend:

$$(1x0 + 2x5 + 3x2 + 4x1 + 5x4 + 6x6 + 7x7 + 8x6 + 9x0) = 173 \pmod{11} = 8.$$

Een probleem van het rekenen modulo 11 is dat er ook wel eens 10 uit kan komen. In dat geval wordt voor het laatste cijfer een X geschreven, om niet twee in plaats van één positie te gerbuiken, waardoor de lengte van de code langer, of niet meer constant zou zijn. Voorbeeld: 0-387-98643-X.

De ISBN code kan fouten in één cijfer constateren. Als er namelijk in één cijfer een fout wordt gemaakt, betekent dat in de som een verschil van 1, 2, …, of 9, vermenigvuldigd met de positie 1, 2, …, of 9. Dit getal is nooit deelbaar door 11, dus zo’n fout geeft modulo 11 een verschil in het eindantwoord. Dit geeft meteen aan waarom je modulo 11 rekent. Als je mo-

dulo 10 zou rekenen, dan geeft een fout met een verschil van 2 op de 5e plek een verschil 2x5 (mod 10) = 0.

Als bekend is op welke positie er een fout is gemaakt, dan kan de juiste waarde worden gevonden. Bijvoorbeeld: 173**k**600273 is een ISBN, dan moet gelden:

(1x1+2x7+3x3+4x**k**+5x6+6x0+7X0+8x2+9x7) mod 11 = 3
ofwel: (133+4**k**) mod 11= 3
dus: 4**k** mod 11 = (3-133) mod 11 = 2

Het enige getal dat hieraan voldoet is **k** = 6. Het juiste ISBN is dus 1736600273.
Als je niet weet waar de fout zit, dan kun je hem niet corrigeren. Als de som op 9 uitkomt, terwijl het laatste cijfer 1 is, dat kan er een fout ter grootte 8 op plek 1 zijn gemaakt, of een fout ter grootte 4 op plek 2, of een fout ter grootte 2 op plek 4, of een fout in de check digit. Deze code is dus geen foutcorrigerende code.

De ISBN code kan verder wel verwisseling van cijfers constateren, een menselijke fout die geregeld wordt gemaakt bij het overnemen van een ISBN. Dit is ook de reden waarom in de som elk cijfer met een verschillend getal wordt vermenigvuldigd. Niet alleen verwisseling van twee naburige cijfers wordt geconstateerd, zoals bij de EAN code, maar een verwisseling van twee willekeurige cijfers. Bijvoorbeeld:

0521467608: (1x0 + 2x5 + 3x2 + 4x1 + 5x4 + 6x6 + 7x7 + 8x6 + 9x0) = 173 = 8 (mod 11).

0**7**2146**5**608: (1x0 + 2x**7** + 3x2 + 4x1 + 5x4 + 6x6 + 7x**5** + 8x6 + 9x0) = **163** = **9** (mod 11).

ISBN's worden nog veel door mensen bewerkt (invullen op bestelling, doorbellen, etc.) Uit de onderstaande tabel zie je dat de foutdetectie in de ISBN codering minstens 90 % van de belangrijke fouten kan constateren.

Meestgemaakte menselijke fouten		
Transcriptie (foute letter)	a → b	79,1 %
Transpositie (verwisseling)	ab → ba	10,2 %
Sprongtranspositie	abc → cba	0,8 %
Tweelingfout (paarfout)	aa → bb	0,5 %
Fonetisch (verdubbeling)	a → aa	0,5 %
Random (iets anders)		8,9 %

In plaats van vermenigvuldigen met 1, 2, 3, 4, etc. kun je ook vermenigvuldigen met 10, 9, 8, 7, 6, etc. (waarom?)

De controle in het Nederlandse SOFI nummer is gebaseerd op een soortgelijk systeem. Het laatste cijfer 1 in het SOFI nummer 090274611 wordt als volgt bepaald:

(9x0 + 8x9 + 7x0 + 6x2 + 5x7 + 4x4 + 3x6 + 2x1) = 1755 = 1 (mod 11) .

Wat gebeurt er nu als hier 10 uit zou zijn gekomen? Nou niets! Deze nummers worden namelijk domweg niet uitgegeven!

4.2.4 De CRC code

CRC staat voor *Cyclical Redundancy Check*. Het is een soort gegeneraliseerde pariteitbit die je voor een heel computerbestand kunt uitrekenen. Laten we als voorbeeld de volgende fileinhoud nemen:

0010010100111010010100100001. We nemen nu de standaard CRC-16 bitstring: 11000000000000101 en we gaan deze bitstring vanaf links bij de filestring optellen (bitsgewijs) om alle 1-en weg te krijgen:

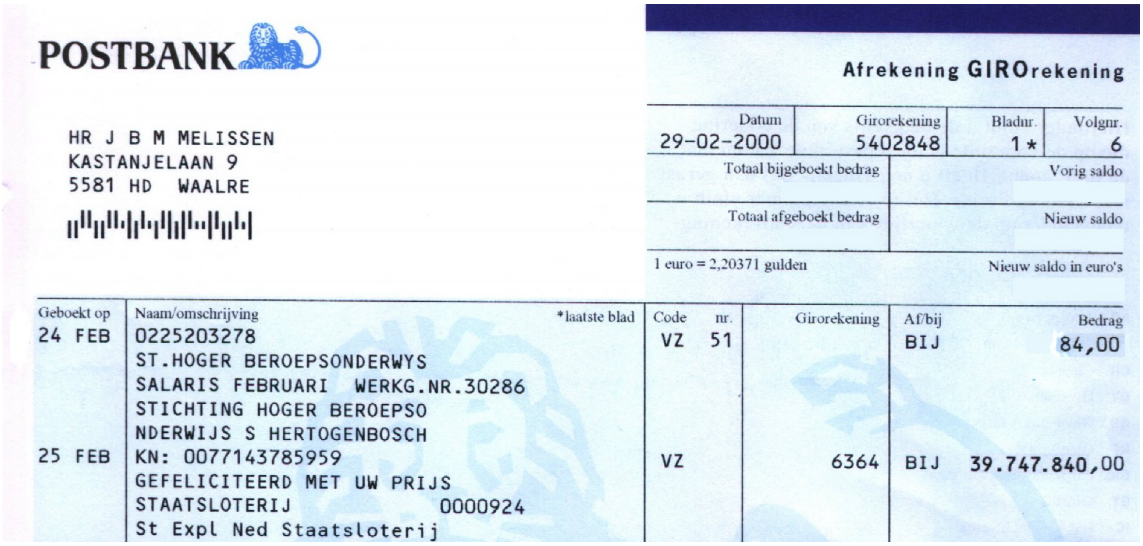
```
001001010011101001010010001
  110000000000000101
-----------------------------
000101010011101011110010001
   110000000000000101
-----------------------------
000011010011101010100010001
    110000000000000101
-----------------------------
000000010011101010001010001
        110000000000000101
-----------------------------
000000001011101010001111001
          110000000000000101
-----------------------------
000000000111101010001101101
            110000000000000101
-----------------------------
000000000001101010001100111
```

Deze laatste kan niet verder worden gereduceerd, dus dat is de CRC controlecode van deze file: 01101010001100111. Deze bewerking kan worden uitgevoerd voor een willekeurig lange file. De CRC code geeft een soort vingerafdruk van de file, waarin zo'n beetje alle bits uit de file zijn verwerkt. Deze controlecode wordt aan de file toegevoegd. Een of meerdere fouten ergens in de file (dat kan ook een ontbrekend stuk zijn) geeft in het algemeen een verschillende CRC code. Andere CRC-strings die wel worden gebruikt zijn CRC-12: 1000000001011, en CRC-32: 10000010011000001000111011011011 1.

4.2.5 De KIX code van PTT Post

PTT Post gebruikt sinds 1999 een KIX code (KlantIndeX) code op bulkpost, gebaseerd op de Royal Mail 4 State Code. Deze code bestaat uit een rij van lange en korte streepjes:

Er zijn vier verschillende streepjes: lang, kort, halflang naar boven en halflang naar beneden.



POSTBANK

HR J B M MELISSEN
KASTANJELAAN 9
5581 HD WAALRE

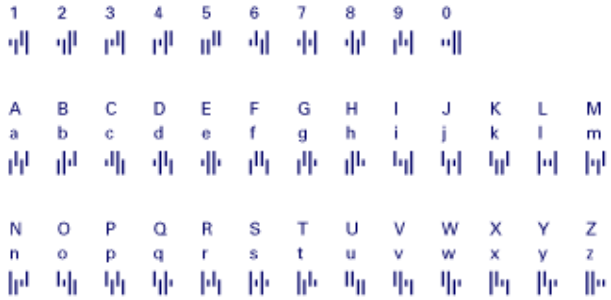
Afrekening GIROrekening

Datum	Girorekening	Bladnr.	Volgnr.
29-02-2000	5402848	1 *	6

Totaal bijgeboekt bedrag — Vorig saldo
Totaal afgeboekt bedrag — Nieuw saldo
1 euro = 2,20371 gulden — Nieuw saldo in euro's

Geboekt op	Naam/omschrijving *laatste blad	Code	nr.	Girorekening	Af/bij	Bedrag
24 FEB	0225203278 ST.HOGER BEROEPSONDERWYS SALARIS FEBRUARI WERKG.NR.30286 STICHTING HOGER BEROEPSO NDERWIJS S HERTOGENBOSCH	VZ	51		BIJ	84,00
25 FEB	KN: 0077143785959 GEFELICITEERD MET UW PRIJS STAATSLOTERIJ 0000924 St Expl Ned Staatsloterij	VZ		6364	BIJ	39.747.840,00

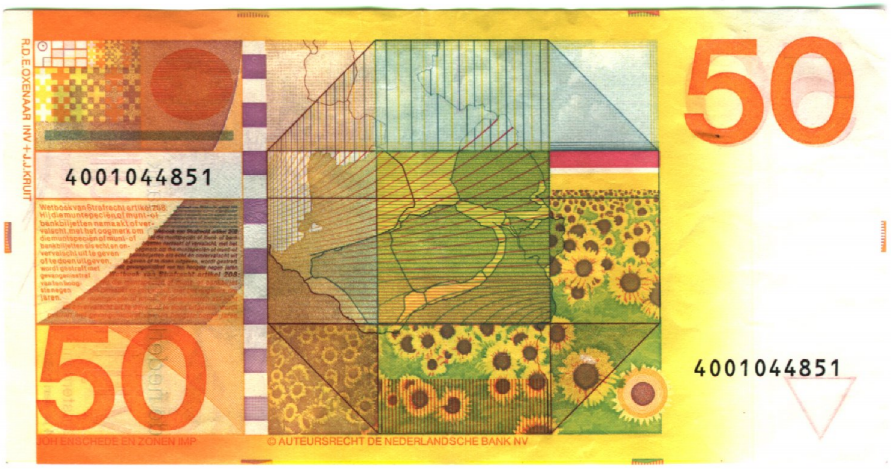
De code werkt als volgt: elk groepje van vier streepjes staat voor een getal of letter. Er wordt geen onderscheid gemaakt tussen hoofd- en kleine letters. De vertaaltabel staat hieronder:



In de code staat de postcode en het huisnummer gecodeerd. Eventuele toevoegingen (c, bis, etc.) kunnen er ook aangeplakt worden, maar dan wordt eerst een scheidingsteken (X) opgenomen, omdat de lengte van het huisnummer variabel is.
Je kunt je afvragen waarom er niet een duidelijkere regelmaat in de code zit. Je zou bijvoorbeeld kunnen beginnen door 1 te coderen als vier kleine streepjes, 2 als drie kleine streepjes plus één half streepje omhoog, etc. Een reden hiervoor is dat de code nu fouten kan corrigeren. Je kunt nagaan dat als je in een willekeurige code één streepje verandert, je niet een bestaande code krijgt. Dat betekent dat je één fout kunt constateren. Je kunt geen fouten corrigeren. Als dat zou kunnen, dan moest elke bol om een code van één fout groot disjunt zijn meet zo'n bol van een andere code. Hoe groot is zo'n bol? Je kunt één streepje op drie manieren veranderen. Dat levert voor elk van de vier posities 12 mogeleijke foute codes. Samen met de juiste code bevat de bol dus 13 codes. Er zijn 26 + 10 = 36 symbolen die je moet kunnen coderen. Als de bollen om elke code disjunct zijn, heb je dus minstens $36 \times 13 = 468$ codes nodig. Dit is meer dan de $4^4 = 256$ mogelijke codes die er in dit systeem zijn, dus volledige foutcorrectie van één fout is niet mogelijk.

4.2.6 Serienummers op bankbiljetten

Op Nederlande bankbiljetten van vóór 1990 (zonder streepjescode) staat een serienummer, waarvan het laatste cijfer een controlegetal is.



Dat controlegetal kun je als volgt uitrekenen: Tel eerst alle cijfers (behalve het laatste) bij elkaar op: 4 + 0 + 0 + 1 + 0 + 4 + 4 + 8 + 5 = 26. Doe dit eventueel nog een keer: 2 + 6 = 8, net zolang tot je één cijfer hebt: 8. Trek dit cijfer van 9 af: 9 – 8 = 1. Dit is het laatste cijfer!

Dit betekent trouwens ook dat het serienummer deelbaar is door 9: 4001044851 / 9 = 4445560539. Dat komt omdat

$x_0 + x_1 \times 10 + x_2 \times 10^2 + x_3 \times 10^3 + x_4 \times 10^4 + \cdots = x_0 + x_1 + x_2 + x_3 + x_4 + \cdots \pmod 9$

Dit betekent dat een getal modulo 9 niet verandert als je z'n cijfers bij elkaar optelt (de zogenaamde negenproef).

De nieuwere bankbiljetten hebben ongetwijfeld ook een controlemechanisme, maar daarover geeft de Nederlandse bank geen uitsluitsel.

4.3 Foutcorrigerende codes

Foutcorrectie is een uitbreiding op foutdetectie. Bitfouten worden hierbij niet alleen gesignaleerd, maar ook meteen gecorrigeerd. Foutcorrectie is nodig om de integriteit en nauwkeurigheid van data te waarborgen (zonder foutcorrectie zouden goedkope harddisks van 2Gb en hoger niet mogelijk zijn), of om fouttolerantie te verhogen, waardoor bepaalde schakels in de informatieketen aan minder stringente specificaties mogen voldoen. Denk bijvoorbeeld aan transmissie over zeer grote afstanden, beschadigingen op een CD en grotere schrijfdichtheid op magnetische schijven (met een grotere foutkans). Door bijvoorbeeld de refreshrate van DRAM's in te verlagen en ze te voorzien van betere foutcorrectie kan het energieverbruik van een notebook worden verlaagd.
Foutcorrectie kan worden gedaan door software, maar voor grote transmissiesnelheden waarbij "*on-the-fly*" moet worden gecorrigeerd, zoals bij telecommunicatie, wordt vaak *special-purpose* hardware gebruikt (digitale logica met embedded software), omdat normale software daarvoor te langzaam is.
Foutcorrectie is altijd "duur" in de zin dat de hoeveelheid verstuurde informatie altijd groter is dan in het originele bericht. In een audio-CD bevat een groep van 588 bits bijvoorbeeld 192 bits aan pure audio-informatie, en 64 check bits.

4.3.1 Voting codes

Eén manier om te zorgen dat fouten gecorrigeerd kunnen worden is, het bericht domweg vaker te verzenden. Het bericht 110 zou je voor alle zekerheid kunnen verzenden als 110**110**. Het probleem met tweemaal verzenden is dat je nog geen fout kunt verbeteren. Als er een bitfout wordt gemaakt kun je na ontvangst alleen maar constateren dát er iets mis is, maar niet wáár de fout heeft plaatsgevonden. Beter is het om het bericht driemaal te verzenden: 110**110**110, want dan kun je namelijk één bitfout corrigeren. Als je bijvoorbeeld 100110110 ontvangt, is duidelijk dat het tweede bit fout is. Je neemt het tripel dat het vaakst voor komt, vandaar de naam ***voting code***. We zullen nu eens analyseren hoe de foutkans verkleind wordt door het driemaal verzenden van een bitstring van drie bits lang. We nemen aan dat de foutkans per bit gelijk is aan p = 0,01 (1%).

Het kan zijn dat je een string van negen bits ontvangt, waarin drie gelijke groepjes van drie voor komen. Je constateert dan dat er geen fouten zijn gemaakt. Goed beschouwd kunnen er echter twee dingen aan de hand zijn:

1. Alle negen bits zijn goed, er is geen fout gemaakt. De kans hierop is $(1-p)^9 = 0{,}91351\ldots$ In dit geval constateer je dat er geen fouten zijn gemaakt, en dat is terecht.
2. In elk groepje van drie is dezelfde fout gemaakt. Dat kan zijn: één fout (kan op drie plekken), twee fouten (kan op drie manieren) of drie fouten. De kans hierop is:

$3(p(1-p)^2)^3 + 3(p^2(1-p))^3 + (p^3)^3 = 0{,}0000028...$ In dit geval constateer je dat er geen fout is gemaakt, maar dat klopt niet! De kans hierop is extreem klein.

Het kan zijn dat je een string van negen symbolen ontvangt, waarin twee gelijke drietallen staan, en één afwijkende. Je constateert dan dat er een fout is gemaakt en dat het tripel dat tweemaal voor komt het juiste moet zijn geweest. Het volgende kan er echter aan de hand zijn:

1. De twee drietallen zijn goed en één is fout (hierin kunnen één, twee of drie bits fout zijn). De kans hierop is $3(1-p)^6(3p(1-p)^2 + 3p^2(1-p) + p^3) = 0{,}08388\ldots$ (de eerste 3 komt omdat er drie mogelijkheden voor het foute tripel zijn). In dit geval is de conclusie juist.
2. In de twee gelijke drietallen is dezelfde fout gemaakt (één, twee of drie bits). Het afwijkende drietal kan goed of fout zijn. De kans hierop is: $3(3(p(1-p)^2)^2 + 3(p^2(1-p))^2 + (p^3)^2)(1-p)^3 = 0{,}000841\ldots$ De uitkomst van de correctieslag is in dit geval niet correct.

Tenslotte kan het zijn dat je een string van negen ontvangt met drie verschillende drietallen. De kans hierop is de overgebleven kans: 1 - 0,91351… - 0,0000028… - 0,08388… - 0,000864… = 0,00174… In dit geval weet je dat er meer dan één fout is gemaakt en je weet niet hoe je het bericht moet corrigeren.

Nu heb je het bericht ontvangen en eventueel gecorrigeerd. De kans dat je dit goed hebt gedaan is: 0,91351… + 0,08388… = 0,99739… De foutkans ná correctie is dus 0,0026, tienmaal zo klein als de kans vóór correctie, want die is voor drie bits $1 - 0{,}99^3 = 0{,}03$. In het algemeen geldt: Als de kans op een bitfout p is (en p is klein), dan zou de foutkans zonder voting code ongeveer gelijk zijn aan 3p, en na correctie ongeveer $27p^2$. De fout wordt dus een factor 9p kleiner.

Laten we een wat realistischer voorbeeld nemen. Stel dat de foutkans 10^{-6} is (gemiddeld één fout op een miljoen bits die worden verstuurd. Als we een tekst van 10^8 bits willen versturen, dan verwacht je dat er 100 bits in de tekst fout zullen gaan. Dat aantal kan onacceptabel hoog zijn. Als we de tekst driemaal sturen, dan is de kans dat een bepaald bit tweemaal fout wordt verstuurd (en dus verkeerd wordt gecorrigeerd) gelijk aan $10^{-6} \times 10^{-6} = 10^{-12}$ (de kans op drie fouten in dezelfde bit is nog veel kleiner: 10^{-18}). Het verwachte aantal plaatsen waarvoor dit in de tekst gebeurt is dus $10^{-12} \times 10^8 = 0{,}0001$. De kans is dus zeer groot dat de tekst helemaal correct is ontvangen (na eventuele correcties).

Om een nog grotere betrouwbaarheid te bereiken kun je natuurlijk nog meer kopieën sturen. Helaas wordt de totale tekst die je moet versturen dan wel erg lang. Er zijn tegenwoordig kortere en meer betrouwbare codes dan *voting codes*.

4.3.2 Pariteitcontrole

Een uitbreiding op het pariteitbit dat we bij foutdetectie zagen is een pariteitcontrole in een rechthoek. Een groepje van 9 bits kun je bijvoorbeeld in een vierkant te zetten. Neem bijvoorbeeld de string 101011100:

1	0	1	**0**
0	1	1	**0**
1	0	0	**1**
0	**1**	**0**	**1**

De pariteitbits krijg je door de bits in een rij of een kolom op te tellen. Het laatste pariteitbit rechtsonder is de som van de pariteitbits in de onderste rij, maar ook de som van de pariteitbits in de rechtse kolom (dat is altijd gelijk, want het is de som van alle informatiebits). De nieuwe gecodeerde rij is nu 101011100**001010**1. Je kunt hiermee constateren of een bitfout is gemaakt, en zelfs wáár die is gemaakt. Een bitfout in de informatiebits geeft twee foute pariteitbits, die de rij en de kolom aanwijzen van de fout. Een bitfout in een pariteitbit geeft

ook een fout in de laatste pariteitbit. Deze code maakt van een string van negen bits een gecodeerde string van zestien bits. Je kunt er één bitfout mee corrigeren, maar twee bitfouten niet.

Een andere manier is als volgt:

0	1	0	0	**1**
1	0	1	**0**	
0	1	**0**		
0	**0**			
1				

10 bits zet je in een driehoek, en je vindt daarbij pariteitbits op de diagonaal door de informatiebits in de bijbehorende kolom én rij bij elkaar op te tellen. Bijvoorbeeld: 1 + 0 + 1 (kolom) + 0 (rij) = **0**. Eén bitfout in een informatiebit geeft twee foute pariteitbits. Deze twee geven de positie van het foute bit aan. Een fout in een pariteitbit geeft maar één fout pariteitbit, dus in dat geval hoef je niet te corrigeren. Dit geeft dus een code met codewoorden van 15 bits die 10 informatiebits en 5 checkbits bevatten. Deze code kan één bitfout corrigeren.
Als de kans op een bitfout 0,0001 is, dan is de kans dat in de 10 informatiebits van een woord geen fout wordt gemaakt $0{,}9999^{10}$, dus de kans op een fout is ongeveer 0,001. Nu gaan we kijken wat de foutkans na foutcorrectie is. We kunnen 0 of 1 fout corrigeren. De kans op nul fouten is $0{,}9999^{15}$. De kans op één fout is 15 x 0,0001 x $0{,}9999^{14}$ (de fout kan op 15 plaatsen staan, de kans op één fout is 0,0001, de kans op de overige veertien goed is $0{,}9999^{14}$). Dit betekent dat de kans op een onjuist correctie is 1 - $0{,}9999^{15}$ - 15 x 0,0001 x $0{,}9999^{14}$ = 0,000001. De foutkans is dus door foutcorrectie een factor 1000 kleiner geworden.

4.3.3 Hammingafstand

Eén manier van foutcorrectie is het gebruik van speciale codewoorden, zoals wij in onze natuurlijke taal doen. We zagen al dat er in een natuurlijke taal zoals het Nederlands een enorme redundantie zit. Vrijwel geen enkel woord dat je met het alfabet kunt maken is een geldig woord. Je kunt foutdetectie doen door elk woord op te zoeken in een woordenboek. Als het er niet in staat is er iets fout gegaan. Als het er wel in staat is het waarschijnlijk goed, maar het kan ook dat er zodanige fouten zijn gemaakt dat het ontvangen woord weer een codewoord is. Je kunt zelfs vaak foutdetectie doen door het dichtstbijzijnde woord te zoeken. Het woord *afsgier* is bijvoorbeeld te corrigeren tot *aasgier*, als je weet dat er maar één fout is gemaakt, maar *aassier* kan komen van *aasgier*, of *aasdier*, of *aasmier*, of *kassier*. Een handige codering zal er dus voor zorgen dat kleine foutjes in een codewoord nooit een ander codewoord opleveren zoals gebeurt in het Nederlands. De taal Nederlands is dus niet echt handig vanuit het oogpunt van foutcorrectie.

Om dit wat te formaliseren zullen we het hebben over **binaire codes**, waarvan de codewoorden bestaan uit bitstrings. De **Hammingafstand** d_H tussen twee codewoorden van dezelfde lengte is het aantal verschillende bits, het aantal bits dat je moet veranderen om van het ene naar het andere woord te komen, of het aantal fouten dat je in de ene bitstring moet maken om de andere te krijgen:

$d_H(101110101,100110111) = 2.$
$d_H(000,111) = 3.$
$d_H(1011101,1011101) = 0.$

Als je fouten wilt kunnen detecteren, dan moet een codewoord waarin je een fout maakt niet opnieuw een codewoord zijn. Als we één fout willen kunnen detecteren is de afstand van een ontvangen woord tot het originele codewoord hoogstens 1 (afstand 0 betekent: goed, afstand 1 betekent: één bit fout). De afstand van zo'n fout woord tot de andere codewoorden moet dan ook minstens 1 zijn, want anders kan een fout woord weer een codewoord zijn en kun je niet constateren dat er een fout is gemaakt. Dit betekent dat de afstand tussen codewoorden minstens gelijk moet zijn aan 2. In het algemeen geldt:

Als er k fouten moeten worden *gedetecteerd* moet de afstand tussen elk tweetal codewoorden minstens k+1 zijn. Omgekeerd, als de afstand tussen elk tweetal codewoorden minstens k+1 is, dan kun je k fouten detecteren.

Als je een code hebt, kun je de afstand tussen alle codewoorden bepalen (het aantal verschillende bits). Als deze afstanden bijvoorbeeld allemaal minstens gelijk zijn aan 5, dan kun je met die code 4 bitfouten per codewoord detecteren.

Bij foutcorrectie willen we niet alleen een fout kunnen constateren, maar deze ook kunnen corrigeren. Dit gebeurt door elk ontvangen woord te vervangen door het dichtstbijzijnde codewoord. Dat is namelijk het woord dat met het minst aantal fouten het foute woord levert. Als we één fout willen kunnen corrigeren, dan heeft een ontvangen woord een afstand 0 of 1 tot het oorspronkelijke codewoord. Dat woord is te corrigeren als er geen ander codewoord op afstand 0 of 1 van dit woord ligt. Dit betekent dat de codewoorden een afstand van minstens 3 tot elkaar moeten hebben.
In het algemeen geldt:

Als er k fouten moeten worden *gecorrigeerd* moet de afstand tussen elk tweetal codewoorden minstens 2k+1 zijn. Omgekeerd, als de afstand tussen elk tweetal codewoorden minstens 2k+1 is, dan kun je k fouten corrigeren.

Als de afstanden tussen de codewoorden bijvoorbeeld allemaal minstens gelijk zijn aan 5, dan kun je met die code 2 bitfouten per codewoord corrigeren.

Voorbeeld: De ruimtesonde Mariner heeft in 1970 foto's van Mars naar de aarde gestuurd, waarbij gebruik werd gemaakt van een code met codewoorden ter lengte 32 met een minimale Hammingafstand van 16. Dat betekent dat deze code 15 fouten kan detecteren (16 = k+1) en 7 fouten kan corrigeren (16 = 2k+1).

4.4 Hammingcodes

Een speciale groep van codes die geschikt zijn om één bitfout te corrigeren is de klasse van de **Hammingcode**s. Zo'n code heeft een aantal codewoorden van een vaste lengte. Voor elke lengte is er een Hammingcode te verzinnen. Dit gaat met behulp van een algoritme dat we aan de hand van een voorbeeld zullen toelichten. We laten zien hoe je een Hammingcode met lengte 5 maakt.
Eerst maak je een tabel waarin op de eerste rij de getallen van 1 t/m 5 (de lengte van de codewoorden) staan. Deze staan in een speciale volgorde. Je begint namelijk met de machten van 2 en vult dit aan met de andere getallen. De volgorde is dus: **1**,**2**,**4**,3,5. In de kolom onder het getal komt de binaire representatie van elk getal te staan (van boven naar beneden):

1	2	4	3	5
0	0	1	0	1
0	1	0	1	0

1	0	0	1	1

Zo heeft 3 de representatie 011. Het aantal rijen in die tabel is drie, omdat we voor de getallen tot en met 5 drie bits nodig hebben voor de binaire representatie. Vanaf lengte 8 hebben we vier of meer rijen nodig. De drie rijen in deze tabel hebben de volgende betekenis: Zij treden op als checkvergelijkingen. Voor een codewoord $x_1x_2x_3x_4x_5$ moet namelijk gelden:

$$0x_1 + 0x_2 + 1x_3 + 0x_4 + 1x_5 = 0,$$
$$0x_1 + 1x_2 + 0x_3 + 1x_4 + 0x_5 = 0,$$
$$1x_1 + 0x_2 + 0x_3 + 1x_4 + 1x_5 = 0.$$

De getallen in een rij van de tabel zijn de coëfficiënten in een van deze checkvergelijkingen en er wordt binair (modulo 2) gerekend (Let op dat de nummering nu weer de normale is, niet de nummering die gebruikt is om de tabel te maken!). In een codewoord van vijf bits zijn in principe vijf bits vrij te kiezen. We hebben nu drie extra vergelijkingen die we opleggen, ter controle. Dat betekent dat we van de vijf vrijheden er maar twee overhouden. De codewoorden gaan nu twee vrije **informatiebits** bevatten en drie **checkbits**. Voor deze checkbits kiezen we de eerste drie: x_1, x_2 en x_3. Deze zijn eenvoudig uit te rekenen uit de informatiebits x_4 en x_5. De drie vergelijkingen hierboven zijn namelijk:

$$x_3 + x_5 = 0,$$
$$x_2 + x_4 = 0,$$
$$x_1 + x_4 + x_5 = 0,$$

ofwel (omdat we binair rekenen is $x = -x$):

$$x_3 = x_5 ,$$
$$x_2 = x_4 ,$$
$$x_1 = x_4 + x_5.$$

Als we de informatiebits x_4 en x_5 weten kunnen we met deze formules de drie check-bits uitrekenen. De informatiebits geven vier mogelijkheden:

c_1 =	0	0	0	0	0
c_2 =	1	0	1	0	1
c_3 =	1	1	0	1	0
c_4 =	0	1	1	1	1
	Check			Info	

Als bijvoorbeeld $x_4 = 1$ en $x_5 = 0$, dan is $x_1 = x_4 + x_5 = 1 + 0 = 1$, $x_2 = x_4 = 1$ en $x_3 = x_5 = 0$.
Je hebt zo alle mogelijke woorden gevonden die aan alledrie de checkvergelijkingen voldoen. De code die we zo krijgen heet de Hamming-[5,2]-code: de codewoorden zijn 5 bits lang en er zijn 2 informatiebits in elk codewoord (en $5 – 2 = 3$ checkbits).
Voor de [5,2]-code kunnen we alle Hammingafstanden tussen de codewoorden uitrekenen (het aantal bits dat verschillend is):

$$d_H(c_1,c_2) = 3,\ d_H(c_1,c_3) = 3,\ d_H(c_1,c_4) = 4,\ d_H(c_2,c_3) = 4,\ d_H(c_2,c_4) = 3,\ d_H(c_3,c_4) = 3.$$

De minimale afstand tussen twee codewoorden is dus gelijk aan 3. Dat betekent dat we volgens de theorie ($k + 1 = 3 \Rightarrow k = 2$) twee foute bits kunnen detecteren en ($2k + 1 = 3 \Rightarrow k = 1$) één foute bit per woord kunnen corrigeren.

Het detecteren van één of twee fouten in een woord gaat automatisch als je de drie checkvergelijkingen uitrekent. Als er niet drie nullen uitkomen is het woord kennelijk geen codewoord en zijn er dus één of meer fouten gemaakt. Als je in een codewoord één fout maakt is de Hammingafstand tot dat codewoord gelijk aan 1, en de afstand tot alle andere codewoorden minstens 2 (omdat alle codewoorden minstens 3 van elkaar af liggen). Je zou dus een woord met één fout kunnen corrigeren door het codewoord te zoeken dat Hammingafstand 1

tot dit woord heeft. Als je pech hebt moet je hiervoor alle codewoorden afzoeken. In ons geval zijn dat er maar acht, maar bijvoorbeeld voor een Hamming-[15,11]-code zijn het er al 2^{11}=2024. Er is een snellere manier om foutcorrectie te doen, en die werkt als volgt. Als je een woord hebt, reken je alle checkvergelijkingen uit. Als er allemaal nullen staan is het woord een codewoord en hoeft er niet gecorrigeerd te worden. Als er niet allemaal nullen staan krijg je een kolommetje met nullen en enen, waaruit je ziet dat er iets fout is. Vervolgens kijk je in de matrix van de checkvergelijking of je de kolom daar terugvindt. Het nummer van de kolom is dan het nummer van de rij waar je een fout hebt gemaakt.
Bijvoorbeeld: in het codewoord 11010 wordt een fout gemaakt in het laatste bit: 11011. De checkvergelijkingen leveren 1, 0 en 1. De kolom met 1 0 en 1 is de laatste kolom in de checkmatrix. Er is dus een fout in de laatste bit gemaakt.
Je kunt zo alleen één bitfout corrigeren. Als er meer fouten zijn gemaakt en je gaat er vanuit dat er maar één fout is gemaakt, ga je ófwel fout corrigeren, of je vindt de kolom niet terug in de checkmatrix.
Bijvoorbeeld: Het codewoord 10101 geeft met twee fouten 11111. De checkvergelijkingen leveren 0,0 en 1. Dit is de eerste kolom in de checkmatrix, dus je zou het eerste bit corrigeren: 01111. Fout!
Bijvoorbeeld: Het codewoord 10101 geeft met twee fouten 10011. De checkvergelijkingen leveren 111. Dit komt niet als kolom voor in de checkmatrix, dus er is meer dan één fout gemaakt. Deze fouten zijn niet te repareren.

5 Literatuur

J.C.A. van der Lubbe, *Information Theory*, Cambridge University Press, 1997.

D. Hankerson, G.A. Harris, and P.D. Johnson*, Information Theory and Data Compression*, CRC Press, 1997, ISBN 0-8493-3985-5.

D. Salomon*, Data Compression, the complete reference*, Springer, 1997, 0-387-98280-9.

M. Nelson and J.-L. Gailly, *The Data Compression Book, 2nd ed.*, M&T Books*,* 1996, ISBN 1-55851-434-1.