KOMPRESJON OG KODING

Et kapittel fra boken

Fritz Albregtsen & Gerhard Skagestein

Digital representasjon
av tekster, tall former, lyd, bilder og video

2. utgave
Unipub 2007

- Med enkelte mindre endringer

Til bruk som del av pensum i kurset INF2310
Institutt for informatikk
Universitetet i Oslo
April 2014

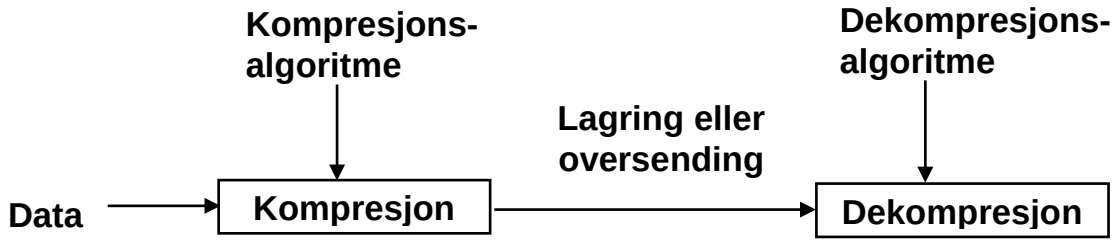
18 Kompresjon og koding

Digital bilder krever mye lagerplass. Et 512 x 512 piksels fargebilde med tre kanaler og 8 biter per kanal krever 786 432 byte. Et digitalt røntgenbilde på 7 112 x 8 636 piksler a 12 biter krever 92 128 848 byte. Heldigvis er lagringskapasitet etter hvert blitt billigere og enklere tilgjengelig. Minnepinner med 1 GB har for lengst erstattet disketter med 1.44 MB, og MP3-spillere med flere titalls GB er etter hvert blitt ganske vanlige. Men det er et økende problem at store mengder digitale bilder, videosekvenser og lydfiler skal overføres pr kabel eller trådløst. Og overføringen tar tid. Selv med en overførings-kapasitet på 1 Mbit/sekund vil det ta ca 12 minutter å overføre røntgenbildet vi nevnte ovenfor.

18.1 Hva er kompresjon?

Kompresjon består i å pakke dataene (tekst, bilde, lydsignal etc.) på en så kompakt måte at overflødig informasjon ikke lagres. Dataene *komprimeres*, og så lagres de. Når de senere skal leses, må vi *dekomprimere* dem (se figur 18-1). De kravene vi stiller til lav kompleksitet og lavt tidsforbruk er ulike i de to prosessene. Det spiller mindre rolle hvor lang tid det tar å komprimere et bilde for lagring sammenlignet med hvor lang tid det tar å hente det fram og dekomprimere det, for når vi trykker på knappen og vil se bildet, så er vi lite villige til å vente. Hvis det er snakk om kompresjon for nær sanntids overføring, så krever vi meget raske algoritmer, både til komprimering og dekomprimering.

Koding er en del av kompresjon, men i denne sammenhengen koder vi for å lagre eller sende effektivt, ikke for å hemmeligholde eller skjule informasjon. For det ***alfabetet*** av ***symboler*** **-** tegn, tall, signalamplituder eller pikselverdier – som vi skal komprimere lager vi oss best mulig koder, slik at hvert symbol har et ***kodeord***. Til sammen utgjør alle kodeordene en ***kodebok***. Og denne kodeboken er ikke hemmelig, for enten sender vi den til mottakeren, eller så vet han/hun hvordan den skal lages. Unicode UTF-8-koden er et av mange eksempler på en slik kodebok.

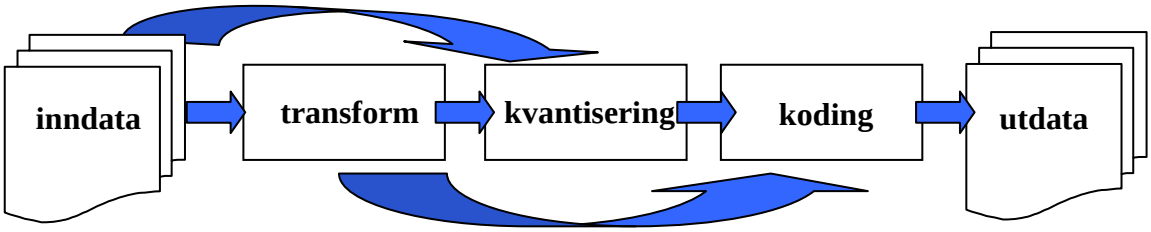


Figur 18-1. Prinsippskisse for kompresjon og dekompresjon.

18.2 Kompresjonsprosessen

Kompresjon kan deles inn i tre steg som følger etter hverandre i en sekvens.

1. ***Transform*** – her representer datasekvensen på en mer kompakt måte.
2. ***Kvantisering*** – her gjøres en avrunding.
3. ***Koding*** – her lages og brukes kodeboken.



Figur 18-2. De tre mulige stegene i kompresjon.

Kompresjonssekvensen kan bestå av alle tre stegene, eller bare 1 og 3, eller 2 og 3. Det er relativt sjelden at dataene foreligger på en slik form at vi velger å gå direkte til steg 3.

Vi kan klassifisere kompresjonsmetodene i

- ***eksakt/tapsfri ("loss-less") kompresjon***
- ***ikke-tapsfri ("lossy") kompresjon***

Ved tapsfri kompresjon kan vi rekonstruere den originale meldingen eksakt.
Dette kan vi ikke ved ikke-tapsfri kompresjon. Resultatet kan likevel være "godt nok".

Det finnes en mengde ulike metoder for begge kategorier kompresjon. Mange av disse er basert på at vi først representerer en tekst, et bilde eller et lydsignal på en annen måte, altså at vi transformer originaldataene (punkt 1 i sekvensen ovenfor). Transformasjonene vi bruker er alltid reversible. Vi kommer tilbake til eksempler som differansetransform og løpelengdetransform.

Hvis vi utfører en kvantisering av originaldataene eller av de transformerte dataene (trinn 2 ovenfor), så kan ikke dette reverseres, og vi får en "lossy" kompresjon.

Det tredje steget i kompresjonssekvensen – koding – bygger ofte på sannsynlighets-fordelinger, det vil si normaliserte histogrammer, som vi beskrev i kapittel 17. Kodingsprosessen er alltid reversibel.

18.3 Melding, data, informasjon – og kapasitet

En melding er den teksten, det bildet eller det lydsignalet som vi skal lagre eller sende. Vi må skille mellom den datamengden som utgjør meldingen, og informasjonen som meldingen inneholder: Dataene er den strømmen av biter som ligger lagret på fil eller sendes, mens informasjonen er et matematisk begrep som kvantifiserer mengden av

overraskelse eller uventethet i meldingen. En melding som ikke inneholder noe nytt, har et lavt informasjonsinnhold. Et signal som varierer, inneholder mer informasjon enn et monotont signal. I et bilde er det kantene rundt objektene – spesielt den delen av kantene med mest krumning – som har høyest informasjonsinnhold. Og utsagnet ”Himmelen er blå” har lavt informasjonsinnhold, mens ”Himmelen er gul” er mer uventet og har høyere informasjonsinnhold.

Filstørrelsen er oftest gitt i binære enheter, gjerne i potenser av 1 024. For å kunne skille potenser av 1 024 fra potenser av 1000, bør vi bruke prefiksene kibibyte (KiB = 2^{10} byte = 1 024 byte), mebibyte (MiB = 2^{20} byte = 1 048 576 byte), gibibyte (GiB = 2^{30} byte = 1 073 741 824 byte) og tebibyte (TiB = 2^{40} byte = 1 099 511 627 776 byte). Som man kan se i appendiks B er det ganske vanlig å bruke SI-symbolene k(eller K), M, G og T om disse toerpotensene, men forskjellen i de faktiske tallverdiene kan jo være betydelig.

En linje kan overføre en bestemt mengde biter i sekundet, og linjens kapasitet uttrykkes derfor som bps (bits per second) Overføringshastigheter og linjekapasitet angis alltid i titallsystemet, og vi bruker SI-prefiksene kB, MB, GB, TB osv når vi mener henholdsvis kilobyte (10^3 byte), megabyte (10^6 byte), gigabyte (10^9 byte) og terabyte (10^{12} byte), se appendiks B. Merk at vi her bruker bokstaven k for 1 000, fordi K i SI-systemet er en temperatur!

1 kbps = 1000 bps = 10^3 biter per sekund.
1 Mbps = 1000 kbps = 10^6 biter per sekund.
1 Gbps = 1000 Mbps = 10^9 biter per sekund.

18.3.1 Redundans

Det er flere måter å overføre en melding på, og vi kan bruke ulike mengder data til å lagre eller overføre samme melding.

- Et gråtonebilde av et 5-tall med 50 x 50 piksler a 8 biter krever 20 000 biter.
- Teksten ”fem” i 8 bit Unicode UTF-8 krever 24 biter.
- Unicode UTF-8 tegnet ”5” krever 8 biter.
- Et binært heltall ”1 0 1” krever 3 biter.

Begrepet redundans sier noe om hvor stor del av datamengden vi kan fjerne, uten at vi mister (relevant) informasjonen i dataene. Et enkelt lite eksempel: Vi skal lagre tallet 0, men hvordan lagres det faktisk?

- Binært i en byte: 00000000 - 8 biter med 0
- Standard 8 biters Unicode UTF-8 kode for 0
- Vet vi at alfabetet utelukkende består av tallene 0 og 1, trenger vi bare 1 bit.

Ved smart komprimering kan vi fjerne redundante biter, og likevel være i stand til å gjenskape den opprinnelige datamengden ved dekomprimering.

Vi kan skille mellom flere ulike typer redundans:

- **Intersampel redundans**
  betyr at flere nabosymboler eller nabopiksler er like.
  - Eksempel: 00001116611 kan ”run-length” transformeres til (0,4),(1,3),(6,2),(1,2). Vi skal straks gå gjennom denne transformen.
- **Psykovisuell/psykoakustisk redundans**
  betyr at det finnes informasjon som vi ikke kan se eller høre.
  - Eksempel: Lyd som er maskert i frekvens eller tid, slik som vi har omtalt i kapittel 9.
- **Interbilde redundans**
  betyr at det er en viss grad av likhet mellom bilder som følger etter hverandre i en sekvens.
  - Eksempel: En video består av en sekvens av stillbilder. Vi kan kode første bilde i sekvensen, og så bare kode endringene fra ett bilde til det neste, slik som vi beskrev i kapittel 15.
- **Kodingsredundans**
  er lik gjennomsnittlig kodelengde minus entropien til sekvensen. (Vi skal redegjøre for entropibegrepet i avsnitt 18.6.1)
  - Et eksempel: 5535552635535 – Vi skal se at Huffman-koding bruker færrest biter på å kode 5-tallet, som forekommer ofte, og flere biter for 6-tallet, som forekommer sjelden i denne sekvensen. Vi skal gå gjennom Huffman-koding og andre koder som bruker ulike lengder på kodeordene.

18.3.2 Kompresjonsrate og redundans

Vi vil lagre en gitt informasjonsmengde ved bruk av færre data. Redundante data må da bort, og vi sitter igjen med en mindre datamengde. Det er flere måter vi kan angi den kompresjonen av datamengden som vi oppnår med forskjellige kompresjons- og kodingsteknikker:

- **Kompresjonsraten** angis som
  $$CR = b/c,$$
  der b er antall biter per sampel originalt, og c er gjennomsnittlig antall biter per sampel i den komprimerte datamengden.
- **Relativ redundans** angis som $$R = 1 - \frac{1}{CR} = 1 - \frac{c}{b}$$
- **”Percentage removed”** = ***100 (1-c/b) %***

Av disse er nok kompresjonsrate den mest brukte angivelsen. Hvis vi starter med et gråtonebilde med 8 biter per piksel, og har oppnådd en kompresjonsrate lik 4, så kan vi også si at vi har komprimert med en faktor 4. Men hvis den vi meddeler oss til vet at dette var et 8 bits gråtonebilde, så vil hun sikkert forstå at kompresjonsraten er 4 hvis vi sier at ”Vi har lagret gråtonebildet med 2 biter per piksel i gjennomsnitt”.

18.4 Noen transformer som brukes i kompresjon

Hensikten med transformene er å representere tekst/bilde/lydsignaler på en annen måte, slik at vi etterpå kan kode dem mer kompakt. Dette kan oppnås ved transformer som utnytter at det finnes redundans i bildet. Vi kan for eksempel unngå å lagre mange tall som er spredt utover et stort tallområde, slik at vi må bruke mange bitposisjoner for hvert tall, men istedenfor lagre flest mulig tall små tall som kan representeres med bare noen få biter.

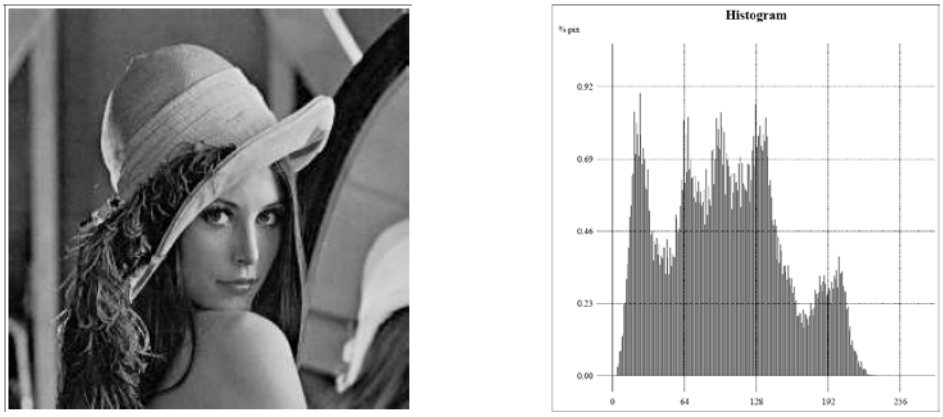
18.4.1 Differansetransform og differansekoder

For hver piksel i et bilde kan vi finne differansen mellom pikselverdien og pikselverdien til nabopikslen til venstre. Gjør vi dette for alle piksler i et bilde, kommer det fram et utbilde som består av alle differensene. Dette kalles en ***differansetransform***. For de aller fleste bilder vil disse differansene ha små tallverdier: de fleste verdiene er nær null, og dette gjør dem lettere å kode kompakt.

Formelt kan vi definere differansetransformen slik: Gitt en horisontal linje i et b biters gråtonebilde med pikselverdier $f_1,.....f_N$ som ligger mellom 0 og 2^b-1. Da kan vi lage en tilsvarende horisontal linje av pikselverdier i et ut-bilde g:

$$g_1 = f_1,\ g_2 = f_2-f_1,\ g_3 = f_3-f_2,.....,\ g_N = f_N-f_{N-1}.$$

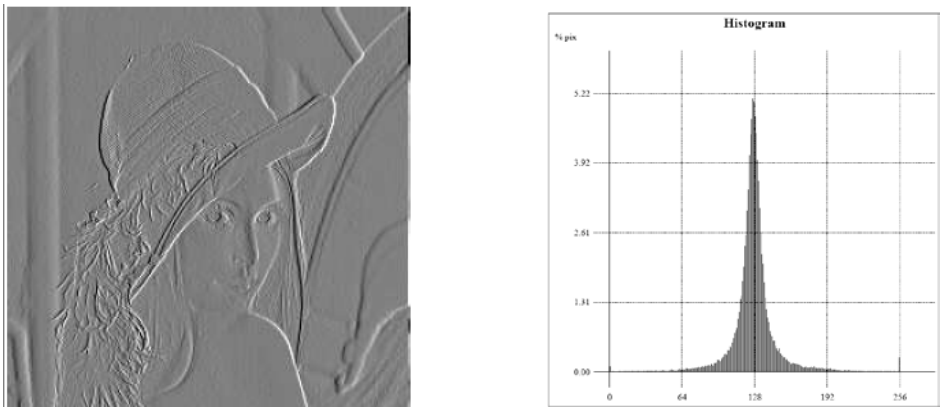
Som eksempel kan vi se på gråtoneportrettet i figur 18-3. Av det tilhørende histogrammet ser vi at mesteparten av den 8-biters gråtoneskalaen er i bruk. Det vil si at bildefilen inneholder et stort spenn av tallverdier, fra 0 til 255.



Figur 18-3. Et 8 biters gråtonebilde og dets histogram

.

Etter en differansetransform blir utbildet og dets histogram som vist i figur 18-4. Her har vi lagt til en bias på 127 i alle pikslene for å unngå negative pikselverdier. Bildet inneholder nå bare første kolonne fra innbildet, resten er som resultatet av en differansedetektor for vertikale kanter. Og vi ser at det tilsvarende *differansehistogrammet* er samlet omkring 127.



Figur 18-4. Det samme bildet og dets histogram etter en linjevis differansetransform.

Differansetransformen er selvsagt reversibel, for vi kan alltid komme tilbake til originalbildet når vi har den første pikselverdien i alle linjene og alle differansene.

Formelt gjelder:

$$f_1 = g_1,\ f_2 = g_2+g_1,\ f_3 = g_3+g_2,\ldots\ldots,\ f_N = g_N+g_{N-1}.$$

Som sagt er differansene i utbildet oftest små, men de kan være både positive og negative, og siden pikselverdiene ligger mellom 0 og 2^b-1, så kan differansen ha 2^{b+1}-1 verdier, mellom -(2^b-1) og 2^b-1.

Vi trenger b+1 bitposisjoner hvis vi skal tilordne like lange binære koder til alle mulig pikselverdier i utbildet fra differansetransformen. Men i differansehistogrammet vil de fleste verdiene samle seg rundt 0, f.eks. mellom -8 og 8. Da kan vi for eksempel lage en 16 ords naturlig kode: c_1 = 0000, c_2 = 0001, ... c_{16} = 1111, og tilordne de 14 kodene i midten, dvs. c_2, ... c_{15} til differansene -7, -6, ..., -1, 0, 1, 2, ..., 5, 6. Kode c_1 og c_{16} kan brukes til å indikere om differansen $\Delta f = f_i - f_{i-1} < -7$ eller om $\Delta f \geq 7$, slik at Δf = 22 kodes med en sekvens av koder: $c_{16}c_{16}c_3$, mens Δf = -22 kodes med sekvensen $c_1c_1c_{15}$, som vist i tabell 18-1. Nettoeffekten av slike ***differansekoder*** er at vi bruker korte koder til de differansene som forekommer hyppig, mens de mindre hyppige differansene får lengre koder. I dette eksemplet har alle de 14 differansene omkring 0 like lange koder (4 biter), og så øker kodens lengde trinnvis med 4 biter for større tallverdi på differansen. Dette er neppe optimalt.

…	-22	-21	…	-8	-7	…	0	…	6	7	…	20	21	…
…	$c_1c_1c_{15}$	c_1c_2	…	c_1c_{15}	c_2	…	c_9	…	c_{15}	$c_{16}c_2$	…	$c_{16}c_{15}$	$c_{16}c_{16}c_2$	…

Tabell 18-1. Kodetabell for en 16-ords differansekode..

Alternativt kan vi kode differansene ved hjelp av andre kodeteknikker som utnytter det faktum at differansehistogrammet har en smal topp ved små differanser. Et eksempel på en slik kodeteknikk er Huffman-koding (se avsnitt 18.7.2), som er direkte basert på histogrammet til bildet.

18.4.2 Løpelengdetransform – ”run-length” koding

Ofte inneholder bildet objekter med lignende gråtoner, for eksempel svarte bokstaver på hvit bakgrunn. Løpelengdetransformen utnytter at nabopiksler på samme linje ofte er like. Den er samtidig et eksempel på at en teknikk blir kalt for en ”koding”, mens det egentlig er en transform. Løpelengdetransform er en enkel, reversibel transform som kan forklares med dette eksemplet. Anta at vi har følgende 3-biters gråtoner etter hverandre på en linje i et bilde: 333333555555555544777777
Når tallet 3 forekommer 6 ganger etter hverandre, trenger vi bare å lagre tallparet (3,6) der det første tallet i parentesen betegner gråtoneverdien, mens det andre betegner ”løpelengden” eller ”run length” – altså hvor mange ganger gråtonen gjentar seg i sammenhengende nabopiksler. Til sammen trenger vi her 4 tallpar, (3,6), (5,10), (4,2), (7,6), altså 8 tall til å lagre hele sekvensen av 24 pikselverdier ovenfor. Hvor mange biter vi bruker per tall, avhenger av den videre kodingen. Vi kan for eksempel bruke Huffman-koding på tallparene.

18.4.2.1 Løpelengdetransform av binære bilder.

Løpelengdetransform av binære bilder er ganske effektiv, fordi vi ikke trenger å kode (pikselverdi, løpelengde) som i gråtonebilder, men kan nøye oss med første pikselverdi (0 eller 1) og deretter angi løpelengden, fordi vi implisitt vet hva neste pikselverdi er når et ”run” er slutt. En slik bit-basert løpelengetransform blir enda mer effektiv dersom vi kan sørge for at kompleksiteten i bitplanet blir mindre. Vi skal nå se hvordan det kan gjøres

18.4.2.2 Bitplan i gråtonebilder

Vi så i kapittel 14.4.5 at et digitalt gråtonebilde der hver piksel er representert med b biter kan deles opp i b bitplan. Hvert bitplan er da et binært bilde med akkurat samme antall piksler som originalbildet. Det i-te bitplan fra et gråtonebilde får vi ved å erstatte hver piksel med det i-te bit i binærkoden til vedkommende pikselverdi.

Bitplanene – fra det mest signifikante til det minst signifikante – finner vi ved å heltalls-dividere alle pikselverdiene i byte-bildet med 128, heltalls-dividere resten med 64, osv. Mens gråtonebildet var lagret i en byte-array, lagres bitplanene i 8 separate bit-arrayer. Et eksempel er de 8 bitplanene i gråtonebildet ”peppers” som vi ser i figur 18.5.

18.4.2.3 Anvendelse av Gray-kode på gråtonebilder

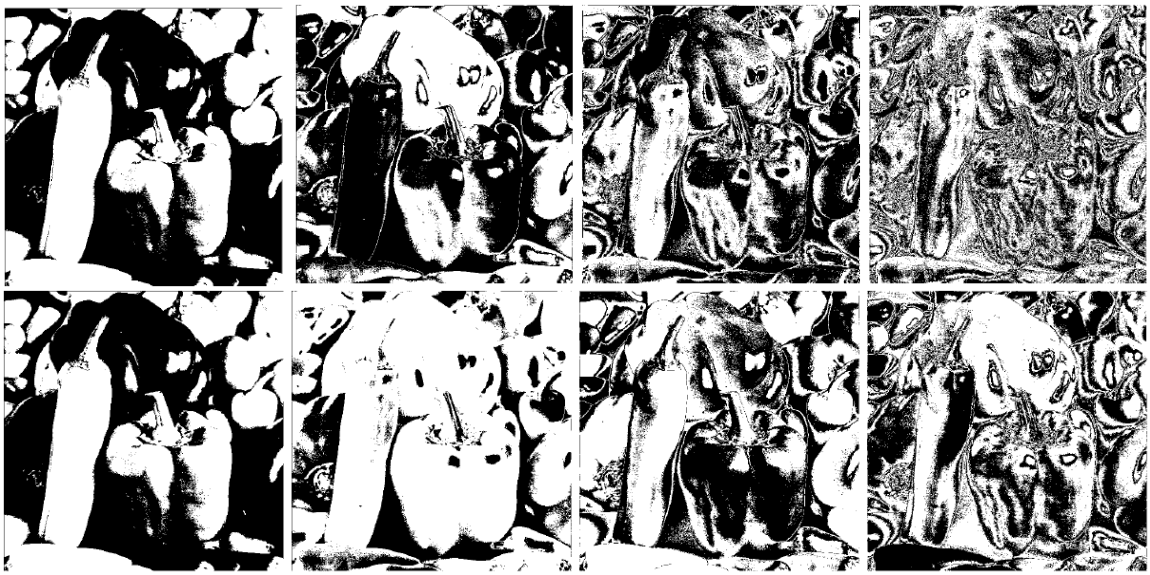
Konvensjonell binær representasjon gir høy kompleksitet i bitplanene. For eksempel er det jo slik at i et område i bildet hvor intensiteten varierer mellom 127 og 128 vil alle bitverdiene skifte: $127_{10} = 01111111_2$, $128_{10} = 10000000_2$.

Det kunne være ønskelig med minst mulig **kompleksitet** i hvert bitplan i bildet – altså at det var store sammenhengende områder i hvert bitplan der den binære pikselverdien ikke skiftet. Da kan vi nemlig effektivt komprimere hvert bitplan separat ved hjelp av binær løpelengdekoding

.
Vi så i kapittel 7.6 at den konvensjonelle binære representasjonen ikke er den eneste måten å representere et tall på. Gray-kode er en binær representasjon der bare én eneste bit endres i representasjonen for suksessive tallverdier. Dette er nettopp det vi trenger for å få mindre kompleksitet i hvert bitplan i et gråtonebilde.

Figur 18-5. Gråtonebildet "peppers".

Testbildet "Peppers" i figur 18-5 har 256 gråtoner, slik at hver pikselverdi er representert med 8 biter. Fra dette bildet kan vi hente ut 8 bitplan. Hvert bitplan blir et binært bilde med det samme antall piksler som originalbildet, og vi får det i-te bitplan ved å erstatte hvert piksel med det i-te bit i binærkoden til vedkommende pikselverdi. Figur 18-6 viser i første rad de 4 mest signifikante bitplanene fra testbildet, fra den mest signifikante biten – bitplan 7 – til venstre, til bitplan 4 til høyre. I neste rad finner vi de tilsvarende bitplanene når pikselverdiene i bildet er kodet med Gray-kode.

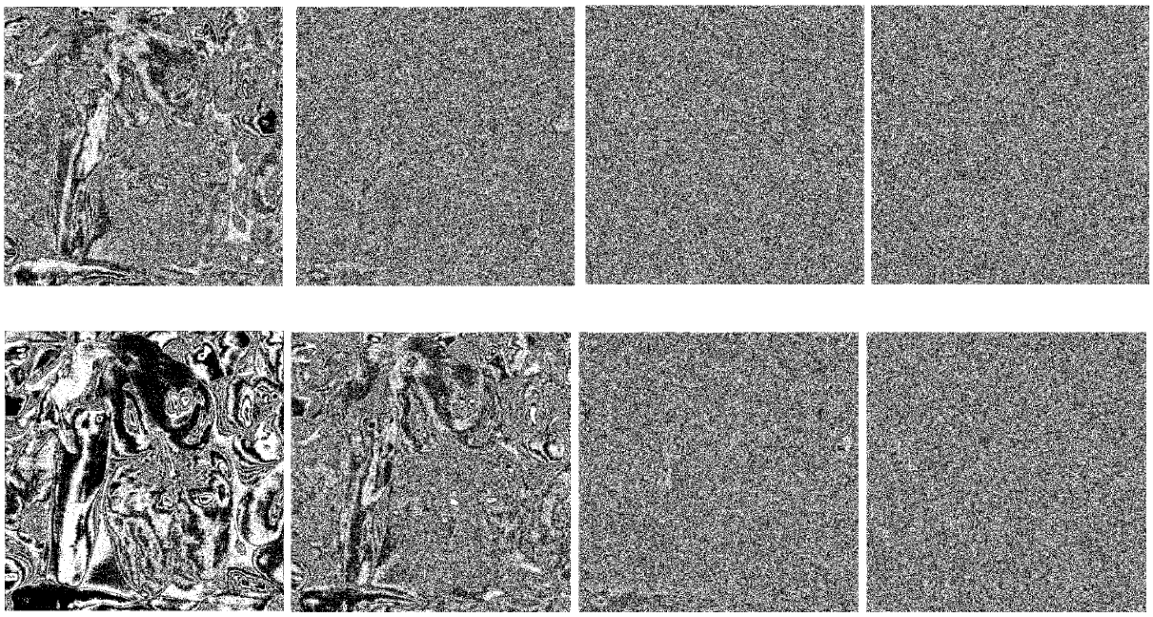


Figur 18-6.Fra venstre mot høyre: De fire mest signifikante bitplanene med naturlig binærkode (øverst) og med Gray-kode (nederst).

Vi ser som forventet at bitplan 7 (mest signifikante bit) er likt i de to representasjonene, men at det ellers er større homogene områder i hvert bitplan i Gray-koding, slik at en løpelengdekoding av hvert av disse bitplanene vil være mer effektiv i Gray kode enn i naturlig binærkode – i hvert fall for disse bitplanene.

I figur 18-7 ser vi i første rad de 4 minst signifikante bitplanene fra testbildet, fra bitplan 3 til venstre til bitplan 0 til høyre. I neste rad finner vi de tilsvarende bitplanene når pikselverdiene i bildet er kodet med Gray-kode.

Vi kan legge merke til at de minst signifikante bitplanene tilsynelatende ikke inneholder noe annet enn tilfeldig støy. Slike bitplan vil være svært vanskelig å komprimere uten tap, men det er færre slike bitplan med Gray-koding enn med naturlig binærkoding.



Figur 18-7. Fra høyre mot venstre: De fire minst signifikante bitplanene med naturlig binærkode (øverst) og med Gray-kode (nederst).

I hvert bitplan kan vi nå gjennomføre en løpelengdetransformasjon. Så kan vi beregne entropien (se avsnitt 18.5.1.2) ut fra et histogram over løpelengdene, for på den måten å finne en grense for hvor mye vi kan komprimere løpelengderepresentasjonen av bitplanene.

For bilder som testbildet ovenfor finner vi da at det selvsagt ikke er noe å spare i bitplan 7 siden dette bitplanet er det samme i Gray-kode som i naturlig binærkode. Det er heller ingen gevinst i de bitplanene som bare inneholder støy. Men i de øvrige bitplanene kan vi høste en gevinst ved å bruke Gray-kode istedenfor naturlig binærkode.

18.5 Litt om informasjonsteori og sannsynlighet

Kompresjon og koding bygger på informasjonsteori og sannsynligheter. Hvis et symbol forekommer ofte, bør vi optimalt sett representere det med et lite antall biter. Hvis et symbol derimot forekommer sjeldent, kan vi tillate oss å bruke mange biter for representasjonen. Et slikt variabelt antall bitposisjoner per symbol vil gi et mindre totalt forbruk av lagerplass enn å bruke like mange biter per symbol slik som for eksempel i ISO 8859-X.

Informasjonsteori bygger på sannsynligheten for at ulike begivenheter skal forekomme. Sannsynligheten for at en begivenhet ***x*** skal skje, skrives som ***p(x)***, og ligger mellom 0 og 1. ***p(x) = 0*** hvis en begivenhet aldri vil skje, og ***p(x) = 1*** hvis ***x*** alltid skjer.

Et lite eksempel: Vi kaster en terning. Alle sidene på terningen har like stor sannsynlighet for å vende opp når terningen kommer til ro, slik at alle tallene fra 1 til 6 har sannsynlighet $1/6 \approx 0.167$. Kaster vi en terning uendelig mange ganger, vil vi få 6 i 16.7% av tilfellene, og tilsvarende for alle de andre tallene. Setter vi opp et normalisert histogram over utfallet av alle kastene blir dette histogrammet nokså flatt, uten noen utpreget topp.

18.5.1 Biter per symbol og entropi

Entropi er et av de fundamentale begrepene i informasjonsteori. Entropi er et matematisk mål på gjennomsnittlig informasjonsmengde i en sekvens av tegn eller tall. Man kan også si at entropi er et mål på uorden eller redundans. Dermed blir entropi-begrepet nært knyttet til kompresjon og koding, for etter at kompresjonen og kodingen er utført vil vi jo helst at det ikke skal være mer redundans i den meldingen, det signalet eller det bildet vi har komprimert. Vi vil at det gjennomsnittlige antall biter vi bruker per symbol skal være lik entropien til meldingen.

Har vi en sekvens av tall eller tegn som kan ha $2^{\mathbf{b}}$ forskjellige verdier og lagres med b biter pr. sampel, så sier vi at vi har et ***alfabet*** med $G = 2^{\mathbf{b}}$ forskjellige mulige ***symboler***.

La oss telle opp antall ganger symbol $s_{\mathbf{i}}$ forekommer i en sekvens av N symboler, og la $n_{\mathbf{i}}$ være dette antallet. Vi gjør dette for alle mulige symboler, dvs. fra i = 0 til $i = 2^{\mathbf{b-1}}$. Dette er det samme som histogrammet til sekvensen. Sannsynligheten til symbolene er da:

$$p_{\mathbf{i}} = n_{\mathbf{i}}/N.$$

Altså det samme som ***det normaliserte histogrammet*** vi beskrev i kapittel 17.

Vi skal se at entropien – som vi ennå ikke har definert matematisk – henger nøye sammen med dette normaliserte histogrammet, eller sannsynlighetsfordelingen til symbolene i alfabetet. Og alfabetet trenger ikke å være bokstaver. Det kan like godt være pikselverdier i et bilde eller amplitudeverdier i et lydsignal.

18.5.1.1 Gjennomsnittlig antall biter per symbol

Anta at vi har laget en kodebok der hvert symbol s_i som forekommer n_i ganger i en sekvens av totalt N symboler har fått et kodeord c_i, der b_i er lengden av kodeordet c_i angitt i biter. Etter at kodingen er ferdig vil alle symbolene av typen s_i ha bidratt med $n_i \cdot b_i$ biter til kodesekvensen. Det totale antall biter vi har brukt for å kode sekvensen kan da skrives som en sum av slike produkter, og for å få et kompakt uttrykk bruker vi summetegnet

$$b_0 n_0 + b_1 n_1 + \ldots + b_{G-1} n_{G-1} = \sum_{i=0}^{G-1} b_i n_i$$

Det gjennomsnittlige antall biter per symbol, c, for denne koden finner vi da ved å dividere med det totale antall symboler, N. Men vi så jo i forrige avsnitt at $n_i / N = p_i$, som er det normaliserte histogrammet for symbolsekvensen:

$$c = \frac{1}{N} \sum_{i=0}^{G-1} b_i n_i = \sum_{i=0}^{G-1} b_i p_i$$

Dette kan kanskje se litt komplisert ut, men det er egentlig ganske enkelt:

- Vi multipliserer lengden b_i av hvert kodeord, angitt i biter, med sannsynligheten p_i for det tilhørende symbolet.
- Så legger vi sammen alle disse produktene.

Istedenfor å skrive dette fullt ut på den tungvinte måten, bruker vi ***summetegnet*** som uttrykker at vi skal summere produktet av b_i og p_i fra $i = 0$ til $i = G-1$.

18.5.1.2 Entropien til en symbolsekvens

Entropien H kan gi oss en nedre grense for hvor mange biter vi gjennomsnittlig trenger per sampel hvis vi lager en kode for hvert symbol.

Vi definer informasjonsinnholdet $I(s_i)$ i hendelsen s_i ved $I(s_i) = \log_2\left(\frac{1}{p_i}\right)$

Vi minner om at $\log_2(x)$ er "2-er logaritmen til x", dvs. at hvis $\log_2(x)=b$ så er $x=2^b$.

Logaritmen med basis 2 til et tall x, her skrevet $\log_2(x)$,
er den potensen vi må opphøye basistallet 2 i for å få tallet x.

Et par eksempler: $\log_2(64)=6$ fordi $64=2^6$, $\log_2(8)=3$ fordi $8=2^3$. Har du ikke $\log_2$ på kalkulatoren, men $\log_{10}$, så er ikke hjelpen langt unna, for $\log_2(x) = \log_{10}(x) / \log_{10}(2)$.

Uttrykket $\boldsymbol{\log_2(1/p_i)}$ gir oss altså informasjonsinnholdet i den hendelsen det er at symbolet $\boldsymbol{s_i}$ forekommer én gang, uttrykt i biter. Hadde vi brukt en annen basis for logaritmen, ville vi fått uttrykt informasjonsinnholdet i et annet tallsystem enn det binære.

Hvis vi tar gjennomsnittet over alle symboler i sekvensen, får vi gjennomsnittlig informasjon pr. symbol. Dette er ***entropien, H*** :

$$H = \sum_{i=0}^{G-1} p_i I(s_i) = -\sum_{i=0}^{G-1} p_i \log_2(p_i)$$

Mer presist er dette ***første ordens entropi***, siden vi baserer oss på et ***første ordens histogram*** – et histogram som bare ser på ett symbol av gangen, ikke på sekvenser av symboler. Entropien setter en nedre grense for hvor kompakt sekvensen av enkeltsymboler kan representeres når vi koder hvert symbol for seg. Den gir oss altså et mål på hvor kompakt representasjonen blir hvis vi lykkes i å fjerne all kodingsredundans. Men fortsatt kan vi ha blant annet ***intersampel redundans***, som vi så i 18.3.1.

18.5.1.3 Øvre og nedre grense for entropi

Vi skal nå vise at det er en veldig enkel øvre og nedre grense for hvor stor entropien til en symbolsekvens, et bilde eller et signal er, og dermed en øvre og nedre grense for hvor mange biter vi i gjennomsnitt trenger for å kode dem slik at vi kan dekode feilfritt, når vi bare koder ett sampel av gangen.

Øvre grense for entropien:

Hvis alle symbol i et gitt alfabet er like sannsynlige – som svarer til at histogrammet til et bilde er helt flatt, alle pikselverdier er like sannsynlige – så er entropien lik antall biter **b** i hvert symbol eller piksel. Det er $G = 2^{\mathbf{b}}$ forskjellige symboler eller pikselverdier – for eksempel 0-255 hvis b = 8, og sannsynligheten for hvert av dem er $p_i = 1/2^{\mathbf{b}}$. Da blir entropien

$$H = -\sum_{i=0}^{G-1}\left[\frac{1}{2^b}\log_2\left(\frac{1}{2^b}\right)\right] = -\log_2\left(\frac{1}{2^b}\right) = \log_2\left(2^b\right) = b$$

Her ser vi at summetegnet forteller oss at vi skal legge sammen det som står inni hakeparentesen $2^{\mathbf{b}}$ ganger. Det som står inni parentesen er konstant, og derfor fører summasjonen bare til at brøken $1/2^{\mathbf{b}}$ forsvinner, og uttrykket blir såre enkelt.
Vi har med dette lært at:

Hvis det er $2^{\mathbf{b}}$ symboler som alle er like sannsynlige, så kan vi ikke representere dem mer kompakt enn med b biter pr symbol.

Nedre grense for entropien:

Hvis alle symbolene eller pikslene er like – som svarer til at det er bare ett symbol eller en pikselverdi som forekommer, så er sannsynligheten for dette symbolet/pikselverdien lik 1, og alle andre sannsynligheter er lik 0. Da blir entropien ganske enkelt

$$H = -\log_2(1) = 0$$

"Bits of data – bits of information"

Anta at vi har et binært bilde med ***M*×*N*** piksler. Ettersom bildet er binært (svart – hvitt) er pikselverdien enten 0 eller 1, altså 1 biter per piksel. Det er lett å finne ut hvor mange biter det er i et slikt bilde – det er ganske enkelt ***M*×*N*** biter. Men hvor mye informasjon er det i bildet?

Hvis det er like mange piksler med verdi 0 som verdi 1 i bildet (og ingen inter-piksel redundans), så er det like stor sannsynlighet for at en piksel har verdien 0 som 1. Informasjonsinnholdet i hver mulig hendelse er da like stort, og entropien til bildet er 1 bit:

$$H = \frac{1}{2}\log_2(\frac{1}{1/2}) + \frac{1}{2}\log_2(\frac{1}{1/2}) = \frac{1}{2}\times 1 + \frac{1}{2}\times 1 = 1$$

Hvis det er 3 ganger så mange 1 som 0 i bildet (og ingen inter-piksel redundans), så er det mindre overraskende å få en 1, og det skjer oftere. Entropien er da mindre:

$$H = \frac{1}{4}\log_2(\frac{1}{1/4}) + \frac{3}{4}\log_2(\frac{1}{3/4}) = \frac{1}{4}\times 2 + \frac{3}{4}\times 0.415 = 0.5 + 0.311 = 0.811$$

I et gråtonebilde med G gråtoner er det på samme måte det normaliserte histogrammet som bestemmer entropien, dvs. det gjennomsnittlige informasjonsinnholdet per piksel, målt i biter:

$$H = -\sum_{i=0}^{G-1} p_i \log_2(p_i)$$

18.6 Naturlig binærkoding

I naturlig binærkoding er alle kodeord like lange. En 3-biters kode gir 8 mulige verdier, som vist i tabell 18-2.

Symbol indeks	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Symbol, s_i	**s_0**	**s_1**	**s_2**	**s_3**	**s_4**	**s_5**	**s_6**	**s_7**
Kode, c_i	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**

Tabell 18-2. Kodetabell for naturlig binærkoding med 3 biters kodeord.

Naturlig binærkoding er bare optimal hvis alle symbolene, pikselverdiene eller de kvantiserte signalamplitudene er like sannsynlige. Dette kravet er svært sjelden oppfylt. Likevel er det slik vi er vant til å kode tekst, tegn og tall. Fordelen er at det er lett å telle seg tegn for tegn framover og bakover i tekster, og tallrepresentasjonenen er lette å regne med, med like lange 7 bits ASCII eller 8 bits ISO 8859-1. Vi slipper også å sende med en kodebok til mottakeren, for koden er en standard som alle kjenner. Men det er altså ikke en optimal måte å lagre eller sende informasjon på hvis vi ønsker å bruke minst mulig plass.

18.7 Koding med variabel lengde

For ulike sannsynligheter er koder med variabel lengde på kode-ordene bedre. Prinsippet er veldig enkelt, og har vært kjent og brukt lenge før informasjonsteoretikerne innførte begrepet entropi: Bruk korte koder for de symbolene som forekommer ofte. På sjeldne symboler kan man koste på seg å bruke lengre kodeord.

Vi ser det for eksempel i de optiske telegrafene som var i bruk på slutten av 1800-tallet: Man hadde koder for hver bokstav og sifrene 0-9. Men med 64 mulige koder kunne man i tillegg ha korte koder for de mest brukte ordene: ”and”, the” etc.

Morse-koden er også et eksempel på dette. Etter å ha eksperimentert med forskjellige løsninger kom Alfred Vail og Samuel Morse omkring 1840 fram til en kodebok som ligner på den vi kjenner som den internasjonale Morse-koden, se tabell 18-3.

Morsekoden består av prikker og streker. Prikker varer 1 enhet, streker 3 enheter, mellomrom mellom prikker og streker varer 1 enhet, mellomrom mellom tegn 3 enheter og mellom ord 5 eller 7 enheter. Så selv om koden tilsynelatende består av prikker og streker, så er den ikke binær, for den inneholder mellomrom. Den er en ***trinær*** kode.

Kodetabellen for den internasjonale Morse-koden viser at den er entropi-basert, for de oftest forekommende bokstavene i engelsk tekst har de korteste kodene.

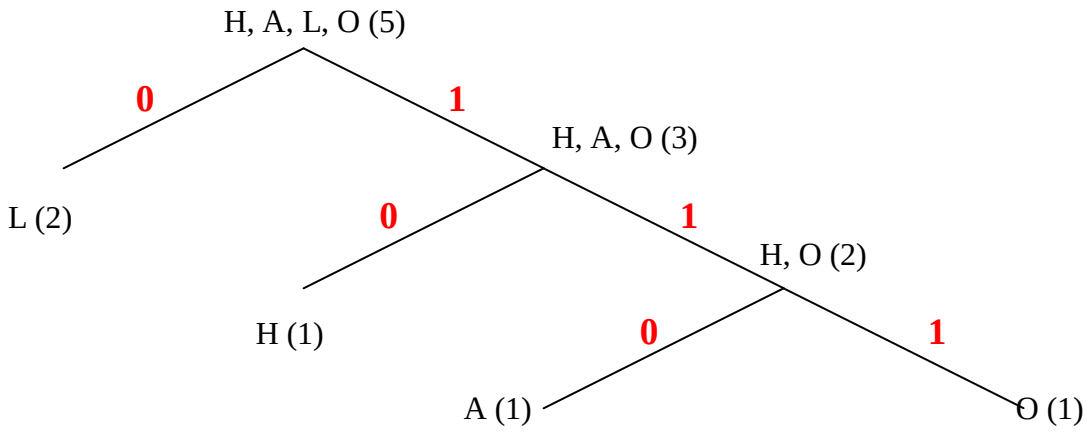
A	.-	I	..	Q	--.-	Y	-.--	6	-....
B	-...	J	.---	R	.-.	Z	--..	7	--...
C	-.-.	K	-.-	S	...	0	-----	8	---..
D	-..	L	.-..	T	-	1	.----	9	----.
E	.	M	--	U	..-	2	..---	.	.-.-.-
F	..-.	N	-.	V	...-	3	...--	,	--..--
G	--.	O	---	W	.--	4	....-	?	..--..
H	....	P	.--.	X	-..-	5	.....	!	-.-.--

Tabell 18-3. En del av kodetabellen for den internasjonale Morse-koden.

18.7.1 Shannon-Fano koding

Dette er en kodingsalgoritme som ble utviklet av Claude Shannon og Robert Fano uavhengig av hverandre. Her trenger man ikke å beregne sannsynlighetene, så man kan benytte det ordinære histogrammet, ikke det normaliserte. La oss ta et kort og lett eksempel: ordet HALLO. Her er det fire forskjellige symboler, med forekomst som vist i andre kolonne i tabell 18-4. Vi starter med å sortere symbolene etter hvor ofte de forekommer, som vist i de to første kolonnene i tabellen nedenfor.

Så deler vi sekvensen av symboler gjentatte ganger (rekursivt) i to omtrent like store deler, inntil hver del bare inneholder ett symbol. En oversiktlig måte å utføre – og illustrere – dette på er å lage et binært tre, som vist i figur 18-8. I hvert steg tilordner man en bit til hver forgrening i det binære treet som oppstår, 1 til høyre, 0 til venstre. Sekvensen av biter fra rota av treet til tuppen av hver forgrening gir da kodeordet for hvert symbol, for eksempel 10 for H, 110 for A, 0 for L og 111 for O. Til sammen bruker vi da 10 biter på de 5 tegnene i ”HALLO”, eller i gjennomsnitt 2 biter per tegn.

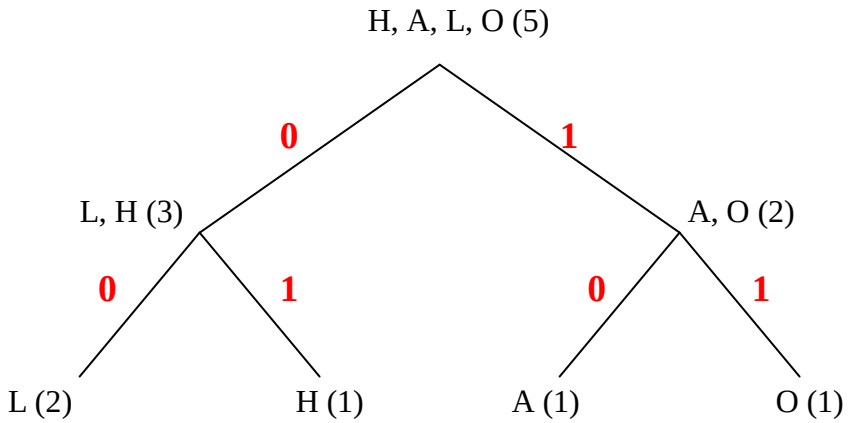


Figur 18-8. Et binært kodetre for Shannon-Fano koding av teksten ”HALLO”.

Symbol	Antall	Kodeord	ordlengde	totalt
L	2	0	1	2
H	1	10	2	2
A	1	110	3	3
O	1	111	3	3
Totalt antall biter				10

Tabell 18-4. Kodetabell for en Shannon-Fano koding av teksten ”HALLO”.

Men den oppdelingen vi gjorde i ”to omtrent like store deler” kunne imidlertid også resultert i et litt annerledes binært tre, som vist i figur 18-9.



Figur 18-9. Et alternativt binært kodetre for Shannon-Fano koding av teksten ”HALLO”.

Og dette ville gitt en annen tabell av kodeord, som vist i tabell 18-5.

Symbol	Antall	$-\log_2(p_i)$	Kodeord	ordlengde	totalt
L	2	1,32	00	2	4
H	1	2,32	01	2	2
A	1	2,32	10	2	2
O	1	2,32	11	2	2
Totalt antall biter					10

Tabell 18-5. Kodetabell for alternativ Shannon-Fano koding av teksten ”HALLO”.

Selv om de to variantene gir forskjellig kodebok, så spiller ikke dette noen rolle for dekodingen av de mottatte kodene. Kodeboken må alltid sendes med, slik at mottakeren av den første kodeboken vil vite at sekvensen 1011000111 betyr ”HALLO”, mens mottakeren av den andre kodeboken vil vite at sekvensen 0110000011 fortsatt betyr ”HALLO”, selv om den ser helt annerledes ut.

Det gjennomsnittlige antall biter pr symbol blir i begge tilfellene ovenfor c=10/5 = 2. Tilfeldigvis gir de to variantene her det samme antall biter, men det kan vi ikke alltid garantere. Den nedre grensen – når vi koder ett og ett symbol - er som vi så i 18.5.1.2 gitt ved første ordens entropi, som her er H = 0.4 × 1.32 + 3×(0.2×2.32) = 1.92. Vi er altså ikke så langt fra entropigrensen.

Shannon viste i sin tid at det gjennomsnittlige antall biter pr symbol fra denne algoritmen ligger mellom H og H+1, altså $H \leq c < H+1$, slik at den øvre grensen for kodingsredundans for denne metoden er 1 bit per symbol.

Shannon-Fano algoritmen er enkel å forstå, lett å implementere, og den gir brukbar kompresjon. Og den illustrerer at man kan kode mer effektivt med kodeord som har ulik lengde. Men den er etter hvert blitt utkonkurrert av Huffman-koding, som i gjennomsnitt koder mer kompakt.

18.7.2 Huffman koding

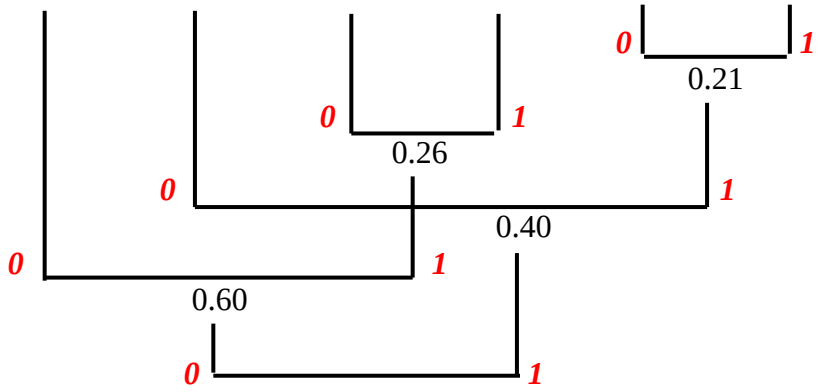
Rett etter at Shannon og Fano hadde klekket ut sin algoritme, fant D.A. Huffman at det var mulig å gjøre dette bedre, og omtrent like enkelt. Huffman-koding er, som Shannon-Fano algoritmen, basert på at vi for hvert symbol kjenner sannsynligheten for at det forekommer, men vi skal se at vi egentlig bare trenger å kjenne antall forekomster av hvert symbol. Likevel bruker vi ofte sannsynlighetene, fordi det er lett å kontrollere at summen av disse er 1. Oppskriften for å kunne utføre en Huffman-koding er slik:

Gitt en sekvens med N symboler:

1. Sorter symbolene etter sannsynlighet, slik at de minst sannsynlige kommer sist.
2. Slå sammen de to minst sannsynlige symbolene i en gruppe, og sorter igjen.
3. Gjenta punkt 2 til det bare er to grupper igjen.
4. Representér det vi har gjort med et binært tre. Traverser treet fra rota til tuppen av hver forgrening, og tilordne én bit til hver forgrening i det binære treet, 1 til høyre, 0 til venstre.
5. Sekvensen av biter fra rota av treet til tuppen av hver forgrening gir kodeordet for hvert symbol.

Et eksempel:
Anta at vi har 6 forskjellige symboler, med sannsynligheter som vist i tabellen nedenfor (dette er de 6 mest sannsynlige enkeltsymbolene i engelsk tekst):
Vi følger oppskriften, og får et binært tre, denne gangen illustrert på en litt annen måte.

Symbol	^	**e**	**t**	**a**	**o**	**i**
sannsynlighet	0.34	0.19	0.14	0.12	0.11	0.10



Dermed får vi følgende kodebok:

Symbol	^	**e**	**t**	**a**	**o**	**i**
Kodeord	00	10	010	011	110	111
antall bit	2	2	3	3	3	3
Entropibidrag	0.529	0.455	0.397	0.367	0.350	0.332

Det gjennomsnittlige antall biter per symbol er gitt ved (se 18.5.1.1):

$$c = b_0 p_0 + b_1 p_1 + \ldots + b_{G-1} p_{G-1} = \sum_{i=0}^{G-1} b_i p_i = 2 \times 0.53 + 3 \times 0.47 = 2.47$$

Mens entropien H er litt mindre enn c:

$$H = -\sum_{i=0}^{G-1} p(s_i) \log_2 (p(s_i)) = 2.431$$

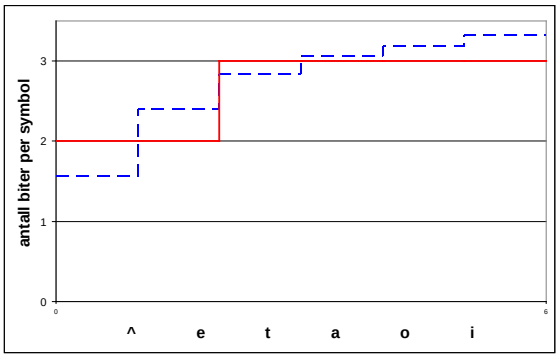
c-H kalles kodingsredundans. Det er faktisk slik at vi helt generelt kan si at **c ≥ H.**

Huffman-koden er optimal i den forstand at den oppnår minst mulig kodingsredundans under antagelsen av at man koder symbol for symbol.
Kodingsredundansen er oppad begrenset til 1 bit per piksel, både for Shannon-Fano og for Huffman, og denne grensen nås når man koder en kilde med to symboler (for eksempel et binært bilde) som har sannsynligheter *p* og *1-p*, og lar *p* gå mot 1.
Når det mest sannsynlige symbolet har en sannsynlighet som er mye mindre enn 1, så kan det vises at kodings-redundansen til Huffman-koden har en øvre grense som er en funksjon av p_{max} , sannsynligheten til det hyppigste symbolet:
$c\text{-}H \leq p_{max} + log_2[2(log_2 e)/e] = p_{max} + 0.086.$
Se for eksempel R.G. Gallagger, "Variations on a Theme by Huffman",
IEEE Transactions on Information Theory, 24(6), 668-674, 1978.

Vi kan oppsummere noen fakta omkring Huffman-koding:

- Ingen kode-ord danner prefiks i en annen kode.
  - Dette sikrer at en sekvens av kodeord kan dekodes entydig, uten at man trenger endemarkører.
  - Det betyr at mottatt kode er instantant (øyeblikkelig) dekodbar.
  - Dette gjelder også for naturlig bitkoding og for Shannon-Fano.
- Hyppigere forekommende symboler har kortere Huffman-koder enn symboler som forekommer mer sjelden.
  - $p_i > p_j \Rightarrow b_i \leq b_j$
  - Dette gjelder også for Shannon-Fano.
- De to minst sannsynlige symbolene har like lange koder, og kodene er bare ulike i siste bit. Dette gjelder også for Shannon-Fano.

Fra entropiberegningen ser vi at den ideelle binære kodeordlengden for symbol s_i er $b_i = -\log_2(p_i)$. Fra eksemplet ovenfor kan vi plotte denne ideelle lengden på kodeordene sammen med den faktiske kodeordlengden, som selvsagt er heltall, se figur 18-10.



Figur 18-10. Ideell og faktisk kodeordlengde

Siden bare heltalls ordlengder er mulig, er det bare i de tilfellene at alle sannsynlighetene kan skrives som en potens av (1/2), altså når

$$p_i = \frac{1}{2^k}$$

for heltalls k, at $-\log_2(p_i)$ blir et heltall, slik at vi får faktiske kodeordlengder som sammenfaller med de ideelle. For eksempel hvis vi har

Symbol	s_0	s_1	s_2	s_3	s_4	s_5
Sannsynlighet	0.5	0.25	0.125	0.0625	0.03125	0.03125
Kodeord	0	10	110	1110	11110	11111

Så blir de ideelle ordlengdene heltall, og gjennomsnittlig ordlengde blir lik entropien, c=1.9375 = H.

En liten oppgave for å illustrere dette, og for å få brukt mye av det man nå skal kunne om entropi, Huffman-koding og toer-logaritmer:

Anta at en telefonsamtale har en båndbredde på 4 kHz, og at det analoge signalet samples i henhold til Nyquist-kriteriet. Anta at det er $G=2^8$ forskjellige nivåer på amplituden til hvert sampel, og at når vi sorterer dem etter hvor ofte de forekommer i en telefon-samtale, så finner vi i dette spesielle tilfellet at sannsynlighetene er ½, ¼, 1/8, 1/16, …, 1/128, 1/256, 1/256.

Q: Hvor mange slike telefonsamtaler kan vi overføre i parallell på en 64 kbits/s ISDN-linje med Huffman-koding av amplitudene?

Hint 1: Entropien er gitt ved

$$H = -\sum_{i=0}^{G-1} p_i \log_2(p_i)$$

Dessuten: log(teller/nevner) = log(teller) – log(nevner)
Og til slutt: $\log_2(2^n) = n$

Hint 2: Summen ½+2/4+3/8 +4/16+5/32 + … konvergerer raskt mot 2.

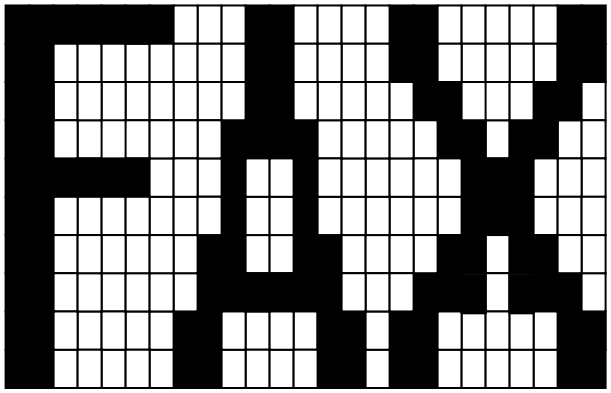
Her er det ikke nødvendig å finne Huffman-koden. Vi har vist at en Huffman-koding der alle sannsynlighetene kan skrives som brøker der telleren er 1 og nevneren er en toer-potens – er optimal i den forstand at det gjennomsnittlige antall bits per sampel er lik entropien til signalet. Samplingsraten må være minst det doble av den høyeste frekvensen som finnes i signalet, dvs 2×4kHz = 8 000 Hz. Entropien er her gitt ved

$H = -(½ \log_2(1/2) + 1/4 \log_2(1/4) + 1/8 \log_2(1/8) + \ldots)$ (se "Hint 1")
$= ½ + 2/4 + 3/8 + \ldots \approx 2$ (som angitt i "Hint 2" ovenfor).

Det gjennomsnittlige antall bits per sampel blir altså bare 2 bits/sampel.
Antall samtaler blir: 64 000 bits/s /(8 000 sampler/s × 2 bits/sampel) = ***4.***

18.7.2.1 Huffman-koding av løpelengder i et binært bilde

Et lite eksempel kan illustrere hvordan Huffman-koding av løpelengder kan brukes i binære bilder, for eksempel de tersklede bildene som sendes fra en fax:
Utsnittet på 25 × 10 piksler av et binært bilde nedenfor kan representeres med 250 biter. Ser vi på løpelengderepresentasjonen av det samme utsnittet, finner vi at det består av 82 "runs" med lengder mellom 1 og 8 piksler. Hvis vi bruker 3 biter på hver, blir dette 246 biter. Imidlertid er det mulig å gjøre dette litt mer kompakt ved å Huffman-kode de 82 løpelengdene. Ved løpelengdetransformasjon av binære bilder trenger vi jo ikke å lagre tallpar (gråtone, løpelengde) slik som for gråtonebilder. Vi trenger bare løpelengdene, for det er bare to mulige intensitetsverdier. Løpelengdene finnes i tabellen til høyre.

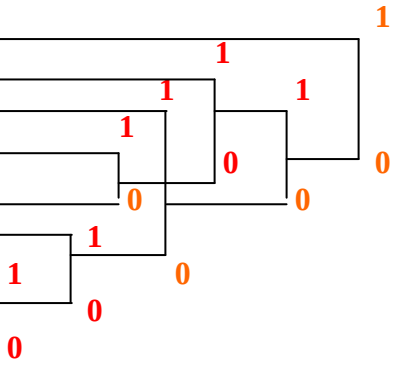


7 3 2 4 2 5 2
2 8 2 4 2 5 2
2 8 2 5 2 3 2 1
2 7 4 5 2 1 2 2
6 3 1 2 1 6 3 3
2 7 1 2 1 6 3 3
2 6 2 2 2 4 2 1 2 2
2 6 6 3 3 1 3 1
2 5 2 4 2 1 2 5 2
2 5 2 4 2 1 2 5 2

Oppgaven blir da å finne Huffmann-koden til løpelengdene i tabellen til høyre over, og finne det totale antall biter etter koding av løpelengdene.

Vi setter opp en tabell over løpelengdene, sortert etter hvor ofte de forekommer, med de minst sannsynlige sist. Vi slår sammen de to minst sannsynlige og sorterer igjen, til vi bare har to grupper igjen. Tabellen nedenfor viser løpelengdene, hyppigheten, kodeordet, lengden på hvert kodeord, og helt til høyre kodetreet.

2	36	1	1
1	11	011	3
3	10	001	3
5	8	0101	4
4	6	0100	4
6	6	0001	4
7	3	00001	5
8	2	00000	5



Og det totale antall biter etter koding blir

36×1+21×3+20×4+5×5 = 36+63+80+25 = **<u>204 biter</u>.**

Vi skal ikke gå inn på detaljene i CCITT-standarden for fax-kompresjon, men den benytter seg av både løpelengdetransformasjon og Huffman-koding. Det brukes en standard (CCITT) Huffmannkode basert på dokument-statistikk, med egen kodebok for svarte og hvite runs. Dessuten er det egne kodeord for ”end of line” (EOL) og ”end of image” (EOI).

18.7.3 Lempel-Ziv-Welch (LZW) algoritmen

Dette er egentlig en hel ”familie” av kompresjonsmetoder, som alle premierer mønstre i dataene, idet man ser på ”samforekomster” av symboler. Det bygges opp en liste av

symbolstrenger både under kompresjon og dekompresjon. En spesiell ting ved disse metodene er at denne listen ikke skal lagres eller sendes, for mottakeren kan bygge opp den samme listen fra de symbolstrengene han mottar. Det eneste man trenger er et standard alfabet (f.eks ISO 8859-X).

Mottaker kjenner bare dette alfabetet, og lager nye fraser ved å ta nest siste streng pluss første symbol i sist tilsendte streng, inntil listen er full (det er en praktisk grense her!). En ulempe er at man av og til lager kodeord som man ikke får bruk for.

Anta at alfabetet er **a**, **b** og **c** som tilordnes kodene 0, 1 og 2.
La dataene være **ababcbababaaaaabab**
Sender bruker regelen: ny frase = sendt streng pluss neste usendte symbol.
Mottakers regel er: ny frase = nest siste streng pluss første symbol i sist tilsendte streng.
Av tabell 18-6 kan vi se at mens Huffman-algoritmen koder enkelt-tegn med koder som har ulik lengde, så kan LZW gi like lange koder for symbolstrenger med variabel lengde.

Avsender ser	Sender	Senders liste	Mottar	Tolker	Mottakers liste
		a=0,b=1,c=2			a=0, b=1, c=2
a	0	ab=3	0	a	
b	1	ba=4	1	b	ab=3
ab	3	abc=5	3	ab	ba=4
c	2	cb=6	2	c	abc=5
ba	4	bab=7	4	ba	cb=6
bab	7	baba=8	7	bab	bab=7
a	0	aa=9	0	a	baba=8
aa	9	aaa=10	9	aa	aa=9
aa	9	aab=11	9	aa	aaa=10
bab	7		7	bab	aab=11

Tabell 18-6. LZW-koding av symbolsekvensen ” ababcbababaaaaabab” med ”a,b,c” som kjent alfabet.

18.7.4 Aritmetisk koding

Aritmetisk koding er en entropibasert koding, men skiller seg fra Huffman-koding ved at man ikke lager kodeord for hvert tegn i en sekvens, men et langt kodeord for hele sekvensen. Dermed kan man få en mer optimal koding enn med Huffman-koding, fordi man ikke lenger er begrenset til et heltalls antall biter per tegn i sekvensen.

En praktisk implementasjon av aritmetisk koding er ganske innfløkt – og er for øvrig patentbeskyttet på alle bauger og kanter – men prinsippet er veldig enkelt. Så derfor tar vi med en beskrivelse av denne kodingsmetoden i sin aller enkleste form.

Aritmetisk koding bygger på at vi kjenner eller antar at vi kjenner sannsynligheten for hvert tegn i alfabetet vårt (tegn, symboler, amplituder eller pikselverdier). Vi kan plassere disse sannsynlighetene etter hverandre på tallinjen fra 0 til 1, og dermed få et bestemt delintervall mellom 0 og 1 som tilsvarer hvert tegn. Har vi to tegn etter hverandre, så blir det delintervallet som representerer dette tegnparet et delintervall av intervallet til det første tegnet, osv. Et kodeord for en lengre sekvens av tegn vil da beskrive et halvåpent delintervall av det halvåpne enhetsintervallet [0, 1), og kodeordet må inneholde akkurat tilstrekkelig mange biter til å gi en entydig peker til det riktige delintervallet.

Sender og mottaker må ha en felles modell for sannsynlighetene til de tegnene som finnes i alfabetet. Hvis modellen bare er en tilnærming til de virkelige sannsynlighetene vil kompresjonen bli dårligere, men kodingen og dekodingen blir likevel riktig. Dermed kan vi enten ha en fast sannsynlighetsmodell, eller vi kan gjøre et gjennomløp gjennom dataene for å finne de eksakte sannsynlighetene. Vi kan også ha en adaptiv modell, der sannsynlighetene endres underveis. Men dette fordrer at modellen endres samtidig hos sender og mottaker. Vi har gjerne en liste med den kumulative sannsynlighet, fra 0 til 1.

Algoritmen kan beskrives både i prosa og i pseudokode:

1. Vi starter med et gjeldende intervall"(current interval") = [0,1)
   – inneholder 0 men ikke 1.
2. For hvert nytt tegn gjør vi to ting:
   a. Vi deler opp "current interval" i nye delintervaller, ett for hvert mulig tegn, der størrelsen på hvert delintervall er proporsjonal med sannsynligheten for vedkommende tegn.
   b. Vi velger det delintervallet av gjeldende intervall som svarer til det tegnet vi faktisk har truffet på, og gjør dette til vårt nye gjeldende intervall.
3. Til slutt bruker vi så mange biter til å representere det endelige intervallet at det entydig skiller dette fra alle andre mulige intervaller.

Anta at vi har et alfabet {*a, b, c*}. Vi antar at de har sannsynlighetene 0.6, 0.2 og 0.2. Vi representerer a med intervallet [0, 0.6), b med [0.6, 0.8) og c med [0.8, 1). Nå skal vi finne hvilket intervall mellom 0 og 1 som entydig vil representere teksten **acaba**.

- Vi starter med intervallet [0, 1). Første tegn er **a**, som ligger i intervallet [0, 0.6). "Current interval" har nå en bredde på 0.6.
- Neste tegn er c, som ligger i intervallet [0.8, 1) i vår modell for sannsynlighetene. Teksten **ac** må da ligge i intervallet [0+0.6×0.8, 0+0.6×1) = [0.48, 0.6) og det nye intervallet har en bredde på 0.12.
- Neste tegn er **a**, i intervallet [0, 0.6) Teksten **aca** ligger da i intervallet [0.48+0.12×0, 0.48+0.12×0.6) = [0.48, 0.552) med en intervallbredde på 0.072.
- Neste tegn er **b**, i intervallet [0.6, 0.8). Teksten **acab** ligger derfor i intervallet [0.48+0.072×0.6, 0.48+0.072×0.8) = [0.5232, 0.5376) med bredde 0.0144.
- Neste tegn er **a**, i intervallet [0, 0.6). Teksten **acaba** ligger da i intervallet [0.5232+0.0144×0, 0.5232+0.0144×0.6) = [0.5232, 0.53184). Intervallbredden er nå 0.00864, som er lik produktet av symbolsannsynlighetene i teksten, 0.6×0.2×0.6×0.2×0.6.
- I en praktisk implementasjon ville vi nå ta med et eget tegn for End-Of-File (EOF) som ville snevre inn intervallet ytterligere. Dette er sløyfet her.

Hvis vi sender over et tall innenfor dette intervallet, så vil det entydig representere teksten **acaba**, forutsatt at mottakeren har den samme modellen.

Nå sender vi jo ikke desimaltall men et bitmønster. Spørsmålet blir da: Hvor mange biter trenger vi for å gi en entydig representasjon av dette intervallet?

Å konvertere fra det binære tallsystemet til titallsystemet er forholdsvis enkelt når vi bruker posisjonstallsystemet fra kapittel 6, også for negative eksponenter. Et lite eksempel: $0.011_{\mathbf{2}} = 0\times2^{-1} + 1\times2^{-2} + 1\times2^{-3} = 0 + ¼ + 1/8 = 0 + 0.25 + 0.125 = 0.375_{\mathbf{10}}$
Eller vi kan skrive det binære tallet som en brøk: $0.011_{\mathbf{2}} = (11/1000)_{\mathbf{2}} = (3/8)_{\mathbf{10}} = 0.375_{\mathbf{10}}$.

Vi kan skrive desimaltallet D som en veiet sum av negative toerpotenser
$D = d_1\times2^{-1} + d_2\times2^{-2} + d_3\times2^{-3} + \ldots + d_n\times2^{-n} + \ldots$
Rekken av koeffisienter $d_1d_2d_3d_4\ldots$ utgjør da bitmønsteret i den binære representasjonen av desimaltallet. Det er dette vi mener når vi skriver tallet som $D=0.d_1d_2d_3d_4\ldots_{\mathbf{2}}$

For å konvertere et slikt desimaltall i titallsystemet til et binært tall kan vi bruke suksessive multiplikasjoner med 2 for å finne koeffisientene $cd_1d_2d_3d_4\ldots$:

1. Vi multipliserer begge sider av ligningen ovenfor med 2.
   Heltallsdelen av resultatet er da lik d_1
   fordi $2D = d_1 + Q$, der $Q = d_2\times2^{-1} + d_3\times2^{-2} + \ldots + d_n\times2^{-(n-1)} + \ldots$
   Hvis resten er 0 er vi ferdige.
2. Multipliser resten Q med 2. Heltallsdelen av resultatet er neste bit.
3. Hvis resten er 0 er vi ferdige. Ellers går vi til 2.

Vi kan bruke 0.375_{10} som eksempel.
$2\times0.375 = 0.750$ -> $d_1 = 0$, rest =0.750
$2\times0.750 = 1.5$ -> $d_2 = 1$, rest =0.5
$1\times0.5 = 1$ -> $d_3 = 1$, rest =0

For intervallet vårt [0.5232, 0.53184) finner vi

0.5232_{10} = $0.100\ 001\ 010\ 10\ldots_2$
0.53184_{10} = $0.100\ 010\ 000\ 01\ldots_2$

Hvilket bitmønster skal vi representere dette intervallet med?
Tallet $0.100\ 00_2 = 0.5_{10}$ er utenfor intervallet mens $\mathbf{0.100\ 01}_2 = 0.531\ 25_{10}$ er innenfor.
Dette er den korteste binære representasjonen som entydig faller innenfor intervallet.
Teksten **acaba** kan altså representeres digitalt med 10001, dvs. med 5 biter.

Vi trenger på det meste $b = \lfloor -\log_2(p) \rfloor + 2$ biter, der p = bredden på det endelige intervallet, for å skille intervallet fra alle andre intervaller. Vi må dessuten ha en mekanisme for å indikere at filen er slutt, enten en EOF-markør eller en ekstern teller for filens lengde. Uansett hvilken mekanisme vi velger gir dette en svært liten økning i kodelengden.

Stemmer det at dette blir dekodet til **acaba**?

- Mottakeren får sekvensen 10001 og konverterer dette til 0.53125_{10}. Modellen representerer a med intervallet [0,0.6), b med [0.6,0.8) og c med [0.8,1). Altså er første bokstav **a**.
- Intervallet for a starter ved 0.0 og har en bredde på 0.6. Vi dividerer derfor med 0.6 og får 0.885417. Dette svarer til intervallet for **c**.
- Intervallet for c starter ved 0.8 og har en bredde på 0.2. Vi subtraherer 0.8 og multipliserer resultatet med intervallbredden 0.2. Det gir (0.885417-0.8)/0.2 = 0.427083. Dette svarer til intervallet for **a**.
- Intervallet for a starter ved 0.0 og har en bredde på 0.6. Vi dividerer derfor med 0.6 og får 0.427083/0.6=0.711806. Dette svarer til intervallet for **b**.
- Intervallet for b starter ved 0.6 og har en bredde på 0.2. Vi subtraherer derfor 0.6, dividerer med 0.2 og får (0.711806-0.6)/0.2=0.559028. Dette svarer til intervallet for **a**.
- I dette eksemplet kan vi ikke vite at vi nå er ferdige med dekodingen. I praksis avsluttes ofte filer med en EOF. Dette er ikke tatt med her.

Bitsekvensen **10001** blir altså dekodet til teksten **acaba**, som forventet.
Selv om dette eksemplet ser greit ut, så innser vi vel umiddelbart at denne basale implementasjonen av aritmetisk koding har to store problemer. For det første vil den stadige krympingen av ”current interval” kreve aritmetikk med stadig økende presisjon etter hvert som teksten blir lengre. For det andre gir ikke metoden noen output før hele sekvensen av tegn er behandlet.

Den enkleste løsningen på disse to problemene er å sende den mest signifikante biten straks den er entydig kjent, og så doble lengden på "current interval", slik at det bare inneholder den ukjente delen av det endelige intervallet. Det finnes flere praktiske implementasjoner av dette, men de er alle ganske regnetunge, og de aller fleste er belagt med patenter som gjør dem noe kostbare i kommersiell bruk.

18.7.5 Populære variabel-lengde koder

En av de mest brukte formene for koding med variabel lengde i dag er SMS-språket. Dels består det av akronymer (ASAP = As Soon As Possible), dels er det rent fonemisk (7K = sjuk, CUL8R = see you later). Det tenderer nok mot å være entropi-basert i den forstand at fraser som skal "tomles" ofte får korte SMS-forkortelser, men dette er ikke et gjennomført prinsipp.

Man finner faktisk mange av de forkortelsene som i dag brukes i det QLE SMS-språket igjen i gamle Morse-forkortelser – selv om Morse-koding er blitt gammel og avlegs. Så det er ikke så mye nytt under solen!

18.8 Koding med informasjonstap

For å få høye kompresjonsrater, er det ofte nødvendig med ikke-tapsfri kompresjon. Ulempen er at man ikke kan rekonstruere det originale bildet, fordi et informasjonstap har skjedd. Enkle metoder for ikke-tapsfri kompresjon er rekvantisering til færre antall gråtoner, eller resampling til dårligere romlig oppløsning.

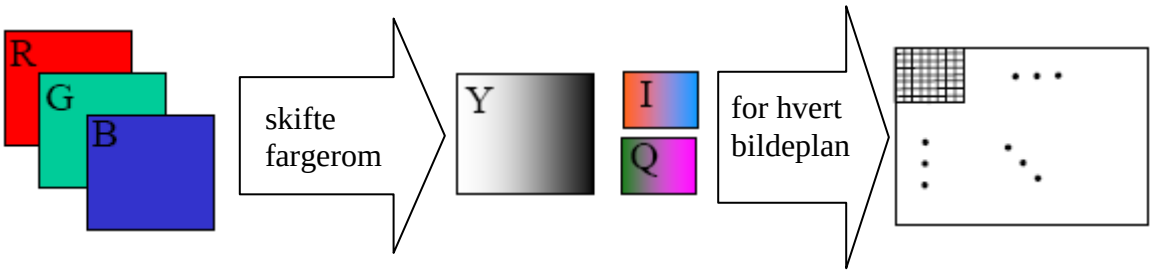
De mest utbredte "lossy" kompresjonsmetodene er betydelig mer avanserte. De viktigste er JPEG for stillbilder, MPEG for video, og MP3 for lydfiler. MP3 ble omtalt i kapittel 11, MPEG i kapittel 16.

18.8.1 JPEG

JPEG er en forkortelse for "Joint Photographic Experts Group", og er samtidig en samling av standarder for kompresjon av stillbilder. Det er flere mulige valg av metoder, og en rekke parametre som kan settes innenfor hver metode. Dette gir stor grad av fleksibilitet for produsenter og brukere innenfor forskjellige anvendelser. JPEG-kompresjon kan brukes både til tapsfri kompresjon – der kompresjonsraten blir relativt beskjeden – og til "lossy" kompresjon – der kompresjonsraten kan bli omtrent 30 med akseptabel reduksjon i kvalitet. Vi skal ikke her gå inn på så mange detaljer i JPEG-kompresjon, men bare beskrive noen steg i en ganske tradisjonell bruk av metoden. Den interesserte leser vil kunne finne mye mer om JPEG på Internett, for eksempel på http://en.wikipedia.org/wiki/JPEG .

Hvis det er et fargebilde vi skal komprimere, så gjøres vanligvis ikke kompresjonen på de like store R, G og B-komponentene. Vi skifter gjerne fargerom slik at vi separerer lysintensitet fra kromatisitet, slik som vi så i overgangen fra RGB til YIQ i kapittel 15 eller til YCbCr i kapittel 16 (se figur 16-1). Da kan vi benytte oss av at vi ikke trenger

den samme geometriske oppløsningen i kromatisitetskomponentene som i intensitetskomponenten. Deretter deles bildet opp i blokker på 8×8 piksler. Vi subtraherer også gjerne 128 fra pikselverdiene i hver blokk, for å få en middelverdi som ligger nær null. Disse stegene er illustrert i figur 18-11.



Figur 18-11. Fra RGB-bilde til 8x8 blokker i hvert bildeplan.

Deretter skal de 8×8 pikselverdiene i hver blokk transformeres med en Diskret Cosinus Transform (DCT). Vi går ikke inn i ligningene for denne transformen her. Den interesserte leser kan finne stoff om dette både i lærebøker om kompresjon og på Internett. Hensikten med denne transformasjonen er at informasjonen om innholdet i de 64 pikslene skal samles i en liten del av de 64 DCT-koeffisientene. Transformkoeffisientene kan sees som en 8×8 matrise, der de høyeste verdiene finnes i øverste venstre hjørne. Vi skal nå se på hva som skjer videre med en blokk på 8×8 piksler. Et eksempel på transformasjon av en slik blokk er vist i figur 18-12.

124	125	122	120	122	119	117	118
121	121	120	119	119	120	120	118
126	124	123	122	121	121	120	120
124	124	125	125	126	125	124	124
127	127	128	129	130	128	127	125
143	142	143	142	140	139	139	139
150	148	152	152	152	152	150	151
156	159	158	155	158	158	157	156

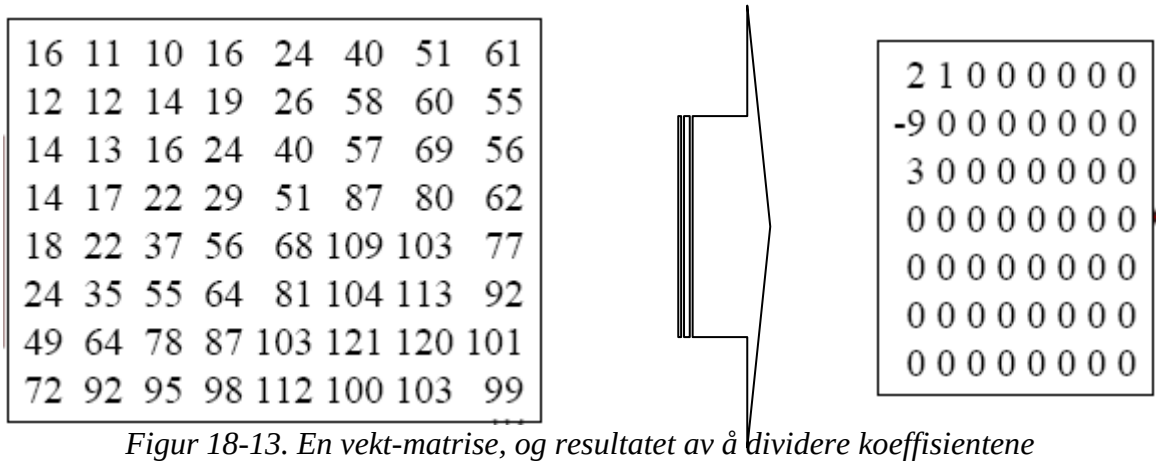
DCT

39.88	6.56	-2.24	1.22	-0.37	-1.08	0.79	1.13
-102.43	4.56	2.26	1.12	0.35	-0.63	-1.05	-0.48
37.77	1.31	1.77	0.25	-1.50	-2.21	-0.10	0.23
-5.67	2.24	-1.32	-0.81	1.41	0.22	-0.13	0.17
-3.37	-0.74	-1.75	0.77	-0.62	-2.65	-1.30	0.76
5.98	-0.13	-0.45	-0.77	1.99	-0.26	1.46	0.00
3.97	5.52	2.39	-0.55	-0.05	-0.84	-0.52	-0.13
-3.43	0.51	-1.07	0.87	0.96	0.09	0.33	0.01

Figur 18-12. DCT-transform av en 8×8 blokk..

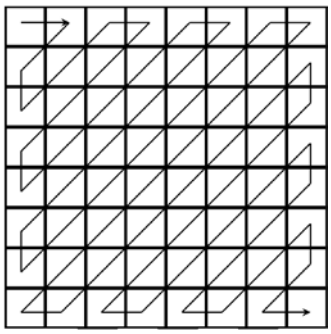
Transformmatrisen til høyre i figur 18-12 blir så dividert med innholdet i en vektmatrise (som kan være forskjellig for forskjellige typer bilder). I slike vektmatriser stiger tallverdiene fra forholdsvis små heltall i øvre venstre hjørne til større heltall i nedre høyre hjørne. Effekten er en skalering som gjør at vi legger mest vekt på de storstilte

variasjonene innenfor bildeblokken og demper betydningen av detaljene. Etter divisjonen rundes svarene av til heltall. Dette er en kvantisering som ikke er reversibel. Vekt-matrisen og resultatet av å anvende den og runde av svarene er illustrert i figur 18-13.



Figur 18-13. En vekt-matrise, og resultatet av å dividere koeffisientene til høyre i figur 18-12 med denne og så runde av til heltall.

Etter denne kvantiseringen vil mange av koeffisientene være lik 0. Ved å sikk-sakk skanne koeffisientene, som vist i figur 18-14, får vi ordnet dem i en sekvens i et endimensjonalt rom. Koeffisientene vil da stort sett avta i verdi utover i rekka, og store deler av tallverdiene i rekka vil være 0.

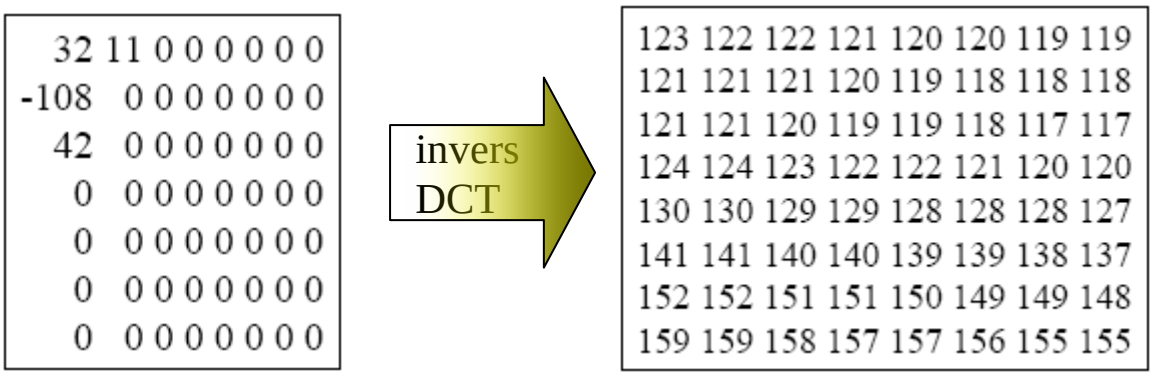


Figur 18-14. Sikk-sakk-skanning av de kvantiserte DCT-koeffisientene.

Dermed ligger det til rette for en løpelengdetransformasjon av de skalerte og kvantiserte koeffisientene, og en Huffman-koding av løpelengdene. Huffman-koden og den tilhørende kodeboken sendes til mottakeren. Etter dekoding av kodeordene og reversering av løpelengdetransformen sitter mottakeren derfor igjen med den samme tallrekken som vi hadde etter at vi hadde brukt sikk-sakk-skanningen i figur 18-14 på resultatet til høyre i figur 18-13. Og denne sikk-sakk'en er selvsagt reversibel, slik at vi faktisk er garantert å komme tilbake til eksakt de samme tallene vi hadde til høyre i figur 18-13.

Hvis vi nå tar de 8×8 verdiene til høyre i figur 18-13 og multipliserer dem med vektmatrisen til venstre i samme figur, så får vi tallene til venstre i figur 18-15. Dette er

ikke helt likt de DCT-koeffisientene vi hadde til høyre i figur 18-12, men de store trekkene er bevart: Vi finner de største tallene i øvre venstre hjørne, selv om tallverdiene er litt endret. Og veldig mange av de små tallene er blitt til 0 -det er faktisk bare 4 av de 64 tallene som er forskjellig fra 0. Deretter gjør vi en invers DCT, og får rekonstruert vår 8×8 piksel bildeblokk, som vist til høyre i figur 18-15.



Figur 18-15. De dekodede DCT-koeffisientene, og den rekonstruerte bildeblokken.

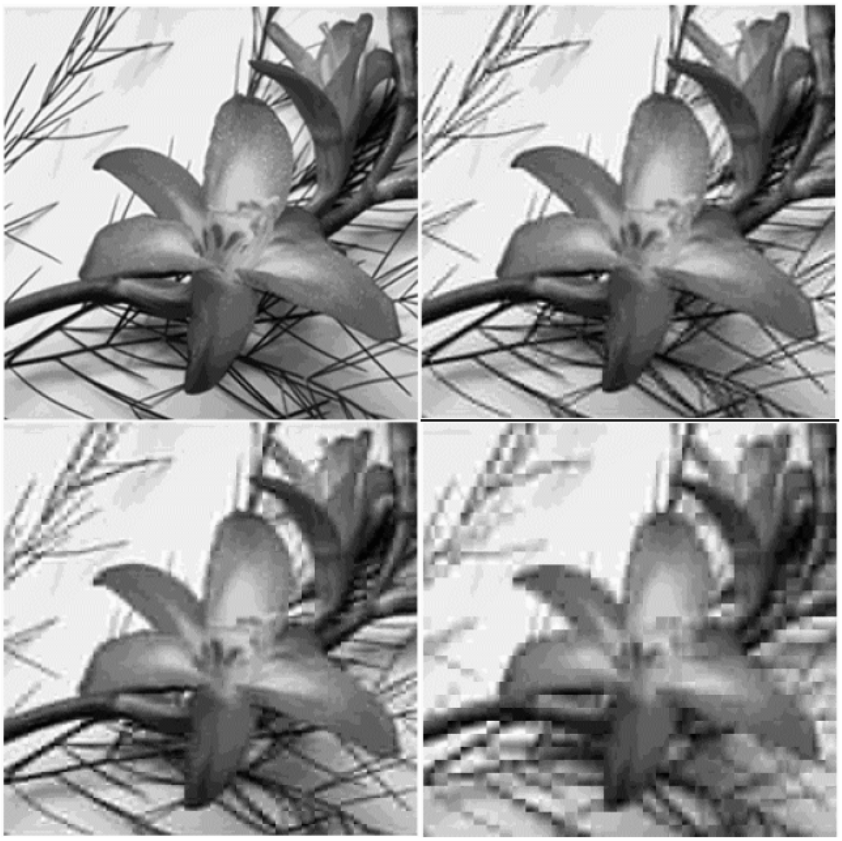
Nå kan vi sammenligne de originale pikselverdiene til venstre i figur 18-12 med de rekonstruerte verdiene til høyre i figur 18-15. Det er jo noen forskjeller, men resultatet er ganske bra, tatt i betraktning at det bare var 4 tall forskjellig fra 0 som egentlig ble tatt vare på.

Det er flere detaljer i en praktisk implementasjon av DCT som vi ikke har tatt med.

- Vi bruker forskjellige vektmatriser for lysintensitet og kromatisitet.
- Vektmatrisene kan skaleres opp og ned ved hjelp av en såkalt kvalitetsfaktor, Q.
- Man kan legge sine egne vekt-matriser inn i en header i bildefilen.

Dessuten er det slik at den første DCT-koeffisienten gir gjennomsnittsverdien innenfor 8×8-blokken. Og siden gjennomsnittet for naboblokker ofte ligner på hverandre, så tar vi ut disse koeffisientene og behandler dem spesielt. Vi gjør en differansetransform på dem, og får en matrise med små tall. Disse kodes så reversibelt med Huffman- eller aritmetisk koding.

Ved å velge hvor mange koeffisienter vi tar vare på, kan vi illustrere den effekten koeffisientene har på kvaliteten på det rekonstruerte bildet. I figur 18-16 ser vi øverst til venstre et originalt gråtonebilde, og så resultatet av å beholde bare 16, 8 eller 4 av de 64 DCT-koeffisientene.



Figur 18-16. Effekten av å redusere antall DCT-koeffisienter.

For RGB fargebilder med 24 biter per piksel (bpp) vil JPEG med 1.5-2 bpp gi så pass god kvalitet at man ikke kan skille det rekonstruerte bildet fra originalen. Selv 0.5-0.75 bpp gir god eller meget god kvalitet, mens 0.25-0.5 bpp gir moderat kvalitet. Eksempler på dette er vist i figur 18-17.

Ser man på gråtonebilder med 8 biter per piksel, vil JPEG med 0.5 bpp gi meget god kvalitet, mens man begynner å se blokk-effekter ved 0.25 bpp, se figur 18-18.

Figur 18-17. Tre JPEG-utgaver av samme bilde. Til venstre et bilde med god kvalitet (Q100), filstørrelse 36 kB. I midten middels kvalitet (Q50), filstørrelse 5.7 kB. Til høyre lav kvalitet (Q10), filstørrelse 1.7 kB. (fra http://en.wikipedia.org/wiki/JPEG)

JPEG kompresjon er kjent for å gi ”blokk-artefakter” i bildene, det vil si at man ser feil i rekonstruksjonen av bildene som skyldes oppdelingen av bildet i 8×8 piksels blokker. I tillegg ser man gjerne den såkalte ”Gibbs-effekten”, dvs sløring og dobbelt-konturer der hvor det er skarpe overganger i intensiteten i original-bildet. I den nyere standarden JPEG 2000, som bruker ”wavelets” istedenfor DCT, er disse problemene borte, og man kan generelt oppnå høyere kompresjonsrate med bedre kvalitet, eventuelt bedre kvalitet med samme kompresjonsrate, som vist i figur 18-18.

Figur 18-18. Et 512x512piksels 8 bits gråtonebilde (venstre) rekonstruert etter en 16:1 kompresjon (0.5 bit/piksel) med JPEG (midten) og JPEG 2000 (høyre).

18.8.2 Fraktaler

Fraktal geometri kan gi oss utrolige komplekse figurer, basert på veldig enkle formler eller regler. De fleste har vel sett Mandelbrotmengden (se for eksempel http://mathworld.wolfram.com/MandelbrotSet.html), der man kan zoome inn på fantastiske detaljer. Det spesielle med fraktaler er at de er like på alle skalaer, og de er "selv-like" – de består av transformerte kopier av seg selv. Men slik selv-likhet er også en slags redundans som har ført til ideen om at man kan bruke prinsippene bak fraktal geometri til å komprimere bilder. Vi trenger ikke hele bildet hvis vi kan finne fragmenter av bildet som ligner veldig på transformerte kopier av andre deler av bildet. Dette er en regnetung prosess, og for virkelige bilder kan vi ikke regne med at vi får en tapsfri kompresjon. Men dekompresjonen er mye enklere og raskere. Hvis vi bare har noen få biter av det originale bilder og noen få transformasjoner, så går det veldig raskt å gjenskape en approksimasjon til det originale bildet.

Metoden har ikke fått noen stor utbredelse, først og fremst fordi den ikke har gitt bedre resultater enn JPEG, og fordi kompresjonen er kompleks. Men fraktaler er et godt eksempel på at vi med noen få biter og noen få regler kan generere veldig komplekse bilder, se eksempel i figur 18-19.

Figur 18-19. Et fraktalt landskap (http://www.fractal-landscapes.co.uk/images.html).

18.9 Oppsummering

Hensikten med kompresjon er mer kompakt lagring eller rask oversending av informasjon. I en beskrivelse av kompresjon og koding er det naturlig at kompresjon av bilder behandles spesielt, fordi det generelt er relativt mye redundans i bilder, både mellom piksler på samme linje, og mellom ulike linjer i bildet. Dette gir opphav til linjebasert differanse- og løpelengdetransformasjon.

Vi har også sett litt overfladisk på Diskret Cosinus Transform (DCT), som samler det meste av informasjonen fra en 8×8 piksels blokk i bildet til noen få transformasjons-koeffisienter, slik at DCT-matrisen etter en kvantifisering og sikk-sakk skanning kan løpelengdetransformeres effektivt.

Vi har også demonstrert at løpelengdekoding i et gråtonebildes bitplan blir mer effektiv dersom gråtonene ikke er representert med naturlig binærkoding, men med Gray-kode.

Kompresjon er basert på informasjonsteori, og vi har derfor lagt vekt på en gjennomgang av begrepene sannsynlighet og entropi. Og vi har i noen detalj vist framgangsmåten ved bruk av Shannon-Fano og Huffman-koding av tekst, og Huffman-koding av løpelengder.

Noen begreper er spesielt viktige:

Har vi en sekvens av N tall eller tegn som kan ha $2^{\mathbf{b}}$ forskjellige verdier og lagres med b biter pr. sampel, så sier vi at vi har et ***alfabet*** med $G = 2^{\mathbf{b}}$ forskjellige mulige ***symboler***. Sannsynligheten $p_{\mathbf{i}}$ til symbolene finnes i ***det normaliserte histogrammet.***

For det ***alfabetet*** av ***symboler*** **-** tegn, tall, signalamplituder eller pikselverdier – som vi skal komprimere lager vi oss best mulig koder, slik at hvert symbol har et ***kodeord***. Til sammen utgjør alle kodeordene en ***kodebok***.

Anta at vi har laget en kodebok der hvert symbol $\boldsymbol{s_i}$ har fått et kodeord $\boldsymbol{c_i}$, og der $\boldsymbol{b_i}$ er lengden i biter av kodeordet $\boldsymbol{c_i}$. ***Det gjennomsnittlige antall biter pr. symbol*** for denne koden er da:

$$c = b_0 p_0 + b_1 p_1 + \ldots + b_{G-1} p_{G-1} = \sum_{i=0}^{G-1} b_i p_i$$

I et gråtonebilde, et lydsignal eller en sekvens av symboler med G mulige verdier er det det normaliserte histogrammet som bestemmer ***entropien***, dvs det gjennomsnittlige informasjonsinnholdet pr piksel, målt i bits.

$$H = -\sum_{i=0}^{G-1} p_i \log_2(p_i)$$

c-H kalles ***kodingsredundans***. Vi kan helt generelt si at **$c \geq H$, Det vil si at vi kan ikke komprimere enkelt-symboler mer kompakt enn det entropien tilsier !**

Appendiks B

Prefiks i titallsystemet og i det binære systemet

B.1 Prefiks i titallsystemet

Når vi skal håndtere store tall i titallsystemet bruker vi en-bokstavs symboler som betegner potenser av 1000 som ble vedtatt som en del av SI-standarden allerede i 1960:

k (kilo) =10^3, M (mega) =10^6, G (giga) =10^9, T (tera) =10^{12},
P (peta) =10^{15} , E (exa) =10^{18}, Z (zetta) =10^{21}, Y (yotta) =10^{24}.

Blant de virkelig store tallene finner vi Googol[1] = 10^{100}, og Googolplex = $(10^{10})^{100}$. Men Googol er neppe et tall vi noen gang kommer til å ta i bruk for å beskrive størrelsen på en digital hukommelse, for den må være oppad begrenset av antall partikler som kan brukes til å lagre en bit. Antall partikler i hele universet må vel være en slik grense. Kan vi finne ut omtrent hvor stort det tallet er? Ja selvfølgelig kan vi det! La oss si at det er 10^{12} galakser som hver inneholder 10^{12} stjerner som hver har en masse som Sola, som er omtrent 2×10^{30} kg. Ett kg hydrogen inneholder omtrent 5×10^{27} atomer. Multipliserer vi sammen dette får vi $2\times5\times10^{(12+12+30+27)} = 10^{81}$. Selv om vi har bommet med en faktor 100 både på antall stjerner per galakse og antall galakser, så er antall partikler fortsatt mellom 10^{77} og 10^{85}. Så det ser ut til at universet inneholder mindre enn en Googol partikler.

Et sted midt i mellom her ligger for eksempel størrelsen på Hr. Skrue McDucks formue, angitt til ”umpteen centrifugillion dollars, or just three cubic acres of money” [2].

B. 2 Prefiks i det binære tallsystemet

Når vi angir størrelsen på en digital lyd- eller bildefil bruker vi gjerne binære prefiks. Dette har nok sin historiske bakgrunn i at alle posisjoner til piksler i digitale bilder representeres i det binære tallsystemet i datamaskinen, og at digitale bilder oftest hadde en størrelse som var gitt som potenser av 2, for eksempel 512 x 512 piksler eller 1024 x 1024 piksler. Hvis hver piksel representeres med 1 byte, vil størrelsen på det sistnevnte bildet oftest bli angitt som 1 MB. Men bildets størrelse er jo 1 024 x 1 024 = 1 048 576 byte = 2^{20} byte, som er 4.8 % større enn 10^6 byte. Og forskjellen mellom titalls- og binære prefiks øker jo dess større tall vi snakker om.

Bruken av prefiksene kilo-, mega-, giga osv, og de tilsvarende symbolene k, M, G osv når vi egentlig snakker om potenser av 1 024, altså 2^{10}, 2^{20}, 2^{30} osv kan altså gi opphav til forvirring og misforståelser. Til tross for dette blir binær-prefiksene ofte skrevet og uttalt identisk med SI prefiksene for titallsystemet.

---

[1] Først brukt av den amerikanske matematikeren Edward Kassners niårige nevø Milton Sirotta i 1938. På fransk oftest skrevet Gogol. Har gitt opphav til navnet på en kjent websøker.

[2] Carl Barks, 1952.

Tabellen nedenfor viser verdiene som svarer til anvendelse av prefiksene i det binære tallsystemet med bruk av SI-symboler, hvilket er det vanligste, men ikke standard. Tabellen viser også tilsvarende verdi i det heksadesimale tallsystemet.

Navn	Symbol	Potens av 2 og verdi i titallsystemet	heksadesimalt
kilo	k/K	2^{10} = 1 024	= $16^{2.5}$
mega	M	2^{20} = 1 048 576	= 16^{5}
giga	G	2^{30} = 1 073 741 824	= $16^{7.5}$
tera	T	2^{40} = 1 099 511 627 776	= 16^{10}
peta	P	2^{50} = 1 125 899 906 842 624	= $16^{12.5}$
exa	E	2^{60} = 1 152 921 504 606 846 976	= 16^{15}
zetta	Z	2^{70} = 1 180 591 620 717 411 303 424	= $16^{17.5}$
yotta	Y	2^{80} = 1 208 925 819 614 629 174 706 176	= 16^{20}

En-bokstavs symbolene er identisk med SI-prefiksene, bortsett fra at man bruker ”k” og ”K” litt om hverandre. I SI-systemet er det bare ”k” som står for 1 000, mens ”K” står for Kelvin, og her er det jo ikke absolutt temperatur vi måler. Det har vært foreslått at ”k” kunne stå for 1 000 og ”K” for 1 024, men dette kan ikke utvides til høyere ordens prefiks, og er aldri blitt vanlig akseptert.

Merk at forskjellen mellom verdien av det binære prefikset og prefikset i titallsystemet øker fra 2.4 % for kilo til over 20 % for yotta. Slikt har det faktisk blitt rettssaker av.

I 1999 publiserte IEC en standard som introduserte prefiksene *kibi-*, *mebi-*, *gibi-*, *tebi-*, *pebi-*, *exbi-* for å angi potenser av 1 024. Navnene er satt sammen av de to første bokstavene i SI-prefiksene pluss bokstavene *bi* for "binær". Fra IECs synspunkt er SI-prefiksene bare desimale enheter, og har aldri noen mening som potenser av 1 024. Denne navnekonvensjonen for binære prefiks blir etter hvert mer brukt, og er akseptert som en standard av IEEE.

Navn	Symbol	binært	heksadesimalt
kibi	Ki	2^{10}	0x400
mebi	Mi	2^{20}	0x10 0000
gibi	Gi	2^{30}	0x4000 0000
tebi	Ti	2^{40}	0x100 0000 0000
pebi	Pi	2^{50}	0x4 0000 0000 0000
exbi	Ei	2^{60}	0x1000 0000 0000 0000
zebi	Zi	2^{70}	0x40 0000 0000 0000 0000
yobi	Yi	2^{80}	0x1 0000 0000 0000 0000 0000

B. 3 Noen konvensjoner

Når vi snakker om ”desimale enheter” mener vi her ”potenser av 1000”.
Når vi snakker om ”binære enheter” mener vi ”potenser av 1024”.
Og vi bruker bokstaven B for en 8 bits byte.

- Noen enheter er alltid desimale selv i en digital sammenheng. To eksempler:

    - Hertz, som måler antall svingninger per sekund i et signal og klokkeraten til elektroniske komponenter.
    - biter/s, som angir bitraten.

    Så en 1 GHz prosessor utfører 1,000,000,000 klokkesykler per sekund, og en 128 kbit/s datastrøm inn eller ut av en MP3-spiller skuffer unna 128 000 biter per sekund, og en 1 Mbit/s Internet forbindelse kan transportere 10^9 biter per sekund.

- Størrelsen på elektroniske minner, enten det er RAM, ROM eller flash, er gitt i binære enheter fordi de produseres med toerpotens-størrelse. Store disk-lignende flashminner kan være et unntak.

- Harddisker har gjerne en kapasitet som er gitt i desimale enheter. Dette har historiske årsaker. Det er blitt en konvensjon at for alle enheter som sees på som et lager på linje med en stor disk, angis kapasiteten desimalt.

- På en disk aksesserer man data sektor for sektor. Sektorstørrelsene er nesten alltid toerpotenser, siden de mapper direkte til RAM. Sektorene varierer fra 512 byte på en floppy-disk til 2 048 byte på en DVD. Dette har gitt opphav til en forvirrende hybrid, der en ”megabyte” betyr ett tusen ”kilobytes” som hver inneholder 1024 byte. Så en ”1.44 MB” diskett (som snart er ute av bruk) er verken 1.44×2^{20} byte eller 1.44×10^6 byte, men $1.44 \times 1\,000 \times 1\,024$ bytes (som er ca 1.406 MiB, eller 1.475 MB).
  Noen produsenter av disk-lignende flashminner har fortsatt denne tvilsomme praksisen med å angi lagringskapasitet til mer moderne lagringsmedier i toerpotens multipler av desimale megabyte!

- Kapasiteten til en CD er alltid gitt i binære enheter. En ”700 MB” CD har en nominell kapasitet på 700 MiB.

- Kapasiteten til en DVD er gitt i desimale enheter. En ”4.7 GB” DVD har en nominell kapasitet på 4.38 GiB.

- Båndbredden til en kommunikasjonsbuss i en datamaskin gis i desimale enheter, fordi klokkeraten er gitt desimalt.