Kapittel 2

Digital representasjon

Innhold

2.1 Læringsmål 8
2.2 Bit – byggesteinen 8
2.3 Posisjonssystemer 9
2.4 Konvertering mellom baser 11
2.5 Heksadesimal og oktal 15
2.6 Byte, verdiområde og designvalg 18
2.7 Negative tall 21
2.8 Overflyt 25
2.9 Tekst og tegn 28
2.10 Flyttall og presisjon 33
2.11 Bilde 38
2.12 Lyd 41
2.13 Oppsummering 44
2.14 Videre lesning 45
2.15 Refleksjon 45
2.16 Selv-test 46

Datamaskinen din kan én ting: skille mellom to tilstander. Det er alt. Likevel ringer du bestemoren din med den, hører på musikk og ser bilder av fjell du aldri har vært på.

Stopp litt ved hvor rart det er. Alt dette – farger, lyd, tekst, tall – er gjort om til et spørsmål som bare har to svar. Et spørsmål maskinen stiller milliarder av ganger i sekundet.

Dette kapitlet handler om hvordan. Men først vil jeg gi deg den ene ideen som binder alt sammen. Får du tak i den, henger resten på plass:

Tankevekker

Et bitmønster betyr ingenting i seg selv. Det er tolkningen – avtalen om hva mønsteret skal bety – som gir det mening. Og hver avtale har en grense. Når du treffer den grensen, får du helt konkrete feil i helt virkelige systemer.

Første setning er temaet for første halvdel av kapitlet. Andre setning er temaet for andre halvdel.

Dette er for øvrig den samme tanken som i kapittel 1, sett fra en ny kant. En representasjon er et grensesnitt. Å velge representasjon er å velge abstraksjon. Og som alle abstraksjoner skjuler den noe, og som alle abstraksjoner koster den noe.

2.1 Læringsmål

Når du er ferdig med dette kapitlet, skal du kunne:

- **forklare** hvorfor det samme bitmønsteret kan bety helt ulike ting, og hva som avgjør hvilken tolkning som gjelder
- **konvertere** tall mellom binær, oktal, heksadesimal og desimal, begge veier – og si hvorfor metodene virker
- **regne ut** hvor mange verdier et gitt antall bit rommer, og velge et fornuftig antall bit til en oppgave
- **regne** med negative tall på toerkomplementform, og forklare hvorfor addisjon virker uendret
- **forutsi** når overflyt skjer, hva svaret blir, og hva konsekvensen er
- **forklare** forskjellen på et kodepunkt og en koding, og telle byte i en UTF-8-streng
- **forklare** hvorfor 0,1 ikke finnes eksakt som flyttall, og hva det betyr for kode du skriver
- **vurdere** hva en representasjon av bilde eller lyd koster i plass, og hva komprimering gir og tar

2.2 Bit – byggesteinen

Ordet *bit* kommer fra engelsk *binary digit*, altså binært siffer. Et bit er rett og slett noe som kan være i én av to tilstander.

Vi skriver de to tilstandene som 0 og 1. Men navnene er tilfeldige. Vi kunne like gjerne sagt usant og sant, av og på, lavt og høyt, rødt lys og grønt lys.

Inne i maskinen din er et bit som regel en transistor som enten slipper strøm gjennom eller ikke. Hvordan den er bygget, kommer vi til i kapittel 6. Foreløpig holder det å vite at den er bitteliten, og at det er veldig mange av dem. En vanlig mobiltelefon har rundt hundre milliarder.

Hvorfor akkurat to?

Et spørsmål som ofte kommer: hvorfor nøyer vi oss med to tilstander? Hadde det ikke vært mer effektivt med ti?

Svaret er at to tilstander er *robust*. Tenk deg at du skal rope et tall over en støyete bakgård. Skal du rope et helt siffer fra 0 til 9, må den andre høre nøyaktig hva du sier. Skal du bare rope «ja» eller «nei», kommer beskjeden fram selv om det bråker.

Slik er det inni maskinen også. En krets som skal skille mellom ti spenningsnivåer, må måle presist. En krets som bare skal avgjøre om det er en spenningen (Volt) eller ikke, tåler støy, varme og at komponentene eldes. Vi kommer tilbake til den avveiningen når vi ser på spenningsnivåer og støymargin.

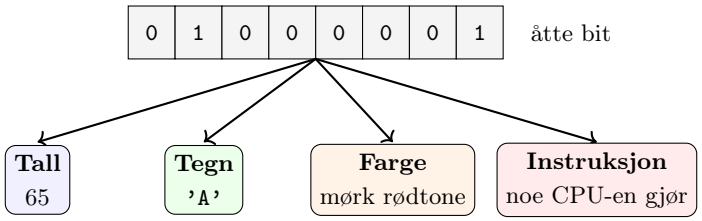
Dette er ikke en naturlov. Det forskes på maskiner som ikke er binære – en kvantedatamaskin har *kubiter*, som kan være i flere tilstander samtidig. Men alle maskiner du kommer til å bruke i dette kurset, og nesten alle som finnes i verden i dag, er binære.

Det samme mønsteret, fire betydninger

Her er åtte bit:

`01000001`

Hvilket tall er dette? Det spørsmålet har ikke ett svar. Det er faktisk ikke engang riktig stilt. Se på figur 2.1.



Bitene er de samme. Det eneste som skiller tolkningene, er hva vi har blitt enige om at de skal bety.

Figur 2.1: Ett bitmønster, fire tolkninger.

Som et positivt heltall er `01000001` tallet 65. I tegnkodingen ASCII er det bokstaven `A`. Som én av tre fargekomponenter i et bilde er det en mørk rødtone. Og ligger byten et sted maskinen er i ferd med å hente instruksjoner fra, er det starten på noe CPU-en vil forsøke å utføre.

Ingen av tolkningene er mer riktig enn de andre. Bitene selv sier ingenting om hvilken som gjelder.

Så hvor står det? Det står i datatypen du skrev i koden din. I formatet filen ble lagret som. I protokollen to maskiner ble enige om før de begynte å snakke sammen. Aldri i bitene.

Dette kommer vi tilbake til i hver eneste seksjon.

Note-to-myself

At et bitmønster kan være både data og instruksjon, er ikke en kuriositet. Det er hele ideen bak maskinen med lagret program, og grunnen til at datamaskinen din kan gjøre noe annet i morgen enn den gjorde i dag. Vi tar tråden opp igjen i kapittel 3.

2.3 Posisjonssystemer

Nå skal vi bli enige om hvordan tall skrives. Det høres opplagt ut. Det er det ikke. De neste sidene handler om ett prinsipp, ikke om fire tallsystemer.

Start med det tallsystemet du allerede kan. Hva betyr egentlig 527?

Det betyr fem grupper à hundre, pluss to grupper à ti, pluss sju enere:

$$527_{10} = 5 \times 10^2 + 2 \times 10^1 + 7 \times 10^0$$

Hvert siffer har en **vekt**. Vekten avhenger av *hvor* sifferet står. Den er basen opphøyd i posisjonen, der posisjonene telles fra høyre og starter på null. Sifferet lengst til høyre står i posisjon 0 og har vekten $10^0 = 1$. Naboen til venstre står i posisjon 1 og har vekten $10^1 = 10$. Og så videre.

Hvorfor virker det slik? Det ligger i måten vi teller på. Du teller 0, 1, 2 og oppover til 9. Da har du brukt opp alle sifrene du har. Så nullstiller du og øker naboen med én. Tierplassen er altså ikke annet enn en opptelling av hvor mange ganger du har vært helt gjennom sifferforrådet.

Det er ikke noe magisk med tallet ti her. Ti er valgt fordi vi har ti fingre. Det er hele grunnen. Prinsippet virker like godt med to siffer, eller åtte, eller seksten.

Note-to-myself

Hvorfor er 10^0 lik 1? Dette spørsmålet kommer alltid, og det er et godt spørsmål. Se på rekka baklengs: $10^3 = 1000$, $10^2 = 100$, $10^1 = 10$. For hvert steg deler du på 10. Fortsetter du ett steg til, får du $10^0 = 1$. Det passer med mønsteret, og det er derfor matematikken har bestemt seg for at det skal være slik. Det samme gjelder alle baser: $2^0 = 1$, $16^0 = 1$, $7^0 = 1$. Du trenger ikke bevise det. Du trenger bare å stole på at enerplassen alltid er enerplassen, uansett hvilket tallsystem du er i.

2.3.1 Telling i binær

I base 2 har vi bare to siffer, 0 og 1. Sifferforrådet er altså tomt allerede etter ett steg. Ellers er alt likt:

0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10

Etter `1` er vi tomme. Da setter vi sifferet til 0 og øker naboen: `10`. Etter `11` er begge tomme, så begge nullstilles og vi tar i bruk en ny posisjon: `100`.

Legg merke til hvor fort tallene blir lange. Det er prisen for å ha få siffer. Vi kommer tilbake til hva vi gjør med det i seksjon 2.5.

2.3.2 Vekt i en hvilken som helst base

Vektene i base 2 er potenser av to: $1, 2, 4, 8, 16, 32, 64, 128$ og videre, fra høyre mot venstre. Generelt gjelder at i base B har sifferet i posisjon k vekten B^k.

Her er en måte å tenke på binære tall som hjelper mange. Se for deg en lommebok med sedler, én av hver valør: 1, 2, 4, 8, 16, 32, 64 og 128 kroner. For hver seddel har du bare to valg – ta den med, eller la den ligge. Et binært tall er ganske enkelt en beskjed om hvilke sedler du tok med.

siffer	1	1	0	0	1
posisjon k	4	3	2	1	0
vekt B^k	2^4	2^3	2^2	2^1	2^0
utregnet	16	8	4	2	1
siffer × vekt	16	8	0	0	1

$$16 + 8 + 0 + 0 + 1 = 25_{10}$$

Figur 2.2: Vektene i det binære tallet 11001_2. Under hvert siffer står posisjonen, vekten og produktet. Summen av produktene er tallets verdi.

Én ting til følger av dette, og vi får bruk for den om litt. Vekten til et siffer er alltid nøyaktig én større enn alt sifrene til høyre kan uttrykke til sammen. Det største tresifrede desimaltallet er 999, og vekten til fjerde siffer er 1000. Det er ingen tilfeldighet.

2.3.3 Notasjon

Skriver du 1001 på en tavle, er det umulig å vite hva du mener. Det kan være det binære tallet, som tilsvarer 9. Det kan være det desimale tallet ett tusen og én.

Vi løser det ved å skrive basen som et lite tall nede til høyre:

- 1001_2 er binært, altså 9_{10}
- 1001_{10} er tusen og én

En liten, men nyttig detalj: *basen selv skrives alltid desimalt*. Toeren i 1001_2 er et vanlig toer. Ellers hadde vi fått et uendelig problem, der vi måtte angi basen til basen til basen.

I programkode brukes andre skrivemåter, og dem kommer vi til i seksjon 2.5: `0b1001` for binær, `0x1001` for heksadesimal, `0o1001` for oktal.

2.4 Konvertering mellom baser

Nå kommer oppskriftene. Det er to retninger, og to metoder i hver retning. Den ene metoden er intuitiv, den andre er mekanisk. Du må kunne begge retninger. Hvilken av de to metodene du velger i hver retning, bestemmer du selv.

2.4.1 Fra en hvilken som helst base til desimal: vektmetoden

Dette har du allerede sett i figur 2.2. Gang hvert siffer med vekten sin, og legg sammen.

Eksempel 1: binær til desimal. Konverter 11001_2.

$$\begin{aligned} 11001_2 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 16 + 8 + 0 + 0 + 1 \\ &= 25_{10} \end{aligned}$$

Eksempel 2: oktal til desimal. Konverter 624_8. Vektene i base 8 er $1, 8, 64, 512$ og videre.

$$\begin{aligned} 624_8 &= 6 \times 8^2 + 2 \times 8^1 + 4 \times 8^0 \\ &= 6 \times 64 + 2 \times 8 + 4 \\ &= 384 + 16 + 4 \\ &= 404_{10} \end{aligned}$$

Eksempel 3: heksadesimal til desimal. Konverter $2A4F_{16}$. Her er A verdien 10 og F verdien 15 – mer om det i neste seksjon. Vektene er $1, 16, 256, 4096$.

$$\begin{aligned} 2A4F_{16} &= 2 \times 16^3 + 10 \times 16^2 + 4 \times 16^1 + 15 \times 16^0 \\ &= 8192 + 2560 + 64 + 15 \\ &= 10\,831_{10} \end{aligned}$$

Eksempel 4: en base som ikke er en toerpotens. Metoden bryr seg ikke om hvilken base du kommer fra. Konverter 142305_6. Vektene i base 6 er $1, 6, 36, 216, 1296, 7776$.

$$\begin{aligned} 142305_6 &= 1 \times 7776 + 4 \times 1296 + 2 \times 216 + 3 \times 36 + 0 \times 6 + 5 \times 1 \\ &= 7776 + 5184 + 432 + 108 + 0 + 5 \\ &= 13\,505_{10} \end{aligned}$$

Det eneste du bytter ut når du skifter base, er basen i vektene. Alt annet står.

Note-to-myself

Ta et overslag først. Det tar fem sekunder og redder deg fra å levere et svar som er dobbelt så stort som det skulle vært.
Det øverste bitet i 110010100_2 har vekten 256. Neste bit er også satt, og legger til 128. Svaret må altså ligge litt over 384. Regner du nøye, får du 404. Gjør dette hver gang.

2.4.2 Fra en hvilken som helst base til desimal: gjentatt multiplikasjon

Den andre metoden går fra venstre mot høyre, og du trenger ingen potenstabell. Start med sifferet lengst til venstre. Gang med basen, legg til neste siffer, gjenta til sifrene er brukt opp.

Eksempel 5. Konverter 13754_8.

$$
\begin{aligned}
1 &\qquad && \text{start med første siffer} \\
1 \times 8 + 3 = 11 & && \\
11 \times 8 + 7 = 95 & && \\
95 \times 8 + 5 = 765 & && \\
765 \times 8 + 4 = 6124 & && \Rightarrow\ 13754_8 = 6124_{10}
\end{aligned}
$$

Hvorfor virker dette? Tell hvor mange ganger hvert siffer blir ganget med 8. Det første blir ganget fire ganger, altså med 8^4. Det andre blir ganget tre ganger, altså 8^3. Det siste blir aldri ganget, og får dermed vekten $8^0 = 1$.

Hvert siffer får altså nøyaktig samme vekt som vektmetoden ville gitt det. Regnestykket er bare delt opp annerledes. Dette er for øvrig metoden maskinen bruker når den skal gjøre tekststrengen `"13754"` om til et tall.

2.4.3 Fra desimal til en hvilken som helst base: største vekt som får plass

Nå den andre veien. Skriv opp vektene i basen du skal til. Finn den største som ikke er større enn tallet ditt, og se hvor mange ganger den får plass. Trekk fra. Gjenta med neste vekt.

Dette er akkurat som å veksle et beløp i sedler: du tar de største først.

Eksempel 6. Konverter 3193_{10} til binær. Vektene vi har å velge mellom er

$$\ldots,\ 4096,\ 2048,\ 1024,\ 512,\ 256,\ 128,\ 64,\ 32,\ 16,\ 8,\ 4,\ 2,\ 1.$$

$$
\begin{aligned}
4096 > 3193 & && \text{for stor, ingen 4096-plass} \\
3193 - 2048 = 1145 & && \text{1 i 2048-plassen} \\
1145 - 1024 = 121 & && \text{1 i 1024-plassen} \\
121 < 512, 256, 128 & && \text{0 i alle tre} \\
121 - 64 = 57 & && \text{1 i 64-plassen} \\
57 - 32 = 25 & && \text{1 i 32-plassen} \\
25 - 16 = 9 & && \text{1 i 16-plassen} \\
9 - 8 = 1 & && \text{1 i 8-plassen} \\
1 < 4, 2 & && \text{0 i begge} \\
1 - 1 = 0 & && \text{1 i 1-plassen}
\end{aligned}
$$

Leser vi av plassene fra 2048 og nedover, får vi $3193_{10} = 110001111001_2$.

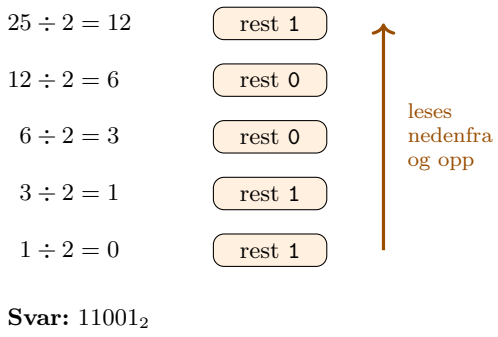
I baser større enn 2 må du i tillegg finne ut *hvor mange ganger* vekten får plass, ikke bare om den får plass. Tar vi samme tall til heksadesimal, med vektene $4096, 256, 16, 1$: 256 får plass 12 ganger, siden $12 \times 256 = 3072$, og resten er 121. Så får 16 plass 7 ganger, og 9 blir stående. Siden 12 skrives C i heksadesimal, er $3193_{10} = \mathrm{C79}_{16}$.

2.4.4 Fra desimal til en hvilken som helst base: divisjonsmetoden

Dette er metoden du kommer til å bruke mest. Den er lett å gjøre helt mekanisk, uten å tenke.

Del tallet på basen. Skriv ned resten. Del kvotienten på basen igjen. Fortsett til kvotienten blir 0. Les restene nedenfra og opp.

Eksempel 7. Konverter 25_{10} til binær. Regnestykket står i figur 2.3, og svaret blir 11001_2 – det samme som vektmetoden ga oss i figur 2.2 on page 11.



Figur 2.3: Divisjonsmetoden for 25_{10} til base 2. Restene leses nedenfra og opp.

Advarsel :)

Nedenfra og opp. Dette er den vanligste feilen på denne oppgavetypen, og den koster poeng hver eneste gang.
Regnestykket over gir restene 1, 0, 0, 1, 1 i den rekkefølgen du skrev dem. Svaret er *ikke* 10011_2. Det er 11001_2.
Grunnen kommer under: den første resten du finner, er enerplassen. Den hører hjemme lengst til høyre i svaret.

Eksempel 8: en større base. Konverter 8151_{10} til heksadesimal.

$$\begin{aligned} 8151 \div 16 &= 509 \quad \text{rest } 7 \\ 509 \div 16 &= 31 \quad \text{rest } 13 = \text{D} \\ 31 \div 16 &= 1 \quad \text{rest } 15 = \text{F} \\ 1 \div 16 &= 0 \quad \text{rest } 1 \end{aligned}$$

Nedenfra og opp: $8151_{10} = 1\text{FD}7_{16}$.

Eksempel 9: en base uten noe med to å gjøre. Konverter 6124_{10} til base 5.

$$
\begin{aligned}
6124 \div 5 &= 1224 \quad \text{rest } 4 \\
1224 \div 5 &= 244 \quad \text{rest } 4 \\
244 \div 5 &= 48 \quad \text{rest } 4 \\
48 \div 5 &= 9 \quad \text{rest } 3 \\
9 \div 5 &= 1 \quad \text{rest } 4 \\
1 \div 5 &= 0 \quad \text{rest } 1
\end{aligned}
$$

Nedenfra og opp: $6124_{10} = 143444_5$. Sjekk gjerne etter med vektmetoden.

2.4.5 Hvorfor divisjonsmetoden virker

Når du deler et tall på basen B, må resten ligge mellom 0 og $B-1$. Det er nøyaktig det samme området som ett siffer i base B dekker. Restene *kan* altså ikke bli noe annet enn gyldige siffer. Det er ingen tilfeldighet – det er hele trikset.

Den første divisjonen spør: hvor mange hele grupper à B får plass i tallet? Kvotienten er antall grupper. Resten er hvor mange enkeltenheter som blir til overs når gruppene er trukket fra. Og de enkeltenhetene *er* enerplassen.

Etter første divisjon er tallet du sitter igjen med allerede delt på B én gang. Neste divisjon spør derfor egentlig: hvor mange grupper à B^2 får plass? Resten denne gangen er hvor mange B-grupper som ble til overs. Det er nest siste siffer.

Slik fortsetter det. For hver divisjon blir gruppene B ganger større. Når kvotienten når null, er det ingenting igjen å gruppere. Og siden det høyre sifferet ble funnet først, må restene leses nedenfra og opp.

Fun Fact

Divisjonsmetoden bruker *modulo*, operasjonen som gir resten av en heltallsdivisjon. I C, Java og Python skrives den `%`. Uttrykket `25 % 2` gir 1.
Skal du skrive et program som konverterer et tall til binær – og det skal du – trenger du bare to operasjoner: `%` og heltallsdivisjon. Det er hele algoritmen.

2.5 Heksadesimal og oktal

Binære tall blir fort uleselige. Én million er 11110100001001000000_2. Det er tjue siffer, og du klarer ikke å sjekke om jeg har skrevet dem riktig uten å telle dem.

Vi mennesker har en løsning på dette fra før. Tenk på et mobilnummer. Ingen skriver 90123456. Vi skriver 90 12 34 56, fordi hjernen takler fire små biter bedre enn åtte siffer på rad. Kontonummer og fødselsnummer grupperes på samme måte.

Vi skal gjøre nøyaktig det samme med bit. Maskinen er og blir binær, og det kan vi ikke gjøre noe med. Men vi kan finne en kortskrift.

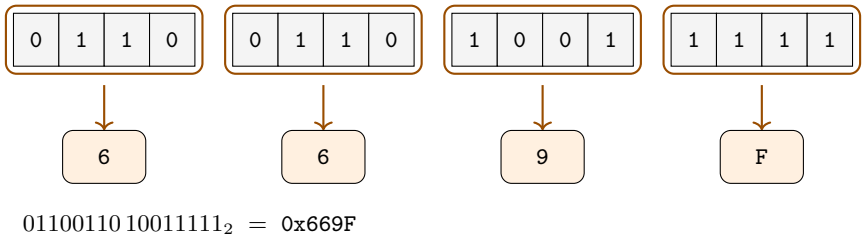
2.5.1 Heksadesimal

Heksadesimal er base 16. Det gir seksten siffer. Siden vi bare har ti sifferttegn, låner vi de seks første bokstavene:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
```

A er 10, B er 11, C er 12, D er 13, E er 14 og F er 15.

Hele poenget ligger i ett lite regnestykke: $16 = 2^4$. Når du har telt fra 0 til F, har du brukt opp nøyaktig alle kombinasjoner av fire bit. Derfor svarer **ett heksadesimalt siffer til nøyaktig fire bit**. Alltid, uten unntak, uten avrunding.



Figur 2.4: Grupper bitene i firere fra høyre, og slå opp hver gruppe. Ingen regning gjennom desimal.

Oppslagstabellen er hele metoden:

Binær	Heks	Desimal	Binær	Heks	Desimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

Tabell 2.1: De seksten firebitsmønstrene. Denne tabellen er verdt å kunne utenat.

2.5.2 Hva om det ikke går opp i fire?

Et spørsmål som kommer hver gang, og som du garantert møter i første øving. Hva gjør du med 1011010_2? Det er sju bit. Grupperer du i firere, sitter du igjen med en gruppe på tre.

Svaret er enkelt: **grupper alltid fra høyre, og fyll på med nuller til venstre.**

```
 1011010      sju bit
 101 1010     gruppert fra høyre -- øverste gruppe mangler ett bit
0101 1010     fylt på med en null foran
  5    A      slått opp i tabellen
```

Så $1011010_2 = 5A_{16}$.

Å legge til nuller foran endrer ingenting. Akkurat som 0042 er det samme tallet som 42, er `0101` det samme som `101`. Nuller helt til venstre er gratis.

I praksis slipper du som regel unna problemet. Inne i en maskin kommer ting nesten alltid i hele byte, altså åtte bit, og åtte går pent opp i to firergrupper. Men regelen er verdt å ha i bakhodet.

2.5.3 Hvorfor skriver vi 0x?

Heksadesimale tall skrives med prefikset `0x` i de fleste programmeringsspråk: `0x669F`.

Grunnen er praktisk og litt gammeldags. Da programmeringsspråkene ble laget, skrev man kode i en editor som bare hadde ett skriftsnitt og én tegnstørrelse. Du kunne rett og slett ikke skrive små, senkede tall. Så man fant på et prefiks i stedet, og det har hengt med siden.

De to skrivemåtene betyr det samme. `0x669F` og $669F_{16}$ er samme tall.

Advarsel :)

Ikke gå veien om desimal. Dette er den klassiske tidstyven på eksamen.
Oppgaven er: her er HTML-fargen `0x6699FF`, skriv rød-, grønn- og blåkomponenten binært. Den er ment å ta to minutter. Mange bruker førti.
Feilen er å regne heks til desimal, og så desimal til binær, tre ganger. Det er seks unødvendige regnestykker, hvert med sin egen sjanse for slurv.
Slå opp i stedet. `6` er `0110`. `9` er `1001`. Ferdig:

rød	`0x66`	→	`0110 0110`
grønn	`0x99`	→	`1001 1001`
blå	`0xFF`	→	`1111 1111`

Å konvertere mellom binær og heksadesimal via desimal er som å oversette fra bokmål til nynorsk via japansk. Det går an. Men hvorfor?

2.5.4 Heksadesimal i det fri

Farger på nettsider er heksadesimale. En farge er seks heksadesimale siffer, altså 24 bit, fordelt på tre komponenter à én byte:

	`#6699FF`	rød 102, grønn 153, blå 255	lyseblå
	`#FF0000`	rød 255, grønn 0, blå 0	full rød
	`#000000`	alt av	svart
	`#FFFFFF`	alt på	hvit
	`#87CEFA`	rød 135, grønn 206, blå 250	himmelblå

Heksadesimal dukker også opp i minneadresser, i feilmeldinger, i MAC-adresser, i IPv6-adresser og i tegnkodetabeller. Det siste møter vi i seksjon 2.9.

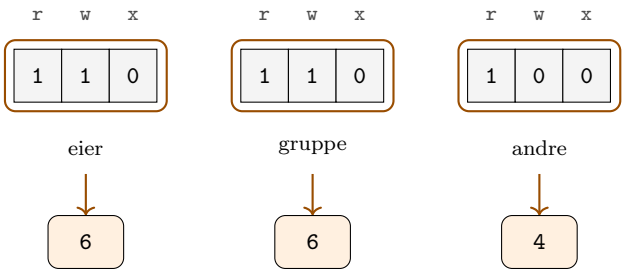
2.5.5 Oktal

Oktal er base 8, med sifrene 0 til 7. Samme idé som heksadesimal, men nå er $8 = 2^3$: **ett oktalt siffer er nøyaktig tre bit**.

Oktal var mer utbredt før. Men det har overlevd på ett sted alle informatikkstudenter møter: filrettigheter i UNIX og Linux.

En fil i Linux har ni rettighetsbit. Tre gjelder eieren av filen, tre gjelder gruppen filen tilhører, og tre gjelder alle andre. Innenfor hver trio betyr bitene lese, skrive og utføre, i den rekkefølgen.

Ni bit deler seg pent i tre grupper à tre. Og tre bit er nøyaktig ett oktalt siffer. Derfor skrives rettigheter som tre oktale siffer.



`chmod 664 notat.txt` – eier kan lese og skrive, gruppa kan lese og skrive, andre kan bare lese

Figur 2.5: Ni rettighetsbit gruppert i tre oktale siffer. `110` er 6, `100` er 4.

Skriver du `chmod 755 skript.sh`, sier du altså `111 101 101`. Eieren kan lese, skrive og utføre. Gruppa og alle andre kan lese og utføre, men ikke skrive. Og når `ls -l` viser deg `-rwxr-xr-x`, er det den samme informasjonen skrevet som bokstaver i stedet for som tall.

En liten felle til slutt: oktal og heksadesimal er ikke beslektet med hverandre. 8 er ingen potens av 16. Skal du fra oktal til heksadesimal, må du innom binær. Men det er ingen omvei, for begge stegene er rene oppslag.

2.6 Byte, verdiområde og designvalg

Ett bit rommer to verdier. To bit rommer fire: `00`, `01`, `10` og `11`. Tre bit rommer åtte. Hvert nytt bit dobler antall kombinasjoner, så

$$n \text{ bit rommer } 2^n \text{ ulike verdier, altså tallene fra 0 til } 2^n - 1.$$

Tre bit er ikke mye. Du får ikke engang lagret alderen din. Derfor grupperer vi bitene.

2.6.1 Byten

En **byte** er åtte bit. Det gir $2^8 = 256$ verdier, altså 0 til 255.

Byten er den minste enheten de fleste maskiner faktisk henter fra minnet. Vil du ha tak i ett enkelt bit, må du hente hele byten og plukke bitet ut selv.

Trenger du mer, tar du flere byte:

Bredde	Bit	Antall verdier	Område (uten fortegn)
1 byte	8	256	0 – 255
2 byte	16	65.536	0 – 65.535
4 byte	32	4.294.967.296	0 – ca. 4,3 milliarder
8 byte	64	$1{,}8 \times 10^{19}$	0 – ca. $1{,}8 \times 10^{19}$

Tabell 2.2: Hvor langt du kommer med et gitt antall byte.

Denne tabellen danner mye av grunnlaget for resten av kapitlet. Legg merke til hva den egentlig sier: *antall bit er et valg, og valget har en konsekvens.* Velger du for få, får du feil svar. Velger du for mange, sløser du med plass og båndbredde. Det finnes ikke noe riktig svar uten at du vet hva tallet skal brukes til.

Fun Fact

Hvorfor heter det egentlig *byte*? Det er det ingen full enighet om. Den vanligste forklaringen er at det er en lek med ordet *bite*, altså en jafs – en bit er en bitte liten flis, en byte er en hel munnfull. En annen forklaring er at det er en sammentrekning av *by eight*.
At en byte er åtte bit, er dessuten en konvensjon og ikke en naturlov. Tidlige maskiner brukte byte på 6, 7 og 9 bit. Standarden satte seg først på 1960-tallet.
Derfor finnes ordet *oktett*, som du har møtt i nettverksdelen av emnet. En oktett er per definisjon åtte bit, verken mer eller mindre. Nettverksstandarder bruker det ordet nettopp fordi de ikke kan ta for gitt hva «byte» betyr på maskinen i den andre enden.

2.6.2 Hvor mange bit trenger jeg?

Dette spørsmålet dukker opp hver gang du lager en variabel. Det finnes en enkel tommelfingerregel.

Siden $2^{10} = 1024 \approx 1000$, rommer ti bit omtrent tusen verdier. Derfra kan du gange deg oppover:

- 10 bit gir omtrent tusen verdier
- 20 bit gir omtrent en million
- 30 bit gir omtrent en milliard
- 18 bit deler seg i $10 + 8$, altså omtrent $1000 \times 256 = 256.000$

Litt mer presist trengs det omtrent **3,3 bit per desimalsiffer**, fordi $2^{3{,}3} \approx 10$. Skal du lagre et sekssifret tall, trenger du rundt 20 bit.

Advarsel :)

MB eller MiB? Her er en av de mest utbredte kildene til forvirring i faget.
I resten av verden betyr prefikset M en million, altså 10^6. I databransjen har det lenge betydd $2^{20} = 1.048.576$, fordi den potensen ligger nær en million og passer med hvordan minne er bygget. De to tallene skiller seg med nesten fem prosent, og forskjellen vokser oppover: for T er avviket ti prosent.
Standarden IEC 80000-13 løser det med egne prefikser for toerpotensene:

1 kB = 10^3 byte	1 KiB = 2^{10} = 1.024 byte
1 MB = 10^6 byte	1 MiB = 2^{20} = 1.048.576 byte
1 GB = 10^9 byte	1 GiB = 2^{30} = 1.073.741.824 byte

Diskprodusenter oppgir kapasitet i GB. Operativsystemer viser ofte GiB. Det er derfor en «1 TB»-disk plutselig har 931 GiB når du kobler den til. Ingen har jukset – det er samme tall, oppgitt i to ulike enheter.
I denne boka betyr MB alltid 10^6 byte, med mindre noe annet står.

2.6.3 BCD – når desimal må være desimal

Før vi forlater heltallene, skal vi se på en representasjon som velger stikk motsatt av alt vi har gjort til nå.

BCD, *binary coded decimal*, koder hvert desimalsiffer for seg i fire bit. Tallet 68 blir ikke 1000100_2, men

`0110 1000` der `0110` er sifferet 6 og `1000` er sifferet 8.

Mønstrene `1010` til `1111` – altså A til F – brukes aldri. Seks av seksten mulige mønstre kastes bort i hver eneste rute. Tabellen under viser hvor mye vi mister etterhvert som antall bitplasser øker:

Bit	BCD	Rent binært
8	0 – 99	0 – 255
16	0 – 9.999	0 – 65.535
20	0 – 99.999	0 – 1.048.575
32	0 – 99.999.999	0 – 4.294.967.295
64	16 siffer	19 hele siffer

Tabell 2.3: BCD kaster bort plass, og det blir verre jo bredere tallet er.

Hvorfor i all verden bruke det, da?

Fordi noen desimaltall rett og slett ikke finnes i binær. Vi skal se i kapittel 2.10 at 0,1 ikke kan skrives eksakt med bit. En bank som regner i kroner og øre, kan ikke leve med at 0,10 kroner blir 0,100000001. BCD gir eksakt desimalpresisjon, og det er verdt både plassen og tiden når pengene skal stemme på øret. IBMs stormaskiner har fortsatt maskinvarestøtte for BCD, og kode i banker og forsikringsselskaper bruker det i dag.

Det finnes en bonusgrunn også. I ASCII og Unicode er de fire nederste bitene i et siffertegn nøyaktig BCD-sifferet. Tegnet `'7'` har koden `0011 0111`, og de fire siste bitene er `0111`, altså 7. Å gjøre innskrevet tekst om til BCD er derfor nesten gratis.

BCD er et godt eksempel på den bærende ideen. Representasjonen har valgt eksakthet framfor kompakthet. Derfor er den riktig noen steder, og feil de fleste andre.

2.7 Negative tall

Alt vi har gjort til nå gjelder positive heltall. Men programmer må kunne trekke fra. Kontoer går i minus, og termometre viser kuldegrader.

Problemet er at vi ikke har noe minustegn. Vi har bare bit. Skal vi klare å si «minus», må vi bygge det av nuller og enere som alle de andre tingene.

Det finnes flere måter. Vi tar den som vant først, og de som tapte etterpå – for det er lærerikt å se hvorfor de tapte.

2.7.1 Toerkomplement: øverste bit har negativ vekt

Toerkomplement er det praktisk talt alle maskiner bruker. Og hele definisjonen får plass i én setning:

Tankevekker

I toerkomplement er alt akkurat som før. Bortsett fra at det øverste bitet har *negativ* vekt.

Det er alt. Ingen ny regnemåte, ingen egen fortegnsregel. Vektene i en byte er fortsatt potenser av to, men den øverste har fått et minustegn foran seg:

posisjon	7	6	5	4	3	2	1	0
vekt	−128	64	32	16	8	4	2	1

Alt annet gjør du som før. Gang hvert bit med vekten sin, og legg sammen.

Husker du lommeboka fra seksjon om vekter? Nå har den fått ett nytt kort: et gjeldskort på 128 kroner. Tar du det med, starter du på minus 128 og kan jobbe deg oppover med de vanlige sedlene.

Eksempel 10. Hvilket tall er `10000000`?

$$1 \times (-128) + 0 + 0 + 0 + 0 + 0 + 0 + 0 = -128$$

Eksempel 11. Hvilket tall er `11111111`?

$$-128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = -128 + 127 = -1$$

Eksempel 12. Hvilket tall er `11111011`?

$$-128 + 64 + 32 + 16 + 8 + 0 + 2 + 1 = -128 + 123 = -5$$

Eksempel 13. Hvilket tall er `00000101`?

$$0 + 0 + 0 + 0 + 0 + 4 + 0 + 1 = 5$$

Legg merke til to ting.

Er øverste bit 0, bidrar det ingenting. Tallet er positivt, akkurat som før. De positive tallene ser helt like ut som de gjorde i forrige seksjon.

Er øverste bit 1, starter du på -128 og legger til. Da havner du et sted mellom -128 og -1.

Derfor kalles øverste bit ofte *fortegnsbitet*. Vær litt forsiktig med det navnet. Det er ikke et rent fortegn slik minustegnet er det. Det er et siffer med en vekt, som alle de andre. Vekten er bare negativ.

2.7.2 Å snu fortegnet

For å negere et toerkomplementtall gjør du to ting: **inverter alle bitene, og legg til 1**.

Eksempel 14. Finn -5 ut fra 5.

$$\begin{aligned} 5 &= \texttt{00000101} \\ \text{inverter} &\to \texttt{11111010} \\ \text{legg til 1} &\to \texttt{11111011} \end{aligned}$$

Og `11111011` er nettopp -5, slik vi regnet ut i eksempel 12.

Metoden virker begge veier. Inverterer du `11111011`, får du `00000100`. Legger du til 1, får du `00000101`, altså 5 igjen.

Hvorfor virker det? Å invertere alle bitene i et åttebitstall er det samme som å trekke tallet fra `11111111`, som er 255. Legger du så til 1, har du regnet ut $256 - x$. Og $256 = 2^8$ er nøyaktig det punktet der en byte «renner rundt». Hva det betyr, kommer med én gang.

2.7.3 Hvorfor addisjon virker uendret

Her ligger hele grunnen til at toerkomplement vant.

Prosessoren trenger ikke vite om tallene er positive eller negative. Den legger sammen bit for bit som om alt var vanlige positive tall, kaster bitet som ikke får plass, og svaret blir riktig.

Eksempel 15. Regn ut $5 + (-3)$ med åtte bit.

```
  00000101     (5)
+ 11111101     (-3)
----------
1 00000010     menten faller utenfor
  ========
  00000010     = 2
```

Svaret er 2, som det skal være. Menten lengst til venstre forsvinner bare. Den er ikke en feil, og du skal ikke ta vare på den.

Eksempel 16. Regn ut $-5+3$.

```
  11111011     (-5)
+ 00000011     (3)
----------
  11111110     = -2
```

Riktig igjen, og denne gangen uten mente i det hele tatt.

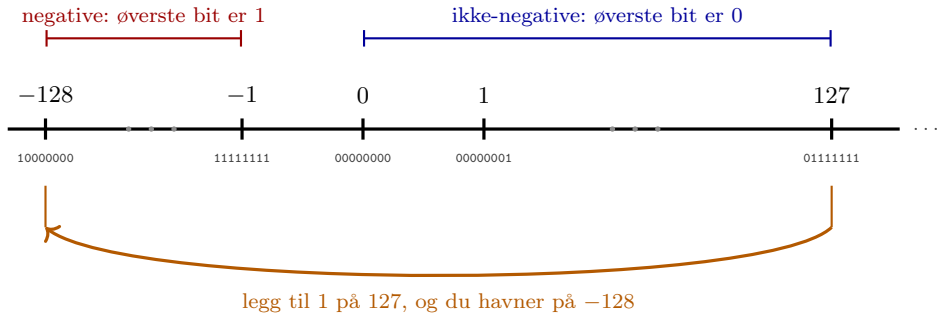
Tankevekker

Tenk på en klokke. Den er analogien som får dette til å klikke på plass for de fleste.
Klokka er 3. Du vil gå fire timer tilbake. Du kan trekke fra 4 og få 11. Eller du kan legge til 8 og gå hele veien rundt skiva – $3+8=11$. Samme svar.
Det er fordi skiva har tolv plasser. Å trekke fra 4 og å legge til $12-4=8$ gir alltid det samme, så lenge du kaster hele runder.
En byte er en klokke med 256 plasser. Å trekke fra 5 og å legge til $256-5=251$ gir samme svar, så lenge du kaster hele runder. Og å kaste hele runder er nøyaktig det som skjer når det niende bitet faller utenfor.
Derfor trenger maskinen aldri en egen subtraksjonskrets. *Trekk fra* blir *negér og legg til*, og ALU-en slipper unna med én operasjon. Vi ser hvordan den bygges i kapittel 5.

2.7.4 Tallinja og verdiområdet

Med åtte bit i toerkomplement får vi tallene fra -128 til 127.

Den øvre grensen er `01111111`, altså $64+32+16+8+4+2+1=127$. Den nedre er `10000000`, altså -128. Til sammen 256 verdier – akkurat like mange som en byte alltid rommer. Vi har bare flyttet halvparten av dem ned under null.



Figur 2.6: Toerkomplement med åtte bit. Tallinja er egentlig en ring – teller du forbi 127, kommer du inn igjen nederst.

To ting følger av dette.

Det finnes ingen minus null. Negerer vi `00000000`, får vi `11111111` etter inverteringen. Legger vi til 1, blir det `1 00000000`. Menten faller utenfor, og vi står igjen med `00000000` – samme mønster vi startet med.

Null er altså sin egen negasjon, og det finnes bare én av den. Det høres ut som en detalj. Men det betyr at maskinen slipper å teste for to ulike nullmønstre etter hver eneste operasjon, og det er mye spart.

Området er ikke symmetrisk. Det finnes et -128, men ikke et $+128$. Det gir en pussighet som faktisk biter: negerer du -128, får du -128 tilbake. Inverter `10000000` og du får `01111111`. Legg til 1, og du er tilbake på `10000000`.

I Java betyr det at `Math.abs(Integer.MIN_VALUE)` gir et negativt tall. Det er ingen feil i Java. Det er en representasjon som har møtt grensen sin.

2.7.5 De som tapte

Toerkomplement er den beste av flere måter, og det er nyttig å vite hva den ble målt mot.

Fortegn og magnitude. Den mest opplagte ideen. Bruk øverste bit som et rent fortegn, og la de sju andre være tallets størrelse. Da er `00100101` lik $+37$ og `10100101` lik -37. Lett å lese for et menneske.

To ting ødelegger den. For det første finnes det to nuller, `00000000` og `10000000`, altså $+0$ og -0. Maskinen må teste for begge hver gang den lurer på om noe er null.

For det andre blir addisjonen et mareritt. Skal maskinen legge sammen 4 og -2, må den først sammenligne fortegn, så sammenligne størrelser, så bestemme om den skal legge til eller trekke fra, og til slutt finne fortegnet på svaret. Fire tilfeller, der toerkomplement har ett.

Fortegn og magnitude er ikke helt borte, forresten. Flyttall bruker det, og vi ser hvorfor i seksjon 2.10.

Enerkomplement. En smartere idé. Negér ved å invertere alle bitene, uten å legge til 1. Da er -37 lik `11011010`.

Addisjonen blir nesten grei. Men bare nesten: renner det ut en mente, må den legges til på nytt nederst. Og de to nullene er fortsatt der – `00000000` og `11111111` er begge null.

Toerkomplement er altså enerkomplement pluss én. Den ene ekstra addisjonen ved fortegnsskifte er hele prisen. Til gjengjeld forsvinner både minus null og den ekstra menten. Det er en veldig god handel, og den er grunnen til at maskinen på pulten din regner slik den gjør.

2.7.6 Hvordan vet du at det er toerkomplement?

Dette er et av de beste spørsmålene man kan stille i dette kapitlet.

Du får utdelt en byte:

```
10000000
```

Er det -128 eller 128?

Svaret er at du ikke kan vite det. Begge tolkningene er helt gyldige. Er byten en `signed char` i C, er den -128. Er den en `unsigned char`, er den 128. Bitene er de samme i begge tilfeller, og maskinen lagrer ikke noe sted hvilken som gjelder.

Opplysningen ligger i *typen*. Den står i kildekoden din. Den finnes i kompilatorens hode mens den oversetter, og den avgjør hvilke maskininstruksjoner kompilatoren velger. Men den finnes ikke i minnet. Når programmet kjører, er det bare bitene igjen.

Dette er den bærende ideen i sin skarpeste form. Det er også kilden til en hel familie virkelige feil. Bytter du en `signed` mot en `unsigned` i en lengdesjekk, kan et negativt tall plutselig bli en veldig stor positiv lengde. Og der har du et sikkerhetshull.

Note-to-myself

I Java har alle heltallstypene fortegn: `byte` (−128 til 127), `short`, `int` (rundt ±2,1 milliarder) og `long`.
C og C++ har i tillegg `unsigned`-varianter. De har dobbelt så stort positivt område, fordi de slipper å bruke halvparten av mønstrene på negative tall.
Valget er ditt, og du må ta det bevisst. Spør alltid: kan denne verdien bli negativ, og hvor stor kan den bli?

2.8 Overflyt

Nå har vi etablert første halvdel av den bærende ideen: et bitmønster betyr det vi har blitt enige om at det betyr. Resten av kapitlet handler om andre halvdel – at hver representasjon har en grense, og hva som skjer når du treffer den.

2.8.1 Kilometerteller

Tenk på kilometertelleren i en gammel bil, den med mekaniske hjul. Den har seks siffer og viser opptil `999999`. Kjører du én kilometer til, ruller alle hjulene rundt samtidig, og telleren viser `000000`.

Bilen har ikke glemt de kilometerne. Telleren har bare ikke plass til å si det. Den har møtt grensen sin, og reagerer på den eneste måten den kan: den begynner på nytt.

Det samme skjer i en datamaskin. Det heter **overflyt**.

2.8.2 Overflyt i praksis

Her er et program som ser ganske uskyldig ut:

```
int v1 = 1000;
int v2 = 1000 * 1000;
int v3 = 1000000 * 1000000;

System.out.println(v1);
System.out.println(v2);
System.out.println(v3);
```

og her er hva det skriver ut:

```
1000
1000000
-727379968
```

Den siste linja er grundig feil. Feil størrelsesorden, feil antall siffer, feil fortegn.

Hva skjedde? En `int` i Java er fire byte, altså 32 bit, og rommer tall opp til 2 147 483 647. Det riktige svaret, 1 000 000 000 000, trenger 40 bit. De åtte øverste bitene har ingen steder å være. De blir kastet – ikke lagret, ikke rapportert, ikke mulig å få tilbake.

Og se hva som blir igjen. Av de 32 bitene som overlever, er det øverste satt. I toerkomplement har det øverste bitet vekten -2^{31}. Derfor blir svaret negativt. Det er ikke tilfeldig og ikke mystisk – det er nøyaktig regelen fra seksjon 2.7, anvendt på et bitmønster som havnet der ved et uhell.

Advarsel :)

Det verste med overflyt er ikke at svaret blir feil. Det er at ingen sier fra.
Programmet over kaster ingen feilmelding, logger ingenting og krasjer ikke. Det regner videre med $-727\,379\,968$ som om det var et helt vanlig tall. Alt som beregnes etterpå blir feil, og feilen kan dukke opp mange steg senere, et helt annet sted i programmet.
Derfor bør du venne deg til å tenke på verdiområdet *før* du velger datatype. Ikke etterpå.

2.8.3 Hva kan vi gjøre?

Det opplagte svaret er å bruke en større type. Bytter vi `int` mot `long`, som er åtte byte, får 10^{12} god plass, og programmet skriver ut riktig svar.

Men det er en utsettelse, ikke en løsning. En `long` har også en grense. Den ligger bare lenger ute.

2.8.4 Python, og hvorfor det ikke er gratis

Noen språk velger en helt annen vei. Python har heltall uten fast bredde. Trenger tallet flere bit, skaffer Python flere bit. Du kan gange sammen tall til minnet ditt tar slutt:

```
v = 1000000 * 1000000
for i in range(10):
    v = v * v
    print(v)
```

Dette gir riktig svar hele veien. Det er behagelig, og en av grunnene til at Python er et godt førstespråk.

Men det koster, og det koster på to måter.

Den ene er den opplagte. Hver eneste regneoperasjon må sjekke om resultatet fortsatt får plass, og eventuelt skaffe mer minne. Der C legger sammen to tall med én maskininstruksjon, gjør Python et titalls.

Den andre er mindre opplagt og verre. Når kompilatoren ikke vet hvor stort et tall kan bli, kan den heller ikke optimalisere. Den kan ikke legge tallet i et register, den kan ikke bruke den raske addisjonsinstruksjonen, den kan ikke slå sammen operasjoner. Usikkerheten om størrelse forplanter seg gjennom hele oversettelsen.

Til sammen er dette en stor del av grunnen til at Python er tregt. Det er et bevisst designvalg: språket har prioritert at det skal være behagelig å programmere i, framfor at det skal gå fort. C har valgt motsatt. Ingen av valgene er feil, men du bør vite hvilket du har gjort.

Advarsel :)

Og her er fella. Det er lett å lese avsnittet over og konkludere med at «i Python slipper jeg å tenke på overflyt». Det stemmer ikke.
Store deler av Pythons biblioteker er skrevet i C. Og C har ikke Pythons forhold til tallstørrelse – der er en 32-bits heltall en 32-bits heltall, punktum.
Regner du med vanlige Python-tall, er du trygg. Regner du i et bibliotek som NumPy, er du tilbake i C-verdenen med faste bitbredder og stille overflyt. Samme språk, samme fil, to helt ulike regler.
Sjekk hvilken datatype du faktisk regner med. Ikke hvilket språk du skriver i.

Prosessoren har for øvrig egne flagg for både mente og overflyt, som et program *kan* sjekke. Vi ser på dem i kapittel 4.

2.8.5 Ariane 5

Den 4. juni 1996 skjøt Den europeiske romfartsorganisasjonen opp Ariane 5 for første gang. Trettini sekunder senere var raketten og de fire satellittene om bord borte. Tapet anslås vanligvis til rundt 370 millioner dollar.

Årsaken var en konvertering mellom to representasjoner.

Rakettens treghetsnavigasjonssystem holdt rede på horisontal hastighet som et 64-bits flyttall. Denne verdien ble konvertert til et 16-bits fortegnstall, som rommer tall opp til 32 767. Under oppstigningen ble den horisontale hastigheten større enn det. Konverteringen feilet, og navigasjonsdatamaskinen slo seg av. Reservesystemet kjørte identisk programvare og hadde feilet av samme grunn noen hundredels sekund tidligere. Styringssystemet tolket feilmeldingen som flydata, svingte dysene helt ut, raketten begynte å brekke, og selvdestruksjonen utløste seg.

Det interessante er hvor koden kom fra. Den var arvet fra Ariane 4, der den hadde virket feilfritt i årevis. På Ariane 4 *kunne ikke* den horisontale hastigheten bli stor nok til å sprenge et 16-bits tall. Flybanen tillot det rett og slett ikke. Representasjonen var riktig valgt, gitt forutsetningene.

Ariane 5 var kraftigere og fløy annerledes. Forutsetningen holdt ikke lenger. Ingen hadde gjort en feil i koden. De hadde arvet en antakelse om verden, og verden hadde endret seg.

Tankevekker

Det er en bedre historie enn «noen valgte for få bit».
Et valg av representasjon er alltid et valg gitt en sammenheng. Koden bærer ikke sammenhengen med seg. Og den varsler deg ikke når sammenhengen forsvinner.
Neste gang du skriver `int` uten å tenke: hvilken antakelse om verden gjør du akkurat nå, og hvem kommer til å arve den?

Tenk om...

I 1999 var mange redde for at datamaskiner verden over skulle slutte å virke ved årsskiftet til 2000. Man kalte det Y2K-frykten, og det ble brukt enorme summer på å forberede seg.
Hva tror du det handlet om? Og hvilken representasjon var det som hadde møtt grensen sin?
Hint: minne var dyrt på 1960- og 70-tallet, og et årstall kan skrives med to siffer hvis du tar for gitt at de to første er 1 og 9.

2.9 Tekst og tegn

Nå forlater vi tallene. Eller: vi later som. For maskinen finnes det fortsatt bare tall. Alt vi skal gjøre i denne seksjonen, er å bli enige om hvilket tall som skal bety hvilken bokstav.

2.9.1 Den enkleste ideen, og problemet med den

Du skal lagre bokstaven Ø. Du har bare tall. Løsningen er opplagt: lag en tabell.

0	A
1	B
2	C
⋮	⋮
27	Ø
28	Å

Lagrer du tallet 27, mener du Ø. Ferdig. Og dette er faktisk slik det gjøres, den dag i dag. Det finnes ingen dypere magi her.

Problemet dukker opp i det øyeblikket to maskiner skal snakke sammen.

Maskin A sender tallet 27 og mener Ø. Maskin B tar imot 27, slår opp i *sin* tabell, og finner noe helt annet. Resultatet har et navn: **mojibake**, fra japansk, omtrent «tegnforvandling». Teksten er intakt som tall, og søppel som tekst.

Dette er ikke et historisk problem. Du har sett det selv. Hver gang du limer inn tekst fra én app til en annen og æøå blir til `Ã¦Ã¸Ã¥` eller til spørsmålstegn, har du sett to tabeller som ikke er enige.

Tankevekker

Legg merke til hva en tegnkoding egentlig er: en *avtale mellom to parter om hva et bitmønster betyr*.
Det er samme type avtale som en nettverksprotokoll, og slike har du allerede jobbet med i første halvdel av emnet. Forskjellen er bare at HTTP avtaler hva en melding betyr, mens UTF-8 avtaler hva en byte betyr.
Prinsippet er det samme. Det er derfor tegnkoding hører hjemme i et kapittel om representasjon – det er den bærende ideen i sin reneste form.

2.9.2 ASCII

Det første virkelig vellykkede forsøket på en felles tabell var **ASCII**, *American Standard Code for Information Interchange*, fra 1963. Den brukte **sju bit**, som gir $2^7 = 128$ plasser.

Sju bit, ikke åtte. Dette er verdt å få med seg, for ASCII omtales ofte upresist som en åttebitskode. Den åttende biten i byten var opprinnelig ledig. Den ble gjerne brukt til *paritet*: en enkel feilsjekk der biten settes slik at antallet enere i byten alltid blir et partall. Linjene var dårlige, og en byte som kom fram med ulikt antall enere var åpenbart forvansket underveis.

	0	1	2	3	4	5	6	7
0	NUL	DLE	␣	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	**BEL**	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	**DEL**

Tabell 2.4: ASCII-tabellen. Kolonnen gir øverste heksadesimale siffer, raden det nederste. Bokstaven `K` står i kolonne 4, rad B, og har altså koden `0x4B`.

Tabellen er ikke satt sammen tilfeldig. Fire trekk er verdt å se etter.

Sifrene ligger på `0x30` til `0x39`. De fire nederste bitene i et siffertegn er dermed nøyaktig sifferverdien. Tegnet `'7'` er `0011 0111`, og de fire siste bitene er `0111`, altså 7. Det er samme mønster som BCD i seksjon 2.6.3, og det er ingen tilfeldighet. Skal du gjøre et innskrevet siffer om til et tall, trekker du bare fra 48.

Sifrene kommer før bokstavene. Vil du sjekke om et tegn er et siffer, holder det å sjekke om koden ligger mellom `0x30` og `0x39`. Én sammenligning, ingen oppslagstabell.

Store og små bokstaver ligger nøyaktig 32 fra hverandre. `A` er `0x41` og `a` er `0x61`. Siden $32 = 2^5$, skiller de seg i ett eneste bit. Å veksle mellom stor og liten bokstav er derfor ikke et oppslag – det er å snu bit nummer 5.

De 32 første plassene er ikke tegn. De er *kontrolltegn*, altså kommandoer til utstyret i den andre enden. `LF` skifter linje, `CR` flytter skrivehodet til venstre marg, `HT` er tabulator, `ESC` innleder en styresekvens. Det finnes koder for «start tekst», «slutt tekst», «mottatt» og «ikke mottatt». ASCII ble laget for teleprintere, ikke for skjermer.

Fun Fact

To av kontrolltegnene fortjener en egen historie.
BEL, kode `0x07`, fikk en fysisk bjelle på mottakermaskinen til å ringe, slik at operatøren kom bort og så hva som foregikk. Koden finnes fortsatt: skriver du `printf("\a")` i C, ber du terminalen om å pipe.
DEL, kode `0x7F`, ligger helt alene nederst i høyre hjørne, langt fra de andre kontrolltegnene. Hvorfor? Fordi `0x7F` er sju enere – alle bit satt.
Tekst ble overført på hullbånd, der hvert tegn var en kolonne av hull og ikke-hull. Slo du feil, kunne du ikke sette hullene tilbake. Men du kunne slå ut *alle* hullene i kolonnen, og dermed gjøre tegnet om til sju enere. Derfor måtte «slett dette tegnet» ha nettopp den koden.
Plasseringen i tabellen er altså en fossil etter en maskin som ikke finnes lenger.

2.9.3 Når 128 tegn ikke holder

ASCII ble laget i USA, for engelsk. Det finnes ingen Æ, Ø eller Å. Og ingen ä, é, ñ eller ß.

Den første norske løsningen var å ofre noe annet. Vi tok tegnene `[`, `\` og `]` og erstattet dem med Æ, Ø og Å. Krøllparentesene `{`, `|` og `}` ble til æ, ø og å. Nå kunne vi skrive norsk. Vi kunne bare ikke skrive C-kode lenger, siden krøllparentesene var borte.

Og en maskin uten den norske tabellen leste tallene med sin egen. Sender du «Gjøvik» fra en norsk maskin til en amerikansk, kommer det fram som

```
Gj|vik
```

Neste steg var **ISO 8859-1**, også kalt Latin-1. Ta i bruk den åttende biten, og få 256 plasser i stedet for 128. De første 128 er identiske med ASCII. De neste 128 rommer de vesteuropeiske tegnene, inkludert våre tre.

Det var en klar forbedring, og den holdt i mange år. Men den løste ikke problemet – den flyttet det. Nå fantes det femten ulike ISO 8859-varianter: én for latinsk, én for kyrillisk, én for gresk, én for hebraisk. Alle uforenlige med hverandre. Og for kinesisk, japansk og koreansk, med titusenvis av tegn, var 256 plasser aldri i nærheten av nok.

2.9.4 Unicode: kodepunkt er ikke koding

Unicode er svaret, og det består i mer enn å utvide tabellen. Unicode gjør noe som er lett å overse: den **deler ett spørsmål i to**.

1. **Hvilket tall er dette tegnet?** Svaret kalles tegnets *kodepunkt*. Det er et rent tall, uavhengig av bit og byte. Bokstaven ø har kodepunktet 248, som skrives `U+00F8` – altså `U+` etterfulgt av kodepunktet i heksadesimal.

2. **Hvordan blir det tallet til byte?** Svaret kalles en *koding*, på engelsk *encoding*. Og det finnes flere, for det samme kodepunktet.

Dette skillet er hovedpoenget i seksjonen. ASCII og Latin-1 hadde ikke noe slikt skille. Der *var* tegnets nummer det samme som byten i minnet, og derfor kunne tabellen aldri bli større enn én byte tillot.

Unicode har plass til litt over en million kodepunkter, fra `U+0000` til `U+10FFFF`. Over 150 000 av dem er tatt i bruk. Der finnes alle levende skriftsystemer, en god del utdødde, matematiske symboler og emojier. Nye tegn kommer til når verden trenger dem – eurosymbolet måtte for eksempel finne seg en plass da valutaen ble innført.

De første 128 kodepunktene er identiske med ASCII, og de første 256 med Latin-1. Det var et bevisst valg. Nå skal vi se hva det kjøpte oss.

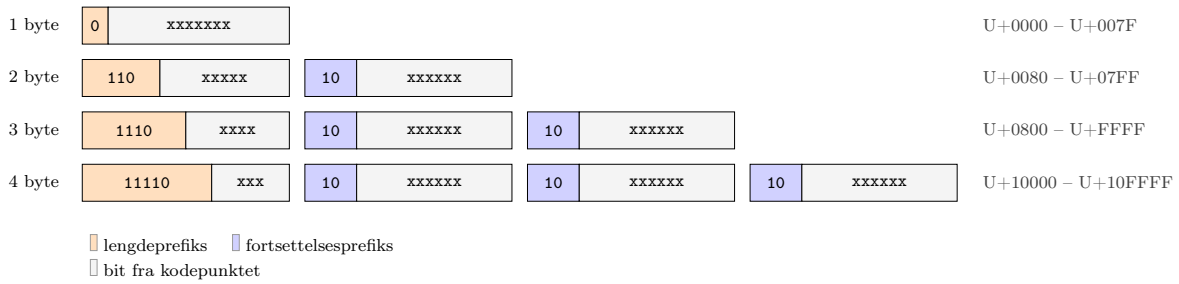
2.9.5 UTF-8

Den mest brukte kodingen heter **UTF-8**, og den er en av de smarteste konstruksjonene i faget.

Utgangspunktet er et dilemma. Skal vi bruke fire byte per tegn, så det er plass til alle kodepunkter? Da blir en helt vanlig engelsk tekstfil fire ganger så stor som nødvendig, siden nesten alle tegnene i den ligger under 128. Skal vi bruke én byte? Da får vi ikke plass til noe.

UTF-8 bruker et *variabelt* antall byte – én, to, tre eller fire. Og den løser det underliggende problemet med noen få **prefiksbit** i starten av hver byte.

Tenk på et togsett. Lokomotivet har et skilt som sier hvor mange vogner som følger etter. Hver vogn har et skilt som sier «jeg er en vogn, ikke et lokomotiv». Ser du et lokomotiv, vet du hvor langt toget er. Ser du en vogn, vet du at du står midt i et tog.



Figur 2.7: UTF-8. Første byte sier hvor mange byte tegnet består av, og hver etterfølgende byte begynner med `10`.

Reglene er:

- Begynner byten med `0`, er tegnet én byte langt, og de sju siste bitene er kodepunktet. **Dette er nøyaktig ASCII.**
- Begynner byten med `110`, følger det én byte til. Med `1110` følger to. Med `11110` følger tre. Antall enere før den første nullen er antall byte.
- Begynner byten med `10`, er den en *fortsettelse* av et tegn som startet tidligere.

To konsekvenser følger, og begge er verdt å tenke gjennom.

En ren ASCII-fil er allerede en gyldig UTF-8-fil. Ikke én eneste endring trengs. All verdens engelskspråklige tekst og all eksisterende programkode ble UTF-8 den dagen standarden kom. Derfor vant UTF-8, og i dag brukes den av rundt 99 prosent av alle nettsider.

Kodingen er selvsynkroniserende. Dette er det virkelig smarte, og det er lettest å se på nettverket.

Du mottar en byte-strøm, og en pakke forsvinner underveis. Nå står du midt inne i noe, og vet ikke hvor. Med UTF-8 ser du det med én gang: begynner byten med `10`, er du inni et tegn som startet før pakketapet. Da hopper du framover til du finner en byte som ikke gjør det, og der starter neste hele tegn.

Du kan altså plukke opp tråden igjen, hvor som helst, uten å lese filen fra starten. Uten prefiksbitene ville én tapt byte ødelagt alt som kom etter.

2.9.6 Ett tegn, kodet bit for bit

La oss gjøre det konkret med ø.

Steg 1: finn kodepunktet. Bokstaven ø har kodepunktet `U+00F8`, altså 248_{10}.

Steg 2: hvor mange byte? 248 er større enn 127 og mindre enn 2048. Vi trenger toformen.

Steg 3: skriv kodepunktet binært. $248_{10} = 11111000_2$. Toformen har plass til 11 bit, så vi fyller på med nuller foran: `000 11111000`.

Steg 4: fordel bitene. Toformen er `110xxxxx 10xxxxxx`, altså fem bit i første byte og seks i andre. Vi deler `00011111000` i `00011` og `111000`:

```
første byte:  110 00011   =  11000011  =  0xC3
andre byte:    10 111000  =  10111000  =  0xB8
```

Bokstaven ø er altså `0xC3 0xB8` i UTF-8.

Åpner du en norsk tekstfil med et program som tror den er Latin-1, blir de to bytene tolket som hvert sitt tegn. `0xC3` er `Ã` og `0xB8` er `¸`. Der har du `Ã¸`-mønsteret du har sett før.

Vil du prøve en til: den smilende emojien har kodepunktet `U+1F600`. Kodepunktet trenger 17 bit, og det er mer enn de 16 treformen har plass til. Altså må vi opp i fireformen, og resultatet blir `0xF0 0x9F 0x98 0x80`.[1]

2.9.7 De andre kodingene

UTF-32 bruker fire byte for alle tegn, alltid. Det er sløsende, men enkelt: tegn nummer n ligger på byte $4n$, og du kan hoppe rett dit.

UTF-16 bruker to byte for de vanligste tegnene og fire for resten. Windows og Java bruker den internt. De fire-byte tilfellene løses med *surrogatpar*, altså to spesielle 16-bitsverdier i området `U+D800`–`U+DFFF` som sammen peker ut ett kodepunkt. Det er grunnen til at `String.length()` i Java kan gi 2 for en emoji som åpenbart er ett tegn.

Det viktige er at dette er *tre kodinger av de samme kodepunktene*. Bokstaven ø er `U+00F8` i alle tre. Bare bytene er ulike. Det er skillet mellom kodepunkt og koding som gjør den setningen mulig å si.

2.9.8 Tekst eller tall?

Her er et spørsmål som er lettere å stille enn å svare på. Du skriver 38,5 i en fil. Hvordan lagres kommaet?

Svaret avhenger av hva du holder på med, og det er nettopp poenget.

Skriver du en setning – «jeg følte meg varm, og temperaturen var 38,5» – er alt sammen tekst. Sifferet 3 er et tegn med et kodepunkt. Sifferet 8 er et tegn. Kommaet er et tegn. Sifferet 5 er et tegn. Fire tegn, fire kodepunkter, fire byte i UTF-8. Maskinen kan ikke regne med dette. Den vet ikke at det er et tall.

Lagrer du derimot verdien 38,5 for å regne med den, bruker du et flyttall, slik vi skal se i seksjon 2.10. Og da lagres kommaet *ikke i det hele tatt*. Det ligger i formatet, ikke i dataene.

Samme tall på papiret. To helt ulike ting i minnet, med helt ulike regler. Dette er den bærende ideen, i en form du møter hver gang du leser inn data fra en fil.

2.9.9 Tegn er ikke byte

Om du tar med deg én praktisk ting fra denne seksjonen, la det være denne.

Ordet «Gjøvik» har seks tegn. I UTF-8 er det sju byte, fordi ø trenger to. «Åsgårdstrand» har tolv tegn og fjorten byte. En emoji er ett tegn, fire byte i UTF-8, og to kodeenheter i UTF-16.

Det betyr at fire ting som virker like, ikke er det:

[1] Den opprinnelige definisjonen av UTF-8 tillot opptil seks byte. RFC 3629 begrenset den til fire i 2003, for å matche Unicodes øvre grense på `U+10FFFF`. Femte og sjette form er ugyldige i dag, og en dekoder skal avvise dem.

- antall tegn en bruker ser på skjermen
- antall kodepunkter i strengen
- antall kodeenheter i den kodingen språket ditt bruker internt
- antall byte strengen tar i en fil eller på nettverket

I Python ser forskjellen slik ut:

```
>>> s = "Gjøvik"
>>> len(s)
6
>>> len(s.encode("utf-8"))
7
```

Feilen som følger av å blande disse sammen, er svært vanlig. Du setter av 20 byte til et navn som kan være 20 tegn. Så kommer det inn et navn med æøå, og du kutter midt i et tegn. Det som kommer ut er en ugyldig byte-sekvens som mottakeren ikke kan tolke i det hele tatt.

Note-to-myself

Regelen er enkel. Lagre og send tekst som UTF-8. Si eksplisitt fra at det er UTF-8. Tell i tegn når du snakker med mennesker, og i byte når du snakker med filer og nettverk. Og ikke kutt en byte-sekvens uten å vite hvor tegngrensene går.

2.10 Flyttall og presisjon

Heltall er fine, men de holder ikke. Noen størrelser er for store, noen er for små, og noen har desimaler.

Avstanden til Andromedagalaksen er omtrent $2{,}4 \times 10^{19}$ kilometer. Et hydrogenatom er omtrent en tidels nanometer bredt. Og π har uendelig mange desimaler, hvorav vi som regel trenger noen få.

Alle tre har noe til felles: vi trenger ikke alle sifrene. Vi trenger noen få signifikante siffer, pluss en opplysning om hvor stort tallet er.

Det gjør du allerede selv. Du sier ikke at Norge har 5 594 340 innbyggere. Du sier 5,6 millioner. Du har beholdt to siffer og kastet resten, fordi resten uansett er utdatert i morgen.

2.10.1 Ideen

Et **flyttall** lagres som to deler:

$$\text{verdi} = \pm\, \text{mantisse} \times \text{base}^{\text{eksponent}}$$

Mantissen bærer presisjonen. Eksponenten bærer størrelsesordenen. I en datamaskin er basen 2 og ikke 10, men ideen er den samme.

Hvorfor heter det flyttall?

Navnet er faktisk beskrivende, og det er verdt å stoppe ved.

Skriver du $2{,}4 \times 10^{19}$, sier eksponenten: flytt kommaet nitten plasser til høyre. Skriver du $9{,}6 \times 10^{-8}$, sier den: flytt kommaet åtte plasser til venstre.

Kommaet står altså ikke fast. Det *flyter*, og eksponenten forteller hvor det skal ende opp. Derfor heter det flyttall, og på engelsk *floating point*.

Motsatsen finnes også: *fastkommatall*, der kommaet alltid står på samme sted. Det brukes fortsatt i enkelte systemer, for eksempel når man regner i hele øre.

Hva slags handel er dette?

Dette er viktig å forstå, for det er lett å tro at flyttall er gratis.

Har du 32 bit, har du 2^{32} mulige bitmønstre. Verken flere eller færre enn du hadde med et heltall. Du kan ikke trylle fram nye verdier.

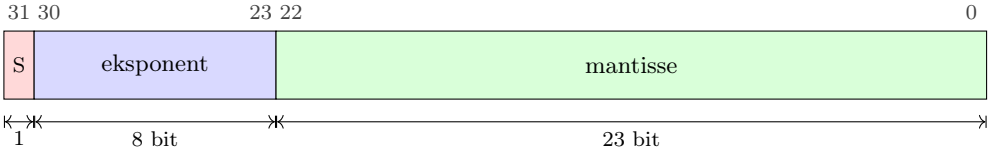
Det du gjør, er å *fordele* de samme mønstrene annerledes. I stedet for fire milliarder tett pakkede heltall får du fire milliarder verdier spredt utover et enormt område. Tett samlet nær null, gissent langt ute.

Tenk på et kart. Zoomer du ut, får du hele landet på skjermen, men du ser ikke lenger hvilken gate du står i. Kartet er like stort uansett – det er detaljene som forsvinner. Flyttall gjør nøyaktig det samme med tall.

2.10.2 IEEE 754

Nesten alle maskiner følger standarden **IEEE 754** fra 1985. Den definerer blant annet et 32-bits format, som heter `float` i C og Java, og et 64-bits som heter `double`.

Det 32-bits formatet deler bitene slik:



S – fortegn: 0 for positivt, 1 for negativt. Rent fortegn, ikke toerkomplement.

eksponent – lagret med forskyvning 127: står det 129, er eksponenten $129 - 127 = 2$.

mantisse – sifrene *etter* det ledende ettallet, som ikke lagres.

Figur 2.8: IEEE 754, enkel presisjon. 32 bit fordelt på 1, 8 og 23.

To detaljer i figuren trenger en forklaring.

Forskyvningen. Eksponenten må kunne være negativ, siden vi skal kunne lage små tall. I stedet for toerkomplement bruker IEEE 754 *forskyvning*: man lagrer den virkelige eksponenten pluss 127. Står det 127, betyr det 2^0. Står det 128, betyr det 2^1. Står det 126, betyr det 2^{-1}.

Hvorfor akkurat sånn? Fordi to positive flyttall da kan sammenlignes som om de var vanlige heltall, bit for bit, uten å pakke ut noe. Det gjør maskinvaren enklere. For negative tall må fortegnsbitet håndteres

for seg, siden IEEE 754 bruker rent fortegn og ikke toerkomplement.

Det skjulte bitet. Et normalisert binært tall har alltid formen 1,noe $\times 2^n$. Det ledende sifferet *kan ikke* være noe annet enn 1, for det finnes ikke andre siffer i base 2. Da er det ingen grunn til å lagre det. Standarden tar det for gitt, og de 23 lagrede bitene gir dermed 24 bit reell presisjon. Det svarer til omtrent sju desimalsiffer.

2.10.3 Et tall bygget opp bit for bit

La oss kode 5,75 som en `float`.

Steg 1: skriv tallet binært. Heltallsdelen og brøkdelen tas hver for seg. $5 = 101_2$. Brøkdelen 0,75 er $\frac{1}{2} + \frac{1}{4}$, altså $0{,}11_2$. Til sammen:

$$5{,}75_{10} = 101{,}11_2$$

Steg 2: normaliser. Flytt binærpunktet til det står rett etter det første ettallet, og tell hvor mange plasser du flyttet:

$$101{,}11_2 = 1{,}0111_2 \times 2^2$$

Steg 3: fyll ut feltene.

- Fortegn: tallet er positivt, så $S = 0$.
- Eksponent: den virkelige eksponenten er 2, så vi lagrer $2 + 127 = 129 = 10000001_2$.
- Mantisse: sifrene etter det ledende ettallet er `0111`, resten fylles med nuller.

```
0  10000001  01110000000000000000000
S  eksponent           mantisse
```

Grupperer vi de 32 bitene i firere og slår opp i tabell 2.1 on page 16, får vi `0x40B80000`. Skriver du ut bitmønsteret til `5.75f` i et hvilket som helst språk, er det dette du får.

Den andre veien. Hvilket tall er `1 10000100 10000111100000000000000`?

Fortegnsbitet er 1, så tallet er negativt. Den lagrede eksponenten er $10000100_2 = 132$, så den virkelige er $132 - 127 = 5$. Vi setter det skjulte ettallet foran mantissen og flytter binærpunktet fem plasser til høyre:

$$1{,}100001111_2 \times 2^5 = 110000{,}1111_2$$

Heltallsdelen 110000_2 er 48. Brøkdelen $0{,}1111_2$ er $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} = 0{,}9375$. Svaret er $-48{,}9375$.

2.10.4 De to formatene og spesialverdiene

	Bit	Fortegn	Eksp.	Mantisse	Presisjon
`float`	32	1	8 (forsk. 127)	23	ca. 7 desimalsiffer
`double`	64	1	11 (forsk. 1023)	52	ca. 16 desimalsiffer

Tabell 2.5: De to formatene du kommer til å møte.

Standarden setter av to eksponentverdier til spesialformål. Det er derfor du av og til ser rare ting i utskrifter:

Eksponentfelt	Mantisse	Betydning
bare nuller	bare nuller	± 0 (det finnes både $+0$ og -0)
bare nuller	noe annet	denormalisert tall, svært nær null
bare enere	bare nuller	$\pm\infty$
bare enere	noe annet	`NaN` – *not a number*

Tabell 2.6: Reserverte bitmønstre i IEEE 754.

Deler du 1 på 0 i flyttallsaritmetikk, får du ingen feilmelding. Du får `Infinity`. Deler du 0 på 0, får du `NaN`.

Og `NaN` har en urovekkende egenskap: den er ikke lik seg selv. Uttrykket `x != x` er sant hvis og bare hvis `x` er `NaN`. Det er faktisk den vanligste måten å teste for `NaN` på.

2.10.5 Hvorfor 0,1 ikke finnes

Nå kommer avsnittet som forklarer en hel familie av merkelige feil du kommer til å møte som programmerer.

Prøv å skrive 0,1 binært. Metoden for brøkdeler er gjentatt multiplikasjon: gang med 2, noter heltallsdelen som neste bit, fortsett med resten.

$$
\begin{aligned}
0{,}1 \times 2 &= 0{,}2 && \to 0 \\
0{,}2 \times 2 &= 0{,}4 && \to 0 \\
0{,}4 \times 2 &= 0{,}8 && \to 0 \\
0{,}8 \times 2 &= 1{,}6 && \to 1 \quad \text{(fortsett med 0,6)} \\
0{,}6 \times 2 &= 1{,}2 && \to 1 \quad \text{(fortsett med 0,2)} \\
0{,}2 \times 2 &= 0{,}4 && \to 0 \\
0{,}4 \times 2 &= 0{,}8 && \to 0 \\
0{,}8 \times 2 &= 1{,}6 && \to 1 \\
&\vdots
\end{aligned}
$$

Vi er tilbake på 0,2 etter fem steg. Derfra gjentar alt seg, for alltid. Resultatet er

$$0{,}1_{10} = 0{,}0\overline{0011}_2 = 0{,}00011001100110011\ldots_2$$

Dette tar aldri slutt. Uansett hvor mange bit maskinen har, må den kutte et sted. Det den lagrer er ikke 0,1, men nærmeste tall den klarer å lage. For en `double` er det

$$0{,}1000000000000000055511151231257827\ldots$$

Dette er nøyaktig samme fenomen som at $\frac{1}{3}$ ikke kan skrives ferdig i desimal. $0{,}3333\ldots$ tar heller aldri slutt, og ingen synes det er maskinens feil.

Regelen er den samme i alle baser: *en brøk har et endelig antall siffer i base B bare hvis nevneren ikke har andre primfaktorer enn dem B har.* Nevneren i $\frac{1}{10}$ er 2×5. Femtallet går ikke opp i 2. Derfor er det umulig.

Legg merke til hvem som egentlig er «feil» her. Det er ikke maskinen. Det er vår vane med å skrive tall i base 10.

Advarsel :)

Konsekvensen ser du med én gang du prøver:

```
>>> 0.1 + 0.2
0.30000000000000004

>>> 0.1 + 0.2 == 0.3
False
```

Dette er ikke en feil i Python. Du får det samme i Java, C, JavaScript og alt annet som bruker IEEE 754 – altså praktisk talt alt.

2.10.6 Feil som hoper seg opp

Én avrundingsfeil er liten. Problemet er at de legger seg oppå hverandre.

```
sum = 0.0
for i in range(10):
    sum = sum + 0.1
print(sum)          # 0.9999999999999999
print(sum == 1.0)   # False
```

Ti addisjoner, og vi bommer allerede. Gjør du det ti tusen ganger, blir avviket større.

Verre blir det når du legger sammen tall av svært ulik størrelse. Det minste tallet skyves helt ut av mantissen til det største, og forsvinner sporløst.

For en `float` skjer det allerede ved $2^{24} = 16\,777\,216$. Fra og med det tallet er avstanden mellom to nabo-flyttall større enn 1, så

`16777216f + 1 == 16777216f` er *sant*.

For en `double` ligger grensen på $2^{53} \approx 9 \times 10^{15}$. Det er derfor JavaScript, som bare har `double`, har en konstant som heter `Number.MAX_SAFE_INTEGER` med verdien $2^{53} - 1$. Over den grensen kan språket ikke lenger telle nøyaktig.

Note-to-myself

Sammenlign aldri flyttall med ==.
Sjekk i stedet om de er nær nok hverandre:

```
abs(a - b) < 1e-9
```

Hvor stor toleransen bør være, avhenger av hva du regner på. Det er en vurdering du må gjøre bevisst hver gang.
Og skal du regne med penger: ikke bruk flyttall. Bruk heltall og tell i øre, eller bruk en egen desimaltype. Det var nettopp dette BCD i seksjon 2.6.3 ble laget for.

2.10.7 Avveiningen, én gang til

Hvorfor akkurat 8 eksponentbit og 23 mantissebit? Fordi noen måtte velge.

Flytter du ett bit fra mantissen til eksponenten, dobler du rekkevidden og halverer presisjonen. Flytter du andre veien, blir tallene mer presise og området mindre. Antall representerbare verdier er det samme uansett – det er 2^{32}, og det endrer seg ikke.

Dette er ikke en akademisk øvelse, og valget er ikke tatt en gang for alle. Maskinvare for kunstig intelligens gjør den samme avveiningen i dag, og svarer annerledes enn IEEE gjorde i 1985. Formatet `bfloat16` bruker 16 bit fordelt på 1 fortegnsbit, 8 eksponentbit og 7 mantissebit. Samme rekkevidde som en `float`, men langt grovere presisjon. Det viser seg at nevrale nett tåler upresise tall godt, og halvert bitbredde gir dobbelt så mye data gjennom minnebussen.

Et flyttall er altså en *tilnærming med en kjent og dokumentert feil*, ikke et eksakt tall. Vet du det, er flyttall et av de mest nyttige verktøyene i faget. Glemmer du det, får du feil som er svært vanskelige å finne.

2.11 Bilde

Tall og tekst er diskrete. De består av avgrensede enheter vi kan telle: siffer, bokstaver.

Bilder og lyd er ikke det. Virkeligheten er kontinuerlig. Mellom to farger finnes det uendelig mange mellomtoner, og mellom to øyeblikk finnes det uendelig mange øyeblikk til.

Derfor må vi nå gjøre noe vi ikke har måttet gjøre før: **kaste bort informasjon med vilje**, på en måte vi kan leve med.

2.11.1 Piksler og farge

En skjerm består av millioner av små lyspunkter, **piksler** – fra engelsk *picture element*. Hvert piksel er satt sammen av tre lyskilder: én rød, én grønn og én blå. Ved å variere styrken på de tre kan pikselet vise nesten hvilken som helst farge. Det kalles **RGB**-modellen.

Tankevekker

Her stusser mange, og med god grunn. På skolen lærte du at grunnfargene er rød, gul og blå. Og blander du all maling du har, blir det grumsete brunt, ikke hvitt.
Forskjellen er at maling *trekker fra* lys. Et rødt malingslag ser rødt ut fordi det svelger alt annet og kaster tilbake det røde. Legger du på flere lag, svelges mer, og til slutt er nesten alt borte.
En skjerm *legger til* lys. Den starter svart og skrur på lamper. Skrur du på alle tre for fullt, får du hvitt – akkurat som hvitt lys gjennom et prisme deler seg i alle regnbuens farger.
Derfor er skjermens grunnfarger rød, grønn og blå. Trykkerier, som jobber med blekk og ikke lys, bruker CMYK i stedet: cyan, magenta, gult og svart.

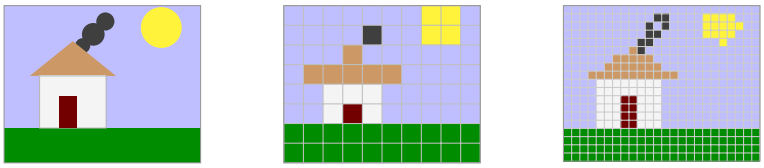
Den vanligste representasjonen gir én byte til hver komponent:

- hver komponent har verdi fra 0 (helt av) til 255 (full styrke)
- ett piksel tar tre byte, altså 24 bit
- det gir $256^3 = 16\,777\,216$ mulige farger

Dette kalles 24 bits *fargedybde*. Det er langt flere farger enn øyet kan skille, og derfor holder det. Noen formater bruker en fjerde byte til gjennomsiktighet, og profesjonell bildebehandling bruker gjerne 16 bit per komponent.

2.11.2 Rutenettet

For å lagre et bilde legger vi et rutenett over det og lagrer én farge per rute. Det er hele ideen.



motivet | 10×8 ruter | 24×19 ruter

Figur 2.9: Samme motiv, samplet på to rutenett. I det grove rutenettet har taket mistet formen, døra er nesten borte, og hele røyken er redusert til én rute. Dobler du antall ruter i begge retninger, firedobles filen.

Se hva som skjer i det grove rutenettet. En rute som dekker både himmel og tak må velge én farge, og velger den som dekker mest. At ruta egentlig inneholdt to farger, er borte for alltid.

Det er dette som kalles sampling. Vi kommer tilbake til akkurat samme grep når vi tar for oss lyd.

2.11.3 Hva koster det?

Regnestykket er enkelt: *antall piksler ganger byte per piksel*.

Et bilde på 1000×1000 piksler med 24 bits farge tar

$$1000 \times 1000 \times 3 = 3\,000\,000 \text{ byte} = 3 \text{ MB}.$$

Husk at MB her betyr 10^6 byte, ikke 2^{20} – det er forskjellen fra seksjon 2.6. I MiB ville samme tall vært 2,86.

Et bilde fra et moderne mobilkamera har gjerne 12 millioner piksler, altså 36 MB rått. Ta hundre bilder, og du har brukt 3,6 GB. Det går ikke. Derfor komprimerer vi.

2.11.4 Fargetabell

Se på motivet i figur 2.9 en gang til. Hvor mange ulike farger er det egentlig i bildet?

Sju: mørkt gress, lys himmel, sol, tak, vegg, dør og røyk.

Men vi bruker 24 bit per piksel, altså plass nok til nesten 17 millioner farger. Det er sløsing på et nivå som roper på en løsning, og løsningen er en **fargetabell**, også kalt en palett.

Ideen er den samme som et fargekart hos malingsforhandleren. I stedet for å beskrive fargen på nytt hver gang, gir du den et nummer. Nummeret er kort. Selve beskrivelsen står på kartet, én gang.

Sju farger trenger tall fra 0 til 6, og det får plass i tre bit.

Regnestykket for et bilde med 348×257 ruter:

- rått: 89 436 piksler × 24 bit = 268 308 byte
- med palett: 89 436 piksler × 3 bit = 33 539 byte, pluss 21 byte til selve tabellen

Vi har spart sju åttedeler av plassen, og *ikke mistet en eneste piksel*. Bildet er bit for bit det samme når det pakkes ut igjen. Det kalles **tapsfri komprimering**.

Men legg merke til forutsetningen. Metoden virker fordi bildet hadde få farger. På et fotografi, der nesten hver piksel har sin egen nyanse, hjelper den ingenting.

2.11.5 Løpelengdekoding

Se på bildet en gang til. Hele den nederste fjerdedelen er gress – rute etter rute etter rute med nøyaktig samme verdi.

Tenk på hvordan en strikkeoppskrift er skrevet. Det står ikke «rett, rett, rett, rett, rett, rett». Det står «6 rette». Samme informasjon, en brøkdel av plassen.

Løpelengdekoding gjør nøyaktig dette med piksler. I stedet for `gress, gress, gress, gress, gress` lagrer vi `gress × 5`. For et bilde med store ensfargede flater er gevinsten dramatisk. Motivet vårt krymper til under tre tusen byte, altså omtrent en hundredel av rå-størrelsen.

Dette er også tapsfritt, og det er en av teknikkene formatene PNG og GIF bruker.

Men forutsetningen igjen: metoden virker fordi nabopiksler ofte er like. Gi den et bilde med tilfeldig støy i hver piksel, og den gjør filen *større*, fordi hver eneste piksel får en teller på kjøpet.

All komprimering bygger på en antakelse om hva slags data den får. Ingen komprimeringsmetode kan gjøre alle filer mindre.

Note-to-myself

Komprimering er ikke gratis, og prisen er ikke bare kompleksitet i koden.
Jo smartere du pakker, jo mer arbeid må til for å pakke ut igjen. En rå fil kan leses rett inn i minnet. En komprimert fil må oversettes underveis, og det koster regnekraft hver eneste gang den åpnes.
For bilder merkes det sjelden – maskinen er rask nok. For video som skal spilles av i sanntid, eller for data som skal leses tusenvis av ganger i sekundet, er det en helt reell avveining.

2.11.6 Komprimering med tap: JPEG

For fotografier holder ikke de tapsfrie metodene. Et fotografi har ingen store ensfargede flater, og ingen liten fargepalett. Ute i naturen er det ikke «her er det lyseblått og her er det grønt» – det er nyanser hele veien.

Løsningen er å slutte å kreve at bildet skal komme tilbake uendret, og i stedet kreve at det skal *se* uendret ut. Det er en helt annen avtale, og det er den **JPEG** inngår.

JPEG bygger på forskning om hva øyet er godt og dårlig til. Vi ser skarpt på lysstyrke og slurvete på farge. Vi legger dårlig merke til fine detaljer i områder med mye struktur. JPEG deler bildet i små blokker, beskriver hvor raskt fargene varierer innenfor hver blokk, og kaster de raskeste variasjonene. Dem ser vi likevel ikke.

Et fotografi tok 40,7 MB i rå rasterform. Lagret som JPEG på full kvalitet ble det 5,5 MB, uten synlig forskjell. Ved 25 prosent kvalitet var det nede i 0,6 MB, og selv da måtte man se etter – det første som

forsvant var de jevne fargeovergangene i himmelen, som ble til synlige trappetrinn.

Legg merke til at JPEG sparte 35 MB allerede på full kvalitet, altså *før* noe ble kastet. Formatet bruker tapsfrie teknikker i tillegg til de tapsbærende. Det er ikke enten eller.

Advarsel :)

Informasjon som er kastet, er kastet. Lagrer du et bilde som JPEG på 25 prosent kvalitet, kommer det aldri tilbake. Uansett hvilket program du åpner det i.
Verre: lagrer du et JPEG-bilde på nytt som JPEG, kastes det litt mer. Redigerer og beskjærer du mellom hver lagring, er bildet merkbart nedbrutt etter noen få runder. Det kalles generasjonstap, og det er derfor arkiver, medisinske bilder og bevismateriale lagres tapsfritt.
Bruk JPEG til fotografier du skal se på. Bruk PNG til strektegninger, skjermbilder og alt du skal redigere videre.

2.11.7 Vektorgrafikk – en helt annen idé

Alt vi har sett til nå er *rastergrafikk*: et rutenett av piksler. Det finnes et alternativ som bygger på en helt annen tanke.

Vektorgrafikk lagrer ikke bildet. Den lagrer *oppskriften på å tegne det*. Flytt til dette punktet. Trekk en linje dit. Tegn en sirkel med denne radien. Fyll flaten med denne fargen.

Fordelene følger direkte. En sirkel trenger tre tall – senter i x og y, pluss radius – uansett hvor stor den er på skjermen. Og siden oppskriften kjøres på nytt hver gang bildet vises, kan du zoome inn så mye du vil uten at det blir hakkete. Maskinen tegner bare sirkelen større.

Ulempen er like tydelig. Du kan ikke skrive en oppskrift på et fotografi av et fjell. Vektorgrafikk passer for logoer, ikoner, diagrammer, arkitekttegninger og skrifttyper. Ikke for noe et kamera har tatt.

Alle figurene i denne boka er for øvrig vektorgrafikk. Det er derfor de fortsatt er skarpe når du zoomer inn i PDF-en.

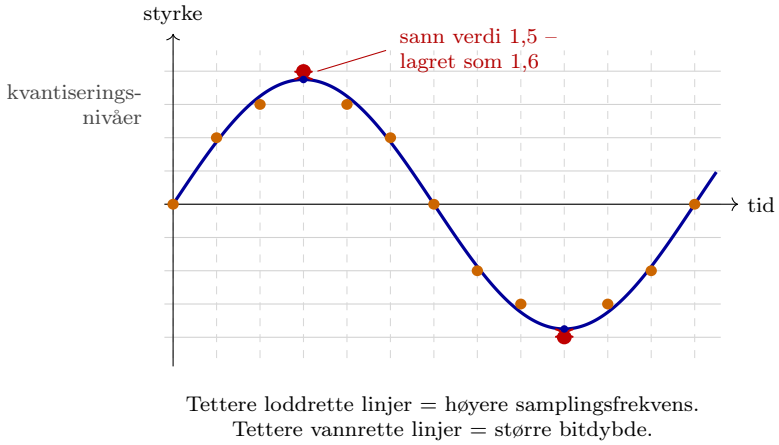
2.12 Lyd

Lyd er trykkbølger i luft. En mikrofon gjør dem om til en varierende elektrisk spenning, og det er den spenningen vi skal lagre.

Signalet er kontinuerlig i to retninger samtidig. Det varierer i tid, og det varierer i styrke. For å få det inn i en datamaskin må vi gjøre begge deler diskrete. De to grepene har hvert sitt navn:

- **Sampling** deler opp *tidsaksen*. Vi måler signalet med jevne mellomrom, og glemmer alt som skjer imellom.

- **Kvantisering** deler opp *styrkeaksen*. Vi har bare et endelig antall nivåer å velge mellom, og hver måling rundes til nærmeste.



Figur 2.10: Sampling og kvantisering. De oransje punktene er det maskinen lagrer. Der kurven treffer mellom to nivåer, går differansen tapt – se de to røde punktene.

2.12.1 Det er samme grep som på bildet

Stopp ved denne figuren et øyeblikk, og sammenlign den med pikselrutenettet i figur 2.9 on page 39. Det er nøyaktig samme operasjon.

I bildet la vi et rutenett over noe kontinuerlig og lagret én verdi per rute. I lyden legger vi et rutenett over noe kontinuerlig og lagrer én verdi per rute. Forskjellen er bare at bildets rutenett går i to romlige retninger, mens lydens går i tid og styrke.

Konsekvensen er den samme begge steder: *det som lå mellom rutene, finnes ikke lenger.* Et finere rutenett gir et bedre resultat og en større fil. Samme handel, to ulike signaler.

2.12.2 Hvor tett må vi sample?

Svaret er ikke «så tett som mulig», og det er ikke en smakssak heller. Det finnes et presist matematisk svar, og det kalles **samplingsteoremet** eller Nyquist-kriteriet:

Tankevekker

For å kunne gjenskape et signal fullstendig må du sample med minst *dobbelt* så høy frekvens som den høyeste frekvensen i signalet.

Det er et sterkt resultat: over den grensen mister du *ingenting.* Sampler du raskere enn det dobbelte, kan det opprinnelige signalet regnes eksakt tilbake.

Et ungt menneske hører opp til omtrent 20 000 Hz. Dobbelt så mye er 40 000, og CD-standarden valgte 44 100 Hz – litt over, for å ha margin til filtrene.

Sampler du for sakte, får du et *feil* signal. Høye frekvenser folder seg ned og dukker opp som lave toner som aldri var der. Det er samme fenomen som gjør at bilhjul kan se ut til å snurre baklengs på film: kameraet tar 24–25 bilder i sekundet, og hjulet går for fort.

2.12.3 CD-en, regnet ut

CD-standarden er verdt å regne gjennom, fordi den viser hvordan tallene henger sammen.

- samplingsfrekvens: 44 100 sampler per sekund
- bitdybde: 16 bit per sample, altså $2^{16} = 65\,536$ nivåer
- kanaler: 2, altså stereo

Datarate:

$$44\,100 \times 16 \times 2 = 1\,411\,200 \text{ bit/s} = 176\,400 \text{ byte/s}$$

En CD rommer 74 minutter, altså 4440 sekunder:

$$4440 \times 176\,400 = 783\,216\,000 \text{ byte} \approx 783 \text{ MB}$$

Note-to-myself

Tallet 783 MB er *ren lyddata*. Feilkorreksjonen er ikke med.
En CD skal tåle riper og fettete fingre, så den lagrer også betydelige mengder feilkorreksjonsdata. De kommer *på toppen* av de 783 megabytene, på det fysiske laget, sammen med rammestruktur og synkroniseringsmønstre. Platen inneholder altså fysisk atskillig mer enn 783 MB.
Det er også derfor en data-CD bare rommer rundt 680 MB der en lyd-CD rommer 783. Den trenger strengere feilkorreksjon enn lyd gjør. En tapt bit i en musikkfil gir et knepp du knapt hører. En tapt bit i et program kan gjøre det ubrukelig, og prisen for den forskjellen er plass.

2.12.4 Komprimering av lyd

Som med bilder finnes begge slag.

Tapsfritt, som FLAC, utnytter mønstre i signalet. To eksempler:

Venstre og høyre kanal i en stereoinnspilling er som regel nesten like. I stedet for å lagre begge, lagrer man den ene pluss *differansen* til den andre. Differansen er et lite tall og trenger få bit.

Samme triks virker i tid. To nabomålinger ligger som regel nær hverandre, siden lyd er en jevn kurve. Så i stedet for hele verdien kan man lagre endringen fra forrige måling.

Tankevekker

Men differanser har en ubehagelig bieffekt. Lagrer du *bare* endringer, må du lese hele filen fra start for å vite hvor du er. Hopper du til midten av en sang, har du en haug med endringer og ingen anelse om hva de er endringer fra.
Løsningen er å legge inn faste punkter der hele verdien står, med jevne mellomrom. Da kan spilleren hoppe til nærmeste slike punkt og lese videre derfra.
Kjenner du igjen problemet? Det er nøyaktig det samme som UTF-8 løser med prefiksbitene i seksjon 2.9. En strøm av data der du må kunne plukke opp tråden hvor som helst, uten å lese alt som kom før.
To helt ulike domener, samme problem, samme type løsning. Det er verdt å merke seg – du kommer til å se dette mønsteret igjen.

Med tap, som MP3 og AAC, går mye lenger. De utnytter begrensninger i hørselen vår:

- vi hører ikke under omtrent 20 Hz eller over omtrent 20 000 Hz
- en kraftig lyd *maskerer* svakere lyder på nærliggende frekvenser – de er der, men vi hører dem ikke
- rett etter en kraftig lyd er øret mindre følsomt en liten stund

Dette kalles *psykoakustikk*. Legg merke til hva algoritmen egentlig gjør: den fjerner det som er uviktig for *et menneske*, ikke det som er uviktig for signalet. Representasjonen er tilpasset mottakeren.

Gevinsten er stor. Der en CD bruker 1411 kbit/s, gir MP3 akseptabel kvalitet på 128–192 kbit/s. Moderne formater som Opus gir CD-liknende kvalitet på rundt 96 kbit/s, altså under en tiendedel. Uten det hadde musikkstrømming ikke vært mulig.

Advarsel :)

Samme regel som for JPEG gjelder her. Skal du jobbe med lyd – ta opp, klippe, mikse – er MP3 et elendig arbeidsformat. Hver gang du lagrer på nytt, kastes mer, og du kommer aldri tilbake til originalen.
Jobb i et ukomprimert format som WAV, eller et tapsfritt som FLAC. Lag MP3 til slutt, når du er ferdig, og bare til folk som skal lytte.

2.13 Oppsummering

Vi har vært gjennom en lang rekke representasjoner. La oss stille samme spørsmål til alle: *hva koster den, og hvor går grensen?*

Representasjon	**Hva den koster**	**Hvor grensa går**
Heltall uten fortegn, n bit	n bit	$2^n - 1$; over det ruller tallet rundt
Toerkomplement, n bit	n bit, halve området går til negative tall	-2^{n-1} til $2^{n-1} - 1$; usymmetrisk, og -2^{n-1} kan ikke negeres
BCD	6 av 16 mønstre kastes; treg aritmetikk	eksakt i desimal, men bruker et helt siffer mer plass ved 20 bit
IEEE 754 `float`	32 bit fordelt 1/8/23	ca. 7 signifikante siffer; 0,1 finnes ikke; over 2^{24} kan den ikke telle
ASCII	7 bit per tegn	128 tegn; ingen æ, ø, å
Latin-1	8 bit per tegn	256 tegn; én region om gangen
UTF-8	1–4 byte per tegn	alle kodepunkter; men tegn er ikke byte, og du kan kutte midt i et tegn
RGB 24 bit	3 byte per piksel	16,7 millioner farger; filen blir stor
Fargetabell	indeks per piksel + tabell	bare så mange farger som tabellen har
Løpelengdekoding	tapsfri, avhenger av dataene	gjør støyete bilder *større*
JPEG	sterk komprimering med tap	informasjonen kommer aldri tilbake; generasjonstap
Sampling og kvantisering	samplingsfrekvens × bitdybde × kanaler	Nyquist setter nedre grense; under den folder frekvenser seg
MP3, AAC, Opus	en tiendedel av CD-raten	tilpasset menneskeøret, ikke signalet

Tabell 2.7: Alle representasjonene i kapitlet, med prisen og grensen.

Ser du på kolonnene, ser du hele kapitlet. Ingen av representasjonene er feil. Hver av dem er et svar på hva som er viktigst akkurat her: presisjon eller plass, rekkevidde eller nøyaktighet.

Og hver av dem har en grense. Ariane 5 traff grensen til et 16-bits heltall. En bank som regner renter i `float`, treffer grensen til IEEE 754. Et skjema som kutter et navn etter 20 byte, treffer grensen til

UTF-8. I alle tre tilfellene virket koden helt til den ikke gjorde det.

Tankevekker

Vil du ha den bærende ideen i sin mest håndfaste form, prøv dette.
Ta en bildefil og be lydspilleren din om å spille den av. Det går helt fint. Det blir neppe vakkert – det høres ut som en fax-maskin i dårlig humør – men maskinen nekter ikke. Den leser bitene og gjør dem om til lyd, slik du ba om.
Gjør det motsatte også. Åpne en musikkfil i et bildeprogram, og du får et bilde. Meningsløst, men et bilde.
Det er tross alt bare bit, alt sammen. Det eneste som skiller et bilde fra en sang, er hva du har bestemt deg for at bitene skal bety.

I neste kapittel tar vi tråden videre på det mest overraskende punktet i figur 2.1 on page 9: at det samme bitmønsteret også kan være en *instruksjon*. Det er ideen som gjør en datamaskin til noe annet enn en kalkulator.

2.14 Videre lesning

Stoffet i dette kapitlet dekkes også i pensumboka av Englander. Det er verdt å lese der, for framstillingen er en annen, og to vinklinger på samme stoff er ofte det som får det til å sitte.

Tallsystemer og konvertering: kapittel 3, særlig 3.2 om telling i ulike baser, 3.4 om konverteringsmetodene og 3.5 om baser som er beslektet. Avsnitt 3.6 om brøker går dypere enn vi har gjort her, og er nyttig hvis du vil forstå flyttall bedre.

Fortegnstall, BCD og flyttall: kapittel 5. Avsnitt 5.1 dekker BCD. Avsnitt 5.2 går gjennom fortegn-og-magnitude, enerkomplement og toerkomplement med en mer matematisk begrunnelse enn vår klokkeanalogi. Avsnitt 5.3 bygger opp flyttallsformatet steg for steg, med et pedagogisk desimalformat før IEEE 754.

Tegn, bilde, lyd og komprimering: kapittel 4, særlig 4.2 om tegnkoder, 4.3 om visuelle data og 4.4 om lyd. Avsnitt 4.7 om internt dataformat er en god parallell til diskusjonen vår om at typen ikke ligger i bitene.

Videre i denne boka: kapittel 3 tar opp igjen tanken om at et bitmønster kan være en instruksjon. Kapittel 5 viser hvordan en krets faktisk legger sammen to toerkomplementtall. Kapittel 6 går helt ned til det fysiske bitet.

Vil du videre på egen hånd: se på Huffman-koding, som bygger en optimal tapsfri kode ut fra hvor ofte hvert symbol forekommer. Og på Shannons kildekodingsteorem, som setter en teoretisk nedre grense for hvor mye en gitt datastrøm kan komprimeres. Det siste er et av de vakreste resultatene i informatikken, og det forklarer hvorfor ingen komprimeringsalgoritme kan gjøre alle filer mindre.

2.15 Refleksjon

Tenk over disse spørsmålene, gjerne sammen med noen andre:

- **Representasjon i hverdagen.** Velg en digital dings du bruker hver dag. Hvor har du merket grensene for representasjonene den bruker – i lydkvalitet, bildekvalitet, batteriprosent, tidsangivelser? Har du noen gang sett et program oppføre seg rart på en måte du nå har en forklaring på?

- **Penger i heltall.** Du skal skrive et system som håndterer transaksjoner i flere valutaer. Hvilke problemer får du hvis du lagrer beløp som flyttall? Og hvilke får du hvis du lagrer dem som heltall antall øre? Hva gjør du med en valuta som ikke har hundredeler, eller en renteberegning som gir en tredjedels øre?

- **Hvem sitt alfabet?** Før Unicode var det vanskelig eller umulig å skrive mange av verdens språk på en datamaskin. Hva slags konsekvenser kan det ha hatt for hvilke språk som ble brukt på nett, og for hvem som fikk delta? Unicode måtte også ta stilling til hvilke tegn som er «samme tegn» på tvers av kinesisk, japansk og koreansk. Hvorfor er slike avgjørelser politiske og ikke bare tekniske?

- **Når er tap greit?** Sammenlign tapsfri og tapsbærende komprimering for fire tilfeller: medisinske røntgenbilder, musikk på en strømmetjeneste, bilder i en rettssak, og et ferieminne på mobilen. Hva bør avgjøre valget i hvert tilfelle, og hvem bør bestemme?

- **Arvede antakelser.** Ariane 5 falt fordi kode som var riktig for én rakett ble brukt på en annen. Kan du komme på et tilfelle fra ditt eget liv eller fra nyhetene der en antakelse om verden lå skjult i et system og sluttet å stemme? Hvordan kunne man oppdaget det på forhånd?

2.16 Selv-test

Test forståelsen din. Spørsmålene står i stigende vanskegrad, og fasiten finnes bakerst i boka.

1. Hva betyr det at et tallsystem er *posisjonelt*? Forklar med utgangspunkt i tallet 407_{10} hva vekten til hvert siffer er.

2. Konverter til desimal: (a) 10110_2, (b) $2\mathrm{F}_{16}$, (c) 351_8.

3. Konverter 200_{10} til (a) binær, (b) heksadesimal, (c) oktal. Bruk divisjonsmetoden i minst ett av tilfellene, og vis mellomregningen.

4. Skriv 0xB3 som binært tall uten å gå veien om desimal. Forklar i én setning hvorfor du kan slippe unna med det.

5. Skriv 1011010_2 som heksadesimalt tall. Hva må du gjøre med den øverste gruppa, og hvorfor er det lov?

6. En Linux-fil har rettighetene `750`. Skriv ut de ni rettighetsbitene, og forklar med ord hva eier, gruppe og andre har lov til.

7. Hvor mange bit trenger du som et minimum for å kunne representere alle tall fra 0 til og med 1000? Begrunn svaret.

8. Hvilket desimaltall er bitmønsteret `11101000` (a) hvis det leses som et heltall uten fortegn, og (b) hvis det leses som toerkomplement?

9. Skriv -37 som et åttebits toerkomplementtall. Vis begge stegene i negeringen.

10. Regn ut $-37 + 20$ i åttebits toerkomplement ved å legge sammen bitmønstrene. Kontroller svaret i desimal.

11. Hvorfor finnes det ingen -0 i toerkomplement, mens det finnes to nuller både i fortegn-og-magnitude og i enerkomplement? Hvilken praktisk fordel gir det?

12. En `short` i Java er 16 bit. Hva blir resultatet av `(short)(30000 + 10000)`, og hvorfor? Forklar hva som skjer med bitene.

13. En venn sier: «Python har ikke overflyt, så da slipper jeg å tenke på det.» Hva er riktig og hva er galt i den påstanden?

14. Hvor mange byte tar strengen `Bodø"` i (a) UTF-8 og (b) UTF-32? Hvor mange tegn består den av? Forklar hvorfor de tre tallene er forskjellige.

15. En byte-strøm inneholder `0xE2 0x82 0xAC`. Hvor mange tegn er dette i UTF-8, og hvordan ser du det uten å slå opp i noen tabell?

16. Du skriver temperaturen 38,5 inn i et program. Forklar hva som lagres hvis verdien havner i en tekststreng, og hva som lagres hvis den havner i en `double`. Hvor er kommaet i hvert av tilfellene?

17. Kod tallet $-6{,}5$ som en 32-bits IEEE 754 `float`. Vis alle stegene: binær form, normalisering, eksponent med forskyvning, og de tre feltene.

18. Hvorfor gir uttrykket `0.1 + 0.2 == 0.3` usant i praktisk talt alle programmeringsspråk? Og hvorfor gir `0.5 + 0.25 == 0.75` sant?

19. Et ukomprimert bilde er 1920×1080 piksler med 24 bits farge. (a) Hvor mange byte tar det? (b) Bildet inneholder bare 16 ulike farger – hvor mye sparer du på å bruke en fargetabell, og er det tapsfritt?

20. En lydopptaker sampler 48 000 ganger i sekundet med 24 bits dybde i stereo. Hvor mange byte blir ett minutt? Og hva er den høyeste frekvensen opptaket kan gjengi korrekt?

21. Både JPEG og MP3 kaster informasjon. Forklar hva de to har til felles i *hvorfor* de kan gjøre det, og gi ett eksempel på en situasjon der ingen av dem burde brukes.

22. Se tilbake på figur 2.1 on page 9. Du får utdelt en fil på disken, og får vite at den er 4 byte stor. Hva kan du si om innholdet, og hva kan du ikke si? Hvor må opplysningen komme fra?