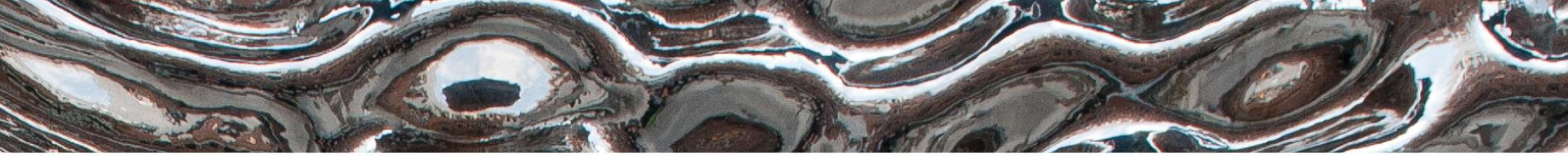
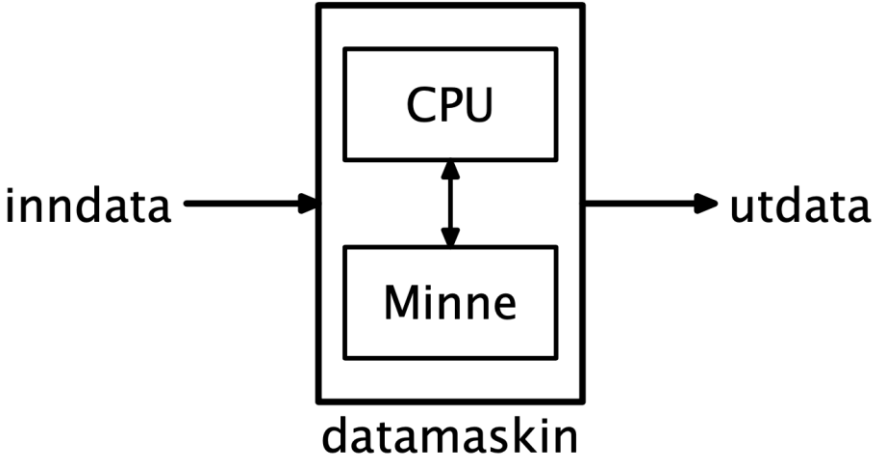
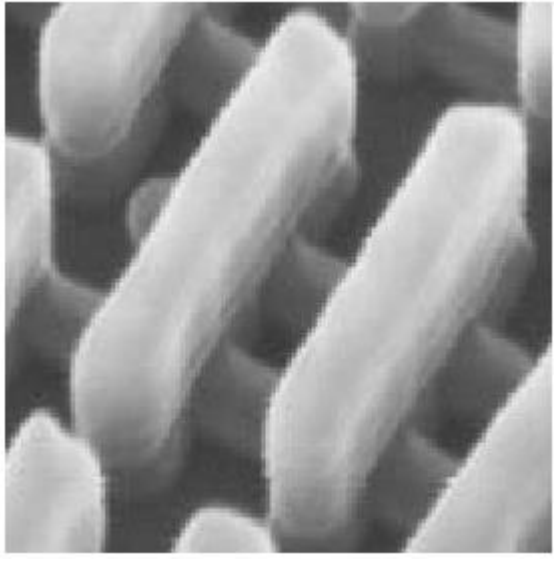


UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Digital representasjon

Kristoffer Robin Stokke (krisrst@ifi.uio.no)
Foiler er basert på materiale fra Dag Langmyhr (dag@ifi.uio.no)


CPU
inndata
utdata
Minne
datamaskin



Digital representasjon – alt er bit!

I dag: Hvordan representerer og lagrer vi

- Tall
- Tekst
- Bilder
- Lyd

som bit i datamaskinen?

Hva er en bit, egentlig?

Bit er en slags forkortelse for *binary digit*, og representerer «noe» som har to tilstander, som f.eks.

0	1
False	True
Rød	Grønn
Lys av	Lys på
0 volt	5 volt

Binære tall

For å kunne bruke bit (0 og 1) som tall, må vi telle *binært*. Dette gjøres i bunn og grunn på samme måte som når vi teller desimalt (vanlig):

- Øk siste siffer i tallet med 1
- Hvis det ikke er flere sifre, sett siste til 0, og gjenta for sifferet til venstre

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

Notasjon

- Noen ganger kan man være i tvil om hvilken base som benyttes; da skriver vi det eksplisitt på:
- 1001 – er det et binærtall eller et desimaltall?
- 1001_2 og 1001_{10} gjør skillet tydelig
- $1001_2 = 9_{10}$

OBS: Selve basen skrives alltid i det desimale systemet

Hvordan fungerer tallsystemer egentlig?

- Moderne tallsystemer er posisjonsbaserte, og sifrene har vekt i henhold til posisjonen.

- I vårt vanlige desimale tallsystem har posisjonene vekt
  1 10 100 1000 osv.
  Dette tilsvarer
  10^0 10^1 10^2 10^3 osv.

(Notasjonen a^n («a i n-te») betyr a * a * ... * a, n ganger.)

Hvordan fungerer tallsystemer egentlig?

- I det binære tallsystemet har posisjonen tilsvarende vekt
  1 2 4 8 osv.
  Dette tilsvarer
  2^0 2^1 2^2 2^3 osv.

Fra Binær til Desimal

- Løsningen her er å gang hvert siffer med sin egen vekt og summere opp.

Dette er posisjonen til sifferne:	4	3	2	1	0
La oss si at vi har følgende binære tall:	1	1	0	0	1
Dette er vekten til posisjonene:	2^4	2^3	2^2	2^1	2^0
Gang hvert siffer med sin vekt:	$1*2^4$	$1*2^3$	$0*2^2$	$0*2^1$	$1*2^0$
Det blir:	16	8	0	0	1

$$16 + 8 + 0 + 0 + 1 = 25$$

Hva så med motsatt? Desimalt til binært

- Løsningen er å dele på grunntallet 2 og se på resten:

Verdi	Rest
25	1
12	0
6	0
3	1
1	1
0	

Svaret leser vi nedenfra: $25_{10} = 11001_2$

Heksadesimal notasjon

Det er lett å gjøre feil når man jobber med binære tall:

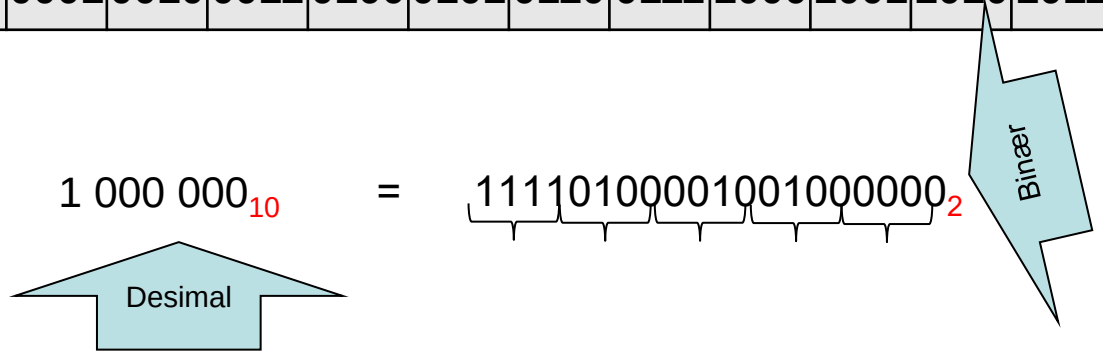
$$1\ 000\ 000_{10} =$$

$$11110100001001000000_2$$

Heksadesimal notasjon gjør livet lettere!

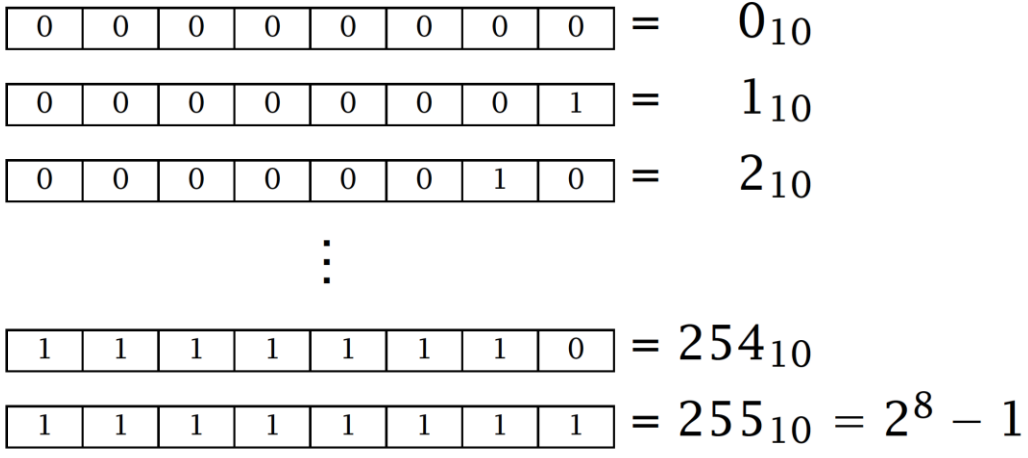
Et tallsystem med base 16; eller 16 forskjellige siffer

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



1 byte er 8 bit

I en datamaskin kan vi ikke hente ut 1 og 1 bit; i stedet lagrer vi dem som byte (dvs grupper av 8 bit):



Vi kan også lagre tall i 2, 4 eller 8 byte satt sammen.

Negative tall

- For å kunne lagre både negative og positive tall, må vi bestemme oss for en representasjon av dette.

- Dette gjør vi på en form kalt 2-er-komplement:

  Øverste bit (gjerne kalt fortegnsbit) tolkes som det negative av den verdien det ellers ville hatt.

Positive tall

0	0	0	0	0	0	0	0	=	0_{10}
0	0	0	0	0	0	0	1	=	1_{10}
0	0	0	0	0	0	1	0	=	2_{10}
⋮									
0	1	1	1	1	1	1	0	=	126_{10}
0	1	1	1	1	1	1	1	=	127_{10}

Negative tall

1	1	1	1	1	1	1	1	=	-1_{10}
1	1	1	1	1	1	1	0	=	-2_{10}
1	1	1	1	1	1	0	1	=	-3_{10}
⋮									
1	0	0	0	0	0	0	1	=	-127_{10}
1	0	0	0	0	0	0	0	=	-128_{10}

Oppsummert

Når vi lagrer heltall på binær form må vi angi

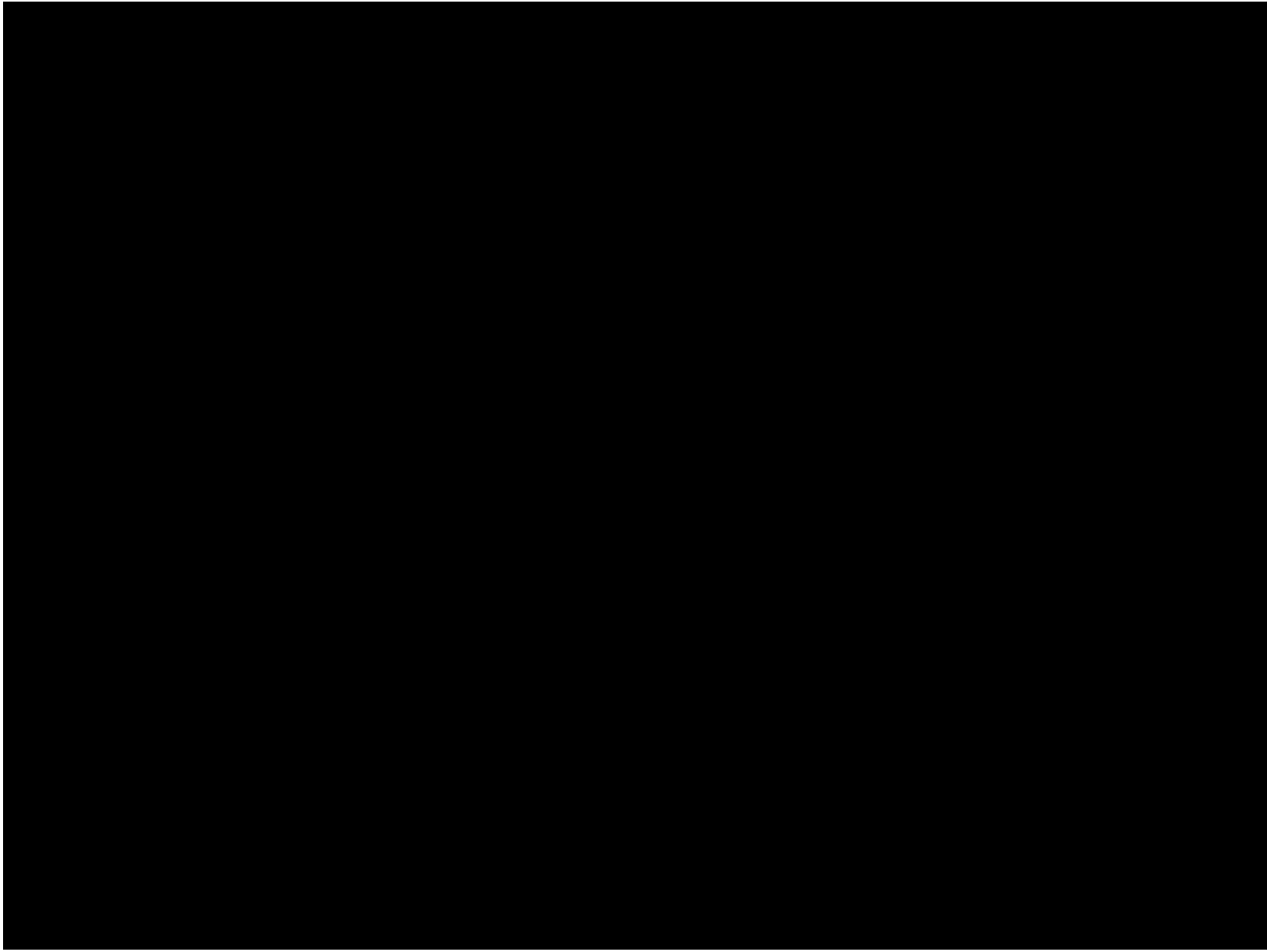
- Hvor mange bit vi bruker
- Kan tallene være negative?
- I så fall, hvordan representerer vi negative tall?

Heltall i Java

Java tilbyr følgende datatyper for å lagre heltall (hvor negative tall bruker 2-er-komplement):

byte	(1 byte, 8 bit)	-128 til 127
short	(2 byte, 16 bit)	-32 768 til 32 767
int	(4 byte, 32 bit)	-2 147 483 648 til 2 147 483 647
long	(8 byte, 64 bit)	-9 223 372 036 854 775 808 til 9 223 372 036 854 775 807

Noen språk, slik som C og C++ har også **uint**, **ulong** etc, for kun positive tall



Overflyt i programmeringsspråk

```java
class Overflyt {
    public static void main(String arg[]) {
        int v = 1000, v2 = v*v, v4 = v2*v2;
        System.out.println("v=" + v + ", v2=" + v2 + " og v4=" + v4);
    }
}
```

```
(base) Brainslug:Temp eyvinda$ java Overflyt
v=1000, v2=1000000 og v4=-727379968
```

Hva er det som skjer her? Hvorfor blir svaret feil?

```
int v = 1000, v2 = v*v, v4 = v2*v2;
```

En **int** i Java har plass til 4 byte (32 bit)

Hva er det som skjer her? Hvorfor blir svaret feil?

```
int v = 1000, v2 = v*v, v4 = v2*v2;
```

En **int** i Java har plass til 4 byte (32 bit)

v: 1 000 =

00000000	00000000	00000011	11101000

Hva er det som skjer her? Hvorfor blir svaret feil?

```
int v = 1000, v2 = v*v, v4 = v2*v2;
```

En **int** i Java har plass til 4 byte (32 bit)

v:	1 000 =	00000000	00000000	00000011	11101000
v2:	1 000 000 =	00000000	00001111	01000010	01000000

Hva er det som skjer her? Hvorfor blir svaret feil?

```
int v = 1000, v2 = v*v, v4 = v2*v2;
```

En **int** i Java har plass til 4 byte (32 bit)

v:	1 000 =	00000000	00000000	00000011	11101000	
v2:	1 000 000 =	00000000	00001111	01000010	01000000	
v4:	1 000 000 000 000 =	11101000	11010100	10100101	00010000	00000000 = -727379968

Hva kan gjøres?

```
class IkkeOverflyt {
    public static void main(String arg[]) {
        long v = 1000, v2 = v*v, v4 = v2*v2;
        System.out.println("v=" + v +", v2=" + v2 + " og v4=" + v4);
    }
}
```

```
(base) Brainslug:Temp eyvinda$ java IkkeOverflyt
v=1000, v2=1000000 og v4=1000000000000
```

Dette var Java, hva så med Python?

```
Python 3.7.4 (default, Aug 13 2019, 15:17:50)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 1000 * 1000
1000000
>>> _ * _
1000000000000
>>> _ * _
1000000000000000000000000
>>> _ * _
1000000000000000000000000000000000000000000000000
>>> _ * _
1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
>>> 
```

- Python setter dynamisk av mer plass når det trengs, så programmereren ikke trenger å bekymre seg for dette
- Men: mange Python-bibliotek er ikke skrevet i Python, men i C, og da har vi samme problemet som i Java

Av og til strekker ikke store heltall til for alt

Hva med disse tallene?

- Andromedagalaksen er 24 029 742 100 000 000 000 km unna.
- $\pi \approx 3{,}14159265$
- Et H-atom er 0,000 000 096 mm stort.

- Vi trenger altså ikke bare heltall, men også tall som:
  - kan ha veldig små og veldig store verdier
  - kan ha desimaler
  - ikke behøver å være helt nøyaktige.

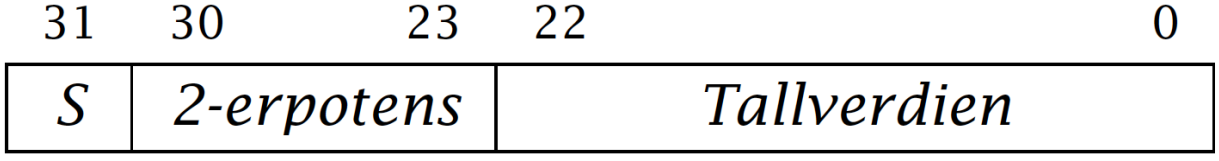
Flyttall er løsningen

Løsningen i det desimale tallsystemet er å oppgi tallet med 10-erpotens:

- Andromedagalaksen er $2{,}4 * 10^{19}$ km unna.
- $\pi \approx 3{,}14159265$
- Et H-atom er $9{,}6 * 10^{-8}$ mm stort.

Så kan vi oppgi den justerte tallverdien (9,6) og 10-erpotensen (-8) hver for seg.

På en datamaskin gjør vi det samme, men vi bruker 2-erpotenser



Flyttall i programmeringsspråk

I Java (og de fleste andre programmeringsspråk) har vi disse flyttallene:

			Minste	Største
float	32 bit	7 sifre	$1{,}2 \cdot 10^{-38}$	$3{,}4 \cdot 10^{38}$
double	64 bit	16 sifre	$2{,}2 \cdot 10^{-308}$	$1{,}8 \cdot 10^{308}$

Python har bare en **double**, men kaller den **float**.

Representasjon av tekst

Hvordan lagrer vi tegn i datamaskinen?

- Husk at maskinen i utgangspunktet bare kjenner til 0 og 1

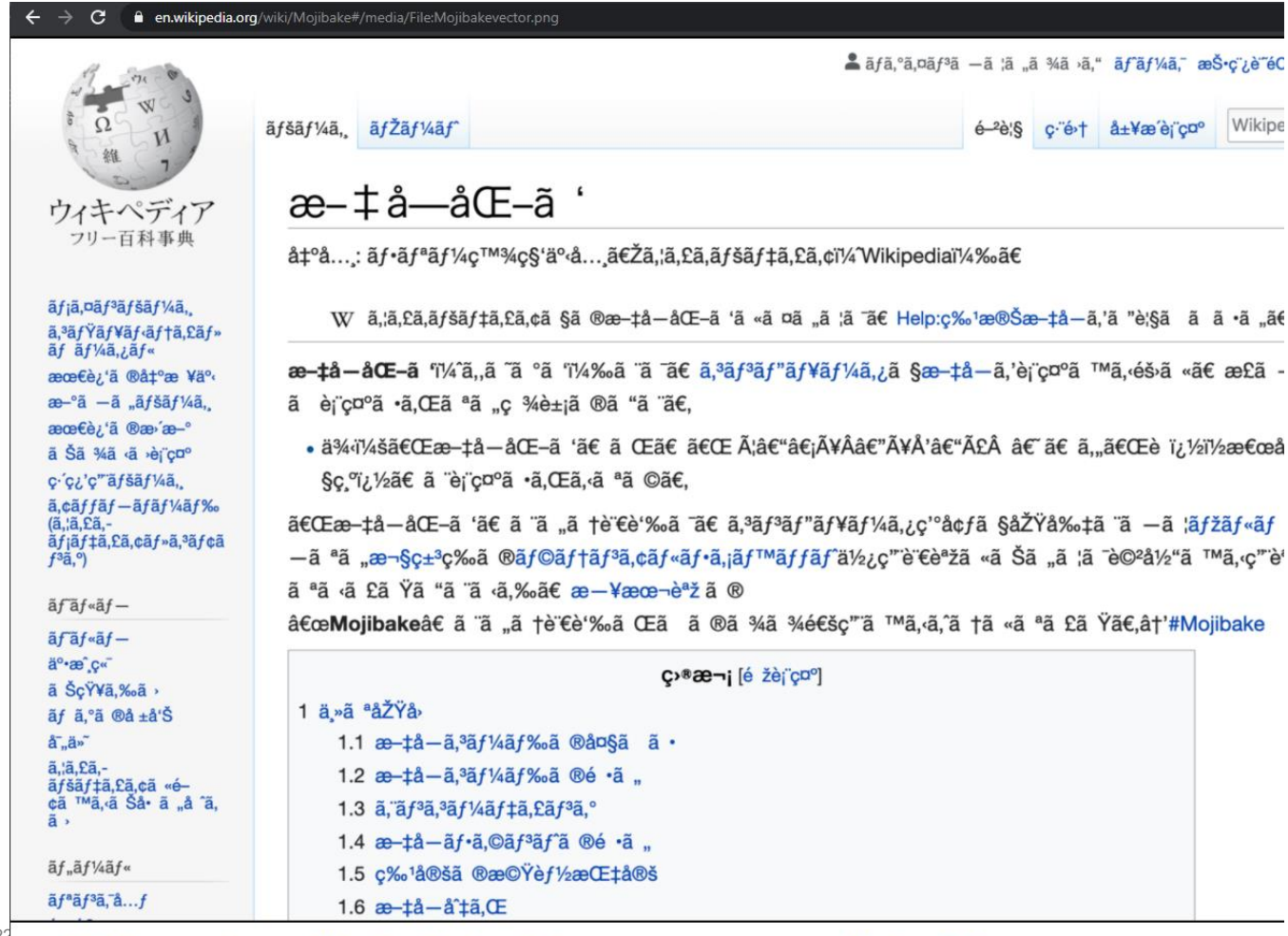
Er ikke det enkelt da?

Sett opp en tabell over tegn vi trenger og gi hvert tegn et nummer:

A=1, B=2, . . .

Så lagrer vi det nummeret for hver bokstav.

Problem: hva om ikke alle bruker samme tabell?

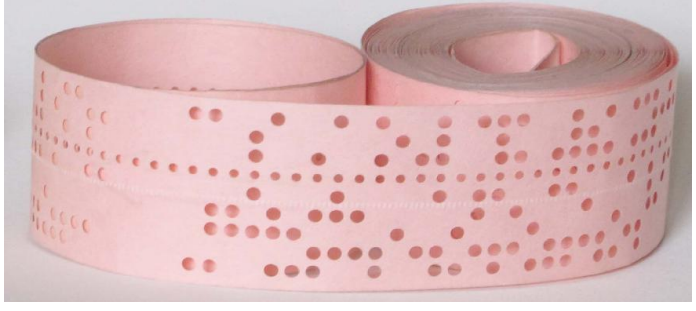


The UTF-8-encoded Japanese Wikipedia article for *mojibake* as displayed if interpreted as Windows-1252 encoding.

Overføring av data

- Tidlig i datamaskinhistorien hadde hver maskintype sin egen tabell
- Men etter hvert ble det behov for at maskiner kunne kommunisere med hverandre

Magnetbånd



Hullbånd

ASCII

Første vellykkede forsøk på standardisering var ASCII (American Standard Code for Information Interchange) i 1963; den har 128 tegn:

ASCII Code Chart

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

ASCII var vellykket

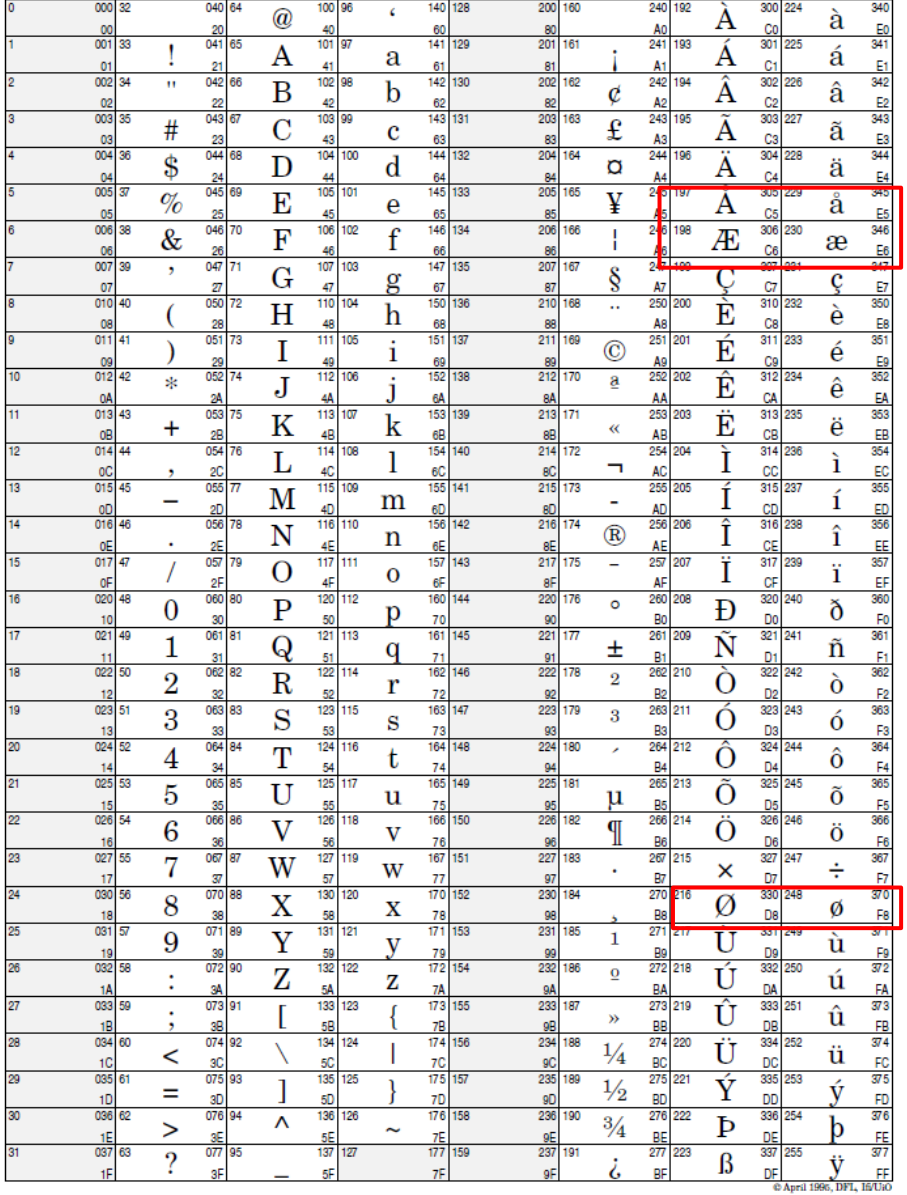
+ ASCII ble etter hvert brukt av de fleste
+ Det er lett å sjekke om et tegn er et siffer eller en bokstav
+ Det er lett å konvertere fra liten til stor bokstav

- Mangler ÆØÅ og mange andre bokstaver og tegn

Det siste ga opphav til et utall av lokale varianter der for eksempel **[\]{|}** ble erstattet av **ÆØÅæøå**. Dette førte til «Gj|vik-syndromet».

Latin-1 / ISO 8859-1

- En klart bedre løsning ble ISO 8859-1 (Latin-1) med 256 tegn.
- Den (og varianter) er ennå i bruk.
- Hvert tegn bruker 1 byte (8 bit)
- Bra i Norge og andre land, men fremdeles ikke en fullverdi erstatning



Unicode

- Den «ordentlige» løsningen på problemet
- Omfatter alle skriftspråk i verden som er, eller har vært, i bruk
- Er stort sett ferdig

π Я 音 æ∞

- Det er plass til drøyt 1 000 000 tegn.
- Litt over 143 000 tegn er foreløpig definert.
- Det meste av nyere programvare støtter Unicode

face-affection

№	Code	Sample	CLDR Short Name
14	U+1F970		smiling face with hearts
15	U+1F60D		smiling face with heart-eyes
16	U+1F929		star-struck
17	U+1F618		face blowing a kiss
18	U+1F617		kissing face
19	U+263A		smiling face
20	U+1F61A		kissing face with closed eyes
21	U+1F619		kissing face with smiling eyes
22	U+1F972		⊙ smiling face with tear

face-tongue

№	Code	Sample	CLDR Short Name
23	U+1F60B		face savoring food
24	U+1F61B		face with tongue
25	U+1F61C		winking face with tongue
26	U+1F92A		zany face

Hvordan representerer vi dette?

Hvordan lagre Unicode-tekst uten å bruke for mye plass på disk eller over nettet?
UTF-8 bruker fra 1 til 4 byte til å lagre et tegn:

Single byte encoding matches ASCII
'A' = 1000001_2

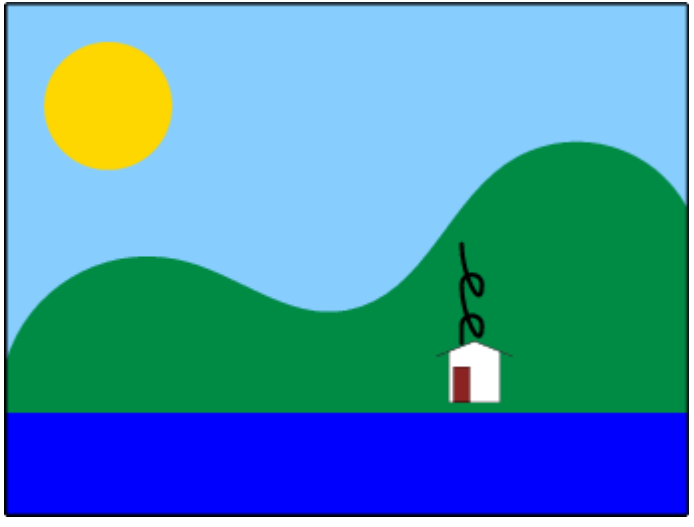
A	U+0041	01000001
Å	U+00C5	11000011 10000101
€	U+20AC	11100010 10000010 10101100
𐎄	U+10384	11110000 10010000 10001110 10000100

01_2 1-byte sequence
110_2 2-byte sequence
1110_2 3-byte sequance
11110_2 4-byte sequence

Trailing bytes always begin with '10_2'

Bilder

Dag har laget dette vakre bildet:



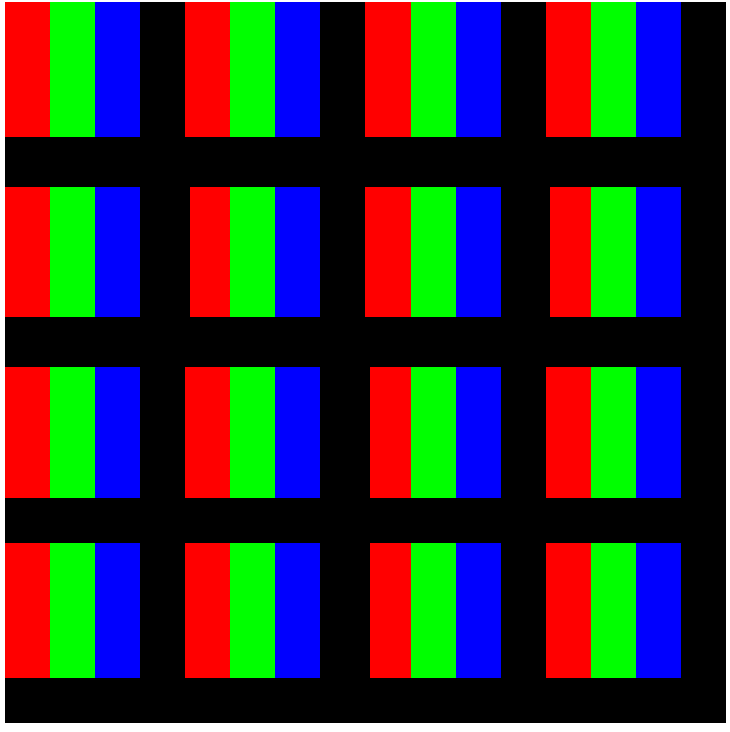
Hvordan kan vi representere dette som bit i en datamaskin?

Pixler - bildepunkter

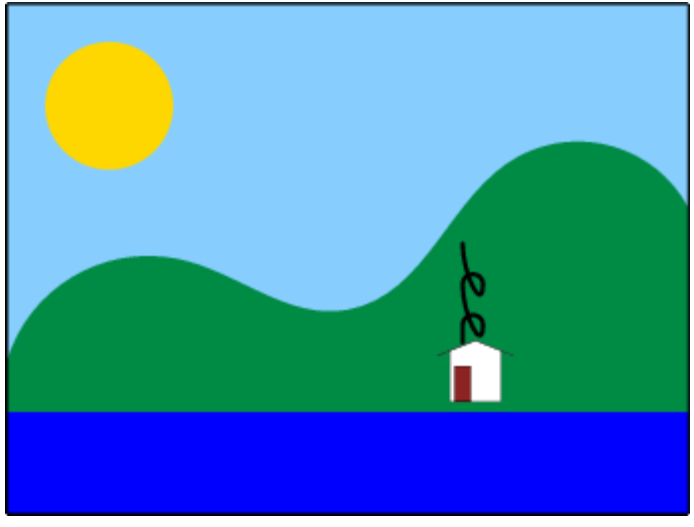
På en fargeskjerm består hvert bildepunkt («pixel») av tre farger:

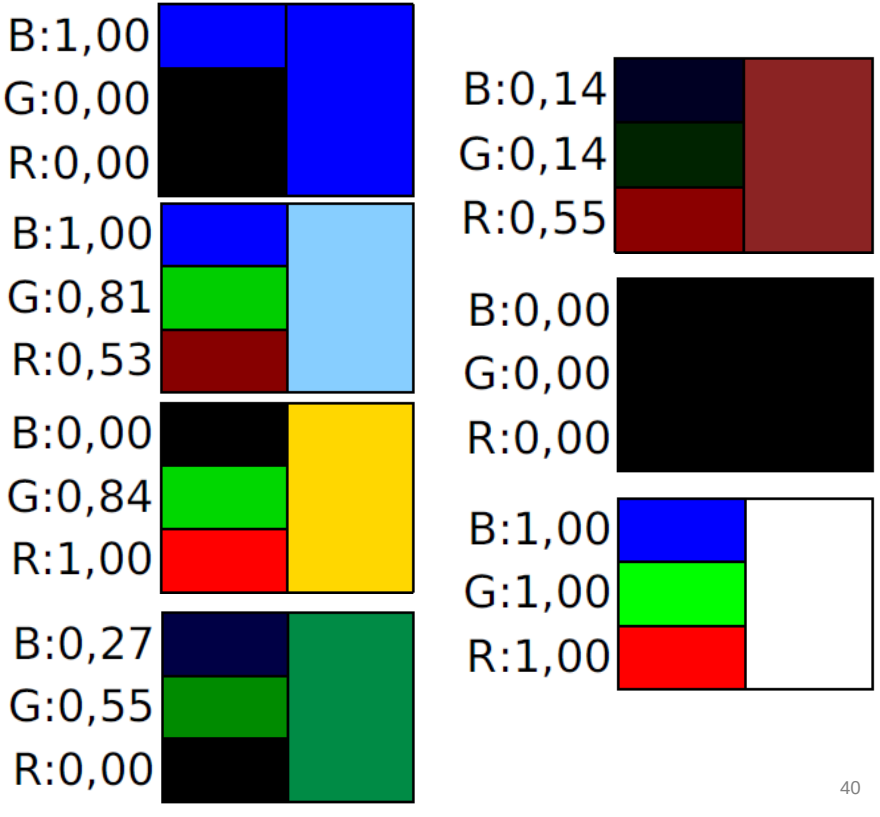
- Rød
- Grønn
- Blå

som kan lyse sterkt eller svakt.



Hvilke farger trenger vi for å lagre Dags bilde?

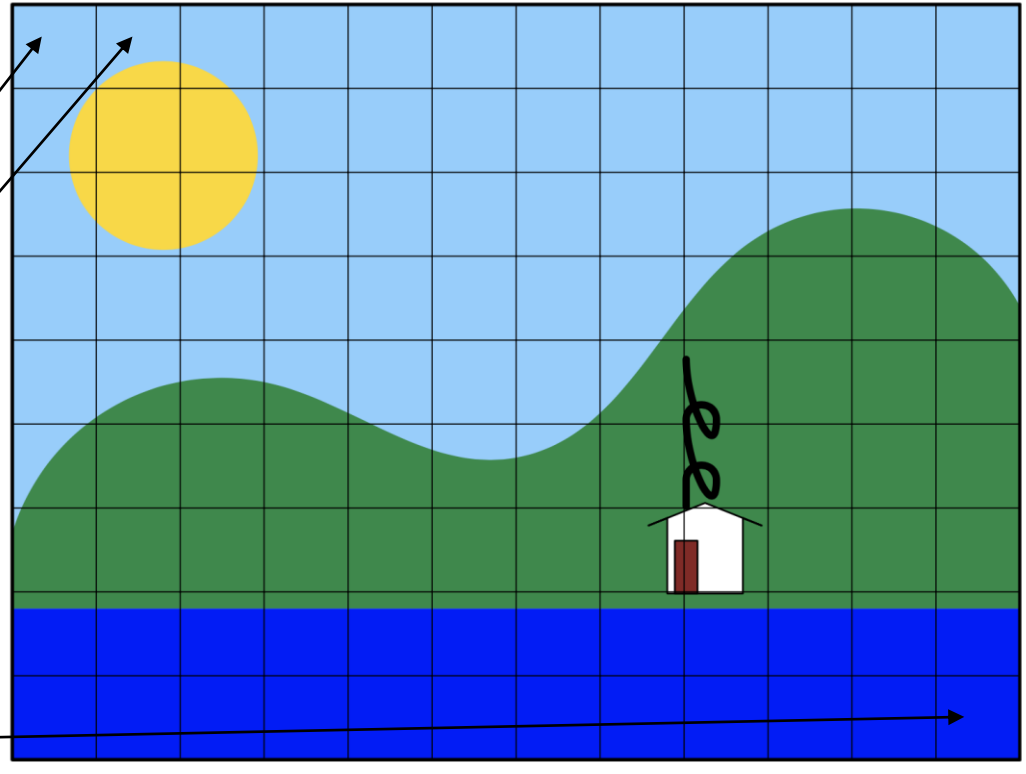




For å lagre et bilde må vi dele det inn i passelige biter som kan representeres binært

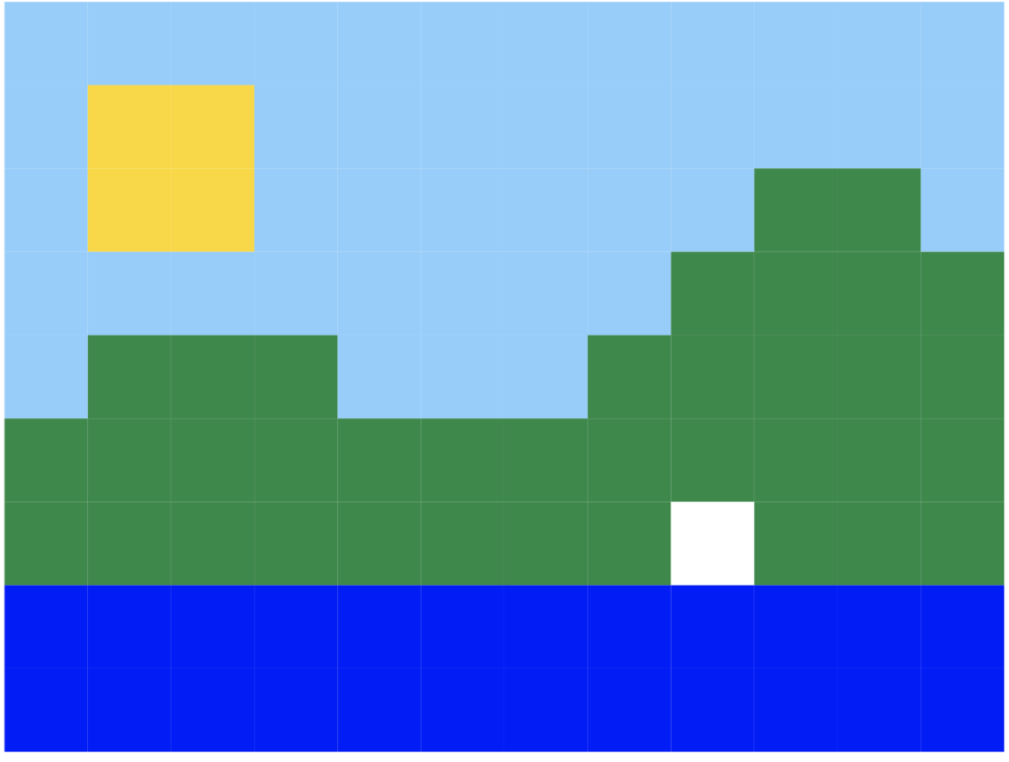
Vi legger et rutenett på bildet, og noterer mengden R, G og B i hver rute. Om vi bruker 1 byte til hver fargemengde, får vi:

R	G	B
135	206	255
135	206	255
135	206	255
...		
0	0	255

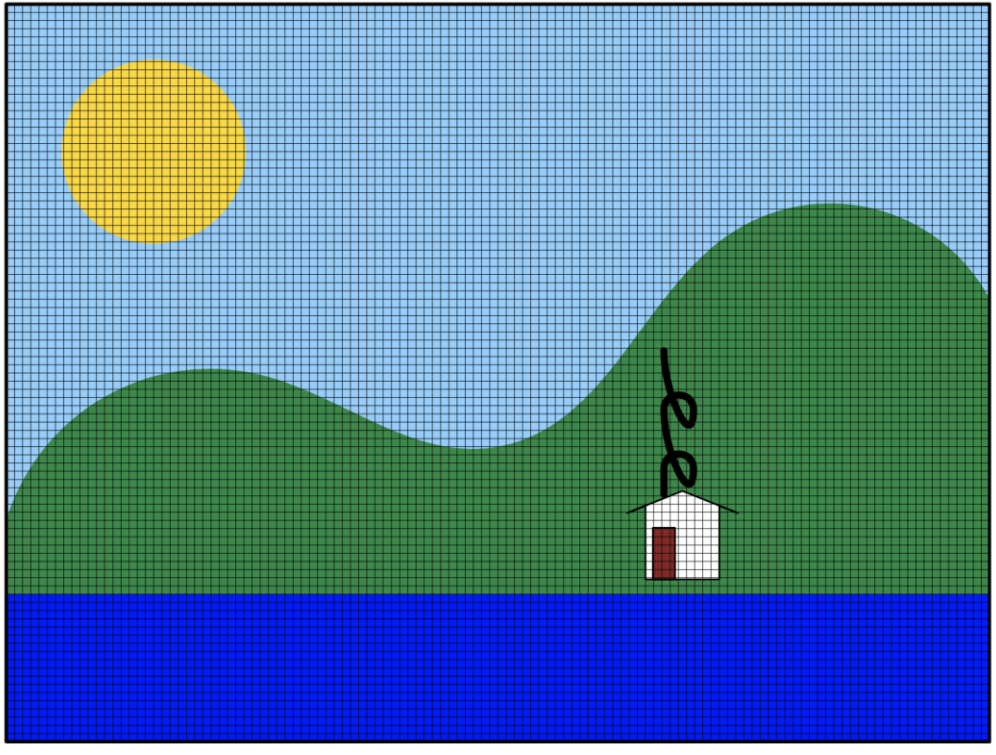


Slike rutenett-representasjoner kalles *rasterbilder*

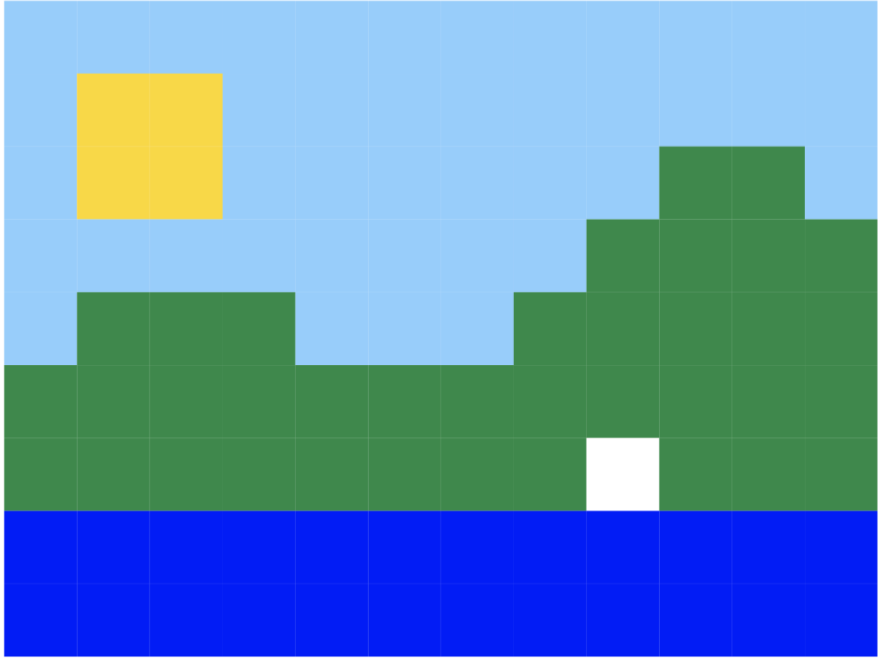
Og voila...



Vi bør kanskje prøve med et mer finmasket rutenett

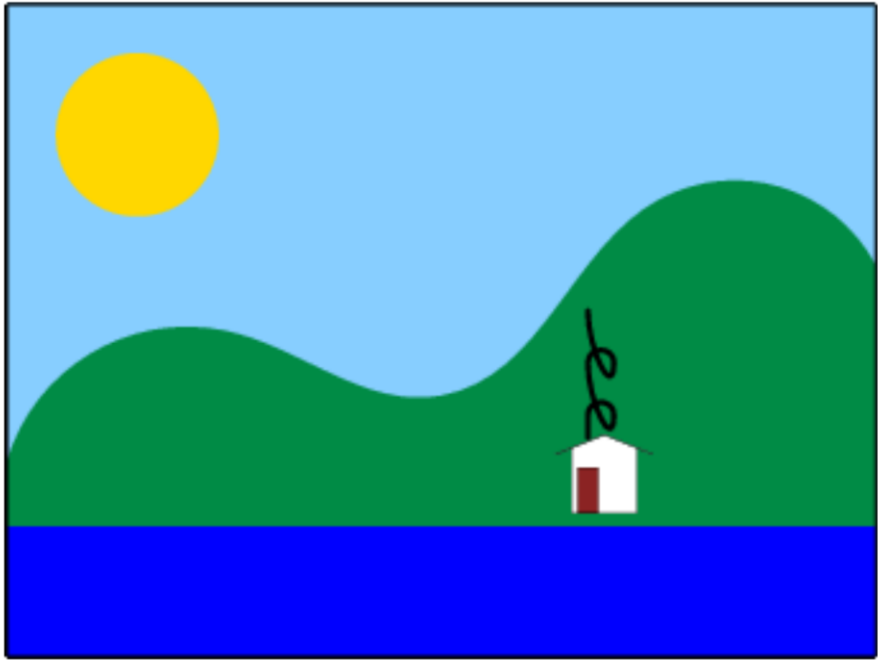


12×9 piksler



324 byte

348×257 piksler



363 682 byte

Kan vi klare oss med mindre plass?

- Har dere noen idéer?

Kan vi klare oss med mindre plass?

- Vi kan lage en **fargetabell**
- Vi bruker 3 byte = 24 bit til hvert piksel, men vi har bare *7 ulike farger* i bildet.
- Da kan vi sette opp en tabell, og da trenger vi bare *3 bit* per piksel.

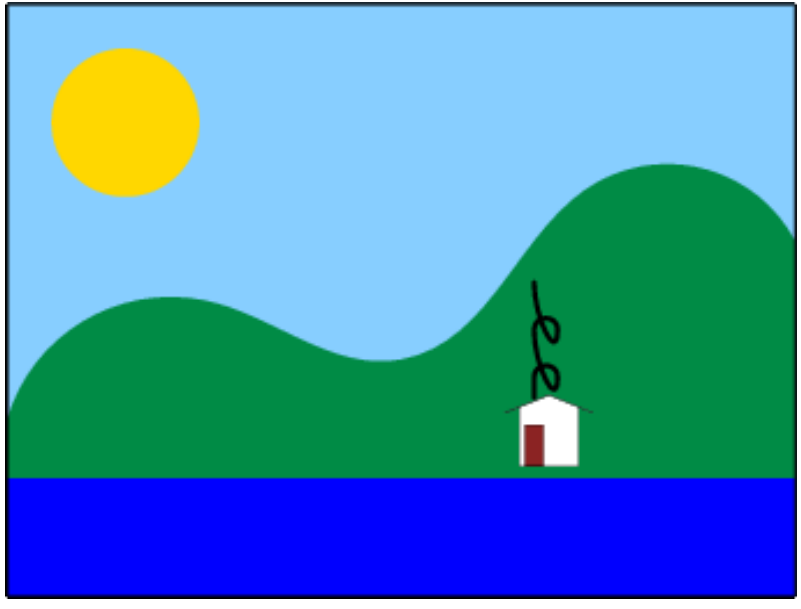
0	0 0 255	mørkeblå
1	135 206 255	lyseblå
2	255 214 0	gul
3	0 140 69	skoggrønn
4	140 36 36	brun
5	0 0 0	svart
6	1 1 1	hvit

363 682 byte ⇒ 45 483 byte

Kan vi være enda smartere?

- Det er mange helt like piksler etter hverandre her
- Vi kan lagre fargen, og hvor mange piksler det er etter hverandre

363 682 byte ⇒
45 483 byte ⇒
2 917 byte



- Formater som PNG og GIF bruker slike teknikker, ofte kalt «*run length-koding*».

Fotografier

- På fotografier er det sjeldnere brå overganger.
- Vi mennesker kan bare skjelne et begrenset antall nyanser.
- Vi søker automatisk etter mønstre.

Dette kan vi utnytte!

JPEG-formatet

- JPEG benytter dette til å lage en komprimert versjon av bildet. Vi kan få ytterligere reduksjon ved å senke kvaliteten.

Ekte rasterbilde	40,7 MB
JPEG 100%	5,5 MB
JPEG 50%	0,94 MB
JPEG 25%	0,61 MB
JPEG 10%	0,34 MB
JPEG 5%	0,24 MB

- Dette er komprimering med **tap**. Det er umulig å komme tilbake til det opprinnelige bildet.

JPEG 100%, 5,5MB

Bilde tatt av Dag Langmyhr

JPEG 50%, 0,94MB

JPEG 25%, 0,61MB

JPEG 10%, 0,34MB

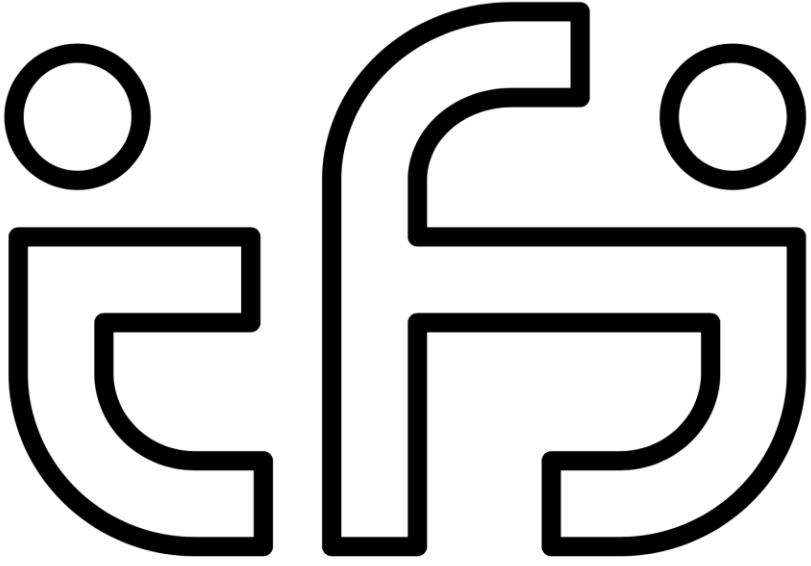
JPEG 5%, 0,24MB

Kvalitet 100%

Kvalitet 5%

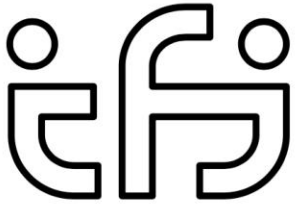
Vektorgrafikk

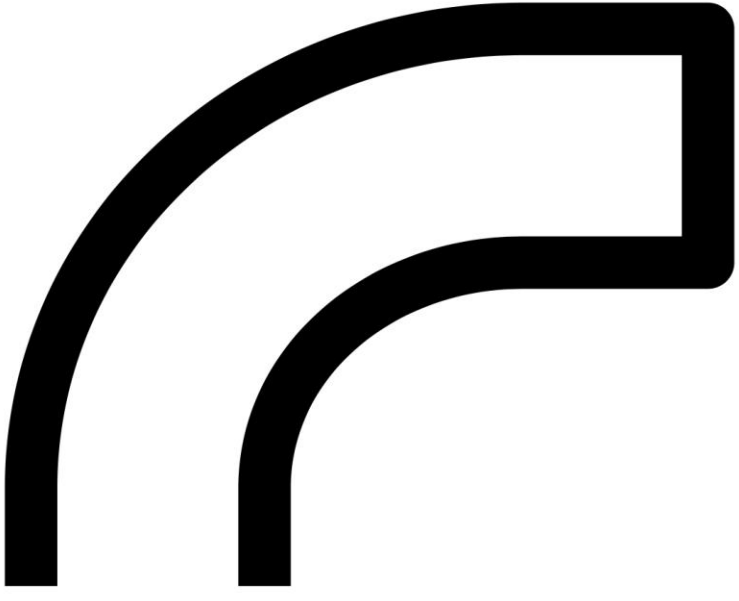
```
newpath
38.4094 2.83464 moveto
38.4094 15.44878 lineto
28.3464 15.44878 lineto
21.03242 15.44878 14.93857
21.1286 14.93857 28.3464 curveto
14.93857 35.77315 lineto
36.56685 35.77315 lineto
36.56685 48.38728 lineto
2.32443 48.38728 lineto
2.32443 28.3464 lineto
2.32443 14.16295 14.0667
2.83464 28.3464 2.83464 curveto
closepath stroke
```



Raster

Vektor

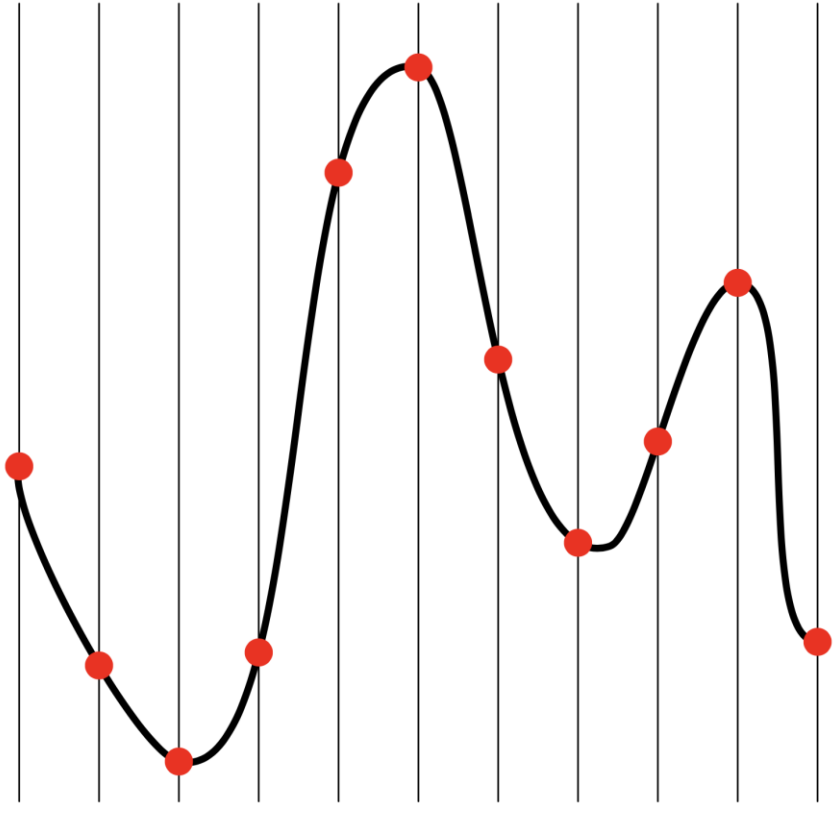




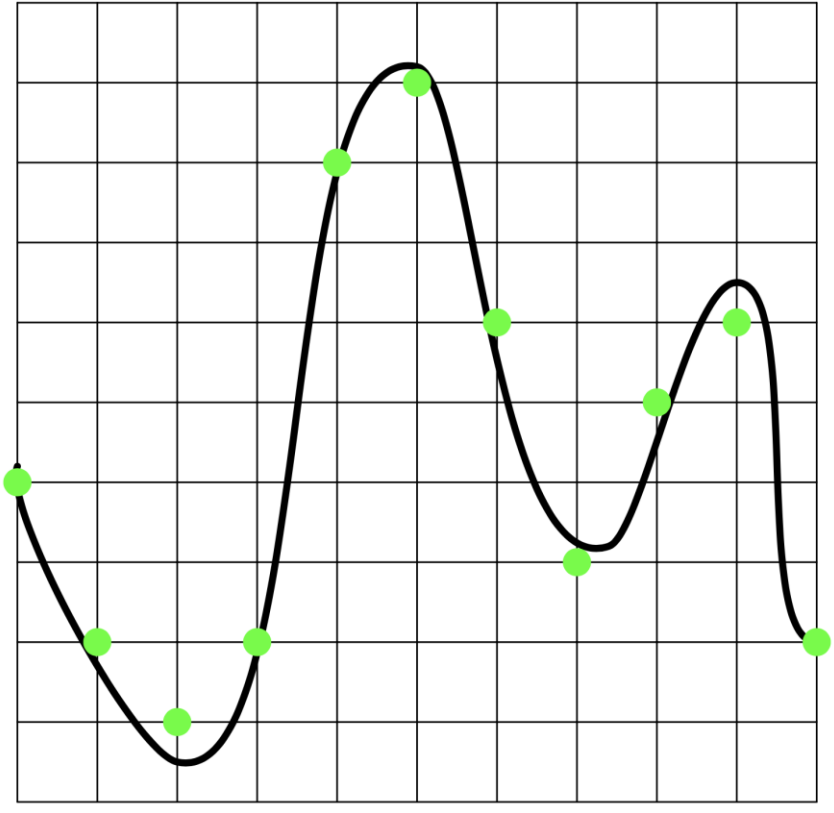
Hvordan lagre lyd?

Lyd er bølger i luften, som kan overføres som strøm:

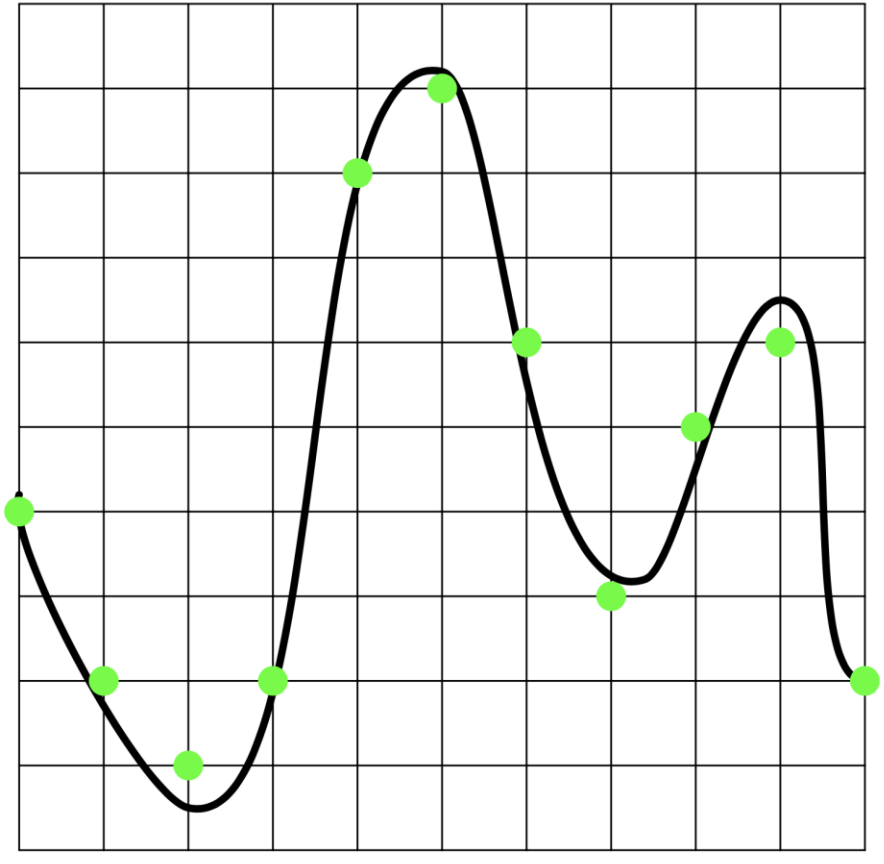
Vi kan lagre lyden ved å måle strømmen med jevne mellomrom:



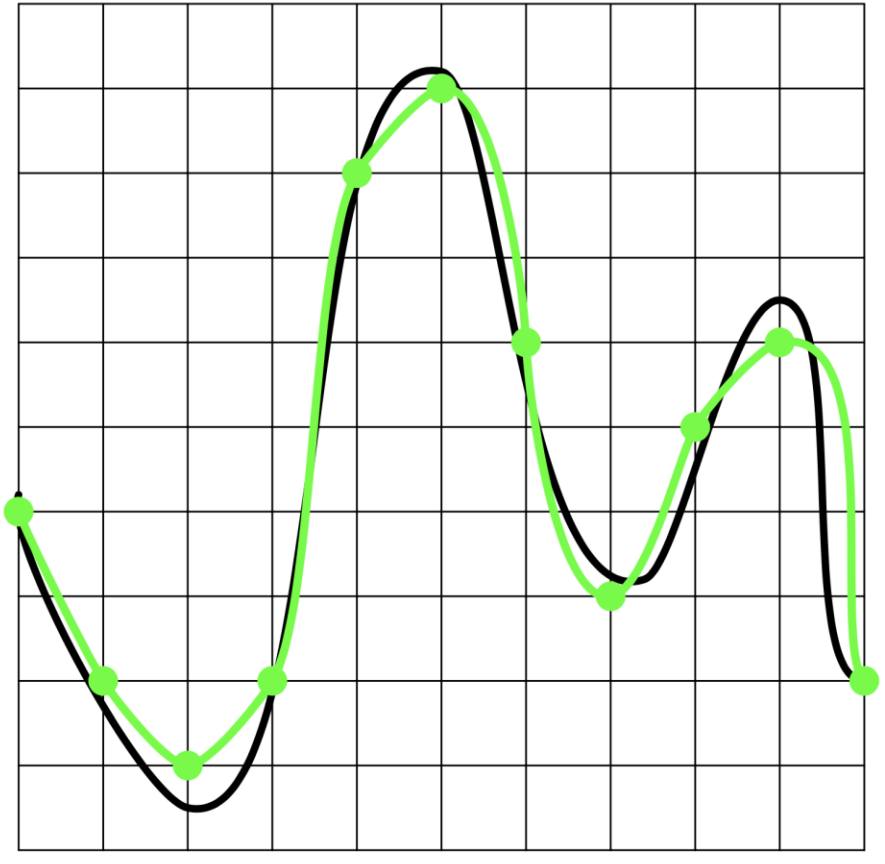
Men for å lagre digitalt må vi måle styrken i faste intervaller:



Med disse målingene



kan vi gjenskape lyden (men ikke helt nøyaktig):



Lydkvalitet

Kvaliteten på lyd representert digitalt avhenger av

- Hvor ofte vi måler
- Hvor mange trinn vi har

- En vanlig CD har 74 minutter spilletid med god lyd: 44 100 målinger per sekund
  - 2^{16} = 65 536 intervaller (dvs 2 byte)
  - 2 kanaler
  - + 37% feilkorreksjonsdata

  En CD må derfor ha plass til 783 MB.

Vårt sedvanlige spørsmål: kan vi spare plass?

- Hva tenker dere?

Vårt sedvanlige spørsmål: kan vi spare plass?

- Høyre og venstre kanal er stort sett nesten like. Det er lurere å lagre én kanal samt forskjellen.
- I stedet for å lagre hver måling med sin verdi, holder det å lagre forskjellen.
- Men: På grunn av feil og spoling må man av og til lagre den ekte verdien.
- Mennesker kan ikke høre lyder under 20 Hz og over 20 000 Hz.
- Om vi hører en kraftig lyd med én frekvens, hører vi ikke litt svakere lyder med noe høyere frekvens.
- Etter å ha hørt en sterk lyd, hører vi dårligere en tid etterpå (inntil 0,2 s).

Komprimering

- Moderne standarder som **MP2** (brukt i DAB), **MP3** og **AAC** (brukt i DAB+, YouTube, iTunes, . . .) utnytter dette og tillater til dels sterk komprimering:

CD-kvalitet	1410 kb/s
MP2	ca 256 kb/s
MP3	ca 160 kb/s
AAC	ca 128 kb/s

- Men kvaliteten går merkbart ned om det komprimeres mer enn dette

Oppsumering av dagens tema

- Alt er bit i en datamaskin; tekst, bilder og lyd
- Unicode og UTF-8 dominerer for koding av tegn men det er ennå andre kodinger man må forholde seg til
- Både rasterbilder og vektorbilder er nyttige, men til hvert sitt formål
- Lydkoding med MP2, MP3 og AAC er blitt standard, men vi bør velge rett kvalitet