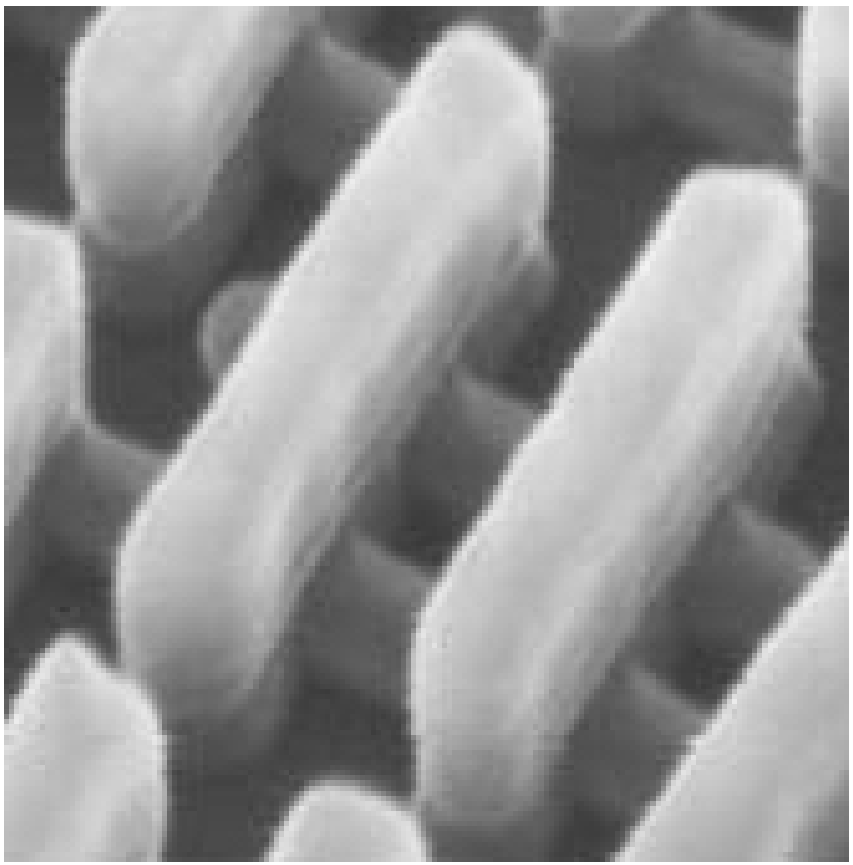
Digital representasjon

Alt er bit!

Hvordan lagre

- tall
- tekst
- bilder
- lyd

som bit i en datamaskin



Binære tall

Skal vi telle med bit (0 og 1), må vi telle binært. Dette gjøres egentlig på samme måte som vi teller desimalt:

- Øk siste siffer med 1.
- Hvis vi ikke har flere siffer, sett til 0 og gjenta for sifferet til venstre.

Binært	0	1	10	11	100	101	110	111	1000	1001	1010	1011
Desimalt	0	1	2	3	4	5	6	7	8	9	10	11

Notasjon

Hvis det mulighet for tvil, skriver vi binære tall som 1001_2 og desimale tall som 1001_{10}.

Et matematisk blikk

I det desimale tallsystemet har posisjonene vekt
$1, 10, 100, 1000, \ldots = 10^0, 10^1, 10^2, 10^3, \ldots$

(Notasjonen a^n («a i n-te») betyr $\underbrace{a \times a \times \cdots \times a}_{n \text{ ganger}}$.)

I det binære tallsystemet har posisjonene vekt
$1, 2, 4, 8, \ldots = 2^0, 2^1, 2^2, 2^3, \ldots$

$$\begin{array}{rcccc} & 8 & 4 & 2 & 1 \\ & 2^3 & 2^2 & 2^1 & 2^0 \\ & \downarrow & \downarrow & \downarrow & \downarrow \\ 1011_2 = & 1 & 0 & 1 & 1 \end{array} = 11_{10}$$

Vi teller binært med grupper av bit, f eks **byte** som inneholder 8 bit:

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	$= 127_{10} = 2^7 - 1$

Vi stopper når vi kommer til øverste bit.

Negative tall

Et negativt tall n lagres i 8 bit som $2^8 + n = 256 + n$:

1	1	1	1	1	1	1	1	=	$-1_{10} = 256 - 1 = 255$
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}

Øverste bit («fortegnsbitet») angir om et tall er negativt.

Heltall i Java

Java tilbyr disse tallene:

byte	8 bit	−128 – +127
short	16 bit	−32 768 – +32 767
int	32 bit	−2 147 483 648 – +2 147 483 647
long	64 bit	−9 223 372 036 854 775 808 – +9 223 372 036 854 775 807

Hvorfor er dette viktig?

```
class Overflyt {
    public static void main(String[] arg) {
        int v = 100000, v2 = v*v;

        System.out.println("v=" + v + " og v2=" + v2);
    }
}
```

gir dette resultatet

```
$ javac Overflyt.java
$ java Overflyt
v=100000 og v2=1410065408
```

Hvor store tall kan vi da lagre?

```
class IkkeOverflyt {
    public static void main(String[] arg) {
        long v = 100000, v2 = v*v;

        System.out.println("v=" + v + " og v2=" + v2);
    }
}
```

gir riktig svar:

```
$ javac IkkeOverflyt.java
$ java IkkeOverflyt
v=100000 og v2=10000000000
```

Heksadesimal notasjon

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

$$1111111011011100101110101001100 0\ldots$$
$$\ldots 0111011001010100001100100001 0000$$

Det er enklere å erstatte fire og fire binære sifre med **heksadesimale** sifre $\mathbf{0}_{16}$–$\mathbf{F}_{16}$:

$$\underbrace{1111}_{F_{16}}\underbrace{1110}_{E_{16}}\underbrace{1101}_{D_{16}}\underbrace{1100}_{C_{16}}\underbrace{1011}_{B_{16}}\underbrace{1010}_{A_{16}}\underbrace{1001}_{9_{16}}\underbrace{1000}_{8_{16}}$$
$$\underbrace{0111}_{7_{16}}\underbrace{0110}_{6_{16}}\underbrace{0101}_{5_{16}}\underbrace{0100}_{4_{16}}\underbrace{0011}_{3_{16}}\underbrace{0010}_{2_{16}}\underbrace{0001}_{1_{16}}\underbrace{0000}_{0_{16}}$$

$$\text{FEDCBA9876543210}_{16}$$

Oktal notasjon

Tidligere brukte man ofte **oktal** notasjon der man slår sammen tre og tre bit:

$$\underbrace{111}_{7_8}\underbrace{110}_{6_8}\underbrace{101}_{5_8}\underbrace{100}_{4_8}\underbrace{011}_{3_8}\underbrace{010}_{2_8}\underbrace{001}_{1_8}\underbrace{000}_{0_8}$$

I dag har hex-notasjonen overtatt.

Men hva med disse tallene?

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

Vi trenger ikke bare heltall men også tall som

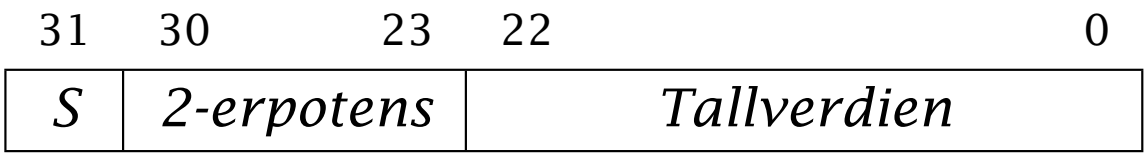
- kan ha veldig små og veldig store verdier
- kan ha desimaler
- ikke behøver å være helt nøyaktige.

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

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

Så lagrer vi den justerte tallverdien (9,6) og 10-erpotensen (−8).

På en datamaskin gjør vi det samme, men bruker vi 2-erpotenser i **float** og **double**.



I Java 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	$2{,}2 \cdot 10^{-308}$	$1{,}8 \cdot 10^{308}$

Vi kan konvertere mellom heltall og flyttall:

```
class Konvertering {
    public static void main(String[] arg) {
        double f1 = 3.94, f2;
        int i;

        i = (int)f1;
        f2 = (double)i;
        System.out.println("i=" + i + " og f2="+f2);
    }
}
```

Svaret blir: i=3 og f2=3.0

Er dette viktig?

```
class FloatTest {
    public static void main(String[] arg) {
        double a = 1.23456789012345678901234567890;

        System.out.printf("a=%.30f\n", a);
    }
}
```

fungerer slik:

```
$ javac FloatTest.java
$ java FloatTest
a=1.234567890123456700000000000000
```

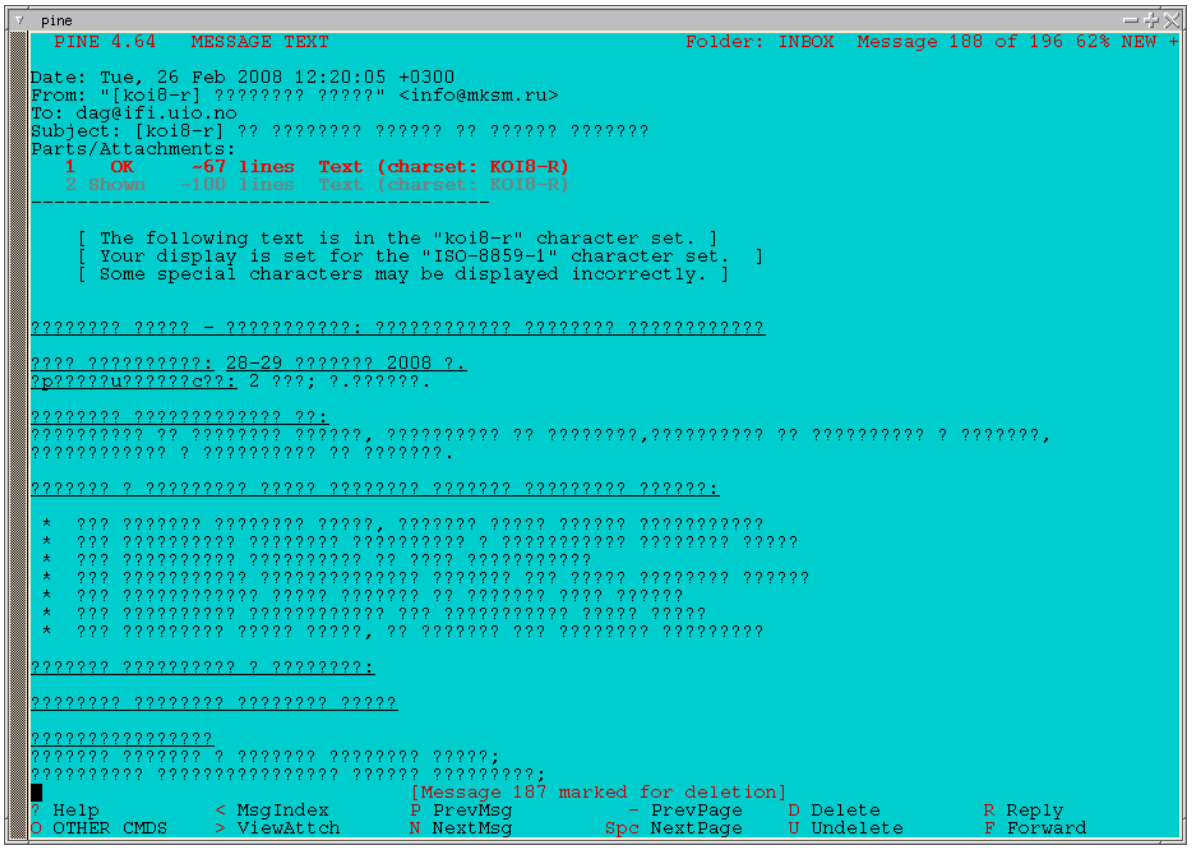
Tekster

Hvordan lagrer vi tegn i datamaskinen? Det er lett:

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

Problemet

Hva hvis ikke alle bruker samme tabell?



ASCII

Første vellykkede forsøk på standardisering var ASCII (American standard code for information interchange) i 1963:

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 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 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

En klart bedre løsning ble **ISO 8859-1** (Latin-1).

Den (og varianter) er ennå i bruk.

ISO 8859–1

0	000 00	32		040 20	64	@	100 40	96	‘	140 60	128	200 80	160		240 A0	192	À	300 C0	224	à	340 E0
1	001 01	33	!	041 21	65	A	101 41	97	a	141 61	129	201 81	161	¡	241 A1	193	Á	301 C1	225	á	341 E1
2	002 02	34	"	042 22	66	B	102 42	98	b	142 62	130	202 82	162	¢	242 A2	194	Â	302 C2	226	â	342 E2
3	003 03	35	#	043 23	67	C	103 43	99	c	143 63	131	203 83	163	£	243 A3	195	Ã	303 C3	227	ã	343 E3
4	004 04	36	$	044 24	68	D	104 44	100	d	144 64	132	204 84	164	¤	244 A4	196	Ä	304 C4	228	ä	344 E4
5	005 05	37	%	045 25	69	E	105 45	101	e	145 65	133	205 85	165	¥	245 A5	197	Å	305 C5	229	å	345 E5
6	006 06	38	&	046 26	70	F	106 46	102	f	146 66	134	206 86	166	¦	246 A6	198	Æ	306 C6	230	æ	346 E6
7	007 07	39	’	047 27	71	G	107 47	103	g	147 67	135	207 87	167	§	247 A7	199	Ç	307 C7	231	ç	347 E7
8	010 08	40	(	050 28	72	H	110 48	104	h	150 68	136	210 88	168	¨	250 A8	200	È	310 C8	232	è	350 E8
9	011 09	41	)	051 29	73	I	111 49	105	i	151 69	137	211 89	169	©	251 A9	201	É	311 C9	233	é	351 E9
10	012 0A	42	*	052 2A	74	J	112 4A	106	j	152 6A	138	212 8A	170	ª	252 AA	202	Ê	312 CA	234	ê	352 EA
11	013 0B	43	+	053 2B	75	K	113 4B	107	k	153 6B	139	213 8B	171	«	253 AB	203	Ë	313 CB	235	ë	353 EB
12	014 0C	44	,	054 2C	76	L	114 4C	108	l	154 6C	140	214 8C	172	¬	254 AC	204	Ì	314 CC	236	ì	354 EC
13	015 0D	45	–	055 2D	77	M	115 4D	109	m	155 6D	141	215 8D	173	-	255 AD	205	Í	315 CD	237	í	355 ED
14	016 0E	46	.	056 2E	78	N	116 4E	110	n	156 6E	142	216 8E	174	®	256 AE	206	Î	316 CE	238	î	356 EE
15	017 0F	47	/	057 2F	79	O	117 4F	111	o	157 6F	143	217 8F	175	¯	257 AF	207	Ï	317 CF	239	ï	357 EF
16	020 10	48	0	060 30	80	P	120 50	112	p	160 70	144	220 90	176	°	260 B0	208	Đ	320 D0	240	ð	360 F0
17	021 11	49	1	061 31	81	Q	121 51	113	q	161 71	145	221 91	177	±	261 B1	209	Ñ	321 D1	241	ñ	361 F1
18	022 12	50	2	062 32	82	R	122 52	114	r	162 72	146	222 92	178	²	262 B2	210	Ò	322 D2	242	ò	362 F2
19	023 13	51	3	063 33	83	S	123 53	115	s	163 73	147	223 93	179	³	263 B3	211	Ó	323 D3	243	ó	363 F3
20	024 14	52	4	064 34	84	T	124 54	116	t	164 74	148	224 94	180	´	264 B4	212	Ô	324 D4	244	ô	364 F4
21	025 15	53	5	065 35	85	U	125 55	117	u	165 75	149	225 95	181	µ	265 B5	213	Õ	325 D5	245	õ	365 F5
22	026 16	54	6	066 36	86	V	126 56	118	v	166 76	150	226 96	182	¶	266 B6	214	Ö	326 D6	246	ö	366 F6
23	027 17	55	7	067 37	87	W	127 57	119	w	167 77	151	227 97	183	·	267 B7	215	×	327 D7	247	÷	367 F7
24	030 18	56	8	070 38	88	X	130 58	120	x	170 78	152	230 98	184	¸	270 B8	216	Ø	330 D8	248	ø	370 F8
25	031 19	57	9	071 39	89	Y	131 59	121	y	171 79	153	231 99	185	¹	271 B9	217	Ù	331 D9	249	ù	371 F9
26	032 1A	58	:	072 3A	90	Z	132 5A	122	z	172 7A	154	232 9A	186	º	272 BA	218	Ú	332 DA	250	ú	372 FA
27	033 1B	59	;	073 3B	91	[	133 5B	123	{	173 7B	155	233 9B	187	»	273 BB	219	Û	333 DB	251	û	373 FB
28	034 1C	60	<	074 3C	92	\	134 5C	124	|	174 7C	156	234 9C	188	¼	274 BC	220	Ü	334 DC	252	ü	374 FC
29	035 1D	61	=	075 3D	93	]	135 5D	125	}	175 7D	157	235 9D	189	½	275 BD	221	Ý	335 DD	253	ý	375 FD
30	036 1E	62	>	076 3E	94	^	136 5E	126	~	176 7E	158	236 9E	190	¾	276 BE	222	Þ	336 DE	254	þ	376 FE
31	037 1F	63	?	077 3F	95	_	137 5F	127		177 7F	159	237 9F	191	¿	277 BF	223	ß	337 DF	255	ÿ	377 FF

Unicode

Men det er bare én løsning som har fremtiden for seg: en tegnkoding som omfatter alle skriftspråk i verden. Den heter **Unicode** og er nå stort sett ferdig.

- Unicode skal omfatte alle skriftspråk som brukes eller har vært brukt

π Я 音 æ∞

- Det er plass til drøyt 1 000 000 tegn.
- 113 021 tegn er foreløbig definert.
- Mer og mer programvare (som Java) støtter Unicode.

UTF-8

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:

$	U+0024	00100100
¢	U+00A2	11000010 10100010
€	U+20AC	11100010 10000010 10101100
𐎄	U+10384	11110000 10010000 10001110 10000100

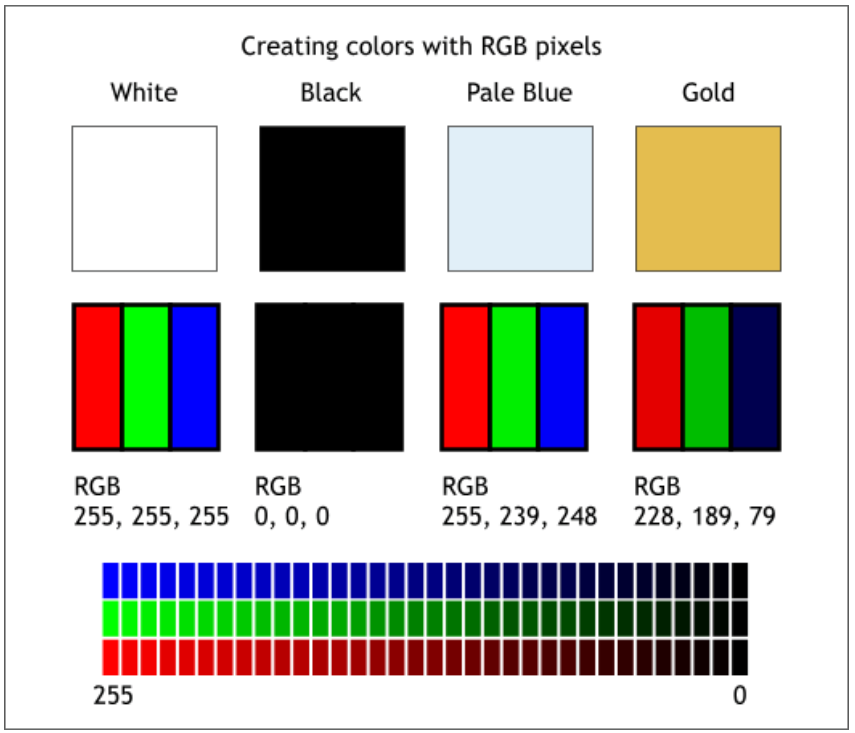
Unicode og UTF-8 er nå standard ved Ifi.

Bilder

Et bilde kan vises på en LCD-fargeskjerm med de tre **RGB**-fargene:

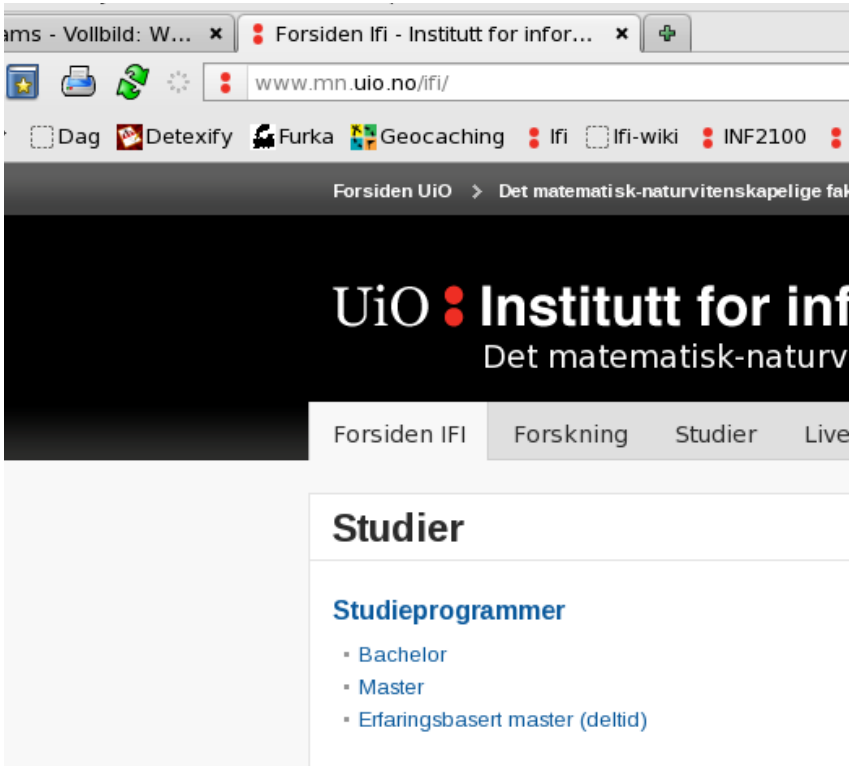
- Rød
- Grønn
- Blå

(Utskrift på papir bruker **CMYK** (Cyan, Magenta, Yellow, blacK) i stedet.)



Et eksempel

På Ifis hjemmeside finnes et «favicon».

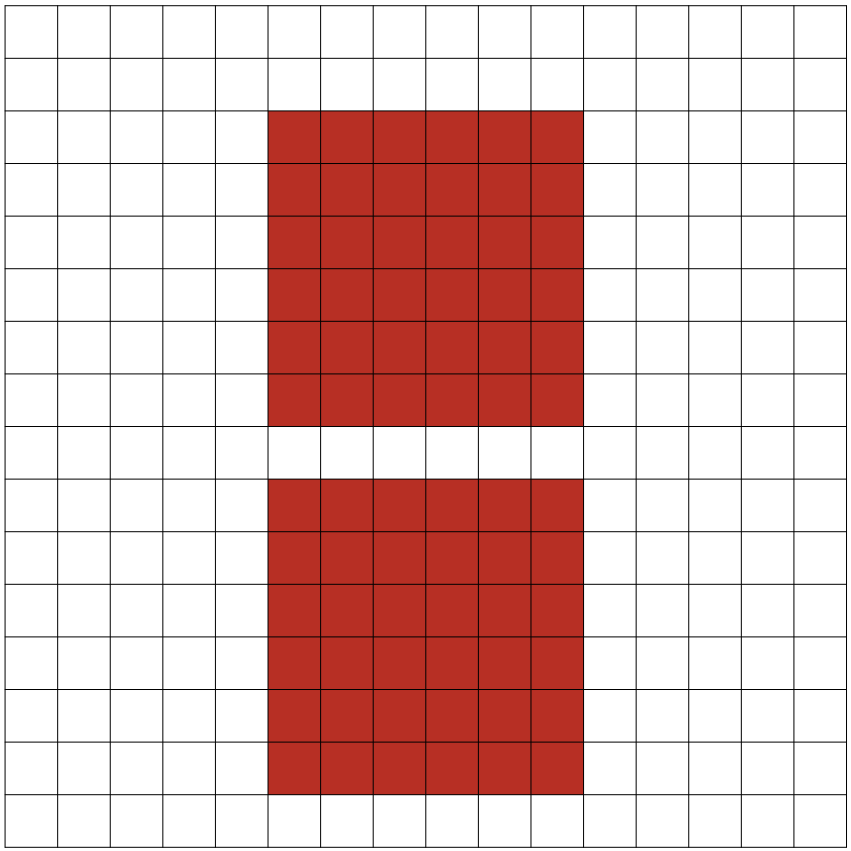


Et veldig enkelt eksempel på et rasterbilde

Dette er et rasterbilde på $16 \cdot 16$ ruter.

I hver rute angir vi RGB-verdiene; her

- Mørk rød: R=183/255, G=47/255, B=36/255
- Gjennomsiktig



Et rasterbilde kan ta stor plass:

- Favicon-en til UiO er på $16 \cdot 16 \cdot 3$ byte = 768 B
- Et skjermbilde kan være på $1920 \cdot 1200 \cdot 3$ byte = 6,9 MB

Er det mulig å komprimere dette?

Fargetabell

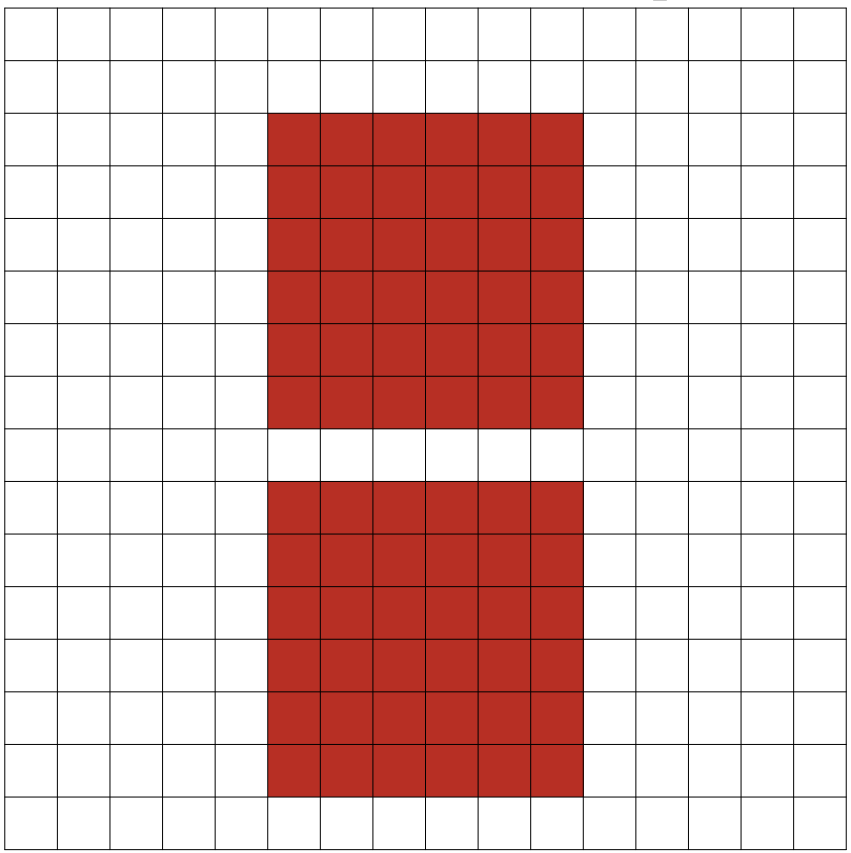
Hvis det ikke er mange ulike farger, bruk en tabell over fargene.

Vårt favicon trenger da $1 + 6 + 16 \cdot 16 \cdot \frac{1}{8}\,\text{byte} = 39\,\text{B}$

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

«Run-length»-koding

I et rasterbilde er ofte piksler ved siden av hverandre like.



16×0
16×0
5×0, 6×1, 5×0
5×0, 6×1, 5×0
5×0, 6×1, 5×0
5×0, 6×1, 5×0
5×0, 6×1, 5×0
5×0, 6×1, 5×0
16×0
5×0, 6×1, 5×0
5×0, 6×1, 5×0
5×0, 6×1, 5×0
5×0, 6×1, 5×0
5×0, 6×1, 5×0
5×0, 6×1, 5×0
16×0

Og om vi er villig til å godta en reduksjon i kvaliteten?

For fotografier gjelder dette:

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

JPEG-formatet

JPEG benytter dette til å lage en forenklet versjon av bildet.

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.)

Originalbildet 100%

Kvalitet 50%

Kvalitet 10%

Kvalitet 5%

En sammenligning

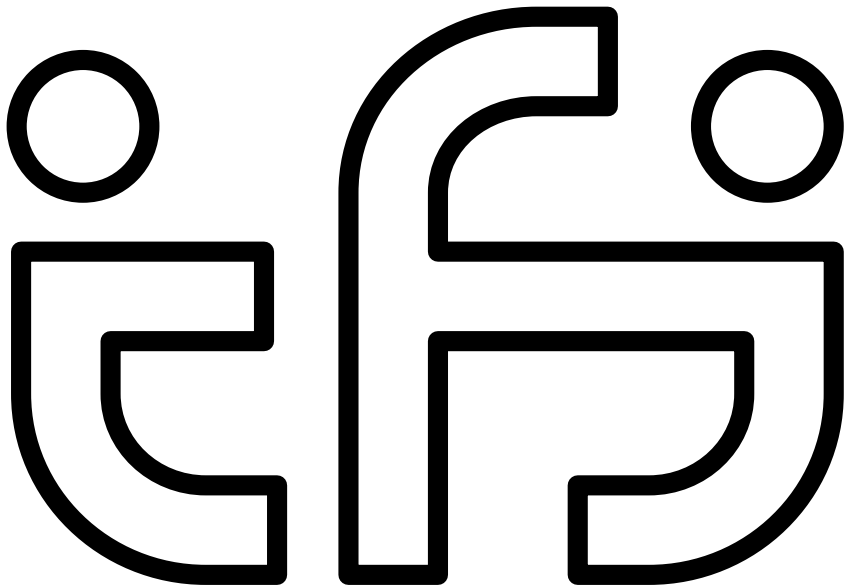
Original

Kvalitet 5%

Vektorgrafikk

Det er også mulig å lagre bilder som linjer og kurver:

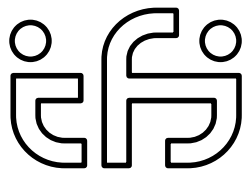
```
   :
newpath
38.4094 2.83464 moveto
38.4094 15.44878 lineto
28.3464 15.44878 lineto
21.03242 15.44878 14.93857
21.1286 14.93857 28.3464 curveto
14.93857 35.77315 lineto
36.56685 35.77315 lineto
36.56685 48.38728 lineto
2.32443 48.38728 lineto
2.32443 28.3464 lineto
2.32443 14.16295 14.0667
2.83464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

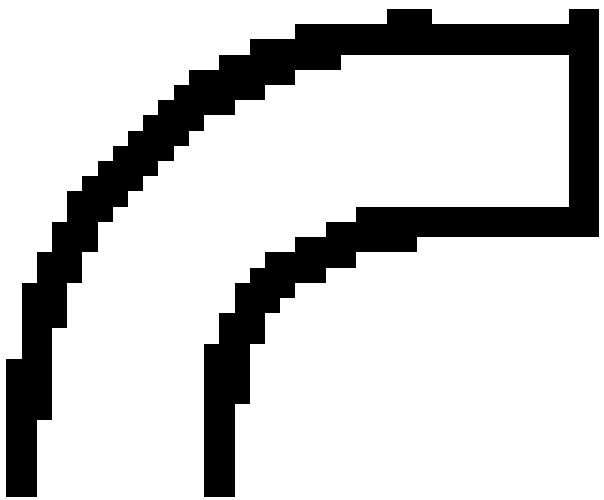


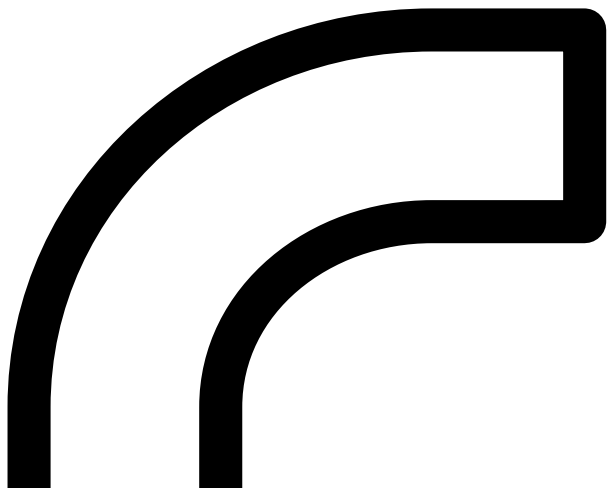
Men hva med vektorer?

Raster

Vektor







Anbefalinger

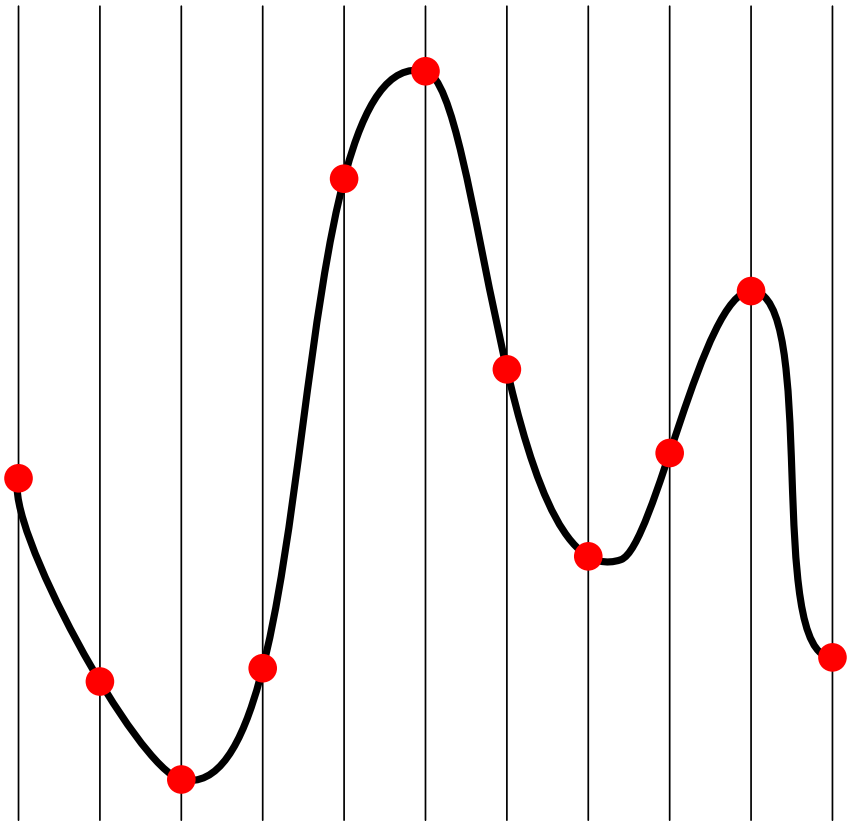
- Bruk vektorgrafikk om mulig: SVG, EPS, PDF.
- For fotografier bruk JPEG i så god kvalitet som mulig.
- For annen rastergrafikk bruk PNG med så mange piksler som mulig.

Hvordan lagre lyd

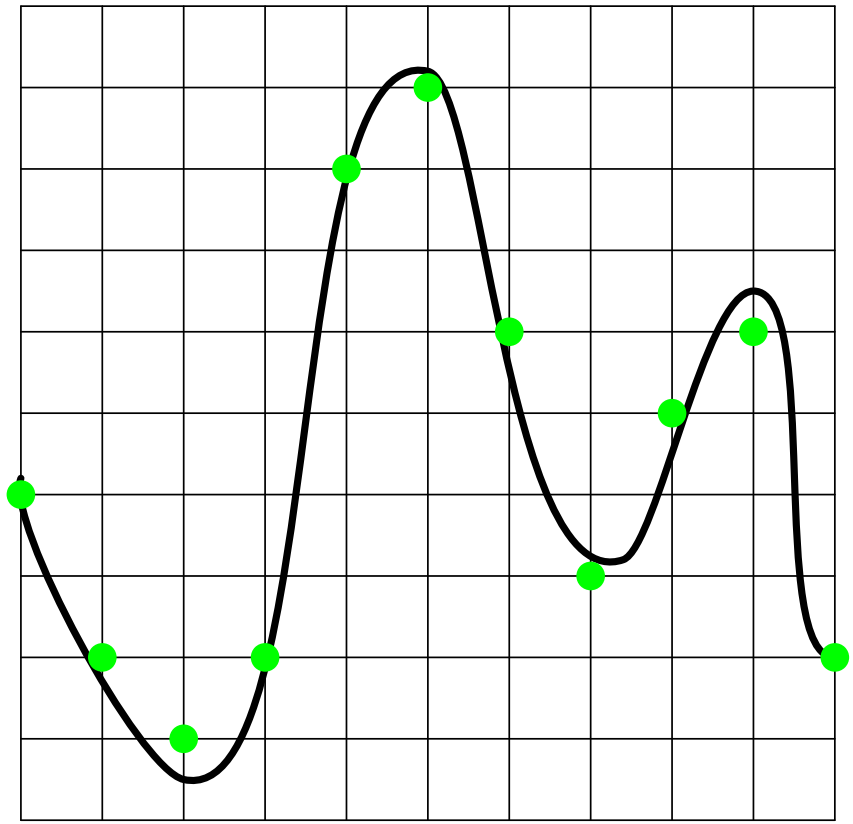
Lyd er bølger i luft, men de kan overføres som strøm:

Hvordan kan vi lagre lyd?

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

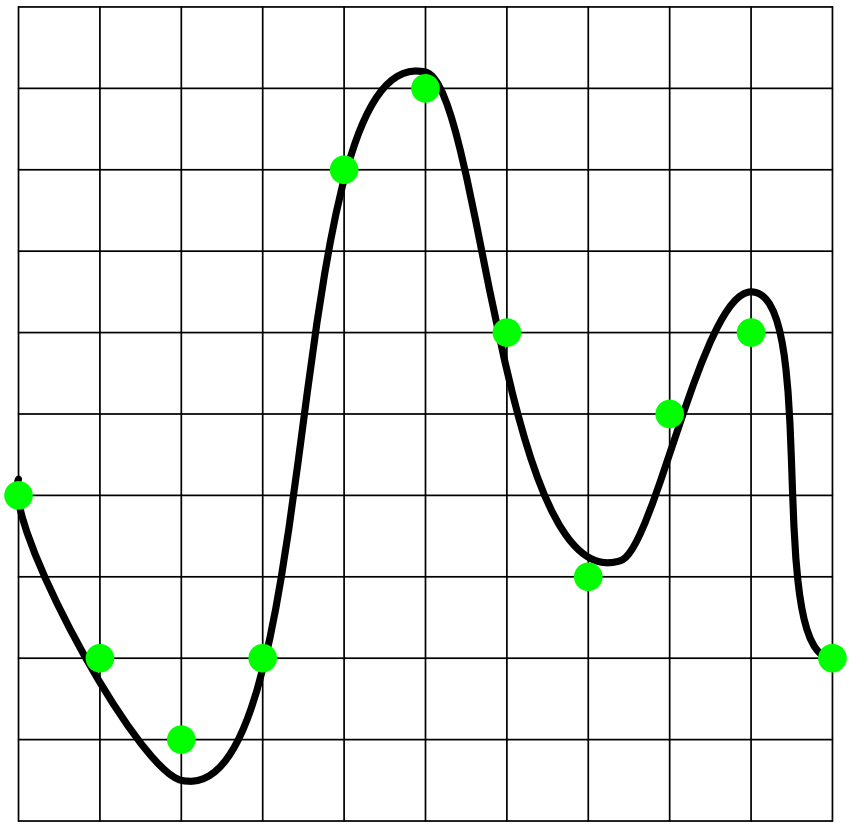


Men for å lagre digitalt må vi måle styrken i faste intervaller:

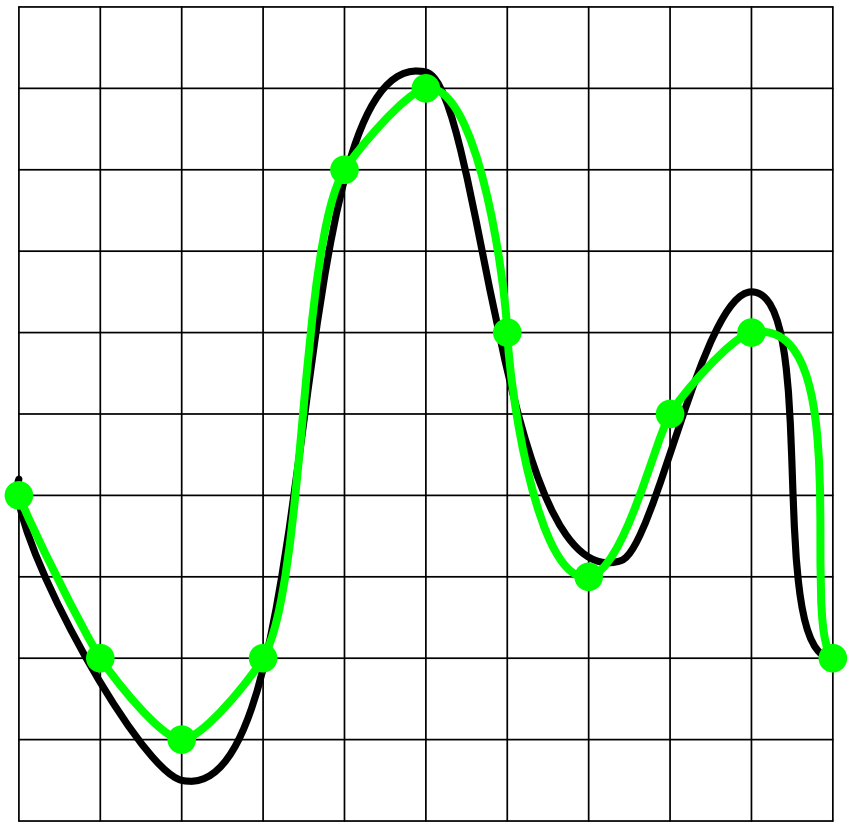


Hvordan kan vi lagre lyd?

Med disse målingene



kan vi gjenskape lyden (men ikke helt nøyaktig):



Kvaliteten på lyden blir da avhengig av

- hvor ofte vi måler
- hvor mange trinn vi benytter til målingen

CD

En vanlig CD har 74 minutter spilletid med god lyd:

- 44 100 målinger («samples») per sekund
- $2^{16} = 65\,536$ intervaller (dvs 2 byte)
- 2 kanaler
- + 37% feilkorreksjonsdata

En CD må derfor ha plass til 783 MB.

Er det mulig å spare plass?

- Høyre og venstre kanal er stort sett nesten like. Det er lurere å lagre venstre 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.

Enda mer plass kan vi spare om vi tar hensyn til hvordan vi mennesker hører:

- Vi 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).

MP3

MP3 utnytter dette og tillater til dels sterk komprimering:

Ekte CD	1410 kb/s
MP3 «CD-kvalitet»	ca 120 kb/s
MP3 «FM-radio»	ca 60 kb/s
MP3 «telefonkvalitet»	ca 10 kb/s

Oppsummering

- Alt er bit i en datamaskin.
- Det finnes ulike typer tallverdier, og programmereren må velge riktig.
- Det er mange ulike tegnkodinger å forholde seg til (ennå).
- Rasterbilder og vektorbilder er nyttige til hvert sitt formål.
- Lydkoding med MP3 er blitt en standard, men vi kan velge kvaliteten.