

Hva skal jeg snakke om i dag?

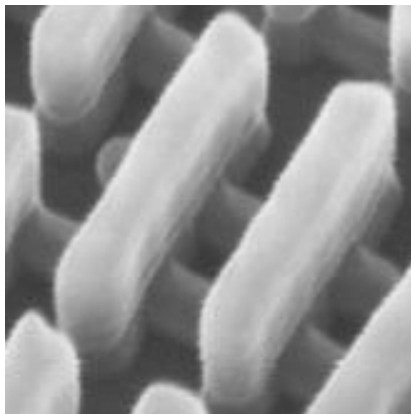
Digital representasjon

Dag Langmyhr
dag@ifi.uio.no

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
 $1, 10, 100, 1000, \dots = 10^0, 10^1, 10^2, 10^3, \dots$

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

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

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



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.

Hvor store tall kan vi da lagre?

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

Hvor store tall kan vi da lagre?

Hvorfor er dette viktig?

Overflyt.java

```
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=1330032704
```

Hvor store tall kan vi da lagre?

IkkeOverflyt.java

```
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
```


Hvor store tall kan vi da lagre?

Heksadesimal notasjon

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

11111110110111001011101010011000...
 ...01110110010101000011001000010000

Det er enklere å erstatte fire og fire binære sifre med **heksadesimale** sifre 0_{16} - F_{16} :

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



Hvor store tall kan vi da lagre?

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.

Flyttall er løsningen

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:

Konvertering.java

```
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



Trenger jeg bekymre meg for dette?

Er dette viktig?

FloatTest.java

```
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=1.23456789012345670000000000000000
```

Kan vi lagre tegn som tall?

Tekster

Hvordan lagrer vi tegn i datamaskinen? Det er lett:

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

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».

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:

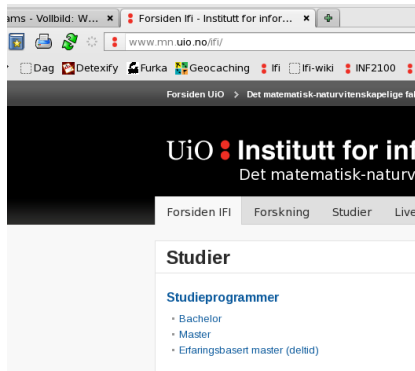
\$	U+0024	<u>00</u> 100100
¢	U+00A2	<u>110000</u> 10 <u>101000</u> 10
€	U+20AC	<u>111000</u> 10 <u>100000</u> 10 <u>101011</u> 00
⚡	U+10384	<u>11110000</u> <u>10010000</u> <u>10001110</u> <u>10000</u> 100

Unicode og UTF-8 er nå standard ved Ifi.

Et veldig enkelt eksempel på et rasterbilde

Et eksempel

På Ifis hjemmeside finnes et «favicon».

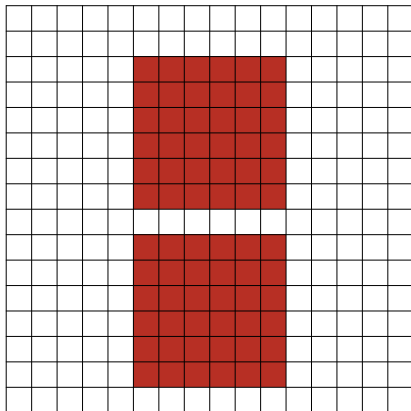


Et veldig enkelt eksempel på et rasterbilde

Dette er et rasterbilde på 16 · 16 ruter.

I hver rute angir vi RGB-verdiene; her

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



Er det mulig å komprimere rasterbilder?

Et rasterbilde kan ta stor plass:

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

Er det mulig å komprimere dette?

Er det mulig å komprimere rasterbilder?

Fargetabell

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

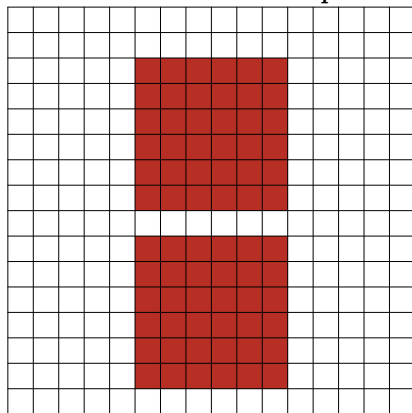
Vårt favicon trenger da $1 + 6 + 16 \cdot 16 \cdot \frac{1}{8}$ byte = 39 B

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

Er det mulig å komprimere rasterbilder?

«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

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%

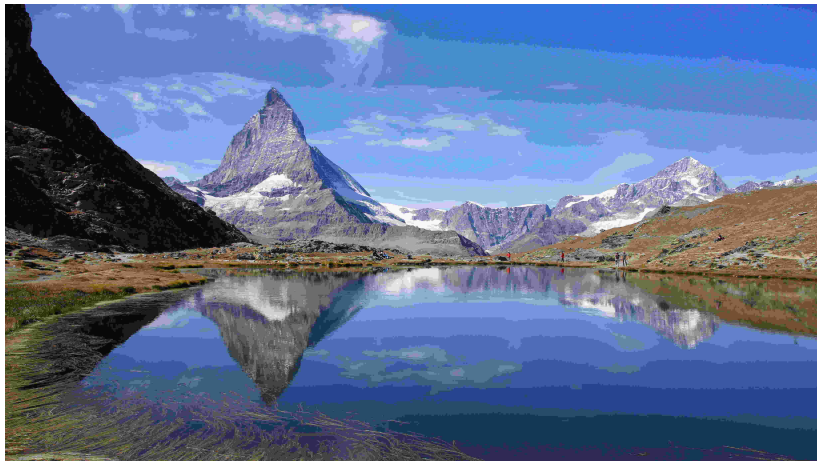


Kvalitet 50%









En sammenligning



Original



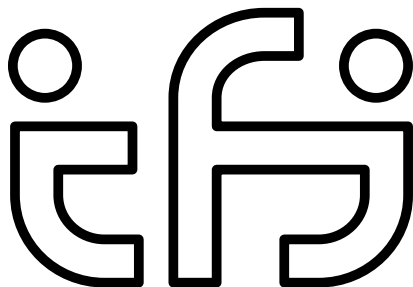
Kvalitet 5%

Vektorgrafikk

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

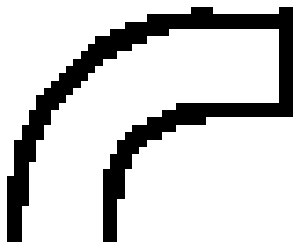
ifl-logo.eps

```
:  
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  
:
```

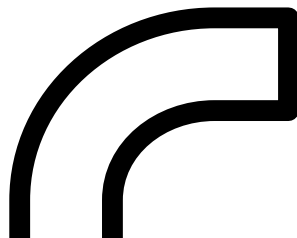


Men hva med vektorer?

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.

Hva er lyd?

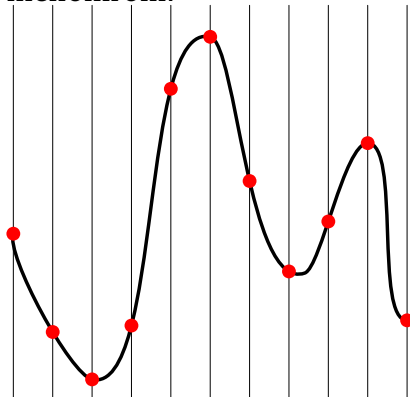
Hvordan lagre lyd

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

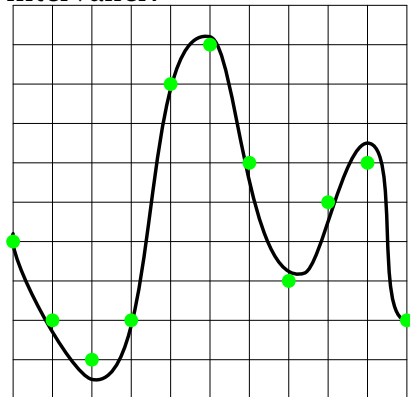


Hvordan kan vi lagre lyd?

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

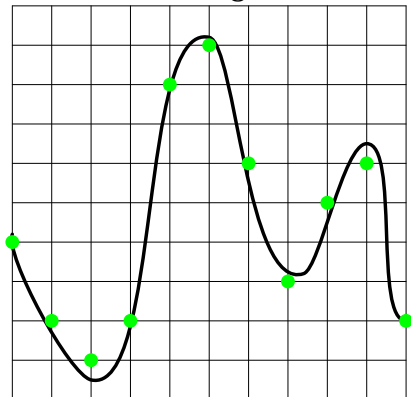


Men for å lagre digitalt må vi måle styrken i faste intervaller:

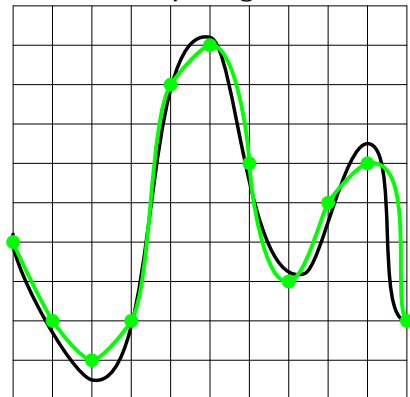


Hvordan kan vi lagre lyd?

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.