Rappresentazione dell'informazione

Come un computer immagazzina i dati

Di cosa parliamo oggi

1. Il substrato fisico: transistor e sistema binario
2. Bit, byte, parola e prefissi
3. Codifica dei caratteri: ASCII → Unicode → UTF-8
4. Notazione posizionale: base 2, 10, 16
5. Codifica degli interi e complemento a due
6. Numeri reali: floating point IEEE 754
7. Codifica di immagini e audio

01

Il substrato fisico: transistor e sistema binario

Il transistor: l'elemento base

Alla base di ogni computer c'è il **transistor**.
Componente a semiconduttore (silicio) che può stare in due stati distinti.

Perché solo due stati?

→ Affidabilità (robusto al rumore)

→ Costo bassissimo

→ Miniaturizzazione: miliardi

in ~1 cm² (circuiti integrati)

Kilby (Nobel Fisica 2000): circuito integrato

Due stati fisici, infiniti significati possibili

Il significato dei due stati è sempre una **convenzione**:

Stato	Interpr. 1	Interpr. 2	Interpr. 3
alto	bianco	cifra 1	VERO
basso	nero	cifra 0	FALSO

Stessa sequenza di bit → significati diversi secondo la convenzione.

Esempio: `01000001` può essere:

```
'A' (carattere ASCII)
65  (intero senza segno)
un'istruzione macchina
```

02

Bit, byte, parola e prefissi

Le unità dell'informazione digitale

Bit (b)

L'unità elementare.

Un singolo elemento a 2 stati.

0 oppure 1
vero oppure falso

Informazione minima.

Byte (B)

Il raggruppamento base.

1 byte = 8 bit
= 2^8 = 256 combinazioni

Sufficiente per un carattere ASCII.

Parola (word)

Il blocco di lavoro della CPU.

32 bit → 4 byte
64 bit → 8 byte

È il blocco che il processore manipola come unità.

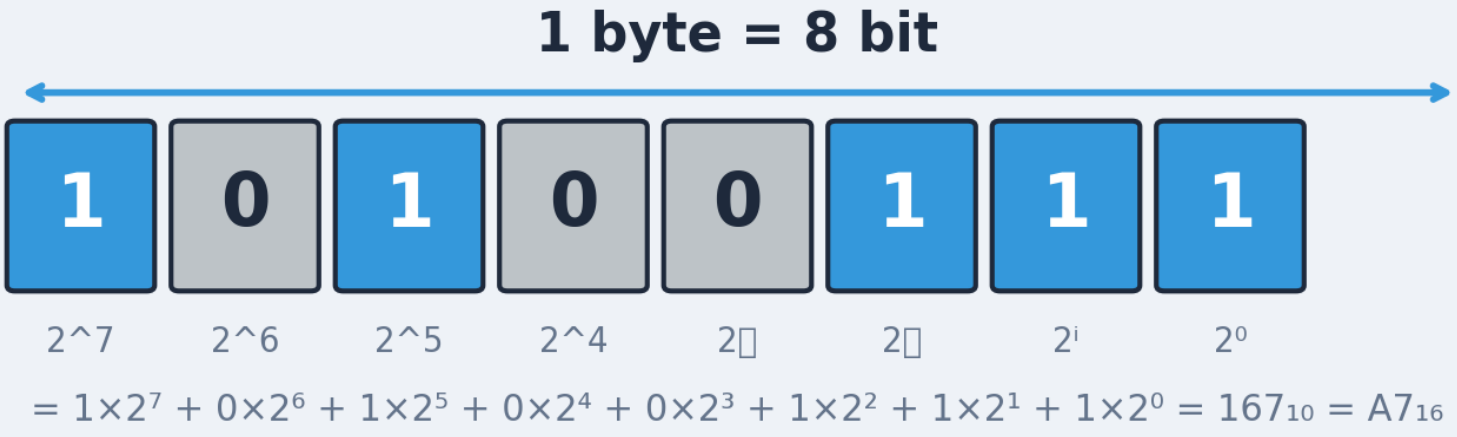
Quanti valori con n bit?

Con n bit possiamo rappresentare 2^n valori distinti:

$n = 1$	$\rightarrow$	2^1	$= 2$	valori	(0, 1)
$n = 2$	$\rightarrow$	2^2	$= 4$	valori	(00, 01, 10, 11)
$n = 4$	$\rightarrow$	2^4	$= 16$	valori	(una cifra esadecimale)
$n = 8$	$\rightarrow$	2^8	$= 256$	valori	(1 byte, 1 carattere ASCII)
$n = 16$	$\rightarrow$	2^{16}	$= 65\,536$	valori	
$n = 32$	$\rightarrow$	2^{32}	$= \sim 4.3 \times 10^9$	valori	(indirizzo IPv4)
$n = 64$	$\rightarrow$	2^{64}	$= \sim 1.8 \times 10^{19}$	valori	(architettura moderna)

Ogni bit in più raddoppia le possibilità!

Anatomia di un byte



Ogni posizione ha un peso (potenza di 2). Il byte mostrato: $10100111_2 = 167_{10} = A7_{16}$.

Con 8 bit: da 00000000 (=0) a 11111111 (=255) → 256 valori possibili.

Prefissi SI vs prefissi binari

Prefissi SI (potenze di 10)

kilo (k) = 10^3 = 1 000
mega (M) = 10^6 = 1 000 000
giga (G) = 10^9
tera (T) = 10^{12}

Usati da produttori di dischi e nella pubblicità.

Un disco da "500 GB" = 500 × 10^9 byte = 465 GiB (come lo vede il SO!)

Prefissi binari (potenze di 2)

kibi (Ki) = 2^{10} = 1 024
mebi (Mi) = 2^{20} = 1 048 576
gibi (Gi) = 2^{30}
tebi (Ti) = 2^{40}

La "i" sta per binario: KiB, MiB, GiB.

La RAM si misura in GiB.

La confusione kilo/kibi causa discrepanze tra capacità dichiarata e reale!

Quanta informazione nel computer?

Testo: 1 pagina ≈ 1.5 KB | dizionario ≈ 1.5 MB | in 1 TB: ~700 000 dizionari!
Immagine: Full HD 1920×1080 × 3 B/pixel ≈ 6 MB (non compressa)
Audio CD: 44 100 Hz × 2 B × 2 canali ≈ 176 KB/s ≈ 10 MB/min
Video Full HD: ~6 MB/frame × 30 fps ≈ 180 MB/s (non compresso!)
Ecco perché servono dischi da TB e la compressione è cruciale.

03

Codifica dei caratteri: da ASCII a Unicode

La tabella ASCII (1963)

Estratto della tabella ASCII

Decimale	Hex	Binario	Carattere
32	20	0010 0000	spazio
48	30	0011 0000	'0'
49	31	0011 0001	'1'
57	39	0011 1001	'9'
65	41	0100 0001	'A'
90	5A	0101 1010	'Z'
97	61	0110 0001	'a'
122	7A	0111 1010	'z'
126	7E	0111 1110	'~'

7 bit → 128 simboli. Lettere consecutive → codici consecutivi. A=65, a=97.

Attenzione: '0' ha codice 48 (30_{16}), NON è il numero zero!

Il problema dell'ASCII esteso

ASCII ha solo 128 simboli: niente è, ñ, ü, niente cinese, arabo, emoji.

ASCII esteso (8 bit, 256 simboli): aggiunge accenti...

...ma con standard diversi per paesi diversi!

```
ISO 8859-1  (Europa occidentale): byte 0xE9 = 'é'
ISO 8859-5  (Cirillico):          byte 0xE9 = 'щ'
ISO 8859-6  (Arabo):              byte 0xE9 = 'ي'
```

Stesso byte, tre caratteri completamente diversi!
Caos totale nello scambio di documenti tra paesi.

La soluzione: Unicode

Unicode (1991–oggi): un unico standard per tutti i caratteri del mondo.

Fino a 21 bit → oltre 2 milioni di punti codice possibili.

Attualmente ~150 000 caratteri definiti.

Include:

- Tutti gli alfabeti storici e moderni
- Simboli matematici: ∫ ∑ ∞ ∂ ∈ ∀ ∃ →
- Notazione musicale, braille, cuneiforme
- Emoji: ogni anno ne aggiungono di nuovi

Ma come si scrivono concretamente in byte? → Le codifiche UTF

UTF-8: la codifica dominante

```
>>> 'A'.encode('utf-8')           # 1 byte (= ASCII)
b'A'
>>> 'è'.encode('utf-8')           # 2 byte
b'\xc3\xa8'
>>> '中'.encode('utf-8')           # 3 byte (cinese)
b'\xe4\xb8\xad'
>>> len('ciao'.encode('utf-8'))    # tutto ASCII → 4 byte
4
>>> len('caffè'.encode('utf-8'))  # 'è' costa 2 byte
6
```

UTF-8: da 1 a 4 byte/carattere. ASCII = 1 byte. Standard del web (>98%) e di Linux.

04

Notazione posizionale: base 2, base 10, base 16

Ripasso: notazione posizionale in base 10

La notazione che conoscete è già posizionale:

$$3725_{10} = 3\times10^3 + 7\times10^2 + 2\times10^1 + 5\times10^0$$
$$= 3000 + 700 + 20 + 5$$

Il valore di ogni cifra dipende dalla sua posizione.

Base 10: 10 cifre possibili (0, 1, 2, ..., 9)

Base 2: 2 cifre possibili (0, 1)

Base 16: 16 cifre possibili (0, 1, ..., 9, A, B, C, D, E, F)

Stessa logica, basi diverse.

Binario → decimale: leggere un numero binario

Il principio è identico alla base 10, ma con potenze di 2:

$$1011_2 = 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$$
$$= 8 + 0 + 2 + 1 = 11_{10}$$

$$11010110_2 = 128 + 64 + 0 + 16 + 0 + 4 + 2 + 0 = 214_{10}$$

Potenze di 2 da ricordare:

$2^0=1$ $2^1=2$ $2^2=4$ $2^3=8$ $2^4=16$ $2^5=32$ $2^6=64$ $2^7=128$ $2^8=256$ $2^9=512$ $2^{10}=1024$

Ci sono `10` tipi di persone al mondo:

quelle che capiscono il binario

e

quelle che non lo capiscono

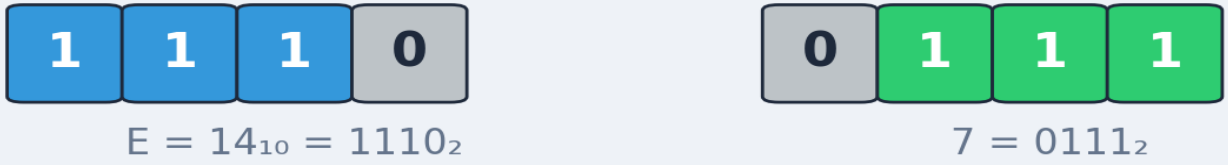
Conversioni con Python

```
>>> bin(25)          # decimale → binario
'0b11001'
>>> int('11001', 2)   # binario → decimale
25
>>> hex(231)          # decimale → esadecimale
'0xe7'
>>> int('E7', 16)     # esadecimale → decimale
231
>>> oct(25)           # decimale → ottale
'0o31'
>>> 0b11001           # literal binario in Python
25
>>> 0xE7              # literal esadecimale
231
```

Python gestisce nativamente numeri in base 2 (`0b`), 8 (`0o`) e 16 (`0x`).

Esadecimale ↔ binario: conversione a vista

$E7_{16}$ → binario



$E7_{16} = 1110\ 0111_2$

1 cifra hex = 4 bit (nibble) → 1 byte = 2 cifre hex

Ogni cifra hex = esattamente 4 bit (nibble). Un byte = 2 cifre hex.

Uso pratico: colori CSS (#FF8800), indirizzi MAC, codici errore, URL (%7E = ~).

05

Codifica degli interi

Interi senza segno (unsigned)

Rappresentazione naturale: notazione posizionale in base 2.

Con n bit: valori da 0 a $2^n - 1$

```
 8 bit  →  0 ... 255                (1 byte)
16 bit  →  0 ... 65 535
32 bit  →  0 ... 4 294 967 295     (~4.3 × 10⁹)
64 bit  →  0 ... ~1.8 × 10¹⁹
```

Ma come rappresentiamo i numeri negativi?

Aritmetica modulare e overflow

Cosa succede se sommo oltre il massimo?

Esempio 8 bit unsigned (max = 255):

```
  11111110      (= 254)
+ 00000011      (= 3)
----------
1 00000001      (= 257, ma il 9° bit non c'è!)
→ Risultato: 00000001 = 1 = 257 mod 256
```

L'overflow è un errore silenzioso: nessun messaggio di errore, il risultato è semplicemente sbagliato!

Interi con segno: complemento a due

Complemento a due (4 bit): da −8 a +7

Binario	Senza segno	Compl. a 2
0000	0	0
0001	1	+1
0010	2	+2
0011	3	+3
...	...	...
0111	7	+7
1000	8	−8
1001	9	−7
...	...	...
1110	14	−2
1111	15	−1

Con n bit: da -2^{n-1} a $+2^{n-1}-1$. Il bit più significativo indica il segno (0=positivo, 1=negativo).

Vantaggio: il processore usa gli stessi circuiti per somme con e senza segno.

Overflow: vederlo in Python

```
>>> import numpy as np

>>> np.uint8(254) + np.uint8(3)   # 8 bit unsigned
/tmp:1: RuntimeWarning: overflow encountered
np.uint8(1)                       # 257 mod 256 = 1!

>>> np.int8(120) + np.int8(10)     # 8 bit signed
/tmp:1: RuntimeWarning: overflow encountered
np.int8(-126)                     # Sorpresa!

>>> 254 + 3   # Python puro: interi a precisione arbitraria
257            # Nessun overflow!
```

Python puro usa interi a precisione arbitraria (nessun overflow). Ma NumPy e i linguaggi compilati (C, Fortran) usano interi a dimensione fissa → **attenzione all'overflow!**

06

Numeri reali:
lo standard floating point

Il problema dei numeri reali

I numeri interi: insieme finito → rappresentazione esatta.

I numeri reali: insieme infinito e denso → con bit finiti

possiamo solo approssimare.

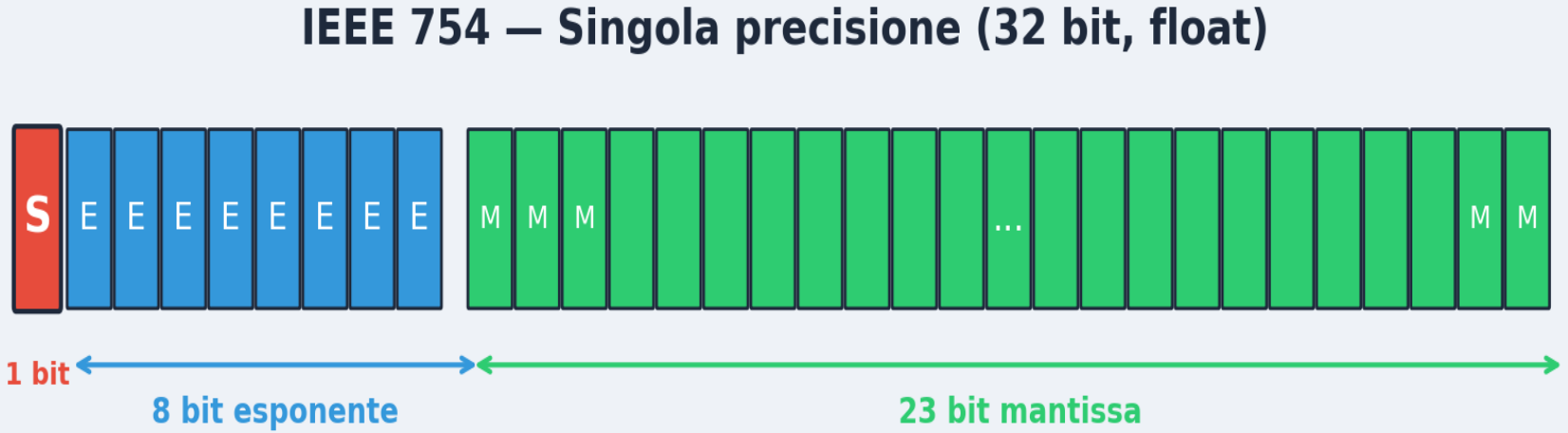
L'idea: notazione scientifica in base 2

```
Come in base 10:   6.022 × 10²³
In base 2:         1.01101 × 2¹⁰
```

```
numero = (−1)ˢ × 1.mantissa × 2^(esponente − bias)
```

Lo standard IEEE 754 (1985, ultimo aggiornamento 2020) definisce come ripartire i bit.

IEEE 754: struttura dei bit



Singola (32 bit): 1+8+23 → ~7 cifre decimali, range ≈ 10^{-38} … 10^{38} (float in C)

Doppia (64 bit): 1+11+52 → ~15 cifre decimali, range ≈ 10^{-308} … 10^{308} (float di Python, default!)

Quadrupla (128 bit): 1+15+112 → ~34 cifre decimali, range ≈ 10^{-4932} … 10^{4932} (raro)

La precisione è finita!

```
>>> 0.1 + 0.2
0.30000000000000004

>>> 0.1 + 0.2 == 0.3
False

>>> f"{0.1:.20f}"
'0.10000000000000000555'
```

0.1_{10} non è rappresentabile esattamente in base 2 ($0.0001100110011\ldots_2$ — periodico!).

In un calcolo scientifico, gli errori di arrotondamento possono propagarsi e alterare il risultato.

Altre sorprese del floating point

```
>>> (0.1 + 0.2) - 0.3              # dovrebbe essere 0
5.551115123125783e-17

>>> 1e16 + 1 - 1e16                 # dovrebbe essere 1
0.0                                  # il +1 si "perde"!

>>> 1e308 * 2                       # overflow → infinito
inf

>>> 1e-324                          # underflow → zero
0.0
```

Regole d'oro: (1) mai confrontare float con ==, (2) sommare numeri di grandezza simile, (3) usare doppia precisione se possibile.

Valori speciali IEEE 754

+Inf / −Inf

Overflow: il risultato è troppo grande.

```
>>> 1e308 * 2
inf
>>> -1e308 * 2
-inf
```

Inf si propaga:

```
inf + 1 = inf
inf * 0 = nan
```

NaN

Not a Number: operazione mal definita.

```
>>> 0.0 / 0.0
nan
>>> float('inf') - float('inf')
nan
```

NaN ≠ NaN (!):

```
>>> nan == nan
False
```

Subnormali e ±0

Numeri molto piccoli vicini allo zero.

Permettono una degradazione graduale (gradual underflow).

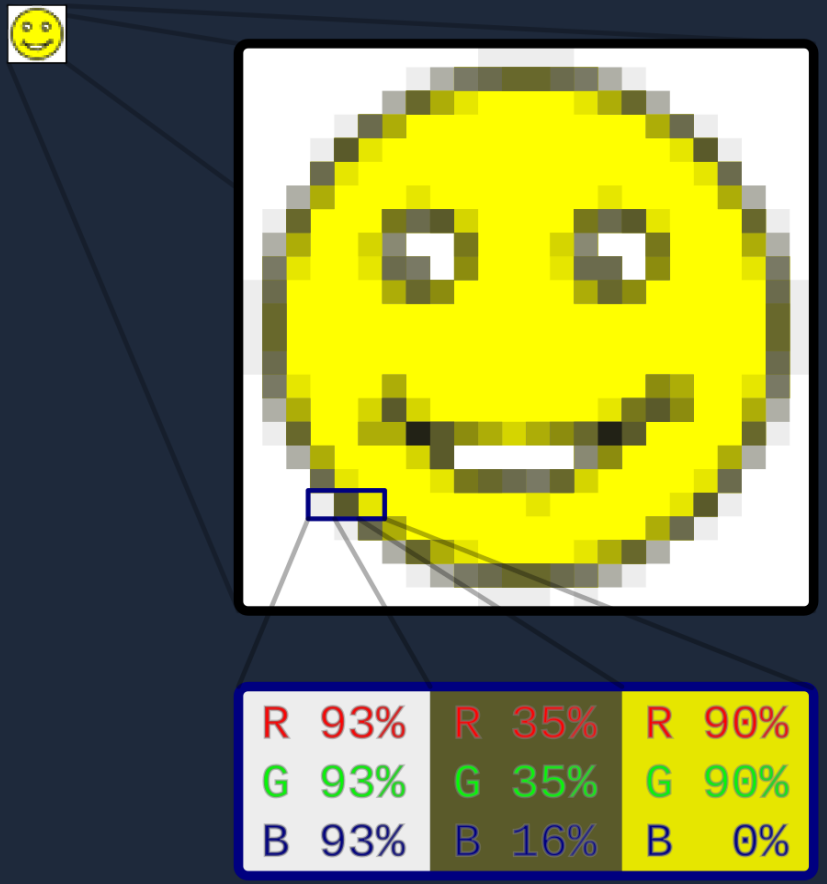
Esistono +0 e −0:

```
>>> -0.0 == 0.0
True
>>> 1/(-0.0)
-inf
```

07

Codifica di immagini e audio

ingrandimento pixel → canali R, G, B



Immagini bitmap RGB

Ogni pixel è composto da tre canali:

R Rosso **G** Verde **B** Blu

Ogni canale: 0-255 (8 bit)

Ogni pixel: 3 byte (24 bit = True Color)

Dimensione file (senza compressione):

```
larghezza × altezza × 3
```

Esempio 1920×1080 non compresso:

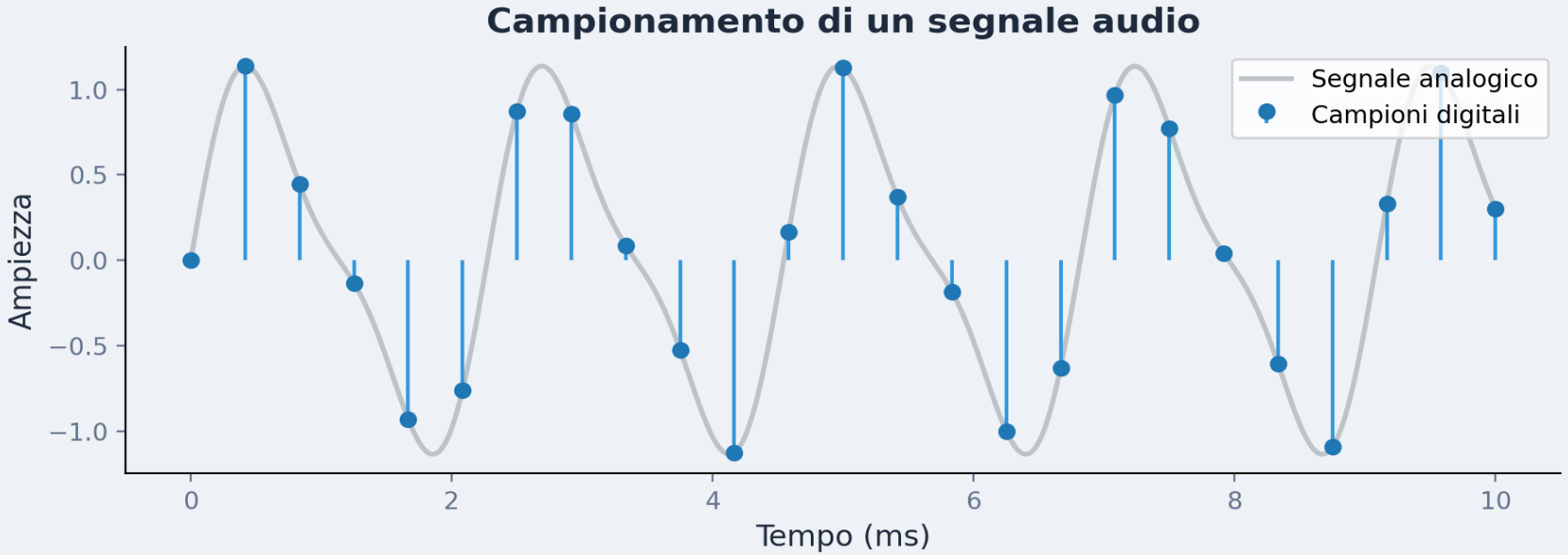
```
1920 × 1080 × 3 = 6.220.800 byte ≈ 6 MB
```

Immagini raster: pixel e RGB (non vettoriale)

L'immagine è una griglia di pixel. Ogni pixel: 3 byte (R, G, B), 256 livelli per canale → 16.7 milioni di colori.

Con trasparenza (RGBA): 4 byte/pixel. Immagini vettoriali (SVG): descrivono forme, non pixel.

Audio digitale: campionamento



CD: 44100 Hz × 16 bit × 2 canali ≈ 176 KB/s ≈ 10 MB/min. Nyquist: campionare ad almeno 2× la freq. massima.

MP3 (lossy): ~1/10 della dimensione eliminando frequenze impercettibili.