

Rappresentazione dei dati e codici

Alessandro Pellegrini
a.pellegrini@ing.uniroma2.it

La necessità della rappresentazione

- I computer elaborano tipologie di dati differenti
 - testo, immagini, musica, programmi, ...
- Tuttavia l'unica informazione che i processori sono in grado di manipolare sono due livelli di informazione
 - alto/basso
 - acceso/spento
 - sì/no
 - 0/1
- Per poter manipolare dati differenti, è necessario accordarsi su *come* rappresentarli

Sistemi di codifica

- L'obiettivo è quello di rappresentare *insiemi* di oggetti da manipolare
 - tali insiemi devono necessariamente essere *finiti*
- Si utilizza un insieme di *simboli* (chiamato *alfabeto*), similmente *finito*
- Ogni elemento dell'insieme di oggetti da rappresentare viene associato ad una *configurazione di simboli*
- Tale associazione è il *codice*, o *sistema di codifica*

▽ 1	◁▽ 11	◀◁▽ 21	≡▽ 31	◀◁▽ 41	◀◁▽ 51
▽ 2	◁▽ 12	◀◁▽ 22	≡▽ 32	◀◁▽ 42	◀◁▽ 52
▽ 3	◁▽ 13	◀◁▽ 23	≡▽ 33	◀◁▽ 43	◀◁▽ 53
▽ 4	◁▽ 14	◀◁▽ 24	≡▽ 34	◀◁▽ 44	◀◁▽ 54
▽ 5	◁▽ 15	◀◁▽ 25	≡▽ 35	◀◁▽ 45	◀◁▽ 55
▽ 6	◁▽ 16	◀◁▽ 26	≡▽ 36	◀◁▽ 46	◀◁▽ 56
▽ 7	◁▽ 17	◀◁▽ 27	≡▽ 37	◀◁▽ 47	◀◁▽ 57
▽ 8	◁▽ 18	◀◁▽ 28	≡▽ 38	◀◁▽ 48	◀◁▽ 58
▽ 9	◁▽ 19	◀◁▽ 29	≡▽ 39	◀◁▽ 49	◀◁▽ 59
▽ 10	◁▽ 20	◀◁▽ 30	≡▽ 40	◀◁▽ 50	

Numeri e numerali

- *Numero*: entità astratta che “esiste” indipendentemente dalla nostra rappresentazione
- *Numerale*: sequenza di caratteri che rappresenta un numero in un dato sistema di numerazione
- Esempio:
 - 𐍲𐍺 in babilonese
 - 四十四 in giapponese
 - 44 in decimale
 - XLIV in numero romano
 - 101100 in binario

Sistemi numerici posizionali

- Il valore di un simbolo dell'alfabeto dipende da dove questo è posizionato
- Vi è necessità dello zero per rappresentare una posizione a valore nullo
- Esempio:
 - $(404,4)_{10}$: 4 centinaia, 0 decine, 4 unità, 4 decimi
 - Notare che il simbolo '4' *codifica* informazioni differenti

- Questo sistema funziona con qualsiasi base numerica:

$$(x)_b = \langle a_n a_{n-1} \cdots a_1 a_0, c_1 c_2 c_3 \cdots \rangle = \sum_{i=0}^n (a_i \cdot b^i) + \sum_{k=1}^{\infty} c_k b^{-k}$$

- Da tale rappresentazione è immediato ottenere l'algoritmo per la conversione di base

Conversione di base

- Per la parte intera:
 - si effettuano divisioni intere successive per la base di destinazione
 - ci si ferma quando si è arrivati al valore 0
 - si ordinano i resti delle divisioni dall'ultimo al primo: $a_n a_{n-1} \cdots a_1 a_0$
- Per la parte frazionaria:
 - si effettuano moltiplicazioni successive per la base di destinazione
 - si sottrae la parte intera
 - ci si ferma quando si è arrivati al valore 0 o quando si individua un *periodo*
 - si ordinano le parti intere sottratte dalla prima all'ultima: $c_1 c_2 c_3 \cdots$

Sistemi più comuni

- Sistema *binario*:
 - Base 2
 - Alfabeto di due simboli: $I = \{0, 1\}$
- Sistema *ottale*:
 - Base 8
 - Alfabeto di otto simboli: $I = \{0, 1, 2, 3, 4, 5, 6, 7\}$
- Sistema *esadecimale*:
 - Base 16
 - Alfabeto di sedici simboli: $I = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$

Scorciatoie di conversione

- Nel caso in cui la base di partenza sia una potenza della base di destinazione o viceversa, ci sono alcune scorciatoie

- Base 2 \leftrightarrow Base 8:

- $(65)_8 = (110\ 101)_2$
- $(17)_8 = (001\ 111)_2$

- $(101100)_2 = (101\ 100)_2 = (54)_8$
- $(110010)_2 = (110\ 010)_2 = (62)_8$

Ottale	Binario
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Scorciatoie di conversione

- Nel caso in cui la base di partenza sia una potenza della base di destinazione o viceversa, ci sono alcune scorciatoie

- Base 2 \leftrightarrow Base 16:

- $(AF)_{16} = (1010\ 1111)_2$
- $(1C)_{16} = (0001\ 1100)_2$

- $(10111001)_2 = (1011\ 1001)_2 = (B9)_{16}$
- $(00001001)_2 = (0000\ 1001)_2 = (09)_{16}$

Esadecimale	Binario
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Intervalli rappresentabili con la notazione posizionale

- Dato un numero k di cifre di un alfabeto associato ad un sistema numerico in base b , è possibile rappresentare tutti i valori nell'intervallo:

$$[0, b^k - 1]$$

- Allo stesso modo, per rappresentare n elementi, sono necessarie un numero di cifre pari a:


$$k = \lceil \log_b n \rceil$$

- È importante conoscere gli intervalli rappresentabili perché le codifiche funzionano su *insiemi finiti*
- Questo è necessario perché un processore è in grado di interpretare unicamente informazioni a dimensione prefissata


Operazioni aritmetiche in altre basi

- Sono di fatto identiche a quelle che conosciamo in base 10
- L'unica accortezza è quella di contare con “più o meno dita”

riporti


$$\begin{array}{r} 110 \\ + 10 \\ \hline 1000 \end{array} \quad \begin{array}{r} + 6 \\ + 2 \\ \hline 8 \end{array}$$

prestiti


$$\begin{array}{r} \cancel{1}110 \\ - 11 \\ \hline 1011 \end{array} \quad \begin{array}{r} - 14 \\ - 3 \\ \hline 11 \end{array}$$

Numeri negativi in base 2

- Fino a questo momento, abbiamo implicitamente trattato soltanto numeri positivi
- È necessario prevedere una codifica anche per i numeri negativi
- È particolarmente interessante ragionare sulla codifica dei numeri negativi in base 2:
 - Tale codifica deve essere ragionevole *per il processore*
 - Deve permettere operazioni veloci
 - Deve permettere un'occupazione di memoria ridotta
- In decimale, come si trasforma 12 nel suo corrispondente negativo?

-12

Rappresentazione in modulo e segno

- Si utilizza uno dei bit della rappresentazione numerica per il segno

1	0	0	0	1	1	0	0	= -12
0	0	0	0	1	1	0	0	= +12

- È interessante notare che con questa rappresentazione esistono due zeri:

1	0	0	0	0	0	0	0	= -0
0	0	0	0	0	0	0	0	= +0

- Tale rappresentazione è inefficiente:
 - circuiti più complessi
 - costi più alti
 - prestazioni minori

Complemento a uno

- I numeri negativi sono rappresentati come negativo aritmetico del valore del numero positivo
- Un numero viene negato invertendo tutti i bit

0	0	0	0	1	1	0	0	= +12
1	1	1	1	0	0	1	1	= -12

- Il bit più significativo rappresenta ancora il segno
- Il nome della rappresentazione deriva dal fatto che, sommando un numero ed il suo negato, si ottiene sempre una sequenza di tutti 1 (*ones' complement*)
- Esistono ancora due zeri!
- Con n cifre, si rappresentano i numeri nell'intervallo $[-(2^{n-1} - 1), 2^{n-1} - 1]$

Complemento a uno: operazione di somma

- L'operazione di somma che coinvolge numeri negativi è più complessa:

$$\begin{array}{r} 1111 \ 1110 \ + \\ 0000 \ 0010 \ = \\ \hline 10000 \ 0000 \end{array} \qquad \begin{array}{r} -1 \ + \\ 2 \ = \\ \hline 0 \end{array}$$

- Il risultato è *sbagliato*
- Il fenomeno si chiama *end-around carry* (riporto di fine giro)
- Per ottenere il risultato corretto, è necessario sommare al risultato ottenuto anche il bit di riporto.
- Lo stesso fenomeno si verifica con le sottrazioni

Complemento a due

- La rappresentazione che consente l'implementazione hardware più semplice ed efficiente
- Largamente utilizzata dalle architetture convenzionali
- Data una rappresentazione ad n cifre, il complemento a due di un numero x è definita come il complemento a 2^n , ossia:
$$2^n - x$$
- Per calcolare il complemento a due di x possiamo ragionare sulla proprietà fondamentale del complemento a uno di x :
 - Il complemento a uno di x è quel valore \bar{x} tale che $x + \bar{x} = 2^n - 1$, quindi:

$$\begin{aligned}x + \bar{x} &= 2^n - 1 \\x + \bar{x} + 1 &= 2^n \\ \bar{x} + 1 &= 2^n - x\end{aligned}$$

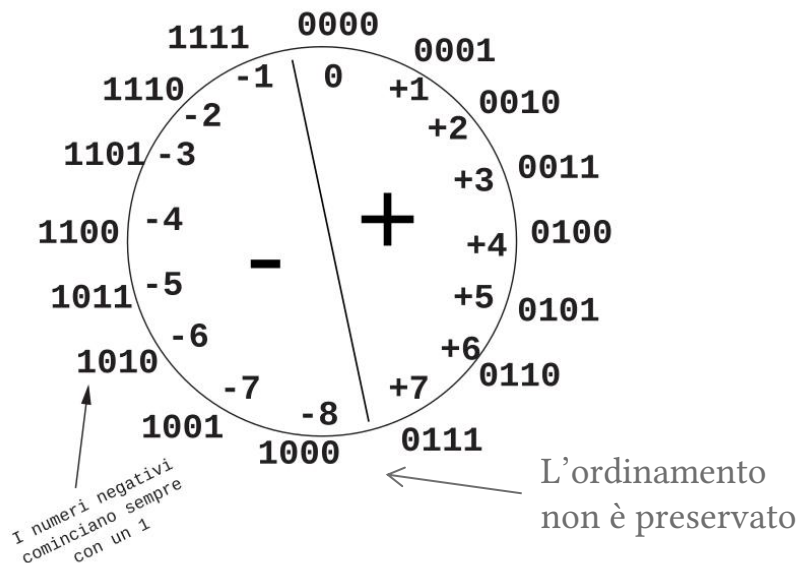
Complemento a due: regola pratica

- Osserviamo i passi di conversione del numero 6 in -6:
 - $x = (0110)_2$
 - $\bar{x} = (1001)_2$
 - $\bar{x} + 1 = (1010)_2 = -6$
- Osservando con attenzione notiamo che le due cifre meno significative di x e $\bar{x} + 1$ sono in entrambi i casi 10. Non è un caso:
 - nel calcolo di \bar{x} tutti gli zeri meno significativi sono stati trasformati in uno
 - sommando uno, si ripristina la configurazione delle cifre meno significative fino al primo 1

Per calcolare il complemento a due di un numero, si parte dal bit meno significativo. Si lasciano inalterati tutti i bit fino a quando non si trova il primo uno. Quindi, si invertono tutti i bit rimanenti.

Complemento a due: organizzazione della codifica

- Esempio in caso di 4 cifre binarie:



- Vi è un solo zero, ma si può codificare -8 e non 8 (il *numero strano*)
- Con n cifre, si rappresentano i numeri nell'intervallo $[-2^{n-1}, 2^{n-1} - 1]$

Complemento a due: somme e sottrazioni

- La somma in complemento a due può essere svolta ignorando il segno:

$$\begin{array}{r} 0000 \ 1111 \ + \quad 15 \ + \\ 1111 \ 1011 \ = \quad -5 \ = \\ \hline \textcolor{red}{1}0000 \ 1010 \quad \quad 10 \end{array}$$

- Ciò significa che per effettuare le sottrazioni, basta *negare* il sottraendo ed effettuare la somma
- È possibile quindi utilizzare *lo stesso circuito* per implementare due operazioni differenti
- Il riporto può essere tralasciato

Complemento a due: condizioni di overflow

- Se i due operandi hanno segno discorde il risultato è sempre rappresentabile.
- Vi sono due casi in cui il risultato di una somma (o sottrazione) in complemento a due non è corretto, poiché soggetto ad *overflow*.
- Primo caso: somma algebrica di due numeri *positivi* A e B . Si ha overflow se $A + B \geq 2^{n-1}$, con n il numero di bit usati per la rappresentazione.
- Secondo caso: somma algebrica di due numeri *negativi* A e B . Si ha overflow se $A + B \geq 2^{n-1}$, con n il numero di bit usati per la rappresentazione.
- Tali circostanze si verificano se gli *ultimi due riporti sono discordi*.

Complemento a due: condizioni di overflow

$$\begin{array}{r} \textcolor{red}{0\ 1} \\ 01111 \ + \\ 00001 \ = \\ \hline 10000 \end{array}$$

$$\begin{array}{r} \textcolor{green}{0\ 0} \\ 01100 \ + \\ 00001 \ = \\ \hline 01101 \end{array}$$

$$\begin{array}{r} \textcolor{red}{1\ 0} \\ 10100 \ + \\ 10101 \ = \\ \hline 01001 \end{array}$$

$$\begin{array}{r} \textcolor{green}{1\ 1} \\ 10111 \ + \\ 11101 \ = \\ \hline 10100 \end{array}$$

Rappresentazione in eccesso

- Anche chiamata *offset binary* o *biased*
- Si seleziona un numero k nell'intervallo rappresentabile
- Viene utilizzata la codifica binaria di k per rappresentare lo zero
- Quando si utilizzano n bit, tipicamente si pone $k = 2^{n-1}$
 - Lo zero è rappresentato con un valore con la sola cifra più significativa pari a 1
 - Viene conservato l'ordinamento dei numeri (non è vero con il complemento a 2)
- La codifica è estremamente semplice:
$$x' = x + k \qquad x = x' - k$$
- Vi è una sola rappresentazione dello zero
- L'intervallo rappresentabile è $[-2^{n-1}, 2^{n-1} - 1]$

Numeri in virgola mobile

Numeri reali

- Esistono molte quantità reali che non possono essere memorizzate accuratamente in numeri interi:
 - lunghezze
 - prezzi
 - temperature
 - frequenze delle note musicali
 - velocità
- Esiste un numero massimo che può essere rappresentato data una parola di n bit:
 - cosa succede se dobbiamo rappresentare una quantità maggiore?
- Vi è un particolare dispositivo, la Floating Point Unit (FPU) che è in grado di manipolare numeri reali
 - occorre utilizzare una rappresentazione specifica

Rappresentazione di base

- La rappresentazione dei numeri in virgola mobile si basa su uguaglianze di questo tipo:

$$12,345 = \underbrace{1,2345}_{\text{mantissa}} \times \underbrace{10^1}_{\text{base}} \quad \left. \vphantom{\underbrace{1,2345}} \right\} \text{esponente}$$

- Il nome *virgola mobile* (floating point) si riferisce al fatto che la virgola può “muoversi” avanti e indietro
 - È sufficiente “adattare” l’esponente
- Sono stati proposti standard differenti nel tempo
 - Studieremo lo standard IEEE 754 a 32 bit
 - Proposto nel 1985, è implementato in tutte le FPU convenzionali

Rappresentazione IEEE 754 a 32 bit



- Un numero reale (float) ha dimensione 32 bit così utilizzati:
 - segno s : 1 bit
 - esponente e : 8 bit
 - mantissa m : 23 bit
- L'esponente decimale E è rappresentato in *eccesso a 127*:
$$e = E + 127$$
- La mantissa rappresenta il valore binario $(1.m)_2$
 - La parte intera viene *omessa* nella rappresentazione

Rappresentazione IEEE 754 a 32 bit



- Il valore decimale può essere calcolato come:
$$(-1)^s \cdot 2^{e-127} \cdot 1.m$$
- Questa rappresentazione viene chiamata *normalizzata*
- Perché l'esponente è rappresentato in eccesso a 127 e non a 128?
- Lo standard permette di rappresentare altri tipi di numero

IEEE 754: tipi di numero

- **Numeri normalizzati:** sono la maggior parte dei numeri rappresentabili dallo standard
- **Numeri denormalizzati:** valori molto prossimi allo zero. Generano alcuni problemi nella gestione degli errori di arrotondamento.
 - La parte intera omessa non è 1, bensì 0
- **zeri:** è possibile rappresentare ± 0 (ci sono due zeri)
 - La maggior parte delle operazioni ignora il segno, ma dividere per ± 0 può dare come risultato $\pm \infty$
- **infiniti** ($\pm \infty$): sono il risultato di una divisione per zero, o di un'operazione che genera un *overflow*.
- **NaNs** (Not a number): sono il risultato di un'operazione che non ha significato ($\infty - \infty$, $0/0$, $\sqrt{-1}$, ...)

Rappresentazione di tutti i tipi di numero

- Numeri differenti sono rappresentabili nello stesso formato
- Valori speciali dell'esponente consentono di individuare le diverse classi di valori

e	m	Tipo di valore
[1,254]	qualsiasi	$(-1)^s \cdot 2^{e-127} \cdot 1.m$ (numeri normalizzati)
0	$\neq 0$	$(-1)^s \cdot 2^{-126} \cdot 0.m$ (numeri denormalizzati)
0	0	$(-1)^s \cdot 0$ (zero con segno)
255	0	$(-1)^s \cdot \infty$ (infinito con segno)
255	$\neq 0$	NaN

Eccezioni

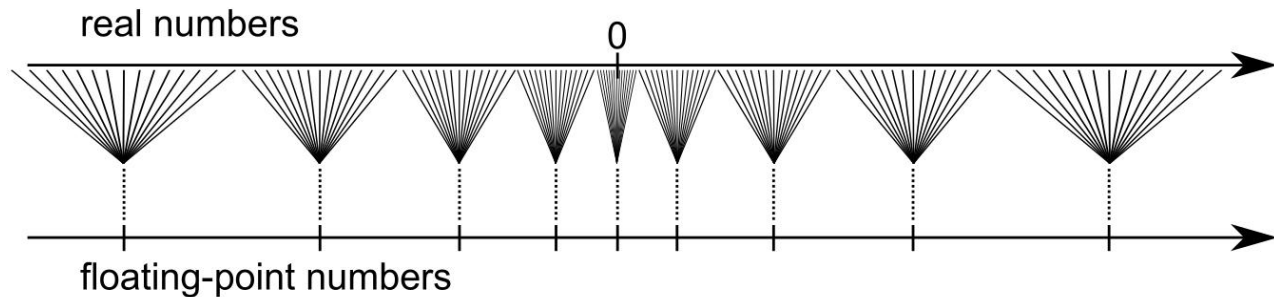
- In alternativa a NaN e infiniti, è possibile richiedere alle FPU di *sollevare delle eccezioni*.
- **Invalid operation**: generata quando si calcola un'operazione non matematicamente corretta;
- **Overflow**: indica che il risultato di un'operazione è troppo grande per essere rappresentato da un numero in virgola mobile;
- **Division by zero**: viene alzata quando si calcola $x / \pm 0$ se $x \neq 0$;
- **Underflow**: analoga all'overflow, per risultati troppo piccoli
- **Inexact**: il risultato "reale" non può essere rappresentato (vi è un errore di arrotondamento)

Massimo e minimo numero positivo rappresentabile

- Caso dei numeri *normalizzati*:
 - La mantissa rappresenta i numeri nell'intervallo:
 $[(1.000000000000000000000000)_2, (1.111111111111111111111111)_2]$
 - Gli esponenti minimo e massimo sono 127 e -126
 - minimo: $2^{-126} \approx 1.1754943508 \times 10^{-38}$
 - massimo: $2^{127} \times (2 - 2^{-23}) \approx 3.4028234664 \times 10^{38}$
- Caso dei numeri *denormalizzati*:
 - La mantissa rappresenta i numeri nell'intervallo:
 $[(0.000000000000000000000001)_2, (0.111111111111111111111111)_2]$
 - L'esponente è -126
 - minimo: $2^{-126} \times 2^{-23} = 2^{-149} \approx 1.4012984643 \times 10^{-45}$
 - massimo: $2^{-126} \times (1 - 2^{-23}) \approx 1.1754942107 \times 10^{-38}$

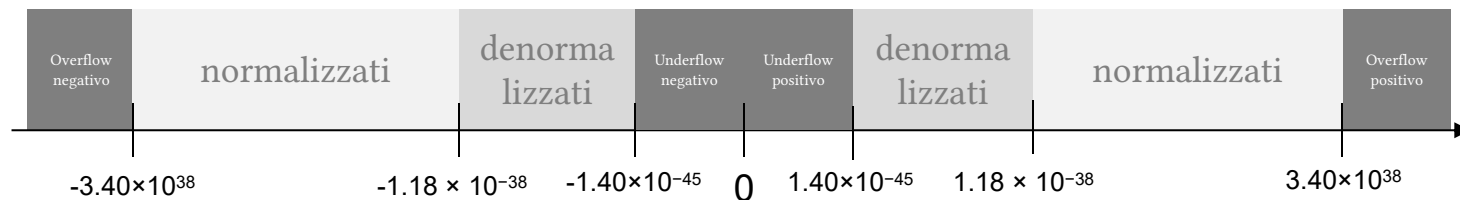
Errori di approssimazione

- I numeri reali sono infiniti, ma i bit utilizzati per la rappresentazione di un numero in virgola mobile sono finiti
- Inoltre, ogni volta che effettuiamo un'operazione su un numero in virgola mobile, commettiamo un errore di approssimazione
- Il risultato è che, anche se l'insieme dei reali è *totalmente connesso*, l'insieme dei numeri in virgola mobile è *sparso*



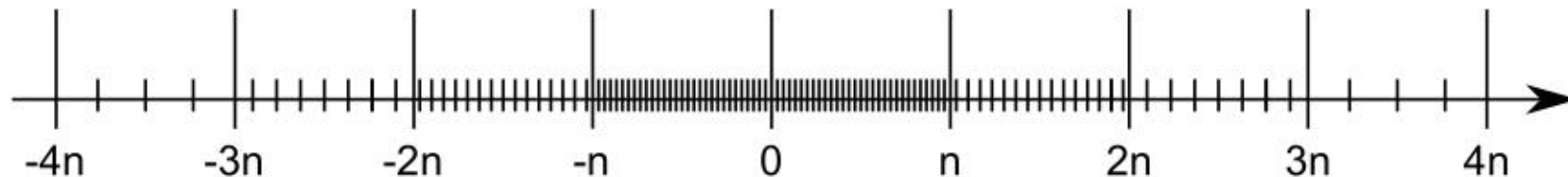
Errori di approssimazione

- La presenza di numeri massimi e minimi rappresentabili crea dei “buchi” nella linea dei numeri rappresentabili
- Altri buchi (piccoli) si creano tra la rappresentazione normalizzata e quella denormalizzata
- Se un numero non è rappresentabile in alcun modo:
 - *overflow*: numero troppo grande o troppo piccolo (negativo)
 - *underflow*: arrotondamento allo zero



Errori di approssimazione

- La divisione in mantissa/esponente fa nascere un'ulteriore peculiarità del formato
- Quando siamo vicini allo zero, l'esponente è piccolo, quindi un incremento nella mantissa di 1 provoca un “salto” piccolo
- Viceversa, per numeri grandi, questo salto sarà maggiore
- La *densità* dei numeri in virgola mobile non è quindi costante
 - con 32 bit, circa metà dei numeri rappresentabili sono compresi tra -1 e 1
 - vi è la stessa quantità di numeri rappresentabili tra 65536 e 131072



Come misurare l'errore

- È possibile quantificare l'errore di approssimazione commesso nella rappresentazione di un numero in virgola mobile

- Si può utilizzare la nozione di *errore assoluto*:

$$\varepsilon_A = x - x'$$

- È una quantità algebrica che ci dice quanto si è perso dell'informazione originale

- Si può utilizzare anche la nozione di *errore relativo*:

$$\varepsilon_R = \frac{x - x'}{x}$$

- È una quantità adimensionale che indica se l'errore commesso è grande o piccolo
- La quantità $-\log_{10} \varepsilon_R$ ci indica il numero di cifre non affette da errore

Un esempio

- Consideriamo $x = 3.5648722189$ e supponiamo di volerlo rappresentare utilizzando soltanto quattro cifre decimali:

$$x \approx \bar{x} = 3.5648$$

- Con questa approssimazione, otteniamo un errore relativo pari a:

$$\varepsilon_R = \frac{x - \bar{x}}{x} = \frac{3.5648722189 - 3.5648}{3.5648722189} = 0.000020258$$

- Il numero di cifre affidabili di \bar{x} è pari a:








$$-\log_{10} (0.000020258) = 4.69$$

Codici ridondanti e irridondanti

Codifiche arbitrarie

- Dovrebbe essere chiaro che ad una data parola di n cifre binarie possiamo far corrispondere quello che vogliamo
- Ad esempio, potremmo scegliere le seguenti rappresentazione per i *giorni della settimana* o per la *frutta*

Giorno	Codifica
Lunedì	000
Martedì	001
Mercoledì	010
Giovedì	011
Venerdì	100
Sabato	101
Domenica	110

Frutta	Codifica
	000
	001
	010
	011
	100
	101
	110

Codifiche arbitrarie

- Fino ad ora abbiamo implicitamente assunto che esiste *una sola codifica* per ciascun elemento degli insiemi che vogliamo rappresentare
- Dati N elementi da rappresentare, n cifre binarie disponibili per la codifica e $m = \lceil \log_2 N \rceil$:
 - Se $n = m$, allora il codice è *irridondante*
 - Se $n > m$, allora il codice è *ridondante*
- Nel caso di un codice ridondante, le $k = n - m$ cifre aggiuntive sono chiamate *cifre di controllo*

Esempio di codice irridondante: codifica ASCII

- American Standard Code for Information Interchange (ASCII): la rappresentazione più comune per i caratteri, basata su un byte
- Superato da Unicode per estendere la quantità di simboli rappresentabili
- ASCII tradizionale: 7 bit. Permette di rappresentare 128 caratteri differenti
 - codici di controllo: [0,31] e 127
 - non permette di rappresentare lettere accentate
 - l'ottavo bit del byte era usato come codice di controllo di errore (ci torniamo tra poco)
- ASCII esteso: 8 bit. Permette di rappresentare 256 caratteri (compresi i codici di controllo)

Codifica ASCII

Binary	Dec	Ascii	Binary	Dec	Ascii	Binary	Dec	Ascii	Binary	Dec	Ascii
000 0000	0	NUL	010 0000	32	space	100 0000	64	@	110 0000	96	`
000 0001	1	SOH	010 0001	33	!	100 0001	65	A	110 0001	97	a
000 0010	2	STX	010 0010	34	"	100 0010	66	B	110 0010	98	b
000 0011	3	ETX	010 0011	35	#	100 0011	67	C	110 0011	99	c
000 0100	4	EOT	010 0100	36	\$	100 0100	68	D	110 0100	100	d
000 0101	5	ENQ	010 0101	37	%	100 0101	69	E	110 0101	101	e
000 0110	6	ACK	010 0110	38	&	100 0110	70	F	110 0110	102	f
000 0111	7	BEL	010 0111	39	'	100 0111	71	G	110 0111	103	g
000 1000	8	BS	010 1000	40	(100 1000	72	H	110 1000	104	h
000 1001	9	HT	010 1001	41)	100 1001	73	I	110 1001	105	i
000 1010	10	LF	010 1010	42	*	100 1010	74	J	110 1010	106	j
000 1011	11	VT	010 1011	43	+	100 1011	75	K	110 1011	107	k
000 1100	12	FF	010 1100	44	,	100 1100	76	L	110 1100	108	l
000 1101	13	CR	010 1101	45	-	100 1101	77	M	110 1101	109	m
000 1110	14	SO	010 1110	46	.	100 1110	78	N	110 1110	110	n
000 1111	15	SI	010 1111	47	/	100 1111	79	O	110 1111	111	o
001 0000	16	DLE	011 0000	48	0	101 0000	80	P	111 0000	112	p
001 0001	17	DC1	011 0001	49	1	101 0001	81	Q	111 0001	113	q
001 0010	18	DC2	011 0010	50	2	101 0010	82	R	111 0010	114	r
001 0011	19	DC3	011 0011	51	3	101 0011	83	S	111 0011	115	s
001 0100	20	DC4	011 0100	52	4	101 0100	84	T	111 0100	116	t
001 0101	21	NAK	011 0101	53	5	101 0101	85	U	111 0101	117	u
001 0110	22	SYN	011 0110	54	6	101 0110	86	V	111 0110	118	v
001 0111	23	ETB	011 0111	55	7	101 0111	87	W	111 0111	119	w
001 1000	24	CAN	011 1000	56	8	101 1000	88	X	111 1000	120	x
001 1001	25	EM	011 1001	57	9	101 1001	89	Y	111 1001	121	y
001 1010	26	SUB	011 1010	58	:	101 1010	90	Z	111 1010	122	z
001 1011	27	ESC	011 1011	59	;	101 1011	91	[111 1011	123	{
001 1100	28	FS	011 1100	60	<	101 1100	92	\	111 1100	124	
001 1101	29	GS	011 1101	61	=	101 1101	93]	111 1101	125	}
001 1110	30	RS	011 1110	62	>	101 1110	94	^	111 1110	126	~
001 1111	31	US	011 1111	63	?	101 1111	95	_	110 0000	127	DEL

Codice Binary Coded Decimal (BCD)

- È un codice irridondante per rappresentare le 10 cifre decimali usando quattro cifre binarie
- Ciascuna cifra è codificata indipendentemente
- Tipicamente, due cifre sono memorizzate in un singolo byte (packed BCD)
- Di facile interpretazione per gli umani, comodo per consentire alle macchine una conversione per la stampa
- Molti bit sono sprecati (circa 1/6) rispetto alla rappresentazione binaria

Base 10	BCD	Base 10	BCD
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

Codice di Gray

- Un codice irridondante a lunghezza fissa, inventato da Frank Gray (1953)
- La rappresentazione è tale per cui tra due numeri adiacenti cambia *una ed una sola cifra binaria*
- Utile nel caso di contatori elettromeccanici per evitare fenomeni transitori:
 - Consideriamo la transizione $(3)_{10} \rightarrow (4)_{10} \equiv (011)_2 \rightarrow (100)_2$
 - Possono verificarsi molte sequenze intermedie, ad esempio: $(011)_2 \rightarrow (010)_2 \rightarrow (000)_2 \rightarrow (100)_2$
 - Il sistema non può distinguere tra configurazioni transitorie e corrette

Base 10	Gray	Base 10	Gray
0	000	4	110
1	001	5	111
2	011	6	101
3	010	7	100

Codici ridondanti e irridondanti

- Si dice *distanza di Hamming* h il numero minimo di cifre diverse tra due parole del codice

$$d(\mathbf{10010}, \mathbf{01001}) = 4$$

$$d(\mathbf{11010}, \mathbf{11001}) = 2$$

- La distanza di Hamming di un codice è quindi:

$$h = \min (d(x, y))$$

per ogni $x \neq y$ appartenenti al codice

- Nel caso di codice irridondante, la distanza è 1
- Un codice ridondante è capace di rivelare errori di peso $\leq h - 1$

Codici ridondanti e irridondanti

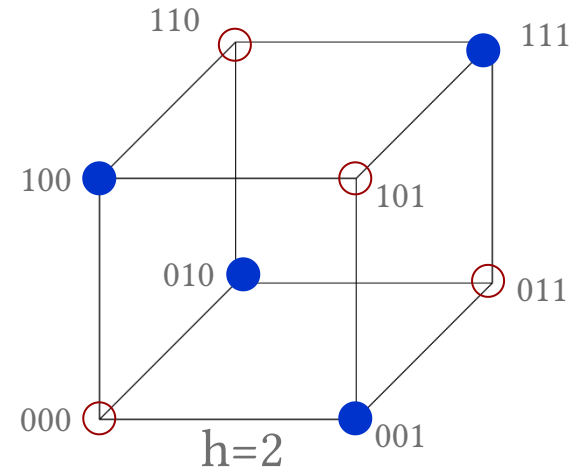
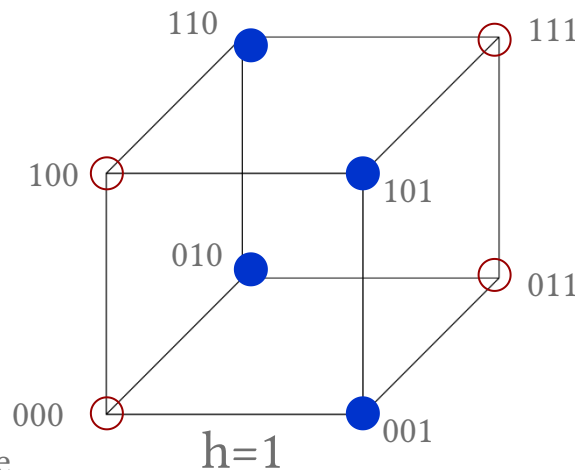
- I codici ridondanti sono di grande importanza pratica:
 - la ridondanza permette di *riconoscere* o *correggere* gli errori
- Errori legati alle inversioni di bit si verificano più frequentemente di quanto si possa immaginare:
 - errori di trasmissione
 - particelle ionizzanti che colpiscono celle di memoria
- Amplissimi spazi di applicazione:
 - aerospazio
 - supercomputer
 - reti di calcolatori



Codici rivelatori di errore

- Il modo in cui utilizziamo lo *spazio* disponibile per rappresentare gli elementi ci può consentire di *rivelare* la presenza di errori

Elemento	Codice 1	Codice 2
A	000	000
B	100	011
C	011	101
D	111	110



○ Parole del codice (legali)
● Parole non appartenenti al codice

Codice di parità

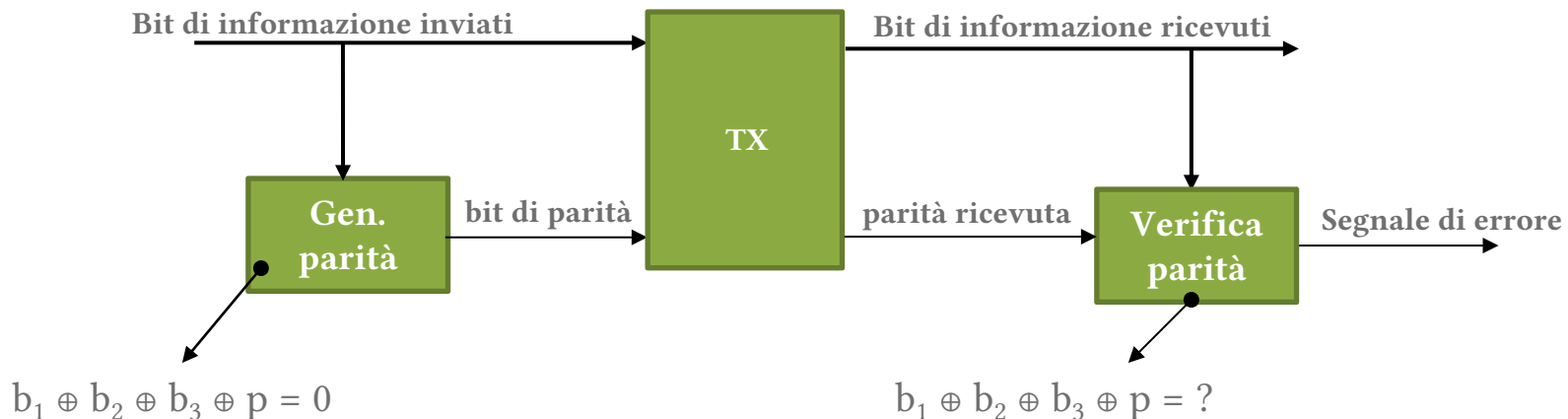
- Un semplice codice ridondante con $h = 2$
- Si ottiene aggiungendo una *cifra di parità* ad un codice irridondante
- Due tipologie:
 - parità (*even parity*): vale 1 se il numero di 1 nella codifica irridondante è *dispari*
 - disparità (*odd parity*): vale 1 se il numero di 1 nella codifica è *pari*

Codice irridondante	Parità	Disparità
000	000 0	000 1
001	001 1	001 0
010	010 1	010 0
011	011 0	011 1
100	100 1	100 0
101	101 0	101 1
110	110 0	110 1
111	111 1	111 0

← rende pari o dispari
il numero di 1 nella parola,
usando la somma modulo
due (\oplus)

Codice di parità: rivelazione degli errori

- Supponendo di usare un codice di parità
- Si può determinare se c'è stato un (singolo) errore di trasmissione verificando la parità in ricezione
 - se pari a 0, non c'è stato errore di trasmissione (o più di uno!)



Esempio con parità pari

- Voglio trasmettere 101
- Il generatore di parità calcola $p = 0$
- Viene trasmesso 1010

Ricevuto	Parità	Segnale di errore
101 <u>0</u>	uguale a zero	OK
111 <u>0</u>	diversa da 0	ERRORE
111 <u>1</u>	uguale a zero	OK

Codici di Hamming

- È un metodo per la costruzione di codici a distanza $h \geq 3$
- Data una parola di codice di $m = n + k$ cifre, con $n \leq 2^k - k - 1$:
 - i bit in posizione 2^i sono bit di parità
 - ciascun bit di parità controlla la correttezza dei bit di informazione la cui posizione, espressa in binario, ha un 1 nella potenza di 2 corrispondente al bit di parità

Posizione cifra		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Dato codificato		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Copertura bit di parità	p1	✓		✓		✓		✓		✓		✓		✓		✓				✓		...
	p2		✓	✓			✓	✓			✓	✓			✓	✓			✓	✓		
	p4				✓	✓	✓	✓					✓	✓	✓	✓					✓	
	p8								✓	✓	✓	✓	✓	✓	✓	✓						
	p16																✓	✓	✓	✓	✓	
...																						

Esempio

- Traformiamo un valore ASCII a $n=7$ bit in un codice di Hamming $h=3$
- Poiché $n \leq 2^k - k - 1$, $k = 4$ e $m = 11$
- I bit di parità sono in posizione 1, 2, 4, 8
- Supponiamo di voler codificare la cifra ASCII 0 (in codice: 0110000)

Posizione bit		1	2	3	4	5	6	7	8	9	10	11	
Dato codificato		p1	p2	0	p4	1	1	0	p8	0	0	0	
Copertura bit di parità	p1	✓		✓		✓		✓		✓		✓	1
	p2		✓	✓			✓	✓			✓	✓	1
	p4				✓	✓	✓	✓					0
	p8								✓	✓	✓	✓	0

- Il valore codificato è quindi 11001100000

Esempio

- Supponiamo di ricevere: 110011010000
- Possiamo calcolare il numero di controllo (*syndrome*) $N_c = \langle p_8 p_4 p_2 p_1 \rangle$:

Posizione bit		1	2	3	4	5	6	7	8	9	10	11	
Dato codificato		1	1	0	0	1	1	1	0	0	0	0	
Copertura bit di parità	p1	✓		✓		✓		✓		✓		✓	1
	p2		✓	✓			✓	✓			✓	✓	1
	p4				✓	✓	✓	✓					1
	p8								✓	✓	✓	✓	0

- Poiché $N_c = (0111)_2 = (7)_{10}$, sappiamo che si è verificato un errore di trasmissione. Inoltre, sappiamo che il bit errato è in *settima posizione*!
- Possiamo quindi ricostruire il valore corretto del dato trasmesso

Esempio

- Supponiamo di ricevere: 110010010000
- Possiamo calcolare il numero di controllo (*syndrome*) $N_c = \langle p_8 p_4 p_2 p_1 \rangle$:

Posizione bit		1	2	3	4	5	6	7	8	9	10	11	
Dato codificato		1	1	0	0	1	0	1	0	0	0	0	
Copertura bit di parità	p1	✓		✓		✓		✓		✓		✓	1
	p2		✓	✓			✓	✓			✓	✓	0
	p4				✓	✓	✓	✓					0
	p8								✓	✓	✓	✓	0

- Poiché $N_c \neq 0$, sappiamo che si è verificato un errore di trasmissione. Tuttavia è incorretto assumere che il bit da correggere sia il primo.
- È un codice a distanza $h=3$
 - può rivelare errori di peso $\leq h - 1$
 - può correggere errori di peso $\leq h - 2$

Riassumendo

Rappresentazioni dei numeri interi

- Esistono differenti rappresentazioni per i numeri interi

Binario	Senza segno	Modulo e segno	Complemento a 1	Complemento a 2	Eccesso a 8
0000	0	0	0	0	-8
0001	1	1	1	1	-7
0010	2	2	2	2	-6
0011	3	3	3	3	-5
0100	4	4	4	4	-4
0101	5	5	5	5	-3
0110	6	6	6	6	-2
0111	7	7	7	7	-1
1000	8	-0	-7	-8	0
1001	9	-1	-6	-7	1
1010	10	-2	-5	-6	2
1011	11	-3	-4	-5	3
1100	12	-4	-3	-4	4
1101	13	-5	-2	-3	5
1110	14	-6	-1	-2	6
1111	15	-7	-0	-1	7

Rappresentazione dei numeri in virgola mobile

- Utilizzano tre campi differenti per rappresentare un numero reale
- A seconda della versione dello standard utilizzato, i campi hanno dimensioni in numero di cifre differenti
- Poiché il numero di cifre utilizzato è finito, non possiamo rappresentare tutti i numeri reali esistenti
- La distribuzione dei numeri rappresentabili *non è uniforme*
- Si possono verificare errori di approssimazione nei calcoli

L'importanza del contesto

- Le varie rappresentazioni non sono intercambiabili tra loro
- Tuttavia, fissato un numero di cifre binarie, una certa parola può essere ambigua:

011101010100100000110

- La corretta interpretazione della parola dipende dal *contesto*
- Il processore non è in grado di discriminare il contesto autonomamente
 - È compito del programmatore
 - Un'interpretazione errata del contesto può portare a errori
 - Nel caso peggiore, a violazioni di sicurezza