Codifica binaria dell’Informazione
Aritmetica del Calcolatore
Algebra di Boole e cenni di Logica

L’informazione

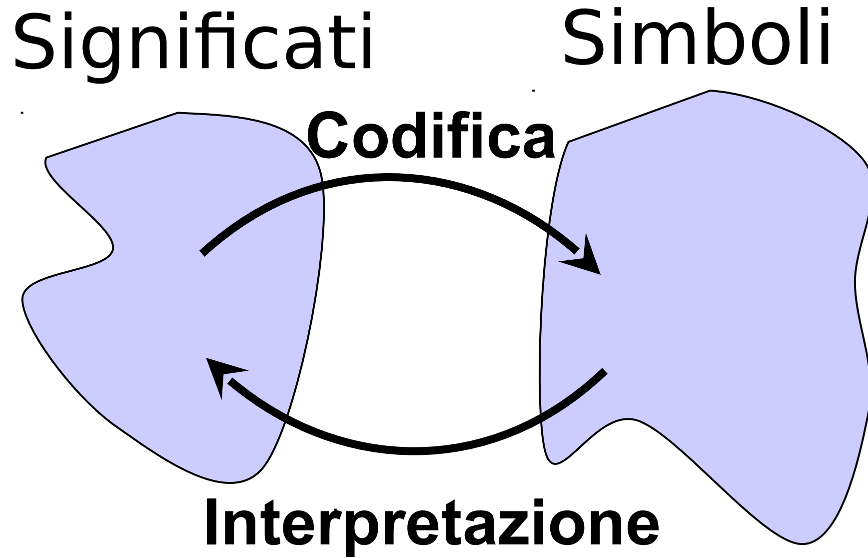
- L’informazione è la conoscenza relativa a oggetti, fatti, concetti, eventi e procedimenti che, in un certo contesto, ha un particolare significato
- **E’ necessario individuare una forma con cui rappresentare le informazioni** affinché queste possano essere comunicate, memorizzate ed elaborate.

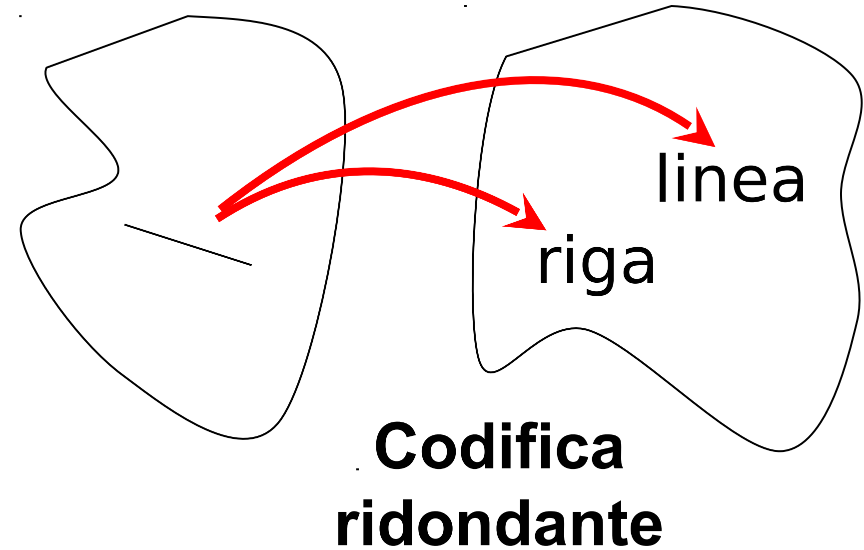
Codifica dell'informazione

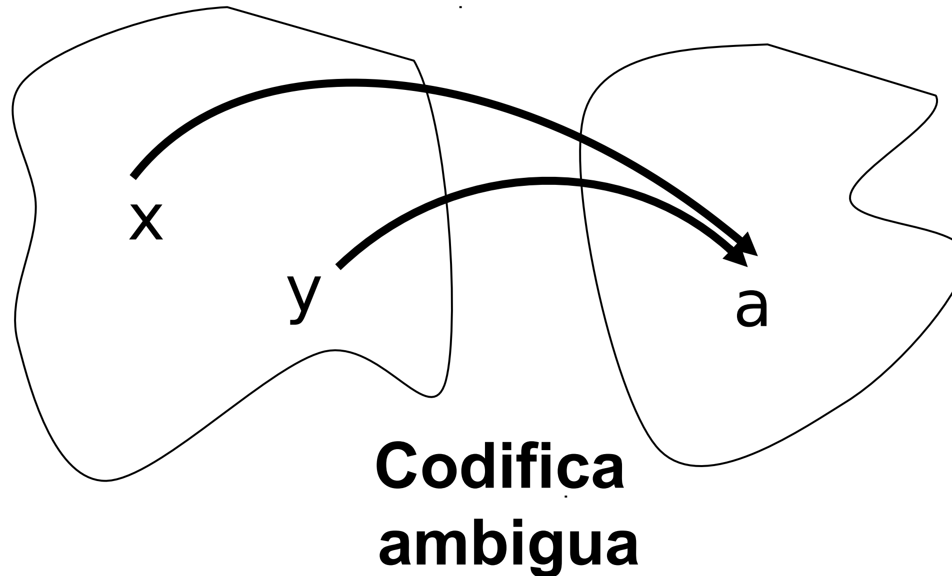
- Rappresentare (codificare) le informazioni
  - con un insieme limitato di simboli (detto *alfabeto* A)
  - in modo non ambiguo (algoritmi di traduzione tra codifiche)

- Esempio: numeri interi
  - Codifica decimale (**dec**, in base dieci)
  - A = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }, |A| = dieci
    - "sette" : 7_{dec}
    - "ventitre" : 23_{dec}
    - "centotrentotto" : 138_{dec}

Significati e simboli







Notazione posizionale per numeri naturali

- **Notazione posizionale:** uno stesso simbolo (che nei sistemi di numerazione si chiama "cifra") assume diversi valori in base alla sua posizione all'interno del numero
  - **Posizione ⇔ peso**, ovvero una potenza della **base, B**
  - Dalla cifra più significativa a quella meno significativa
- Es.: B = 10

$$333_{10}$$

$$3 * 10^2 + 3 * 10^1 + 3 * 10^0 =$$

$$3 * 100 + 3 * 10 + 3 * 1$$

Notazione posizionale per numeri naturali

- Permette di rappresentare un qualsiasi numero naturale (**intero non negativo**), in una qualsiasi base B, nel modo seguente:
  - A = { ... }, con |A| = B
  - la sequenza di **cifre c_i**:

    $c_n c_{n-1} \ldots c_2 c_1$ con $c_i \in \{0, 1, 2, \ldots, B-1\}$, $1 \leq i \leq n$

    rappresenta in **base B il valore**
    $c_n \times B^{n-1} + \ldots + c_2 \times B^1 + c_1 \times B^0$

- Esistono notazioni non posizionali
- Es.: i numeri romani II IV VI XV XX

Numeri naturali in varie basi

- "ventinove" in varie basi
  - B = otto — A = {0,1,2,3,4,5,6,7} — $29_{10} = 35_8$
  - B = cinque — A = {0,1,2,3,4} — $29_{10} = 104_5$
  - B = tre — A = {0,1,2} — $29_{10} = 1002_3$
  - B = sedici — A = {0,1,...,8,9,A,B,C,D,E,F} — $29_{10} = 1D_{16}$

- Codifiche notevoli
  - Esadecimale (sedici), ottale (otto), binaria (due)

Codifica **binaria**

- Usata dal calcolatore per rappresentare **tutte** le informazioni
  - B = due, **A = { 0, 1 }**
  - **BIT** (crasi di “BInary digIT”):
    - unità **elementare** di informazione
  - Dispositivi che assumono **due** stati
    - Ad esempio due valori di tensione V_A e V_B
- *Numeri binari naturali:*

  la sequenza di **bit** $\mathbf{b_i}$ (cifre binarie):

  $b_n\ b_{n-1}\ \ldots\ b_1$ con $b_i \in \{0, 1\}$

  rappresenta in base 2 il valore:

  $b_n \times 2^{n-1} + b_{n-1} \times 2^{n-2} + \ldots + b_1 \times 2^0$

Codifica binaria

- **Quanti valori** diversi posso codificare con parole binarie composte da **K** bit?
  - 1 bit: $2^1 = 2$ stati (0,1) → 2 valori
  - 2 bit: $2^2 = 4$ stati (00,01,10,11) → 4 valori
  - 3 bit: $2^3 = 8$ stati (000,001,010,011,100,101,110,111) → 8 valori
  - ...
  - k bit: 2^k stati → 2^k valori distinti
- Se si passa da k bit a k+1 bit si raddoppia il numero di valori rappresentabili

- In generale, con **n** bit codifichiamo $\mathbf{2^n}$ valori: **da 0 a** $\mathbf{2^n-1}$
- **Quanti bit** mi servono per codificare **N** valori?
  - $N <= 2^k$ → $k >= \log_2 N$ → $k = \lceil \log_2 N \rceil$

Numeri binari naturali (bin)

- Ipotesi: **le parole di un codice hanno tutte la stessa lunghezza**
- Con 1 Byte (cioè una sequenza di 8 bit):
  - $00000000_{bin} = 0_{dec}$
  - $00001000_{bin} = 1 \times 2^3 = 8_{dec}$
  - $00101011_{bin} = 1 \times 2^5 + 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 = 43_{dec}$
  - $11111111_{bin} = \Sigma_{i = 1,\ldots,8} \, 1 \times 2^{i-1} = 255_{dec}$
- Conversione bin $\rightarrow$ dec e dec $\rightarrow$ bin
  - bin$\rightarrow$dec: $11101_{bin} = \Sigma_i b_i 2^i = 2^4+2^3+2^2+2^0 = 29_{dec}$
  - dec$\rightarrow$bin: ***metodo dei resti***

Conversione dec $\rightarrow$ bin

Si calcolano i resti delle divisioni per due

In pratica basta:

1. Decidere se il numero è pari (resto 0) oppure dispari (resto 1), e annotare il resto
2. Dimezzare il numero (trascurando il resto)
3. Ripartire dal punto 1. fino a ottenere 0 come quoziente della divisione

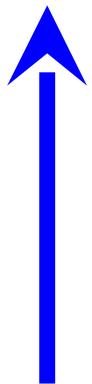
$19 : 2 \rightarrow 1$

$9 : 2 \rightarrow 1$

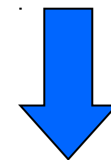
$4 : 2 \rightarrow 0$

$2 : 2 \rightarrow 0$

$1 : 2 \rightarrow 1$



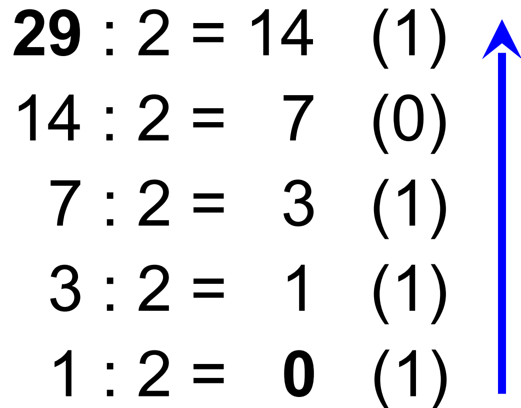
si ottiene 1: fine



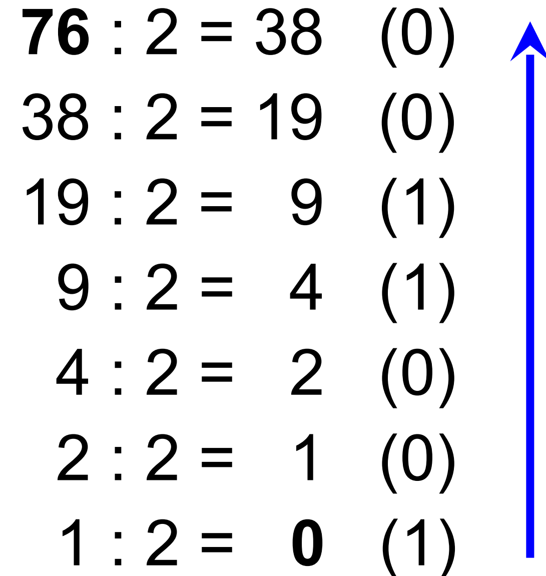
$\mathbf{19_{dec} = 10011_{bin}}$

Ecco un esempio, per quanto modesto, di **algoritmo**

Metodo dei resti



$29_{dec} = 11101_{bin}$



$76_{dec} = 1001100_{bin}$

Del resto 76 = 19x4 = **1001100**

Per raddoppiare, in base due, si aggiunge uno zero in coda, così come si fa in base dieci per decuplicare

N.B. Il metodo funziona con tutte le basi!

$29_{10}=45_{6}=32_{9}=27_{11}=21_{14}=10_{29}$

Conversioni rapide bin $\rightarrow$ dec

- In binario si definisce una *notazione abbreviata*:

  K $= 2^{10} = 1.024 \approx 10^3$ (Kilo)

  M $= 2^{20} = 1.048.576 \approx 10^6$ (Mega)

  G $= 2^{30} = 1.073.741.824 \approx 10^9$ (Giga)

  T $= 2^{40} = 1.099.511.627.776 \approx 10^{12}$ (Tera)

- Diventa molto facile e quindi *rapido* calcolare il valore *decimale approssimato* delle *potenze di 2*, anche se hanno esponente grande
- Es.: quanto vale, approssimativamente, 2^{17}?
  - $2^{17} = 2^{7+10} = 2^7 \times 2^{10} = 128$ K
  - Basta scomporre in modo *additivo* l'esponente

Aumento e riduzione dei bit in bin

- ***Aumento*** dei bit
  - premettendo in modo progressivo un bit 0 a sinistra, il valore del numero non muta

$4_{dec} = 100_{bin} = 0100_{bin} = 00100_{bin} = \dots\ 000000000100_{bin}$

$5_{dec} = 101_{bin} = 0101_{bin} = 00101_{bin} = \dots\ 000000000101_{bin}$

- ***Riduzione*** dei bit
  - cancellando in modo progressivo un bit 0 a sinistra, il valore del numero non muta, *ma bisogna arrestarsi quando si trova un bit* 1!

$7_{dec} = 00111_{bin} = 0111_{bin} = 111_{bin}$ STOP !

$2_{dec} = 00010_{bin} = 0010_{bin} = 010_{bin} = 10_{bin}$ STOP !

Numeri interi in modulo e segno (m&s)

- *Numeri binari interi* (positivi e negativi) *in modulo e segno* (**m&s**)
  - il primo bit a sinistra rappresenta il segno del numero (*bit di segno*)
    - 0 per il segno positivo
    - 1 per il segno negativo
  - gli altri bit rappresentano il **valore assoluto**

- Esempi con n = 9 (8 bit + un bit per il segno)
  - $000000000_{m\&s} = +\ 0 =$
  - $000001000_{m\&s} = +\ 1 \times 2^3 = 8_{dec}$
  - $100001000_{m\&s} = -\ 1 \times 2^3 = -8_{dec}$
  - … e così via …

Osservazioni sul m&s

- Il bit di segno è *applicato* al numero rappresentato, ma non fa propriamente *parte* del numero in quanto tale
  - il bit di segno non ha significato numerico
- *Distaccando* il bit di segno, i bit rimanenti rappresentano il **valore assoluto** del numero
  - che è intrinsecamente positivo

Il complemento a 2 (C_2)

- *Numeri interi in complemento a* 2: il C_2 è *un* sistema binario, ma il primo bit (quello a sinistra, il più significativo) ha *peso negativo*, mentre tutti gli altri bit hanno peso positivo
- La sequenza di bit:

  $$b_n \, b_{n-1} \, \ldots \, b_1$$

  rappresenta in C_2 il valore:

  $$-b_n \times 2^{n-1} + b_{n-1} \times 2^{n-2} + \ldots + b_1 \times 2^0$$

Il bit più a sinistra è ancora chiamato *bit di segno*

Numeri a tre bit in C_2

- $000_{C2} = -0\times2^2 + 0\times2^1 + 0\times2^0 = \quad 0_{dec}$
- $001_{C2} = -0\times2^2 + 0\times2^1 + 1\times2^0 = \quad 1_{dec}$
- $010_{C2} = -0\times2^2 + 1\times2^1 + 0\times2^0 = \quad 2_{dec}$
- $011_{C2} = -0\times2^2 + 1\times2^1 + 1\times2^0 = \quad 2+1 = 3_{dec}$
- $100_{C2} = -1\times2^2 + 0\times2^1 + 0\times2^0 = -4_{dec}$
- $101_{C2} = -1\times2^2 + 0\times2^1 + 1\times2^0 = -4+1 = -3_{dec}$
- $110_{C2} = -1\times2^2 + 1\times2^1 + 0\times2^0 = -4+2 = -2_{dec}$
- $111_{C2} = -1\times2^2 + 1\times2^1 + 1\times2^0 = -4+2+1 = -1_{dec}$

N.B.: in base al bit di segno lo zero è considerato positivo

Interi relativi in m&s e in C_2

Se usiamo 1 Byte: da –128 a 127

dec.	m&s	C_2
127	**0**1111111	**0**1111111
126	**0**1111110	**0**1111110
...	...	...
2	**0**0000010	**0**0000010
1	**0**0000001	**0**0000001
+0	**0**0000000	**0**0000000
-0	**1**0000000	-
-1	**1**0000001	**1**1111111
-2	**1**0000010	**1**1111110
...	...	...
-126	**1**1111110	**1**0000010
-127	**1**1111111	**1**0000001
-128	-	**1**0000000

Invertire un numero in C_2

- L'*inverso additivo* (o *opposto*) –N di un numero N rappresentato in C_2 si ottiene:
  - Invertendo (negando) ogni bit del numero
  - Sommando 1 alla posizione meno significativa
- Esempio:
  - $01011_{C2} = 1\times2^3+1\times2^1+1\times2^0 = 8+2+1 = 11_{dec}$
  - $10100 + 1 = 10101_{C2} = -1\times2^4+1\times2^2+1\times2^0 = -16+4+1 = -11_{dec}$
- Si provi a invertire $11011_{C2} = -5_{dec}$
- Si verifichi che con due applicazioni dell'algoritmo si riottiene il numero iniziale [–(–N) = N] e che lo zero in C2 è (correttamente) opposto di se stesso [–0 = 0]

Conversione dec $\rightarrow$ C_2

- Se $D_{dec} \geq 0$:
  - Converti D_{dec} in binario naturale
  - Premetti il bit 0 alla sequenza di bit ottenuta
  - Esempio: $154_{dec} \Rightarrow 10011010_{bin} \Rightarrow 010011010_{C2}$
- Se $D_{dec} < 0$:
  - Trascura il segno e converti D_{dec} in binario naturale
  - Premetti il bit 0 alla sequenza di bit ottenuta
  - Calcola l'opposto del numero così ottenuto, secondo la procedura di inversione in C_2
  - Esempio: $-154_{dec} \Rightarrow 154_{dec} \Rightarrow 10011010_{bin} \Rightarrow$
    $\Rightarrow 010011010_{bin} \Rightarrow 101100101 + 1 \Rightarrow 101100110_{C2}$
- Occorrono 9 bit sia per 154_{dec} che per -154_{dec}

Aumento e riduzione dei bit in C_2

- ***Estensione*** del segno:
  - *replicando* in modo progressivo il bit di segno a sinistra, il valore del numero non muta

  4 = 0100 = 00100 = 00000100 = … (indefinitamente)

  –5 = 1011 = 11011 = 11111011 = … (indefinitamente)

- ***Contrazione*** del segno:
  - *cancellando* in modo progressivo il bit di segno a sinistra, il valore del numero non muta
  - *purché il bit di segno non abbia a invertirsi* !

  7 = 000111 = 00111 = 0111 **STOP**! (111 è $<$ 0)

  –3 = 111101 = 11101 = 1101 = 101 **STOP**! (01 è $>$ 0)

Osservazioni sul C_2

- Il segno è *incorporato* nel numero rappresentato in C_2, non è semplicemente *applicato* (come in m&s)
- Il bit più significativo *rivela* il segno: 0 per numero positivo, 1 per numero negativo (il numero zero è considerato positivo), ma…
- **NON** si può *distaccare* il bit più significativo e dire che i bit rimanenti rappresentano il valore assoluto del numero
  - questo è ancora vero solo se il numero è positivo

Intervalli di rappresentazione

- Binario naturale a $n \geq 1$ bit: $[0, 2^n)$
- Modulo e segno a $n \geq 2$ bit: $(-2^{n-1}, 2^{n-1})$
- C_2 a $n \geq 2$ bit: $[-2^{n-1}, 2^{n-1})$
  - In modulo e segno, il numero zero ha due rappresentazioni *equivalenti* (00..0, 10..0)
  - L'intervallo del C_2 è *asimmetrico* (-2^{n-1} è compreso, 2^{n-1} è escluso);

Operazioni – **Numeri binari naturali**

Algoritmo di “addizione a propagazione dei riporti”

È l’algoritmo decimale elementare, adattato alla base 2

Pesi	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*		
Riporto			1	1	1					
Addendo 1	0	1	0	0	1	1	0	1	+	77_{dec}
Addendo 2	1	0	0	1	1	1	0	0	=	156_{dec}
Somma	1	1	1	0	1	0	0	1		233_{dec}

addizione naturale (a 8 bit)

Operazioni – **Numeri binari naturali**

***overflow* (o *trabocco*)**

Pesi		*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*		
Riporto	1	1	1	1	1	1					
Addendo 1		0	1	1	1	1	1	0	1	+	125_{dec}
Addendo 2		1	0	0	1	1	1	0	0	=	156_{dec}
Somma		0	0	0	1	1	0	0	1		25_{dec} !

Riporto "perduto"

overflow

risultato errato!

addizione **naturale** con overflow

Riporto e overflow (addizione naturale)

- Si ha ***overflow*** quando il risultato corretto dell'addizione eccede il potere di rappresentazione dei bit a disposizione
  - 8 bit nell'esempio precedente
- Nell'addizione tra numeri binari naturali si ha overflow **ogni volta** che si genera un riporto addizionando i bit della colonna più significativa (riporto "perduto")

Operazioni – **Numeri in C_2**

Pesi	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*		
Riporto			1	1	1					
Addendo 1	0	1	0	0	1	1	0	1	+	77_{dec}
Addendo 2	1	0	0	1	1	1	0	0	=	-100_{dec}
Somma	1	1	1	0	1	0	0	1		-23_{dec}

addizione ***algebrica*** (a 8 bit)

L'algoritmo è ***identico*** a quello naturale

(come se il primo bit non avesse peso negativo)

Operazioni – **Numeri in C_2**

ancora overflow

Pesi	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*		
Riporto	1		1	1	1					
Addendo 1	0	1	0	0	1	1	0	1	+	77_{dec}
Addendo 2	0	1	0	1	1	1	0	0	=	92_{dec}
Somma	1	0	1	0	1	0	0	1		-87_{dec} !

nessun riporto "perduto"

Overflow: risultato negativo!

risultato errato!

addizione **algebrica** con overflow

Riporto e overflow in C_2 (addizione algebrica)

- Si ha ***overflow*** quando il risultato corretto dell'addizione eccede il potere di rappresentazione dei bit a disposizione
  - La definizione di overflow non cambia
- Si può avere overflow senza "riporto perduto"
  - Capita quando da due addendi positivi otteniamo un risultato negativo, come nell'esempio precedente
- Si può avere un "riporto perduto" senza overflow
  - Può essere un innocuo effetto collaterale
  - Capita quando due addendi discordi generano un risultato positivo (**si provi a sommare +12 e -7**)

Rilevare l’overflow in C_2

- Se gli addendi sono tra loro ***discordi*** (di segno diverso) non si verifica mai
- Se gli addendi sono tra loro ***concordi***, si verifica se e solo se il risultato è discorde
  - addendi positivi ma risultato negativo
  - addendi negativi ma risultato positivo
- Criterio di controllo facile da applicare!

Perchè il C_2

Rappresentiamo in modulo e segno -37_{10} su 8 bit

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	0	0	1	0	1

Calcoliamo ora -37 + 1

```
  1 0 1 0 0 1 1 0 1 +
  0 0 0 0 0 0 0 1 =
 ----------------
 1 0 1 0 0 1 1 0   =  -38!!
```

L'operazione di somma fornisce risultati sbagliati quando gli addendi hanno segni diversi.

- E' quindi necessario eseguire qualche verifica in piu'…

Inoltre, in MS lo zero ha due rappresentazioni diverse…

- Un altro caso specifico da gestire!

Rappresentazione ottale ed esadecimale

- *Ottale* o in base otto (oct):
  - Si usano solo le cifre 0-7

$$534_{oct} = 5_{oct} \times 8_{dec}^{2} + 3_{oct} \times 8_{dec}^{1} + 4_{oct} \times 8_{dec}^{0} = 348_{dec}$$

- *Esadecimale* o in base sedici (hex):
  - Si usano le cifre 0-9 e le lettere A-F per i valori 10-15

$$B7F_{hex} = B_{hex} \times 16_{dec}^{2} + 7_{hex} \times 16_{dec}^{1} + F_{hex} \times 16_{dec}^{0} =$$

$$= 11_{dec} \times 16_{dec}^{2} + 7_{dec} \times 16_{dec}^{1} + 15_{dec} \times 16_{dec}^{0} = 2943_{dec}$$

- Entrambe queste basi sono facili da convertire in binario, e viceversa
  - Le basi sono entrambe potenze di 2

Conversioni tra basi

- Per passare da una base B_i a una base B_j è sempre possibile passare attraverso la base 10
- Se B_i e B_j sono una la potenza dell'altra, la trasformazione può avvenire in modo diretto

Conversioni bin → ottale

Corrispondenza biunivoca tra i simboli 0,1, …, 7 e le codifiche 000, 001, …, 111

$010011_{bin} \rightarrow ?_{oct}$

$8=2^3$ →Si raggruppano i bit in sequenze di 3 a partire dal bit meno significativo

Si converte ciascuna tripletta nella cifra corrispondente in base 8.

$010_{bin}\,011_{bin} =$

$2_{oct}\quad 3_{oct}$

Conversioni hex → bin

- Converti: 010011110101011011_{bin} =

  $0001_{bin}\ 0011_{bin}\ 1101_{bin}\ 0101_{bin}\ 1011_{bin}$ =

  = 1_{dec} 3_{dec} 13_{dec} 5_{dec} 11_{dec} =

  = 1_{hex} 3_{hex} D_{hex} 5_{hex} B_{hex} =

  = $13D5B_{hex}$

- Converti: $A7B40C_{hex}$

  A_{hex} 7_{hex} B_{hex} 4_{hex} 0_{hex} C_{hex} =

  (= 10_{dec} 7_{dec} 11_{dec} 4_{dec} 0_{dec} 12_{dec} =)

  = 1010_{bin} 0111_{bin} 1011_{bin} 0100_{bin} 0000_{bin} 1100_{bin} =

  = $101001111011010000001100_{bin}$

Operazioni tra esadecimali

- Si procede come in qualunque altra base, facendo attenzione ai riporti
- Es. somma : $A3D_{16} + CA5_{16}$

1		1		
	A	3	D	+
	C	A	5	=
1	6	E	2	

$D_{16} + 5_{16} = 12$

$D_{16} = 13_{10} \rightarrow (D+5)_{16} = (13+5)_{10} = 18_{10} = 12_{16}$

Per la sottrazione si procede in modo analogo, facendo attenzione ai prestiti

Numeri frazionari in virgola fissa

- $0{,}1011_{bin}$ (in binario)
  $0{,}1011_{bin} = 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3} + 1\times2^{-4} = 1/2 + 1/8 + 1/16 =$
  $= 0{,}5 + 0{,}125 + 0{,}0625 = 0{,}6875_{dec}$
- Si può rappresentare un numero frazionario in *virgola fissa* (o *fixed point*) nel modo seguente:

  $19{,}6875_{dec} = 10011{,}1011_{\text{virgola fissa}}$

  poiché si ha:

  $19_{dec} = 10011_{bin}$

  $0{,}6875_{dec} = 0{,}1011_{bin}$

  N.B.: Per la conversione della parte frazionaria, sia adotta il metodo delle **moltiplicazioni ripetute**

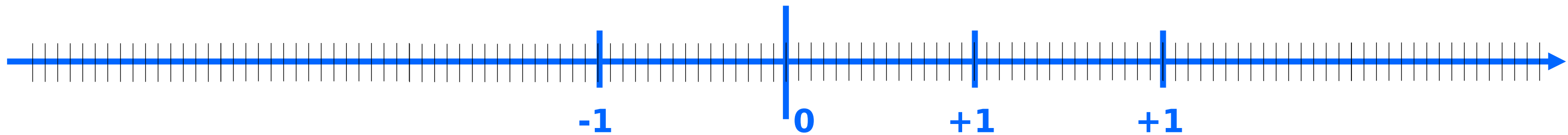
  0,6875 x 2 = 1,375 → 1 riporto 0,375
  0,375 x 2 = 0,75 → 0 riporto 0,75
  0,75 x 2 = 1,5 → 1 riporto 0,5
  0,5 x 2 = 1 → 1

Numeri frazionari in virgola fissa

Si utilizza una proporzione fissa per il nro di bit:

Es. 5 bit per la parte intera, 4 bit per quella frazionaria

- Avremo 2^9 diversi valori codificati, e avremo 2^4 valori tra 0 e 1, 2^4 valori tra 1 e 2, … e così via, con tutti i valori distribuiti su un asse a distanze regolari



Nota: alcuni numeri frazionari con rappresentazione finita in base 10 sono periodici in base 2.

Es.e: 0.6 => 0.10011001100110001... = 0.1001

La rappresentazione binaria può causare troncamento

Numeri frazionari in virgola fissa

- La sequenza di bit rappresentante un **numero frazionario** consta di due parti di lunghezza prefissata
  - Il numero di bit a sinistra e a destra della virgola è stabilito a priori, anche se alcuni bit restassero nulli
- È un sistema di rappresentazione semplice, ma poco flessibile, e può condurre a sprechi di bit
  - Per rappresentare in virgola fissa numeri molto grandi (o molto precisi) occorrono molti bit
  - La precisione nell'intorno dell'origine e lontano dall'origine è la stessa
    - Anche se su numeri molto grandi in valore assoluto la parte frazionaria può non essere particolarmente significativa

Numeri frazionari in virgola mobile

- La rappresentazione in *virgola mobile* (o *floating point*) è usata spesso in base 10 (si chiama allora *notazione scientifica*):

  $0{,}137 \times 10^{8}$ notazione scientifica per intendere $13.700.000_{dec}$

- La rappresentazione si basa sulla relazione

  $$\mathbf{R}_{\text{virgola mobile}} = \mathbf{M} \times \mathbf{B}^{\mathbf{E}}$$ [attenzione: ***non*** $(MxB)^E$]

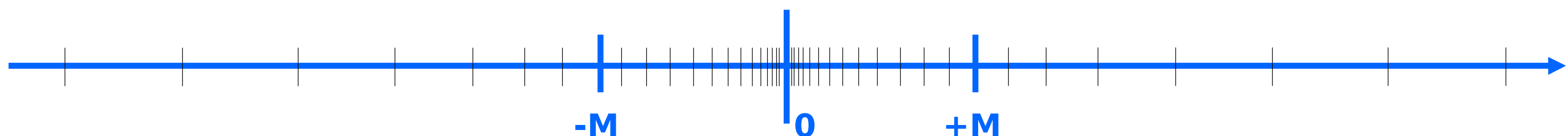
- In binario, si utilizzano **m** ≥ 1 bit per la ***mantissa M*** e **n** ≥ 1 bit per **l'*esponente E***
  - mantissa: un numero frazionario (tra -1 e +1)
  - la base B non è rappresentata (è implicita)
  - in totale si usano m + n bit

Numeri frazionari in virgola mobile

- Esempio
  - Supponiamo B=2, m=3 bit, n=3 bit, M ed E in binario naturale

    $M = 011_2$ ed $E = 010_2$

    $R_{\text{virgola mobile}} = 0{,}011 \times 2^{010} = (1/4 + 1/8) \times 2^2 = 3/8 \times 4 = 3/2 = 1{,}5_{dec}$
- M ed E possono anche essere negativi
  - Normalmente infatti si usa il *modulo e segno* per M, mentre per E si usa la rappresentazione cosiddetta *in eccesso* (qui non spiegata)
- **Vantaggi della virgola mobile**
  - si possono rappresentare con pochi bit numeri molto grandi **oppure** molto precisi (cioè con molti decimali)
  - Sull'asse dei valori i numeri rappresentabili si affollano nell'intorno dello zero, e sono sempre più sparsi al crescere del valore assoluto



Aritmetica standard

- Quasi tutti i calcolatori oggi adottano lo *standard aritmetico* **IEEE 754**, che definisce:
  - I *formati di rappresentazione* binario naturale, C_2 e virgola mobile
  - Gli *algoritmi* di somma, sottrazione, prodotto, ecc, per tutti i formati previsti
  - I metodi di *arrotondamento* per numeri frazionari
  - Come trattare gli *errori* (overflow, divisione per 0, radice quadrata di numeri negativi, ...)
- Grazie a IEEE 754, i programmi sono *trasportabili* tra calcolatori diversi senza che cambino né i *risultati* né la *precisione* dei calcoli svolti dal programma stesso

Standard IEEE 754-1985

S	E	M

- Bit destinati alla rappresentazione divisi in
  - un bit per il segno della mantissa – parte S (0 = +, 1 = -)
  - alcuni bit per l'esponente – parte E
  - altri bit per la mantissa (il suo valore assoluto) – parte M
- Problema: il segno dell'esponente notazione "eccesso K"
  - si memorizza il valore dell'esponente aumentato di K
  - se n bit dedicati all'esponente, $K = 2^{n-1} - 1$
  - es: n=8 si memorizza esponente aumentato di $K=2^7-1=127$
- $\Rightarrow$ valore memorizzato 0: esponente = -127;
  255: esponente = 128;
  132: esponente = 5
- Inoltre, Mantissa viene normalizzata:
  - scegliendo esponente opportuno, posta a un valore (binario) tra 1.00000... e 1.11111…
  - il valore 1 sempre presente può essere sottinteso $\Rightarrow$ guadagno di un bit di precisione

Previsti tre possibili gradi di precisione: singola, doppia, quadrupla

Campo	Precisione singola	Precisione doppia	Precisione quadrupla
ampiezza totale in bit di cui	32	64	128
Segno	1	1	1
Esponente	8	11	15
Mantissa	23	52	111
massimo E	255	2047	32767
minimo E	0	0	0
K	127	1023	16383

Il valore rappresentato vale quindi $X = (-1)^S \times 2^{E-K} \times 1.M$

Esempio

- Esempio di rappresentazione in precisione singola
- $X = 42.6875_{10} = 101010.1011_2 = 1.010101011 \times 2^5$
- Si ha
  - $S = 0$ (1 bit)
  - $E = 5 + K = 5_{10} + 127_{10} = 132 = 10000100_2$ (8 bit)
  - $M = 01010101100000000000000$ (23 bit)

Proprietà fondamentale

- I circa 4 miliardi di configurazioni dei 32 bit usati consentono di coprire un campo di valori molto ampio grazie alla distribuzione non uniforme.
- Per numeri piccoli in valore assoluto valori rappresentati sono «fitti»,
- Per numeri grandi in valore assoluto valori rappresentati sono «diradati»

- Approssimativamente gli intervalli tra **valori contigui** sono
  - per valori di 10000 l’intervallo è di un millesimo
  - per valori di 10 milioni l’intervallo è di un’unità
  - per valori di 10 miliardi l’intervallo è di mille

Non solo numeri!
codifica dei caratteri

- Nei calcolatori i caratteri vengono *codificati* mediante *sequenze* di $n \geq 1$ bit, ognuna rappresentante un carattere distinto
  - Corrispondenza biunivoca tra numeri e caratteri
- Codice ASCII (*American Standard Computer Interchange Interface*): utilizza $n=7$ bit per 128 caratteri
- Il codice ASCII a 7 bit è pensato per la lingua inglese. Si può estendere a 8 bit per rappresentare il doppio dei caratteri
  - Si aggiungono così, ad esempio, le lettere con i vari gradi di accento (come À, Á, Â, Ã, Ä, Å, ecc), necessarie in molte lingue europee, e altri simboli speciali ancora
- Varie versioni a carattere nazionale

Alcuni simboli del codice ASCII

# (in base 10)	Codifica (7 bit)	Carattere (o simbolo)
0	0000000	<terminator>
9	0001001	<tabulation>
10	0001010	<carriage return>
12	0001100	<sound bell>
13	0001101	<end of file>
32	0100000	blank space
33	0100001	!
49	0110001	1
50	0110010	2
64	1000000	@
65	1000001	A
66	1000010	B
97	1100000	a
98	1100001	b
126	1111110	~
127	1111111	Ω

Unicode

- Assegna un numero univoco ad ogni carattere usato per la scrittura di testi, in maniera indipendente dalla lingua
- Il codice assegnato al carattere viene rappresentato con U+, seguito dalle quattro (o sei) cifre esadecimali del numero che lo individua
- Repertorio di codici numerici che possono rappresentare circa un milione di caratteri

Altre codifiche alfanumeriche

- Codifica **ASCII** esteso a 8 bit (256 parole di codice). È la più usata.
- Codifica **FIELDATA** (6 bit, 64 parole codificate) Semplice ma compatta, storica
- Codifica **EBDC** (8 bit, 256 parole codifiate) Usata per esempio nei nastri magnetici
- Codifiche **ISO-X** (rappresentano i sistemi di scrittura internazionali). P. es.: ISO-LATIN

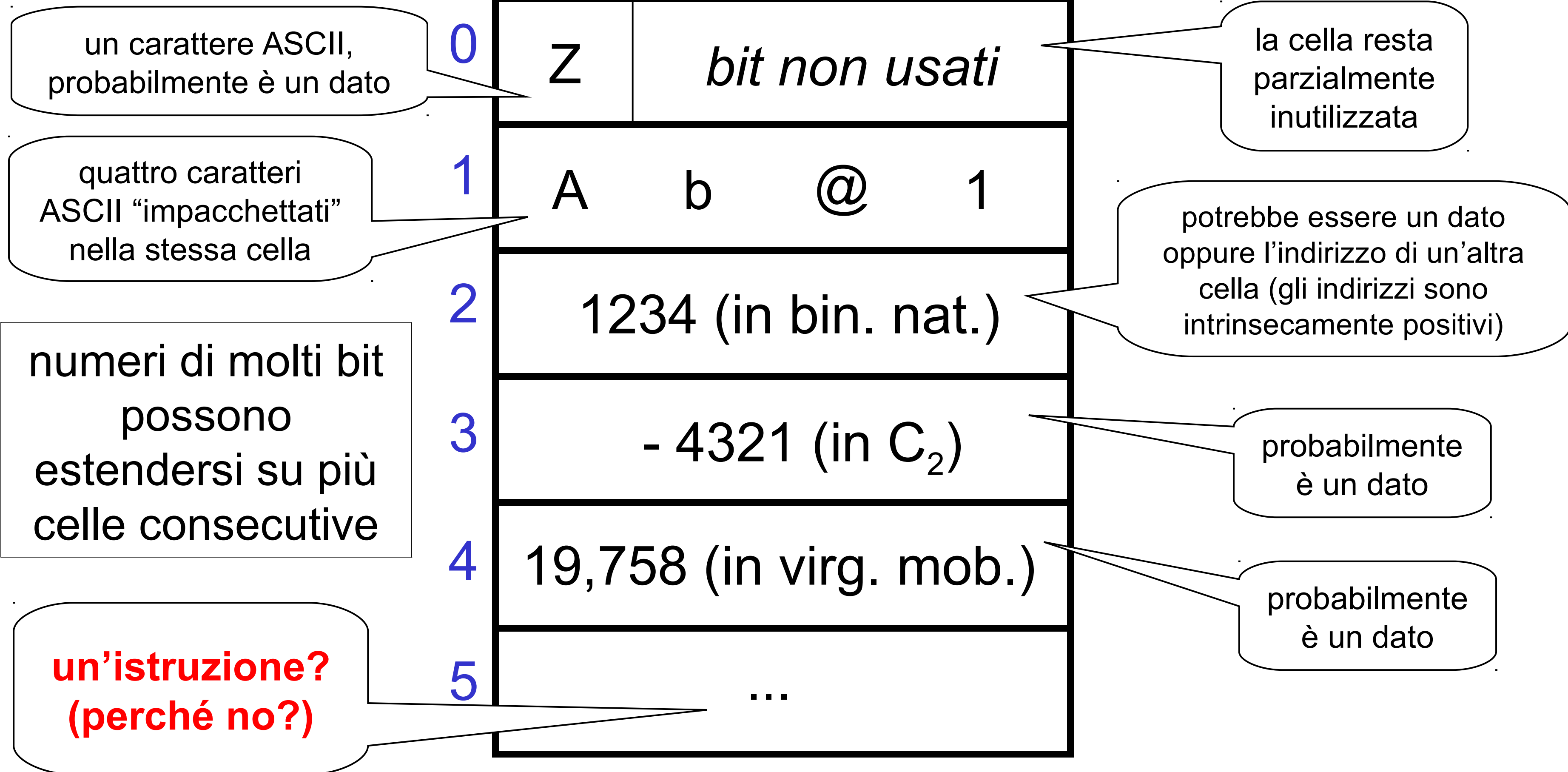
Codifica di testi, immagini, suoni, ...

- Caratteri: sequenze di bit
  - Codice ASCII: utilizza 7(8) bit: 128(256) caratteri
  - 1 Byte (l'8° bit può essere usato per la *parità*)
- Testi: sequenze di caratteri (cioè di bit)
- Immagini: sequenze di bit
  - bitmap: sequenze di pixel (n bit, 2^n colori)
  - jpeg, gif, pcx, tiff, …
- Suoni (musica): sequenze di bit
  - wav, mid, mp3, ra, …

Dentro al calcolatore... Informazione e memoria

- Una *parola di memoria* è in grado di contenere una *sequenza* di n ≥ 1 bit
- Di solito si ha: n = 8, 16, 32 o 64 bit
- Una parola di memoria può dunque contenere gli *elementi d'informazione* seguenti:
  - Un carattere (o anche più di uno)
  - Un numero intero in binario naturale o in C_2
  - Un numero frazionario in virgola mobile
  - Alcuni bit della parola possono essere non usati
- Lo stesso può dirsi dei registri della CPU

Per esempio …



Algebra di Boole
ed
Elementi di Logica

Cenni all’algebra di Boole

- *Algebra di Boole* (inventata da G. Boole, britannico, seconda metà ’800), o *algebra della logica*
- Regole per il calcolo logico basato su *operazioni logiche*
  - applicabili a *operandi logici*, cioè a operandi in grado di assumere solo i valori ***vero*** (1) *e* ***falso*** (0)
- Base per il funzionamento dei moderni calcolatori
  - Qualsiasi informazione è rappresentata tramite sequenze di valori binari
  - I circuiti complessi del calcolatore sono realizzati combinando numerosissimi circuiti elementari che implementano operazioni logiche
- Base per l’espressione di condizioni nei linguaggi di programmazione

Operazioni logiche fondamentali

- Operatori logici binari (con 2 operandi logici)
  - Operatore **OR**, o *somma logica*
  - Operatore **AND**, o *prodotto logico*
- Operatore logico unario (con 1 operando)
  - Operatore **NOT**, o *negazione*, o *inversione*

Operatori logici di base e loro tabelle di verità

Poiché gli operandi logici ammettono due soli valori, si può definire compiutamente ogni operatore logico tramite una ***tabella*** di associazione operandi-risultato

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

(somma logica)

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

(prodotto logico)

A	not A
0	1
1	0

(negazione)

Le tabelle elencano tutte le possibili combinazioni in ingresso e il risultato associato a ciascuna combinazione

Espressioni logiche (o Booleane)

- Come le espressioni algebriche, costruite con:
  - Variabili logiche (letterali): p. es. A, B, C = 0 oppure 1
  - Operatori logici: and, or, not
- Esempi:

  A or (B and C)

  (A and (not B)) or (B and C)
- **Precedenza**: l'operatore "not" precede l'operatore "and", che a sua volta precede l'operatore "or"

  A and not B or B and C = (A and (not B)) or (B and C)
- Per ricordarlo, si pensi OR come "+" (più), AND come "×" (per) e NOT come "–" (cambia segno)

Tabelle di verità delle **espressioni logiche**

A	B	NOT ((A OR B) AND (NOT A))
0	0	1
0	1	0
1	0	1
1	1	1

Specificano i valori di verità
per tutti i possibili valori delle variabili

Tabella di verità
di un’espressione logica

A and B or not C

A B C	**X** = A and B	**Y** = not C	**X** or **Y**			
0 0 0	0 and 0 = 0	not 0 = 1	0	or	1	**= 1**
0 0 1	0 and 0 = 0	not 1 = 0	0	or	0	**= 0**
0 1 0	0 and 1 = 0	not 0 = 1	0	or	1	**= 1**
0 1 1	0 and 1 = 0	not 1 = 0	0	or	0	**= 0**
1 0 0	1 and 0 = 0	not 0 = 1	0	or	1	**= 1**
1 0 1	1 and 0 = 0	not 1 = 0	0	or	0	**= 0**
1 1 0	1 and 1 = 1	not 0 = 1	1	or	1	**= 1**
1 1 1	1 and 1 = 1	not 1 = 0	1	or	0	**= 1**

Due esercizi

A	B	NOT	((A	OR	B)	AND	(NOT	A))
0	0	**1**	0	0	0	0	1	0
0	1	**0**	0	1	1	1	1	0
1	0	**1**	1	1	0	0	0	1
1	1	**1**	1	1	1	0	0	1

A	B	C	(B	OR	NOT	C)	AND	(A	OR	NOT	C)
0	0	0	0	1	1	0	**1**	0	1	1	0
0	0	1	0	0	0	1	**0**	0	0	0	1
0	1	0	1	1	1	0	**1**	0	1	1	0
0	1	1	1	1	0	1	**0**	0	0	0	1
1	0	0	0	1	1	0	**1**	1	1	1	0
1	0	1	0	0	0	1	**0**	1	1	0	1
1	1	0	1	1	1	0	**1**	1	1	1	0
1	1	1	1	1	0	1	**1**	1	1	0	1

A che cosa servono le espressioni logiche?

- A *modellare* alcune (non tutte) forme di *ragionamento*
  - A = è vero che 1 è maggiore di 2 ? (sì o no, qui è no) = 0
  - B = è vero che 2 più 2 fa 4 ? (sì o no, qui è sì) = 1
  - A and B = è vero che 1 sia maggiore di 2 e che 2 più 2 faccia 4 ?
    Si ha che A and B = 0 and 1 = 0, dunque no
  - A or B = è vero che 1 sia maggiore di 2 o che 2 più 2 faccia 4 ?
    Si ha che A or B = 0 and 1 = 1, dunque sì
- OR, AND e NOT vengono anche chiamati *connettivi logici,* perché funzionano come le congiunzioni coordinanti “o” ed “e” e come la negazione “non” del linguaggio naturale
- Si modellano ragionamenti (o *deduzioni*) basati solo sull’uso di “o”, “e” e “non” (non è molto, ma è utile)

Che cosa **non** si può modellare tramite espressioni logiche?

- Le espressioni logiche (booleane) *non modellano*:
  - Domande *esistenziali*: "***c'è almeno*** un numero reale x tale che il suo quadrato valga –1 ?" (si sa bene che *non c'è*)
    $\exists x \mid x^2 = -1$ è falso
  - Domande *universali*: "***ogni*** numero naturale è la somma di quattro quadrati di numeri naturali ?" (si è dimostrato *di sì*)
    $\forall x \mid x = a^2+b^2+c^2+d^2$ è vero ("teorema dei 4 quadrati")
    Più esattamente andrebbe scritto: $\forall x\ \exists a,b,c,d \mid x = a^2+b^2+c^2+d^2$
- $\exists$ e $\forall$ sono chiamati "operatori di quantificazione", e sono ben diversi da or, and e not
- La parte della logica che tratta solo degli operatori or, and e not si chiama ***calcolo proposizionale***
- Aggiungendo gli operatori di quantificazione, si ha il ***calcolo dei predicati*** (che è molto più complesso)

Tautologie e Contraddizioni

- *Tautologia*
  - Una espressione logica che è sempre **vera**, per qualunque combinazione di valori delle variabili
    - Esempio: principio del “terzo escluso”: **A or not A** (*tertium non datur*, non si dà un terzo caso tra l’evento A e la sua negazione)
- C*ontraddizione*
  - Una espressione logica che è sempre **falsa**, per qualunque combinazione di valori delle variabili
    - Esempio: principio di “non contraddizione”: **A and not A** (l’evento A e la sua negazione non possono essere entrambi veri)

Equivalenza tra espressioni

- Due espressioni logiche si dicono ***equivalenti*** (e si indica con ⇔) **se hanno la medesima tabella di verità**. La verifica è *algoritmica*. Per esempio:

A B	not A and not B	⇔	not (A or B)
0 0	1 and 1 = 1		not 0 = 1
0 1	1 and 0 = 0		not 1 = 0
1 0	0 and 1 = 0		not 1 = 0
1 1	0 and 0 = 0		not 1 = 0

- Espressioni logiche equivalenti modellano gli stessi *stati di verità* a fronte delle medesime variabili

Proprietà dell'algebra di Boole

- L'algebra di Boole gode di svariate *proprietà*, formulabili sotto specie di *identità*
  - (cioè formulabili come equivalenze tra espressioni logiche, valide per qualunque combinazione di valori delle variabili)
- Esempio celebre: le "Leggi di De Morgan"

  not (A **and** B) = not A **or** not B (1a legge)

  not (A **or** B) = not A **and** not B (2a legge)

Ancora sulle proprietà

- Alcune proprietà somigliano a quelle dell'algebra numerica tradizionale:
  - Proprietà *associativa*: A or (B or C) = (A or B) or C (idem per AND)
  - Proprietà *commutativa*: A or B = B or A (idem per AND)
  - Proprietà *distributiva* di AND rispetto a OR:
    A and (B or C) = A and B or A and C
  - Proprietà *distributiva* di OR rispetto a AND:
    A or B and C = (A or B) and (A or C)
  - … e altre ancora
- Ma parecchie altre sono alquanto insolite…
  - Proprietà di *assorbimento* (A assorbe B):
    A or A and B = A
  - *Legge dell'elemento* 1: not A or A = 1
  - … e altre ancora

Uso delle proprietà

- *Trasformare* un'espressione logica in un'altra, differente per aspetto ma equivalente:

not A and B or A =	(assorbimento)
= not A and B or (A or A and B) =	(togli le parentesi)
= not A and B or A or A and B =	(commutativa)
= not A and B or A and B or A =	(distributiva)
= (not A or A) and B or A =	(legge dell'elemento 1)
= ***true*** and B or A =	(vero and B = B)
= B or A	è più semplice dell'espressione originale !

- Si *verifichi* l'equivalenza con le tabelle di verità!
- Occorre conoscere un'ampia lista di proprietà e si deve riuscire a "vederle" nell'espressione (qui è il difficile)