Corso di Laurea triennale in Ingegneria Chimica

in condivisione con

Corso di Laurea triennale in

Ingegneria Navale e Scienze dei Materiali

# Elementi di Informatica

A.A. 2016/17

prof. Mario Barbareschi

Rappresentazione e codifica delle informazioni – Parte Seconda

# Informazioni di Licenza

# Codifiche dei calcolatori e sistema binario

- Abbiamo visto che i calcolatori operano su **codifiche binarie** dei dati.
  - Per **definire una codifica occore creare una tabella-codice** per mettere in relazione **l'alfabeto origine** con le **parole-codice**.
- **Se le codifiche degli insiemi numerici** sono scelte in modo da **basarsi sul sistema di numerazione posizionale pesato binario**, allora:
  - Le **operazioni di calcolo** sulle **parole-codice** possono basarsi **sull'aritmetica binaria e NON su associazioni arbitrarie** tra parole-codice!
- Ciò si traduce in un forte vantaggio, poiché tutte le funzioni aritmetiche preservano le loro proprietà ed osservano le medesime regole dell'aritmetica decimale!

# Codifica di numeri interi $\mathbb{Z}$ in segno e modulo

- La **codifica binaria in segno e modulo** associa ad ogni numero $z^* \in \mathbb{Z}^*$ parole-codice a **<u>lunghezza fissa</u>** secondo le seguenti regole:
  – Usa un bit per rappresentare il **segno** di $z^*$ ("+" oppure "–" è una informazione binaria).
  – Usa i **rimanenti bit** della parola-codice per **codificare il modulo** $|z^*|$ secondo il **sistema di numerazione posizionale binario**.
- Usando parole-codice di **lunghezza $l$** possiamo codificare i numeri nell'intervallo:

$$\mathbb{Z}^* = [-2^{l-1} + 1; 2^{l-1} - 1];$$

$$|\mathbb{Z}^*| = (2^{l-1} - 1) - (-2^{l-1} + 1) + 1 = 2^l - 1$$

- Notiamo che con parole codice di lunghezza $l$ codifichiamo $2^l - 1$ valori e non $2^l$, ovvero c'è almeno una parola codice che non usiamo…
  - Questa parola codice non usata è quella usata per codificare lo "zero con consegno negativo"!

# Esempio codifica segno e modulo

- Codificare in segno e modulo su 1 byte i seguenti numeri interi:

Tabella-codice per codifica in segno e modulo con $l = 3$

| | | Parole-Codice di lunghezza = 3 (P) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Alfabeto origine ($T=\mathbb{Z}^*$) | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| | -3 | | | | | | | | • |
| | -2 | | | | | | | • | |
| | -1 | | | | | | • | | |
| | 0 | • | | | | • | | | |
| | 1 | | • | | | | | | |
| | 2 | | | • | | | | | |
| | 3 | | | | • | | | | |

- Esempi:
  - $(+73)_{10} = (01001001)_2 = +1001001$
  - $(-42)_{10} = (10101010)_2 = -0101010$

# Osservazioni sulla codifica segno e modulo

- La codifica segno e modulo, sebbene immediata, conduce a scomode condizioni causa obbligatoria gestione del **doppio 0**:
  - +0, cioè la codifica dello 0 preceduta da uno 0;
  - -0, cioè la codifica dello 0 preceduta da un 1;
- La gestione di questo specifico caso ha un costo in sia in termini computazionali (efficienza) sia in configurazioni (due rappresentazioni per lo stesso simbolo).

| | | Parole-Codice di lunghezza = 3 ($P$) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Alfabeto origine ($T=\mathbb{Z}^*$) | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| | -3 | | | | | | | | • |
| | -2 | | | | | | | • | |
| | -1 | | | | | | • | | |
| | 0 | • | | | | • | | | |
| | 1 | | • | | | | | | |
| | 2 | | | • | | | | | |
| | 3 | | | | • | | | | |

# Codifica di interi $\mathbb{Z}$ in complemento alla base

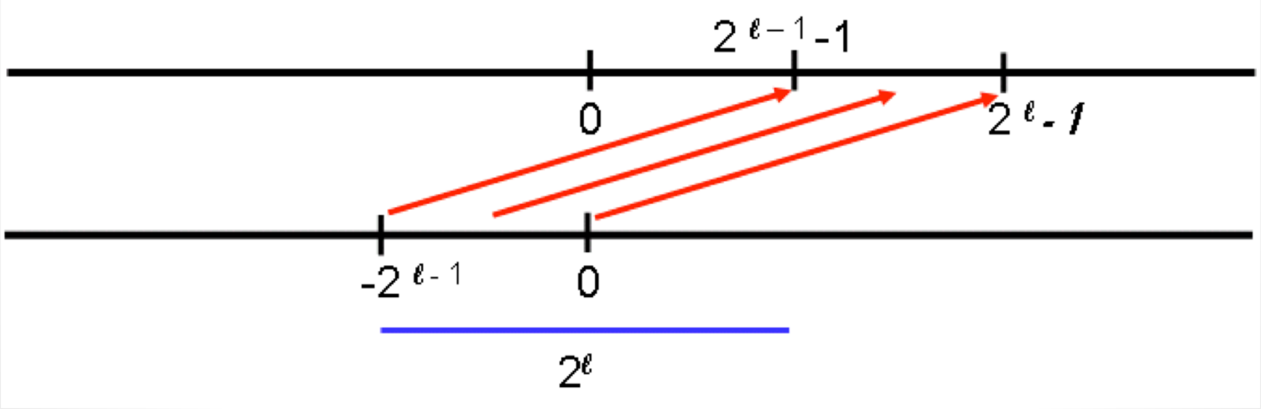
- La codifica in **complemento alla base** associa ad ogni numero $z^* \in \mathbb{Z}^*$ **parole-codice a lunghezza fissa** $l$ secondo le seguenti regole:
  - Gli **interi positivi** $z^*$ nell'intervallo $[0, b^{l-1}-1]$ sono codificati secondo il sistema di numerazione binario;
  - Per gli **interi negativi** $z^*$ nell'intervallo $[-b^{l-1},-1]$ si codifica in binario il valore del loro complemento alla base $b^l$ (ovvero $z^{**}=b^l-|z^*|$);
- In altre parole, si adotta una traslazione dell'insieme dei numeri $[0, b^l]$ in avanti, in modo tale che i numeri negativi vengano rappresentati con codifica positiva.

Complemento di n $\rightarrow$ $b^l$-n

- Esempio di complementi a 10:
  - $(+73)_{10}$ = 073 (Positivo)
  - $(-42)_{10}$ = 1000-42= 958 (Negativo!)

# Codifica di interi $\mathbb{Z}$ in complemento a due

- La codifica in **complemento a due** associa ad ogni numero binario $z^* \in \mathbb{Z}^*$ **parole-codice binarie a lunghezza fissa** $l$:
  - Gli **interi positivi** $z^*$ nell'intervallo $[0, 2^{l-1}-1]$ sono codificati secondo il sistema di numerazione binario;
  - Per gli **interi negativi** $z^*$ nell'intervallo $[-2^{l-1},-1]$ si codifica in binario il valore del loro complemento alla base $2^l$ (ovvero $z^{**}=2^l-|z^*|$);




Esempi di codifiche complementi a due su 3 bit: $2^l = 2^3 = 8$

$(+3)_{10}$ POSITIVO, converto in binario $3 \rightarrow 011$

$(-2)_{10}$ NEGATIVO, converto in binario $2^l - |z^*| = 8 - 2 = 6 \rightarrow 110$

# Esempio parole-codice in complementi a due

Codificare in complementi a due su 1 byte i seguenti numeri interi:

$(+73)_{10}$ Positivo, converto in binario!
→ 01001001

$(-42)_{10}$ Negativo!
converto $2^l$ - |-42| =
256 - 42 = 214
→ 11010110

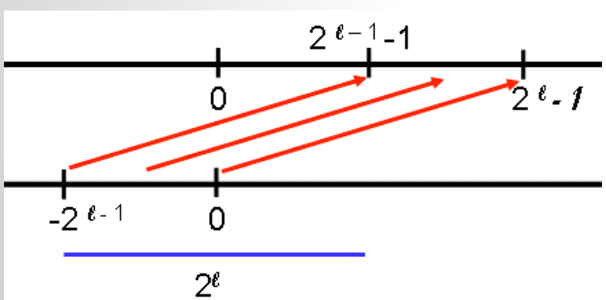



Tabella-codice per codifica complemento a 2 con $l = 3$

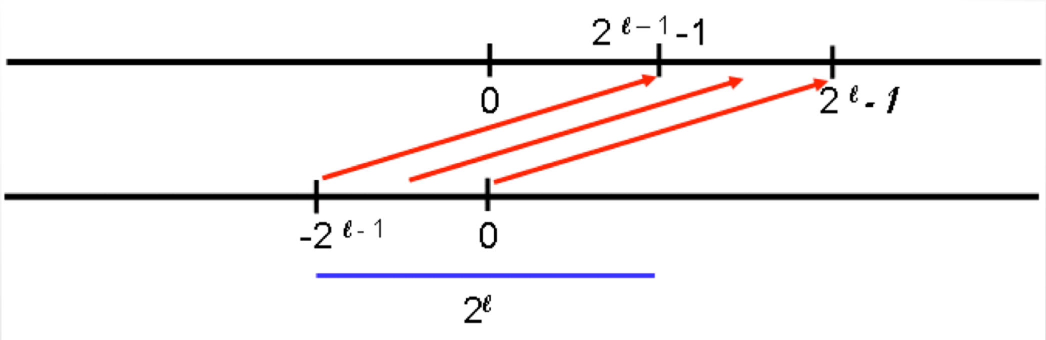
| | | Parole-Codice di lunghezza = 3 (P) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Alfabeto origine ($T=\mathbb{Z}^*$) | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| | -4 | | | | | • | | | |
| | -3 | | | | | | • | | |
| | -2 | | | | | | | • | |
| | -1 | | | | | | | | • |
| | 0 | • | | | | | | | |
| | 1 | | • | | | | | | |
| | 2 | | | • | | | | | |
| | 3 | | | | • | | | | |

# Codifica in complementi a due: osservazioni (1/2)

- Questa codifica permette di rappresentare interi in un intervallo non simmetrico:

$$\mathbb{Z}^{*} = [-2^{l-1}; 2^{l-1} - 1];$$
$$|\mathbb{Z}^{*}| = (2^{l-1} - 1) - (-2^{l-1}) + 1 = 2^{l}$$

- Siccome il complemento effettua una traslazione dei numeri negativi nell'intervallo $[2^{l-1}; 2^{l} - 1]$ i numeri negativi si riconoscono dal bit più significativo posto ad 1 (diventano maggiori uguali a $2^{l-1}$!)
  – A tutti gli effetti anche questa codifica ha un bit che discrimina il segno!



# Codifica in complementi a due: osservazioni (2/2)

- La codifica binaria in complementi a due è più complessa ma:
  - Tutte le parole-codice sono usate per la codifica.
  - Permette di discriminare i numeri negativi dai positivi in base al bit più significativo.
  - Sfruttando l’overflow si possono sommare numeri positivi e negativi senza badare al segno degli addendi (con risparmi in termini circuitali).

| | Parole-Codice di lunghezza = 3 (P) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Alfabeto origine ($T=\mathbb{Z}^*$) | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| | -4 | | | | | • | | | |
| | -3 | | | | | | • | | |
| | -2 | | | | | | | • | |
| | -1 | | | | | | | | • |
| | 0 | • | | | | | | | |
| | 1 | | • | | | | | | |
| | 2 | | | • | | | | | |
| | 3 | | | | • | | | | |

Esempio di somma nella codifica complementi a due di lunghezza 3 bit:

$$(+3)_{10} + (-2)_{10}$$

$$\rightarrow \quad 011 +$$

$$110 =$$

$$\underline{1}001 \text{ overflow!}$$

$$= \quad 001 = (1)_{10}$$

# Esempi di sottrazione in complemento a due

$(+73)_{10} \rightarrow 01001001$ $(-42)_{10} \rightarrow 11010110$

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $(+73)_{10}$ | | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | + |
| $(-42)_{10}$ | | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | = |
| Overflow! | $\cancel{1}$ | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | |

| $b^i$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|---|
| $c^i$ | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| $(c_i \times b^i)_{10}$ | 0 | 0 | 0 | 16 | 8 | 4 | 2 | 1 |
| $\sum_i (c_i \times b^i)_{10}$ | 16+8+4+2+1 = 31 | | | | | | | |

# Esempio di sottrazione in complemento a 10

Mostriamo la stessa operazione usando una codifica in complementi alla base 10 su 3 cifre!

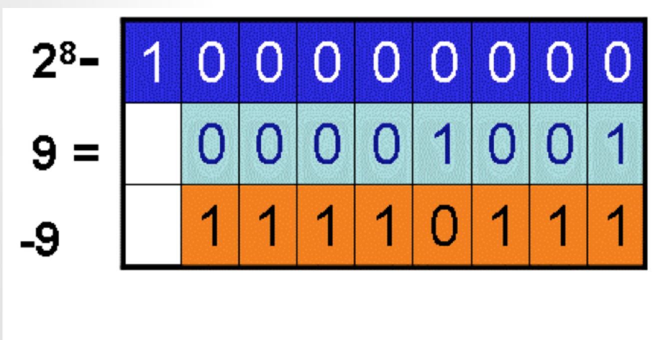
Per la sottrazione siamo stati abituati ad eseguire un algoritmo diverso dalla somma: $73 - 42 = 31$
Codificando in complementi alla base possiamo applicare l'algoritmo della somma anche nel sistema decimale per eseguire sottrazioni nell'intervallo rappresentabile:

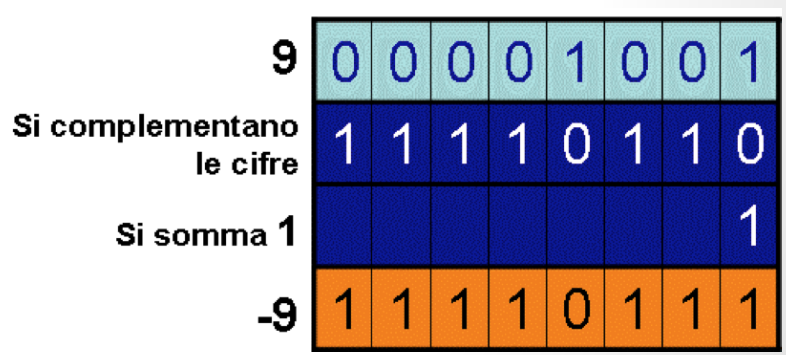
$(+73)_{10} \rightarrow 073$ $(-42)_{10} \rightarrow 958$

| | | | | | |
|---|---|---|---|---|---|
| $(+73)_{10}$ | | 0 | 7 | 3 | + |
| $(-42)_{10}$ | | 9 | 5 | 8 | = |
| Overflow! | ~~1~~ | 0 | 3 | 1 | |

# Codifica in complementi a due: calcolo dell’opposto

- Per trovare l’opposto della somma (ovvero per passare da un numero positivo al suo negativo e viceversa):
  - O si calcola direttamente il complemento alla base, effettuando la sottrazione tra la base ed il valore.
  - Oppure si applica la seguente regola:
    1. Si complementa il numero (si invertono i bit 1/0)
    2. **Si corregge il risultato sommando +1**



# Complemento diminuito

- Il **complemento alla base diminuito (detto anche complemento a 1)** è una codifica simile al complemento a due, eccetto che le codifiche dei numeri negativi sono "corrette" sottraendo 1:

$$z^{***} = z^{**} - 1$$

- Il vantaggio di tale codifica è che l'opposto di ogni numero è semplicemente il suo complemento (si evita la "**correzione**" per trovare l'opposto della somma).
- È poco in uso perché l'intervallo di rappresentazione diventa simmetrico non assegnando una parola-codice (il complemento diminuito dello zero), e ciò complica le operazioni di somma e sottrazione.

Tabella-codice per codifica complemento a 2 diminuito con $l = 3$

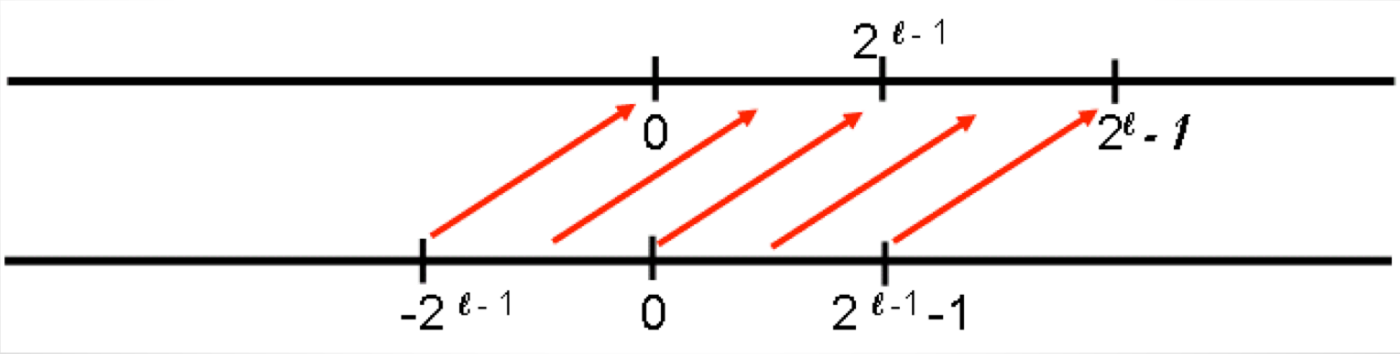
| | | Parole-Codice di lunghezza = 3 ($\mathbb{P}$) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Alfabeto origine ($\mathbb{T}=\mathbb{Z}^*$) | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| | -3 | | | | | • | | | |
| | -2 | | | | | | • | | |
| | -1 | | | | | | | • | |
| | 0 | • | | | | | | | |
| | 1 | | • | | | | | | |
| | 2 | | | • | | | | | |
| | 3 | | | | • | | | | |

# Complemento diminuito: osservazioni

- Un ulteriore svantaggio è la doppia rappresentazione dello 0, dovuta proprio alla sua simmetria!
- Il complemento diminuito di una data rappresentazione si ottiene invertendo una ad una le cifre binarie per rappresentare il suo opposto:
  – E.g.: 56 → 00111000
  
    -56 → 11000111
- Se sommassimo (56) e (-56) otterremmo (1111111), codifica in complementi dello 0 (tutte le cifre binarie dello 0 sono invertite!).

# Rappresentazione per eccessi

- Questa rappresentazione codifica in binario i numeri nell'intervallo $[-2^{l-1}, 2^{l-1}-1]$ sommandovi la costante $2^{l-1}$ per traslare l'intervallo dei numeri negativi in positivo, ovvero nell'intervallo $[0, 2^{l}-1]$.
  – Ne deriva che lo zero sarà associato a $2^{l-1}$ mentre i valori minori di $2^{l-1}$ ai numeri negativi e quelli maggiori a quelli positivi: anche in questo caso il primo bit permette di discriminare il segno, ma in modo invertito rispetto alla codifica in complementi alla base.



# Codifiche di numeri interi

| Valore Decimale di N | N in binario (1 byte) | -N in segno e modulo | -N in complemento a 1 | -N in complemento a 2 | -N in eccesso |
|---|---|---|---|---|---|
| 0 | 00000000 | 10000000 | 11111111 | Non rappresentabile | 10000000 |
| 1 | 00000001 | 1000001 | 11111110 | 11111111 | 01111111 |
| 10 | 00001010 | 10001010 | 11110101 | 11110110 | 01110110 |
| 50 | 00110010 | 10110010 | 11001101 | 11001110 | 01001110 |
| 100 | 01100100 | 11100100 | 10011011 | 10011100 | 00011100 |
| 127 | 01111111 | 11111111 | 10000000 | 10000001 | 00000001 |
| 128 | 10000000 | Non rappresentabile | Non rappresentabile | 10000000 | 00000000 |

Doppia rappresentazione dello 0

# Codifiche di interi e calcolatori

- Generalmente per rappresentare numeri naturali $\mathbb{N}^*$ i calcolatori impiegano il sistema di numerazione binario, mentre per numeri naturali $\mathbb{Z}^*$ adottano codifiche in complemento alla base 2.

| **Bit** | **Byte** | $\mathbb{N}^* = [0; 2^l - 1]$ | $\mathbb{Z}^* = [-2^{l-1}; 2^{l-1} - 1]$ |
|---|---|---|---|
| 8 | 1 | [0; 255] | [-128; 127] |
| 16 | 2 | [0; 65˙535] | [-32˙768; -32˙767] |
| 32 | 4 | [0; 4˙294˙967˙295] | [-2˙147˙483˙648; 2˙147˙483˙647] |
| 64 | 8 | [0; 18˙446˙744˙073˙709˙551˙615] | [-9˙223˙372˙036˙854˙775˙808; 9˙223˙372˙036˙854˙775˙807] |

- Le rappresentazioni in complemento a due ed eccesso sono le più efficienti per svolgere operazioni in aritmetica binaria poiché permettono di trattare la sottrazione tra numeri come una somma tra numeri di segno opposto:

$$\mathbf{(X - Y) = (X + (-Y))}$$

  – È così possibile eseguire solo addizioni per risolvere anche le sottrazioni, cioè si utilizza un unico circuito elettronico!

# Rappresentazione di numeri reali $\mathbb{R}$ a virgola mobile

- I numeri reali $\mathbb{R}$ si rappresentano **a virgola mobile** attraverso la notazione:

$$r = m \times b^e$$

  - $m$ è un numero frazionario, detto **mantissa**.
  - $b$ è la **base**.
  - $e$ è un numero intero, chiamato **esponente** o caratteristica.
- La notazione a virgola mobile permette di rappresentare una ampio intervallo di valori con lo stesso numero di cifre, grazie alla **flessibilità della posizione della virgola**, che dipende dal valore dell'esponente.
  - Esempio: considerando 3 cifre per la mantissa e 2 per l'esponente
    - PI Greco: $0.314 \times 10^1$
    - Massa di un Elettrone (kg): $0.911 \times 10^{-30}$
    - Massa della Via Lattea (kg): $0.136 \times 10^{43}$

# Valori rappresentabili in virgola mobile

- La **virgola mobile** si differenzia dalla notazione scientifica per l'uso di rappresentazioni finite e il modo di utilizzare i numeri.

$$r = m \times b^e$$

**Le cifre dell'esponente** determinano **l'ampiezza** dell'intervallo di valori rappresentabili $\mathbb{R}^*$ e **il più piccolo numero diverso da zero rappresentabile** ($r_{\min}$).

Esempi:

**Considerando** $\boldsymbol{m \in [0; 1]}$

1 cifra: $|r| < 10^{10}$ $\quad r_{\min} = m_{\min} \times 10^{-9}$

2 cifre: $|r| < 10^{100}$ $\quad r_{\min} = m_{\min} \times 10^{-99}$

3 cifre: $|r| < 10^{1000}$ $\quad r_{\min} = m_{\min} \times 10^{-999}$

**Le cifre della mantissa** determinano la **distanza** e la **quantità di numeri rappresentabili** nell'intervallo.

Esempi di numeri adiacenti nell'intervallo:

2 cifre: $0.40 \times 10^{-30}$ e $0.41 \times 10^{-30}$ distanza: $0.01 \times 10^{-30}$

3 cifre: $0.404 \times 10^{-30}$ e $0.405 \times 10^{-30}$ distanza: $0.001 \times 10^{-30}$

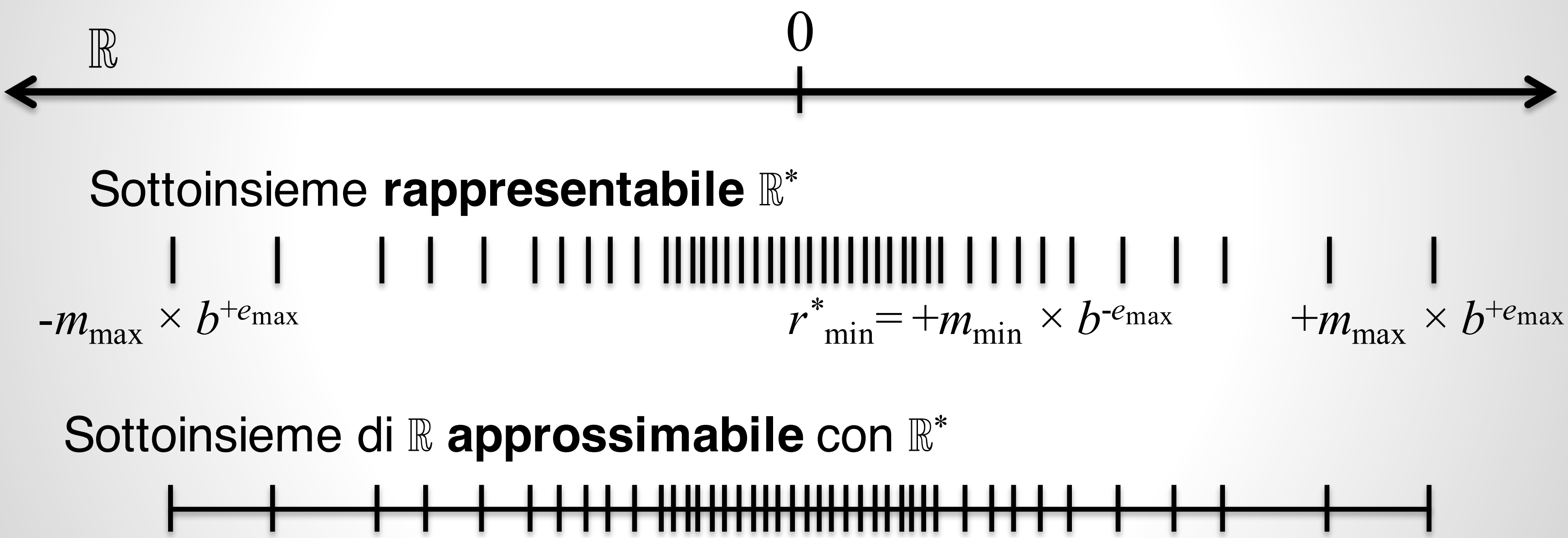
2 cifre: $0.40 \times 10^{15}$ e $0.41 \times 10^{15}$ distanza: $0.01 \times 10^{15}$

3 cifre: $0.404 \times 10^{15}$ e $0.405 \times 10^{15}$ distanza: $0.001 \times 10^{15}$

I valori rappresentabili **NON sono equidistanti** nell'intervallo di rappresentazione!

# La discretizzazione di $\mathbb{R}$

- Essendo l'insieme reale **denso**, per ogni coppia di elementi distinti di $\mathbb{R}$ vi è sempre un elemento compreso tra i due:
  - I valori rappresentabili di $\mathbb{R}^*$ sono un sottoinsieme che contiene un numero finito di valori reali.




I valori rappresentabili **NON sono equidistanti** nell'intervallo di rappresentazione!

Ogni elemento in $\mathbb{R}^*$ approssima un intervallo di valori del continuo!

# Errore assoluto e relativo (1/2)

- Per valutare l'entità degli errori di approssimazione, usiamo le definizioni di **errore assoluto**, ed **errore relativo**:

$$\varepsilon_a = |\text{ valore vero} - \text{valore approssimato }|$$

$$\varepsilon_r = |\text{ valore vero} - \text{valore approssimato }| \,/\, |\text{valore vero}|$$

- L'**errore relativo** valuta l'errore in termini dipendenti dall'**ordine di grandezza** del valore vero.
  - Un errore assoluto di 1 centimetro nel costruire un ponte di 300 metri ha rilevanza minore di tale errore su un chiodo di 3 centimetri:

$$\varepsilon_{r,1} = |\ 300 - 300.01\ | \,/\, |\ 300\ | = 0.003 = 0.3\ \%$$

$$\varepsilon_{r,2} = |\ 0.03 - 0.04\ | \,/\, |\ 0.03\ | = 0.33 = 33\ \%$$

# Errore assoluto e relativo (2/2)

- L'**errore assoluto** da indicazioni sulle **cifre decimali corrette**, l'**errore relativo** sulle **cifre significative (ovvero, le cifre della mantissa che contribuiscono a determinare il valore).**
  - Se $\varepsilon_a < 10^{-m}$ allora approssimiamo bene il valore ad almeno **m cifre decimali**.
  - Se $\varepsilon_r < 5 \times 10^{-m}$ allora approssimiamo bene il valore ad almeno **(m-1) cifre significative**.

$x_{vero} = 10.3325; \quad x_{approssimato} = 10.335;$

$\varepsilon_a = 0.0025 < 10^{-2}$; $m = 2$ → almeno due cifre decimali corrette (10.33)

$\varepsilon_r = 0.00024 < 5 \times 10^{-4}$; $m = 4$ → almeno tre cifre significative corrette (10.3)

$x_{vero} = 10.3325 \times 10^{10}; \quad x_{approssimato} = 10.335 \times 10^{10};$

$\varepsilon_a = 0.0025 \times 10^{10} < 10^{8}$; $m = -8$ → nessuna cifra decimale corretta!

$\varepsilon_r = 0.00024 < 5 \times 10^{-4}$; $m = 4$ → almeno tre cifre significative corrette (10.3)

# Operazioni in Virgola Mobile (1/3)

- Sebbene potente, la rappresentazione in virgola mobile rende tutte le **operazioni aritmetiche molto complicate** che generano **errori di approssimazione**.
- Per le operazioni di somma e sottrazione è richiesto **l'allineamento degli esponenti:**

$$100\times10^{0} + 100\times10^{-2} =$$
$$100\times10^{0} + \quad 1\times10^{0} = 101\times10^{0}$$

- Per quelle di prodotto e divisione bisogna effettuare separatamente operazioni di somma/sottrazione, prodotto/divisione su **mantissa ed esponente,** rispettivamente**:**

$$100\times10^{0} * 100\times10^{-2} =$$
$$(100 * 100)\times10^{(0-2)} = 10000\times10^{-2}$$
$$= 100\times10^{0}$$

- Inevitabilmente, le trasformazioni che coinvolgono mantissa ed esponente conducono alla **perdita di accuratezza** del numero rappresentante il risultato.

# Operazioni in Virgola Mobile (2/3)

- **L'allineamento degli esponenti** può produrre la **perdita di cifre significative:**
  - E.g., considerando cinque cifre fisse per la mantissa:

$$1{,}9099\times10^{1} + 5{,}9009\times10^{4} =$$

$$0{,}0001\cancel{9099}\times10^{4} + 5{,}9009\times10^{4} = 5{,}9010\times10^{4}$$

  - **L'effetto dell'allineamento della mantissa del moltiplicando ha come effetto il troncamento di alcune cifre significative.**

# Operazioni in Virgola Mobile (3/3)

- Inoltre, le operazioni sui dati possono **amplificare gli errori di approssimazione**, compromettendo i risultati finali dell'elaborazione:
  - Es: Supponiamo $\{1.13, 1.15, 1.17, 1.19\} \subset \mathbb{R}^*$

    $r_1 = 1.14212 \rightarrow r^*_1 = 1.15 \times 10^0$; $\quad \varepsilon_{ro\_r1} = |\, 1.14212 - 1.15 \,| = 0.00788$

    $r_2 = 1.1799 \rightarrow r^*_2 = 1.17 \times 10^0$; $\quad \varepsilon_{ro\_r2} = |\, 1.1799 - 1.17 \,| = 0.0099$
  - Eseguiamo una sottrazione in $\mathbb{R}$ e $\mathbb{R}^*$ e valutiamo l'errore del risultato:

    $r_3 = r_2 - r_1 = 1.1799 - 1.14212 = 0.03778$

    $r^*_3 = r^*_2 - r^*_1 = 1.17 \times 10^0 - 1.15 \times 10^0 = 0.02 \times 10^0$

    $\varepsilon_{ro\_r3} = |\, r_3 - r^*_3 \,| = 0.01778$
  - Ovvero l'errore è cresciuto rispetto agli operandi di:

    $+\ 126\%$ rispetto $\varepsilon_{ro\_r1}$

    $+\ 80\%$ rispetto $\varepsilon_{ro\_r2}$

# Rappresentazione Normalizzata

- La struttura dei numeri ricorrendo alla virgola mobile consente di esprimere con infinite coppie <mantissa, esponente> il medesimo valore, e.g. $48.0 \times 10^3 = 48000.0 \times 10^0 = 4.8 \times 10^4 = \dots$
- Tuttavia, possiamo ricorrere alla notazione con mantissa normalizzata, ovvero la mantissa configurata in modo tale che **preservi il maggior numero di cifre significative.**
  - Es., se abbiamo 3 cifre per la mantissa e 2 per l'esponente, ci conviene scegliere la rappresentazione che non ha zero "in testa"

$0.00\cancel{456} \times 10^3$ [NO] $\rightarrow 4.56\cancel{000} \times 10^0$ [OK]

$0.03\cancel{141} \times 10^2$ [NO] $\rightarrow 3.14\cancel{159} \times 10^0$ [OK]

- L'immediato beneficio è la **riduzione dell'errore di round-off**.
- In generale, la forma normalizzata della mantissa obbliga che **la sua prima cifra sia diversa da zero e che la sua parte intera sia in generale un numero minore dalla base.**

# Standard IEEE 754 (1/2)

- L'**IEEE 754** (Institute of Electrical and Electronics Engineers, IEEE) è uno **standard del 1985 per il calcolo a virgola mobile**, attualmente utilizzato sulla maggior parte dei calcolatori.
  Esso definisce principalmente **tre codifiche**:
  - **Singola precisione**, o precisione semplice **(32 bit)**,
  - **Doppia precisione (64 bit)**,
  - Precisione estesa (80 bit).

| Segno | Esponente (8 bit) | | | Mantissa (23 bit) | | |
|---|---|---|---|---|---|---|
| 31 | 30 | … | 23 | 22 | … | 0 |

| Segno | Esponente (11 bit) | | | Mantissa (52 bit) | | |
|---|---|---|---|---|---|---|
| 63 | 62 | … | 52 | 51 | … | 0 |

# Standard IEEE 754 (1/2)

- La **codifica IEEE 754** impiega:
  - **1 bit per il segno** della mantissa (zero positivo ed uno negativo).
  - **8 o 11 bit per l'esponente**, codificato per **eccesso**, così da non doverne indicare il segno.
  - 23 o 52 bit per la mantissa.
- La **mantissa è normalizzata**, per cui comincia sempre con un bit pari ad 1, seguito dalla “virgola binaria”, e poi dal resto delle cifre.
  - Lo standard esclude dalla codifica il primo bit (**hidden bit**) e la virgola perché sono implicitamente presenti in tutte le parole-codice.

| Argomento | Singola Precisione | Doppia Precisione |
|---|---|---|
| Bit segno | 1 | 1 |
| Bit esponente | 8 | 11 |
| Bit mantissa | 23 | 52 |
| Cifre decimali mantissa | ~7 (23/3.3) | ~15 (52/3.3) |
| Rappresentazione esponente | Base 2 eccesso 127 | Base 2 eccesso 1023 |
| Valori esponente | [-126, 127] | [-1022, 1023] |

# IEEE 754: Configurazioni particolari

- Lo standard IEEE 754 definisce alcune configurazioni straordinarie, agli estremi dell'intervallo dell'esponente, riconoscibili dalle sequenze con tutti 0 o tutti 1:
  - **Esponente al valore minimo (tutti 0):** in questa sola configurazione l'hidden bit è considerato non esserci.
    - **Mantissa uguale a 0**
      - Allora il codice rappresenta lo **zero** (perché non c'è l'hidden bit).
    - **Mantissa diversa da 0**
      - Si considera una rappresentazione **denormalizzata** (senza hidden bit), per ampliare l'intervallo di rappresentazione!
  - **Esponente al valore massimo (tutti 1):**
    - **Mantissa uguale a 0**
      - Il codice rappresenta **infinito**: in particolare, grazie al bit di segno, possiamo rappresentare $+\infty$ e $-\infty$
    - **Mantissa diversa da 0**
      - rappresenta un simbolo chiamato "**Not a Number**" (**NaN**), che indica un valore **indefinito**. Esso è impiegato all'occorrenza di operazioni di calcolo non definite**, come la divisione per 0** o **la radice quadrata di un numero negativo**.

# IEEE 754: overview

| Parametro | Precisione | |
|---|---|---|
| | Singola | Doppia |
| Bit mantissa (Codifica segno e modulo) | 23 + bit segno (+ hidden bit) | 52 + bit segno (+hidden bit) |
| Bit esponente (Codifica per eccesso) | 8 | 11 |
| Bias esponente | +127 | +1023 |
| Max numero rappresentabile | 3.4028E+38 | 1.7976E+308 |
| Min numero rappresentabile div. zero | 1.1754E-38 (1.4012E-45 denormalizzato) | 2.2250E-308 (4.9406E-324 denormalizzato) |
| Rappresenta senza alcun errore almeno numeri fino* | 6 cifre significative | 15 cifre significative |

*Ovviamente i calcoli possono comunque generare ed amplificare l'errore di round-off!

# IEEE 754: Esempi di Codifica (1/3)

- Codificare in IEEE 754 il numero 1.75:
  - Codifichiamo in binario la parte intera e frazionaria della mantissa:
    - 1.75 → 1.11 (infatti, $1\times 2^0 + 1\times 2^{-1} + 1\times 2^{-2}$)
    - Normalizziamo la mantissa: nessuna operazione (solo rimozione dell'hidden bit);
    - Codifichiamo l'esponente: 0 in eccesso → 0+127 = 011111111
    - Codifica del segno: 0
    - Risultato: 0 01111111 11000000000000000000000
    - Errore di round-off = 0

# IEEE 754: Esempi di Codifica (2/3)

- Codificare in IEEE 754 il numero -1999.665:
  - Codifichiamo in binario la parte intera e frazionaria della mantissa:
    - 1999.665 → 1999 = 11111001111
      → 0.665 si adotta l'algoritmo della divisione per la parte decimale

- Partendo da 0.665, moltiplico per 2 la parte decimale. Se la parte decimale è non nulla, moltiplico ancora la parte decimale per 2 e proseguo.
- Prendo come risultato la parte intera di ogni numero nell'ordine in cui compare
- Nel nostro caso, l'algoritmo di conversione non trova parte decimale 0 nelle prime 16 iterazioni
- Risultato: 1010101000111101

| 0,665 | |
|---|---|
| 1,330 | 1 |
| 0,66 | 0 |
| 1,32 | 1 |
| 0,64 | 0 |
| 1,280 | 1 |
| 0,560 | 0 |
| 1,120 | 1 |
| 0,240 | 0 |
| 0,498 | 0 |
| 0,960 | 0 |
| 1,920 | 1 |
| 1,840 | 1 |
| 1,680 | 1 |
| … | … |

# IEEE 754: Esempi di Codifica (3/3)

- Codificare in IEEE 754 il numero -1999.665:
  - Codifichiamo in binario la parte intera e frazionaria della mantissa:
    - 1999.665 → 11111001111.1010101000111101…
    - Normalizziamo la mantissa: spostiamo la virgola verso sinistra di 10 posizioni, in modo da avere 1., bisogna poi togliere il bit 1 (hidden bit)
      - Vi è errore di round-off poiché possiamo conservare solo 23 bit dopo la virgola (1.11110011111010101001000~~111101~~)
      - **Per questo motivo, è inutile dare a 665 molti bit per la sua codifica perché vengono persi durante la normalizzazione!**
    - Codifichiamo l'esponente: 10 in eccesso → 10+127 = 10001001
    - Codifica del segno: 1 (numero negativo)
    - Risultato: 1 10001001 11110011111010101001000
    - Errore di round-off = $|-1999.665 + 1999.6650390625| = 0{,}0000390625 = 3{,}90625 \times 10^{-5}$
      - Equivale ad un errore sulla 5 cifra decimale

# IEEE 754: Esempi di Calcolo della Somma

- Effettuare la somma tra A=9.37 e B=-4.39:
  - Codificare entrambi i numeri:
    - A= 9.37 = 0 10000010 00101011110101110000101
    - B= -4.39 = 1 10000001 00011000111101011100001
  - Allineare l'esponente più piccolo (B) in modo tale da raggiungere lo stesso valore dell'altro (più grande, A):
    - A ha un'esponente pari a 3 (che in eccesso equivale a 130), mentre B ha un'esponente pari a 2 (che equivale a 129);
    - Dunque bisogna dividere per 2 (shift a destra) la mantissa di B e portare l'esponente a 130:
      - B = 1 10000010 (1)0001100011110101110000~~1~~
      - N.B.: nello shift a destra compare l'**hidden bit**! (riportato in rosso)
      - In più c'è un errore di round-off dovuto alla perdita dell'1 finale, $2^{-24}$
      - Questa rappresentazione di B è **denormalizzata**, ma ci serve solo per effettuare il calcolo dell'addizione: una volta ottenuto il risultato, procederemo alla normalizzazione.

# IEEE 754: Esempi di Calcolo della Somma

– Effettuare la sottrazione delle mantisse, considerando il segno:

Mantissa di A 1 00101011110101110000101 +

Mantissa di B 0 10001100011110101110000 =

0 10011111010111000010101

– Normalizzare la mantissa risultato: 1.00111110101110000101010

  • N.B.: quando effettuiamo la normalizzazione, compare uno 0 alla fine (riportato in giallo)

– Calcolo dell'esponente: abbiamo effettuato uno shift a sinistra per normalizzare la mantissa, cioè abbiamo moltiplicato il numero per 2, per cui dobbiamo sottrarre 1 all'esponente: 10000010 – 1 = 10000001

– Risultato: 0 10000001 00111110101110000101010

# IEEE 754: Esempi di Calcolo della Somma

- Round off: $\varepsilon_a(A) = |9.37-9.369999885559082| =$

  $= 0.000000114440918 = 1.14 \times 10^{-7}$

  $\varepsilon_a(B) = |-4.39+4.389999866485596| =$

  $= 0,000000133514404 = 1.34 \times 10^{-7}$

  $\varepsilon_a(A+B) = |4.98- 4.980000495910644 5| =$

  $= 0,000000495910644 = 4.96 \times 10^{-7}$

- L'errore sul risultato è circa 4 volte più grande!

# Rappresentazione dei Caratteri

- I **caratteri** sono **informazioni per loro natura discrete**, pertanto la rappresentazione di tali informazioni si riduce ad **adottare una codifica condivisa.**
- Esistono numerose codifiche per i caratteri. Le più importanti e diffuse sono:
  - La codifica ASCII (7 bit)
  - Le codifiche sotto il nome di ASCII Esteso (8 bit)
  - Le codifiche UTF-8, UTF-16, UTF-32, basati sullo standard UNICODE

# Codice ANSI ASCII (1/2)

- **L'ASCII** (**American Standard Code for Information Interchange***)* **nasce alla fine degli anni sessanta** dall'ente americano di standardizzazione ANSI (American National Standards Institute) che fissò una codifica **per consentire anche a calcolatori (all'epoca più per telescriventi) di produttori diversi di poter comunicare tra loro**.
- ASCII codifica caratteri alfabetici, numerici, di punteggiatura, simboli, e anche alcuni codici da usare come controllo della comunicazione tra una macchina e l'altra (per esempio, per segnalare l'inizio o la fine di una trasmissione).
- Essendo concepito per l'USA e per macchine con risorse limitate, ASCII è di 7 bit e riesce a codificare con le 128 parole-codice solamente le lettere dell'alfabeto inglese, escludendo, ad esempio, i caratteri accentati in uso nelle lingue europee (è, é, à, ì, ò, …).

# Codice ANSI ASCII (2/2)

**USASCII code chart**

| $b_4$ | $b_3$ | $b_2$ | $b_1$ | $b_7$ $b_6$ $b_5$ → | 0 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1 0 0 | 1 0 1 | 1 1 0 | 1 1 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Row ↓ / Column → | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 0 | 0 | 0 | 0 | NUL | DLE | SP | 0 | @ | P | ` | p |
| 0 | 0 | 0 | 1 | 1 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 0 | 0 | 1 | 0 | 2 | STX | DC2 | " | 2 | B | R | b | r |
| 0 | 0 | 1 | 1 | 3 | ETX | DC3 | # | 3 | C | S | c | s |
| 0 | 1 | 0 | 0 | 4 | EOT | DC4 | $ | 4 | D | T | d | t |
| 0 | 1 | 0 | 1 | 5 | ENQ | NAK | % | 5 | E | U | e | u |
| 0 | 1 | 1 | 0 | 6 | ACK | SYN | & | 6 | F | V | f | v |
| 0 | 1 | 1 | 1 | 7 | BEL | ETB | ' | 7 | G | W | g | w |
| 1 | 0 | 0 | 0 | 8 | BS | CAN | ( | 8 | H | X | h | x |
| 1 | 0 | 0 | 1 | 9 | HT | EM | ) | 9 | I | Y | i | y |
| 1 | 0 | 1 | 0 | 10 | LF | SUB | * | : | J | Z | j | z |
| 1 | 0 | 1 | 1 | 11 | VT | ESC | + | ; | K | [ | k | { |
| 1 | 1 | 0 | 0 | 12 | FF | FS | , | < | L | \ | l | \| |
| 1 | 1 | 0 | 1 | 13 | CR | GS | - | = | M | ] | m | } |
| 1 | 1 | 1 | 0 | 14 | SO | RS | . | > | N | ^ | n | ~ |
| 1 | 1 | 1 | 1 | 15 | SI | US | / | ? | O | _ | o | DEL |

# Le estensioni di ASCII: ASCII Esteso

- Con **ASCII Esteso** si intendono quelle codifiche con parole codice di 8 bit che introducono ulteriori caratteri ad ASCII, adottando codifiche compatibili con esso per i primi 7 bit.
  - Es.: ISO 8859-1 (ISO Latin 1) è una codifica ASCII Estesa per le lingue dell'Europa occidentale.

**Codepage 819 - Latin 1 - ISO 8859-1**

| | -0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -A | -B | -C | -D | -E | -F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8- | 0080 | 0081 | 0082 | 0083 | 0084 | 0085 | 0086 | 0087 | 0088 | 0089 | 008A | 008B | 008C | 008D | 008E | 008F |
| 9- | 0090 | 0091 | 0092 | 0093 | 0094 | 0095 | 0096 | 0097 | 0098 | 0099 | 009A | 009B | 009C | 009D | 009E | 009F |
| A- | 00A0 | ¡ 00A1 | ¢ 00A2 | £ 00A3 | ¤ 00A4 | ¥ 00A5 | ¦ 00A6 | § 00A7 | ¨ 00A8 | © 00A9 | ª 00AA | « 00AB | ¬ 00AC | - 00AD | ® 00AE | ¯ 00AF |
| B- | ° 00B0 | ± 00B1 | ² 00B2 | ³ 00B3 | ´ 00B4 | µ 00B5 | ¶ 00B6 | · 00B7 | ¸ 00B8 | ¹ 00B9 | º 00BA | » 00BB | ¼ 00BC | ½ 00BD | ¾ 00BE | ¿ 00BF |
| C- | À 00C0 | Á 00C1 | Â 00C2 | Ã 00C3 | Ä 00C4 | Å 00C5 | Æ 00C6 | Ç 00C7 | È 00C8 | É 00C9 | Ê 00CA | Ë 00CB | Ì 00CC | Í 00CD | Î 00CE | Ï 00CF |
| D- | Ð 00D0 | Ñ 00D1 | Ò 00D2 | Ó 00D3 | Ô 00D4 | Õ 00D5 | Ö 00D6 | × 00D7 | Ø 00D8 | Ù 00D9 | Ú 00DA | Û 00DB | Ü 00DC | Ý 00DD | Þ 00DE | ß 00DF |
| E- | à 00E0 | á 00E1 | â 00E2 | ã 00E3 | ä 00E4 | å 00E5 | æ 00E6 | ç 00E7 | è 00E8 | é 00E9 | ê 00EA | ë 00EB | ì 00EC | í 00ED | î 00EE | ï 00EF |
| F- | ð 00F0 | ñ 00F1 | ò 00F2 | ó 00F3 | ô 00F4 | õ 00F5 | ö 00F6 | ÷ 00F7 | ø 00F8 | ù 00F9 | ú 00FA | û 00FB | ü 00FC | ý 00FD | þ 00FE | ÿ 00FF |

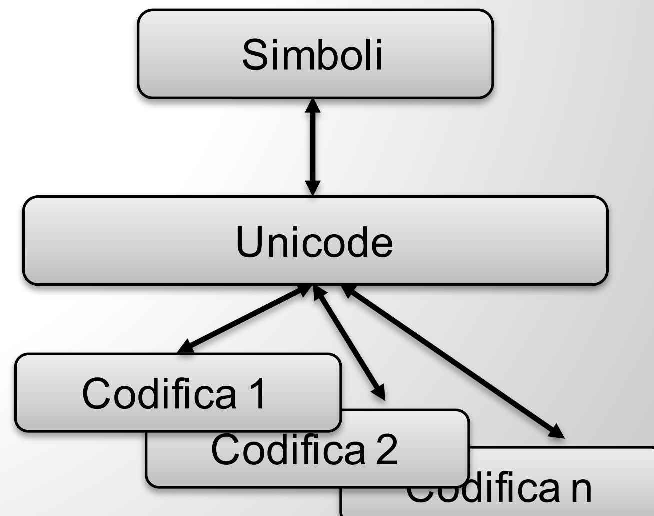
# Unicode (1/2)

- **Unicode (Universal Encoding)** è uno standard che si propone di creare una **codifica delle scritture a livello universale**.
  - Cataloga tutti i **simboli usati nei testi** ed assegna a loro **un numero**, in maniera indipendente dalla rappresentazione, alfabeto e lingua.
  - È **compatibile con ASCII** e supporta **tutte le lingue del mondo**, codificando oltre **un milione di simboli**.
- Sullo standard dei numeri assegnati ai simboli Unicode sono poi definite le codifiche concrete, tra cui le **UTF** (**Unicode Transformation Format**).

**Faces**

| | | |
|---|---|---|
| 1F600 | 😀 | GRINNING FACE |
| 1F601 | 😁 | GRINNING FACE WITH SMILING EYES |
| 1F602 | 😂 | FACE WITH TEARS OF JOY |
| 1F603 | 😃 | SMILING FACE WITH OPEN MOUTH<br>→ 263A ☺ white smiling face |
| 1F604 | 😄 | SMILING FACE WITH OPEN MOUTH AND SMILING EYES |
| 1F605 | 😅 | SMILING FACE WITH OPEN MOUTH AND COLD SWEAT |

Fonte: http://unicode.org/charts/PDF/U1F600.pdf



# Unicode (2/2)

- Creare nuove codifiche basandosi su Unicode permette di **identificare univocamente i simboli a prescindere dalla rappresentazione**, permettendo il dialogo tra tutti i sistemi basati sullo standard.
  - I numeri assegnati da Unicode ai simboli assumono la funzione di una “lingua franca” per tutte le codifiche basate su di esso.
- UTF-32 è un codifica basata Unicode **a lunghezza fissa di 32 bit**.
  - Poiché **la maggior parte dei simboli Unicode sono usati raramente**, è spesso più conveniente adottare una **codifica a lunghezza variabile**.
- UTF-8 e UTF-16 sono codifiche basate su Unicode **a lunghezza variabile**, aventi lunghezza minima di 8 e 16 bit.
  - UTF-16 è la codifica predefinita utilizzata dai moderni sistemi operativi, tra cui Windows e Mac OS X.