

Rappresentazione dei numeri naturali ed interi



Prof.ssa Lucia Migliorelli

lmigliorelli@unite.it

Corso di Sistemi multimediali e web per il turismo

Dipartimento di Scienze Politiche, Università di Teramo

Agenda

- Numeri naturali
- Numeri interi

Numeri Naturali

Numeri naturali in varie basi

- Quando scriviamo un numero, di solito usiamo il **sistema decimale** (base 10). È quello che impariamo a scuola: le cifre vanno da 0 a 9 e il valore del numero dipende dalla posizione delle cifre (unità, decine, centinaia, ecc.).
- Ma i computer non ragionano in base 10: utilizzano la **base 2 (binario)**, cioè si servono solo delle cifre 0 e 1. Ogni cifra binaria si chiama **bit**.
- Oltre al binario e al decimale, in informatica è molto comune anche la **base 16 (esadecimale)**, perché permette di rappresentare numeri binari molto lunghi in modo più compatto e leggibile. In esadecimale le cifre vanno da 0 a 9, poi si usano le lettere A, B, C, D, E, F per rappresentare i valori da 10 a 15.

Numeri naturali in varie basi

250 in base 10, in base 2 e in base 16 (A=10, B=11, C=12, ... F=15):

Esponente	2	1	0
	2	5	0
Base	10	10	10

$$2 \times 10^2 + 5 \times 10^1 + 0 \times 10^0 = 250$$

Esponente	7	6	5	4	3	2	1	0
	1	1	1	1	1	0	1	0
Base	2	2	2	2	2	2	2	2

$$1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 250$$

Esempio – metodo pratico

Conversione in base due del numero 13 in base dieci

$$13 : 2 = 6 \text{ resto } 1$$

$$6 : 2 = 3 \text{ resto } 0$$

$$3 : 2 = 1 \text{ resto } 1$$

$$1 : 2 = 0 \text{ resto } 1 \rightarrow \textbf{STOP} \text{ (Poiché il quoziente è 0)}$$

La conversione binaria di 13 possiamo ottenerla scrivendo da sinistra a destra i resti presi dal basso verso l'alto

$$(13)_{10} = (1101)_2$$

Esempio – metodo pratico (1)

Come prova convertiamo 1101 da base due a base dieci. La posizione di ogni bit da destra verso sinistra ci informa dell'esponente da applicare alla base 2, a partire da zero.

$$\begin{aligned}(1101)_2 &= (1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)_{10} = \\ &= (8 + 4 + 0 + 1)_{10} = (13)_{10}\end{aligned}$$

$$\mathbf{(1101)_2 = (13)_{10}}$$

Esempio – metodo pratico (2)

Torniamo al numero 250...

Esponente	7	6	5	4	3	2	1	0
	1	1	1	1	1	0	1	0
Base	2	2	2	2	2	2	2	2

$$1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 250$$

- $250 \div 2 = 125$, resto **0**
- $125 \div 2 = 62$ resto **1**
- $62 \div 2 = 31$, resto **0**
- $31 \div 2 = 15$, resto **1**
- $15 \div 2 = 7$, resto **1**
- $7 \div 2 = 3$, resto **1**
- $3 \div 2 = 1$, resto **1**
- $1 \div 2 = 0$, resto **1**

11111010

Esempio – metodo pratico (3)

Torniamo al numero 250...

Esponente	1	0		
	<table border="1"><tr><td>F</td><td>A</td></tr></table>	F	A	
F	A			
Base	16	16		

$$15 \times 16^1 + 10 \times 16^0$$

- $250 \div 16 = 15$, resto **10**
- $15 \div 16 = 0$ resto **15**

$$16 \times 15 = 240, \\ 250 - 240 = 10$$

$$16 \times 0 = 0, \\ 15 - 0 = 15$$

Esempio – metodo pratico (4)

Esponente 1 0
 F A
Base 16 16 $15 \times 16^1 + 10 \times 16^0$

- $250 \div 16 = 15$, resto **10**
 A
 16x15=240,
 250-240=10
- $15 \div 16 = 0$ resto **15**
 F
 16x0=0,
 15-0=15

DECIMALE	Esadecimale	BINARIO
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Aritmetica modulare – modulo 2^N

I numeri naturali sono infiniti, ma la memoria di un elaboratore digitale è finita!

Se usiamo N bit, possiamo rappresentare solo un numero finito di valori $\{0, \dots, 2^N - 1\}$

- $N=3$ bit $\rightarrow$ posso rappresentare $2^3=8$ valori: da 0 a 7
- $N=4$ bit $\rightarrow$ posso rappresentare $2^4=16$ valori: da 0 a 15

Posto che si riservino N bit per rappresentare i numeri naturali in un elaboratore digitale, possiamo rappresentare qualunque numero naturale M con la convenzione «modulo 2^N »

A cosa serve?

Quando un numero è troppo grande, il computer conserva solo il **resto** della divisione per 2^N

Ovvero, due numeri che differiscono di un multiplo di 2^N hanno la stessa rappresentazione

Aritmetica modulare – modulo 2^N

Quando un numero è troppo grande, il computer conserva solo il **resto** della divisione per 2^N

Ovvero, due numeri che differiscono di un multiplo di 2^N hanno la stessa rappresentazione

Numero decimale	Divisione per 8	Resto	Rappresentazione in 3 bit
5	$5 \div 8 = 0$ (resto 5)	5	101
13	$13 \div 8 = 1$ (resto 5)	5	101

Quando diciamo
 $a \div b = q$ con resto r
intendiamo che:
 $a = q \cdot b + r$



Esempio Modulo 2^N (1)

Supponiamo di lavorare con $N = 3$, cioè con 3 bit. In questo caso $2^N = 2^3 = 8$

Dec	Bin	Dec	Bin	Dec	Bin
0	000	8	000	16	000
1	001	9	001	17	001
2	010	10	010	18	010
3	011	11	011	19	011
4	100	12	100	20	100
5	101	13	101	21	101
6	110	14	110	22	110
7	111	15	111	23	111

Nella tabella i numeri da 8 a 23 non sono rappresentati con 3 bit.

Quello che si vede è solo una troncatura della loro rappresentazione binaria completa.

Esempio: 8 in binario è 1000 (serve almeno 4 bit!)

Se si taglia il primo bit e si tengono solo gli ultimi 3 → si ottiene 000

Esempio Modulo 2^N (2)

Supponiamo di lavorare con $N = 3$, cioè con 3 bit. In questo caso $2^N = 2^3 = 8$

Se uso 3 bit, il computer riesce a contare fino a 7, poi riparte da 0

Dec	Bin	Dec	Bin	Dec	Bin
0	000	8	000	16	000
1	001	9	001	17	001
2	010	10	010	18	010
3	011	11	011	19	011
4	100	12	100	20	100
5	101	13	101	21	101
6	110	14	110	22	110
7	111	15	111	23	111

Nella tabella i numeri da 8 a 23 non sono rappresentati con 3 bit.

Quello che si vede è solo una troncatura della loro rappresentazione binaria completa.

Esempio: 8 in binario è 1000 (serve almeno 4 bit!)

Se si taglia il primo bit e si tengono solo gli ultimi 3 → si ottiene 000

Addizione numeri naturali (1)

Nei numeri naturali l'addizione e la sottrazione sono operazioni sempre definite.

- Se faccio $25+10=35$ ottengo un nuovo numero naturale.
- Se faccio $35-10=25$ ottengo ancora un numero naturale.

Non ci sono limiti: i naturali sono infiniti

Un computer usa i bit per rappresentare i numeri.

- Con 8 bit si possono rappresentare solo i numeri naturali da 0 a 255 ($2^8 - 1$)
- Quindi non tutti i naturali sono rappresentabili: c'è un vincolo di memoria.

Addizione numeri naturali (2)

Addizione con 8 bit (overflow)

- Se il risultato resta tra 0 e 255 → nessun problema.
- Se il risultato supera 255 → il numero non «sarebbe» più rappresentabile con 8 bit. In questo caso il computer lavora in modulo 2^8 : il risultato “riparte da 0”.
- $100+50=150$ → rappresentabile.
- $200+100=300$ → (???)

Addizione numeri naturali (3)

Addizione con 8 bit (overflow)

- Se il risultato resta tra 0 e 255 → nessun problema.
- Se il risultato supera 255 → il numero non «sarebbe» più rappresentabile con 8 bit. In questo caso il computer lavora in modulo 2^8 : il risultato “riparte da 0”.
 - $100+50=150$ → rappresentabile.
 - $200+100=300 \rightarrow 300 \div 256 = 1$ (resto 44) → il pc memorizza 44



Addizione numeri naturali (4)

$$5+3=8$$

$$\begin{array}{r} 0101 \quad (5) \\ + 0011 \quad (3) \\ \hline 1000 \quad (8) \end{array}$$

$$\mathbf{13+7?}$$

Addizione numeri naturali (5)

$$5+3=8$$

$$\begin{array}{r} 0101 \quad (5) \\ + 0011 \quad (3) \\ \hline 1000 \quad (8) \end{array}$$

$$\mathbf{13+7?}$$

$$\begin{array}{r} 1101 \\ + 0111 \\ \hline 1\ 0100 \quad (5 \text{ bit!}) \end{array}$$

Addizione numeri naturali (6)

13+7?

1	1	0	1
0	1	1	1

+

1	0	1	0	0
---	---	---	---	---



Posizione (bit)	Operazione	Somma decimale	Scrivo nel risultato	Riporto al bit successivo
bit 0 (destra)	1 + 1	2	0	1
bit 1	0 + 1 + 1 (riporto)	2	0	1
bit 2	1 + 1 + 1 (riporto)	3	1	1
bit 3 (sinistra)	1 + 0 + 1 (riporto)	2	0	1 (nuovo bit extra!)

overflow

Addizione numeri naturali (7)

0000	=	0
0001	=	1
0010	=	2
0011	=	3
0100	=	4
0101	=	5
0110	=	6
0111	=	7
1000	=	8
1001	=	9
1010	=	10
1011	=	11
1100	=	12
1101	=	13
1110	=	14
1111	=	15

16
17
18
19
20



13+7?

1	1	0	1
0	1	1	1

+

★	0	1	0	0
---	---	---	---	---

Per poter rappresentare il 20 dovrei avere un bit in più (5).
Su 4 bit «tronco» il bit di overflow ottenendo $(0100)_2 = (4)_{10}$

Aritmetica modulare

Osservazioni

Questi 8 bit: 01000001 non dicono *da soli* se rappresentano un numero, una lettera, o un colore. Sono solo una sequenza di 0 e 1.

Il significato dipende dal *programma* che li legge.

Interpretazione	Significato
Intero senza segno (decimale)	65
Carattere ASCII	A
Colore (canale rosso)	intensità $65/255 \approx 25\%$ rosso

```
x = 0b01000001    ### il prefisso 0b serve solo per dire che sto scrivendo un numero in base 2
print(int(x))
print(chr(x))
```

```
⇒ 65
  A
```

Numeri interi

Numeri interi in un elaboratore digitale – modulo e segno

Come rappresentare i numeri interi {..., -2, -1, 0, 1, 2, ...} in un elaboratore digitale?

La soluzione più semplice sarebbe quella di codificarli esattamente come i numeri naturali, riservando **il bit più a sinistra per rappresentare il segno** (0 = positivo, 1 = negativo).

Questa codifica è detta modulo e segno e consiste nell'usare **un bit per il segno**, lasciando **gli altri bit a rappresentare il valore assoluto del numero intero...**

Modulo e segno

Tale codifica però include una **doppia rappresentazione dello 0** che può essere fonte di **ambiguità**.

Per questo, la codifica in modulo e segno, che sarebbe esprimibile come

$$\text{codifica}(x) = \begin{cases} x & \text{se } x \geq 0 \\ |x| + 2^{N-1} & \text{se } x \leq 0 \end{cases}$$

non viene usata. La tabella a lato mostra la codifica modulo e segno con 4 bit ($N = 4$)

Decimale	Mod e segno
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111
-8	-

Numeri interi in un elaboratore digitale - complemento a 1

Come rappresentare i numeri interi {..., -2, -1, 0, 1, 2, ...} in un elaboratore digitale?

Un'altra soluzione sarebbe codificare gli **interi positivi come i numeri naturali** e i numeri negativi come il **complemento a 1** (cioè l'inversione di tutti i bit) dei positivi.

Es: 4 potrebbe essere codificato come 0100, mentre -4 come 1011

Complemento a 1

Anche tale codifica però include una **doppia rappresentazione dello 0** che può essere fonte di **ambiguità**.

Nemmeno la codifica basata sul complemento è adatta per rappresentare i numeri interi in un elaboratore digitale

Decimale	Compl. a 1
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-0	1111
-1	1110
-2	1101
-3	1100
-4	1011
-5	1010
-6	1001
-7	1000
-8	-

Numeri interi in un elaboratore digitale – notazione traslazionale

Come rappresentare i numeri interi {..., -2, -1, 0, 1, 2, ...} in un elaboratore digitale?

Una soluzione al problema delle codifiche in modulo e segno e in complemento a 1 è traslare verso l'alto tutti i valori che possono essere inclusi su N bit.

$$\text{codifica}(x) = \begin{cases} 2^{N-1} + |x| & \text{se } x \geq 0 \\ 2^{N-1} - |x| & \text{se } x \leq 0 \end{cases}$$

In questo modo la codifica non è più ambigua: sia 0 che -0 sono rappresentati come 2^{N-1}

Se si sta centrando l'intervallo degli interi intorno a 2^{N-1} .
Per esempio, con **N = 4 bit** $\rightarrow 2^{N-1} = 8$:

Lo zero viene codificato come 8.

- I positivi (1, 2, 3, ...) stanno sopra 8: 9,10,11
- I negativi (-1, -2, -3, ...) stanno sotto 8: 7,6,5

Risolvo l'ambiguità dello 0

Notazione traslazionale

Se ho a disposizione 4 bit per la rappresentazione devo decidere quali numeri interi associare a queste ($2^4=16$) combinazioni.

Nella codifica in traslazione, si sceglie di far partire la numerazione dal numero più negativo possibile.

16 valori=da -8 (numero più piccolo rappresentabile) fino a +7 (numero più grande rappresentabile)

-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7

0000 → 0
0001 → 1
0010 → 2
0011 → 3
...
1111 → 15

Ad ogni passo aggiungo 1

0000 → -8
0001 → -7
0010 → -6
0011 → -5
...
1111 → 7

Rappresentazione in binario dei numeri naturali
(lo 0 è il minor numero rappresentabile)

Rappresentazione in binario dei numeri interi in
notazione traslazionale (in questo caso il -8 è il
minor numero rappresentabile)

Notazione traslazionale

Un metodo «intuitivo» è partire a contare dal minimo numero rappresentabile (con $N = 4$ bit tale numero è -8) e ad ogni incremento si aggiunge 1 (-8 = 0000, -7 = 0001, -6 = 0010, ...).

Tuttavia, la somma algebrica di due numeri traslati **non è** la traslazione della somma

$$\begin{array}{r} 2 + \\ -3 = \\ \hline -1 \end{array}$$

$$\begin{array}{r} 1010 + \\ 0101 = \\ \hline 1111 \end{array}$$



0000 → -8
0001 → -7
0010 → -6
0011 → -5

...

0111 → -1
1000 → 0
1001 → 1
1010 → 2

...

1111 → 7

1111 non è la codifica in traslazione di -1! Nemmeno questa codifica si usa per gli interi.

Numeri interi in un elaboratore digitale – complemento a 2

Come rappresentare i numeri interi {..., -2, -1, 0, 1, 2, ...} in un elaboratore digitale?

I numeri interi in un elaboratore digitale sono universalmente rappresentati con la codifica in «complemento a 2», definita come segue

$$\text{codifica}(x) = \begin{cases} x & \text{se } x \geq 0 \\ 2^N - |x| & \text{se } x \leq 0 \end{cases}$$

In altre parole, su N bit, gli interi positivi sono rappresentati esattamente come i naturali, gli interi negativi sono rappresentati come $2^N - |x|$

Complemento a 2 (1)

Si noti che su 4 bit ($N = 4$) nel caso di «-0» la rappresentazione sarebbe $2^N - |x|$ e cioè $(16)_{10} = (10000)_2$.

Ma con 4 bit non posso scrivere 5 cifre, quindi -0 non esiste: rimane solo 0000

Ecco perché in complemento a 2 lo zero è unico.

Inoltre **la somma algebrica di due numeri in complemento a 2 restituisce il risultato già correttamente codificato!**

Decimale	Compl. a 2
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Codifica
binaria dei
naturali

0000 = 0
0001 = 1
0010 = 2
0011 = 3
0100 = 4
0101 = 5
0110 = 6
0111 = 7
1000 = 8
1001 = 9
1010 = 10
1011 = 11
1100 = 12
1101 = 13
1110 = 14
1111 = 15

Complemento a 2 (2)

Esempio

7+3=10 (10 non è rappresentabile con 4 bit, → con 4 bit posso rappresentare i numeri da -8 a 7)

Calcolo in binario

- 7 (0111)
- 3 (0011)
- 7+3 → 0111+0011=

```
  0111
+ 0011
-----
1010
```

Decimale	Compl. a 2
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Complemento a 2 (3)

Esempio

7+3=10 (10 non è rappresentabile con 4 bit, → con 4 bit posso rappresentare i numeri da -8 a 7)

Calcolo in binario

- 7 (0111)
- 3 (0011)
- 7+3 → 0111+0011=

0111
+ 0011

1010

Questo è l'**overflow**: il risultato "sfora" l'intervallo e diventa un numero negativo.

Decimale	Compl. a 2
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Complemento a 2 (4)

Perché in complemento a 2 lo sforo (overflow) trasforma +10 in -6?

Con **4 bit** possiamo rappresentare solo i numeri da -8 a +7

Quindi il numero +10 **non entra** in questo intervallo. Ma il circuito HW fa la somma comunque, cioè calcola:

$7+3=10$ (mod16, perché con 4 bit lavoro in **modulo $2^4=16$**).

Ora 10 in binario a 4 bit è

- $10:2=5$ resto 0
- $5:2=2$ resto 1
- $2:2=1$ resto 0
- $1:2=0$ resto 1 $\rightarrow (10)_{10}=(1010)_2 \rightarrow$ Questo non viene letto come +10 perché??

Complemento a 2 (5)

Perché in complemento a 2 lo sforno (overflow) trasforma +10 in -6?

Con **4 bit** possiamo rappresentare solo i numeri da -8 a +7

Quindi il numero +10 **non entra** in questo intervallo. Ma il circuito HW fa la somma comunque, cioè calcola:

$7+3=10 \pmod{16}$, perché con 4 bit lavoro in **modulo $2^4=16$** .

Ora 10 in binario a 4 bit è

- $10:2=5$ resto 0
- $5:2=2$ resto 1
- $2:2=1$ resto 0
- $1:2=0$ resto 1 $\rightarrow (10)_{10}=(1010)_2 \rightarrow$ Questo non viene letto come +10 perché?? L'estremo massimo rappresentabile è +7. **Come interpreto (1010 in complemento a 2)?**

Complemento a 2 (6)

Perché in complemento a 2 lo sforo (overflow) trasforma +10 in -6?

$(10)_{10} = (1010)_2 \rightarrow$ Questo non viene letto come +10 perché?? L'estremo massimo rappresentabile è +7.

Come interpreto (1010 in complemento a 2)?

- Se il bit più a sinistra (MOST SIGNIFICANT BIT - MSB) è 0 allora il numero è positivo
- Se il bit più a sinistra è 1 allora il numero è negativo

Nel nostro caso il numero è negativo (1010)

Come calcolo il valore di 1010 in complemento a 2?

Metodo 1) $1010 = (-1) \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = -8 + 0 + 2 + 0 = -6$

Metodo 2) si parte sapendo che il numero 1010 è negativo dato il MSB

- Inverto i bit 1010 $\rightarrow$ 0101 (è il complemento a 1 che aveva un problema di rappresentazione dello 0)
- Aggiungo 1 $\rightarrow 0101 + 1 = (0110)_2 = (6)_{10}$
- Il MSB (bit più a sinistra) era 1 $\rightarrow$ quindi il numero era negativo $\Rightarrow -6$

Complemento a 2 (7)

Perché in complemento a 2 lo sforno (overflow) trasforma +10 in -6?

Qual è la connessione tra +10 e -6?

$$10 - 16 = -6$$

Cioè:

$$10 \equiv -6 \pmod{16}$$

Complemento a 2 (8)

Ci sarebbe un metodo 3 per il complemento a 2 di un intero negativo

- Data la rappresentazione binaria di un numero positivo
- Esamino il numero **bit per bit da destra verso sinistra**
- Per scrivere il complemento a 2 del numero negativo lascio ogni bit invariato **fino al primo 1 compreso**
- **Inverto tutti i bit che seguono il primo 1 da destra verso sinistra.**

Numero decimale	Binario positivo	Metodo classico (inverti +1)	Metodo intuitivo (copia fino a 1, poi inverti)	Risultato finale
-1	(1 ₁₀) 0001	Inverti → 1110, +1 → 1111	Copio 1 → 1, resto invertito → 111	1111
-2	(2 ₁₀) 0010	Inverti → 1101, +1 → 1110	Copio 10 → 10, resto invertito → 11	1110
-8	(8 ₁₀) 1000 (*)	Inverti → 0111, +1 → 1000	Caso limite: direttamente 1000	1000

Complemento a 2 – esempio (1)

Esempio: -2 rappresentato su 4 bit ($N = 4$)

L'intero positivo $(2)_{10}$ in base due vale $(0010)_2$

Applicando da destra verso sinistra quanto spiegato nella slide precedente per il «complemento a 2» in un elaboratore digitale si ha che $(-2)_{10} = (1110)_2$. Applicando a 0010 passaggio per passaggio quanto descritto:

–	–	–	0
–	–	1	0
–	1	1	0
1	1	1	0

Complemento a 2 – esempio (2)

Esempio: -5 rappresentato su 4 bit ($N = 4$)

L'intero positivo $(5)_{10}$ in base due vale $(0101)_2$

Applicando il «complemento a 2» si ha che $(-5)_{10} = (1011)_2$. Applicando a 0101 passaggio per passaggio quanto descritto:

–	–	–	1
–	–	1	1
–	0	1	1
1	0	1	1

Rappresentazioni a lunghezza fissa: problemi (un breve recap) (1)

Quando un numero viene rappresentato con un numero limitato di bit, non tutti i valori sono possibili. Questo può creare due tipi di errori:

Overflow

Succede quando il risultato di un'operazione è **troppo grande** per essere contenuto nei bit disponibili.

Esempio: con 8 bit il massimo rappresentabile è 255.

Se faccio $130+150=280$, 280 non sta nell'intervallo, è troppo grande!

Underflow

Succede quando il risultato è **troppo piccolo** per essere rappresentato.

Esempio: con i soli numeri interi, $25/50=0,5$ non può essere scritto (perché 0,5 non è un intero).

L'unico risultato possibile è 0.

Rappresentazioni a lunghezza fissa: problemi (un breve recap) (2)

Con 8 bit posso rappresentare:

- Interi positivi da 0 a 255
- Interi con segno da -128 a +127

Supponiamo di dover rappresentare interi positivi (0,..., 255)

$$11111111 = (255)_{10}$$

$255 + 1 = ??? \rightarrow$ servirebbero 9 bit

Rappresentazioni a lunghezza fissa: problemi (un breve recap con esempi)

Es 1

Con 8 bit posso rappresentare:

- Interi positivi da 0 a 255
- Interi con segno da -128 a +127

Supponiamo di dover rappresentare interi positivi (0,..., 255)

$$11111111 = (255)_{10}$$

$(255+1)_{10} = 100000000 \rightarrow$ il 9° bit «trabocca» e viene scartato quindi rimane 00000000 ovvero: se ho a disposizione 8 bit il computer mi restituisce 0 anziché 256. Si tratta di overflow: il risultato non ci sta nei bit e «riparte da capo», come se i numeri fossero un cerchio

Rappresentazioni a lunghezza fissa: problemi (un breve recap con esempi)

Es. 2

Con 8 bit posso rappresentare:

- Interi positivi da 0 a 255
- Interi con segno da -128 a +127

Supponiamo di dover rappresentare interi con segno (-128,..., +127)

$$01111111 = (127)_{10}$$

$(127+1)_{10} = 10000000 \rightarrow$ il risultato è negativo come si evince dal primo bit che è destinato al segno (-128)

Rappresentazioni a lunghezza fissa: problemi (un breve recap con esempi)

Es. 3

Con 8 bit posso rappresentare:

- Interi positivi da 0 a 255
- Interi con segno da -128 a +127

Supponiamo di dover rappresentare interi con segno $(-128, \dots, +127)$

01111111 = (127)₁₀

$$(127-128)_{10}$$
[illegible]

Rappresentazioni a lunghezza fissa: problemi (un breve recap con esempi)

Es. 3

Con 8 bit posso rappresentare:

- Interi positivi da 0 a 255
- Interi con segno da -128 a +127

Supponiamo di dover rappresentare interi con segno (-128,..., +127)

○ $1111111 = (127)_{10}$

$(127 - 128)_{10}$

0	1	1	1	1	1	1	1 +
1	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1

Complemento a 2

- inverti) 00000000

- Sommo 1) 00000001 → il binario positivo è 1)

Rappresentazioni a lunghezza fissa: problemi (un breve recap con esempi)

Es. 3

Con 8 bit posso rappresentare:

- Interi positivi da 0 a 255
- Interi con segno da -128 a +127

Supponiamo di dover rappresentare interi con segno (-128,..., +127)

01111111 = (127)₁₀

(127-128)₁₀

0	1	1	1	1	1	1	1 +
1	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1

Non c'è overflow perché -1 è perfettamente rappresentabile in 8 bit!