

RAPPRESENTAZIONE DELL'INFORMAZIONE

- Internamente a un elaboratore, ogni informazione è **rappresentata** tramite ***sequenze di bit (cifre binarie)***
- Una sequenza di bit ***non dice “che cosa” essa rappresenta***

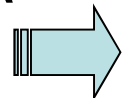
Ad esempio, 01000001 può rappresentare:

- l'intero 65, il carattere 'A', il boolean 'vero', ...
- ... il valore di un segnale musicale,
- ... il colore di un pixel sullo schermo...

Rapida Nota sulla Rappresentazione dei Caratteri

Ad esempio, un tipo fondamentale di dato da rappresentare è costituito dai ***singoli caratteri***

Idea base: associare ***a ciascun carattere un numero intero (codice)*** in modo convenzionale

 ***Codice standard ASCII*** (1968)

ASCII definisce univocamente i primi 128 caratteri (7 bit – vedi tabella nel lucido seguente)

I caratteri con codice superiore a 127 possono variare secondo la particolare codifica adottata (dipendenza da linguaggio naturale: ISO 8859-1 per alfabeto latino1, ...)

Visto che i caratteri hanno un codice intero, essi possono essere considerati un insieme ordinato (ad esempio: 'g' > 'O' perché 103 > 79)

Tabella ASCII standard

Byte	Cod.	Char	Byte	Cod.	Char	Byte	Cod.	Char	Byte	Cod.	Char
00000000	0	Null	00100000	32	SpC	01000000	64	@	01100000	96	`
00000001	1	Start of heading	00100001	33	!	01000001	65	A	01100001	97	a
00000010	2	Start of text	00100010	34	"	01000010	66	B	01100010	98	b
00000011	3	End of text	00100011	35	#	01000011	67	C	01100011	99	c
00000100	4	End of transmit	00100100	36	\$	01000100	68	D	01100100	100	d
00000101	5	Enquiry	00100101	37	%	01000101	69	E	01100101	101	e
00000110	6	Acknowledge	00100110	38	&	01000110	70	F	01100110	102	f
00000111	7	Audible bell	00100111	39	'	01000111	71	G	01100111	103	g
00001000	8	Backspace	00101000	40	(	01001000	72	H	01101000	104	h
00001001	9	Horizontal tab	00101001	41	)	01001001	73	I	01101001	105	i
00001010	10	Line feed	00101010	42	*	01001010	74	J	01101010	106	j
00001011	11	Vertical tab	00101011	43	+	01001011	75	K	01101011	107	k
00001100	12	Form Feed	00101100	44	,	01001100	76	L	01101100	108	l
00001101	13	Carriage return	00101101	45	-	01001101	77	M	01101101	109	m
00001110	14	Shift out	00101110	46	.	01001110	78	N	01101110	110	n
00001111	15	Shift in	00101111	47	/	01001111	79	O	01101111	111	o
00010000	16	Data link escape	00110000	48	0	01010000	80	P	01110000	112	p
00010001	17	Device control 1	00110001	49	1	01010001	81	Q	01110001	113	q
00010010	18	Device control 2	00110010	50	2	01010010	82	R	01110010	114	r
00010011	19	Device control 3	00110011	51	3	01010011	83	S	01110011	115	s
00010100	20	Device control 4	00110100	52	4	01010100	84	T	01110100	116	t
00010101	21	Neg. acknowledge	00110101	53	5	01010101	85	U	01110101	117	u
00010110	22	Synchronous idle	00110110	54	6	01010110	86	V	01110110	118	v
00010111	23	End trans. block	00110111	55	7	01010111	87	W	01110111	119	w
00011000	24	Cancel	00111000	56	8	01011000	88	X	01111000	120	x
00011001	25	End of medium	00111001	57	9	01011001	89	Y	01111001	121	y
00011010	26	Substitution	00111010	58	:	01011010	90	Z	01111010	122	z
00011011	27	Escape	00111011	59	;	01011011	91	[	01111011	123	{
00011100	28	File separator	00111100	60	<	01011100	92	\	01111100	124	
00011101	29	Group separator	00111101	61	=	01011101	93	]	01111101	125	}
00011110	30	Record Separator	00111110	62	>	01011110	94	^	01111110	126	~
00011111	31	Unit separator	00111111	63	?	01011111	95	_	01111111	127	Del

INFORMAZIONI NUMERICHE

Originariamente la ***rappresentazione binaria*** è stata utilizzata per la ***codifica dei numeri e dei caratteri***

Oggi si digitalizzano comunemente anche suoni, immagini, video e altre informazioni (informazioni multimediali)

La rappresentazione delle ***informazioni numeriche*** è ovviamente di particolare rilevanza

A titolo di esempio, nel corso ci concentreremo sulla rappresentazione dei ***numeri interi (senza o con segno)***

Dominio: $N = \{..., -2, -1, 0, 1, 2, ...\}$

NUMERI NATURALI (interi senza segno)

Dominio: $N = \{ 0,1,2,3, \dots \}$

Rappresentabili con diverse notazioni

♦ ***non posizionali***

- ad esempio la notazione romana:
I, II, III, IV, V, IX, X, XI...
- Risulta difficile l'utilizzo di regole generali per il calcolo

♦ ***posizionale***

- 1, 2, .. 10, 11, ... 200, ...
- Risulta semplice l'individuazione di regole generali per il calcolo

NOTAZIONE POSIZIONALE

- Concetto di **base di rappresentazione B**
- Rappresentazione del numero come **sequenza di simboli (cifre)** appartenenti a un **alfabeto di B simboli distinti**
- **ogni simbolo rappresenta un valore compreso fra 0 e $B-1$**

Esempio di rappresentazione su N cifre:

$$d_{n-1} \dots d_2 d_1 d_0$$

NOTAZIONE POSIZIONALE

Il valore di un numero espresso in questa notazione è ricavabile

- ◆ a partire dal valore rappresentato da ogni simbolo
- ◆ pesandolo in base alla *posizione* che occupa nella sequenza



NOTAZIONE POSIZIONALE

In formula:

dove

$$v = \sum_{k=0}^{n-1} d_k B^k$$

♦ $B = \text{base}$

♦ ogni cifra d_k rappresenta un valore fra 0 e $B-1$

Esempio (base B=4):

1 2 1 3

d_3 d_2 d_1 d_0

$$\text{Valore} = 1 * B^3 + 2 * B^2 + 1 * B^1 + 3 * B^0 = \text{centotre}$$

NOTAZIONE POSIZIONALE

Quindi, ***una sequenza di cifre non è interpretabile*** se non si precisa **la base** in cui è espressa

Esempi:

Stringa	Base	Alfabeto	Calcolo valore	Valore
"12"	<i>quattro</i>	{0,1,2,3}	$4 * 1 + 2$	<i>sei</i>
"12"	<i>otto</i>	{0,1,...,7}	$8 * 1 + 2$	<i>dieci</i>
"12"	<i>dieci</i>	{0,1,...,9}	$10 * 1 + 2$	<i>dodici</i>
"12"	<i>sedici</i>	{0,...,9, A,., F}	$16 * 1 + 2$	<i>diciotto</i>

NOTAZIONE POSIZIONALE

Inversamente, ogni numero può essere espresso, *in modo univoco*, *come sequenza di cifre in una qualunque base*

Esempi:

Numero	Base	Alfabeto	Rappresentazione
<i>venti</i>	<i>due</i>	{0,1}	“10100”
<i>venti</i>	<i>otto</i>	{0,1,...,7}	“24”
<i>venti</i>	<i>dieci</i>	{0,1,...,9}	“20”
<i>venti</i>	<i>sedici</i>	{0,...,9, A,.., F}	“14”

Non bisogna *confondere* un numero con una sua RAPPRESENTAZIONE!

NUMERI E LORO RAPPRESENTAZIONE

- **Internamente**, un elaboratore adotta per i *numeri interi* una **rappresentazione binaria (base $B=2$)**
- **Esternamente**, le costanti numeriche che scriviamo nei programmi e i valori che stampiamo a video/leggiamo da tastiera sono invece **sequenze di caratteri ASCII**

Il passaggio dall'una all'altra forma richiede dunque un **processo di conversione**

Esempio: RAPPRESENTAZIONE INTERNA/ESTERNA

- Numero: *centoventicinque*
- Rappresentazione interna binaria (16 bit):

00000000 01111101

- Rappresentazione esterna in base 10:

*occorre produrre la sequenza di
caratteri ASCII '1', '2', '5'*

00110001 00110010 00110101

vedi tabella ASCII

Esempio: RAPPRESENTAZIONE INTERNA/ESTERNA

- Rappresentazione esterna in base 10:

*È data la sequenza di caratteri ASCII
'3', '1', '2', '5', '4'*

vedi tabella ASCII

00110011 00110001 00110010 00110101 00110100

- Rappresentazione interna binaria (16 bit):

01111010 00010110

- Numero:

trentunomiladuecentocinquantaquattro

CONVERSIONE STRINGA/NUMERO

Si applica la definizione:

$$v = \sum_{k=0}^{n-1} d_k B^k$$

**le cifre d_k sono note,
il valore v va calcolato**

$$= d_0 + B * (d_1 + B * (d_2 + B * (d_3 + ...)))$$

Ciò richiede la valutazione di un polinomio

→ ***Metodo di Horner***

CONVERSIONE NUMERO/STRINGA

- Problema: ***dato un numero, determinare la sua rappresentazione in una base data***
- Soluzione (***notazione posizionale***): ***manipolare la formula*** per dedurre un algoritmo

$$v = \sum_{k=0}^{n-1} d_k B^k$$

**v è noto,
le cifre d_k vanno calcolate**

$$= d_0 + B * (d_1 + B * (d_2 + B * (d_3 + ...)))$$

CONVERSIONE NUMERO/STRINGA

Per trovare le cifre bisogna *calcolarle una per una*, ossia bisogna trovare un modo per *isolarne una dalle altre*

$$v = d_0 + B * (...)$$

Osservazione:

d_0 è la sola cifra non moltiplicata per B

Conseguenza:

d_0 è ricavabile come v modulo B

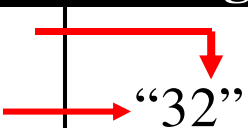
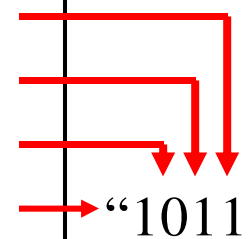
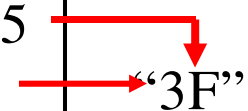
CONVERSIONE NUMERO/STRINGA

Algoritmo delle divisioni successive

- si divide v per B
 - *il resto* costituisce la cifra meno significativa (d_0)
 - *il quoziente* serve a iterare il procedimento
- se tale quoziente è zero, l'algoritmo termina;
- se non lo è, lo si assume come nuovo valore v' , e si itera il procedimento con il valore v'

CONVERSIONE NUMERO/STRINGA

Esempi:

Numero	Base	Calcolo valore	Stringa
<i>quattordici</i>	4	$14 / 4 = 3$ con resto 2 $3 / 4 = 0$ con resto 3	 "32"
<i>undici</i>	2	$11 / 2 = 5$ con resto 1 $5 / 2 = 2$ con resto 1 $2 / 2 = 1$ con resto 0 $1 / 2 = 0$ con resto 1	 "1011"
<i>sessantatre</i>	10	$63 / 10 = 6$ con resto 3 $6 / 10 = 0$ con resto 6	 "63"
<i>sessantatre</i>	16	$63 / 16 = 3$ con resto 15 $3 / 16 = 0$ con resto 3	 "3F"

NUMERI NATURALI: valori rappresentabili

- **Con N bit, si possono fare 2^N combinazioni**
- **Si rappresentano così i numeri da 0 a 2^N-1**

Esempi

□ **con 8 bit, [0 255]**

In C: unsigned char = byte

□ **con 16 bit, [0 65.535]**

In C: unsigned short int (su alcuni compilatori)

In C: unsigned int (su alcuni compilatori)

□ **con 32 bit, [0 4.294.967.295]**

In C: unsigned int (su alcuni compilatori)

In C: unsigned long int (su molti compilatori)

OPERAZIONI ED ERRORI

La rappresentazione binaria rende possibile fare *addizioni e sottrazioni con le usuali regole algebriche*

Esempio:

5	+	0101
3	=	0011
---		-----
8		1000

ERRORI (primitissimo assaggio 😊)

Esempio (supponendo di avere solo 7 bit per la rappresentazione)

$$\begin{array}{rcl} 60 & + & 0111100 \\ 99 & = & 1100011 \\ \hline 135 & & \textcolor{red}{1}0011111 \end{array}$$

Errore!
Massimo numero
rappresentabile:
 $2^7 - 1$ cioè 127

- Questo errore si chiama **overflow**
- **Può capitare sommando due numeri dello stesso segno** il cui risultato non sia rappresentabile utilizzando **il numero massimo di bit** designati

ESERCIZIO RAPPRESENTAZIONE

Un elaboratore rappresenta numeri interi su **8 bit** dei quali **7** sono dedicati alla rappresentazione del modulo del numero e **uno** al suo **segno**. Indicare come viene svolta la seguente operazione aritmetica:

$$59 - 27$$

in codifica binaria

ESERCIZIO RAPPRESENTAZIONE

Soluzione

59 → 0 0111011

-27 → 1 0011011

Tra i (moduli dei) due numeri si esegue una sottrazione:

$$\begin{array}{r} 0111011 \\ - 0011011 \\ \hline 0100000 \end{array}$$

che vale 32 in base 10

DIFFERENZE TRA NUMERI BINARI

Che cosa avremmo dovuto fare se avessimo avuto
 $27-59$?

Avremmo dovuto invertire i due numeri, calcolare il risultato, e poi ricordarci di mettere a 1 il bit rappresentante il segno

Per ovviare a tale problema, si usa la ***notazione in “complemento a 2”*** (vedi nel seguito), che permette di eseguire tutte le differenze tramite semplici somme

INFORMAZIONI NUMERICHE

- La rappresentazione delle *informazioni numeriche* è di particolare rilevanza
- Abbiamo già discusso i *numeri naturali* (*interi senza segno*) $N = \{ 0, 1, 2, 3, \dots \}$
- Come rappresentare invece i ***numeri interi*** (***con segno***)?

$$\mathbf{Z} = \{ -x, x \in N - \{0\} \} \cup N$$

NUMERI INTERI (con segno)

Dominio: $\mathbb{Z} = \{ \dots, -2, -1, 0, 1, 2, 3, \dots \}$

Rappresentare gli interi in un elaboratore pone alcune problematiche:

- *come rappresentare il “**segno meno**”?*
- *possibilmente, come rendere **semplice l'esecuzione delle operazioni aritmetiche**?*

Magari riutilizzando gli stessi algoritmi e gli stessi circuiti già usati per i numeri interi senza segno

NUMERI INTERI (con segno)

Due possibilità:

- ***rappresentazione in modulo e segno***
 - ❑ semplice e intuitiva
 - ❑ ma inefficiente e complessa nella gestione delle operazioni → *non molto usata in pratica*
- ***rappresentazione in complemento a due***
 - ❑ meno intuitiva, costruita “ad hoc”
 - ❑ ma efficiente e capace di rendere semplice la gestione delle operazioni → *largamente usata nelle architetture reali di CPU*

NUMERI INTERI (con segno)

Rappresentazione in modulo e segno

- 1 bit *per rappresentare il segno*

$$0 + 1 -$$

- (N-1) bit *per il valore assoluto*

Esempi (su 8 bit, Most Significant Bit –MSB- rappresenta il segno):

+ 5 = 0 0000101

- 36 = 1 0100100

NUMERI INTERI (con segno)

Rappresentazione *in modulo e segno*

Due difetti principali:

- **occorrono *algoritmi speciali* per fare le operazioni**

❑ se si adottano le usuali regole
non è verificata la proprietà $X + (-X) = 0$

❑ occorrono regole (e quindi circuiti) ad hoc

- ***due diverse rappresentazioni per lo zero***

$+ 0 = 00000000$

$- 0 = 10000000$

Ad esempio:

$(+5) + (-5) = -10 ???$

+5	0	0000101
-5	1	0000101
---	---	-----
0	1	0001010

NUMERI INTERI (con segno)

Rappresentazione in complemento a due

- *si vogliono poter usare le regole standard per fare le operazioni*
- in particolare, si vuole che
 - $X + (-X) = 0$
 - la rappresentazione dello zero sia *unica*
- *anche a prezzo di una notazione più complessa, meno intuitiva, e magari non (completamente) posizionale*

RAPPRESENTAZIONE in COMPLEMENTO A DUE

- **idea: cambiare il valore del bit più significativo da $+2^{N-1}$ a -2^{N-1}**
- **peso degli altri bit rimane lo stesso** (come numeri naturali)

Esempi:

$$\mathbf{0\ 0000101} = \mathbf{+5}$$

$$\mathbf{1\ 0000101} = \mathbf{-128 + 5 = -123}$$

$$\mathbf{1\ 1111101} = \mathbf{-128 + 125 = -3}$$

NB: in caso di MSB=1, gli altri bit NON sono il valore assoluto del numero naturale corrispondente

INTERVALLO DI VALORI RAPPRESENTABILI

- ***se $MSB=0$, stesso dei naturali con $N-1$ bit***

da 0 a $2^{N-1}-1$

Esempio: su 8 bit, $[0, +127]$

- ***se $MSB=1$, stesso intervallo traslato di -2^{N-1}***

da -2^{N-1} a -1

Esempio: su 8 bit, $[-128, -1]$

- **Intervallo globale = unione $[-2^{N-1}, 2^{N-1}-1]$**

con 8 bit, $[-128 \dots +127]$

con 16 bit, $[-32.768 \dots +32.767]$

con 32 bit, $[-2.147.483.648 \dots +2.147.483.647]$

CONVERSIONE NUMERO/STRINGA

- Osservazione: poiché si opera su N bit, questa è in realtà una *aritmetica mod 2^N*
- ***Rappresentazione del numero v coincide con quella del numero $v \pm 2^N$***
- In particolare, la rappresentazione del negativo v coincide con quella del positivo $v' = v + 2^N$

$$v = -d_{n-1}B^{n-1} + \sum_{k=0}^{n-2} d_k B^k$$

Questo è un naturale

$$v' = +d_{n-1}B^{n-1} + \sum_{k=0}^{n-2} d_k B^k$$

CONVERSIONE NUMERO/STRINGA

Esempio (*8 bit*, $2^N = 256$):

per calcolare la rappresentazione di -3, possiamo calcolare quella del naturale

$$-3 + 256 = 253$$

- con la definizione di compl. a 2 ($2^{N-1} = 128$):

$$\textcolor{red}{-3} = \textcolor{red}{-128} + 125 \rightarrow \textcolor{red}{\text{"11111101"}}$$

- con il trucco sopra:

$$\textcolor{red}{-3} \rightarrow \textcolor{red}{253} \rightarrow \textcolor{red}{\text{"11111101"}}$$

CONVERSIONE NUMERO/STRINGA

Come svolgere questo calcolo in modo semplice?

- Se $v < 0$:

$$v = -|v|$$

$$v' = v + 2^N = 2^N - |v|$$

- che si può riscrivere come $v' = (2^N - 1) - |v| + 1$
- dove la quantità $(2^N - 1)$ è, in binario, una **sequenza di N cifre a “1”**

Ma la sottrazione $(2^N - 1) - |v|$ si limita a ***invertire tutti i bit*** della rappresentazione di $|v|$

Infatti, ad esempio, su 8 bit:

- $2^8 - 1 = 11111111$

- se $|v| = 01110101$

$$(2^8 - 1) - |v| = 10001010$$

CONVERSIONE NUMERO/STRINGA

Conclusione:

- per calcolare il numero negativo $-|v|$, la cui rappresentazione coincide con quella del positivo $v' = (2^N - 1) - |v| + 1$, occorre
- ***prima invertire tutti i bit*** della rappresentazione di $|v|$ (calcolando così $(2^N - 1) - |v|$)
- ***poi aggiungere 1*** al risultato

Algoritmo di calcolo del complemento a due

CONVERSIONE NUMERO/STRINGA

Esempi

- **$v = -3$**

valore assoluto 3 → “00000011”

inversione dei bit → “11111100”

somma con 1 → “11111101”

- **$v = -37$**

valore assoluto 37 → “00100101”

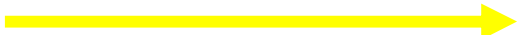
inversione dei bit → “11011010”

somma con 1 → “11011011”

CONVERSIONE STRINGA/NUMERO

Importante:

l'algoritmo funziona *anche a rovescio*

- stringa = “1111101”  -3
 - inversione dei bit → “00000010”
 - somma con 1 → “00000011”
 - calcolo valore assoluto → 3
- stringa = “11011011”  -37
 - inversione dei bit → “00100100”
 - somma con 1 → “00100101”
 - calcolo valore assoluto → 37

OPERAZIONI SU NUMERI INTERI

Rappresentazione in complemento a due rende possibile fare *addizioni e sottrazioni con le usuali regole algebriche*

Ad esempio:

-5	+	11111011
+3	=	00000011
---		-----
-2		11111110

In certi casi occorre però *una piccola convenzione: ignorare il riporto*

Un altro esempio:

-1	+	11111111
-5	=	11111011
---		-----
-6		(1) 11111010

COMPLEMENTO a 2: NUMERI a 4 bit

a. Using patterns of length three

Bit pattern	Value represented
011	3
010	2
001	1
000	0
111	-1
110	-2
101	-3
100	-4

b. Using patterns of length four

Bit pattern	Value represented
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

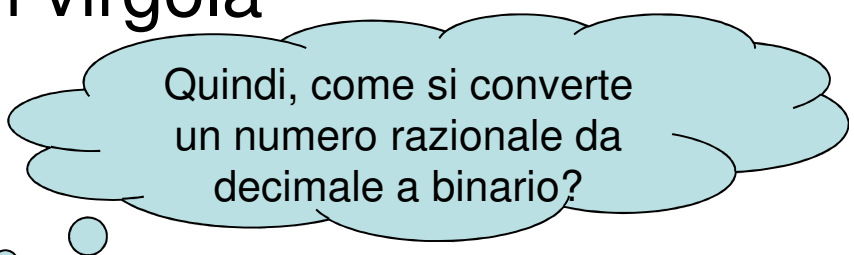
ESEMPI DI SOMME

Problem in base ten		Problem in two's complement		Answer in base ten
$\begin{array}{r} 3 \\ + 2 \\ \hline \end{array}$	→	$\begin{array}{r} 0011 \\ + 0010 \\ \hline 0101 \end{array}$	→	5
$\begin{array}{r} -3 \\ + -2 \\ \hline \end{array}$	→	$\begin{array}{r} 1101 \\ + 1110 \\ \hline 1011 \end{array}$	→	-5
$\begin{array}{r} 7 \\ + -5 \\ \hline \end{array}$	→	$\begin{array}{r} 0111 \\ + 1011 \\ \hline 0010 \end{array}$	→	2

Rappresentazione dei Numeri: NUMERI RAZIONALI

Ovviamente ***rappresentazione posizionale*** vale anche per numeri “con virgola”

$$a = \sum_{i=-N}^{M-1} c_i * b^i$$



Quindi, come si converte
un numero razionale da
decimale a binario?

Rappresentazione in ***virgola mobile*** (***mantissa ed esponente***), utilizzando base 2

Standard IEEE 754 (rappresentazione ***float a 32 bit***):

- ***Bit più significativo per segno mantissa*** (0 per positivi, 1 per negativi)
- ***8 bit per l'esponente*** (notazione in complemento a 2)
- Ultimi ***23 bit per mantissa*** in forma normale

Rappresentazione dei Numeri: NUMERI RAZIONALI

- **zero** si rappresenta ponendo a **0 tutti i bit di mantissa ed esponente**
- **$+\infty$ e $-\infty$** si possono rappresentare ponendo a **1 tutti i bit dell'esponente e a 0 tutti i bit della mantissa**
- **Not a Number (NaN)** ponendo a **1 tutti i bit dell'esponente e la mantissa diversa da 0**

Nel caso di **doppia precisione – double**: 11 bit per esponente, 52 per mantissa

Valori massimo e minimo rappresentabili

Interi senza segno	0	4294967295
Interi con segno	-2147483647	2147483647
Interi in complemento a 2	-2147483648	2147483647
Reali in sing. prec. (modulo)	$1.401298 \cdot 10^{-45}$	$3.402823 \cdot 10^{38}$
Reali in doppia prec. (modulo)	$4.940656 \cdot 10^{-324}$	$1.797693 \cdot 10^{308}$

Errori di Approssimazione: OVERFLOW

- Se si sommano due numeri positivi tali che il risultato sia ***maggiore del massimo numero positivo rappresentabile*** con i bit fissati (lo stesso per somma di due negativi)
- Basta guardare il bit di segno del risultato: se 0 (1) e i numeri sono entrambi negativi (positivi) → ***overflow***

ERRORI NELLE OPERAZIONI

Attenzione ai casi in cui ***venga invaso il bit più significativo (bit di segno)***

Esempio

60	+	00111100
75	=	01100011
135		1 0011111

Errore!

Si è invaso il bit di segno,
il risultato è *negativo*!

Questo errore si chiama ***invasione del bit di segno***; è una **forma di *overflow***

Può accadere solo ***sommando due numeri dello stesso segno, con modulo sufficientemente grande***

Errori di Approssimazione: ERRORI DI ARROTONDAMENTO e UNDERFLOW

Tutti i numeri non interi si ***approssimano al numero razionale più vicino con rappresentazione finita***
(pensare a $\sqrt{2}$ ma anche a 0.1)

Precisione dell'ordine di 2^n , dove n è il numero di bit dedicati alla mantissa

- ❖ Due numeri che ***differiscono per meno della precisione*** sono considerati uguali => ***errore di arrotondamento***
- ❖ Se un ***numero è più piccolo della precisione***, allora viene approssimato con 0 => ***errore di underflow***

PROPAGAZIONE degli ERRORI

Propagazione “catastrofica” in alcuni algoritmi
quando **operandi** (numeri razionali) **molto diversi o molto simili**

1) Molto diversi: il più piccolo dei due operandi perde precisione a causa spostamento mantissa (necessario per svolgere operazione: stessa mantissa del più grande)

Esempio:

se a sufficientemente grande e b sufficientemente piccolo

$$\Rightarrow a + b = a$$

(chi propone un esempio concreto in rappres. IEEE754?)

PROPAGAZIONE degli ERRORI

Propagazione “catastrofica” in alcuni algoritmi
quando **operandi** (numeri razionali) **molto diversi o molto simili**

2) Molto simili: differenza fra due numeri molto vicini

Esempio (in base 10 per semplificare; mantissa con 3 cifre significative dopo virgola):

$$a = 1.000 \times 10^3; b = 999.8 \times 10^0$$

$$a - b = (1.000 \times 10^3) - (0.999 \times 10^3) = 1.000 \times 10^0, \text{ che è piuttosto diverso da } 0.2!$$

Nota che **diversi algoritmi possono produrre errori molto differenti.**

Ad esempio ($x = 3.451 \times 10^0$, $y = 3.450 \times 10^0$):

$$x^2 - y^2 = 1.190 \times 10^1 - 1.190 \times 10^1 = 0!!!$$

$$x^2 - y^2 = (x + y) (x - y) = (6.901 \times 10^0) (1.000 \times 10^{-3}) = 6.901 \times 10^{-3}$$

Esempio di PROPAGAZIONE in ALGORITMO ITERATIVO

Attenzione ai ***problemi di arrotondamento*** in algoritmi iterativi, possibili anche in casi in cui non ce lo aspettiamo!

Esempio di somma di N numeri naturali

```
#include <stdio.h>
```

```
#define N 10000000
```

```
main () {
```

```
    float s=0, x=7;
```

```
    unsigned int i, is=0; ix=7;
```

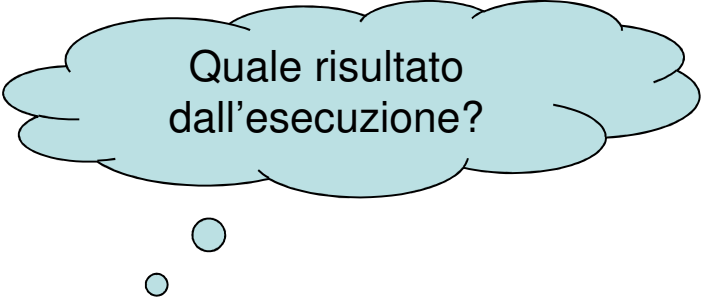
```
    for (i=0; i<N; i++) {
```

```
        s += x; is += ix; }
```

```
    printf("usando float:  %.0f x %d = %.0f\n", x, N, s);
```

```
    printf("usando integer:  %d x %d = %d\n", ix, N, is);
```

```
}
```



Quale risultato
dall'esecuzione?

Esempio di PROPAGAZIONE in ALGORITMO ITERATIVO

Errore già per $i = 2396746$, $s = 16777222$

Notazione IEEE 754; perdita della cifra più a destra
dell'addendo 7 (approssimato a 6)

Quindi un risultato finale più basso? NO, NEMMENO

Situazione ancora più complessa: rappresentazione interna alla
Floating Point Unit (FPU) a 80 bit, troncata quando si va in
memoria centrale

Ovviamente ***stesso problema*** di arrotondamento su
somma di ***qualunque coppia di numeri che
differiscono troppo fra loro***

E se avessimo a che fare con un ***ottimizzatore di codice***
sufficientemente intelligente?

COME RISOLVERE/ATTENUARE IL PROBLEMA?

Algoritmi equivalenti alternativi che riducano la differenza tra i numeri da sommare

Ad esempio: sommare prima m valori fra loro; poi sommare tra loro i k risultati (dove ovviamente $k*m = n$)

Oppure somma di Kahan:

```
float sum = 0., corr = 0., x = 7., tmp, y; int i;
for (i=0; i<N; i++) {
    y = corr + x;
    tmp = sum + y;
    corr = (sum - tmp) + y;
    sum = tmp; }
sum += corr;
```