

# 1 - Nozioni di background su Architetture degli Elaboratori, Linguaggi di Programmazione e Algoritmi

Programmazione e analisi di dati  
Modulo A: Programmazione in Java

Paolo Milazzo

Dipartimento di Informatica, Università di Pisa  
<http://www.di.unipi.it/~milazzo>  
[milazzo@di.unipi.it](mailto:milazzo@di.unipi.it)

Corso di Laurea Magistrale in Informatica Umanistica  
A.A. 2014/2015

# Sommario

- 1 Come è fatto un computer
- 2 Rappresentazione binaria dell'informazione
- 3 Linguaggi di programmazione e algoritmi

# Com'è fatto un computer (1)

In questo corso impareremo a scrivere programmi che dovranno essere eseguiti da un computer

E' quindi bene sapere com'è fatto e come funziona un computer...

## Com'è fatto un computer (2)

Il miglior modo per capire com'è fatta una cosa è... smontarla!



Foto da tutorial PC professionale

<http://www.pcprofessionale.it/2012/03/28/costruire-un-pc-la-guida-passo-passo-completa/>

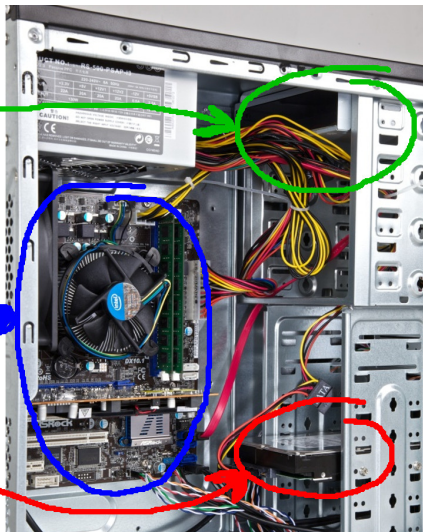
# Com'è fatto un computer (3)

Identifichiamo i componenti più "appariscenti"

Lettoce DVD/  
Masterizzatore

Scheda madre

Hard disk



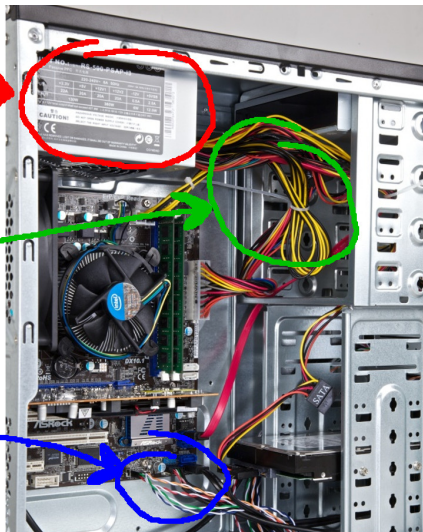
# Com'è fatto un computer (4)

Innanzitutto dobbiamo scollegare i cavi di alimentazione...

Alimentatore →

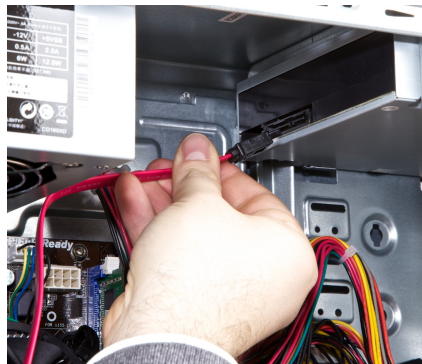
Cavetti di alimentazione

Cavetti vari  
(pulsante on/off,  
led frontali,  
altop. interno,... )



# Com'è fatto un computer (5)

... e i cavi-dati



## Com'è fatto un computer (6)

Ora possiamo rimuovere il DVD e l'hard disk





## Com'è fatto un computer (7)

A questo punto possiamo dedicarci alla scheda madre e ai componenti ad essa collegati. Per prima la scheda video (a cui era collegato il monitor)



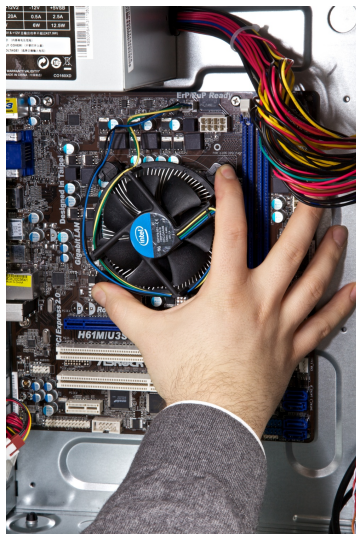
# Com'è fatto un computer (8)

Ora la memoria RAM



## Com'è fatto un computer (9)

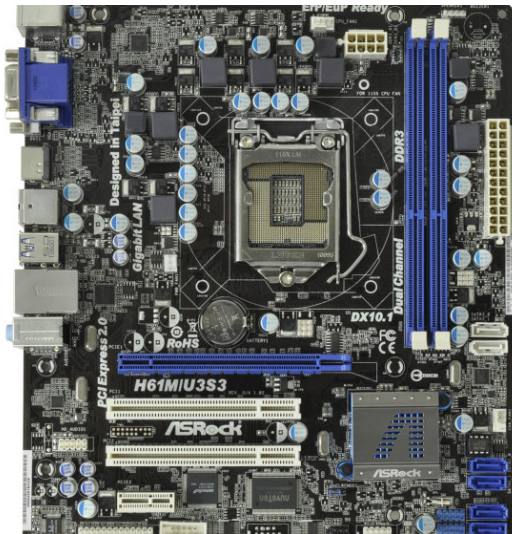
Sotto il grosso dissipatore (con ventola) posizionato al centro della scheda madre...





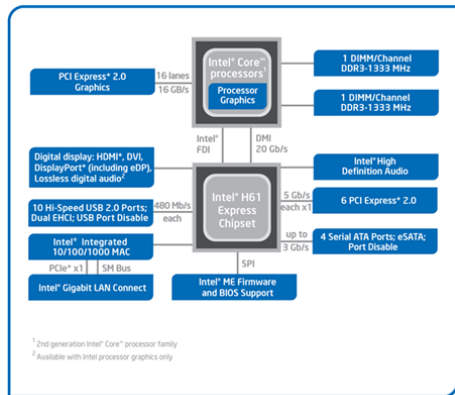
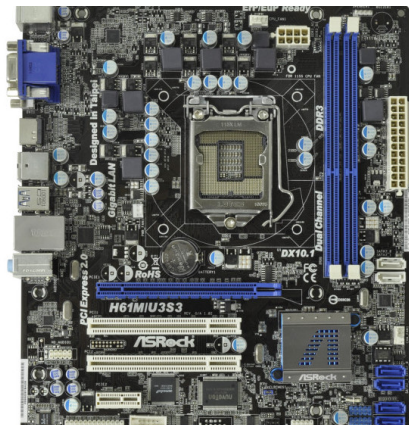
# Com'è fatto un computer (10)

E infine non ci rimane che rimuovere la scheda madre



# Com'è fatto un computer (11)

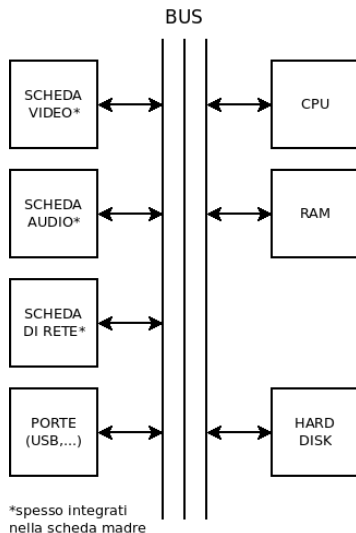
Una rappresentazione schematica (ufficiale – Intel). I componenti sono collegati tramite BUS di comunicazione gestiti dal processore e da un apposito chipset.



Intel® H61 Express Chipset Platform Block Diagram

## Com'è fatto un computer (12)

Una rappresentazione semplificata (unico bus di comunicazione in cui il chipset non è rappresentato)



# La CPU e la memoria RAM

I componenti chiave per l'esecuzione di un programma sono la CPU e la memoria RAM.

- La **CPU** è un chip capace di eseguire operazioni molto semplici (operazioni aritmetiche, operazioni logiche, scrittura/lettura di piccole quantità di dati nella memoria RAM, ecc...) in tempi rapidissimi, nell'ordine dei miliardi di operazioni al secondo (in teoria...).
- La **memoria RAM** è una memoria volatile (si cancella quando non è alimentata) che consente di memorizzare grandi quantità di dati rappresentati in formato binario (ossia come sequenze di valori 0 e 1).



# La CPU

Le operazioni che può svolgere la CPU sono codificate tramite un linguaggio in codice binario detto **linguaggio macchina**

```
Program Fragment:      Y = Y + X
Machine Language Code
(Binary Code)

Opcode      Address
1100 0000   0010 0000 0000 0000
1011 0000   0001 0000 0000 0000
1001 0000   0010 0000 0000 0000
```

Il linguaggio macchina è di solito rappresentato tramite una notazione simbolica detta **linguaggio assembly**

```
mov bx,ax      ;copy the content of ax into bx
add ax,bx      ;add value in bx to value in ax
sub bx,1       ;subtract 1 from the bx value
add [bx],ax    ;add value in ax to memory _word_ pointed to by bx
add [bx],al    ;add value in al to memory _byte_ pointed to by bx
```

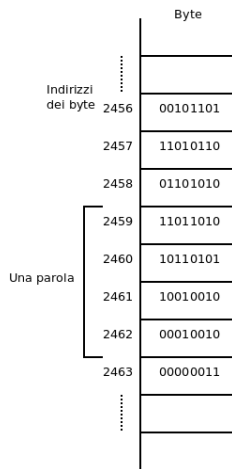
# La memoria RAM

Un modulo di memoria RAM può memorizzare decine di miliardi di bit 0/1.

- I bit sono organizzati in gruppi di 8, detti **byte**
- I byte sono organizzati in gruppi di 4 (32 bit) oppure 8 (64 bit) detti **word** (o **parola**)

I singoli byte della memoria RAM sono associati ad un valore numerico (**indirizzo di memoria**) che li distingue.

Una parola solitamente è la quantità **massima** di memoria che una CPU può utilizzare in una singola operazione. Operazioni che lavorano su dati più grandi vengono svolte in più passi dalla CPU.



# E ora?

Prima di poter programmare un computer dobbiamo:

- trovare un modo per rappresentare i nostri dati in memoria
  - ▶ **Rappresentazione binaria**
- trovare dei linguaggi più semplici da usare rispetto all'assembly
  - ▶ **Linguaggi di programmazione e algoritmi**

Vediamo questi due argomenti in dettaglio...

# Sommario

- 1 Come è fatto un computer
- 2 Rappresentazione binaria dell'informazione**
- 3 Linguaggi di programmazione e algoritmi

# Rappresentazione binaria dell'informazione (1)

Per informazione intendiamo tutto quello che viene manipolato da un calcolatore:

- Numeri (naturali, interi, reali, ...)
- Caratteri
- Immagini
- Suoni
- Programmi

## Rappresentazione binaria dell'informazione (2)

La più piccola unità di informazione memorizzabile o elaborabile da un calcolatore, il **bit**, corrisponde allo stato di un dispositivo fisico (ad esempio spento/acceso) che viene interpretato come 0 o 1.

In un calcolatore tutte le informazioni sono rappresentate in forma **binaria**, come sequenze di 0 e 1.

Per **motivi tecnologici**: distinguere tra due valori di una grandezza fisica è più semplice che non ad esempio tra dieci valori

- Ad esempio, verificare se su un connettore c'è una tensione elettrica di 5V o meno **è più facile** che verificare se sullo stesso connettore c'è una tensione di 0V, 1V, 2V, 3V, 4V o 5V.

# Rappresentazione posizionale (1)

Un numero naturale può essere **rappresentato** mediante una **sequenza di simboli** in diversi modi

E' importante distinguere tra un numero e la sua rappresentazione:

- la **sequenza di cifre** "234" è la rappresentazione decimale del numero 234
- la **sequenza di cifre romane** "CCXXXIV" è un'altra rappresentazione dello stesso numero

La rappresentazione decimale è un esempio di rappresentazione **posizionale**:

- ogni cifra contribuisce al numero con un valore che dipende dalla posizione in cui si trova nella sequenza
- ad esempio: nella rappresentazione "2426" il primo 2 contribuisce più del secondo (2000 vs 20) a rappresentare il numero 2426

## Rappresentazione posizionale (2)

Nella sequenza di cifre:

$$c_{n-1} c_{n-2} \cdots c_1 c_0$$

- la cifra  $c_0$  viene detta cifra **meno significativa**
- la cifra  $c_{n-1}$  viene detta cifra **più significativa**

Ad esempio, nella rappresentazione "2435"

- la cifra meno significativa è 5
- la cifra più significativa è 2



# Rappresentazione di numeri naturali (1)

La rappresentazione decimale dei numeri naturali si basa su un insieme di cifre costituito da 10 simboli (0,1,...,9).

Il numero **b** di cifre usate è detto **base** del sistema di numerazione. Ad ogni cifra è associato un valore compreso tra 0 e  $b-1$ .

Base	Cifre	Sistema	Esempio
2	0,1	binario	1001010110
8	0,1,...,7	ottale	40367
10	0,1,...,9	decimale	3954
16	0,1,...,9,A,...,F	esadecimale	2DE4

Nel sistema esadecimale **A vale 10, B vale 11, ..., F vale 15.**

## Rappresentazione di numeri naturali (2)

Esempi di rappresentazioni nelle diverse basi (tralasciamo il sistema ottale):

<b>Decimale</b>	<b>Binario</b>	<b>Esadecimale</b>
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
9	1001	9
10	1010	A
15	1111	F
16	10000	10
20	10100	14
26	11010	1A

## Rappresentazione di numeri naturali (3)

Il numero  $N$  rappresentato da una sequenza di cifre  $c_{n-1} c_n \cdots c_0$  **dipende dalla base  $b$**  e si ottiene tramite la seguente formula:

$$N = c_{n-1} \cdot b^{n-1} + c_{n-2} \cdot b^{n-2} + \cdots + c_1 \cdot b^1 + c_0 \cdot b^0$$

Ad esempio:

- il numero binario 1101 corrisponde a  $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13$
- il numero ottale 345 corrisponde a  $3 \cdot 8^2 + 4 \cdot 8^1 + 5 \cdot 8^0 = 229$
- il numero decimale 243 corrisponde a  $2 \cdot 10^2 + 4 \cdot 10^1 + 3 \cdot 10^0 = 243$
- il numero esadecimale 2FA corrisponde a  $2 \cdot 16^2 + 15 \cdot 16^1 + 10 \cdot 16^0 = 762$

# Rappresentazione binaria di numeri naturali (1)

Nella pratica, **per convertire un numero binario in formato decimale** si può costruire una semplice tabellina come nel seguente esempio:

Numero da convertire: 10110101

Numero da convertire	1	0	1	1	0	1	0	1
Potenze di 2 corrispondenti	$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$

Il risultato è la somma delle potenze di 2 associate a valori 1 nella tabellina (in rosso):

$$\begin{aligned} & 2^7 + 2^5 + 2^4 + 2^2 + 2^0 \\ &= 128 + 32 + 16 + 4 + 1 \\ &= 181 \end{aligned}$$

Il risultato coincide con quanto calcolato dalla formula definita in precedenza.

## Rappresentazione binaria di numeri naturali (2)

Anche **per convertire un numero decimale in formato binario** si può costruire una tabellina, in cui si divide ripetutamente il numero decimale per 2.

Il numero binario sarà dato dai **resti delle divisioni** presi in ordine inverso.

Esempio: numero da convertire: 25

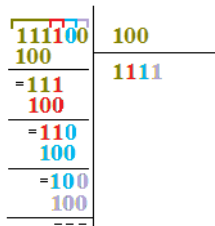
<b>N : 2</b>	<b>Quoziente</b>	<b>Resto</b>	<b>Cifra</b>
25 : 2	12	1	$c_0$
12 : 2	6	0	$c_1$
6 : 2	3	0	$c_2$
3 : 2	1	1	$c_3$
1 : 2	0	1	$c_4$

Il risultato è quindi  $c_4 c_3 c_2 c_1 c_0$  che corrisponde a 11001.

# Operazioni aritmetiche su numeri binari

Le operazioni aritmetiche su numeri binari sono analoghe a quelle su numeri decimali

- Ma ricordando che  $1 + 1 = 10$

$$\begin{array}{r} 11100 + 1011 = 100111 \\ 1011 = 00110011 = 01100110 \\ 10011001 \times 1011 = 11010010011 \end{array}$$


Le operazioni aritmetiche possono creare situazioni di **overflow** (trabocco) quando la rappresentazione del risultato richiede più bit di quelli a disposizione

- Queste situazioni vengono monitorate dalla CPU, ciò nonostante usando certi linguaggi di programmazione (ad es. il C) bisogna porvi attenzione

# Intervallo di rappresentazione

Quanti numeri si possono rappresentare con un byte? E con una parola?

Nel sistema binario con  $n$  cifre si possono rappresentare  $2^n$  valori diversi,

- i valori vanno da 0 a  $2^n - 1$

Ad esempio:

Con 1 cifra binaria (1 bit)	$2^1$ valori	{0, 1}
Con 8 cifre binarie (1 byte)	$2^8$ valori	{0, 1, ..., 255}
Con 16 cifre binarie (2 byte)	$2^{16}$ valori	{0, 1, ..., 65.535}
Con 32 cifre binarie (4 byte)	$2^{32}$ valori	{0, 1, ..., 4.294.967.295}

# Rappresentazione binaria di numeri interi (1)

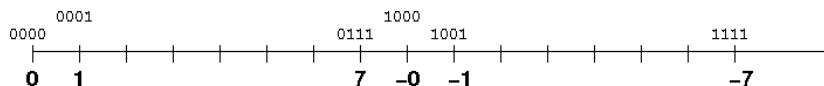
Nella rappresentazione vista abbiamo considerato solo numeri senza segno

Quando si considerano valori interi, per rappresentare il **segno** si può pensare di usare uno dei bit (quello più significativo)

Questa convenzione viene detta rappresentazione tramite **modulo e segno**

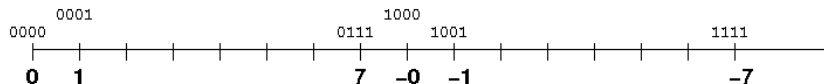
- Il bit più significativo ( $c_{n-1}$ ) rappresenta il segno
- le altre cifre ( $c_{n-2} \cdots c_0$ ) rappresentano il valore assoluto

Esempio con 4 bit:





## Rappresentazione binaria di numeri interi (2)



La rappresentazione tramite **modulo e segno** presenta diversi **problemi**:

- doppia rappresentazione dello zero (come 00...00 e 10...00)
- le operazioni aritmetiche diventano complicate (analisi per casi)

## Rappresentazione in complemento a 2 (1)

Una rappresentazione alternativa a quella modulo e segno è la rappresentazione **in complemento a 2**.

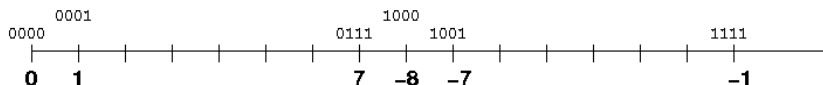
Come prima si usa il bit più significativo per rappresentare il **segno**

Anche la rappresentazione dei **numeri positivi** non cambia

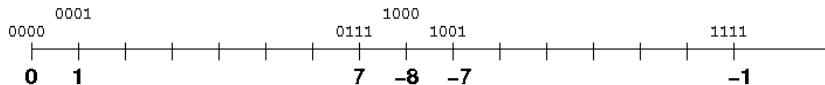
Un numero negativo  $-N$  invece viene rappresentato come  $2^n - N$ , dove  $n$  è il numero dei bit a disposizione (incluso il bit del segno).

Ad esempio, con 4 bit ( $n = 4$ ) rappresentiamo il numero negativo  $-3$  come  $2^4 - 3 = 16 - 3 = 13$  che in binario è 1101.

Quindi:



## Rappresentazione in complemento a 2 (2)

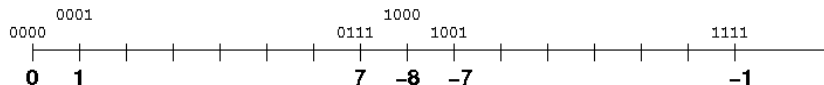


La rappresentazione in **complemento a 2** presenta diversi **vantaggi**

- Lo zero ha una sola rappresentazione (00...00)
- Le operazioni aritmetiche rimangono facili
  - ▶ Si può trasformare un numero positivo in un negativo semplicemente (vedere dopo...)
  - ▶ La somma è come abbiamo visto in precedenza (bit a bit)
  - ▶ La sottrazione si ottiene trasformando il secondo operando in negativo e sommando
  - ▶ Moltiplicazione e divisione si basano su somma e sottrazione

Per trasformare un numero positivo nel corrispondente negativo è sufficiente invertire tutti i bit e sommare 1

## Rappresentazione in complemento a 2 (3)



Esempio: trasformare 5 in -5 (con 4 bit)

- 5 è 0101
- inverto tutti i bit: 1010
- sommo uno: 1011
- quello che ottengo è -5

# Rappresentazione di numeri reali (1)

Anche in un intervallo chiuso, l'insieme dei numeri reali (e dei razionali) è infinito.

- Non è possibile rappresentare tutti i possibili valori di un certo intervallo

Rappresentazione in **virgola fissa** (poco usata):

- Si rappresentano separatamente, usando un numero fissato di cifre, la **parte intera** e la **parte frazionaria**
- Ad esempio:
  - ▶ usando 8 bit, di cui 4 per la parte intera e 4 per la parte frazionaria
  - ▶ 5.75 in binario diventa 0101.1100
  - ▶ Per la parte frazionaria:  $1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} = 0.75$

## Rappresentazione di numeri reali (2)

Rappresentazione in **virgola mobile**:

- Per rappresentare un numero reale  $N$  usa **notazione esponenziale**:

$$N = m \cdot 2^e$$

dove:

- ▶  $m$  è la **mantissa** in base 2: numero frazionario tale che

$$0.5 \leq |m| < 1$$

- ▶  $e$  è l'**esponente** in base 2: numero intero
- Esempio: per rappresentare 0.125 avremo
  - ▶  $m = 0.5$  (in binario 0.1)
  - ▶  $e = -2$  (in binario con 4 bit 1110)
  - ▶ infatti:  $0.5 + 2^{-2} = 0.125$

## Rappresentazione di numeri reali (3)

I numeri rappresentabili in virgola mobile:

- Sono distribuiti simmetricamente rispetto allo 0
- **NON** sono uniformemente distribuiti sull'asse reale
  - ▶ sono più densi intorno allo 0!
- Ad esempio, con 3 bit per la mantissa e 3 bit per l'esponente:

m\e	-4	-3	-2	-1	0	1	2	3
<b>0.100</b>	0.03125	0.0625	0.125	0.25	0.5	1	2	4
<b>0.101</b>	0.0390625	0.078125	0.15626	0.3125	0.625	1.25	2.5	5
<b>0.110</b>	0.046875	0.09375	0.1875	0.375	0.75	1.5	3	6
<b>0.111</b>	0.0546875	0.109375	0.21875	0.4375	0.875	1.75	3.5	7

Graficamente (i punti sono numeri rappresentabili):



# Rappresentazione di numeri reali (4)

**Problemi** della rappresentazione in virgola mobile:

- Molti razionali non sono rappresentabili (ad esempio  $1/3$ )
- Non è chiuso rispetto ad addizioni e moltiplicazioni
- Per rappresentare un reale  $N$  si sceglie l'elemento rappresentabile **più vicino** ad  $N$
- Perdita di precisione:
  - ▶ **Arrotondamento**: mantissa non sufficiente a rappresentare tutte le cifre significative del numero
  - ▶ **Errore di overflow**: esponente non sufficiente (numero troppo grande)
  - ▶ **Errore di underflow**: numero troppo piccolo (rappresentato come 0)



# Rappresentazione di numeri reali (5)

Esempi reali di rappresentazione in virgola mobile:

- Standard IEEE 754 a 32 bit (precisione singola– **float**):
  - ▶ 1 bit per il segno della mantissa
  - ▶ 8 bit per l'esponente
  - ▶ 23 bit per la mantissa
  - ▶ alcune configurazioni riservate (0, Not-a-number, Infinito)
- Standard IEEE 754 a 64 bit (precisione doppia– **double**):
  - ▶ 1 bit per il segno della mantissa
  - ▶ 11 bit per l'esponente
  - ▶ 52 bit per la mantissa
  - ▶ alcune configurazioni riservate (0, Not-a-number, Infinito)

# Rappresentazione di caratteri e testi (1)

I singoli caratteri di un testo possono essere rappresentati come numeri.

La seguente tabella riporta la codifica **ASCII** (o meglio, US-ASCII) è nata negli anni 60 per rappresentare (con 7 bit) i simboli della tastiera americana

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

## Rappresentazione di caratteri e testi (2)

La codifica ASCII è stata superata negli anni da codifiche più ricche.

In primis, la codifica Extended-ASCII (8 bit)

Attualmente, una codifica che si sta affermando (specialmente in ambito web) è **UTF-8**

- usa da 1 a 4 byte (variabile) per rappresentare un carattere

Le **stringhe** sono sequenze di caratteri che possono contenere un testo

- possono essere rappresentate nella memoria di un computer come una sequenza di caratteri terminata dal carattere NUL (= numero 0)

## Rappresentazione di immagini e altri dati

Una immagine può essere rappresentata nella memoria di un computer codificando **il colore dei singoli pixel** che la compongono tramite valori numerici

Ad esempio, nel formato bmp a 24 bit ogni pixel dell'immagine viene rappresentato da 3 byte

- ogni byte (256 valori) rappresenta il livello di un colore fondamentale RGB
- 111111110000000000000000 = (255,0,0) = rosso
- 000000001111111100000000 = (0,255,0) = verde
- 000000000000000011111111 = (0,0,255) = blu
- 111111111111111111111111 = (255,255,255) = bianco
- 111111111001100110011001 = (255,204,204) = rosa

Altri formati più sofisticati (gif, png, jpeg) utilizzano metodi di **compressione** per ridurre la dimensione della rappresentazione

Approcci simili sono adottati per rappresentare suoni e filmati...

# Sommario

- 1 Come è fatto un computer
- 2 Rappresentazione binaria dell'informazione
- 3 Linguaggi di programmazione e algoritmi**

# Problemi computazionali

L'Informatica è una scienza che studia principalmente metodi e strumenti per la risoluzione di problemi computazionali (**problem solving**)

Un **problema computazionale** è un problema che richiede

- di calcolare un risultato (**output**)
- a partire da determinati valori noti (**input**)

**Esempi** di problemi computazionali:

- Calcolo del massimo comune divisore di due numeri
- Preparazione di un risotto ai funghi
- Tessitura di tappeto con un telaio meccanico
- Ordinamento di una sequenza di numeri
- Ricerca di una parola in un testo
- Risoluzione di un Sudoku

# Algoritmi (1)

I problemi computazionali possono essere risolti tramite algoritmi.

Un **algoritmo** è una sequenza finita di passi di elaborazione che, dato un input, consentono di ottenere l'output ad esso corrispondente.

## Algoritmi (2)

Un algoritmo si può esprimere in molti modi diversi:

- In linguaggio naturale:
  - ▶ “Per calcolare il massimo comune divisore di  $X$  e  $Y$  maggiori di 0 bisogna ripetutamente sottrarre il più piccolo dei due dal più grande. Quando i due numeri saranno diventati uguali tra loro, il loro valore corrisponderà al risultato.”

- Tramite formule matematiche:

$$\text{▶ } mcd(X, Y) = \begin{cases} mcd(X - Y, Y) & \text{se } X > Y \\ mcd(X, Y - X) & \text{se } Y > X \\ X & \text{altrimenti} \end{cases}$$

- In **pseudo-codice**:

```
finchè X!=Y ripeti {  
  se Y>X scambia(X,Y)  
  X = X-Y  
}  
stampa X
```



# Algoritmi (3)

Un buon algoritmo deve soddisfare alcune proprietà:

- **Non ambiguità**: I singoli passi devono essere “elementari”: facilmente eseguibili (atomici) e non ambigui
- **Determinismo**: Eseguito più volte sullo stesso input, l'algoritmo deve eseguire sempre la stessa sequenza di passi (e conseguentemente deve dare lo stesso risultato)
- **Terminazione**: L'esecuzione dell'algoritmo deve prima o poi terminare e fornire un risultato

# Esempi di Algoritmi (1)

Supponiamo di voler comprare un'auto il più possibile economica:

Ci vengono proposte due alternative:

	
10000 euro	15000 euro
8 km/l	20 km/l

Quale scegliamo?

## Esempi di Algoritmi (2)

Ci serve qualche informazione in più:

- Vogliamo percorrere circa 50000 km
- Il prezzo della benzina è circa 2 euro al litro

Ora possiamo usare il seguente algoritmo (in pseudo-codice):

```
per ogni auto calcola {
    costo_gas = (km_percorsi/km_al_litro) x prezzo_benzina
    costo_tot = prezzo_acquisto + costo_gas
}
se costo_tot(auto1) < costo_tot(auto2)
    compra auto1
altrimenti
    compra auto2
```

Questo algoritmo è non ambiguo (i passi sono semplici), termina (il ciclo “per ogni” prima o poi finisce) ed è deterministico

Qual è il risultato?

## Esempi di Algoritmi (3)

Vogliamo scrivere un algoritmo che possa essere usato da un braccio meccanico per ordinare i mattoncini di Lego mettendo i pezzi blu a sinistra e pezzi gialli a destra.



## Esempi di Algoritmi (4)

Primo Tentativo (in pseudo-codice):

```
solleva tutti i mattoncini  
posali tutti in ordine giusto
```

- Questo algoritmo è **ambiguo** in quanto i singoli passi non sono atomici (elementari)
  - ▶ il braccio meccanico può sollevare un mattoncino per volta

Secondo Tentativo (in pseudo-codice):

```
finche i mattoncini non sono in ordine giusto {  
  scambia un mattoncino blu e uno giallo  
}
```

- Questo algoritmo **non è deterministico**: ad ogni passo ho più scelte di mattoncini da scambiare
- Inoltre **potrebbe non terminare**

## Esempi di Algoritmi (5)

Terzo Tentativo (in pseudo-codice):

```
finche i mattoncini non sono in ordine giusto {  
    indiviuda la coppia di mattoncini giallo-blu piu a sinistra  
    scambia i due mattoncini  
}
```

Questo algoritmo:

- **non è ambiguo**: in quanto i singoli passi sono semplici
- è **deterministico**: la scelta della coppia di mattoncini da scambiare è precisamente ben determinata (la più a sinistra)
- ed è facile convincersi che **termini**, perchè ad ogni iterazione un mattoncino blu si sposta a sinistra e uno giallo verso destra

Questo è un **buon algoritmo**!

## Esempi di Algoritmi (6)

Prova di esecuzione:

```
finche i mattoncini non sono in ordine giusto {  
  indiviuda la coppia di mattoncini giallo-blu piu a sinistra  
  scambia i due mattoncini  
}
```



Input:



Passo 1:



Passo 2:



Passo 3: come continua?

# Algoritmi: calcolabilità

Ora sappiamo che si possono risolvere problemi computazionali tramite algoritmi

In realtà non tutti i problemi computazionali possono essere risolti

- Alcuni problemi sono **indecidibili** ( o **non calcolabili**)
- Ad esempio: non è possibile scrivere un algoritmo che prenda in input la descrizione di un qualunque altro algoritmo A e dia come output “SI” o “NO” a seconda che l’algoritmo A termini o meno (**problema della fermata**)



# Algoritmi: complessità computazionale

Inoltre, tra i problemi calcolabili si distinguono:

- Problemi **trattabili**: che possono essere risolti da un algoritmo in tempi “ragionevoli”
  - ▶ Esempio: ordinare una sequenza di  $n$  numeri naturali si può fare con un algoritmo che esegue circa  $n^2$  passi (o anche meno...)
- Problemi **intrattabili**: i cui algoritmi di soluzione richiedono tempi di “non ragionevoli”
  - ▶ Esempio: risolvere un sudoku con una griglia di  $n \times n$  caselle può richiedere un algoritmo che esegue circa  $2^n$  passi<sup>1</sup>
  - ▶ Quando  $n$  diventa grande (circa 45), il tempo di esecuzione dell'algoritmo su un computer può diventare di **milioni di anni**...

Per approfondimenti:

- Capitolo 1 di: Crescenzi, Gambosi, Grossi. **Strutture dati e algoritmi. Progettazione, analisi e visualizzazione.**  
Pearson–Addison Wesley

---

<sup>1</sup>questa è una semplificazione... il discorso sarebbe molto più complicato. >

# Dagli algoritmi ai programmi

Ora: come dare in pasto un algoritmo al computer?

Il computer parla il linguaggio macchina... è piuttosto difficile esprimere un algoritmo in linguaggio macchina!

Ecco: i **linguaggi di programmazione!**

# I linguaggi di programmazione

Un **linguaggio di programmazione** è un “linguaggio formale” che consente scrivere programmi che realizzano algoritmi

- Un **linguaggio formale** (a differenza del linguaggio naturale) è un linguaggio con regole sintattiche e semantiche ben precise che rendono i costrutti del linguaggio stesso **privi di ambiguità**
- I programmi potranno poi essere **tradotti** in linguaggio macchina per essere eseguiti dal computer
- Essendo il linguaggio specificato su regole ben precise, la traduzione può essere fatta da un altro programma!

Terminologia:

- I linguaggi di programmazione sono detti **linguaggi di alto livello**
  - ▶ **astraggono** dai dettagli di funzionamento dell'elaboratore
- Il linguaggio macchina e l'assembly sono **linguaggi di basso livello**

# Compilazione e interpretazione (1)

Come tradurre un programma dal linguaggio di programmazione in linguaggio macchina?



## Compilazione e interpretazione (2)

Pensiamo al mondo reale: come si può comunicare con una persona che parla solo cinese?



Due soluzioni:

scriviamo una lettera in italiano e la portiamo all'ambasciata cinese per farla tradurre



troviamo un interprete che faccia una traduzione simultanea



Image courtesy of [stockimages](#) / [FreeDigitalPhotos.net](#)

## Compilazione e interpretazione (3)

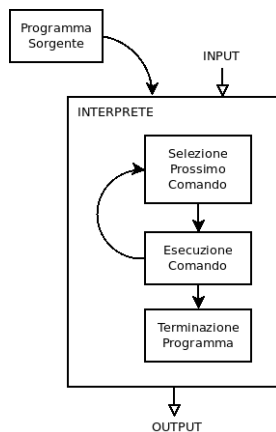
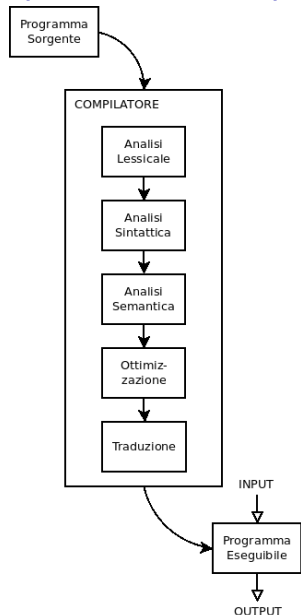
Analogamente, per tradurre un programma (**sorgente**) in linguaggio macchina si può usare:

- Un **compilatore**: ossia un programma che prende in input il programma sorgente e produce in output il corrispondente programma in linguaggio macchina, **eseguibile successivamente** dal computer
- Un **interprete**: ossia un programma che prende in input il programma sorgente, traduce un comando per volta e **lo esegue** man mano.

Esempi:

- Il linguaggio C è un linguaggio compilato
- Il linguaggio JavaScript è un linguaggio interpretato

# Compilazione e interpretazione (4)



## Compilazione e interpretazione (5)

Vantaggi  $\uparrow$  e svantaggi  $\downarrow$  della **compilazione**:

- $\downarrow$  Compilare un programma può richiedere molto tempo
- $\uparrow$  Il programma eseguibile che si ottiene è veloce
- $\uparrow$  Il programma eseguibile può essere eseguito più volte senza ricompilare
- $\uparrow$  Il compilatore può controllare e ottimizzare il programma prima che venga eseguito
- $\downarrow$  Gli errori che sfuggono al controllo del compilatore non potranno più essere gestiti durante l'esecuzione
- $\downarrow$  Un compilatore produce un eseguibile che funziona su una sola architettura (Intel, ARM, ...) e sistema operativo (Windows, Linux, MacOS, ...)



## Compilazione e interpretazione (6)

Vantaggi  $\uparrow$  e svantaggi  $\downarrow$  della **interpretazione**:

- $\downarrow$  L'esecuzione del programma è rallentata dall'interprete, che deve tradurre ogni comando
- $\uparrow$  Non è necessario ri-compilare tutto il programma ogni volta che si fa una modifica
- $\uparrow$  Portabilità: lo stesso identico programma può essere eseguito su architetture (Intel, ARM,...) e sistemi operativi (Windows, Linux, MacOS, ...) diversi
- $\downarrow$  Nessun controllo sui programmi prima di iniziarne l'esecuzione
- $\uparrow$  Gli errori che si incontrano a tempo di esecuzione possono essere gestiti meglio

## Compilazione e interpretazione (6)

Alcuni linguaggi combinano compilazione e interpretazione

- Java, ad esempio, è uno di questi (vedremo...)