La "macchina" da calcolo

- Abbiamo detto che gli algoritmi devono essere scritti in un linguaggio "comprensibile all'esecutore"
- Se il nostro esecutore è il "calcolatore", questo che linguaggio capisce?
  - che linguaggio usa per "rappresentare l'informazione"?
  - alla base, dicevamo che l'informatica ha a che fare con tutti i problemi di *rappresentazione* ed *elaborazione* dell'informazione...
- Alla sua base, un calcolatore capisce solamente un linguaggio estremamente scarno, che rappresenta tutto come (sequenze di) ***bit***
  - un bit è l'informazione più elementare, quella che può assumere solo *2 valori*: {0,1} o {Vero, Falso}, …
    - 2 valori ===> è una informazione *binaria*
- Tutta l'informazione *discreta* può essere scomposta (ossia rappresentata) come una opportuna sequenza di 0 e 1
  - l'informazione continua può essere *approssimata* in questa maniera
- Siccome noi usiamo per lo più *sequenze* di bit per rappresentare dell'informazione, spesso useremo come "mattoncino", più che il bit, il ***byte***
  - un byte è una *sequenza di 8 bit*:
    00000000
    00000001
    00000010
    00000011
    ...
    11111111

Esempi di rappresentazione di informazione

- Un byte può rappresentare i numeri naturali da 0 a 255 (= 2^8-1):
  - zero = 00000000 (= $0\cdot2^7+0\cdot2^6+0\cdot2^5+0\cdot2^4+0\cdot2^3+0\cdot2^2+0\cdot2^1+0\cdot2^0$)
  - 8 = 00001000 (= $0\cdot2^7+0\cdot2^6+0\cdot2^5+0\cdot2^4+1\cdot2^3+0\cdot2^2+0\cdot2^1+0\cdot2^0$)
  - ...
  - 255 = 11111111 (= $1\cdot2^7+1\cdot2^6+1\cdot2^5+1\cdot2^4+1\cdot2^3+1\cdot2^2+1\cdot2^1+1\cdot2^0$)
- Oppure con un byte si possono rappresentare i numeri interi compresi fra –127 e 127, ossia fra $-(2^{(8-1)}-1)$ e $(2^{(8-1)}-1)$
  - se il primo bit è 0 ==> numero positivo,
    se il primo bit è 1 ==> numero negativo
    - per esempio 7 = 00000111, -7 = 10000111
    - zero è rappresentato in 2 maniere!
      0 = 00000000 ed anche 0 = 10000000
  - sono possibili rappresentazioni diverse; ne vedremo una un po' più efficace più avanti
- Posso anche usare i byte per rappresentare i numeri reali: li rappresento come numeri razionali contenenti una parte intera e una frazionaria che *approssimano* il numero reale con *precisione arbitraria*
  - notazione in *virgola fissa*: si codificano separatamente la parte intera e la parte frazionaria
    - esempio di codifica del numero 8.345:
      primo byte ==> 00001000
      - è la rappresentazione delle parte intera, cioè 8
    - secondo byte ==> 01011000
      - è la rappresentazione (approssimata) della parte frazionaria, cioè 0.345:
        $0\cdot2^{-1}+1\cdot2^{-2}+0\cdot2^{-3}+1\cdot2^{-4}+1\cdot2^{-5}+0\cdot2^{-6}+0\cdot2^{-7}+0\cdot2^{-8}$ = 0.34375

Esempi di rappresentazione di informazione (2)

- Si possono usare i byte anche per codificare i *caratteri*
- La codifica classica di caratteri in byte è la codifica ASCII (American Standard Code for Information Interchange):
  - sette bit vengono usati per rappresentare 128 caratteri (l'ottavo bit è usato per controllo, e spesso ignorato)
- Nella codifica ASCII, a ogni lettera (le maiuscole da A a Z, le minuscole da a a z), cifra (da 0 a 9) o separatore (usato per la punteggiatura o come operatore aritmetico) viene assegnato un numero naturale rappresentabile in forma binaria
  - ad esempio la lettera 'A' viene codificata in ASCII come numero 65 e la sua forma binaria è 01000001
  - il separatore ';' viene codificato come 59, e la sua forma binaria è 00111011
  - ...

- *NB: la stessa stringa di bit ha diversi significati, a seconda del tipo di informazione rappresentata!*

Operazioni elementari sui bit

- Inversione di un bit:
  0 —> 1
  1 —> 0
- "Somma" di due bit:
  0+0 = 0
  1+0 = 1
  0+1 = 1
  1+1 = 1
- "Prodotto" di due bit:
  0*0 = 0
  1*0 = 0
  0*1 = 0
  1*1 = 1
- Se interpretiamo 0 come *Falso* (False, F) e 1 come *Vero* (True, T), le operazioni di cui sopra possono essere viste come operatori logici:
  - inversione = negazione, (NOT)
  - somma = somma logica, (OR)
  - prodotto = prodotto logico, (AND)
- Mediante le suddette operazioni si possono quindi elaborare sequenze di bit in modo arbitrario ===> queste sono le operazioni di base per costruire algoritmi
- Queste operazioni non solo hanno un'intuitiva interpretazione logica, sono anche facilmente realizzabili mediante dispositivi elettronici (a loro volta elementari e combinabili in circuiti integrati)

Approfondimento: l'aritmetica binaria

- Il fatto che l'informazione base sia il bit porta a codificare i numeri come sequenze di bit
- Di conseguenza, i calcolatori adottano la numerazione in base 2:
  0 = 000; 1 = 001; 2 = 010; 3 = 011; ….
- Conversione da base 10 a base 2:
  - continuo a dividere il valore in base 10 per 2, fino a che non ottengo 0; i resti (presi in ordine inverso rispetto a quello in cui li ho ottenuti) danno il valore in base 2
  - Esempio di conversione da base 10 a base 2:
    13_2 : 13/2 = 6; resto = 1 1
    6/2 = 3; resto = 0 0
    3/2 = 1; resto = 1 1
    1/2 = 0; resto = 1 1
    13_2 ⟶ 1101
- Da base 2 a base 10:
  - basta moltiplicare le varie cifre binarie per la potenza apposita:
    $(1101)_{10} = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 0 + 1 = 13$
- *Esercizio per casa:* codificare (per esempio con diagrammi di flusso) gli algoritmi di conversione da base 10 a base 2 e viceversa
- Somma cifra per cifra: eseguirla in base 2 è realtà la stessa cosa che eseguirla in base 10, occorre solo ricordarsi che, in base 2, 1+1 fa 0 con riporto di 1:

riporto	11100		001100000000100
	8731		10001000011011
	5698		010110010000010
risultato	14429		11100001011101

I numeri negativi

- Abbiamo già visto una rappresentazione con un byte degli interi (anche negativi) compresi fra –127 e 127, ossia fra $-(2^{(8-1)}-1)$ e $(2^{(8-1)}-1)$
  - questa è la cosiddetta rappresentazione in *modulo e segno*:
    primo bit = 0 ==> numero positivo
    primo bit = 1 ==> numero negativo
    - 0 = 00000000 e 0 = 10000000
- Problemi di questa rappresentazione:
  - è "sprecona", in quanto usa due rappresentazioni diverse per un solo numero
  - la realizzazione delle varie operazioni (e.g. differenza) con questa rappresentazione richiede algoritmi nuovi e non efficienti
- Una tecnica di rappresentazione più efficace: la rappresentazione in ***complemento a due***:
  - se uso *m* bit, -*N* è rappresentato come 2^m-N
    - per esempio, se uso 3 bit (*m* = 3):
      -4 ⟶ 100
      - -4 non era rappresentabile con la rappresentazione in modulo e segno!

      -3 ⟶ 101
      -2 ⟶ 110
      -1 ⟶ 111
      0 ⟶ 000
      1 ⟶ 001
      2 ⟶ 010
      3 ⟶ 011
- Algoritmo per passare da N a -N in complemento a 2:
  1. inverto le cifre una ad una (gli '1' diventano '0', gli '0' diventano '1')
  2. sommo 1 alla cifra così ottenuta
     - passo da 3 a -3: 011 ⟶ 100 ⟶ 101
     - passo da 2 a -2: 010 ⟶ 101 ⟶ 110

Somma e sottrazione in complemento a 2

- Se uso la rappresentazione in complemento a 2, per eseguire la sottrazione tra 2 numeri M ed N mi basta fare
  M + (-N)
- Esempi:

+ 5	0000101		+ 5	0000101
+ 8	0001000		- 8	1111000
+ 13	0001101		- 3	1111101
- 5	1111011		- 64	1000000
+ 8	0001000		- 8	1111000
+ 3	(1) 0000011		- 72	[1] (1) 0111000

  - attenzione al fenomeno di "overflow"