1 Rappresentazione dell'informazione e aritmetica dei calcolatori

Per poter parlare di computer dobbiamo per prima cosa parlare di numeri.
Questo perché ogni computer è un sistema numerico e discreto, che lavora solo con numeri che hanno un numero finito di cifre. Un computer è perciò un sistema "numerico" o "digitale".
Qualunque forma abbiano gli ingressi che proponiamo ad un computer, essi devono essere in qualche modo trasformati in numeri per poter essere elaborati. Analogamente, anche se otteniamo i risultati sotto qualsiasi forma, il "cuore" del computer li ha prodotti come numeri, che sono stati successivamente trasformati.

1.1 Rappresentazioni dei numeri

Parlando di numeri dobbiamo anzitutto distinguere fra "l'essenza" di un numero e la sua "scrittura". Un numero è un'entità matematica astratta, che ha la sua origine nell'operazione del contare e che esiste indipendentemente da come noi lo vediamo o lo scriviamo.
Un numero può essere "rappresentato" mostrando le dita di una mano, pronunciandone il "nome" in una lingua parlata, scrivendolo sulla carta secondo una ben precisa notazione, quale quella romana, araba, occidentale moderna, o anche scrivendolo nelle memorie elettroniche di un computer. Qualunque sia la rappresentazione che diamo di un numero, il numero in sé rimarrà sempre lo stesso.

!!!!TODO!!!

Figura 1: rappresentazioni di un numero

Ogni diversa rappresentazione dei numeri ha i suoi propri procedimenti per i calcoli aritmetici, cioè per fare la somma, la divisione e le altre operazioni. Chiamiamo "**algoritmi**" questi procedimenti, avvertendo del fatto che in seguito vedremo una definizione rigorosa e più estensiva del concetto.
Il più grande vantaggio della notazione occidentale moderna, comune anche a quelle indiana e araba che l'hanno ispirata, è il fatto che gli algoritmi che si devono usare per eseguire i calcoli aritmetici sono molto semplici, come abbiamo imparato nella scuola elementare.
La notazione occidentale moderna fa uso di dieci cifre, non a caso tante quante le dita delle mani, e per questo viene detta "notazione decimale".
Ogni cifra ha un suo valore, fra zero e nove, un numero è composto da una o più cifre ed in esso una cifra è più o meno "importante" delle altre che sono nello stesso numero, in funzione della posizione occupata.
Le cifre che stanno a sinistra sono più importanti di quelle che stanno a destra. La nostra rappresentazione dei numeri è perciò di tipo "posizionale".

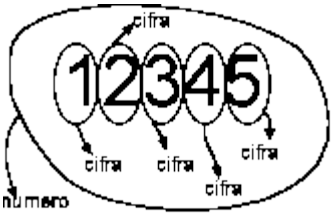


Figura 2: cifre e numeri

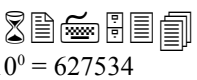
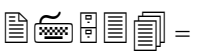
Notazione posizionale
Una notazione **posizionale** è una rappresentazione dei numeri che dà alle cifre in essi contenute diversa importanza in dipendenza dal posto che occupano.
Prendiamo un numero ad esempio e discutiamo come viene scritto, poi ne trarremo delle indicazioni generali. Supponiamo di scegliere il numero 22. Avendo tutti fatto le elementari, sappiamo che il due più a sinistra non è uguale a quello di destra ma è dieci volte più "pesante", possiamo cioè scrivere questa uguaglianza: 22 = 20 + 2, che ancora non ci dice molto. Riscriviamo il tutto in questo modo: 22 = 2 * 10 + 2 * 1, lo scopo di questa complicazione verrà più chiaro fra un momento.
Prendiamo un altro numero, un po' più complicato: 12345. Innanzitutto diciamo che l'uno di questo numero viene detto "**cifra più significativa**" (most significant digit, o leftmost digit), mentre il cinque è la "**cifra meno significativa**" (least significant digit o rightmost digit).
Poi scriviamo, analogamente a prima: 12345 = 1 * 10000 + 2 * 1000 + 3 * 100 + 4 * 10 + 5 * 1. Notiamo che il valore della cifra viene moltiplicato per un fattore, che possiamo chiamare "peso", che, come ci aspettavamo, è tanto più grande quanto più quella cifra sta a sinistra nel numero.
Ma "di quanto" il peso è più grande? Se proviamo a riscrivere la stessa espressione in un altro modo la cosa dovrebbe risultare chiara:
$12345 = 1 * 10^4 + 2 * 10^3 + 3 * 10^2 + 4 * 10^1 + 5 * 10^0$
Scrivendo il numero in questo modo diventa chiaro che il peso di ogni cifra è una potenza di dieci che dipende dal posto della cifra all'interno del numero. Se la cifra è alla destra di tutti quella potenza è zero, ed aumenta di uno ogni volta che

si va di una cifra più a sinistra (non ci dobbiamo stupire, si tratta solo di un modo diverso di scrivere il concetto di unità, decine, centinaia, migliaia, decine di migliaia,..).

A questo punto ci vogliamo staccare dai nostri simboli. Se supponiamo che gli alieni sul pianeta Ork abbiano dieci dita come noi, e che rappresentino le cifre come in questa tabella:

Parola su Terra(Italia)	Parola su Terra(G.B.)	Cifra su Terra (Italia)	Cifra su Ork (Gluk)
Zero	Zero	0	
Uno	One	1	
Due	Two	2	
Tre	Three	3	
Quattro	Four	4	
Cinque	Five	5	
Sei	Six	6	
Sette	Seven	7	
Otto	Eight	8	
Nove	Nine	9	

Tabella 1: cifre su pianeti diversi

Allora il numero "gluckiano"  sarebbe, se scritto in Italia:  = $6 * 10^5 + 2 * 10^4 + 7 * 10^3 + 5 * 10^2 + 3 * 10^1 + 4 * 10^0 = 627534$

Se gli uomini non avessero avuto dieci dita, avrebbero potuto inventare lo stesso una notazione posizionale per i numeri? La risposta è senz'altro sì (tan'è vero che lo hanno fatto effettivamente), ma probabilmente l'avrebbero inventata con una "base" diversa. La **base** di una notazione posizionale è il numero di simboli diversi che essa comprende. Gli uomini avevano a disposizione facilmente dieci simboli diversi, che erano le dita. Tutti abbiamo cominciato a contare con le dita (io lo faccio tuttora) ed esistono e sono ancora usati in Cina algoritmi di calcolo che permettono di effettuare le operazioni sulle dita, memorizzando i risultati intermedi con un complicato gioco di falangi.
Al giorno d'oggi gli uomini usano numeri in base 10, in base 2, in base 16 e, più raramente, in base 8. In passato si erano usate basi diverse, i babilonesi, per esempio, avevano una notazione posizionale, a base 60, dalla quale origina il nostro modo di misurare il tempo e gli angoli.
I numeri in base 10 vengono detti numeri "**decimali**".
Per distinguere i numeri in una certa base da numeri in un'altra base in questo capitolo scriveremo la base a pedice.
Per esempio $82_{10} = 1010010_2 = 122_8 = 52_{16}$. Il significato degli uguali in quest'ultima espressione dovrebbe divenire chiaro fra poco. Se non si mette la base a pedice di solito si considera che il numero sia decimale.

Facciamo notare fin da ora che i linguaggi di programmazione hanno convenzioni diverse da quella appena illustrata per indicare in quale base sono rappresentati i numeri. Per acquisire consuetudine spieghiamo sin da subito qual è la notazione usata nel linguaggio Assembly. Un numero si può riconoscere perché inizia con una cifra decimale. La base utilizzata si riconosce perché alla fine del numero, senza spazi in mezzo, c'è una lettera che la indica. Se il numero termina con la lettera b è a base 2 ("**b**inario", p.es. 100101b), se termina con h è a base 16 ("**h**exadecimal", p.es. 0F104h), se termina con d, oppure se alla fine del numero non c'è nessuna lettera, il numero è considerato decimale (p.es. 28, oppure 28d). In linguaggio C, e nei suoi molti derivati, i numeri decimali sono scritti normalmente, mentre quelli esadecimali sono preceduti dai caratteri 0x (p.es. 0x3FA è il numero FA_{16}). In BASIC i numeri esadecimali sono preceduti dai caratteri &H e terminati con una & (p.es. &H3FA&).

Rappresentazione binaria (base 2)

Per ragioni pratiche ed economiche i computer sono in grado di rappresentare i numeri solo con due simboli (si potrebbe dire che hanno solo due dita ..). Ciò avviene perché i computer possono elaborare i numeri facendo uso di dispositivi elettronici detti "transistor", che si possono facilmente far commutare fra due stati: conduzione o non conduzione ("acceso" o "spento"). Per quanto sia possibile mantenere i transistor in più di due stati, usarne solo due è molto più affidabile e semplice, perciò più economico. E' con transistor che si realizzano tutti i computer e le loro memorie. Ciò significa che i mattoni fondamentali con i quali è costruito un computer funzionano in modo "o tutto o niente".

Per rappresentare i numeri un computer deve perciò usare una notazione in base due.

I numeri scritti in base due vengono anche detti numeri "**binari**".

Una cifra in un numero binario viene detta **bit**, che è l'unione delle due parole "**Bi**nary digi**t**" (cifra binaria), ma che in Inglese significa anche "un pochino". E' un buon nome, dato che il bit è la minima quantità di informazione immaginabile, tant'è vero che viene usato come anche unità di misura dell'informazione.
I simboli usati per scrivere un numero binario possono essere solo due; si potrebbe usare "Spento" o "Off" per lo zero, "Acceso" o "On" per l'uno, ma la maggior parte delle volte si prende come simbolo per lo zero il segno "0" e come simbolo per l'uno il segno "1".
Se usiamo quest'ultima convenzione il numero 101101 potrebbe essere un numero binario. Per distinguerlo da un numero decimale, come abbiamo già visto, ci dobbiamo mettere un 2 a pedice, così: 101101_2.
Se abbiamo solo due simboli come possiamo rappresentare tutti i numeri? Proviamo a contare e vediamo di inventare un modo. Per i numeri zero e uno abbiamo già i simboli, quindi zero = 0 e uno = 1. Il primo problema sorge quando dobbiamo rappresentare il due: non abbiamo più simboli.
Allora facciamo in base due la stessa cosa che faremmo in base 10, cioè mettiamo uno zero nella cifra meno significativa e "riportiamo" un uno nella cifra accanto. Quindi , se in decimale da "9" si passa a "10", in binario da "1" si passa a "10_2". Perciò il numero binario due è: due = $2_{10} = 10_2$. La conseguenza è che $1_2 + 1_2 = 10_2$.
Questa somma non è per nulla strana perché si deve leggere: uno più uno uguale "unozero" oppure uguale due (unozero è pur sempre il numero due, anche se scritto in modo diverso!).
Il numero successivo è tre e deve essere due più uno. Dobbiamo semplicemente aggiungere 1 alla cifra meno significativa di 10_2, perciò otteniamo 11_2. Tre = $3 = 11_2$.
A questo punto se vogliamo aggiungere 1 non abbiamo più simboli, dobbiamo "riportare", proviamo a mettere in colonna:

Rip. 1	Rip. 1		
	1	1	+
		1	=
1	0	0	

Vediamo che nella seconda colonna c'è uno più uno del riporto per cui, dato che $1_2 + 1_2 = 10_2$, il riporto si propaga alla terza colonna da destra e ci costringe ad aggiungere un nuovo bit a sinistra del numero. Perciò $4 = 100_2$.
Continuando ad aggiungere uno si ottengono i numeri binari riportati nella relativa colonna nella Tabella 4, sulla quale consiglio di spendere ora un po' di tempo a pensare.
Continuando il conteggio il "riporto" deve avvenire ogni 2 numeri e si propaga fino in fondo, costringendo ad aggiungere un bit ogni volta che il numero raddoppia (vedere p.es. la tabella in corrispondenza ai numeri decimali 2, 4, 8, 16).
Osservando i valori binari nelle righe appena citate vediamo che un uno nella cifra meno significativa vale 1, mentre nella seconda cifra meno significativa vale 2, nella terza quattro, nella quarta otto, e così via. La posizione di ogni bit "pesa" dunque la potenza di due relativa, cominciando a contare dalla potenza zero ed aumentando di uno ogni volta andando verso sinistra. Allora si può scrivere una formula analoga a quella scritta per i numeri decimali, nella quale appare la base due al posto di dieci, in tutti i punti ove si eleva a potenza. Usando come esempio il numero 101010_2 si ha:

$101010_2 = 0 * 2^0 + 1 * 2^1 + 0 * 2^2 + 1 * 2^3 + 0 * 2^4 + 1 * 2^5 = 2_{10} + 8_{10} + 32_{10} = 42_{10}$
Un numero di 6 bit "tutti uno" sarebbe invece così:

$111111_2 = 1 * 2^0 + 1 * 2^1 + 1 * 2^2 + 1 * 2^3 + 1 * 2^4 + 1 * 2^5 = 1 + 2 + 4 + 8 + 16 + 32 = 63_{10}$

Si noti come il numero fatto con "tutti uno" sia il più grande che si può avere con un certo numero di bit; l'aggiunta di un'ulteriore unità ad un siffatto numero dà una potenza di due, che ha un bit in più del numero precedente ed è composto di un 1 come cifra più significativa e di tutti zero alla sua destra.
Segue una tabella dei pesi dei bit in un numero binario da 17 cifre (bit da 0 a 16), che è conveniente imparare "a memoria" dal bit meno significativo almeno fino al nono bit (bit di "esponente" 8):

n. del bit	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Peso	65536	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

Tabella 2: potenze di due

Generalizzando a tutte le basi ed usando una notazione che piace tanto ai matematici:

$$Numero = \sum_{0}^{nCifreDelNumero-1} Cifra_i * Base^i = Cifra_0 * Base^0 + Cifra_1 * Base^1 + Cifra_2 * Base^3 + Cifra_4 * Base^4 + ..$$

ove la grande lettera greca Sigma significa "sommatoria", cioè la somma di tutti gli elementi indicati, nCifreDelNumero è il numero di cifre di cui il numero è composto.

Rappresentazione ottale (base 8)
Sarebbe superfluo dirlo, ma i numeri ottali hanno otto simboli.
Per comodità i simboli che si usano per scrivere un numero ottale sono lo zero e le prime 7 cifre "normali". Naturalmente il numero in base otto andrà "a riportare" la prima volta quando si giunge al numero otto che sarà quindi 10: $8_{10} = 10_8$.
Dopo la prima volta, ci sarà un riporto ad ogni multiplo di 8, come si può vedere nella Tabella 4.
Oggi è abbastanza raro utilizzare numeri in base 8.
La base otto è stata usata in passato, quando esistevano computer che avevano registri a parallelismo 12, per la facilità di conversione con i numeri binari, che si potevano dividere in gruppi di 3 bit.

Rappresentazione esadecimale (base 16)
La notazione binaria ha lo svantaggio di richiedere molte cifre, anche per numeri non eccessivamente grandi. Per esempio, per rappresentare il numero 32770 ci vogliono 16 bit. Se si usa la base 16 (numeri esadecimali) il numero che si scrive ha quattro volte meno cifre del numero binario corrispondente. Dato che la conversione fra numeri binari ed esadecimali è facilissima e si fa a mente, spesso, quando è scomodo scrivere tutti i bit di un numero binario grande, si usa al suo posto la sua notazione esadecimale.
Siccome nella base esadecimale dobbiamo avere 16 simboli, per i primi 10 siamo già a posto: useremo le normali cifre numeriche da 0 a 9. Per gli altri 6 simboli la convenzione usata universalmente è di utilizzare le lettere da A ad F. In questo caso questi simboli alfabetici vengono ad assumere il significato di cifre numeriche.
Il valore delle cifre in notazione esadecimale è quindi quello della seguente tabella:

Simbolo	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Valore della cifra	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Tabella 3: cifre esadecimali

Segue una tabella che mostra come si susseguono i numeri nelle diverse basi. In essa si possono vedere le analogie e le differenze fra i vari sistemi di numerazione, apprezzare i vantaggi e gli svantaggi di ciascuno e dedurre alcune proprietà. In particolare c'è da notare cosa succede quando finiscono i simboli e c'è un "riporto" che si aggiunge alla cifra successiva e quando aumentano le cifre del numero.

Numero binario	Numero ottale	Numero decimale	Numero esadecimale
0	0	0	0
1	1	1	1
10_2	2	2	2
11_2	3	3	3
100_2	4	4	4
101_2	5	5	5
110_2	6	6	6
111_2	7	7	7
1000_2	10_8	8	8
1001_2	11_8	9	9
1010_2	12_8	10	A_{16}
1011_2	13_8	11	B_{16}
1100_2	14_8	12	C_{16}
1101_2	15_8	13	D_{16}
1110_2	16_8	14	E_{16}
1111_2	17_8	15	F_{16}
10000_2	20_8	16	10_{16}

Tabella 4: successione dei primi 16 numeri nelle diverse basi

Altre basi
I Maya, popolazione precolombiana dell'America centrale, rappresentavano i numeri in notazione posizionale a base 20. Nei documenti cerimoniali e nelle in epigrafi i simboli erano venti tipi di facce umane diverse. Per i calcoli correnti, che svolgevano con algoritmi simili ai nostri, usavano invece una notazione molto semplice, composta di punti e linee. Una linea valeva cinque, un punto uno.
La notazione dei Maya era posizionale, ma aveva un peso minore sulla terza cifra meno significativa, che invece di 20^2 arrivava a 19^2. Questo significa che la seconda cifra non veniva mandata fino a 19, ma solo fino a 18. Con questo "trucco" il numero 360, che aveva per loro un grande significato, per via del calendario, era un numero con tutte le cifre al valore massimo, e 361 era un numero "tondo", cioè aveva le due cifre meno significative nulle (maya19, maya19 = $0*19^2+ 18 * 19^1+ 20$). La terza cifra meno significativa di un numero Maya indicava perciò, almeno approssimativamente, il numero degli anni.

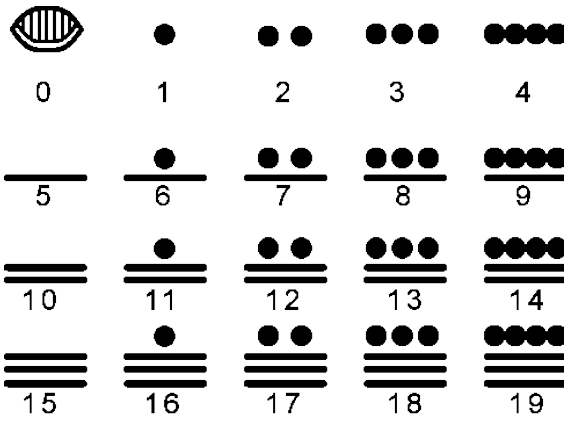


Figura 3: cifre Maya

GTGATAGGCCACTTACGGAA ..
La base 4 è molto usata in natura. Il DNA codifica informazioni che servono nella sintesi delle proteine, con le quali sono "costruiti" tutti gli esseri viventi. Le istruzioni biochimiche che determinano le caratteristiche di ogni essere vivente sono memorizzate nella sequenza di sole quattro basi[1]: **A**denina, **T**imina, **C**itosina e **G**uanina), che si susseguono una dopo l'altra, nella lunga elica del DNA.
In questo senso il DNA è un numero a base quattro, dato che è una sequenza di "simboli" ciascuno dei quali può assumere solo quattro valori (A, T C o G). E' interessante pensare che ogni batterio ed ogni uomo sono in ultima analisi un numero, anche se con un grande numero di cifre!
Il DNA della piccola mosca "Drosophila Melanogaster", "mappato" completamente, ha 120 milioni di "simboli". Si può dire che è un numero a base quattro di 120 milioni di cifre. Il DNA dell'uomo ha circa 3 miliardi di "cifre" e viene "mappato" nel progetto "genoma umano".

Un numero finito di cifre
In un computer dovrà esserci almeno un transistor per ogni bit di ogni numero che si vuole rappresentare. Dato che su di un circuito elettronico non si possono realizzare un numero infinito di transistor, ogni numero che un computer può trattare deve avere necessariamente un numero finito di cifre.
L'aritmetica di un computer è dunque "finita". Per tutte le operazioni si ha a disposizione un numero di cifre limitato, grande quanto si può (dipende da quanto si vuole "spendere"), ma non infinito. Ciò vuol dire che i computer non memorizzano esattamente i numeri irrazionali come per esempio la radice quadrata di 2 od i numeri che risultano periodici in base 2.
Per visualizzare la cosa possiamo pensare alle cifre del contachilometri di un motorino. Se continuiamo a viaggiare, prima o poi le cifre finiranno ed il numero ricomincerà da zero. In questo caso sul contachilometri non avremo più il numero giusto di chilometri percorsi.
Lo stesso accade se continuiamo ad aggiungere qualcosa ad un registro o ad una locazione di memoria di un computer. Verrà il momento in cui il risultato "non ci sta più"; allora il numero "ricomincia daccapo". Naturalmente se ciò accade il numero memorizzato non è il risultato corretto dell'operazione di somma.
Avendo a disposizione un numero finito di cifre ha senso chiedersi quanti numeri possiamo rappresentare con i bit di cui disponiamo.

Quanti numeri con n bit?
Guardiamo la colonna "Numero binario" della Tabella 4 e poniamo l'attenzione a quando il numero aumenta di una cifra. Questo succede in corrispondenza di 2, 4, 8 e 16, cioè delle potenze di 2.
Le potenze di due hanno dunque tutti zeri a destra di un unico uno. I numeri che li precedono sono fatti tutti di 1 e sono i numeri più grandi che si possono fare con quel numero di bit.
Dunque il numero più grande che si può fare con n bit è quello è quello prima di quando scatta l'aumento di cifre.
Per cui il numero massimo che si può rappresentare avendo a disposizione nBit bit è:

$$nMax = 2^{nBit} - 1$$

Quanti numeri si possono rappresentare? Siccome c'è anche lo zero essi sono 2^{nBit}.

Quanti bit per fare un numero?
Vediamo ora come si risolve il problema contrario. Vogliamo sapere con quanti bit si deve rappresentare un numero qualsiasi.
Prendiamo il caso del massimo numero rappresentabile, nell'equazione precedente e risolviamo per nBit:

$$nMax + 1 = 2^{nBit}$$

1 nel senso "chimico" della parola, cioè il contrario di "acidi".

che diventa, passando ambo i membri al logaritmo base 2:

$$nBit = log_2(2^{nBit}) = log_2(nMax + 1)$$

Se consideriamo un numero qualsiasi, n, e non il più grande nMax, possiamo sempre fare il logaritmo base 2, però poi dovremo arrotondare per eccesso:

$$nBit = IntSup[\log_2(n + 1)]$$

Dove IntSup è la parte intera superiore del numero frazionario che ha come argomento. In sintesi: aggiungiamo 1 al numero n, ne facciamo il logaritmo base 2, arrotondiamo all'intero successivo, e abbiamo ottenuto il numero di bit che ci servono.
Se il risultato del logaritmo è intero questo significa che il numero è una potenza di due, cioè il numero con il quale "avviene il riporto" (uno in più di nMax) e bisogna aggiungere un bit. In questo caso perciò bisogna aggiungere uno al risultato del logaritmo.

Vediamo tre esempi:
511 -> $\log_2(511)$ = 8,997179480938 => ci vogliono 9 bit

512 -> $\log_2(512)$ = 9 => ci vogliono 10 bit

513 -> $\log_2(513)$ = 9,002815015607 => ci vogliono 10 bit

Domanda: Come si fa a fare il logaritmo base 2 con una calcolatrice che ha solo i logaritmi base 10 e/o quelli base e (naturali)?

Si usa questa proprietà dei logaritmi:

$$log_a(x) = \frac{log_b(x)}{log_b(a)}$$

Che nel nostro caso diventa :

$$log_2(nBit) = \frac{log_{10}(nBit)}{log_{10}(2)} = \frac{ln(nBit)}{ln(2)}$$

Entrambe le formule sono estensibili a tutte le altre basi semplicemente cambiando il 2 con la base desiderata.

Numeri di molti bit
Gli informatici, per intendersi rapidamente sulle entità con cui hanno a che fare hanno dato nomi speciali ai numeri binari, in base al numero di cifre che possiedono.
Nibble
Un numero binario di quattro cifre viene detto "**nibble**" (parola che in Italiano non si traduce mai, ma che significa "piccolo morso", in contrapposizione a Byte (vedi in seguito)).
Un numero binario di quattro bit può assumere i valori da 0 a 15. Un nibble può essere utile per rappresentare una singola cifra esadecimale o per contenere una cifra BCD (vedi oltre).
Byte
Il **Byte** è un numero binario di otto bit (anche Byte non si traduce mai in Italiano. In Inglese è un termine inventato, ma "bite" significa "morso"). Il Byte è un numero che può assumere tutti i valori che vanno da 0 a 255 (2^8 -1).

Un Byte si usa tipicamente per contenere un carattere, o, meglio, il codice ASCII o ANSI del carattere.

Word
Il termine **word** può essere usato in modo ambiguo. Il significato originario del termine ha a che fare con l'architettura specifica di ogni tipo di computer. Al giorno d'oggi però il termine word di solito indica un numero di 16 bit.

Peraltro non è sempre così, dato che, per esempio, nei computer con PowerPC 16 bit sono una "half word", la word ha 32 bit, mentre nei computer con Alpha o 8086 il termine word significa 16 bit.
Potremo capire solo più avanti il perché di questa ambiguità.

Un numero di 16 bit può assumere 65536 valori (2^{16}), da 0 a 65535 (2^{16}–1).
Double Word (oppure Long Word)
E' un numero di 32 bit, che può assumere i valori che vanno da 0 a 4 294 967 295 (2^{32}–1)
Quad Word
E' un numero di 64 bit, che può assumere i valori che vanno da 0 a circa 1,84 * 10^{19} (cioè 2^{64}-1)
Si ribadisce che il termine Word è ambiguo e di conseguenza anche Double Word e Quad Word.

1.1.1 Conversioni fra le basi

Da tutte le basi a base 10
Questa conversione è molto semplice, si tratta solo di sviluppare il polinomio che genera il numero in notazione posizionale. I seguenti esempi dovrebbero chiarire tutto:

$1010_2 = 0 * 1 + 1 * 2 + 0 * 4 + 1 * 8 = 10_{10}$

$263_8 = 3 + 6 * 8 + 2 * 8^2 = 3 + 6 * 8 + 2 * 64 = 179_{10}$

$0F104_{16} = 4 + 0 * 16 + 1 * 16^2 + 15 * 16^3 = 4 + 256 + 15 * 4096 = 61700_{10}$

Da decimale a binario (base 2)
La conversione consiste in una serie di divisioni, nella quale ogni resto costituisce una cifra del risultato.
Preso il numero da convertire lo si divide per due, ottenendo il quoziente intero (la parte intera del risultato della divisione) ed il resto.
Il resto della prima divisione è la cifra meno significativa del numero convertito in binario.
A questo punto si prende il quoziente della prima divisione e lo si divide ancora per due. Il resto della seconda divisione è il secondo bit meno significativo del risultato. Il quoziente della seconda divisione si divide per due, e così via, fino a che il quoziente dell'ultima divisione non è zero. L'algoritmo è abbastanza semplice e si può svolgere anche senza calcolatrice, dato che le divisioni sono sempre e solo per due.

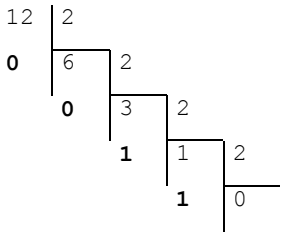


Figura 4: conversione di $12_{10} = 1100_2$

Da decimale ad esadecimale (base 16)
L'algoritmo è analogo al precedente, con l'unica differenza che tutte le divisioni sono per 16 invece che per 2. Le divisioni per 16 sono facili solo se si dispone di una calcolatrice. Se non si ha una calcolatrice a portata di mano tutto sommato è più veloce convertire in binario, poi da binario in esadecimale.

```
Se si dispone di una calcolatrice, di solito essa non è in grado di dare il risultato
delle divisioni come quoziente e resto. Quando si divide per sedici si otterrà un numero
decimale. Per ottenere il resto, togliere al risultato la parte intera e moltiplicare per
16.
Esempio:
61700 : 16 = 3856,25     3856,25 - 3856 = 0,25    0,25 * 16 =  4 (il primo resto)
3856  : 16 = 241                                  resto = 0
241   : 16 = 15,0625     15,0625 - 15 = 0,0625resto = 0,0625 * 16 = 1
15    : 16 = 0                                    resto 15 = Fh
Il numero convertito è dunque: F104
```

La conversione da decimale ad ottale è analoga e non la spieghiamo.

Fra binario ed esadecimale
Come già accennato questa conversione è molto facile e veloce. Dato che 16 è un multiplo di 2, c'è una corrispondenza fra le cifre del numero binario e quelle del numero esadecimale, per cui la conversione può funzionare in entrambi i sensi sostanzialmente con lo stesso algoritmo.
Per convertire basta dividere il numero binario in gruppi di 4 bit (nibble), a partire dal bit meno significativo. Poi si prenderà ciascuno dei gruppi e lo si convertirà nella corrispondente cifra esadecimale (naturalmente se il nibble corrisponde a 10 si metterà la cifra A, se è 11, B e così via sino a 15 (1111_2), che si sostituirà con F).

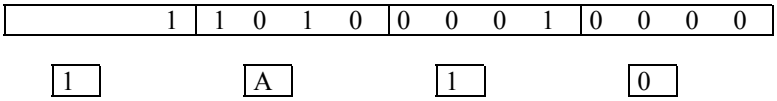


Figura 5: conversione di $1101000010000_2 = 1A10_{16}$

Fra ottale e binario si procederà a gruppi di 3 cifre, il resto sarà immutato.

1.1.2 Operazioni aritmetiche

Somma fra numeri binari
Date un paio di regole un po' strane, l'operazione in colonna diventa più semplice che in base 10. Posto che, come abbiamo già visto, $1_2 + 1_2 = 10_2$, questo significa che in una operazione in colonna 1 + 1 fa zero con il riporto di uno. L'altro caso che si può presentare sommando due numeri è $1 + 1 + 1 = 3_{10} = 11_2$, che si presenta quando c'è un riporto dalla colonna precedente. Ciò significa che, in base 2, 1 + 1 + 1 fa 1 con il riporto di 1.
Esempio, 52 + 29 = 81:

```
 1  1  1  1
    1  1  0  1  0  0  +
       1  1  1  0  1  =
-------------------------
 1  0  1  0  0  0  1
```

Sottrazione fra numeri binari
Anche nella sottrazione bisogna giustificare due regole strane. Se il bit del sottraendo è zero e quello del minuendo è uno si presenta il caso 0 - 1 e si deve andare a "prestito" dalla cifra più a sinistra. Con il prestito l'operazione diventa $10_2 - 1 = 2_{10} - 1 = 1$; perciò 0 - 1 = 1 con prestito. L'altro caso è -1 -1, che può capitare in presenza di prestiti dalla colonna precedente. In questo caso è necessario prendere prestito dalla colonna successiva e l'operazione diviene $10_2 - 1 - 1 = 2 - 1 - 1 = 0$. Per cui -1 -1 = 0 con prestito.

```
-1                          -1 -1
    0  -                        0  -
    1  =                        1  =
-------                     -------
    1                           0
```

Figura 6: prestiti nelle sottrazioni.

Esempio, 163 - 105 = 58:

```
-1 -1 -1 -1
 1  0  1  0  0  0  1  1  -        sottraendo
    1  1  0  1  0  0  1  =        minuendo
-------------------------
 0  0  1  1  1  0  1  0
    @  #  §  &
```

La colonna § equivale a 0 - 1 = 1 con prestito. Nella colonna # il -1 del prestito e l'uno del sottraendo si cancellano, così rimane solo il -1 del minuendo e la situazione è la stessa del caso &. La colonna @ è un caso di - 1 - 1 = 0 con prestito.

Moltiplicazione fra numeri binari
Per moltiplicare numeri binari non c'è bisogno di sapere le tabelline, perché ci sono solo 0 e 1, ma si può applicare lo stesso algoritmo visto alle scuole elementari. Per non complicare le somme potrebbe essere necessario compierle "due alla volta".

Esempio, 11 * 11 = 121:

				1	0	1	1	*
				1	0	1	1	
	r.1	r.1	r.1					
				1	1	0	1	+
			1	1	0	1	-	+
		0	0	0	0	-	-	+
	1	1	0	1	-	-	-	=
1	0	0	0	1	1	1	1	

Divisione fra numeri binari

Anche qui non è necessario conoscere le tabelline. L'algoritmo è molto più semplice a farsi che a dirsi.
Proviamo a dare una descrizione a parole di cosa bisogna fare. Il procedimento è per sottrazioni successive, identico a quello adottato per la base 10.
1 Si cerca nella parte più significativa del dividendo un numero più grande del divisore (segni ^^^^ nell'esempio)
2 Si mette un 1 nel quoziente
3 Si sottrae il divisore al numero trovato, ottenendo un resto (0011 nell'esempio), cui si aggiunge a destra la cifra più significativa fra quelle rimaste da elaborare del dividendo (primo segno v dell'esempio).
4 Si vede se il numero ottenuto (00111) è maggiore o minore del dividendo.
  4.A Se il numero è maggiore si aggiunge un 1 nel quoziente e si sottrae il dividendo (risultato 001 nell'esempio)
  4.B Se il numero è minore si scrive uno zero nel quoziente
5 Si prosegue aggiungendo un'altra cifra a destra prendendo la prima non ancora elaborata del dividendo (segni v) e tornando al punto 4, fino a che il dividendo non è stato elaborato tutto.
6 A questo punto abbiamo ottenuto il quoziente e ciò che rimane dall'ultima sottrazione è il resto della divisione intera.

Come anticipato questa descrizione è più complicata del calcolo; provando un paio di esempi si imparerà benissimo.

Esempio: 152 : 6 = 25 con resto di 2

^	^	^	^	v	v	v	v						
-1	-1												
1	0	0	1	1	0	0	0	1	1	0			
	1	1	0					1	1	0	0	1	
0	0	1	1	1									
		1	1	0									
		0	0	1	0	0	0						
					1	1	0						
				0	0	1	0	r.					

Si noti che in realtà i computer reali adottano algoritmi per la divisione intera molto più efficienti e veloci di quello appena descritto.

Somma fra numeri esadecimali

Per fare bene sulla carta le somme esadecimali bisognerebbe imparare il risultato di molte somme strane come D + 9, F + C, e così via. Dato che non si usano così spesso è meglio aiutarsi abbondantemente con le dita, oppure passare in binario, magari solo per quattro bit, fare la somma e ripassare in esadecimale.

	r.1		r.1		
	F	E	D	E	+
		F	1	8	=
1	0	D	F	6	

Operazioni con numeri interi anche negativi

La numerazione binaria che abbiamo visto non sembra essere adatta a rappresentare i numeri negativi. Infatti il polinomio che genera i numeri in qualsiasi base non prevede numeri negativi. I numeri binari "puri" sono perciò senza segno. Dato che è indispensabile usare anche i numeri con segno, vediamo due modi per poterlo fare, ma prima facciamo notare che il primo dei due metodi NON è quello usato nei computer.

Rappresentazione dei numeri interi e negativi in modulo e segno

La cosa più "naturale" che viene in mente è dedicare uno dei bit del numero al segno, lasciando tutto il resto come prima. Questo modo di procedere equivale ad utilizzare il valore assoluto del numero (detto anche "modulo" in matematica) ed il suo segno. Per il segno possiamo scegliere di scrivere 0 per i numeri positivi e 1 per i negativi.
Considerando il numero 12 a 7 bit, con un ottavo bit che rappresenta il segno, abbiamo:

0	0	0	0	1	1	0	0

La rappresentazione di -12 in modulo e segno sarebbe:

1	0	0	0	1	1	0	0

Per quanto questa sia la rappresentazione più semplice da fare sulla carta, essa non è il modo migliore di rappresentare i numeri negativi in un computer. Infatti tutti i computer usano la rappresentazione in complemento a due.

Rappresentazione dei numeri negativi in complemento a due

Questo modo un po' strano di rappresentare i numeri negativi ha motivazioni pratiche. Se si usano numeri negativi in complemento a due i computer possono utilizzare gli stessi circuiti elettronici per fare sia le somme sia le sottrazioni. Se al contrario, per qualche insana ragione, un computer usasse numeri negativi in modulo e segno, si dovrebbero dedicare molti circuiti per fare la somma ed altri per fare la sottrazione. Siccome i circuiti risparmiati possono essere impiegati più utilmente per fare altre cose, non c'è nessun computer che usi un'algebra in modulo e segno per fare le operazioni con i numeri negativi.
Per questo quando un computer deve cambiare il segno di un numero intero ne fa sempre il complemento a due.
Detto della motivazione pratica, vediamo come si ottiene il complemento a due di un numero.
Per ottenere il complemento a due di un numero si fanno due operazioni:

1. operazione di "**complemento ad uno**": si inverte il valore di ciascun bit del numero. Se il bit vale 1 lo si fa diventare 0, 0 se vale 1.
2. incremento: al primo risultato si aggiunge 1.

Considerando ancora 12 e -12:

0	0	0	0	1	1	0	0	12

0	0	0	0	1	1	0	0	NOT
1	1	1	1	0	0	1	1	+
							1	=
1	1	1	1	0	1	0	0	

1	1	1	1	0	1	0	0	-12

Si nota come facendo il complemento a due cambia tutto il numero, non solo il bit più significativo.

Proviamo a rifare il complemento a due di quello che abbiamo detto essere -12:

1	1	1	1	0	1	0	0	-12

1	1	1	1	0	1	0	0	NOT
0	0	0	0	1	0	1	1	+
							1	=
0	0	0	0	1	1	0	0	

0	0	0	0	1	1	0	0	12

L'operazione di complemento a due si comporta effettivamente come il cambio di segno, se la applichiamo due volte fa tornare il numero originario, questo succede sempre.
Al bit più significativo di un numero con segno rappresentato in complemento a due, può comunque essere attribuito il significato di segno.

Un numero intero in complemento a due è positivo se il bit più significativo è zero, è negativo se è uno.

I numeri in complemento a due hanno senso solo se hanno un numero finito di bit, bisogna sapere con quanti bit è fatto un numero in complemento a due, se non altro per sapere dov'è il bit del segno!
Una sequenza di bit viene detta "**numero senza segno**" se è un normalissimo numero binario, sempre positivo, e "numero **con segno**" se consideriamo come segno il suo bit più significativo ed il numero è rappresentato in complemento a due.
È molto importante capire che la stessa sequenza di bit può rappresentare numeri diversi, se la si considera un numero con segno oppure senza segno, in questo aiuta la successiva tabella:

Numeri con segno			Sequenza di bit	Numeri senza segno		
a 8 bit	a 16 bit	a 32 bit		a 32 bit	a 16 bit	a 8 bit
0	0	0	000 .. 000	0	0	0
1	1	1	000 .. 001	1	1	1
			..			
126	32766	2147483646	011 .. 110	2147483646	32766	126
127	32767	2147483647	011 .. 111	2147483647	32767	**127**
-128	-32768	-2147483648	100 .. 000	2147483648	32768	**128**
-127	-32767	-2147483647	100 .. 001	2147483649	32769	129
			..			
-2	-2	-2	111 .. 110	4294967294	65534	254
-1	-1	-1	111 .. 111	4294967295	65535	255

Tabella 5: numeri con segno o senza segno

Ragionare per un po' su questa tabella fa capire com'è fatta la notazione in complemento a due.
Confrontando le colonne di sinistra con quelle di destra si vede che:
- gli stessi bit rappresentano numeri diversi
- le sequenze di bit che hanno il bit più significativo a zero sono quelli più piccoli della metà e rappresentano sempre lo stesso numero, siano che li consideriamo con segno oppure senza segno
- il numero con segno costituito di tutti 1 è comunque –1, quale che sia il numero di bit che lo compone

Puntiamo ora l'attenzione su cosa succede nei numeri in complemento a due quando si aggiunge 1 al numero fatto con uno 0 e tutti 1. Questa è la riga della tabella in cui cambia il segno, a 8 bit si passa da 127 a -128. Ma com'è possibile? Proviamo a fare il complemento a due di 128:

1	0	0	0	0	0	0	0	+128 senza segno

1	0	0	0	0	0	0	0	NOT
r.1	r.1	r.1	r.1	r.1	r.1	r.1		
0	1	1	1	1	1	1	1	+
							1	=
1	0	0	0	0	0	0	0	

1	0	0	0	0	0	0	0	-128 con segno

Il complemento a due del numero "del mezzo" è uguale a se stesso!
Ciò vale anche per lo zero, il cui complemento a due è:

0	0	0	0	0	0	0	0	Zero senza segno

0	0	0	0	0	0	0	0	NOT
r.1	r.1	r.1	r.1	r.1	r.1	r.1		
1	1	1	1	1	1	1	1	+
							1	=
1	0	0	0	0	0	0	0	

0	0	0	0	0	0	0	0	0 con segno

Naturalmente, mentre l'operazione 127 + 1 funziona fra numeri senza segno, essa è un errore fra numeri con segno. Nei capitoli successivi verrà spiegato come viene segnalata questa anomalia nei computer.
I numeri che sono scritti nella tabella sono più importanti degli altri e ogni tanto possono servire. Così sarebbe bene conoscerli "a memoria" (non è il caso per quelli a 32 bit!).

Operazioni in complemento a due
Le magiche proprietà dei numeri in complemento a due si apprezzano facendo qualche somma.

	r.1	r.1		r.1	r.1	r.1	r.1		
	0	1	1	0	0	0	1	1	+
	1	0	1	0	1	1	0	1	=
1	0	0	0	1	0	0	0	0	

c

Se vediamo questa operazione come una somma fra numeri senza segno essa è 99 + 173 = 272. Da notare che se tutti questi numeri, risultato compreso, devono essere di 8 bit il risultato "non ci sta" e ci sarebbe un riporto, indicato con c (c sta per "carry" che in Inglese significa riporto (oltre che "carriola", ma non c'entra!).
Se al contrario vediamo la somma precedente come una operazione fra numeri con segno di 8 bit, rappresentati in complemento a due, questa operazione è la somma fra 99 e un numero negativo, dato che il secondo numero ha il bit più significativo uguale a uno.
Quanto vale quel numero negativo? Se si vuole sapere il valore assoluto di un numero negativo lo si deve cambiare di segno, ciò è esattamente quello che andremo a fare. Vediamo dunque il complemento a due:

1	0	1	0	1	1	0	1	-?
0	1	0	1	0	0	1	0	+
							1	=
0	1	0	1	0	0	1	1	
0	1	0	1	0	0	1	1	83

Il secondo numero sommato precedentemente era quindi -83 e l'operazione fra numeri con segno era 99 + (-83) = 16. $16_{10} = 00010000_2$ è proprio il risultato della somma, se escludiamo il bit di riporto (c). Nelle operazioni fra numeri con segno si ignora sempre il bit di carry, per cui l'operazione appena vista è lecita e giusta.
Dunque la stessa, identica, operazione può essere considerata due diverse operazioni, una con segno e l'altra senza segno.
Avendo a disposizione un'operazione di cambio di segno si può pensare di fare le sottrazioni utilizzando una somma nella quale il secondo operando sia cambiato di segno.
Qui sta il magico dei numeri in complemento a due: con la stessa operazione, cioè un definitiva lo stesso circuito elettrico, possiamo fare sia le addizioni che le sottrazioni fra numeri interi.

Moltiplicazione e divisione fra numeri interi
Per la moltiplicazione e la divisione il complemento a due non aiuta e bisogna usare algoritmi diversi per le operazioni con numeri senza segno e quelle con numeri con segno. Da ciò consegue che ci dovranno essere delle istruzioni particolari per effettuare i due tipi di operazioni. Gli algoritmi che si usano non sono molto diversi da quelli per le operazioni sulla carta, anche se sono ottimizzati rispetto ad essi.

Rappresentazione dei numeri a virgola mobile (floating point)

Fino ad ora abbiamo parlato di numeri interi ed abbiamo detto che è facile per un computer eseguire tutte le operazioni fra questi numeri.
Ma i numeri interi non sono il solo tipo di numeri che sono indispensabile nell'informatica. Infatti, per svolgere molti tipi di funzioni, è indispensabile per i computer eseguire operazioni anche con i numeri frazionari. Sarà perciò necessario stabilire una convenzione per la rappresentazione dei numeri frazionari e algoritmi per la realizzazione delle operazioni fra di essi.
La rappresentazione dei numeri frazionari viene detta "a virgola mobile" (**floating point**) ed è simile alla notazione scientifica, che esprime i numeri "spostando la virgola" con potenze di 10.
Il numero 123,456 può essere scritto $123{,}456 * 10^0$, oppure $1{,}23456 * 10^2$, $0{,}123456 * 10^3$ o anche $123456 * 10^{-3}$.
Il numero al quale viene elevato l'esponete della base viene detto "esponente", le cifre significative vengono dette "mantissa" (in Inglese mantissa o "fraction").
Lo stesso si fa con i numeri binari a virgola mobile, solo che tutto è in base 2:

$$numero = Segno * 1,mantissa * 2^{esponente}$$

Segno è +1 se il relativo bit è zero, oppure -1 se il bit è uno. La scritta "1,mantissa" significa che l'esponente sposta la virgola in modo che essa sia subito dopo il primo 1. Se la virgola sta sempre in quella posizione si dice che il numero è "normalizzato". La memorizzazione normalizzata è interessante perché nella prima posizione c'è sempre un uno. Così quell'uno non si memorizza e questo fa risparmiare un bit di memoria.
I numeri a virgola mobile sono rappresentati con tre campi: segno, esponente e mantissa (fraction).
Attualmente la lunghezza ed il contenuto di questi campi sono codificati da uno standard. Quasi tutti le librerie di software aritmetico ed i coprocessori matematici, cioè le componenti del computer che devono fare i calcoli in virgola mobile, sono conformi alle specifiche della norma IEEE 754.
Lo standard IEEE 754 prevede sostanzialmente tre tipi di numeri a virgola mobile: in singola precisione, in doppia precisione ed estesi. Un numero in singola precisione occupa 32 bit, uno in doppia precisione ne occupa 64, mentre un esteso 128.
La struttura dei numeri dello standard IEEE 754 è quella di seguito illustrata:

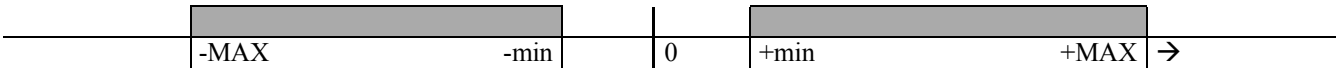
Numeri floating point a singola precisione:

S	Esponente (8bit)		Mantissa (23 bit)	
31	30	23	22	0

Numeri floating point a doppia precisione:

S	Esponente (11 bit)	Mantissa (52 bit)
63	62 52	51 0

Il numero esteso, di 128 bit, ha 1 bit di segno, 15 bit di esponente, 108 bit di mantissa
La norma prevede numeri positivi e negativi assolutamente "simmetrici", l'unica differenza che ha un numero positivo da uno negativo è il bit di segno (in questo si assomiglia alla notazione in modulo e segno dei numeri interi).
Il numero zero viene rappresentato con mantissa =0 ed esponente = 0, il bit di segno è indifferente per lo zero. Esistono perciò uno zero "positivo" ed uno "negativo".
Oltre ai numeri usuali la norma codifica anche dei casi particolari, delle entità non numeriche. Esse sono +(infinito), -(infinito) e NaN (Not a Number).
I numeri infiniti sono generati quando il risultato è maggiore del massimo o minore del minimo rappresentabile. NaN viene generato nei casi in cui si hanno indeterminazioni del risultato, come in 0/0.
NaN può anche segnalare che tipo di errore aritmetico è stato commesso. Quando l'esponente è tutti 1 il "numero" è un caso particolare.
Detto min il numero rappresentabile più vicino allo zero e MAX il numero più grande rappresentabile, il campo dei numeri rappresentabili in virgola mobile è quello indicato nel seguente diagramma, che mostra in particolare come il numero 0 è rappresentabile, ma non il suo intorno più vicino, fra –min e +min:



I valori di min e MAX per i tre tipi di numeri della IEEE 754 sono i seguenti:

Singola precisione (32 bit)
$min = 1,0 * 2^{-126} = 1,17549435 * 10^{-38}$
$MAX = (2,0 – 2^{-23}) * 2^{127} = 3,40282347 * 10^{38}$

Doppia precisione (64 bit)
$min = 1,0 * 2^{-1022} = 2,2250738585072013 * 10^{-308}$
$MAX = (2,0 – 2^{-52}) * 2^{1023} = 1,7976931348623158 * 10^{308}$

Precisione estesa (128 bit)
$min = 1,0 * 2^{-16382} = 3,3621031431120935062626778173217526 0 * 10^{-4932}$
$MAX = (2,0 - 2^{-112}) * 2^{16383} = 1,1897314953572317650857593266280070 2 * 10^{4932}$

Operazioni fra numeri in virgola mobile
Per fare le operazioni in virgola mobile bisogna "mettere in colonna" i termini in modo da riportarsi ad operazioni aritmetiche normali.

E' importante tener presente che anche i numeri floating point hanno un numero finito di cifre.

Per quanto si riescano a rappresentare, muovendo la virgola con l'esponente, numeri molto grandi, non tutte le cifre di quei numeri sono significative.
Vediamo per esempio cosa succede nella somma fra un numero troppo piccolo ed uno troppo grande. Prendiamo un numero dieci volte più piccolo di MAX in singola precisione:
$3,40282347 * 10^{37} = 340282347000000000000000000000000000000$

Dopo l'ultimo 7 ci possono essere solo zeri, perché la mantissa non ha abbastanza cifre significative!
Se al numero di prima provo ad aggiungere $1 * 10^{27} = 100000000000000000000000000$, che pure non è un numero piccolo, il risultato rimarrà identico al primo numero, perché dopo aver messo in colonna i due numeri il più piccolo sarà troppo piccolo per influenzare il valore del grande.
Ciò significa che la somma di molti numeri fatta con un computer può dare risultati diversi se fatta sommando dal numero più grande al più piccolo o viceversa (i risultati più accurati si otterranno sommando dal più piccolo al più grande).

Vediamo ora un esempio che riguarda una somma a virgola mobile in base 2. Stabiliamo ad arbitrio una rappresentazione dei numeri, senza pensare allo standard e cercando solo di semplificare l'esempio.
Supponiamo di disporre di 8 bit per la mantissa, quattro bit per l'esponente ed un bit per il segno.
Realizziamo la somma fra i numeri 193,79 e

Molte CPU hanno istruzioni specifiche per facilitare la "messa in colonna" dei numeri, tipica delle operazioni a virgola mobile (istruzioni di "denormalizzazione").
Per effettuare le operazioni a virgola mobile si usano librerie di programmi, che si possono acquistare e collegare al nostro programma, oppure dispositivi hardware dedicati, che sono detti "coprocessori aritmetici" (FPU, **F**loating **P**oint **U**nit, Unità di calcolo in virgola mobile).
Le CPU più moderne hanno unità aritmetiche a virgola mobile al loro interno e possono quindi realizzare in hardware le operazioni floating point.

Codice BCD
Il codice BCD (Binary Coded Decimal) è un modo un po' "sprecone" di rappresentare i numeri. Esso infatti non impiega completamente i bit che occupa.
Codificare una cifra in BCD consiste nell'usare 4 bit per rappresentare solo le dieci cifre numeriche decimali. Usare meno di 4 bit non si potrebbe, perché con 3 bit si arriva solo fino al numero 7. Con 4 bit si può arrivare sino al numero 15, per cui, codificando il numero in BCD non si utilizzano 6 combinazioni di bit per ogni cifra, cioè quelle che in esadecimale sono le cifre da A ad F.

Vediamo come esempio la codifica in BCD del numero 2096, che richiede 16 bit:

0	0	1	0	0	0	0	0	1	0	0	1	0	1	1	0
2				0				9				6			

Il codice BCD era utilizzato nella notte dei tempi ;-) per memorizzare i numeri negli hard disk dei mainframe, oggi è ancora molto usato nei display numerici a LED. Quando si devono usare tali display è probabile dover fare operazioni con i numeri in BCD. Diversi tipi di computer, fra i quali anche quelli con microprocessore 8086, hanno istruzioni specifiche per effettuare conversioni fra binario e BCD ed anche operazioni aritmetiche che funzionano direttamente con numeri in BCD.

Vediamo quanto si "spreca" con un numero di 9 cifre, poniamo 1 miliardo (999 999 999). In BCD ciascuna cifra decimale prenderà 8 bit, per cui per il nostro miliardo ci vorranno 9 x 8 = 72 bit. Se invece rappresentiamo un miliardo con un normale numero binario, il numero di bit richiesti sarà 30, infatti:

$$nBit = IntSup\,[\log_2(999999999+1)$$
$$¿ = IntSup[29,89] = 30\,¿$$

Il che significa che 42 dei 72 bit della rappresentazione in BCD sono sprecati.

1.2 Rappresentazioni delle lettere

Per elaborare le informazioni in modo che possano essere utilizzate dagli esseri umani è indispensabile codificare, oltre ai numeri, anche altri simboli con i quali essi comunicano, in particolare le lettere dell'alfabeto. Le informazioni devono essere presentate in modo "alfanumerico", cioè facendo uso di caratteri dell'alfabeto e di numeri. Per rappresentare le lettere bisogna fare uso di codici diversi rispetto a quelli già presentati. Alcuni di questi codici esistono da prima dell'era dei computer, altri sono recenti.
Nel gergo informatico una lettera, o una cifra da stampare, viene detta "**carattere**" **alfanumerico** (character) e una successione di caratteri viene detta "**stringa**" (string).

Codice Hollerit
La prima macchina elettrica di elaborazione delle informazioni fu la macchina di Hollerit (1890), inventata per automatizzare i calcoli nelle statistiche demografiche in USA.
La macchina di Hollerit utilizzava schede di carta perforate per far avanzare dei contatori elettromeccanici. Se la carta era forata veniva fatto un contatto che incrementava il valore del totalizzatore, altrimenti il valore non avanzava.
Il codice Hollerit fu usato nelle schede perforate, che venivano utilizzate per i dati ed i programmi dei primi computer.

Codice EBCDIC
Nei "mainframe", i grossi computer centrali che imperavano nei tempi andati e che sono ancora usati estesamente, l'IBM introdusse un codice, derivato da quello che si usava nelle schede perforate, che chiamò EBCDIC. Esso è del tutto diverso dal codice ASCII ed estende il codice BCD ai caratteri alfabetici.

La seguente tabella mostra un parziale confronto fra codice ASCII e EDCDIC:

Carattere	Codice ASCII	Codice EBCDIC
A	97d	129d
?	63d	111d
0	48d	240d
H	72d	200d
I	73d	201d
J	74d	209d
K	75d	210d

Come si vede non ci sono relazioni fra i due tipi di codifica. Inoltre si può notare che nel codice EBCDIC c'è un "salto" nel passare dalla codifica di I a quella di J, nel codice EBCDIC l'alfabeto viene perciò "interrotto".

Codice ASCII

ASCII significa "American Standard Code for Information Interchange" ed è una codifica alfanumerica che data al 1963. Esso fu ratificato come standard dall'ANSI nel 1968. L'ANSI (**A**merican **N**ational **S**tandards **I**nstitute) è l'organismo di normalizzazione degli USA, corrispondente al nostro UNI.
Il codice ASCII standard ha 7 bit, per cui è possibile rappresentare 128 simboli. I primi 32 numeri del codice ASCII non sono codici di caratteri di tastiera, ma sono riservati a comandi per il dispositivo che li riceve. Sono i cosiddetti "caratteri non stampabili", o "caratteri di controllo" (control characters).
Chi riceve i caratteri di controllo li può interpretare come comandi o azioni da compiere, come p.es. "carriage return" (codice ASCII = 13), che manda a capo di riga il carrello di una stampante, "line feed" (nuova linea, codice ASCII = 10), che manda avanti di una linea la carta di una stampante, "form feed" che manda avanti fino alla fine del foglio, "escape", che viene usato da tastiera per significare "torna indietro" ed altri.
Nel codice ASCII:
- con i codici da 30h a 3Fh vengono codificati simboli numerici e d'interpunzione
- con i codici da 40h a 5Fh le lettere maiuscole
- con i codici da 60h a 7Fh le minuscole.

Codice ASCII esteso
Il codice ASCII esteso è a 8 bit. Esistono molti codici ASCII estesi, per esempio il Commodore 64 aveva il suo. Di gran lunga il più popolare è quello introdotto dall'IBM quando produsse il primo PC.
Nel set di caratteri esteso tutti i primi 128 codici sono identici all'ASCII standard mentre gli altri 128 includono caratteri semigrafici per disegnare rettangoli sullo schermo, segni speciali matematici, qualche lettera greca e soprattutto le lettere particolari degli alfabeti occidentali, quali le vocali accentate per l'Italiano oppure la o con la dieresi per la lingua tedesca.
Purtroppo il numero di codici a disposizione non bastava per coprire tutte le lettere strane dei vari alfabeti, così esistono di fatto diversi set di caratteri ASCII estesi anche per il PC, sostanzialmente uno per ogni lingua, ciascuno diverso dall'altro.

Codice ISO 8859-1 (Windows ANSI) a 8 bit

L'ISO (l'organizzazione mondiale degli standard) ha normalizzato un'estensione ad 8 bit della tabella ASCII, con lo standard ISO 8859-1. Questa norma prevede una codifica che riprende una normativa ANSI precedente.
I primi 128 caratteri sono identici al codice ASCII.
I nuovi 128 caratteri sono stati usati per alcuni simboli di monete, segni d'interpunzione particolari e per le lettere accentate e modificate delle lingue europee.
I caratteri dal 128 al 159 di ISO 8859-1 sono esplicitamente indicati come caratteri di controllo.
Windows usa nella sua rappresentazione delle stringhe, il codice ISO 8859-1, ma solo per tutti i caratteri dal 159 in su. Al di sotto del 159, Windows usa alcuni dei codici da 128 a 159 per caratteri stampabili.
Per questo Windows non è completamente compatibile con ISO 8859-1, ma usa un suo codice che il produttore di Windows chiama genericamente "ANSI", senza specificare in numero della norma cui sarebbe conforme.
Data l'ubiquità di Windows, la codifica "Windows ANSI" è diventata il modo più usato di codificare i caratteri.

Codice Unicode

Per la rappresentazione alfanumerica "universale" di tutti i caratteri degli alfabeti umani non bastano 8 bit.
Per questo l'ISO ha steso una norma generica per la realizzazione di codici alfanumerici, che va sotto il nome di ISO 10460, che definisce l'UCS (**U**niversal **C**haracter **S**et).
Il codice Unicode è stato sviluppato per essere compatibile con l'ISO 10460, da un consorzio fondato nel 1991.
Unicode è a 16 bit. Avendo 65536 simboli esso può contenere, almeno in parte, tutti gli alfabeti del mondo. Si ricorda che l'alfabeto cinese comprende circa 40000 caratteri, ma moltissimi non sono usati praticamente mai e gli stessi concetti possono essere espressi con caratteri di uso più comune. Unicode comprende anche alcuni set di caratteri speciali per campi specialistici, come segni d'interpunzione o simboli tecnici e matematici, ed alfabeti di lingue antiche.
I primi 128 caratteri di Unicode sono i normali caratteri ASCII, i successivi, fino a 256, coincidono in gran parte con il codice ANSI.

I set di caratteri compresi in Unicode sono divisi per tipo di carattere, non per lingua. Il tipo di carattere viene definito "script" e comprende tutte le variazioni per adattarsi alle varie lingue che lo usano. Per esempio lo script "Latin" comprende le classiche lettere codificate anche nel codice ASCII, ma viene usato per molte decine di lingue e quindi comprende anche la o con i due puntini per il Tedesco o le lettere accentate per l'Italiano. Nella versione 2.0 Unicode contiene 38 885 diversi codici e 25 script.
Alcuni degli script di Unicode sono: Arabic, Bopomofo | 8- |, Cyrillic, Greek, Hebrew, Hiragana, Katakana (due tipi di caratteri giapponesi fonetici), Latin, Phonetic (i simboli che si trovano nei dizionari per indicare la pronuncia),
Una tabella aggiornata dei codici Unicode si può trovare in http://www.unicode.org/
Il codice Unicode è utilizzato e/o supportato dai Sistemi Operativi e dai linguaggi moderni.
Il modo più comune di utilizzare i caratteri Unicode è quello che fa uso della codifica "**UTF-8**", una **versione compressa di Unicode che usa 8 bit per rappresentare i primi 256 caratteri di Unicode e 16 per gli altri. In questo modo i file che contengono solo caratteri ANSI occupano la metà di quelli con "puro Unicode".**
Si possono portare alcuni esempi significativi: le stringhe gestite da Windows NT sono sempre basate su Unicode, così come quelle del linnguaggio C#; lo possono essere anche quelle del linguaggio Java. Lo stesso codice sorgente Java si può scrivere in Unicode. Windows 95 e 98 sono basati su stringhe ANSI, ma mettono a disposizione funzionalità di conversione ed utilizzano Unicode in alcune loro funzioni, lo stesso discorso vale per il linguaggio Visual BASIC; esistono librerie per il linguaggio C. Naturalmente tutti i sistemi operativi ed i linguaggi scritti prima che Unicode venisse sviluppato, p.es. MS DOS e tutti i linguaggi per applicazioni MS DOS non hanno nessun tipo di supporto per Unicode e lavorano con codice ASCII esteso.

UCS (*Universal Character System*)

Dato il numero esorbitante di caratteri delle scritture ideografiche è stato definito un ulteriore standard, ISO 10646, che usa 4 Byte per ciascun simbolo, con la possibilità perciò di poter definire 4 Giga di simboli (circa quattro miliardi).
Il sito della IANA (*Internet Assigned Numbers Authority*) contiene riferimenti ai tipi di codici esistenti, all'indirizzo www.isi.edu/in-notes/iana/assignments/character-sets.

Stringhe

Una **stringa** è una successione di caratteri alfanumerici, espressi con uno dei codici appena visti. Detta in modo un po' brutale una stringa è una "scritta". Ogni linguaggio ha il suo modo per rappresentare le stringhe, i modi possono essere considerevolmente diversi. Più avanti vedremo qualcosa al riguardo.

Problemi:

Problema extraterrestre: Nel nostro pianeta alieno gli abitanti di Gluk sono molto beneducati e non hanno parolacce nel loro vocabolario. Per le targhe dei loro motorazzetti possono mescolare lettere dell'alfabeto e numeri, senza dover scartare nessuna combinazione di lettere perché "sconveniente". Per "puro caso" hanno un alfabeto con 26 simboli, come in Italia, e già sappiamo che hanno 10 simboli per i numeri. Le targhe possono quindi contenere numeri in base 36. Per fare le targhe devono usare il minor numero possibile di cifre, così tengono meno posto (i motorazzetti sono piccoli!). Siccome devono prevedere di targare 1 000 000 di veicoli, quante cifre ci vogliono per il numero?
Determinato il numero di cifre che servono, qual è il numero massimo di motorazzetti che potranno targare, con le targhe così lunghe?

Altro problema, più "terrestre" ed un po' più complicato: quante targhe si possono fare con l'attuale sistema di numerazione delle targhe automobilistiche italiane (considerare che vengano usate tutte le lettere, anche quelle che danno problemi "di lettura" come "I", "O" oppure "Q")?