Dispense
Introduzione al calcolatore

Corso: Fondamenti Informatica I

Prof. Paolo Nesi

A.A. 2002/2003

Nota: Queste dispense integrano e non sostituiscono quanto scritto sul libro di testo.

1 Sistemi di numerazione e numeri binari

Un numero è una entità astratta che noi rappresentiamo con una sequenza di simboli, dette cifre. Il sistema di numerazione che usiamo usualmente è un sistema di numerazione posizionale in base 10. Nel nostro sistema di numerazione usiamo 10 simboli: 0,1,2,3,4,5,6,7,8,9
Ogni cifra del numero ha un peso diverso a seconda della posizione in cui si trova. Ad esempio il numero:

3624

lo si può vedere come

$3 \cdot 1000 + 6 \cdot 100 + 2 \cdot 10 + 4 \cdot 1$

o meglio come

$3 \cdot 10^{\mathbf{3}} + 6 \cdot 10^{\mathbf{2}} + 2 \cdot 10^{\mathbf{1}} + 4 \cdot 10^{\mathbf{0}}$

l'esponente indica appunto la posizione della cifra, dove la posizione 0 è quella più a destra e le posizioni crescono verso sinistra, mentre il valore che viene elevato a potenza è proprio la base (10).

Questo succede con la base 10 ma cosa si ha nel caso il numero sia espresso in una base diversa? Ad esempio i *numeri binari* sono espressi in base 2 e usano solo due simboli 0 e 1.

Il numero

100110

espresso in base 2 (si legge "uno zero zero uno uno zero" e non "centomilacentodieci") rappresenta il numero:

$1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$

e quindi il numero espresso in base 10:

$1 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = 32+4+2 = 38$

Quando non è chiaro dal contesto quale sia la base in cui è espresso un numero per evitare ambiguità questa viene scritta come pedice, ad esempio $(10110)_2$ è un numero binario mentre $(10110)_{10}$ è un numero decimale.

Per facilitare la conversione di un numero binario in decimale può essere di aiuto scrivere sotto la cifra il peso che questa ha, partendo da quella più a destra che ha peso 1 e raddoppiando il peso via via che ci si sposta verso sinistra:

	1	0	1	0	1	1	0	1
Peso:	***128***	*64*	***32***	*16*	***8***	***4***	*2*	***1***

Sommando i pesi in corrispondenza delle cifre pari a 1 si ottiene il numero rappresentato in base 10, nel nostro caso 128+32+8+4+1=173.

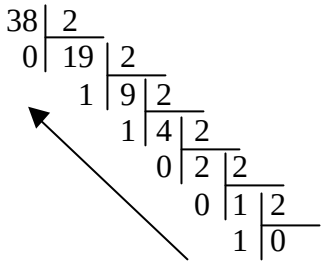
Con 3 cifre binarie che numeri si possono rappresentare?

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Si possono quindi rappresentare 8 numeri diversi ($8=2^3$)
Con *n* cifre binarie si possono rappresentare 2^n possibili configurazioni, mentre il massimo numero rappresentabile con n cifre binarie è pari a $2^n - 1$ ed il valore minimo è pari a 0.

Quindi con 8 cifre binarie si possono rappresentare 256 numeri, con 16 cifre 65.536 numeri, con 32 cifre 4.294.967.296 numeri, con 64 cifre 18.446.744.073.709.551.616 numeri etc.

Abbiamo visto come dato il numero binario si ottenga il numero decimale che questo rappresenta. Vediamo ora il contrario cioè come da un numero decimale si possa ottenere la sua rappresentazione come numero binario. Il metodo che si segue è detto delle divisioni successive, si prende il numero e lo si divide per due, il resto che si ottiene (che può essere solo 0 o 1) è la cifra di posizione 0, il quoziente ottenuto lo si divide per due e il nuovo resto che si ottiene è la cifra di posizione 1, l'ultimo quoziente ottenuto si divide ancora per due, il resto è la cifra di posizione 2 e si prosegue fino ad ottenere come quoziente il valore 0.
Ad esempio:



Il numero binario quindi lo si ottiene scrivendo i resti partendo dall'ultimo ottenuto:

$$(100110)_2$$

che è proprio il numero atteso.

2 Algebra di Boole

I fondamenti dell'algebra Booleana sono stati delineati dal matematico inglese George Boole, nato a Lincoln nel 1815 e morto nel 1864, in un lavoro pubblicato nel 1847 riguardante l'analisi della logica e in particolare l'algebra della logica. Questa algebra include una serie di operazioni che si effettuano su delle variabili logiche, dette appunto variabili Booleane: quantità che permettono di codificare le informazioni su due soli livelli. Nell'algebra di Boole essendo, a differenza di quella tradizionale, un'algebra binaria, le variabili possono assumere soltanto due *stati*:

- 0/1;
- v/f (vero, falso);
- l/h (low, high);
- t/f (true, false);
- on/off; (acceso/spento)
- T,⊥ (true/false, vero/falso)

Ad esempio la variabile logica *x* può avere valore **vero** o valore **falso**. La variabile logica *oggi_piove* ha associato un valore che indica appunto se oggi piove oppure no, quindi può essere vera oppure no.

Le *Funzioni logiche* o booleane sono funzioni che associano ad una sequenza di variabili booleane un valore booleano, ad esempio:

$$F(x,y,z)$$

La funzione F associa in base ai tre valori logici assunti dalle variabili logiche *x*,*y*,*z* un valore logico vero o falso. Si può osservare che si hanno solo 8 possibilità per i valori assunti dalle tre variabili *x*, *y*, *z*:

1. *x* falso, *y* falso, *z* falso
2. *x* falso, *y* falso, *z* vero
3. *x* falso, *y* vero, *z* falso
4. *x* falso, *y* vero, *z* vero
5. *x* vero, *y* falso, *z* falso
6. *x* vero, *y* falso, *z* vero
7. *x* vero, *y* vero, *z* falso
8. *x* vero, *y* vero, *z* vero

Per ognuna di queste otto possibilità la funzione F assume valore vero oppure falso.
Un modo semplice per rappresentare graficamente questa associazione usa una tabella detta *tabella di verità:*

x	*y*	*z*	*F(x,y,z)*
F	*F*	*F*	*V*
F	*F*	*V*	*F*
F	*V*	*F*	*F*
F	*V*	*V*	*V*
V	*F*	*F*	*V*
V	*F*	*V*	*F*
V	*V*	*F*	*F*
V	*V*	*V*	*V*

L'algebra di Boole introduce tre operatori di base AND, OR e NOT tramite i quali può essere espressa qualsiasi funzione logica.

- prodotto logico *a* **AND** *b* *a* Ù *b* $a \cdot b$
- somma logica *a* **OR** *b* *a* Ú *b* $a + b$
- negazione **NOT** *a* Ø *a* $\overline{a}$
- vero **true** ⊤ 1
- falso **false** ⊥ 0

Le precedenti sono tre notazioni diverse per esprimere i tre operatori logici, la prima è una notazione "linguistica" ed è usata in molti linguaggi di programmazione, la seconda è la notazione matematica, la terza è la notazione aritmetica che enfatizza la similarità tra l'algebra booleana e l'algebra dei numeri.

Nella logica le variabili logiche sono chiamate anche proposizioni semplici. Mentre le espressioni booleane che contengono operatori AND, OR e NOT sono chiamate anche proposizioni composte.

Esempio di proposizione semplice o variabile logica: *a*
Esempio di proposizione composta o espressione logica: *a* **AND NOT** *b* **OR** *c*

L'esempio precedente è ambiguo perché potrebbe essere interpretato nei modi seguenti:

(a **AND** (**NOT** b)) **OR** c
a **AND** ((**NOT** b) **OR** c)
a **AND** (**NOT** (b **OR** c))

Per risolvere l'ambiguità o si usano obbligatoriamente le parentesi oppure si assume un ordine di priorità degli operatori, quello usuale è NOT, AND, OR. Questo vuol dire che nel valutare una espressione prima si valutano i NOT poi gli AND e poi gli OR, questo corrisponde alla prima interpretazione riportata. Questo è anche l'ordine con cui si risolve comunemente l'ambiguità nelle espressioni algebriche numeriche con gli operatori -, · e + infatti l'espressione a · –b + c viene comunemente interpretata come (a · (- b)) + c.

Definiamo ora tramite le tabelle di verità gli operatori di base:

2.1 *Operatori AND, OR e NOT*

A **AND** *B* è una funzione logica che è vera solo se entrambi gli operandi A e B sono veri.

A	*B*	*A* **AND** *B*
F	*F*	*F*
F	*V*	*F*
V	*F*	*F*
V	***V***	***V***

A **OR** *B* è una funzione logica che è vera solo se almeno uno dei due operandi A o B è vero.

A	*B*	*A* **OR** *B*
F	*F*	*F*
F	*V*	***V***
V	*F*	***V***
V	***V***	***V***

NOT *A* è una funzione logica che inverte il valore logico del suo operando.

A	**NOT** *A*
F	***V***
V	***F***

Questi operatori possono essere combinati per scrivere delle espressioni logiche:

A **AND** (*B* **OR** (**NOT** *A*))

(**NOT** *X*) **OR** *X*

Nelle espressioni logiche si possono usare anche due costanti logiche **true** e **false** ad indicare il valore logico vero e il valore logico falso.

A **AND** (B **OR true**)

Esattamente come per gli operatori aritmetici +, -, x, / gli operatori logici hanno delle proprietà che permettono di manipolare le espressioni logiche lasciando invariato il valore logico della espressione.

Proprietà:

Simmetrica

A **AND** B = B **AND** A	A B = B A
A **OR** B = B **OR** A	A+B = B+A

Associativa

A **AND** (B **AND** C) = (A **AND** B) **AND** C	A (B C) = (A B) C
A **OR** (B **OR** C) = (A **OR** B) **OR** C	A + (B + C) = (A + B) + C

Elemento neutro

A **AND true** = A	A 1 = A
A **OR false** = A	A + 0 = A

Elemento inverso

A **AND NOT** A = **false**	$A\ \overline{A} = 0$
A **OR NOT** A = **true**	$A + \overline{A} = 1$

Distributiva

A **AND** (B **OR** C) = (A **AND** B) **OR** (A **AND** C)	A (B + C) = A B + A C
A **OR** (B **AND** C) = (A **OR** B) **AND** (A **OR** C)	A + (B C) = (A + B) (A +C)

Idempotenza

A **AND** A = A	A A = A
A **OR** A = A	A + A = A

Assorbimento

A **OR** (A **AND** B) = A	A + (A B) = A
A **AND** (A **OR** B) = A	A (A + B) = A
A **AND false = false**	A 0 = 0
A **OR true = true**	A + 1 = 1

De Morgan

NOT (A **AND** B) = (**NOT** A) **OR** (**NOT** B)	$\overline{A\ B} = \overline{A} + \overline{B}$
NOT (A **OR** B) = (**NOT** A) **AND** (**NOT** B)	$\overline{A + B} = \overline{A}\ \overline{B}$

Doppia negazione

NOT NOT A = A	$\overline{\overline{A}} = A$

Sulla sinistra le proprietà sono espresse nella notazione “linguistica” mentre a destra le stesse proprietà sono espresse nella notazione aritmetica. SI noti proprio come molte proprietà siano comuni con l'algebra dei numeri.

Queste proprietà permettono appunto di semplificare una espressione logica, vediamo un esempio:

NOT (A **OR** (B **AND NOT** A)) = -- prop. distributiva
NOT ((A **OR** B) **AND** (A **OR NOT** A)) = -- elemento inverso OR
NOT ((A **OR** B) **AND** true) = -- elemento neutro AND
NOT (A **OR** B) = -- De Morgan
NOT A **AND NOT** B

Nell'algebra di Boole sono stati introdotti anche altri operatori:

2.2 Operatore XOR

A **XOR** *B* è una funzione logica che è vera solo se solo uno dei due operandi A o B è vero.

A	*B*	*A* **XOR** *B*
F	*F*	*F*
F	*V*	***V***
V	*F*	***V***
V	*V*	*F*

A **XOR** B = (**NOT** A **AND** B) **OR** (A **AND NOT** B)

Proprietà:
A **XOR true** = -- definizione
(NOT A AND true) OR (A AND NOT true) = -- elem. neutro AND
(NOT A) OR (A AND false) = -- elem. assorbente AND
(NOT A) OR false = -- elem. neutro OR
NOT A
A **XOR false** = A
((A **XOR** B) **XOR** B) = A

2.3 Operatore "® " (implica)

A ® B (si legge "A implica B" oppure "se A allora B" oppure "B se A") è vero se e solo se "A è falso oppure A e B sono veri" (forma asserita), o analogamente A ® B è falsa solo quando "A è vero e B è falso" (forma negata).
Vediamo come si può derivare l'espressione di A® B usando gli operatori di base partendo dalla forma negata:

(A ® B) = **NOT** (A **AND NOT** B) = -- per De Morgan
(**NOT** A) **OR** (**NOT NOT** B) = -- per la doppia negazione
(**NOT** A) **OR** B

Lo stesso risultato si può ottenere partendo dalla forma asserita:

(A ® B) = (**NOT** A) **OR** (A **AND** B) = -- prop. distributiva
(NOT A OR A) AND (NOT A OR B) = -- elemento inverso OR
true AND (NOT A OR B) = -- elemento neutro AND
NOT A **OR** B

La tabella di verità di questo operatore è:

A	*B*	*A ® B*
F	*F*	*V*
F	*V*	*V*
V	*F*	*F*
V	*V*	*V*

Proprietà:
(A **OR** B) ® C = (A ® C) **AND** (B ® C)
A ® (B **AND** C) = (A ® B) **AND** (A ® C)
(**NOT** A) ® B = (**NOT** B) ® A

2.4 Operatore " « "

A « B (si legge "A se e solo se B") è vero solo se "A implica B e B implica A", quindi si può scrivere:

A « B = (A ® B) **AND** (B ® A) = (NOT A OR B) AND (NOT B OR A) = -- prop. distributiva
((NOT A OR B) AND (NOT B)) OR ((NOT A OR B) AND A) = --prop. distributiva
((NOT A AND NOT B) OR (B AND NOT B)) OR ((NOT A AND A) OR (B AND A)) = --elem. inverso
((NOT A AND NOT B) OR false) OR (false OR (B AND A)) = -- elem. neutro OR
(**NOT** A **AND NOT** B) **OR** (A **AND** B)

La tabella di verità è la seguente:

A	*B*	*A « B*
F	*F*	*V*
F	*V*	*F*
V	*F*	*F*
V	*V*	*V*

In pratica controlla se i due operandi hanno lo stesso valore logico.

Proprietà:
A « B = B « A
(**NOT** A) « (**NOT** B) = A « B
A « A **= true**
A « **NOT** A **= false**

2.5 Esempio

Dobbiamo organizzare un incontro tra quattro persone A, B, C e D si hanno però i seguenti vincoli:

- A e B non possono esserci entrambi;
- se è presente B allora ci deve essere anche C;
- se è presente C allora deve esserci D e A.

L'algebra di Boole ci può aiutare a capire meglio la situazione.

Usiamo 4 variabili logiche A,B,C e D che indicano se la persona deve essere invitata (true) o non invitata (false).
I vincoli possono essere espressi usando l'algebra di Boole:

- A **XOR** B = (A **AND NOT** B) **OR** (**NOT** A **AND** B)
- B ® C = **NOT** B **OR** C
- C ® (D **AND** A) = **NOT** C **OR** (D **AND** A)

Questi tre vincoli devono essere tutti verificati quindi sono tutti in **AND,** dobbiamo risolvere quindi la seguente equazione logica:

((A **AND NOT** B) **OR** (**NOT** A **AND** B)) **AND** (B ® C) **AND** (C ® (D **AND** A)) = **true**

cioè determinare i valori logici di A, B, C e D per rendere vera l'uguaglianza.
Riscriviamo la prima parte dell'equazione usando la notazione aritmetica che è più compatta:

$(A\overline{B} + \overline{A}B)$ **$(\overline{B}$ + C) $(\overline{C}$ + DA)** =
$(A\overline{B} + \overline{A}B)$ $(\overline{B}\overline{C} + \overline{B}DA +$ **$C\overline{C}$** + CDA) =
$(A\overline{B} + \overline{A}B)$ $(\overline{B}\overline{C} + \overline{B}DA +$ **0 + CDA**) =
$(A\overline{B} + \overline{A}B)$ $(\overline{B}\overline{C} + \overline{B}D\overline{A} + CDA)$ =
$A\overline{B}$**$\overline{B}$**$\overline{C} + \overline{A}$**$\overline{B}\overline{B}$**$DA +$ **A**$\overline{B}$CD**A** + A**$\overline{B}\overline{B}$**$C + \overline{A}$**$\overline{B}\overline{B}$**$DA +$ **$\overline{A}$**BCD**A** =
$A\overline{B}\overline{C} + A\overline{B}D +$ **$A\overline{B}$CD + 0 + 0 + 0** =
$A\overline{B}\overline{C} + A\overline{B}D + A\overline{B}CD = A\overline{B}\overline{C} + A\overline{B}D(1 + C) = A\overline{B}\overline{C} + A\overline{B}D$ =
A **AND NOT** B **AND** (**NOT** C **OR** D) = A **AND NOT** B **AND** (C → D)

L'espressione è vera per:
A = **true**, B = **false**, C = **false** D = **true**/**false**
oppure per
A = **true**, B = **false**, C = **true**/**false** D = **true**

Quindi si deve invitare A, non invitare B e se si invita C si deve invitare D.

2.6 Operazioni logiche e numeri binari

Le operazioni logiche AND, OR, NOT, XOR possono essere applicate anche ai numeri binari, in questo caso si parla di "operazioni logiche bit a bit" nel senso che l'operatore logico viene applicato ai bit che si trovano nella stessa posizione.
Esempi:

01001011 AND
10001111 =
00001011

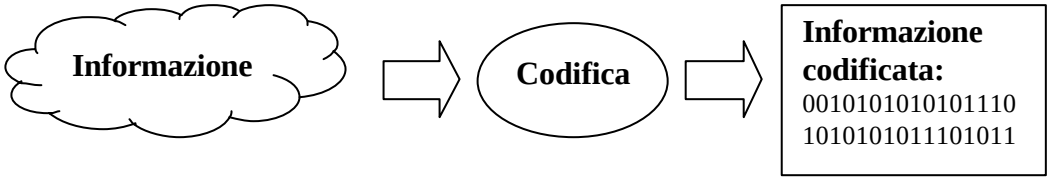
01001011 OR
10110111 =
11111111

NOT 01010010 =
10101101

01001011 XOR
10110111 =
11111100

3 Codifica delle Informazioni

I calcolatori elettronici memorizzano e gestiscono esclusivamente dati rappresentati con una sequenza di bit quindi è necessario per ogni tipo di informazione determinare un metodo per codificare l'informazione in binario.



La codifica deve essere reversibile cioè una volta codificata l'informazione deve poter essere decodificata e ottenere nuovamente l'informazione di partenza.

Le informazioni possono essere classificate in due grandi categorie:

- le *informazioni semplici* che sono numeri e sequenze di caratteri;
- le *informazioni complesse* che sono le immagini, i suoni, i video.

3.1 Codifica di informazioni semplici

- **Numeri**
  - **Interi senza segno** sono codificati come numeri binari
    - Codifica con 8 bit: 0 … 255
    - Codifica con 16 bit: 0 … 65.535
    - Codifica con 32 bit: 0 … 4.294.967.295
  - **Interi con segno** sono codificati in modo particolare (detto complemento a 2)
    - Codifica con 8 bit: -128 … 127
    - Codifica con 16 bit: -32.768 … 32.767
    - Codifica con 32 bit: -2.147.483.648 … 2.147.483.647
  - **Numeri con la virgola**, sono codificati in un modo simile alla notazione scientifica, ad esempio il numero 23,645 può essere scritto come +0,23645 10^3. L'idea è appunto di codificare il numero codificando separatamente l'esponente, il segno e la mantissa (il numero 0,23645)
    - Codifica con 32 bit, 1 bit per il segno(+/-), 8 bit per l'esponente e 23 bit per la mantissa
      - Più grande numero rappresentabile ± 3.402823466 10^{+38}
      - Più piccolo numero rappresentabile ± 1.17549435110^{-38}
    - Codifica con 64 bit, 1 bit per il segno (+/-), 11 bit per l'esponente, 52 bit per la mantissa
      - Più grande numero rappresentabile ± 1.7976931348623158 10^{+308}
      - Più piccolo numero rappresentabile ± 2.2250738585072014 10^{-308}

- **Caratteri**
  I caratteri sono codificati con un numero binario di 8 o 16 bit. Le codifiche a 8 bit (ASCII e EBCDIC) possono codificare al massimo 256 caratteri diversi che non sono sufficienti per lingue come il cinese o il giapponese nei quali i caratteri possibili sono moltissimi. Per questo è stata introdotta la codifica UNICODE che usa 16 bit per carattere con un massimo di 65536 caratteri.
  Le codifiche sono date attraverso delle tabelle che indicano per ogni carattere con che numero debba essere codificato. La codifica più frequente per i PC è quella ASCII (American Standard Code for Information Interchange) che però standardizza solo i primi 128 caratteri (7bit) dove sono codificati:
  - numeri
    ‘0’ → 48 ‘1’→ 49 ‘2’→ 50 … ‘9’→ 57
  - lettere maiuscole
    ‘A’ → 65 ‘B’→ 66 ‘C’→ 67 … ‘Z’ → 90
  - lettere minuscole
    ‘a’ → 97 ‘b’→ 98 ‘c’→ 99 … ‘z’ → 122
  - segni di interpunzione
    ‘.’ → 46 ‘,’ → 44 ‘:’ → 58 ‘;’ → 59 ‘?’ → 63 …
  - spazio ‘ ’ → 32

  Ad esempio data una lettera maiuscola codificata in ASCII come posso fare per trasformarla nella corrispondente lettera minuscola?

  Si osserva che l’ordine dei codici ASCII delle lettere non cambia quindi se la lettera è maiuscola e tolgo dal valore del codice 65 (codice della A maiuscola) ottengo un valore che 0 per A, 1 per B, 2 per C etc. e rappresenta la posizione della lettera nell’alfabeto. Se a questo valore ottenuto ci sommo 97 (codice della a minuscola) ottengo proprio il codice della minuscola della lettera di partenza.
  Quindi:

  CodMinuscola = (CodMaiuscola – 65) + 97

  solo se CodMaiuscola è maggiore uguale a 97 e minore uguale a 90 (è proprio una lettera maiuscola)

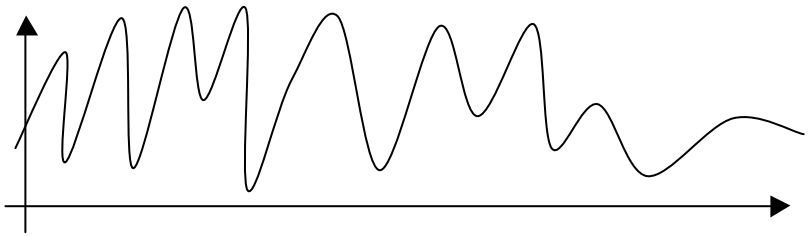
  I codici oltre il 127 non sono stati standardizzati e varie estensioni sono state proposte, in questa parte della tabella di codifica sono presenti i codici specifici per le varie lingue (es lettere accentate).

  Per una discussione approfondita sulla questione dei caratteri si veda
  http://www.cs.tut.fi/~jkorpela/chars.html

3.2 Codifica di informazioni complesse

3.2.1 Codifica di suoni (audio)

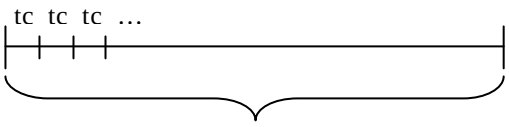
I suoni sono rappresentabili come segnali continui del tempo:



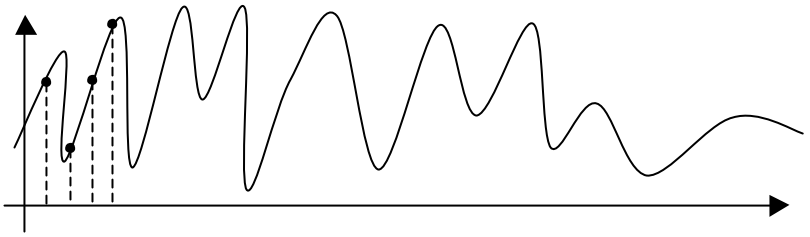
Come possiamo rappresentare questo segnale in binario?
Osserviamo che abbiamo a che fare con due grandezze continue, il tempo e l'intensità del suono, che dobbiamo trasformare in numeri interi per essere codificati in numeri binari.

Per quanto riguarda il tempo, si attua quello che viene chiamato "campionamento", cioè viene stabilito un tempo detto di campionamento (in genere molto basso al di sotto dei millesimi di secondo) ed il segnale viene codificato solo negli istanti che sono multipli di questo tempo di campionamento, quindi se tc=0,125ms è questo tempo di campionamento, il segnale viene considerato a tc, 2 tc, 3 tc, 4tc, 5tc, … 8000 tc, … cioè a 0,125ms , 0,25ms, 0,375 ms, 0,5ms, 0,625ms, … 1s…
L'informazione che viene estratta dal segnale ad ogni istante di campionamento è detta campione.

Quanti campioni si possono estrarre in un secondo se tc è il tempo di campionamento?

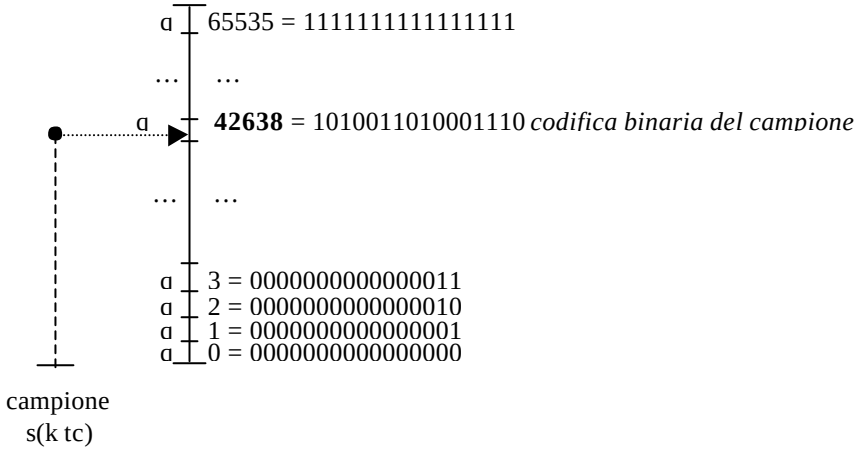


In un secondo ci sono 1/tc campioni, quindi per tc = 0,125ms = 0,125 10^{-3} s ci sono 1 / (0,125 10^{-3}) = 8000 campioni, questo numero indica la frequenza di campionamento e si esprime in Hertz (Hz) quindi la frequenza di campionamento è di 8 kHz = 8000Hz.
Tanto più alta è la frequenza di campionamento tanto maggiori sono i campioni estratti dal segnale tanto più è fedele la rappresentazione del segnale che otteniamo. Per i segnali audio di alta qualità, detta qualità CD, la frequenza di campionamento è di 44,1kHz, mentre ad esempio per la telefonia bastano 8kHz.



Quindi ora abbiamo trasformato il segnale in una sequenza di campioni però ancora non possiamo trasformare il segnale in binario, dobbiamo codificare in binario il singolo campione come facciamo? Il metodo più semplice è di considerare che il segnale non può avere più di una certa intensità e non può essere minore di 0, a questo punto fissiamo il numero di bit che vogliamo

utilizzare per rappresentare l'intensità del segnale, diciamo siano 16 bit, con 16 bit posso codificare $2^{16} = 65536$ possibilità. A questo punto dividiamo il massimo per 65536 e otteniamo il fattore di quantizzazione q, il valore del segnale s(k tc) può essere trasformato in binario dividendo s() per q e codificando in binario la parte intera.



Ora possiamo codificare in binario in segnale audio, i due parametri fondamentali per questa codifica sono, la *frequenza di campionamento* e il *numero di bit usati per codificare un campione*, tanto maggiori sono questi due numeri tanto più fedele è la rappresentazione del segnale.

La codifica del segnale audio è data da una sequenza di numeri binari, per esempio con una codifica dei campioni con 8 bit avrei:

00010101 1001100 10010100 01101111 11101110 01111011 ….
campione 1 *campione 2* *campione 3* *campione 4* *campione 5* *campione 6*

Supponiamo di avere un audio di 1 minuto campionato a 44,1kHz e codificato con 16 bit per campione quanto spazio di memorizzazione occupa?

1 minuto = 60 secondi, in un secondo ci sono 44100 campioni quindi ci sono un totale di 60 44100 = 2.646.000 campioni in un minuto. Ogni campione occupa 2 byte (16 bit) quindi il numero totale di byte occupati da un minuto di audio sono 2.646.000 2 = 5.292.000 byte = 5,04 MB

E se invece l'audio fosse campionato a 8kHz e usasse 8 bit per campione quanto occuperebbe?

1 minuto = 60 secondi, in un secondo ci sono 8000 campioni quindi ci sono 60 8000 = 480.000 campioni in un minuto e siccome un campione occupa un byte l'audio occupa 480.000 byte = 468,75 kB cioè il 9% della codifica precedente. Comunque la prima codifica è molto più fedele all'originale della seconda codifica che è da considerarsi di scarsa qualità.

Questo nel caso di segnali audio "mono", nel caso di segnali audio "stereo" l'occupazione viene raddoppiata dovendo codificare sia il canale destro che il canale sinistro. La codifica avviene alternando i campioni del canale destro e di quello sinistro:

00010101 00110011 1001100 10010100 01101111 11101110 01111011 11001010….
1 destro *1 sinistro* *2 destro* *2 sinstro* *3 destro* *3 sinistro* *4 destro* *4 sinistro*

E' da osservare che lo spazio occupato per la codifica di un audio con questa tecnica è sempre lo stesso anche nel caso l'audio sia un minuto di silenzio!!

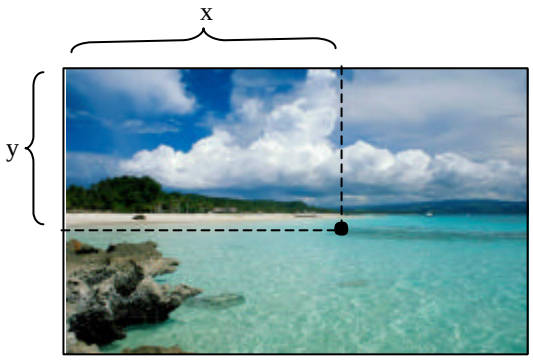
Questo metodo di codifica dell'audio è praticamente quello adottato nei file WAV.

Un audio reale (1,120 secondi, mono, 11,025 kHz, 8 bit)

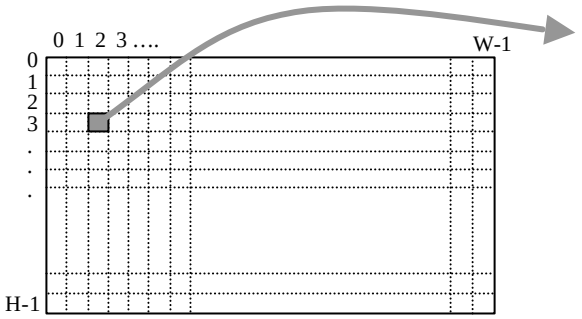
La codifica MP3 è una codifica molto più complessa ma riesce a codificare in molto meno spazio il segnale audio e si basa sul fatto che il nostro orecchio non riesce a percepire tutte le variazioni presenti nel segnale audio. Le codifiche che si basano sui limiti dei nostri sensi si dicono codifiche percettive.

3.2.2 Codifica di immagini

Una immagine può essere vista come un insieme di punti ognuno caratterizzato da un proprio colore e da una posizione (x,y).



Il modo più semplice per codificare un immagine è di dividere l'immagine in una griglia di larghezza W e di altezza H e associare ad ogni elemento della griglia, detto pixel (da picture element), un colore. Se la griglia è abbastanza fitta si ottiene una rappresentazione fedele dell'originale.



C(2,3) codifica del colore del pixel di posizione: x=2 y=3

La codifica del colore la affronteremo più avanti assumiamo di avere un modo per ottenere la codifica del colore di ogni pixel: C(x,y). La codifica dell'immagine la si ottiene mettendo in sequenza le codifiche dei colori dei pixel della prima riga (da sinistra verso destra) poi quelli della seconda riga, della terza riga e così via fino a codificare tutta l'immagine.

C(0,0) C(1,0) C(2,0) … C(W-1,0) C(0,1) C(1,1) C(2,1) … C(W-1,1) C(0,2) C(1,2) C(2,2) … C(W-1,2)…

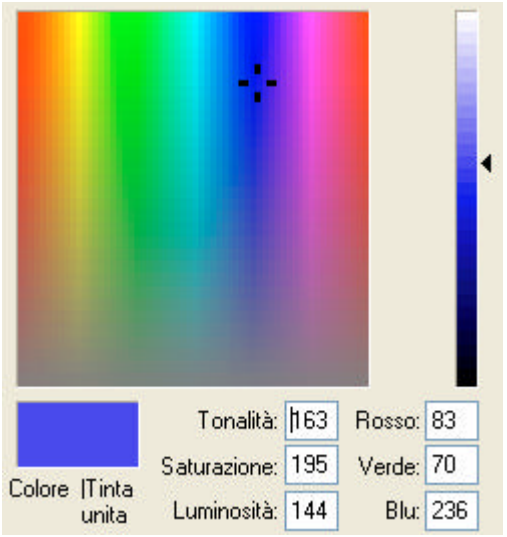
Prima riga | Seconda riga | Terza riga

… C(0,H-1) C(1,H-1) C(2,H-1) … C(W,H-1)

Ultima riga

La codifica del colore del pixel occupa un certo numero di bit: 24, 16, 8, 1 bpp (bit per pixel)
Quindi una immagine larga W pixel, alta H pixel e con il colore codificato a 16bpp=2 byte occupa W x H x 2 byte.

Vediamo ora come codificare il colore di un pixel. Un colore può essere rappresentato tramite tre componenti: rossa, verde e blu o RGB (Red Green Blu). Ogni componente ha una intensità che può variare da 0 (assente) a una intensità massima. Assumiamo che l'intensità delle componenti sia codificata usando 8 bit quindi queste potranno assumere valori da 0 a 255.
Il colore nero ad esempio è codificato con tutte le componenti a 0 (R=0,G=0,B=0), mentre il colore bianco è codificato da tutte le componenti al massimo (255,255,255)



Quindi usando 3 byte (24 bit), un byte per ogni componente, si riesce a codificare il colore. I colori rappresentabili con questa codifica sono 2^{24} = 16.777.216
La codifica a 24 bit è completa ma può occupare troppa memoria e spesso piccole variazioni delle componenti non sono percettibili per esempio il colore (38, 99, 240) non è percettibilmente differente dal colore (39,99,240) quindi si potrebbero anche utilizzare meno livelli per codificare le componenti. Ad esempio si potrebbero usare 5 bit per il rosso, 6 bit per il verde e 5 bit per il blu per un totale di 16 bit. In questo caso il rosso ed il blu hanno un livello che va da 0 a 31 mentre il verde va da 0 a 63, in questo caso il bianco è codificato con (31,63,31).

Il colore grigio è rappresentabile con tutte e tre le componenti uguali tra loro e quindi l'intensità del grigio è codificata con 8 bit da 0 (nero) a (255) bianco.
Per questo una immagine in toni di grigio è codificabile usando 8 bit per pixel che rappresentano appunto il tono di grigio del pixel. Se invece l'immagine è in bianco e nero (fotocopie, fax) allora basta un solo bit per codificare il colore del pixel 0=nero 1=bianco.

Le codifiche che abbiamo visto sono tutte codifiche dirette del colore esistono anche codifiche indirette.
Supponiamo l'immagine da codificare usi un numero ridotto di colori ad esempio 20 colori.

A questo punto è conveniente costruire una tabella dei colori e assegnare ad ogni colore un numero:

	R	G	B
0 = 00000	120	30	10
1 = 00001	70	80	34
2 = 00010	10	0	255
3 = 00011	5	50	20
…	…	…	…
19 = 10011	10	10	10
20 = 10100	67	30	167

Nella codifica dell'immagine invece di codificare il colore direttamente si usa la tabella e il colore viene rappresentato con l'indice della riga della tabella che lo contiene. Per codificare questo indice sono necessari 5 bit (0..31) perché con 4 bit avrei potuto codificare fino a 16 elementi. Quindi un pixel viene rappresentato usando 5 bit invece di 24 bit.

Ad esempio la sequenza di bit:

00000 00000 00001 10011 00001 00010 …

è la codifica della sequenza di colori

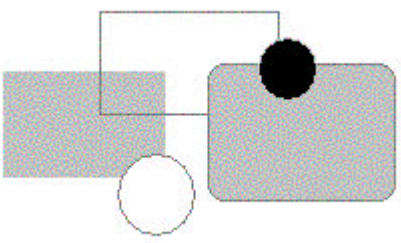
(120,30,10) (120,30,10) (70,80,34) (10,10,10) (70,80,34) (10,0,255) …

Supponiamo di avere una immagine 640 x 480 vediamo quanto spazio occupa variando la codifica del colore:

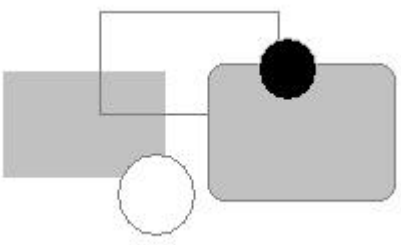
true color (24 bpp) 640 x 480 x 3 byte = **900 kB**
16 bpp 640 x 480 x 2 byte = **600 kB**
toni di grigio (8 bpp) 640 x 480 x 1 byte = **300 kB**
max 32 colori (5bpp) 640 x 480 x 5 bit = 1.536.000 bit = 192.000 byte = **187,5 kB**
bianco e nero (1 bpp) 640 x 480 x 1 bit = 307.200 bit = 38.400 byte = **37,5kB**

Anche in questo caso (come per il silenzio nell'audio) l'occupazione è indipendente dal contenuto, occupa lo stesso spazio una fotografia digitalizzata rispetto ad una immagine tutta di uno stesso colore. Questa codifica praticamente è quella usata dai file BMP o bitmap.

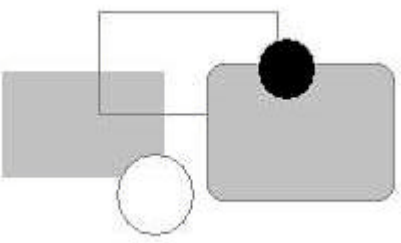
Esistono anche altri modi per codificare le immagini, molto più complessi, per i quali l'occupazione dell'immagine codificata dipende dal contenuto come ad esempio la codifica GIF, TIFF o JPEG (e molte molte altre).
In particolare la codifica JPEG è stata studiata per codificare immagini fotografiche e si differenzia dalle altre in quanto è una codifica percettiva, l'immagine codificata non è esattamente l'originale ma in realtà è percettibilmente "uguale" all'originale se la si usa per immagini non fotografiche si possono avere degli strani effetti.



Codifica GIF
(8bpp)



Codifica BMP
(24 bpp)



Codifica JPEG
(24 bpp)

3.2.3 Codifica di filmati (video)

Anche i video possono essere codificati in binario, un filmato è formato da una sequenza di immagini (fotogrammi o frame) che se fatti vedere in sequenza ad una velocità sufficientemente elevata ci danno l'impressione che la sequenza sia continua. La velocità tipica è di 25 frame al secondo.

Quindi ilo metodo più semplice per codificare un video (solo la parte visuale) è di codificare in sequenza le varie immagini:

$I_0 \quad I_1 \quad I_2 \quad I_3 \quad \ldots \quad I_{25} \quad I_{26} \ldots$

Si vede subito però che questo tipo di codifica è estremamente dispendioso dal punto di vista di occupazione di spazio. Nel caso si abbia una velocità di 25 fps (frame per second) in 10 minuti di video ci sono 10 x 60 x 25 = 15.000 immagini e se ogni immagine è da 640 x 480 x 8 bpp = 300kB L'occupazione totale è di 15.000 x 300kB = 4.394,53MB = 4,29 GB

Per questo sono stati studiati molti criteri di codifica che riducono sensibilmente l'occupazione di spazio. Una tecnica possibile è basata sulla osservazione che tra due immagini successive non c'è molta differenza per questo può essere conveniente codificare la differenza tra due immagini che dovrebbe occupare meno spazio.

$I_0 \quad I_1-I_0 \quad I_2-I_1 \quad I_3-I_2 \ldots$

Anche per i video la codifica percettiva permette di risparmiare spazio come accade per i video codificati in MPEG.