Teoria dell'Informazione e Crittografia

UniShare

Davide Cozzi
@dlcgold

Indice

Capitolo 1

Introduzione

Questi appunti sono presi a lezione. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: `https://github.com/dlcgold/Appunti`.

Capitolo 2

Introduzione agli argomenti del corso

Si ha una sorgente che emette messaggi e li vuole mandare tramite un canale di comunicazione, che contiene del rumore che agisce sui messaggi e li rovina. Il destinatario per capire che il messaggio è rovinato ha varie tecniche. Una prima cosa che potrebbe fare è chiedere di rimandare il messaggio ma sarebbe meglio correggere *in loco* e si hanno algoritmi per farlo (tra cui lo **schema di Hamming** usando il modello del **rumore bianco**).
Un'altra tematica è la codifica stessa del sorgente, comprimendo flussi di dati, senza perdere informazioni.
Si parlerà anche dei canali di comunicazione, delle capacità e dei **teoremi di Shannon**.
Si vedranno poi le basi della crittografia, i crittosistemi storici, vari standard etc...

Capitolo 3

Teoria dell'informazione

Si hanno due sottoparti principali:

- **teoria dei codici**, per individuare e correggere gli errori. Si studia il canale di trasmissione cercando di contrastare il rumore che c'è nel canale
- **teoria dell'informazione**, in cui il focus è la sorgente delle informazioni

Vediamo il classico schemino della teoria dell'informazione:

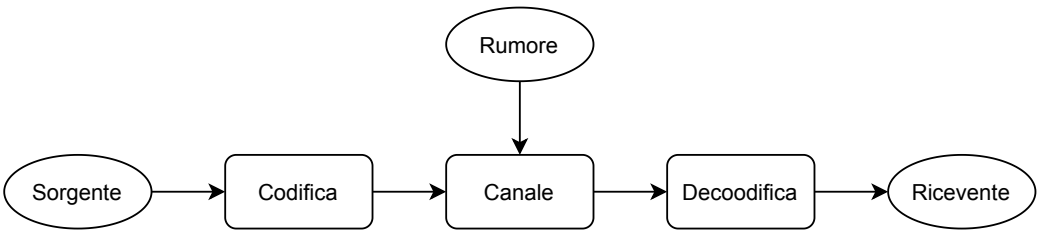


Figura 3.1: Schema di un sistema generale di comunicazione tipico della teoria dell'informazione

Avendo:

- la **sorgente** che produce segnali, dei simboli, che potrebbero essere continui (come la corrente), anche se noi li assumeremo come simboli di un alfabeto finito, avendo quindi una **sorgente discreta**
- i simboli, per poter essere spediti all'interno di un canale, vanno codificati, avendo una parte di **codifica**

- una volta codificati i simboli vanno nel **canale di trasmissione**, dove si ha del **rumore**. Tale *rumore* prende un simbolo di quelli inseriti e lo cambia
- dal canale esce o il simbolo che è entrato o il simbolo modificato dal rumore e, tipicamente, non è immediatamente utilizzabile ma deve passare per una fase di **decodifica**
- il simbolo decodificato arriva al **destinatario**

Si hanno alcune assunzioni sulla sorgente:

- è **discreta**, i simboli emessi appartengono ad un alfabeto finito. Normalmente tali simboli sono $S = \{s_1, s_2, \ldots, s_q\}$
- i simboli vengono emessi uno alla volta ad ogni **colpo di clock**. Non si ha mai che in un colpo di clock non escano simboli o che ne escano più di uno solo
- la sorgente è **senza memoria (*memoryless*)**, avendo che i simboli già usciti non influenzano per nulla il simbolo che sta per uscire. Ogni simbolo che esce non tiene conto del passato, *è come se fosse il primo*
- è **probabilistica e randomizzata**. Si ha quindi che i simboli $S = \{s_1, s_2, \ldots, s_q\}$ escono con le probabilità $(p_1, p_2, \ldots, p_q)$. Deve valere che, ovviamente, che $p_i \in [0, 1], \forall i = 1, \ldots, q$. Si ha inoltre che le varie probabilità, nel loro insieme, devono formare una distribuzione di probabilità, avendo che:

  $$\sum_{i=1}^{q} p_i = 1$$

  Potrei avere simboli con probabilità nulla di comparire ma nella pratica non è qualcosa di sensato. La sorgente la costruisco o da zero (e a quel punto un simbolo con probabilità nulla non lo metterei) o ho una sorgente che devo studiare (e qui potrei avere simboli con probabilità bassissime se non nulle, in tal caso bisognerebbe rivalutare l'assunzione dei simboli di quella sorgente). Possiamo quindi meglio dire che $p_i \in (0, 1], \forall i = 1, \ldots, q$.
  D'altro canto vedo se posso avere probabilità pari a 1 per un simbolo ma in tal caso avrei solo quello e non sarebbe interessante. Si ha quindi che:

  $$p_i \in (0, 1), \forall i = 1, \ldots, q$$

$$\sum_{i=1}^{q} p_i = 1$$

Le sorgenti che emettono i simboli secondo uno schema prefissato, deterministico, sono poco interessanti, avendo un comportamento banale

Il concetto di *spedire in un canale* può anche essere generalizzato in altre "idee", come il disco su cui salvo dei dati e il tempo per cui li salvo.
La parte di *codifica e decodifica* può essere approfondita. Nello schema in figura 3.1 ci si è infatti concentrati sul canale, avendo che la codifica serve a fare in modo che il simbolo trasmesso vada bene per essere trasmesso nel canale. La **codifica** è a sua volta suddivisa in due parti:

1. **codifica di sorgente**, che ha come obiettivo rappresentare nel modo più efficiente e compatto i simboli emessi dalla *sorgente*. Si vuole quindi comprimere la sequenza di simboli (messaggi) emessi dalla sorgente, per impegnare meno banda possibile quando andremo a spedire. Si deve considerare che ogni bit in un file compresso è essenziale per permettere di poter recuperare il contenuto compresso

2. **codifica di canale**, che ha quasi uno scopo opposto rispetto alla *codifica di sorgente*, infatti ha come obiettivo quello di contrastare il rumore e per farlo aggiunge ridondanza al messaggio (da qui il discorso sull'obiettivo opposto)

Si cerca quindi di comprimere il più possibile nella prima fase, quella di *codifica di sorgente* e di ridondare il meno possibile nella seconda, quella di *codifica di segnale*.
Per capire meglio quanto detto diamo alcune formalità.

Definizione 1. *Una **codifica** è una funzione cod che prende i simboli della sorgente $S = \{s_1, s_2, \ldots, s_q\}$ e ad ogni simbolo s_i gli assegna una stringa formata coi caratteri di un certo alfabeto Γ, l'**alfabeto della codifica**. Le stringhe di Γ^* sono tutte quelle costruite sull'alfabeto Γ di lunghezza arbitraria e finita, compresa la stringa vuota ε, che posso quindi formare coi simboli di Γ. Quindi ad ogni $s_i \in S$ assegno un $\gamma_i \in Gamma^*$, avendo che:*

$$cod : S \to \Gamma^*$$

generalmente si ha che:

$$|\Gamma| < |\Sigma|$$

e quindi i simboli di Σ *sono mappati da cod in sequenze di simboli di* Γ*, a meno che non si ritenga accettabile il fatto che due o più simboli di* Σ *vengano mappati nellostesso simbolo di* Γ*.*
Più avanti nel corso vedremo casi in cui $|\Gamma| > |\Sigma|$

Si hanno quindi i simboli $S = \{s_1, s_2, \ldots, s_q\}$ che escono con probabilità $(p_1, p_2, \ldots, p_q)$ e che vengono codificati con le stringhe $\gamma_1, \gamma_2, \ldots, \gamma_q$. Le varie $gamma_i$ sono dette **codeword**. Chiamando $l_i = |\gamma_i|$ la lunghezza di tali stringhe si ha che tali stringhe hanno associati i vari $l_1, l_2, \ldots, l_q$.
L'obiettivo quindi della *codifica di sorgente* è quello di minimizzare la lunghezza media L delle stringhe, avendo quindi una media pesata (pesata sulle probabilità):

$$L = \sum_{i=1}^{q} p_i \cdot l_i$$

Tenendo conto delle probabilità, per minimizzare L, si deve, avendo a che fare con termini che sono tutti > 0 (avendo supposto che non si hanno probabilità nulla e avendo che una codeword pari alla stringa vuota ha poco senso), fare in modo che i termini siano tutti il più piccolo possibile. Dato che le probabilità sono date mentre la codifica la sto costruendo, calcolando le codeword e di conseguenza le loro lunghezze, devo fare in modo che se la probabilità è grande la lunghezza deve essere piccola. Se invece la probabilità è piccola posso permettermi una lunghezza più grande. Parto quindi dai simboli con probabilità più grande e inizio a usare codeword più piccole possibili, usando via via quelle più lunghe.
Un'idea simile è usata nel *codice Morse* dove le lettere meno comuni hanno le sequenze più lunghe di punti, linee e spazi (avendo una codifica ternaria). La lettera più comune, la "e", ha infatti solo con un punto, la codifica più breve mentre le meno comuni hanno sequenze multiple di punti, linee e spazi che le separano (e gli spazi contano nella lunghezza di queste codeword).
Noi non sappiamo in anticipo che messaggi verranno prodotti dalla sorgente e quindi le codeword vanno studiate passo a passo, valutando i simboli più probabili per associare le codeword più brevi e i meno probabili per le codeword più lunghe.
Analizziamo meglio i codici, le codeword. Possono essere:

1. *a lunghezza fissa*, ovvero si ha che $l_1 = l_2 = \cdots = L_q$
2. *a lunghezza variabile*, avendo che ogni codeword può avere lunghezza diversa

Ne segue quindi che il discorso di minimizzare L ha senso solo in presenza di *codeword a lunghezza variabile* (potendo decidere per ogni simbolo che

codeword associare), avendo la **codifica a lunghezza variabile**.
Con *codeword a lunghezza fissa* avrei tutte le l_i uguali e quindi avrei, avendo $l_i = l, \forall i$:

$$L = \sum_{i=1}^{q} p_i \cdot l_i \sum_{i=1}^{q} p_i \cdot l = l \cdot \sum_{i=1}^{q} p_i = l \cdot 1 = l$$

avendo, come facilmente intuibile, che la lunghezza media è la lunghezza fissa stessa. Si hanno codifiche a lunghezza fissa, come banalmente numeri a 64bit etc... in tal caso si parla di **codici a blocchi**.
Usando codifiche a lunghezza fissa si hanno anche esempi interessanti come quello del *codice pesato*, detto **codice pesato 01247**. Il nome deriva dal fatto che si possono codificare le cifre da 0 a 9 (da 1 a 9 con poi lo 0 dopo il 9) sotto forma di stringhe di 5 bit usando i pesi 0,1,2,4,7 associati a ciascun bit. Vediamo la tabella con la codifica di questo codice:

	0	1	2	4	7
1	1	1	0	0	0
2	1	0	1	0	0
3	0	1	1	0	0
4	1	0	0	1	0
5	0	1	0	1	0
6	0	0	1	1	0
7	1	0	0	0	1
8	0	1	0	0	1
9	0	0	1	0	1
0	0	0	0	1	1

Si nota che ogni codeword ha sempre 3 bit pari a 0 e due bit pari a 1, avendo un **codice 2-su-5**. Banalmente i pesi si associano ai numeri 0,1,2,4,7 in modo tale che essi, sommati, formino il numero voluto (ad esempio per 1 avrò i pesi su 0 e 1, per 9 su 2 e 7 etc...). L'unico caso è il caso dello 0, che non può essere ottenuto come somma di due pesi (spesso si hanno nei codici casi speciali da gestire a parte). Per lo 0 viene quindi presa una codeword non usata per altri numeri e quindi l'unica scelta possibile è avere i pesi su 4 e 7 (visto che farebbe 11).
Su un totale ci 5 bit, avendo due bit a 1 e tre bit a 0, posso avere un numero di codeword pari a:

$$n = \binom{5}{2} = 10$$

avendo che i 5 bit sono associati ai 5 elementi dove 1 segnala che "sto usando quel peso", prendendo quindi i sottoinsiemi di due elementi a partire da un

insieme di cinque elementi, ovvero "in quanti modi posso formare sottoinsiemi che contengono due elementi a partire da un insieme di cinque elementi" o detto altrimenti "quanti sono i modi in cui posso disporre due uni all'interno di una stringa di cardinalità cinque".
In un linguaggio di programmazione privo di una struttura dati dedicata posso simulare un insieme di questo tipo tramite un vettore di bit (con 1 se l'elemento associato all'indice c'è).
Il codice a barre è detto ***codice 39*** *ed è un* ***codice 3-su-9***.
In merito alla **decodifica** si ha che anch'essa sarà di due tipi:

1. **decodifica di canale**, vedendo e c'è stato un errore di trasmissione ed eventualmente correggendolo in automatico se il codice mi consente di farlo

2. **ulteriore decodifica** che non è esattamente una *codifica di sorgente* ma quanto una *trasformazione*, dove le *codeword* vengono trasformate nel formato leggibile dal **ricevente**

3.1 Codici per individuare errori

Ci concentriamo ora sulla *codifica di canale* ignorando per ora la *codifica di sorgente*, avendo come obiettivo l'aggiunta di ridondanza a simboli, che si suppongono già codificati con codeword, in modo tale che in queste codeword, spedite nel canale dove eventualmente si possono avere modifiche causate dal rumore, vengano eventualmente riconosciuti (ed eventualmente corretti) errori in fase di *decodifica*.
Parlando di codici per individuare errori solitamente, nei disorsi, si ha che $|\Gamma| < |\Sigma|$ Qualora il ricevente con la sua decodifica si accorga che è successo qualcosa ma non si è in grado di correggere quel qualcosa si hanno i cosiddetti **codici per individuare gli errori (*error detection codes*)**. Nel caso in cui il ricevente con la sua decodifica si accorga dell'errore ci si chiede anche se può correggerlo autonomamente senza chiedere che la sorgente spedisca nuovamente il messaggio. Non sempre questa cosa si può fare ma quanto accade si parla di **codici a correzione d'errore (*error correction codes*)**.

3.1.1 Controllo di parità semplice

Vediamo come capire se un messaggio ricevuto è valido.
Si supponga di spedire un pacchetto di n bit (ma potrebbe essere qualsiasi altra cosa ma per praticità prendiamo un bit) nel canale e che da esso esca un certo pacchetto sempre di n bit (per il rumore potrebbe non essere lo stesso).

Definizione 2. *Definiamo il* ***controllo di parità****.*
Avendo una sequenza di n *bits in cui si ha* $n-1$ *bits, dette* ***cifre di messaggio*** *di* ***messaggio vero****, che chiameremo* ***msg*** *e un bit che è la* ***cifra di controllo****, che chiameremo* ***check****. Le cifre di messaggio si indicano con* $\circ$ *mentre la cifra di controllo con* $\times$ *e quindi il messaggio è del tipo:*

$\circ$	$\circ$	$\circ$	$\circ$	$\cdots$	$\circ$	$\times$

avendo $n-1$ $\circ$ *e un solo* $\times$ *(che potrebbe anche non essere in fondo, basta avere coscienza della posizione nel pacchetto, concordando la cosa tra mittente e ricevente).*
Si ha che:

- *chi spedisce ha le cifre di messaggio e deve calcolare la cifra di controllo*
- *chi riceve controlla che la cifra di controllo sia coerente con le cifre di messaggio*

*Nell'****error detection code*** *il ricevente è solo in grado di capire che la sequenza non è valida ma per farlo bisogna assumere di avere limitazioni nella sequenza di* n *bit che è entrata nel canale. Questa limitazione è che gli* n *bit entranti nel canale siano una* ***codeword valida****, avendo che, preso un sottoinsieme* M *di tutto l'insieme di* n *bit, ovvero* $M \subseteq \{0,1\}^n$*,* M *è un insieme di codeword valide. Quindi solo un messaggio appartenente a* M *può entrare nel canale. Fatta questa premessa, quando esce un messaggio, ho che, a causa del rumore, questo messaggio viene rovinato, non avendo più un messaggio valido (cosa che viene capita dal ricevente). Purtroppo può succedere che il rumore trasformi un messaggio valido in un altro messaggio valido ma non considereremo questa opzione per ora.*
Nel ***controllo di parità semplice*** *il pacchetto di* n *bit, come visto è formato da* $n-1$ *bit di messaggio e un bit di controllo, la* ***check digit****. Si procede quindi, ricordando che siamo in un caso binario, a contare il numero di* 1 *nei primi* $n-1$ *bit e se questo è dispari setto il bit di check a* 1 *e si nota che così il numero di* 1 *nel pacchetto intero di* n *bit diventa pari, avendo la cosiddetta* ***parità pari*** *(ovvero ogni sequenza valida ha un numero pari di* 1*). Controllando il numero di* 1 *il ricevente capisce se il rumore ha modificato il messaggio anche se non può capire cosa è successo (avendo quindi che il* controllo di parità semplice *è solo un* error detecting code *e non un* error correcting code*). Non potendo fare nulla, in caso di errore identificato, il ricevente può solo chiedere al mittente di inviare nuovamente il messaggio. Qualora il rumore modificasse il messaggio in modo tale che si abbia comunque*

un numero apri di 1 *si rientrerebbe nella casistica sopra descritta in cui il rumore forma ancora un messaggio valido. Questa cosa può succedere se, nel caso binario, il rumore modifica un numero pari di cifre e quindi il* controllo di parità semplice *funziona solo se viene modificato un* numero dispari di cifre*.*

Vediamo quindi meglio come calcolare la **check digit**.
Si rinominiamo gli n bit come:

$$x_1 x_2 \cdots x_{n-1}\, y$$

quindi con y *check digit*.
Si ha che, con $\oplus$ *xor*:

$$y = x_1 \oplus x_2 \oplus \cdots \oplus x_{n-1}$$

Ricordando che:

a	b	$a \wedge b$	$a \oplus b$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

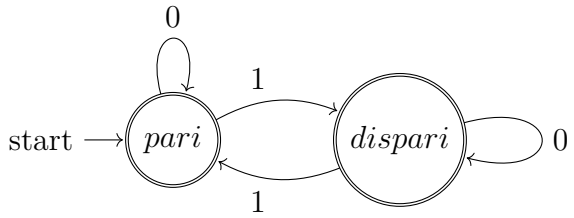
quindi vale 1 sse i due bit in input sono diversi ma questo non ci aiuta su $n-1$ input. Altrimenti si ha che vale 1 sse il numero di 1 in input è dispari e questo ci aiuta su $n-1$ input infatti la generalizzazione dello *xor* a più di due input è detta **funzione di parità**. Un altro punto di vista per considerare lo *xor* è quello della **somma a modulo 2** usando la notazione:

$$y = \bigoplus_{i=1}^{n-1} x_i = \sum_{i=1}^{n-1} x_i \bmod 2$$

avendo che faccio prima la somma e poi il modulo 2 mi dice 0 se è pari e 1 se è dispari.
Si ha inoltre una relazione interessante tra le formule scritte usando solo $\oplus$ e $\wedge$ nella cosiddetta **forma algebrica normale (*ANF*)**. Queste formule booleane possono essere trasformate in formule aritmetiche con *modulo 2*, quindi in $\mathbb{Z}_2$ dicendo che lo *xor* equivale alla *formula modulo due* e l'*and* al *prodotto modulo due* (la cosa vale in entrambi i versi).
La funzione di parità è così usata che in tutti i microprocessori, fin dagli anni settanta, si ha un *flag di parità* tra i flag della CPU, che viene settato come

appena visto a seconda dei bit caricati su un registro particolare (a volte detto *accumulatore*). Tale calcolo è facilmente mappabile in un circuito, avendo che lo *xor* gode della proprietà associativa (avendo un circuito che fa un albero di porte *xor*).

Posso anche simulare lo *xor* con un **automa a stati finiti**, con due stati "pari" e "dispari", con il "pari" stato iniziale (diciamo che input vuoto è pari):



Questo metodo ha senso se il canale è pochissimo rumoroso, avendo pochissima probabilità di avere la modifica di un bit e ancora meno di due (due modifiche si ricorda che non verrebbero rilevate essendo pari), così poca da poter ipotizzare che non avvengano mai due errori (e se mai dovesse succedere bisognerà valutare l'impatto del problema e le conseguenze). Stiamo assumendo quindi che la **probabilità d'errore** può essere **trascurabile** infatti canali di buona qualità dovrebbero sbagliare non più di un bit su un milione, per canali più affidabili anche uno su un miliardo. Possiamo quindi trascurare che possano accadere due errori e dire che il **controllo di parità** va bene.
Parliamo ora meglio di **ridondanza**, definendola formalmente.

Definizione 3. *La* ***ridondanza*** *R è definita come:*

$$R = \frac{\textit{il numero totale di simboli/cifre spediti}}{\textit{numero di simboli/cifre che sono effettivamente parte del messaggio}}$$

Nel caso del controllo di parità *i simboli che vogliamo spedire sono n bit a fronte di $n-1$ bit di vero messaggio. Si ha quindi:*

$$R = \frac{n}{n-1} = \frac{(n-1)+1}{n-1} = 1 + \frac{1}{n-1}$$

Mettendo in evidenza che la ridondanza è sempre $R \geq 1$, visto che a numeratore abbiamo almeno una cifra in più (quella della check digit*) e che quindi è sicuramente maggiore del denominatore. In realtà per avere $R = 1$ dovrei avere numeratore e denominatore uguali che non ha molto senso parlando di ridondanza, quindi nei casi* interessanti *si ha che $R > 1$. Guardando la formula la cosa è confermata da $1 + \frac{1}{n-1}$ con $\frac{1}{n-1}$ che viene detto* ***eccesso di ridondanza****.*

Definizione 4. *Possiamo **generalizzare** la definizione di **ridondanza**:*

$$R = \frac{msg + check}{msg}$$

avendo:

- *msg numero di simboli/cifre di messaggio*
- *check numero di cifre di controllo*

Ma allora (avendo check $<$ msg per avere qualcosa di sensato):

$$R = \frac{msg + check}{msg} = 1 + \frac{check}{msg}$$

*che è la forma "generale" della ridondanza. Si ha che $\frac{check}{msg}$ è **eccesso di ridondanza**.*

Si può dire di non avere necessità di "proteggere" di più il bit di parità in quanto, per la macchina, conta come tutti gli altri. Tutti vanno "protetti" nello stesso modo.
*Come ho la **parità pari** potrei avere la **parità dispari**, dove i messaggi validi hanno un numero dispari di 1. I vari ragionamenti sono analoghi, essendo tutto uguale dal punto di vista matematico, avendo un isomorfismo tra le due tecniche. La scelta tra i due dipende dai casi è dalla celta di cosa rappresentiamo con 0 e 1 (pensiamo con 0 che rappresenta assenza di segnale, in questo caso meglio usare la parità dispari, mentre se 0 e 1 rappresentassero diverse quantità di Volt andrebbe bene la parità pari).*
Nel corso si userà comunque solo la *parità pari*.

3.1.2 Il rumore bianco

Introduciamo ora un primo *modello di rumore*, il **modello del rumore bianco**.

Definizione 5. *Un **modello di rumore** è un modello matematica che descrive cosa succede nel canale quando il rumore rovina i bit.*

Definizione 6. *Il **modello del rumore bianco** consiste nell'avere il messaggio con i bit $x_1x_2\cdots x_n$ (con magari x_n come controllo di parità ma dato che "i bit non sono colorati" la cosa non ci interessa davvero) e avere una certa probabilità p. Si hanno due condizioni:*

1. *si ha che* $p \in (0,1)$ *che è la probabilità che avvenga un errore in ogni posizione* $i \in [1,n]$ *del messaggio. Si ha quindi che la probabilità* p *è uguale in tutte le posizioni*

2. *le posizioni sono tutte indipendenti, ovvero il fatto che magari si ha un errore nella posizione* i *non influisce sulle altre. Avendo quindi l'eventi casuale* E_i *con:*

   $$E_i = \text{è avvenuto un errore in } i$$

   allora:

   $$E_i \text{ ed } E_j \text{ sono indipendenti, } \forall i \neq j$$

Le due proprietà sopra elencate rendono molto semplice il modello.
Questo però non è molto realistico, basti pensare al rumore dovuto ad uno sbalzo di corrente, dove da un bit in poi e per diversi bit si avranno alte probabilità d'errore. Quando l'errore influisce su una certa porzione di bit si dice che si ha un ***burst di errori*** *(che non può essere gestito con le tecniche per il rumore bianco, anche se si riesce con qualche workaround).*
Si è visto che $p \in (0,1)$ *infatti:*

- *se si avesse* $p = 0$ *si avrebbe che ogni bit arriverebbe sempre corretto, ma questo può avvenire solo in un mondo utopico e non in quello reale/fisico. Non esiste un canale reale non affetto da errori, quindi si ha* $p \neq 0$

- *se si avesse* $p = 1$ *si avrebbe che ogni bit del messaggio arriverebbe errato ma questa non sarebbe una brutta situazione, anzi sarebbe ottima infatti mi basterebbe avere una porta logica not della linea di trasmissione per riottenere il messaggio corretto, ottenendo un canale* $p = 0$ *d'errore. Anche questo però è irrealistico quindi* $p \neq 1$

Supponiamo ora che $p > \frac{1}{2}$ *quindi ho più probabilità che un bit arrivi sbagliato che giusto. Anche in questo caso una porta logica not alla fine della linea di trasmissione per ottenere un canale con* $1 - p$ *come probabilità d'errore. Quindi anche questo non ha molto senso quindi si considera che:*

$$p \in (0, \frac{1}{2})$$

Manca solo da valutare $p = \frac{1}{2}$.
Con $p = \frac{1}{2}$ *si ha che il bit di output è completamente causale e indipendente*

da cosa sia stato spedito. È come se il canale generasse n bit casuali con probabilità uniforme ($\frac{1}{2}$), avendo un cosiddetto ***canale completamente rumoroso****. Dal punto di vista pratico sarebbe interessante un tale canale, per altri punti di vista (come quello della crittografia), avendo infatti un* ***generatore di bit completamente casuali****. Purtroppo questo non si può fare quindi si assume $p \neq \frac{1}{2}$.*

Cerchiano di capire quale sia la probabilità che avvengano k errori con $0 \leq k \leq n$ (quindi da nessun errore a tutti gli n bit errati), che indichiamo con:

$$p[\text{ k errori }]$$

Valutiamo i vari casi:

- partiamo con 1 errore, quindi $k = 1$, avendo $p[\text{ 1 errori }]$.
  Questo significa che per il messaggio di n bit si immagina un vettore di bit associato con 0 e 1 come "bandierine" che indicano se è avvenuto un errore o no in una certa posizione. Quindi se in un certa posizione ho 0 diciamo che significa che non ho un errore di trasmissione mentre se ho 1 ho un errore. Il messaggio di n bit diventa quindi una sorta di *maschera* che con gli 1 mi dice dove è avvenuto l'errore. Se suppongo che ne è avvenuto uno solo avrò un solo 1 e bisogna calcolare la probabilità che questo avvenga.
  Supponga che l'errore sia al primissimo bit, quindi in posizione $i = 1$, avendo quindi, per il discorso delle "bandierine" che $msg = 10000\ldots0$ e quindi si ha, avendo che la probabilità che avvenga la trasmissione avvenga correttamente è $1 - p$ (cosa che avviene $n - 1$ volte), mentre p che avvenga sbagliata (cosa che avviene una sola volta):

  $$p[\text{ 1 errori }] = p^1 \cdot (1-p)^{n-1} = p \cdot (1-p)^{n-1}$$

  Posso fare $\cdot$ in quanto si è supposta l'*indipendenza* (non avendo intersezioni tra gli eventi).
  Ma questo non sta considerando tutto ma solo la prima posizione. Completando il calcolo avendo di volta in volta in somma la probabilità di un errore nella posizione i ho che:

  $$p[\text{ 1 errori }] = p^1 \cdot (1-p)^{n-1} + (1-p)^1 \cdot p^1 \cdot (1-p)^{n-2} + \cdots$$

  ma questo conto si può semplificare, avendo sempre gli stessi termini che si ripetono:

  $$p[\text{ 1 errori }] = n \cdot p^1 \cdot (1-p)^{n-1} = n \cdot p \cdot (1-p)^{n-1}$$

Infatti so che $p \cdot (1-p)^{n-1}$ è la probabilità di avere un errore in una certa posizione fissata. Mi chiedo dove posso mettere questa posizione in tutti i modi possibili nel pacchetto di n bit e ho che, avendo un solo errore, ho n modi per posizionarlo, ciascuno con probabilità $p \cdot (1-p)^{n-1}$

- passiamo a due errori, avendo $p[$ 2 errori $]$.
  Ho un ragionamento analogo. Parto supponendo di avere i due errori nelle prime due posizioni del messaggio/pacchetto, avendo quindi 1 nelle prime due posizioni della maschera. Abbiamo comunque già visto che poi il ragionamento si generalizza per qualsiasi posizione, in questo caso coppie (anche non consecutive) di posizioni. Si ha che, ipotizzando che le prime due siano errate:

  $$p[\text{ 2 errori }] = p^1 \cdot p^1 \cdot (1-p)^{n-2} = p^2 \cdot (1-p)^{n-2}$$

  Ma anche qui dobbiamo vedere la probabilità per qualsiasi coppia, facendo variare le due posizioni d'errore in tutti i modi possibili ma questo è come prendere un qualsiasi sottoinsieme di due elementi a partire da un insieme di n elementi ma questo altro non è che il calcolo che si fa tramite il coefficiente binomiale, avendo quindi:

  $$p[\text{ 2 errori }] = \binom{n}{2} \cdot p^2 \cdot (1-p)^{n-2}$$

- analogamente a quanto fatto per due errori potrei fare con tre, quattro, etc...

- possiamo generalizzare con k errori, avendo $p[$ k errori $]$.
  Si hanno quindi k uni da disporre in tutti i modi possibili nel vettore di n bit. Si ha quindi:

  $$p[\text{ k errori }] = \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k}$$

  E quindi posso valutare la cosa nei due casi estremi:

  - $k = 0$, avendo 0 errori. Ho un solo modo per mettere zero 1 nella maschera di bit (da nessuna parte) e infatti (avendo poi tutti gli n bit la stessa probabilità di uscire corretti):

    $$p[\text{ 0 errori }] = \binom{n}{0} \cdot p^0 \cdot (1-p)^{n-0} = 1 \cdot 1 \cdot (1-p)^n = (1-p)^n$$

– $k = n$, avendo n errori[1]. Ho un solo modo per mettere tutti 1 nella maschera di bit (ovunque) e infatti (avendo poi tutti gli n bit la stessa probabilità di uscire errati):

$$p[\text{ n errori }] = \binom{n}{n} \cdot p^n \cdot (1-p)^{n-n} = 1 \cdot p^n \cdot 1 = p^n$$

Si nota che i due casi estremi sono "speculari".

In questo elenco puntato si è quindi ragionato sulle celle della maschera di bit associata al pacchetto e non del pacchetto in se, anche se spesso risulti ambiguo.
Consideriamo ora nuovamente $p[\text{ 1 errore }]$, si ha che, dalla generalizzazione è:

$$p[\text{ 1 errore }] = \binom{n}{1} \cdot p^1 \cdot (1-p)^{n-1} = n \cdot p \cdot (1-p)^{n-1}$$

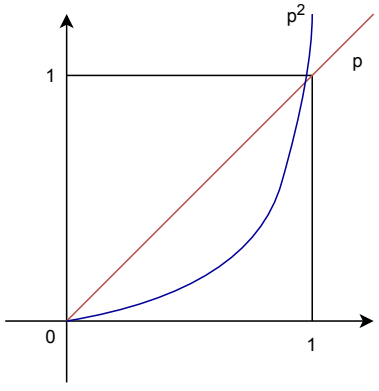
Introduciamo un'approssimazione interessante dell'analisi matematica che vale per $\alpha \in \mathbb{R}$ e $|X| < 1$, ovvero $-1 < x < 1$:

$$(1+x)^\alpha \simeq 1 + \alpha \cdot x$$

Ovvero $1 + \alpha \cdot x$ sono i primi due termini dello sviluppo in serie di $(1+x)^\alpha$.
Tratto quindi la formula per un errore in base a questa approssimazione:

$$p[\text{ 1 errore } = n \cdot p \cdot (1-p)^{n-1} \simeq n \cdot p \cdot [1 - p \cdot (n-1)] = n \cdot p - n^2 \cdot p^2 + n \cdot p^2$$

Ma so che $p \in (0,1)$ e quindi $p^2 < p$, infatti (*grafico approssimativo*):



[1]Su dispense del prof grafico con $n = 8$, $p = 0.1$ e k che varia tra 0 e 8

ma quindi, sempre approssimando (avendo quindi già due approssimazioni):

$$p[\text{ 1 errore } = n \cdot p \cdot (1-p)^{n-1} \simeq n \cdot p - n^2 \cdot p^2 + n \cdot p^2 \simeq n \cdot p$$

Quindi:

$$p[\text{ 1 errore } \simeq n \cdot p$$

Analogamente ragiono per due errori:

$$p[\text{ 2 errori }] = \binom{n}{2} \cdot p^2 \cdot (1-p)^{n-2} \simeq \binom{n}{2} \cdot p^2 \cdot [1 - p \cdot (n-2)] = \binom{n}{2} \cdot (p^2 - n \cdot p^3 + 2p^3)$$

Ma anche qui si ha che $p \in (0,1)$ e quindi $p^3 < p$, e quindi si ha:

$$p[\text{ 2 errori }] \simeq \binom{n}{2} \cdot p^2 = \frac{n \cdot (n-1)}{2} \cdot p^2$$

In generale, per k errori, con gli stessi passaggi:

$$p[\text{ k errori }] \simeq \binom{n}{k} \cdot p^k$$

I conti diventano molto più semplici.
Si è detto che se la probabilità di due errori è piccoli si può decidere di trascurarla (usando poi solo il *controllo di parità semplice*). Da queste approssimazioni vediamo che la probabilità di un errori è $\simeq n \cdot p$ mentre per due $\simeq \frac{n \cdot (n-1)}{2} \cdot p^2$. Se ipotizziamo $p \sim 10^{-6}$, quindi uno su un milione, si ha che $p^2 = 10^{-12}$ quindi assolutamente trascurabile. Si segnala comunque che questa non è una pratica standard. Normalmente si hanno, ad esempio, **low density parity codes (LDPC)** dove si sparano a caso vari controlli di qualità nell'ottica di mantenere le proprietà di correzione degli errori usando meno controlli possibile, avendo, tornando alla ridondanza $R = 1 + \frac{check}{msg}$, che si vuole usare il minor numero di cifre di controllo per abbassare l'*eccesso di ridondanza*, abbassando la ridondanza stessa, avvicinandosi quindi a 1^+ (ci si avvicina da destra ovviamente). Riducendo l'*eccesso di ridondanza* si ha che ogni cifra di controllo *copre/protegge* il maggior numero ci simboli/cifre del messaggio. *A parità di simboli inviati si vuole quindi ridurre il numero di cifre di controllo.*
Approfondiamo e usiamo quindi il *modello del rumore bianco* per vedere qual è a probabilità che il ricevente non riesca a capire che c'è stato un errore utilizzando il *controllo di parità semplice.*
Ricordiamo che il messaggio è della forma:

$$x_1 x_2 \cdots x_{n-1} y$$

con *y controllo di parità semplice* calcolato come:

$$y = \bigoplus_{i=1}^{n-1} x_i$$

Un numero dispari di errore mi segnala che ci sono stati problemi, usando la *parità pari* (i pacchetti inseriti nel canale hanno un numero apri di 1).
Vogliamo quindi la probabilità che il controllo di parità fallisca, ovvero:

$$p[\text{ controllo } \bigoplus \text{ fallisce }]$$

ma questo è uguale alla probabilità che avvenga un numero pari di errori (**zero escluso**, ovviamente):

$$p[\text{ controllo } \bigoplus \text{ fallisce }] = p[\text{ numero pari di errori }] = p[\text{ 2 errori }] + p[\text{ 4 errori }] + \cdots$$

Diciamo che, per comodità:

$$1 = (1-p) + p = [(1-p) + p]^n$$

Ma so che $(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$, quindi:

$$1 = [(1-p) + p]^n = \sum_{k=0}^{n} \binom{n}{k} (1-p)^{n-k} p^k$$

ma questa è $\binom{n}{k}(1-p)^{n-k}p^k = p[\text{ k errori }]$ (infatti la somma di tutte le probabilità è appunto 1), quindi:

$$1 = [(1-p) + p]^n = \sum_{k=0}^{n} \binom{n}{k} (1-p)^{n-k} p^k = \sum_{k=0}^{n} p[\text{ k errori }]$$

D'altro canto posso anche dire che, sempre applicando l'espansione di $(a+b)^n$, scomponendo però $(-p)^k$ in $(-1)^k \cdot p^k$ (dove $(-1)^k$ vale 1 per k pari e -1 per k dispari). Si ha quindi:

$$(1 - 2 \cdot p)^n = [(1-p) - p]^n = \sum_{k=0}^{n} (-1)^k \cdot \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k}$$

Ma quindi ho:

$$(1 - 2 \cdot p)^n = \begin{cases} \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k} & \text{sse } k \text{ è pari} \\ -\binom{n}{k} \cdot p^k \cdot (1-p)^{n-k} & \text{sse } k \text{ è dispari} \end{cases}$$

Quindi se k è dispari le espansioni di 1 e $(1-2\cdot p)^n$ sono uguali ma di segno opposto mentre se k è pari sono uguali con lo stesso segno. Ma quindi questa somma delle due espansioni mi lascia col doppio dei soli termini con k pari che ci aiuta volendo calcolare proprio le probabilità con un numero di errori pari. Si ha quindi, dividendo già per due avendo il discorso del doppio:

$$\frac{1+(1-2\cdot p)^n}{2} = \sum_{t=0}^{\lfloor \frac{n}{2} \rfloor} \binom{n}{2\cdot t} \cdot p^{2t} \cdot (1-p)^{n-2t}$$

La somma va quindi da 0 alla parte intera di $\frac{n}{2}$. Nel coefficiente binomiale ho $2\cdot r$ che è una quantità sicuramente pari. In generale è come se avessi $k = 2\cdot t$ ciclando solo sui k pari. Ho quindi ottenuto:

$$p[\text{ 0 errori }] + p[\text{ 2 errori }] + p[\text{ 4 errori }] + \cdots$$

Non vogliamo però $t = 0$ quindi:

$$p[\text{ controllo } \bigoplus \text{ fallisce }] = p[\text{ numero pari di errori }] = \frac{1+(1-2\cdot p)^n}{2} - p[\text{ 0 errori }]$$

$$= \frac{1+(1-2\cdot p)^n}{2} - \binom{n}{0} \cdot p^0 \cdot (1-p)^{n-0} = \frac{1+(1-2\cdot p)^n}{2} - (1-p)^n$$

E quindi:

$$p[\text{ controllo } \bigoplus \text{ fallisce }] = p[\text{ numero pari di errori }] = \frac{1+(1-2\cdot p)^n}{2} - (1-p)^n$$

D'altro canto potrei anche calcolare $p[$ numero dispari di errori $]$:

$$p[\text{ numero dispari di errori }] = 1 - p[\text{ numero pari di errori }]$$

(*o anche modificando la sommatoria per ciclare sui k dispari*).
Facendo qualche conto[2] si ottiene che:

$$p[\text{ numero dispari di errori }] = 1 - (p[\text{ numero pari di errori }] + p[\text{ 0 errori }])$$

ovvero:

$$p[\text{ numero dispari di errori }] = \frac{1-(1-2\cdot p)^n}{2}$$

In generale il numero dispari di errori è meno interessante.

[2] i calcoli per il numero dispari di errore sono materiale extra sulle dispense del docente

3.1.3 Gestione dei burst

Come abbiamo introdotto con il solo *controllo di parità* e con il *rumore bianco* non si possono gestire i **burst di errori**. Vediamo quindi un modo semplice per gestirli.
Si supponga di voler spedire dei messaggi formati da lettere, ad esempio:

c	i	a	o

Posiamo di rappresentare ogni lettera tramite l'**ASCII standard** a 7 bit:

char	bit
c	1000011
i	1001001
a	1000001
o	1001111

Si supponga di avere dei burst di errori di lunghezza L e per semplicità assumo L di lunghezza pari alle singole word, quindi $L = 7$. Per gestire il burst spedisco prima i 7 bit della prima lettera poi quelli della seconda etc... Infine spedisco un intero pacchetto di bit di controllo di 7 bit dove ogni bit viene calcolato controllando quella posizione di bit in tutti i pacchetti precedenti, sempre tramite lo *xor*. Nel caso d'esempio si ha quindi, con x per indicare il **check** (se nella colonna sopra ho un numero apri di 1 metto 0 altrimenti 1):

c	1000011
i	1001001
a	1000001
o	1001111
x	0000100

Suppongo un burst che rovini dalla posizione 2 alla 4 incluse (avendo che quindi molto probabilmente non ci torneranno i conti facendo il check su $x[2, 4] = 000$).
Ovviamente anche qui un numero pari di errori inganna il sistema avendo comunque un *controllo di parità semplice* e anche in caso d'errore il ricevente non sa comunque dove sia avvenuto e quindi fa **detection** ma non può fare **correction**.

3.1.4 Codici pesati

Abbiamo già parlato del **codice 01247** vediamo ora un codice pesato più interessante e utilizzato.

Definizione 7. *Definiamo questo* ***codice pesato*** *come un codice per cui si hanno alcune cifre di messaggio* $msg = m_1 m_2 \cdots m_n c$ *alle quali associamo dei pesi che dipendono dalla posizione in cui si trovano le varie cifre. In particolare si ha peso:*

- 1 *per la* ***check digit*** c
- 2 *per* m_n
- *si prosegue sempre aumentando di 1 per le altre cifre*
- n *per* m_2
- $n+1$ *per* m_1

Questo si fa perché la cifra di controllo è calcolata per far ottenere:

$$m_1 \cdot (n+1) + m_2 \cdot n + \cdots + m_n \cdot 1 + c \cdot 1 = 0$$

ma ovviamente questo non sembra possibile e infatti i conti sono fatti in **modulo numero primo**, avendo per esempio, se scegliamo come numero prima 37:

$$m_1 \cdot (n+1) + m_2 \cdot n + \cdots + m_n \cdot 1 + c \cdot 1 \equiv 0 \bmod 37$$

La scelta di 37 non è causale, infatti volendo:

- rappresentare le 21 lettere dell'alfabeto inglese
- rappresentare dieci cifre da 0 a 9
- un simbolo per lo spazio

e quindi siamo a 32 simboli e ci serve un numero primo ≥ 31 e quindi va bene 37.
Vogliamo un numero prima perché se vogliamo fare i conti con le congruenze è più semplice farle in *modulo numero primo*.
Lavoriamo quindi nella classe dei resti:

$$[0]_{37}, [1]_{37}, \ldots, [36]_{37}$$

e in questo modo se facciamo le varie operazioni è tutto uguale al solito fino a 36 (cosa che non succede per le classi dei resti in modulo non numero primo). La classe dei resti in modulo numero primo è un **campo** mentre se non fosse primo si avrebbe un **anello**. In un campo se $x \cdot y = 0$ o che $x = 0$ oppure $y = 0$ (cosa che non succede negli anelli). Inoltre in un campo ho che se $x \cdot y = z$ allora $x = z \cdot y^{-1}$ (in un anello non per tutti gli y esiste un y^{-1} mentre in un campo sì).
Facendo dipendere il calcolo del peso della **check digit** da tutti gli altri pesi perché, così facendo, sopratutto nelle comunicazioni di tipo **seriale** (dove si spedisce una cifra alla volta), ci si accorge subito se una cifra è andata persa oppure se si è aggiunta cifra o se due cifre si sono scambiate (cosa comunque difficile in un sistema di comunicazione elettronico ma è utile in altre situazioni, sopratutto di conti "a mano").
Si supponga di avere delle cifre b e a, la prima con peso $k+1$ e la seconda con peso k, avendo una scrittura del tipo $cifra(peso)$:

$$b(k+1) + a(k)$$

Ipotizziamo di scambiare a e b (ora a pesa $k+1$ e b pesa k), avendo:

$$a(k+1) + b(k)$$

Ma, facendo la differenza non è nulla:

$$[b(k+1) + a(k)] - [a(k+1) + b(k)] \neq 0$$

infatti ho:

$$b \cdot k + b + a \cdot k - a \cdot k - a - b \cdot k = b - a$$

ma $b - a = 0$ sse $b = a$ e quindi l'unico caso in cui non ci si accorge dello scambio è avere lo scambio di due cifre uguali che non fa cambiare il risultato. Questa idea viene usata anche nei codici a barre. Vediamo quindi un algoritmo per calcolare la cifra di controllo:

Algorithm 1 Algoritmo di calcolo dei pesi per codice pesato

```
function CheckCalc
    sum ← 0
    ssum ← 0
    while not EOF do
        read sym
        sum ← sum + sym(mod37)
        ssum ← ssum + sum(mod37)
    temp ← ssum + sum(mod37)
    c ← 37 − temp(mod37)
    return c
```

Dove:

- *sum* tiene conto della somma numerica della nostro calcolo, accumulando i vari termini
- *ssum* che è una *somma delle somme* e tiene conto implicitamente dei vari persi che crescono spostandoci da destra a sinistra come visto sopra

I mod37 nel ciclo sono in realtà superflui ma conviene farli per non non far diventare i numeri troppo grossi.
Vediamo una più chiara simulazione. Il mittente ha un messaggio e ci calcola la **check digit**. Avendo, simulando per un messaggio $wxyzc$, con c *check digit*:

msg	sum	ssum
w	w	w
x	$w + x$	$2 \cdot w + x$
y	$w + x + y$	$3 \cdot w + 2x + y$
z	$w + x + y + z$	$4 \cdot w + 3 \cdot x + 2 \cdot y + z$
c	$w + x + y + z + c$	$5 \cdot w + 4 \cdot x + 3y + 2 \cdot z + c$

Arrivato alla fine voglio calcolare c in modo che:

$$5 \cdot w + 4 \cdot x + 3y + 2 \cdot z + c \equiv 0 \bmod 37$$

Chi riceve fa lo stesso calcolo e alla fine controlla la **check digit**. Un altro modo per il ricevente è quello di fare solo l'ultimo calcolo se farli tutti step by step.