Crittografia

Alessandro Pioggia, Luca Rengo, Federico Brunelli, Leon Baiocchi

23 febbraio 2022

Indice

Capitolo 1

Introduzione

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 2

One-time pad

Un cifrario è perfetto se, una volta scelto il cifrario (o progettato), occorre valutare la sua sicurezza (quanto è resistente agli attacchi). Abbiamo due metodi classici che ci permettono di capire se un cifrario è robusto. Ovvero

- Protocollo a chiave segreta (metodo empirico) : si distribuisce agli scienziati il cifrario, se dopo tot tempo nessuno lo viola significa che è sicuro. Probabilmente è sicuro però, sarebbe più opportuna una dimostrazione pratica che dimostra che il protocollo non è violabile (staremmo più tranquilli).
- (metodo formale) : Dimostro che, quel protocollo crittografico non è sicuro, però per violarlo è necessario disporre di una determinata quantità di risorse (tempo di calcolo). Se le risorse sono inarrivabili, il protocollo è sicuro ed è migliore del metodo empirico esposto nel punto precedente.

Il one-time-pad è un esempio di cifrario perfetto, in cui riusciremo a dimostrare (cosa rara nei cifrari), che in determinate situazioni, il cifrario non è violabile, con qualsiasi quantità di risorse a disposizione. Problema : è troppo costoso, è quasi inutilizzabile, tranne in certi casi specifici (molto ristretto). Introduce però alcuni concetti che saranno utili nel proseguo del corso.

2.1 Esistono cifrari inviolabili?

Inviolabile = qualcuno intercettando il crittogramma che passa sul canale di comunicazione, riesce a risalire al messaggio? Anche conoscendo le funzioni di cifratura/decifratura? Considerando che l'unica cosa che non conosciamo è la chiave segreta.
La risposta è : esistono cifrari inviolabili, come ad esempio il one-time-pad ma l'impiego è quasi impossibile per via del loro altissimo costo. Dunque fra i cifrari che si utilizzano in pratica non esistono quelli inviolabili, si utilizzano invece cifrari sicuri ed economici e che vengono utilizzati ogni giorno per comunicazione di massa(acquistiamo efficienza e perdiamo sicurezza).

2.2 Processo stocastico (Modello matematico)

Abbiamo una comunicazione tra mittente (Mitt) e destinatario (Dest), questa comunicazione è modellata come un processo stocastico (ovvero probabilistico), in cui il comportamento del mittente è descritto da una variabile aleatoria M (assume certi valori con una determinata probabilità). Valori assunti da M = possibili messaggi. I possibili messaggi vengono presi dallo spazio dei messaggi (msg). Abbiamo un canale, dove viaggia il crittogramma corrispondente al

messaggio dopo averlo codificato e sul canale abbiamo una variabile aleatoria C, che assume i valori dei crittogrammi generati a partire dai messaggi. In sintesi :

- M : variabile aleatoria;
- m : valore assunto dalla variabile aleatoria in un tempo t;
- C : processo che trasforma il messaggio;
- c : crittogramma generato a partire dal messaggio, C(m) = c. il crittogramma è un valore possibile del canale. Quando m viene codificato con C, sul canale leggo c.

La distribuzione di probabilità della M dipende dalla sorgente, chi è la sorgente? la sorgente è colui che genera il messaggio (tira fuori i bit, che vengono organizzati in blocchi e poi spenditi sul canale).

Nota : Non è possibile che due messaggi uguali generino due c diverse, allo stesso modo non è possibile che due messaggi diversi abbiano uno stesso c, dal momento che in quel caso il destinatario non sarebbe in grado di ottenere con precisione il messaggio in seguito alla codifica.

Esempio applicativo : Il mittente è il signor dado, che vuole spedire un messaggio al destinatario. I messaggi che spedisce il signor dado sono $M = \{1, 2, 3, 4, 5, 6\}$. Il modello matematico dice che io spedisco $1conp = 0.5; (2, 4, 5, 6)conp = 0.1$ (questa è la distribuzione di probabilità del nostro mittente dado). Quando dado vuole spedire 4, prende la sua codifica e la cifra, $C(4)$, che genererà un crittogramma c che poi verrà spedito sul canale. Chi guarda il canale (ad esempio l'intruso), prima che passi c, sa già che il messaggio 4 esce con probabilità 0.1, stessa cosa per ogni valore. Ora supponiamo di aver letto c, noi vogliamo che conoscendo c la probabilità che il messaggio fosse 4 rimanga sempre 0.1, questo vuol dire che l'informazione rivelata dall'evento "ho letto c" è nulla, non serve a nulla. Un cifrario è perfetto se la probabilità iniziale rimane inalterata anche se viene visto ciò che è passato sul canale di comunicazione.

2.2.1 Notazione

$P(M = m) =$ { con quale probabilità il mittente vuole spedire il messaggio m } .
$P(M = m|C = c) =$ { con che probabilità il mittente ha spedito il messaggio m, sapendo che sul canale abbiamo letto il crittogramma c. }

Deduciamo che Se le due probabilità sono uguali, abbiamo che la lettura del crittogramma c non mi ha dato nessuna informazione riguardo il messaggio che è stato spedito (concetto di indipendenza : $P(A|B) = P(A)$).

2.3 Scenario ipotetico

Il crittoanalista conosce la distribuzione di probabilità con cui il mittente genera il messaggio, il cifrario e lo spazio k delle chiavi, l'unica cosa che non conosce è la chiave che è stata utilizzata (la chiave segreta).
Secondo Shannon il cifrario è perfetto quando, per ogni $m \in Msg$ e $c \in Crittogrammi$, vale la relazione : $P(M = m|C = c) = P(M = m)$. Supponendo di avere messaggi composti dalla coppia nome+cognome, non mi basta che l'intruso non sappia nè nome nè cognome, non voglio nemmeno che l'intruso scopra se l'utente sia maschio o femmina, non voglio che vengano capite nemmeno caratteristiche legate al messaggio.

2.3.1 Descrizione scenario

Supponiamo di avere dei messaggi contenenti la coppia <nome, cognome> e che tutte le coppie siano equiprobabili. Supponiamo inoltre che sulla terra ci siano 1000 coppie <nome, cognome>. Supponiamo inoltre che sul pianeta terra ci siano 300 maschi e 700 femmine. Senza leggere nulla sul canale quale è la probabilità che sia spedito un certo <nome, cognome>? 1/1000. Se io invece leggo il crittogramma c e riesco a capire che si tratti di una femmina, la probabilità varia! Perchè la probabilità che il nome contenuto nel messaggio sia <Luigi, Rossi> diventa pari a 0, perchè Luigi è un nome maschile. In un contesto perfetto, questo non deve accadere.

2.4 Numero di chiavi in un cifrario perfetto

La prima dimostrazione formale la si può fare sulla base che: In un cifrario perfetto, il numero delle chiavi possibili deve essere maggiore o uguale al numero dei messaggi possibili.

Vuol dire che se io voglio spedire messaggi lunghi 100 bit, noi sappiamo che i messaggi lunghi 100 bit sono 2^{100}. Ora dimostriamo che, se il cifrario è perfetto, allora io ho bisogno di almeno 2^{100} chiavi. In sostanza le chiavi devono cifrare i messaggi, nascondendo tutte le possibili informazioni, di conseguenza ne servono tante.

2.4.1 Dimostrazione formale

Sia N_m il numero di messaggi possibili, cioè tali che $P(M = m) > 0$ e sia N_k il numero di chiavi possibili. Poniamo per assurdo che $N_m > N_k$, se questo è vero, esiste un crittogramma c con $P(C = c) > 0$ (ovvero con probabilità che passi sul canale) a cui corrispondono $s \leq N_k$ messaggi (non necessariamente distinti), ottenuti decrittando c con tutte le possibili chiavi. Poichè $N_m \geq N_k \geq s$, esiste almeno un messaggio m con $P(M = m|C = c) = 0 \neq P(M = m)$, ovvero per $N_m > N_k$ il cifrario **non** è perfetto.

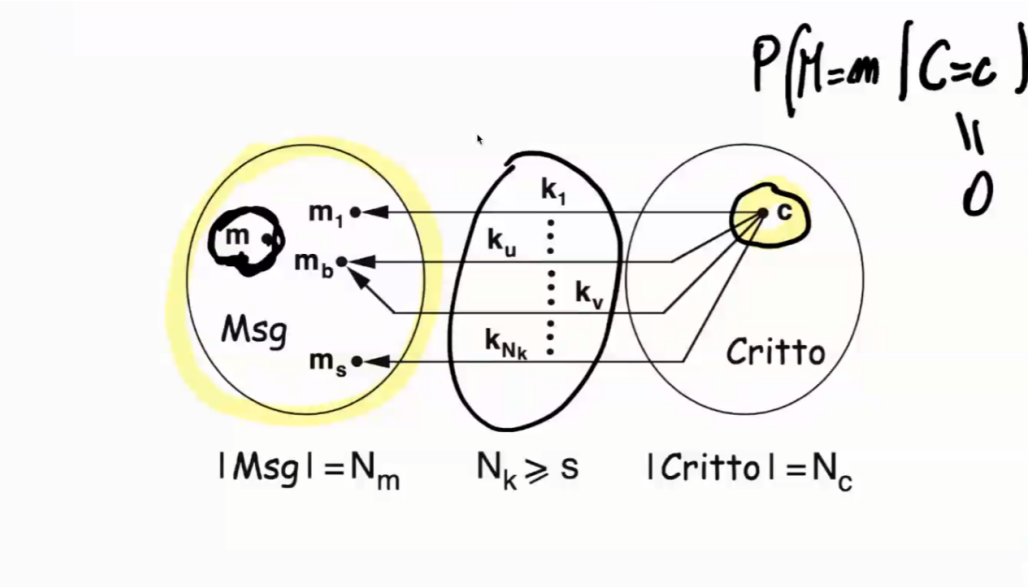


Figura 2.1: Nell'immagine viene esplicitato il ragionamento, si nota che m rimane da solo, in assenza di chiavi con cui crittografare il messaggio. Questo comporta che la $P(M = m)$ non è indipendente rispetto $P(M = m|C = c)$, dunque non si tratta di un cifrario perfetto.

2.5 Deduzione logica

Conseguentemente alla dimostrazione, otteniamo che, in un cifrario perfetto, il numero delle chiavi deve essere maggiore o uguale al numero dei messaggi possibili. Questo ci dice che i cifrari perfetti sono inutilizzabili, perchè in questo caso, se devo spedire 1 gb di messaggio, ho necessità di 1 gb di chiave ed è molto oneroso. Ci dice inoltre che se le chiavi si consumano, un messaggio di 56 bit, una volta spedito, consuma i 56 bit di chiave, dunque non può essere riutilizzata. Dal momento che le chiavi più sicure sono quelle random, inizia a maneggiare in noi il fatto che la randomness sia una risorsa, dunque avere bit a disposizione rappresenta una risorsa (e se usiamo, come in questo caso, chiavi usa e getta, la perdiamo, il cifrario diventa poco robusto). .

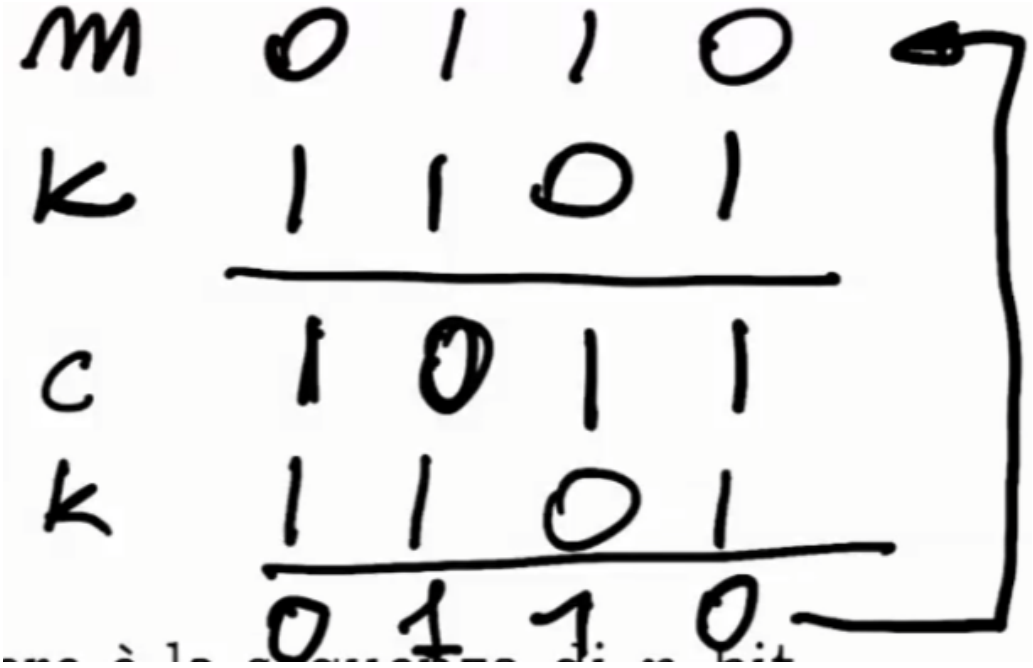
2.6 One-Time Pad

L'one-time-pad veniva usato durante la guerra fredda (Washington e Mosca), perchè la comunicazione non era così frequente (e le chiavi non si consumavano così in fretta), più si riciclano le chiavi, più il protocollo è imperfetto.

2.6.1 Funzionamento

Si costruisce una chiave segreta: Si costruisce una sequenza di bit $k = k_1, k_2, ..., k_n$, nota al mittente e al destinatario. Supponiamo di voler inviare un messaggio più corto di questa chiave e che la chiave sia scelta a caso ((p bit a 0)/(p bit a 1) = 0.5).

Cifratura: Il messaggio da trasmettere $m = m_1, m_2, ..., m_n$, il crittogramma $c = c_1, c_2, ..., c_n$ lo si ottiene eseguendo uno xor fra m e k, per ogni i.
Decifratura : Riapplicando lo xor al c risultante riottengo il messaggio originale.



Dove sta il trucco?? Sta nel fatto che i bit di k, essendo generati a caso, cancellano l'informazione contenuta nel messaggio (dunque un intruso cosa può dire del messaggio? nulla, la chiave lo ha obliterato).

2.6.2 Dimostrazione perfezione One time pad

Per semplicità assumiamo che:

- tutti i messaggi abbiano la stessa lunghezza n;
- tutte le sequenze di n bit sono messaggi possibili (messaggi possibili = tutte le possibili permutazioni della sequenza di bit).

Enunciato

Utilizzando una chiave scelta totalmente a caso per ogni messaggio, il cifrario One-Time pad è perfetto ed impiega un numero minimo di chiavi.

Dimostrazione

Per dimostrare l'enunciato è sufficiente dimostrare che $P(M = m|C = c) = P(M = m)$. E' possibile farlo applicando il teorema di bayes, per il quale :

$$P(M = m|C = c) = \frac{P(M = m, C = c)}{P(C = c)} = \frac{P(C = c|M = m)P(M = m)}{P(C = c)}$$

Osserviamo ora che l'evento $\{M = m; C = c\}$ descrive la situazione in cui Mitt ha generato il messaggio m e l'ha cifrato come crittogramma c. Per la definizione di XOR, fissato il messaggio, chiavi diverse danno origine a crittogrammi diversi, e ogni chiave può essere generata, in maniera randomica, con probabilità $(\frac{1}{2})^n$ (questo perchè o il bit è 0 o 1, per una sequenza di n elementi ed è random). Dunque, fissato m, risulta $P(C = c) = (\frac{1}{2})^n$ per ogni c, ovvero una costante, questo comporta che i due eventi siano indipendenti. Di conseguenza si giunge alla conclusione che (seguendo tutti i passaggi):

$$P(M = m|C = c) = \frac{P(C = c|M = m)P(M = m)}{P(C = c)} = \frac{P(C = c)P(M = m)}{P(C = c)} = P(M = m).$$

Inoltre è implicitamente dimostrato che Il numero di chiavi non può essere inferiore al numero di messaggi. Nel One-Time Pad il numero di chiavi è 2n che è uguale al numero di messaggi (non ne possiamo avere di meno).

Capitolo 3

Il caso

La randomness è la regina della crittografia, è vista come una risorsa, costa e richiede energie per produrla : più la randomness è randomica, più i protocolli sono sicuri.

3.1 Bit casuale

Un bit casuale è un qualcosa che può assumere valore 0 o 1 con probabilità $\frac{1}{2}$, questa definizione ci porta però a scontrarci con questa situazione:

- sequenza 1 : 11111111
- sequenza 2 : 10100100

Notiamo che la prima sequenza non è casuale mentre la seconda sì (per la legge dei grandi numeri la prima è molto improbabile che sia casuale, specie se estesa ad un numero di cifre alto). Ma secondo la prima definizione che abbiamo dato considera che $P(11111111) = P(10100100) = \frac{1}{2^{10}}$, il che non è in linea con il nostro ragionamento.

3.2 Casualità in crittografia

In crittografia casuale = "non facilmente prevedibile", un ipotetico avversario (o intruso) non deve essere in grado di prevedere ciò che noi generiamo a caso.

3.3 Casualità secondo Laplace

Laplace osserva che se tu lanci 100 volte una moneta, le combinazioni di risultati (T o M), formano sequenze regolari, facili da comprendere oppure sequenze irregolari che sono incomparabilmente più numerose (ci sono più combinazioni irregolari rispetto a quelle regolari).

3.4 Casualità secondo Kolmogorov

Kolmogorov dice che una sequenza binaria h è casuale se non ammette alcun algoritmo di generazione A la cui rappresentazione binaria sia più corta di h. Se noi prendessimo le cifre dopo la virgola di π notiamo che queste sono abbastanza irregolari. Noi possiamo però creare un programma che, a seconda della complessità, mi può restituire le cifre di π sempre più grandi (il programma però ha una lunghezza finita!). Secondo Kolmogorov quindi, la generazione di π non

è casuale, le cifre dopo la virgola è possibile determinarle in maniera assoluta da un programma di lunghezza costante. Kolmogorov introduce il concetto di complessità computazionale, se ho un algoritmo definito che genera una stringa, in maniera deterministica, non si può parlare di casualità.

3.4.1 Riflessioni

Supponiamo di avere una procedura Random(n) che produce una sequenza di n bit. Quando n cresce le sequenze prodotte da Random non saranno casuali avendo una lunghezza maggiore della rappresentazione binaria della procedura Random.

Quindi secondo Kolmogorov non esistono generatori di numeri casuali!!!

3.5 Pseudo-casualità

Noi abbiamo bisogno di casualità, è la nostra risorsa, quindi sarà necessario considerare il concetto di **pseudo-casuale** (quasi casuale). I generatori pseudo-casuali sono algoritmi che producono sequenze di bit che supereranno i test di casualità. I test di casualità sono proprietà che le stringhe prodotte dai generatori devono soddisfare. I generatori di bit pseudo-casuali sono deterministici, significa che : se prendo un algoritmo, voglio che ogni volta che gli do lo stesso input lui mi dia lo stesso output (non può inventarsi scelte casuali). Per non essere deterministici avrebbero bisogno a loro volta di bit casuali (eh però è un loop infinito, vogliamo un generatore di bit casuali, che per funzionare ha bisogno di bit casuali... determinati con? generatore di bit casuali?). Quindi producono la stessa sequenza di bit ogni volta che li invochiamo a meno che non gli forniamo un "seme" in input. Stesso seme, stessa sequenza! Stiamo dicendo che per generare casualità al mio algoritmo devo passargli casualità, perchè per conto suo l'algoritmo non è in grado di generarla, essendo l'algoritmo un processo deterministico. Seme di casualità : seme perchè cresce, germoglia e diventa più grande, dando un seme di randomness all'algoritmo, la randomness crescerà. Cosa significa? : io darò un numero all'algoritmo e lui con questo numero tirerà fuori una sequenza amplificata. Piuttosto che parlare di generatori, parliamo di amplificatori di casualità, prendo un seme piccolo (es : un numero, tipo il ciclo di clock, la data, ecc.) e lo amplifico.

3.5.1 Definizione formale : generatore di numeri pseudo-casuali

Un generatore di numeri pseudo-casuali è un algoritmo che parte da un piccolo valore iniziale detto seme, solitamente fornito come dato di ingresso, e genera una sequenza arbitrariamente lunga di numeri. Questa a sua volta contiene una sottosequenza detta periodo che si ripete indefinitamente (andando molto avanti ad un certo punto le cifre si ripeteranno). In linea di principio un generatore è tanto migliore quanto più lungo è il periodo (è migliore se riesce ad ingannarci per più tempo possibile, fino a quando crediamo che sia random siamo dentro al periodo, una volta scavallato il periodo riusciamo a capire che l'algoritmo ha finito la sua amplificazione di randomness). Questi generatori possono però essere considerati amplificatori di casualità perché se innescati da un seme casuale di lunghezza m, fornito dall'utente, generano una sequenza "apparentemente" casuale di lunghezza n » m. Una inerente limitazione è che il numero di sequenze diverse che possono essere così generate è al massimo pari al numero di semi possibili, cioè 2m nel caso binario, enormemente minore del numero complessivo 2n delle sequenze lunghe n.

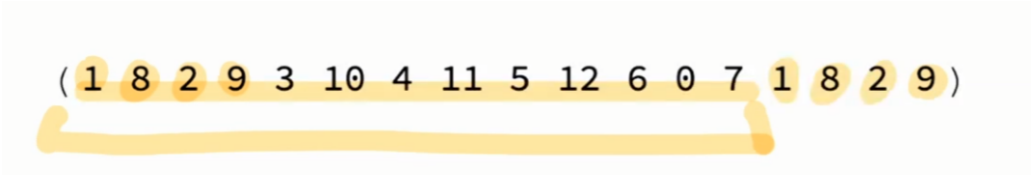


Figura 3.1: Esempio di periodo

3.6 Test statistici di casualità

Si prende una stringa, gli si applicano dei test che ci dicono se, almeno statisticamente, le parti dentro alla stringa sembrano casuali. I test di casualità sono:

- Test di frequenza : (es : la stringa 00000 non passerà il test di frequenza, perchè gli zeri dovrebbero essere circa metà della stringa, deve essere uniformemente generata. La stringa 0000011111 ironicamente passerebbe il test, anche se sembra tutto tranne che casuale)
- Poker test : Si prendono le sottosequenze di una stringa e si osserva come sono distribuite. Con la stringa 0000011111, abbiamo che la sottosequenza lunga 3 appare 3 volte, non passa il test, perchè ci sono sequenze che hanno una frequenza innaturale per essere random, non posso avere troppe stringhe con 3 zeri consecutivi o 3 uni consecutivi.
- Test di autocorrelazione : si verifica che il numero di elementi ripetuti a distanza prefissati abbiano certe caratteristiche. Nella stringa 0000011111 ci accorgiamo che, prendendo i bit, a distanza 5, risulta sempre la coppia [0, 1], non è random.
- Run test : Prendo la sottosequenza massimale tutta uguale e guardo ogni quanto deve comparire una stringa lunga n (nell'esempio è 5) di tutti zeri o uni. Nell'esempio questa ripetizione deve apparire ogni $2^{-stringLength}$ stringhe.

3.7 Generatore lineare

Generatore che supera i 4 test appena enunciati: Un generatore pseudo-casuale molto semplice che supera con successo i quattro test citati è il generatore lineare, che produce una sequenza di interi positivi $x_1, x_2, ..., x_n$ a partire da un seme casuale $x0$ secondo la relazione:
$x_i = (ax_{i-1} + b) \bmod m$, dove a, b, m sono interi positivi. Il seme x_0 è un seme di casualità, un valore di innesco che permette di far funzionare l'algoritmo (deve rispettare le proprietà definite sopra). a, b, m sono 3 valori fissati (che devono rispettare precise proprietà). Mod m significa fare a fette un numero in parti lunghe m e prendere ciò che rimane (resto della divisione per m), serve per rimpicciolire robe gigantesche.

Quindi parto da un valore di innesco x_0 (il seme), che utilizzerò per calcolare la relazione $x_i = (ax_{i-1} + b) \bmod m$, dati a, b, m fissati, per ogni valore i preso in considerazione.

3.7.1 Condizione importante (non serve memorizzarla)

Affinché il generatore abbia periodo lungo m, e quindi induca una permutazione degli interi $0, 1, ..., m-1$, i suoi parametri devono essere scelti in modo tale che $gcd(b, m) = 1$, $(a1)$ sia divisibile per ogni fattore primo di m, e $(a-1)$ sia un multiplo di 4 se anche m è un multiplo di 4 (valori consigliati sono per esempio: $a = 3141592653, b = 2718281829, m = 232$ e seme 0.
Note : gcd è il massimo comun divisore.

Conclusione Nei generatori ci sono proprietà auspicabili per a, b, m, in modo che il generatore lineare amplifichi molto la randomness del seme. I generatori soddisfano i test statistici, però non vanno comunque bene in crittografia, perchè ci sono algoritmi che riescono, dato ad un pezzo di sequenza a risalire ad a, b, m.

3.8 Generatore polinomiale

Il generatore polinomiale è una generalizzazione del generatore lineare, che segue le formulazione:
$x_i = (a_1 x_{i-1}^t + a_2 x_{i-1}^{t-1} + ... + a_t x_{i-1} + a_{t+1})$

3.9 Generatore binario

Noi non vogliamo numeri interi grandi come output, vogliamo dei bit, quindi estraiamo dei bit nel modo più casuale possibile rispetto a quelli generati dal generatore. Si calcola $r = x_i/m$: se la prima cifra decimale di r è dispari pongo il bit a 1, altrimenti a 0.

3.9.1 Difetti

I generatori lineari e polinomiali sono particolarmente efficienti (sono veloci a calcolare i bit casuali) ma non impediscono di fare previsioni sugli elementi generati, neanche quando il seme impiegato è strettamente casuale. Esistono infatti algoritmi che permettono di scoprire in tempo polinomiale i parametri del generatore partendo dall'osservazione di alcune sequenze prodotte, e questo ne svela completamente il funzionamento.

3.10 Generatori basati su funzioni one-way

One-way : senso unico, in una direzione è semplice procedere (il calcolo), nell'altra è difficile (invertire). Le funzioni one-way sono computazionalmente facili da calcolare e difficili da invertire: cioè si conosce un algoritmo polinomiale per il calcolo di $y = f(x)$, ma si conoscono solo algoritmi esponenziali per il calcolo di $x = f^{-1}(y)$. Notiamo che si opera su numeri, quindi il costo degli algoritmi deve essere riferito alla lunghezza della rappresentazione di x : un algoritmo che richiede un numero di operazioni proporzionale al valore di x sarà dunque esponenziale. Il costo computazionale è riferito alla lunghezza del numero (input), se io ho 13215798 la lunghezza del numero è 8 (ovvero il numero di cifre), è l'n in base al quale calcoliamo il costo. La lunghezza dei numeri è correlata al numero stesso tramite un logaritmo (e non con il valore stesso del numero, consideriamo il numero di cifre).

3.11 Generatori basati su funzioni one-way

f è la nostra funzione one-way e definiamo una sequenza applicandola ad un valore.
$S = xf(x)f(f(x))f(f(f(x)))...$
Se io uso questo sistema, quando conosco $f(x)$ posso ricavare $f(f(x))$, quindi i numeri sono facilmente prevedibili. Ma cosa succede se io all'esterno fornisco questi numeri in numero inverso? ovvero f(f(x)), piuttosto che f(x)? Che per ricavare f(f(x)) è necessario invertire la funziona, che abbiamo visto, nelle funzioni one-way, essere difficile. Formalmente:
Se dunque si calcola la S per un certo numero di passi senza svelare il risultato, e si comunicano poi gli elementi uno dopo l'altro in ordine inverso, ciascun elemento non è prevedibile in tempo polinomiale pur conoscendo quelli comunicati prima di esso.

3.12 Test di prossimo bit

Test che ci dice se ciò che stiamo vedendo è abbastanza random per i nostri scopi crittografici. Dice che:
Se io ho 10001101 e voglio scoprire cosa viene dopo l'ultimo 1 mi butto ad indovinare, avendo in ogni caso probabilità $\frac{1}{2}$ di indovinare (questo vale per un qualcuno che non ha informazioni legate alla generazione di bit). Se un qualcuno non ha $P(\text{"}indovinare\text{"}) \geq \frac{1}{2} + \delta$ ho vinto, ho passato il test, se l'algoritmo passa il test viene detto crittograficamente sicuri ed è dimostrabile che passano anche i 4 test standard definiti in precedenza. Essi vengono costruiti impiegando particolari predicati delle funzioni one-way, cioè proprietà che possono essere vere o false per ogni valore di x.

3.13 Predicati hard-core

Un predicato, in logica, è una funzione che si applica ad un qualcosa e restituisce vero o falso. Formalmente:
Un predicato $b(x)$ è detto hard-core per una funzione one-way $f(x)$ se $b(x)$ è facile da calcolare conoscendo il valore di x , ma è difficile da calcolare (o anche solo da prevedere con probabilità di successo maggiore di 1/2) conoscendo solo il valore di $f(x)$. In sostanza la proprietà hard-core, letteralmente "nucleo duro", concentra in un bit la difficoltà computazionale della f.

Abbiamo che se f è una funzione difficilmente invertibile è difficile, a partire da x_1 definire x_0 (la f è one-way). In questo caso, il predicato hardcore mi dice : è difficile calcolare tutto x_0, o c'è solo un bit difficile da calcolare? b deve essere una spremuta di x_0, di un solo bit, che comunque deve essere difficile da calcolare.

3.13.1 Esempio

Un esempio di funzione one-way è la $f(x) = x^2 \bmod n$ se n non è primo. Esempio: $n = 77$ e $x = 10$, è (polinomialmente) facile calcolare il valore $y = 102 \bmod 77 = 23$ ma è (esponenzialmente) difficile risalire al valore di $x = 10$ dato $y = 23$. Ora il predicato: $b(x) =$ "xè$dispari$" è hard-core per la funzione suddetta. Infatti il valore di x permette di calcolare immediatamente $b(x) = 1$ se x è dispari, $b(x) = 0$ se x è pari, ma il problema di calcolare $b(x)$ conoscendo solo il valore $y = x^2 \bmod n$ è esponenzialmente difficile.

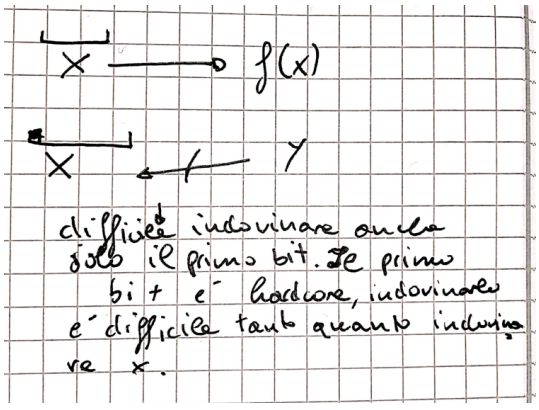


Figura 3.2: Quello che in breve, dobbiamo sapere su questo generatore

3.14 Generatore Blum, Blum, Shub

Figura 3.3: Quello che in breve, dobbiamo sapere su questo generatore

Capitolo 4

DES

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 5

RSA

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 6

DH

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 7

Curve ellittiche

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 8

Firma digitale

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 9

SSL

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 10

Protocolli Zero Knowledge

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 11

Bitcoin

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 12

Teoria dell'informazione

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 13

RFID e protocolli crittografici

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```