Introduzione ai **Generatori di numeri pseudo-casuali**[*] [†]

Nicola Gigante

9 marzo 2016

Life's most important questions are,
for the most part, nothing but
probability problems.

Pierre-Simon de Laplace

Introduzione

Ad ognuno di noi, in molte situazioni della vità è utile poter lanciare una moneta per decidere cosa fare. Ai computer torna utile molto più spesso, e molto più di quanto si possa immaginare. Moltissimi problemi di natura algoritmica sono infatti risolvibili in modo relativamente semplice e molto efficace sfruttando **algoritmi probabilistici**. In un algoritmo di questo tipo, si prendono delle decisioni in base al tiro di una moneta virtuale, cioè in base ad un valore scelto casualmente.[1]

L'esempio forse più semplice di algoritmo probabilistico è il **randomized quick sort** [5], una variante probabilistica dell'omonimo algoritmo di ordinamento, che scegliendo l'elemento pivot in modo casuale riesce a garantire una complessità ottima nel caso medio, indipendentemente dalla distribuzione dell'input. Di altri esempi se ne possono fare a decine, scegliendo tra problemi di teoria dei grafi, geometria computazionale, intelligenza artificiale, calcolo numerico e altro ancora.

La crittografia è un campo in cui la casualità gioca un ruolo fondamentale, e merita una menzione specifica. In questo contesto, la casualità non porta a vantaggi di tipo computazionale, ma è essenziale per garantire la sicurezza di protocolli di autenticazione e algoritmi di cifratura.

Molte altre applicazioni di natura non strettamente algoritmica necessitano di buone fonti di casualità, come ad esempio computer grafica o simulazioni di vario tipo (fenomeni fisico/chimici, processi industriali, meteorologia, ecc.).

[†]Per ottenere l'ultima revisione dell'articolo visitare `http://www.gigabytes.it/data/prng/prng.pdf`

[1]Capire cosa significhi "casualmente" in questo contesto sarà l'argomento di una buona parte di questo articolo

Se ci chiediamo però come ottenere nella pratica queste fonti di casualità così utili e, a volte, strettamente necessarie, ci scontriamo immediatamente con la natura deterministica delle architetture di calcolo che abbiamo a disposizione. Come può un algoritmo, per sua stessa natura deterministico, generare valori scelti casualmente? La risposta è che, ovviamente, non può: fissato un input, l'algoritmo darà sempre lo stesso output. Ci si deve quindi accontentare di algoritmi che generano deterministicamente delle sequenze di valori che, osservate di per sé, **sembrano** ottenute casualmente, e vengono perciò chiamate **sequenze pseudo-casuali**.

È chiaro quanto sia importante, per tutte le applicazioni citate precedentemente, avere a disposizione algoritmi efficaci ed efficienti per risolvere questo problema. Nonostante ciò, la presenza e la buona implementazione di questa componente viene molto spesso data per scontata, e questo argomento raramente approfondito.

Lo scopo di questo lavoro è quindi quello di gettare le basi per una comprensione del funzionamento di questo tipo di algoritmi, e delle questioni da tenere in considerazione quando ci si trovi a sceglierne o (più raramente) a progettarne uno.

La nostra discussione comincia analizzando nella sezione 1 le caratteristiche generali dei generatori di numeri casuali, e i criteri a cui deve sottostare un buon generatore. La questione di come giudicare la "qualità" delle sequenze generate merita una sezione apposita e viene quindi approfondita nella sezione 2. Le sezioni 3 e 4, invece, analizzano due algoritmi specifici, il primo dei quali è un classico ancora fortemente attuale, mentre il secondo è molto vicino allo stato dell'arte del settore. Infine, la sezione 5 introduce la classe dei generatori usati a scopi crittografici, che necessitano di una cura molto maggiore, sia nei requisiti da soddisfare in fase di progettazione, sia nell'implementazione. In questo contesto verranno anche accennati i metodi utilizzati per accumulare "vera" entropia dall'ambiente fisico, per aumentare l'efficacia dei sistemi esistenti.

1 Caratteristiche di un PRNG

Un generatore di numeri pseudo-casuali (Pseudo Random Number Generator, PRNG d'ora in poi), è un algoritmo studiato per produrre in output una sequenza di valori che **sembri** generata in modo casuale.

Come già detto, è chiara l'impossibilità di generare un output in modo "veramente" casuale vista la natura prettamente deterministica del concetto stesso di algoritmo. Il punto chiave è che, per le nostre applicazioni, basta che le sequenze sembrino casuali, per un opportuna definizione di questo concetto. Nella sezione 2 si vedrà come definire in modo utile il concetto di casualità e come giudicare la qualità delle sequenze pseudo-casuali prodotte.

Quali approcci si possono usare per progettare un algoritmo di questo tipo? Il primo a cercare di rispondere a questa domanda in termini moderni fu John von Neumann [28], che propose un metodo chiamato **middle-square**.

L'idea consiste nel partire da un valore di un dato numero n di cifre, chiamato **seme**, elevarlo al quadrato ed estrarre le n cifre intermedie, che costituiranno l'output dell'algoritmo e verranno usate come seme per l'iterazione successiva. Ad esempio, partendo dal valore di sei cifre 654321, l'elevamento al quadrato

produce il valore 428135971041, le cui sei cifre intermedie formano il valore 135971, che verrà poi elevato al quadrato, e via dicendo.

Da questo metodo, anche se rudimentale, si possono capire alcune cose. La prima è la necessità di fornire in input all'algoritmo un **seme**, ovvero un valore che dia il via alla sequenza. Ciò è necessario per poter generare sequenze diverse ogni volta, tuttavia è importante assicurarsi che il buon comportamento del generatore non dipenda da che seme viene usato. Questo è proprio il primo difetto del metodo middle-square, perchè si può vedere subito che, ad esempio, usando come seme il valore zero si otterrà solo una sequenza di zeri.

Un altro difetto di questo metodo è la ripetitività delle sequenze. Come in tutti gli altri PRNG che verranno analizzati, ogni valore dipende da quello precedente, e al più da delle variabili di stato interne al generatore, ma trattandosi di un numero limitato di cifre, la sequenza non può che ripetersi da un certo punto in poi. La lunghezza della sequenza, prima che questa cominci a ripetersi, è chiamata **periodo**. È chiaro che un periodo lungo sia una caratteristica importante, perchè molte applicazioni pratiche richiedono una grande quantità di dati casuali, e una sequenza ripetitiva non è ammissibile. Inoltre, anche in questo caso, è importante che la scelta del seme non abbia influenza. Il middle-square, in presenza di alcuni semi, genera sequenze che si ripetono dopo pochissimi passi e ciò lo rende inutilizzabile.

Un altro aspetto non meno importante è l'efficienza dell'algoritmo. Tuttavia, la dimensione dei valori dati in output e dello stato interno, e quindi dell'input del generatore (ovvero il seme), è spesso una caratteristica intrinseca dell'algoritmo ed è quindi costante. Per questo motivo, l'efficienza di un PRNG è da valutare non tanto sul piano della complessità computazionale, quanto sul piano della possibilità di un'implementazione veloce ed efficiente sulle architetture di calcolo effettivamente a disposizione. Infatti, a seconda dell'architettura su cui si lavora, la scelta di PRNG diversi, o di diversi parametri di progettazione di un certo PRNG, possono risultare in un implementazione più veloce anche di molti ordini di grandezza.

Tuttavia, come si vedrà ad esempio con i generatori congruenziali nella sezione 3, è molto facile degradare a livelli inaccettabili la qualità delle sequenze prodotte a causa di trucchetti di micro−ottimizzazione che possono a prima vista sembrare banali ed innoqui.

1.1 Distribuzioni non uniformi e sequenze floating-point

Tutti i generatori analizzati in questo articolo forniranno in output sequenze di numeri naturali, compresi in un certo intervallo $[0, N]$, con **distribuzione di probabilità uniforme** (senza ripetizioni nel caso dei generatori congruenziali). Non sempre questa è la distribuzione realmente richiesta per l'applicazione che si sta realizzando. Alcune delle più comuni sono la distribuzione **normale** di Gauss, e quella di **Poisson**. A partire da una o più variabili casuali con distribuzione uniforme è però possibile ottenere una variabile casuale di distribuzione arbitraria, attraverso vari procedimenti, ben delineati in Knuth [9], a cui si rimanda per ulteriori approfondimenti.

A volte invece è richiesta una sequenza di numeri in virgola mobile, anzichè interi. Se i valori della sequenza sono compresi in $[0, N]$, la soluzione ovvia è quella di dividere ogni valore per N, ottenendo valori compresi tra 0 e 1. Bisogna essere coscenti però del fatto che questa operazione potrebbe causare una

perdita di precisione. Ad esempio, nel caso molto comune in cui la sequenza è composta da valori di 32 bit, normalizzarla in valori floating-point a precisione singola (32 bit in tutto, 23 di mantissa, 8 di esponente) causa la perdita dei 9 bit meno significativi. A seconda delle applicazioni, questo potrebbe anche non essere un problema, altrimenti è opportuno usare valori floating point a precisione doppia (53 bit di mantissa).

2 Sequenze casuali e pseudo-casuali

Possiamo informalmente definire una **sequenza di numeri pseudo-casuale** come una sequenza di numeri generati deterministicamente che **sembrano** estratti in modo casuale. Definire però cosa si intende per "casuale" è molto più difficile di quanto possa sembrare a prima vista.

Una prima definizione che può venire in mente può essere quella che associa la casualità alla "imprevedibilità".

Ad esempio, si immagini di lanciare una moneta, di segnare i risultati di testa (1) e croce (0) entrambi equiprobabili, e di ottenere le seguenti due sequenze di lanci:

1. `0101010101010101010101010101010101010101`
2. `0010010000111111011010101000100010000101`

Chiunque storcerebbe il naso di fronte alla prima sequenza, segno inequivocabile di una moneta truccata, perchè sembra molto più prevedibile della seconda. Dovendo scommettere, nel primo caso non si avrebbero dubbi a puntare su uno zero come quarantunesima cifra, mentre nel secondo caso le due alternative sembrerebbero equiprobabili, o al massimo si potrebbe considerare leggermente più favorito lo zero, essendo più frequente. In altre parole, la prima sequenza sembra più **prevedibile** della seconda.

Eppure, dal punto di vista puramente probabilistico, tra tutte le sequenze possibili di quaranta caratteri binari, queste due hanno la stessa probabilità $1/2^{40}$ di essere estratte, supponendo una distribuzione di probabilità uniforme e lanci totalmente indipendenti. Nonostante ciò, la prima è indiscutibilmente priva della qualifica di "sembrare casuale".

D'altro canto, la prevedibilità è un fattore soggettivo. Sempre rimanendo su questo esempio, la situazione cambia drasticamente se si realizza che la seconda sequenza è costituita semplicemente dalle prime quaranta cifre dell'espansione binaria di π, rendendola prevedibile quanto la prima.

Questo esempio vuole sottolineare quanto "casualità" e "prevedibilità" siano concetti tra loro correlati ma distinti e, inoltre, prettamente soggettivi, così come soggettivo e dipendente dal contesto sarà la definizione di casualità che stavamo cercando.

Nel contesto applicativo generale dei PRNG, si è indubbiamente interessati all'aspetto statistico delle sequenze generate, e per questo motivo, sono stati nel tempo progettati vari **test statistici** atti a misurare il livello di "casualità" di una sequenza. Per la maggior parte delle applicazioni accennate nell'introduzione, il buon comportamento di una sequenza pseudo-casuale rispetto a determinati test statistici è più che sufficiente. Ad esempio, nel caso di numeri casuali utilizzati in un algoritmo probabilistico o in una simulazione fisica, è sufficiente

che la sequenza utilizzata soddisfi le ipotesi statistiche assunte dal problema specifico (ad esempio, la distribuzione uniforme della scelta del pivot nel randomized quicksort), e non è necessario preoccuparsi di proprietà statistiche aggiuntive (ad esempio l'uniformità su più dimensioni) né della prevedibilità della sequenza.

In un contesto di applicazioni crittografiche, invece, il buon comportamento statistico delle sequenze non è sufficiente, e la prevedibilità è un grosso problema. Infatti, in questi sistemi è essenziale rendere impossibile per un attaccante prevedere il prossimo numero estratto data una sufficente quantità di valori precedenti. Si può intuire, tuttavia, che questa proprietà dipende non tanto dall'aspetto statistico delle sequenze, quanto dalla complessità della relazione che lega ogni valore a quelli precedenti, e quindi dal funzionamento del generatore. Nella sezione 5 si entrerà più nel merito di questi aspetti, mentre nel seguito ci si concentrerà quindi su valutazioni di tipo statistico.

2.1 Test statistici

L'idea su cui si basano la maggior parte dei test statistici è intuitivamente semplice. La frequenza con cui ogni valore compare nella sequenza forma una **distribuzione empirica**, che viene direttamente o indirettamente confrontata con il comportamento esatto di una qualche distribuzione teorica. Molti test statistici sono stati proposti e utilizzati nel corso degli anni. Storicamente, il primo test statistico proposto è stato il test χ^2, ideato da Pearson nel 1900 [24]. Il nome del test deriva dal fatto che una particolare statistica ricavata dalla distribuzione empirica viene confrontata con quella di una particolare distribuzione χ^2. In Knuth [9] ne vengono elencati e analizzati alcuni, tra cui anche il χ^2, proprio nel contesto della verifica di sequenze pseudo-casuali. Una batteria di test statistici particolarmente restrittivi, chiamata **DieHard** [2], è stata proposta negli anni '90. Nonostante l'indubbia utilità, questi test cosiddetti **empirici** non ci forniscono una definizione di un concetto di casualità generale, ma attestano solamente l'effettiva conformità di una data sequenza a predeterminate ipotesi.

Molti autori nel tempo si sono impegnati per rispondere in maniera più profonda e completa a questo quesito. Si accenneranno ora due definizioni di casualità sviluppate parallelamente, che partono da due punti di vista diversi ma in qualche modo legati.

2.2 Casualità come equidistribuzione

Una sequenza di valori compresi tra 0 ed N si dice **equidistribuita**, se ogni sotto intervallo $[a, b]$ con $a, b \in [0, N]$, contiene una quantità di valori proporzionale alla lunghezza $b - a$ dell'intervallo [10]. In altre parole, non ci sono regioni di punti più "dense" di altre. Questo criterio da solo è di per sé fondamentale. Si pensi, ad esempio, al seguente algoritmo (molto inefficiente in verità), per il calcolo del valore di π: si generano casualmente dei punti all'interno di un quadrato di lato unitario e si contano quelli che cadono all'interno della circonferenza inscritta che saranno in quantità proporzionale all'area della circonferenza, da cui si può quindi calcolare π. È chiaro che in un algoritmo di questo tipo l'equidistribuzione dei punti è fondamentale. A ben vedere, in questo esempio è necessario che **coppie** di valori della sequenza siano equidistribuite nel **piano**. Spesso, infatti, si richiede l'equidistribuzione su più dimen-

sioni della sequenza, da cui segue la naturale generalizzazione: una sequenza si dice k-distribuita se tutte le k-uple di valori consecutivi della sequenza sono equidistribuite nella relativa ipersfera nello spazio di k dimensioni.

Ciò equivale a dire che, se B è il numero di simboli di cui è composta la sequenza ($B = 2$ per sequenze binarie), tutte le B^k possibili k-uple hanno uguale probabilità pari a $1/B^k$ di comparire nella sequenza.

Parlando in astratto di sequenze di lunghezza infinita, si può definire una sequenza ∞-distribuita se è k-distribuita per qualsiasi k. Knuth [9] riassume la teoria alle spalle di una definizione di casualità molto profonda che si basa su questo concetto.

Ciò si applica a sequenze infinite, ma una definizione di casualità sarebbe utile anche per sequenze finite, più probabili in pratica. Come già detto, dal punto di vista probabilistico qualsiasi sequenza finita è equiprobabile a tutte le altre della stessa lunghezza, ma vogliamo un criterio che corrisponda all'intuizione che la prima stringa data come esempio all'inizio di questa sezione sia "meno" casuale della seconda. Il criterio di k-distribuzione torna utile, ma una sequenza di lunghezza finita m non può ovviamente essere k-distribuita per qualunque k (ad esempio se $k = m - 1$, ci sono solo due k-uple di valori consecutivi, ovvero la sequenza intera senza il primo elemento o senza l'ultimo, per cui non possono essere equiprobabili tutte le B^k possibili k-uple).

Quindi, definiamo casuale una sequenza di lunghezza m se è k-distribuita per ogni $k \leq \log_B m$. In questo modo, ci sono in tutto al massimo un numero di k-uple pari a:

$$B^k = B^{\log_B m} = m$$

ed è possibile che ognuna compaia con probabilità $1/B^k = 1/m$ all'interno della sequenza.

2.3 Casualità come assenza di pattern

Un diverso punto di vista parte dall'osservazione che una sequenza casuale non sembra essere regolata da una struttura precisa. L'assenza di pattern sembra quindi un criterio su cui basare una nozione di casualità.

Per formalizzare questa **assenza di regolarità**, consideriamo una stringa w in un certo alfabeto Σ e fissiamo una macchina di Turing universale M [22]. Definiamo la **complessità di Kolmogorov** $K(w)$ della stringa come la lunghezza del più corto programma per M che produce in output la stringa w e poi termina [13].

Ad esempio, la prima stringa all'inizio di questa sezione può essere prodotta da un programma **corto** che essenzialmente dice:

```
stampa '01' dieci volte
```

In altre parole, sfruttiamo le regolarità e la struttura della stringa per descriverla in maniera concisa. Intuitivamente, invece, una stringa prodotta in modo perfettamente casuale non avrebbe nessuna regolarità del genere, e l'unico modo per descriverla sarebbe esibirla per intero.

Questo ci porta, almeno intuitivamente, ad una utile caratterizzazione. Stringhe prive di regolarità, e quindi apparentemente casuali, non possono essere

compresse con efficacia.[2] Più formalmente, si definisce una **stringa algoritmicamente casuale**, o **stringa c-incomprimibile**, una stringa w tale che:

$$K(w) \geq |w| - c \quad \text{per qualche costante } c$$

Ora è necessario legare il concetto di stringa algoritmicamente casuale con il concetto di casualità che stavamo cercando di testare. In un famoso articolo del 1966, Martin-Löf [15] dimostra che una stringa c-incomprimibile presenta tutte le caratteristiche di casualità testabili da qualunque test statistico **effettivamente calcolabile**, per opportune definizioni. Esibisce quindi un criterio universale basato sul valore di $K(w)$ che permette di stabilire l'incomprimibilità di una sequenza.

Una stringa incomprimibile è quindi casuale nel senso che supera qualsiasi test statistico le si possa applicare. La presenza di regolarità non può quindi essere esclusa con certezza, ma si possono escludere tutte quelle per le quali esiste un test. Ciò comprende anche i criteri di k-distribuzione accennati in precedenza.

A questo punto va sottolineato, però, che la funzione $K(w)$ **non è calcolabile**, e quindi non può essere usata direttamente per questo test. Infatti se lo fosse, si potrebbe costruire un programma P che, tramite una ricerca estesa, trova e produce in output una stringa di complessità arbitrariamente grande. Ma qui si giunge ad una contraddizione perchè una tale stringa sarebbe di complessità bassa, perchè prodotta da un algoritmo corto come P.

Sono però disponibili, in realtà, parecchi algoritmi di compressione **asintoticamente ottimi**, come il famoso algoritmo Ziv-Lempel [30]. Questi algoritmi, al crescere della lunghezza del testo, permettono di ottenere un tasso di compressione che tende al limite minimo teorico fissato dall'entropia del testo stesso.

L'idea è quindi quella di comprimere una sequenza con un algoritmo di compressione asintoticamente ottimo, ed etichettarla come non-casuale in caso di successo. Il **test universale di Maurer** [20] funziona proprio in base a questa idea, ma invece di comprimere effettivamente la stringa, calcola il tasso di compressione che si otterrebbe.

È chiaro che questo test non è universale quanto quello teoricamente proposto da Martin-Löf, per il semplice motivo che gli algoritmi di compressione a disposizione sono ottimi solo in senso asintotico. D'altronde non può che essere così, altrimenti avremmo un modo per calcolare la complessità di Kolmogorov di una stringa. Maurer, tuttavia, caratterizza precisamente la classe di difetti statistici intercettabili da questo test, che risulta essere incredibilmente ampia. Il test di Maurer è anche relativamente efficiente e semplice da implementare.

3 Lo standard minimo: generatore di Lehmer

I generatori **congruenziali lineari**, detti anche generatori di **Lehmer** [11] [9], sono una categoria di PRNG molto usata e discretamente efficace. Il loro successo deriva da una relativa facilità di implementazione, una teoria solida alle

[2] Questa conclusione è simile a quanto si ottiene nella classica teoria dell'informazione partendo dal concetto di **entropia** formulato da Shannon in [27]. I due concetti sono legati, ma qui si adotta un punto di vista algoritmico invece che sintattico. Per dettagli si veda [7].

spalle ma relativamente semplice da comprendere, e dalle buone caratteristiche di casualità che si possono ottenere, sufficienti nelle applicazioni meno impegnative (anche se sicuramente sorpassate da algoritmi più recenti come il Mersenne Twister, di cui si parlerà nella sezione 4).

A partire dal seme X_0 (intero positivo), la sequenza di valori $\{X_n\}$ si ottiene da questa semplice relazione:

$$X_{n+1} = aX_n + c \,(\mathrm{mod}\, m) \tag{1}$$

I parametri che compaiono nella formula, tutti interi maggiori di zero, sono:

- Il **modulo** m
- Il **moltiplicatore** a
- L'**incremento** c

Ci si può tranquillamente restringere alla scelta di moltiplicatore, incremento e seme **minori** del modulo m.

Questi parametri caratterizzano il comportamento del generatore e vanno scelti accuratamente. Alcuni casi particolari si possono escludere già a priori, ad esempio con $a = 1$ e $a = 0$ la sequenza si riduce ad una serie di numeri tutti equidistanti.

3.1 Scelta dei parametri

Il primo parametro su cui conviene concentrarsi è il **modulo**, che deve essere sufficientemente grande in quanto il periodo della sequenza sicuramente non potrà essere maggiore di m.

Su computer binari la scelta più naturale sembrerebbe una grande potenza di due, in quanto l'operazione di modulo si ridurrebbe ad un mask dei bit meno significativi. Sembrerebbe addirittura più conveniente scegliere il modulo in base alla dimensione della parola dell'architettura su cui si lavora, perchè sfruttando a proprio vantaggio l'overflow si può addirittura risparmiare del tutto l'operazione di modulo. Tuttavia, queste scelte portano a dei problemi.

Per accorgercene prendiamo un numero d divisore di m e definiamo:

$$Y_n = X_n \,(\mathrm{mod}\, d) \tag{2}$$

Partendo dalla 1 sappiamo che $X_{n+1} = aX_n + c + km$ per qualche k, e prendendo entrambi i membri in modulo d, otteniamo:

$$\begin{aligned}
Y_{n+1} &= aX_n + c + km && (\mathrm{mod}\, d) && \\
Y_{n+1} &= aX_n + c && (\mathrm{mod}\, d) && \text{perchè } m \text{ è multiplo di } d \\
Y_{n+1} &= a(Y_n - k'd) + c && (\mathrm{mod}\, d) && \text{perchè } Y_n = X_n + k'd \\
Y_{n+1} &= aY_n + c && (\mathrm{mod}\, d) &&
\end{aligned}$$

In un computer binario, i j bit meno significativi della sequenza $\{X_n\}$ sono proprio la sequenza $Y_n = X_n \,(\mathrm{mod}\, 2^j)$, e questo risultato ci dice quindi che scegliendo come modulo m una potenza di due, i j bit meno significativi formeranno una sequenza congruenziale lineare di periodo al massimo 2^j. Ad esempio, i 4 bit meno significativi di una sequenza di questo tipo si ripeteranno

almeno una volta ogni 16. Questo comportamento è sicuramente indesiderabile, ed esclude quindi la possibilità di scegliere come modulo una potenza di 2.

Anche escludendo le potenze di 2, questo fatto è da tenere in considerazione nella scelta del modulo, in quanto i residui modulo d dei valori della sequenza per ogni d divisore di m si ripeteranno con un periodo potenzialmente molto più corto di quello totale, e ciò può essere indesiderabile in ugual misura (ad esempio, con un modulo pari la sequenza potrebbe alternarsi strettamente tra numeri pari e dispari). La soluzione può essere quella di scegliere come modulo un numero primo, ad esempio un grande numero primo rappresentabile comunque all'interno di un intero di macchina. Ad esempio, una scelta molto comune su architetture a 32bit è il numero primo $m = 2^{31} - 1 = 2147483647$.

Per quanto riguarda l'incremento, ci concentreremo sul caso particolare in cui $c = 0$, che è il caso originariamente preso in considerazione da Lehmer. In questo caso, il generatore è chiamato **congruenziale moltiplicativo**.

È chiaro che avendo scelto $c = 0$ ci siamo esclusi la possibilità che nella sequenza compaia il numero zero, che farebbe azzerare tutti i valori successivi. Il periodo non potrà quindi essere esattamente uguale a m. In generale, per qualsiasi divisore d di m, se un valore X_n dovesse essere multiplo di d, tutti i valori successivi sarebbero multipli di d, e ciò degraderebbe di molto le caratteristiche di "casualità" della sequenza. Vogliamo dunque ottenere nella sequenza solo valori X_n **coprimi** con m, e ciò vale anche per il seme X_0. Se m è un numero primo, tutti gli $m - 1$ valori minori di m sono coprimi, e quindi si ha la possibilità di raggiungere un periodo lungo $m - 1$, che è a tutti gli effetti un periodo pieno (oltre a non doversi preoccupare della scelta del seme). Se invece il modulo m è composto, il periodo massimo raggiungibile diminuisce.[3]

Questo ragionamento conferma ulteriormente l'opportunità di scegliere come modulo un numero primo.

Resta dunque da considerare la scelta del moltiplicatore. Essendoci ridotti al caso di un modulo primo, la vita è più semplice. Per ottenere il periodo massimo raggiungibile di $m - 1$ è necessario che per a valga che iterando le potenze a^i per $m - 1$ volte si ottengano tutti gli elementi tra 1 e $m - 1$.

In altre parole, vogliamo che a sia un **generatore** del **gruppo moltiplicativo** degli interi modulo m. Dai primi risultati di teoria dei gruppi[4], si deduce che se m è primo, qualsiasi $a < m$ ha questa proprietà, e quindi siamo liberi nella scelta.

In Knuth [9] vengono analizzati i criteri per la scelta di moltiplicatore ed incremento anche nel caso generale di un modulo non primo e di incremento $c > 0$, per ottenere in ogni caso il massimo periodo possibile.

Tuttavia, la lunghezza del periodo è solo uno dei fattori da tenere in considerazione. Una volta isolate le possibili scelte di parametri che garantiscono il periodo richiesto, bisogna fare i conti con le proprietà statistiche delle sequenze generate. Ad esempio, la scelta di $a = 2$ (ogni valore è il doppio di quello precedente) produce sicuramente una sequenza inadeguata. In [23], Park e Miller citano un numero di lavori antecedenti in cui vengono studiate le caratteristi-

[3] Il numero di elementi minori di e coprimi con m è dettato dalla **funzione di Eulero** e si indica $\phi(m)$. Per un numero composto $m = pq$, un breve ragionamento combinatorio fa risultare che $\phi(m) = \phi(p)\phi(q) = (p-1)(q-1)$ (vedi [6]).

[4] In particolare, $\mathbb{Z}/_{p\mathbb{Z}}$ è un gruppo ciclico, ed essendo di ordine primo, per il teorema di Lagrange non ha sottogruppi, per cui qualsiasi elemento è un generatore. Vedi [6].

che statistiche di una serie di parametri per questo tipo di PRNG. Il loro scopo era quello di suggerire un generatore implementabile in modo efficiente con artimetica a 32bit, per cui la scelta del modulo ricadde su $m = 2^{31} - 1$ come già accennato in precedenza. Tenuto poi conto dei risultati di vari test statistici e di altre questioni di efficienza implementativa, il loro suggerimento per il moltiplicatore è $a = 7^5 = 16807$.

In definitiva, Park e Miller sostengono che lo **standard minimo** per la scelta di un PRNG sia costituita dal generatore:

$$X_{n+1} = 7^5 X_n \,(\mathrm{mod}\, 2^{31} - 1)$$

in quanto si tratta di un generatore con un periodo sufficientemente lungo per molte applicazioni, con buone caratteristiche di casualità e implementabile in modo molto efficiente e portabile su qualunque architettura a 32bit.

Dall'epoca dell'articolo di Park e Miller, sono cambiate alcune cose. Innanzitutto, la disponibilità di architetture a 64bit permette in teoria di ottenere periodi più lunghi mantenendo il requisito dell'efficienza, anche se la ricerca esaustiva di parametri con caratteristiche statistiche ottimali diventa molto più difficile. In ogni caso, la possibilità di usare efficientemente aritmetica a 64bit rende superflui alcuni accorgimenti tecnici necessari per evitare overflow su macchine a 32bit, presenti nell'implementazione proposta nel loro articolo. Inoltre, molti progressi sono stati fatti riguardo altri tipi di PRNG con caratteristiche di casualità e di lunghezza del periodo molto migliori. Il Mersenne Twister, di cui si parlerà nella prossima sezione, è uno degli esempi più recenti del risultato di queste ricerche, ed è molto vicino allo stato dell'arte del settore.

Ora ci si soffermerà brevemente su alcuni aspetti del comportamento statistico dei generatori di Lehmer, compreso quello di Park e Miller, per capirne i limiti e avere un punto di partenza con cui confrontare le performance del Mersenne Twister.

3.2 Equidistribuzione dei generatori di Lehmer

I generatori congruenziali lineari soffrono di un problema intrinseco che limita le loro caratteristiche di equidistribuzione. Per illustrarlo prendiamo come esempio direttamente da Knuth [9] un generatore con periodo sicuramente inadeguato nella pratica, ma abbastanza corto da permetterci di esaminare l'intera sequenza. Il generatore in questione è:

$$X_{n+1} = 137 X_n + 187 \,(\mathrm{mod}\, 256) \tag{3}$$

In figura 1 vengono riportate tutte le triple di valori consecutivi estratti dall'intera sequenza di 256 valori prodotti da questo generatore. Secondo il criterio di equidistribuzione introdotto nella sezione 2.2, ogni sottointervallo dello spazio tridimensionale dovrebbe contenere un numero di punti proporzionale al volume dell'intervallo. Si può intravedere che ciò non accade, anzi i punti sembrano allinearsi su un piccolo numero di piani.

Questo comportamento è stato scoperto e analizzato da Marsaglia in [14], dove si dimostra che si tratta di una conseguenza diretta del funzionamento dei generatori congruenziali e non può essere evitato. Più in particolare, ha dimostrato che le k-uple di valori consecutivi si allineano su un certo numero di piani $(k-1)$-dimensionali, e il numero di piani è sempre minore o uguale

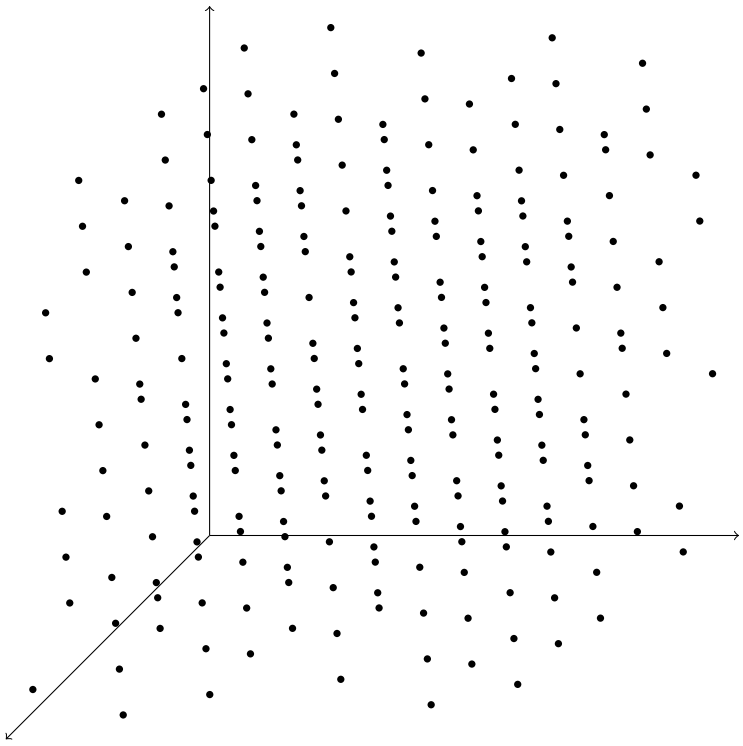

Figura 1: Triple di valori consecutivi prodotte dal generatore definito in (3)

a $\sqrt[k]{k!m}$, dove m è il modulo. In questo esempio, i piani su cui giacciono triple di valori consecutivi sono quindi al massimo 11. Per scelte del modulo più alte il numero di piani cresce, ad esempio sale a 2344 se $m = 2^{31} - 1$ (per $k = 3$). Questo numero può sembrare alto ma su un ampiezza di due miliardi in realtà non lo è, e a seconda delle applicazioni può risultare troppo basso. Inoltre, si tratta solo di un limite superiore e nella pratica, a seconda di moltiplicatore ed incremento, il numero effettivo di piani può essere anche molto minore. Ad esempio, il generatore di numeri casuali `RANDU`, implementato da IBM negli anni '70 nei suoi mainframe della serie S/360, è rimasto alla storia come uno dei peggiori PRNG mai implementati, perchè aveva un modulo di 2^{32} ma, oltre ad avere un periodo in realtà molto più basso a causa di una scelta infelice del moltiplicatore, produceva sequenze in cui triple di valori consecutivi si allineavano su solamente 15 piani distinti (vedi [23]).

A volte, soprattutto quando il numero k di dimensioni di cui si tiene conto sale, è sufficiente richiedere che solo i primi v bit più significativi dei valori della sequenza risultino k-distribuiti. In questo caso, si dice che la sequenza è k-distribuita con **precisione** di v bit.

La massima precisione ottenibile dipende dal numero di piani su cui si allineano i valori della sequenza, e quindi dalla distanza tra i piani. Considerando solo i v bit più significativi, si raggruppano i punti distanti tra di loro al più 2^{w-v}, dove $w = \log_2 m$ è la lunghezza in bit dei valori estratti dal generatore. Più la distanza tra i piani diminuisce, meno bit è necessario scartare per fare in

modo che la distanza minima tra i punti sia uniforme e che i punti siano quindi equidistribuiti.

Quindi per aumentare la precisione di k-distribuzione di una sequenza è necessario aumentare il numero di piani su cui si allineano i punti e quindi aumentare il modulo. Se non ci si accontenta della precisione ottenuta è quindi necessario aumentare di molto il modulo del generatore, o passare a generatori di tipo diverso.

Esiste un test, chiamato **test spettrale**, pensato specificatamente per i generatori congruenziali, che misura la precisione con cui una sequenza risulta k-distribuita per determinati k (vedi [9] per i dettagli).

4 Verso lo stato dell'arte: il Mersenne Twister

Il **Mersenne Twister** è un generatore di numeri casuali relativamente recente, proposto nel 1998 in [18], con caratteristiche molto interessanti. Nella sua forma più comune, conosciuta come `MT19937`, il Mersenne Twister ottiene un lunghissimo periodo pari a $2^{19937} - 1$ (un numero primo di Mersenne per l'appunto), con una equidistribuzione su ben 623 dimensioni, su tutti i 32bit dei valori prodotti. Inoltre, è facile ottenere un'implementazione molto veloce sfruttando le caratteristiche delle architetture moderne.

La comprensione profonda del Mersenne Twister richiede concetti di algebra dei campi finiti che non possono essere esaustivamente riassunti in un articolo come questo. Qui, questi concetti verranno solo accennati quanto basta per avere un'idea generale del funzionamento dell'algoritmo. Per un riferimento riguardo agli aspetti di algebra di base e algebra lineare si può vedere [6] e [3].

Il Mersenne Twister è in ultima analisi un'evoluzione di una classe di generatori chiamati **generalized feedback shift register** (GFSR), introdotti nel 1973 in [12]. Questi ultimi sono stati migliorati nei primi anni '90 da un tipo di generatori chiamati **Twisted GFSR** [16] [17], proposti dagli stessi autori che qualche anno più tardi proposero il Mersenne Twister come ulteriore raffinamento dei TGFSR. Questa sarà anche la strada che percorreremo in questa panoramica.

4.1 Ricorrenze lineari e GFSR

Consideriamo una sequenza di x_i prodotta da una generica ricorrenza lineare del tipo:

$$x_n = c_1 x_{n-1} + c_2 x_{n-2} + \ldots + c_d x_{n-d} \tag{4}$$

dove ogni elemento della sequenza dipende al più dai d elementi precedenti.

I **generalized feedback shift register** si basano su una ricorrenza lineare di questo tipo, considerando però sequenze di singoli bit e operazioni aritmetiche **bitwise**, e quindi operazioni sul campo $\mathbb{Z}_2$ degli interi modulo 2. In $\mathbb{Z}_2$, gli unici due elementi sono 0 e 1 e risulta che somma e prodotto modulo 2 corrispondano semplicemente alle operazioni logiche di XOR e di AND, rispettivamente.

Nei GFSR, quindi, i valori x_i e i coefficienti c_i appartengono a $\mathbb{Z}_2$. Ognuno dei d valori precedenti a x_n è quindi sommato se il valore del rispettivo coefficiente è uguale a 1 oppure ignorato in caso contrario. Il nome di questi

generatori deriva da quello del circuito logico che può essere utilizzato per implementarli, con un registro a scorrimento di d bit in cui l'input viene calcolato con un'operazione di XOR i cui input sono i bit del registro corrispondenti a coefficienti c_i uguali a 1.

Partendo dunque dal seme, costituito dai valori iniziali $x_0, \ldots, x_{d-1}$, e iterando la ricorrenza, viene generata una sequenza pseudocasuale di bit. Su un architettura di calcolo con parole lunghe w bit (32 o 64, di solito), l'operazione di base del GFSR può essere eseguita in parallelo su tutti i bit della parola, generando quindi sequenze pseudo-casuali di numeri interi di lunghezza w (l'aggettivo "generalized" nel nome deriva proprio da questa caratteristica di parallelizzazione).

Purtroppo, in questo generatore la scelta del seme è critica. Ovviamente, le singole sequenze vanno inizializzate con semi diversi per non avere parole formate da bit tutti uguali, ma non tutti i semi vanno ugualmente bene. Ogni bit della parola proviene da fasi diverse della stessa sequenza, per cui è necessario evitare che le sequenze di bit diversi della parola vengano inizializzati non solo con lo stesso seme, ma anche con semi troppo vicini tra di loro, per evitare un'eccessiva correlazione tra diversi bit della stessa parola. Bisogna dunque assicurare un minimo **delay** tra le sequenze dei singoli bit della parola, tramite un'opportuna procedura di inizializzazione (vedere [12] per i dettagli).

4.1.1 Periodo delle sequenze

La ricorrenza (4) si può anche rappresentare con una matrice $d \times d$ del tipo:

$$C = \begin{pmatrix} 0 & 1 & & \\ & \ddots & \ddots & \\ & & 0 & 1 \\ c_d & c_{d-1} & \cdots & c_1 \end{pmatrix} \tag{5}$$

Infatti, dato il vettore $x = (x_{n-d} \ldots x_{n-1})$, si verifica subito che il prodotto con la matrice C produce proprio un nuovo vettore formato da x_n e i $d-1$ elementi precedenti:

$$Cx = \begin{pmatrix} 0 & 1 & & \\ & \ddots & \ddots & \\ & & 0 & 1 \\ c_d & c_{d-1} & \cdots & c_1 \end{pmatrix} \begin{pmatrix} x_{n-d} \\ \vdots \\ x_{n-2} \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} x_{n-d+1} \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix}$$

Questa matrice è già in **forma normale**, per cui ricavarne il **polinomio caratteristico**[5] è immediato, e risulta semplicemente:

$$p_C(t) = t^d - c_d t^{d-1} - \ldots - c_1$$

che, tenuto conto che in $\mathbb{Z}_2$ somma e differenza risultano essere equivalenti, si può scrivere anche così:

$$p_C(t) = t^d + c_d t^{d-1} + \ldots + c_1 \tag{6}$$

[5]Il polinomio caratteristico di una matrice C, definito come $p(x) = \det(xI - C)$, è il poliniomo le cui radici sono tutti e soli gli **autovalori** di C

La scelta di una sequenza con un polinomio caratteristico adatto è fondamentale per ottenere un generatore con un buon comportamento. Infatti, essendo le radici del polinomio caratteristico gli **autovalori**[6] della matrice, è necessario scegliere un polinomio che sia **irriducibile** in $\mathbb{Z}_2$ per evitare che la sequenza degeneri su un autovettore, ovvero un vettore x di valori $(x_n \cdots x_{n-d})$ per cui $Cx = \lambda x$ per qualche λ. Un comportamento del genere farebbe crollare le qualità statistiche della sequenza.

Anche la lunghezza del periodo del generatore dipende dal polinomio caratteristico. Si può dimostrare [12] che se il polinomio caratteristico è **primitivo**[7] di grado d, la sequenza ha periodo $2^d - 1$.

La difficoltà sta quindi tutta nel trovare dei polinomi primitivi di grado alto, attività non facile dovuta alla scarsità di tali polinomi e al costo computazionale richiesto.

Oltre a buone caratteristiche di equidistribuzione, il vantaggio principale dei GFSR è la possibilità di ottenere periodi molto lunghi utilizzando sempre aritmetica di macchina. Ad esempio, con il polinomio primitivo $t^{532} + t^{37} + 1$ si raggiunge un periodo di ben $2^{532} - 1$. Dal punto di vista dell'efficienza, le operazioni coinvolte ad ogni passo sono poche e molto efficienti, ma la procedura di inizializzazione del seme è dispendiosa.

L'uso di memoria invece è un fattore importante in questo tipo di generatori. Per come abbiamo derivato il polinomio caratteristico, è chiaro che il grado d del polinomio, da cui dipende il periodo, corrisponde al numero di elementi della sequenza che devono essere mantenuti in memoria ad ogni passo. Ottenere un periodo lungo a piacere è quindi possibile ma a fronte di un utilizzo di memoria considerevole.

4.2 Twisted GFSR

I **Twisted GFSR** eliminano la necessità di una fase di inizializzazione, migliorano le già buone qualità di equidistribuzione, e aumentano ulteriormente il periodo raggiungiubile a parità di quantità di memoria utilizzata per lo stato interno.

Il vantaggio più significativo dal punto di vista della praticità di utilizzo è l'aumento del periodo a parità di memoria utilizzata. Abbiamo visto che i GFSR ottengono un periodo di $2^n - 1$ utilizzando n parole di memoria. Nonostante si tratti di un periodo molto lungo, non è il massimo teoricamente raggiungibile, ovvero $2^{nw} - 1$, il numero totale di stati rappresentabile da n parole di w bit ciascuna, escluso lo zero. I TGFSR ottengono proprio questo periodo, e quindi diminuiscono di un fattore w (come minimo 32 oggigiorno, quindi non proprio irrilevante) la memoria utilizzata, a parità di periodo.

D'ora in poi, con $\mathbf{x}$ si intenderanno vettori di lunghezza w (la dimensione di una parola). Il funzionamento dei TGFSR può essere riassunto da una ricorrenza di questo tipo, per $l = 0, 1, 2, \ldots$

$$\mathbf{x}_{l+n} = \mathbf{x}_{l+m} - \mathbf{x}_l A \tag{7}$$

[6] Data una matrice A, un vettore v e uno scalare λ, se $Av = \lambda v$ si dice che v è un **autovettore** e λ un **autovalore** della matrice A

[7] Un polinomio con coefficienti in $\mathbb{F}_p$ è **primitivo** se ha una radice $\alpha \in \mathbb{F}_{p^k}$ le cui potenze generano tutto $\mathbb{F}_{p^k}$ ed è il polinomio con il grado minimo a possedere questa radice. Essendo un polinomio minimo, è anche irriducibile.

dove $n > m$, A è una matrice $w \times w$ su $\mathbb{Z}_2$, a partire dal seme iniziale $\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}$.

La differenza rispetto ad un GFSR è quindi il prodotto dell'ultimo termine della ricorrenza per questa matrice A, da cui il termine "twisted", che tra le altre cose elimina la necessità di un'inizializzazione particolare del seme, necessaria nei GFSR.

Oltre alla matrice, i parametri m ed n che compaiono nella definizione (7) sono fondamentali per ottenere un buon comportamento della sequenza, e in particolare per la determinazione del periodo.[8]

Infatti, indicando con $p_A(t)$ il polinomio caratteristico della matrice A, si dimostra che il polinomio caratteristico dell'intera sequenza è il polinomio $p_A(t^n + t^m)$. Come prima dunque, se questo polinomio è primitivo di grado k, la sequenza ha periodo $2^k - 1$. Ma essendo A una matrice $w \times w$, $p_A(t)$ ha grado w, e il suddetto polinomio ha quindi grado nw, portando ad un periodo di esattamente $2^{nw} - 1$.

Il tutto si riconduce quindi nuovamente all'individuazione di un polinomio primitivo di grado sufficientemente alto. Nel farlo, il grado del polinomio $p_A(t)$ è di solito determinato dalla dimensione w delle parole dell'architettura che si ha a disposizione, mentre si può giocare di più sul valore di n ed m.

La necessità di trovare polinomi primitivi di grado alto è il principale fattore limitante nella progettazione di TGFSR con periodo molto alto, a causa della complessità computazionale del problema. In [16] si riassume il procedimento. Una volta fissati w ed n, occorre trovare un polinomio irriducibile $p_A(t)$ di grado w, e poi per ogni $m \in \{1, \ldots, n-1\}$, testare la primitività di $p_A(t^n + t^m)$. Il problema è che per il test della primitività è necessario scomporre in fattori primi il numero $2^{nw} - 1$, problema che diventa velocemente intrattabile al crescere di n, visto che non conosciamo ancora algoritmi efficienti di fattorizzazione. Il Mersenne Twister innova proprio su questo aspetto, aggirando il problema della fattorizzazione e permettendo di trovare in tempi brevi i parametri necessari ad ottenere periodi incredibilmente lunghi.

Oltre al periodo, i Twisted GFSR migliorano anche il comportamento statistico delle sequenze prodotte. In [17] si citano molti lavori in cui la qualità delle sequenze viene messa in relazione con il numero di termini non nulli del polinomio caratteristico. In un TGFSR, si riscontra che il numero dei termini del polinomio caratteristico è considerevole.

Si dimostra, inoltre, che il valore di $k(v)$, ovvero il numero k per cui la sequenza prodotta risulta k-distribuita con precisione di v bit, è un multiplo di n, e inoltre vale il seguente limite superiore:

$$k(v) \leq n \left\lfloor \frac{w}{v} \right\rfloor$$

Quindi, volendo una precisione massima con $v = w$, l'equidistribuzione si può ottenere per un massimo di n dimensioni, fatto che aggiunge un criterio in più nella scelta di questo parametro. Si tratta comunque solo di un limite superiore, che può benissimo rivelarsi troppo largo. Per assicurarsi quindi un effettiva n-distribuzione con la voluta precisione, si esegue un'ulteriore fase, chiamata di **tempering**, che assicura il raggiungimento del limite superiore.

[8] La definizione (7) usa gli indici in modo diverso rispetto alla notazione utilizzata finora, con la quale la definizione sarebbe stata $\mathbf{x}_n = \mathbf{x}_{n-k_1} + \mathbf{x}_{n-k_2} A$, e i coefficienti n ed m usati qui sarebbero stati k_2 e $k_2 - k_1$, rispettivamente, appesantendo la notazione

La sequenza si evolve secondo la ricorrenza indicata in (7), ma i valori $\mathbf{x}_l$ vengono trasformati prima di essere prodotti in output, per migliorare l'equidistribuzione della sequenza. La trasformazione avviene tramite una opportuna matrice T, progettata per ottenere il desiderato effetto sull'equidistribuzione della sequenza, ma in modo che sia possibile implementare il prodotto con poche ed efficienti operazioni bitwise. Queste operazioni vengono ereditate dal Mersenne Twister ma leggermente modificate, per cui verranno illustrate in seguito.

Un'ultima cosa che merita di essere accennata riguarda il seme. Nei TGFSR, il problema della correlazione tra i diversi bit dei valori prodotti non sussiste più, grazie all'operazione di twisting. L'intero seme, d bit per ogni bit della parola, quindi in tutto una matrice $d \times w$, può quindi essere generato a partire da un seme più piccolo, di solito una singola parola di w bit, magari tramite un generatore più semplice.

4.3 Mersenne Twister

Così com'è, il TGFSR sarebbe virtualmente perfetto: un algoritmo capace di ottenere un periodo arbitrariamente lungo con ottime proprietà di k-distribuzione con precisione arbitraria. Purtroppo, la possibilità di ottenere periodi ed equidistribuzioni arbitrariamente alti è limitata dalla complessità computazionale del problema di fattorizzazione in numeri primi di grandi numeri interi, di cui ancora non conosciamo un limite inferiore. Infatti, già per $p > 2000$ fattorizzare $2^p - 1$ senza un algoritmo efficiente è improponibile (tanto che, ad esempio, RSA consiglia chiavi da 2048 bit per documenti che debbano restare sicuri fino al 2030 [1]).

Non avendo ancora a disposizione un algoritmo efficiente per questo problema, il Mersenne Twister aggira l'ostacolo garantendo che il periodo sia un numero primo, evitando quindi di doverlo fattorizzare per testare la primitività del polinomio caratteristico. In particolare, si garantisce che il periodo sia un numero primo di Mersenne, ovvero un numero primo nella forma $2^p - 1$, con p primo a sua volta, da cui l'algoritmo prende il nome.

Ciò si ottiene modificando la ricorrenza (7) nel modo seguente. Una volta scelto un intero r compreso tra 0 e w, si definisce $\mathbf{x}^l$ come la sequenza degli r bit meno significativi del vettore $\mathbf{x}$, e $\mathbf{x}^u$ i restanti $w - r$ bit più significativi. La ricorrenza su cui si basa il Mersenne Twister è quindi, per $k = 0, 1, \ldots$, la seguente:

$$x_{k+n} = x_{k+m} + (x_k^u | x_{k+1}^l)A \tag{8}$$

dove $|$ rappresenta la parola ottenuta dalla concatenazione delle due parti.

In altre parole, la matrice A viene moltiplicata per una parola ottenuta concatenando i $w - r$ bit più significativi di $\mathbf{x}_k$, con quelli meno significativi di $\mathbf{x}_{k+1}$. Facendo in questo modo, il numero totale di bit da cui dipende la parola non è più nw ma diventa $nw - r$, perchè gli r bit meno significativi di $\mathbf{x}_k$ vengono ignorati. Come viene quindi dimostrato nell'articolo originale [18], il polinomio caratteristico della sequenza diventa di grado $nw - r$, e il periodo massimo ottenibile risulta quindi pari a $2^{nw-r} - 1$.

Scegliendo dunque n, w e r in modo da ottenere come periodo un primo di Mersenne, si evita di doverlo fattorizzare per testare la primitività del polinomio, e anzi, sempre in [18] viene presentato un algoritmo, chiamato **inverse-**

decimation method, per valutare la primitività di un polinomio di grado un primo di Mersenne $2^p - 1$ in tempo $O(p^2)$, potendo di conseguenza calcolare dei valori adatti per m in tempi ragionevoli.

Come il TGFSR, anche il Mersenne Twister si affida a delle operazioni di tempering successive per migliorare l'equidistribuzione della sequenza. Come già accennato, il tempering corrisponde al prodotto del valore $\mathbf{x}$ per una matrice T appositamente scelta. Il tutto è pensato però per poter essere implementato con poche operazioni bitwise. Infatti, il prodotto $\mathbf{z} = \mathbf{x}T$ equivale alle seguenti operazioni:

$$\begin{aligned}
\mathbf{z} &:= \mathbf{x} + (\mathbf{x} >> u) \\
\mathbf{z} &:= \mathbf{z} + ((\mathbf{z} << s) \,\&\, \mathbf{b}) \\
\mathbf{z} &:= \mathbf{z} + ((\mathbf{z} << t) \,\&\, \mathbf{c}) \\
\mathbf{z} &:= \mathbf{z} + (\mathbf{z} >> l)
\end{aligned}$$

Dove u, s, t, l sono interi da 1 a w e $\mathbf{b}$ e $\mathbf{c}$ sono due parole di dimensione w. Questi parametri vanno scelti in modo da massimizzare le proprietà di equidistribuzione della sequenza.

In [18] viene presentato un algoritmo per il calcolo del valore di $k(v)$ in tempo $O(v^4 p^2)$. I parametri utilizzati per costruire la matrice di tempering possono quindi venire cercati esaustivamente tramite questo algoritmo per ottenere un valore di $k(v)$ più alto possibile per ogni v.

4.4 Implementazione e MT19937

Si possono fare alcune osservazioni per ottenere un'implementazione efficiente. Innanzitutto, non è necessario esprimere la matrice A esplicitamente. Una volta scelto un polinomio adatto, la matrice A può essere costruita in forma normale, in modo da avere automaticamente il polinomio caratteristico desiderato, come nella sottosezione 4.1.1. Quindi, l'intera matrice si può determinare essenzialmente con i coefficienti $\mathbf{a} = a_{w-1}, \ldots, a_0$ del polinomio caratteristico. Inoltre, sfruttando la struttura della matrice si può implementare il prodotto in due sole operazioni bitwise. Infatti, si può riscrivere nel seguente modo:

$$\begin{aligned}
\mathbf{x}A &= (x_{w-1}, \ldots, x_0) \cdot \begin{pmatrix} 0 & 1 & & \\ & \ddots & \ddots & \\ & & 0 & 1 \\ a_{w-1} & a_{w-2} & \cdots & a_0 \end{pmatrix} \\
&= (0 + x_0 a_{w-1}, x_{w-1} + x_0 a_{w-2}, \ldots, x_1 + x_0 a_0) \\
&= (\mathbf{x} >> 1) + x_0 \mathbf{a}
\end{aligned}$$

Per inciso, la necessità di scegliere la matrice A in questo modo è uno dei fattori che rende necessaria la successiva fase di tempering. Infatti una matrice diversa, anche se con lo stesso polinomio caratteristico, può portare a evoluzioni più caotiche dei bit della parola e quindi a migliori proprietà di equidistribuzione. Restringendosi solo a matrici in questa forma ci si preclude la possibilità di ottenere l'ottimalità in questo senso, e quindi si rende necessaria una correzione successiva.

La variante più comune del Mersenne Twister è chiamata MT19937, nome che si riferisce al periodo di $2^{19937} - 1$. Questa variante produce valori di 32 bit, per cui $w = 32$, e di conseguenza $n = 624$ e $r = 31$ ($19937 = 624 \cdot 32 - 31$). La tabella 1 mostra tutti i parametri che gli autori suggeriscono per questa variante. Il valore dei vettori binari **a**, **b** e **c** vengono indicati in esadecimale.

Parametro Periodo	Valore	Parametro Tempering	Valore
n	624	u	11
w	32	s	7
r	31	t	15
m	397	l	18
a	$9908B0DF_{16}$	**b**	$9D2C5680_{16}$
		c	$EFC60000_{16}$

Tabella 1: Parametri per MT19937

4.5 Conclusioni e ulteriori sviluppi

Il Mersenne Twister ha delle caratteristiche migliori di tutti i generatori fino ad allora conosciuti, ma la ricerca non si è fermata li. Avendo un periodo così lungo, l'algoritmo è stato anche abbondantemente analizzato sulla base del comportamento statistico di sottosequenze molto più corte, per evidenziare difetti statistici o comportamenti anomali. Il risultato è la classe di generatori chiamata **Well Equidistributed Long-period Linear generators** (WELL in breve), proposta nel 2006 in [21].

Rispetto al Mersenne Twister, i bit delle sequenze generate da WELL evolvono in modo più caotico, nel senso che molti più bit in media vengono cambiati da un'iterazione all'altra. Questo si riflette su una maggiore velocità con cui le sequenze si riprendono da una fase di **zero-excess** causata da valori iniziali del seme contenenti troppi zeri.

Dal punto di vista implementativo, il Mersenne Twister è stato anche adattato per ottenere una maggiore efficienza nelle moderne architetture di calcolo. La variante, conosciuta come **SFMT** (SIMD-oriented Fast Mersenne Twister), sfrutta le istruzioni SIMD (Single Instruction Multiple Data) delle CPU moderne per ottenere alte prestazioni, e incidentalmente migliora anche alcune caratteristiche di casualità delle sequenze prodotte. Vedere [25] per i dettagli.

Un'ultima cosa che merita di essere notata è che, così com'è, il Mersenne Twister (ma anche le evoluzioni successive come WELL), non è adatto ad un uso crittografico a causa della linearità della relazione che lega i valori della sequenza. Sono stati comunque proposti dei generatori adatti ad uso crittografico che basano il proprio funzionamento su una manipolazione della sequenza ottenuta dal Mersenne Twister, mantenendone le caratteristiche statistiche e il lungo periodo, ma eliminando la relazione lineare che lega i valori della sequenza. Vedere [19] per una di queste proposte.

5 Panoramica sui PRNG crittografici

La crittografia è un settore sempre più importante e in continua crescita. Il requisito di riservatezza e affidabilità delle comunicazioni, sempre più essenziale nella nostra era dell'informazione, innesca un gioco di guardie e ladri in cui nuove tecniche di protezione di dati e comunicazioni devono resistere a strumenti sempre più avanzati, e una potenza di calcolo sempre crescente, a disposizione di chi vuole aggirare queste protezioni.

Nel funzionamento di molte, importanti tecniche di cifratura dei dati, di protocolli di scambio di chiavi, di autenticazione o di instaurazione di connessioni sicure, compare un passo in cui i soggetti in gioco devono scegliere un valore casuale. Molte di queste tecniche basano la loro sicurezza sull'assunzione che nessuno possa essere in grado di prevedere il valore scelto. Si veda [26] per una classica introduzione all'argomento.

Come accennato nella sezione 2, la **prevedibilità** delle sequenze è un fattore distinto dalle loro caratteristiche statistiche. Ad esempio, le cifre decimali di π notoriamente soddisfano tutti i test statistici che gli siano mai stati applicati, tanto che si congettura che la sequenza sia ∞-distribuita. Tuttavia, mentre usare le cifre di π come sorgente di numeri casuali in una simulazione fisica può essere un'ottima idea, per un'applicazione crittografica è senz'altro sconsigliabile, perchè chi ad un certo punto riuscisse a riconoscerne la fonte potrebbe prevedere tutti i valori successivi e ricostruire anche tutti i precedenti.

La prevedibilità di una sequenza pseudo-casuale può venire essenzialmente da due fonti: la bassa **complessità** della relazione che lega i valori della sequenza, e la prevedibilità del **seme** con cui la sequenza è stata generata.

I PRNG introdotti finora non risultano adatti ad un uso crittografico principalmente per il primo punto. Sia in un generatore di Lehmer che in uno più sofisticato come il Mersenne Twister, ogni valore è legato a quello precedente da una relazione **lineare**. Avendo a disposizione una sottosequenza sufficientemente lunga è quindi possibile impostare un sistema lineare e risolverlo per ottenere lo stato attuale del generatore e/o il seme, da cui poi è possibile ricostruire l'intera sequenza. La prima cosa da fare è quindi quella di dotarsi di algoritmi di generazione di numeri casuali in cui la relazione tra gli elementi della sequenza non sia lineare, ma più complessa possibile. I requisiti da soddisfare si possono riassumere in due punti[9]:

- **Next bit test**
  Data una sequenza di valori, non esiste un algoritmo **polinomiale** in grado di prevedere il valore successivo con una probabilità di successo superiore al 50%. È stato dimostrato [29] che una sequenza che soddisfi questa proprietà passa qualunque altro test statistico eseguibile in tempo polinomiale.

- **Forward security**
  Nel caso in cui lo stato attuale del generatore venisse scoperto in qualche modo, viene compromessa solo la sicurezza della parte successiva

[9] È chiaro che questi requisiti, allo stato attuale della ricerca, spesso vengono soddisfatti in senso lato, cioè basandosi su problemi di cui si hanno buoni motivi per congetturarne l'intrattabilità, in attesa di determinanti sviluppi teorici che la certifichino, come d'altronde succede in molti altri casi nell'ambito della crittografia.

della sequenza. In altri termini, non è possibile risalire in tempo polinomiale ai valori precedenti della sequenza, conoscendo lo stato attuale del generatore.

Un'altro fattore determinante per la prevedibilità dell'intera sequenza è, ovviamente, la prevedibilità del **seme**. Una sequenza di milioni di elementi generati con il più forte dei PRNG crittografici può essere completamente prevedibile se l'imprevedibilità del seme non è garantita in modo altrettanto forte.[10] Si pone quindi il problema di come garantire questa imprevedibilità, tenuto conto sempre dei modelli di calcolo deterministici con cui abbiamo a che fare. Diventa dunque indispensabile valicare i confini della teoria algoritmica e mettere a punto sistemi in grado di raccogliere vera, imprevedibile casualità dal mondo fisico. Bisogna quindi selezionare opportune fonti fisiche di casualità, valutarne le caratteristiche e l'affidabilità, e applicare le tecniche necessarie per combinare fonti diverse. Questi sistemi non sostituiscono, come spesso invece si pensa, i PRNG di natura algoritmica, perchè la quantità di dati che possono produrre o raccogliere è spesso insufficiente o prodotta in modo troppo lento, bensì vengono usati proprio per ottenere un seme genuinamente imprevedibile con cui generare la sequenza vera e propria.

Per riassumere, la generazione di una buona sequenza di numeri casuali adatta in contesti crittografici passa per le seguenti fasi:

- Raccolta di entropia da varie fonti fisiche di casualità.
- Combinazione delle varie fonti in una unica, da cui estrarre il seme per il passo successivo.
- Uso di un PRNG crittograficamente forte per la generazione della sequenza.

Di seguito verranno accennate alcune delle tecniche attualmente raccomandate per ognuna di queste fasi, secondo quanto riportato dall'**Internet Engineering Task Force** nella `RFC4086` [8].

5.1 Raccogliere entropia dall'ambiente

Prima di indagare su quali aspetti del mondo esterno possono essere sfruttati per introdurre nel calcolatore la quantità necessaria di casualità, è opportuno ragionare su come valutare in modo quantitativo l'effettiva imprevedibilità dei dati che raccogliamo.

Gli strumenti teorici per farlo vengono dalla **teoria dell'informazione**, e in particolare dalla definizione di **entropia** data originariamente da Shannon in [27]. L'entropia di una certa sorgente di informazione S è definita a partire dalla probabilità p_i che ogni simbolo i venga emesso dalla sorgente, nel seguente modo:

$$H(S) = -\sum_{i=1}^{k} p_i \log_2 p_i$$

[10] Si noti che, in applicazioni di altro tipo, è molto utile questo aspetto di prevedibilità dei PRNG. In uno studio scientifico, per esempio, è essenziale la capacità di riprodurre esattamente gli stessi risultati ripetendo la simulazione partendo dallo stesso seme.

Il significato di questa definizione può essere interpretato in vari modi,[11] ma forse il più adatto al nostro discorso è quello secondo cui l'entropia corrisponde alla **quantità di informazione** emessa dalla sorgente. In particolare, avendola definita con un logaritmo in base due, l'entropia misura quanti **bit** di informazione sono presenti.

Tutta l'intera sequenza prodotta da un PRNG, ad esempio, che può avere anche un volume di svariati milioni di bit, in realtà contiene una quantità minima di informazione, corrispondente a quella del seme utilizzato per inizializzare il generatore. Se il seme è lungo n bit e tutte le 2^n possibili combinazioni sono equiprobabili, allora l'entropia del seme sarà pari a n (basta porre $p_i = 1/2^k$ nella definizione data sopra per verificarlo) , mentre invece se alcuni valori per il seme sono molto più probabili di altri l'entropia inevitabilmente si abbassa.[12]

Se si usano fonti esterne per raccogliere dati da cui attingere per la scelta del seme, va quindi valutata con attenzione l'entropia di queste sorgenti, perchè il numero di bit di entropia effettivamente forniti può essere anche molto inferiore al volume dei dati emessi in totale.

Ad esempio, tutti i testi o i manuali di programmazione, quando spiegano l'utilizzo delle primitive offerte dal tal linguaggio per la generazione di numeri casuali (la funzione `rand()` del C, ad esempio), consigliano sbrigativamente di utilizzare come seme del generatore il valore attuale dell'orologio di sistema. Per una qualsiasi applicazione può andar bene, ma sicuramente non è sufficiente in un contesto crittografico. Prendendo come esempio un sistema UNIX, il valore dell'orologio di sistema è rappresentato da un valore di 32 bit che conta i secondi trascorsi dalla mezzanotte del primo gennaio 1970. È vero che un valore del genere è in un certo senso imprevedibile, ma il numero di bit effettivamente poco prevedibili corrisponde solo a quelli che rappresentano una forbice temporale di qualche secondo o, al massimo, di qualche minuto, quindi dell'ordine di 11 bit. Inizializzare un PRNG con un seme di questo tipo lascia dunque ad un attacco di forza bruta uno spazio di ricerca di 2^{11} combinazioni, facilmente esauribile in pochissimo tempo.

Questo però non significa che l'orologio di sistema non sia una buona fonte di entropia. Significa semplicemente che l'entropia fornita non è sufficiente. Raramente si avranno a disposizione singole sorgenti sufficienti a fornire tutta l'entropia necessaria, e per questo è opportuno combinarne diverse con le tecniche accennate nella prossima sottosezione.

Si costruisce, cioè, quella che viene chiamata una **entropy pool**, ovvero un buffer in cui i dati provenienti da svariate sorgenti di entropia vengono combinati, e dal quale può venire estratto un seme affidabile.

A seconda del sistema su cui si sta lavorando, le sorgenti a disposizione possono essere molto diverse. L'ideale sarebbe disporre di dispositivi in grado di sfruttare qualche fenomeno fisico legato intrinsecamente a fenomeni aleatori. Molti generatori hardware di entropia utilizzano, ad esempio, la fluttuazione intorno al valore medio della temperatura percepita da un sensore. Altri dispositivi più avanzati vanno oltre, sfruttando proprietà aleatorie di fenomeni

[11] Il nome di "entropia" deriva proprio dall'interpretazione data originariamente da Shannon come quantità di "disordine" del testo, ossia assenza di pattern regolari, in analogia con l'entropia di un sistema termodinamico nel senso dato dalla fisica.

[12] Si può vedere che il valore della funzione $H(S)$ è massimo quando S ha una distribuzione uniforme.

come il decadimento radioattivo. Tuttavia, questi dispositivi sono relativamente costosi e raramente disponibili[13] e bisogna spesso accontentarsi di fonti più facilmente accessibili. Molti sistemi operativi forniscono un proprio entropy pool, che utilizza come sorgenti gli eventi di sistema quali la frequenza degli interrupt, i tempi di latenza delle operazioni di I/O, il susseguirsi di chiamate di sistema dei processi, e così via. In un sistema interattivo si aggiungono fonti come l'input dell'utente tramite mouse e tastiera.

Bisogna tuttavia tenere presente che anche avendo a disposizione molte sorgenti diverse, l'entropia disponibile potrebbe comunque scarseggiare. Ad esempio, un sistema embedded, quale un router, è privo di periferiche di input, privo di dischi fissi meccanici, con un set di processi e di operazioni poco variegato. Un sistema del genere, magari appena acceso, si trova in seria difficoltà nel raccogliere sufficiente entropia per le proprie necessità. È opportuno quindi tenere conto di questi aspetti nel momento in cui si progettano applicativi che fanno affidamento sull'entropy pool fornito dal sistema.

Un'altra importante questione di cui tenere conto quando si scelgono queste fonti è la possibilità che vengano manipolate dall'esterno. Ad esempio, il traffico di rete potrebbe sembrare una buona fonte di casualità, ma è altamente soggetta a manipolazione e potrebbe dunque non fornire l'entropia necessaria.

5.2 Mixing e de-skewing

Il processo con cui le varie fonti esterne vengono combinate nell'entropy pool è chiamato **mixing**. In generale, le varie fonti non forniranno dati con entropia massima, ma l'entropia disponibile sarà distribuita su un volume di dati più ampio, e ciò si riflette sul fatto che la distribuzione di probabilità dei bit non sarà uniforme. Il processo di **de-skewing** ha lo scopo di estrarre da questi dati una sequenza di bit con distribuzione uniforme, che quindi avranno entropia massima. La quantità di bit che è possibile estrarre corrisponderà all'entropia dei dati originari.

Con ragionamenti di tipo probabilistico è possibile ottenere dati con una distribuzione arbitrariamente vicina all'uniformità a partire da dati qualsiasi, ma non è detto che si riesca in questo modo ad estrarre tutta l'entropia disponibile. Delle tecniche generali ed efficaci si appoggiano ad algoritmi crittografici come cifrari a blocchi o funzioni di hash. Il funzionamento di entrambi, infatti, si basa sull'idea di distribuire e concentrare l'entropia dell'input in modo uniforme su tutto l'output. Un cifrario a blocchi AES, ad esempio, prende in input 128bit di dati e 256 bit di chiave, e restituisce altri 128 bit in cui viene distribuita uniformemente l'entropia presente nell'input. Una funzione di hash crittografico prende in input una quantità variabile di dati e fornisce in output un blocco di lunghezza fissa in cui viene concentrata l'entropia dell'input.

Se dunque una certa quantità di dati con un entropia di n bit viene data in input ad un hash crittografico, l'output conterrà ancora n bit di entropia ma distribuiti uniformemente su tutto il blocco, e quindi n bit qualsiasi estratti dall'hash avranno entropia massima e quindi distribuzione uniforme.

Questo discorso si può estendere a più sorgenti per la fase di mixing. Si può calcolare l'hash crittografico della concatenazione dei dati di due sorgenti,

[13] Una novità in questo senso è stata introdotta nei microprocessori Intel® basati su tecnologia Haswell®, che includono un generatore hardware di entropia accessibile da una nuova istruzione della CPU

che concentrerebbe quindi l'entropia di entrambe. Altri schemi sono possibili, basati su crifrari a blocchi applicati in varie modalità [8].

5.3 PRNG crittograficamente forti

Una volta ottenuto un seme idoneo con le tecniche esposte finora, è possibile generare una sequenza di numeri pseudo-casuali veramente imprevedibile. È necessario però che questa sequenza non riveli nulla sullo stato interno del generatore, ovvero che vengano rispettati i criteri esposti all'inizio della sezione.

Una tecnica efficace prevede, ancora una volta, di affidarsi ad algoritmi di cifratura esistenti. In particolare gli **stream cipher** possono essere visti, da un certo punto di vista, proprio come dei generatori di numeri pseudo-casuali, il cui output viene applicato in XOR ai dati da cifrare. Usando quindi il seme come chiave e applicando uno stream cipher, come ad esempio **RC4**, ad una semplice sequenza crescente di interi, è possibile ottenere una sequenza pseudo-casuale crittograficamente forte. Lo stesso effetto è ottenibile con dei cifrari a blocchi, come ad esempio AES, applicati in **counter mode**, o in **output feedback mode**, modalità che di fatto trasformano un cifrario a blocchi in uno stream cipher (vedere [26] per i dettagli).

Esistono, comunque, generatori crittograficamente forti che non si basano su schemi di cifratura. L'idea, come già accennato, è quella di fare in modo che la relazione che lega due valori consecutivi della sequenza sia sufficientemente complessa.

5.3.1 Generatore Blum Blum Shub

Il generatore **BBS**, nome che deriva dai suoi autori, è stato introdotto nel 1986. In questo paragrafo si riassumeranno le sue caratteristiche principali, mentre per i dettagli si rimanda all'articolo originale [4].

La formulazione di questo generatore è veramente molto semplice. Partendo dal seme x_0, la sequenza di bit z_i viene prodotta nel seguente modo:

$$\begin{aligned} x_{n+1} &= x_n^2 \pmod m \\ z_i &= x_i \pmod 2 \end{aligned}$$

dove $m = pq$ è il prodotto di due grandi numeri primi. Il modo in cui z_i viene ottenuto da x_i corrisponde in pratica all'estrazione del bit meno significativo. In realtà, si può dimostrare che la sicurezza del generatore rimane invariata estraendo al massimo i $\log_2(\log_2(x_i))$ bit meno significativi di ogni x_i.

Questa ricorrenza ricorda molto da vicino i generatori congruenziali descritti nella sezione 3, ma la differenza sostanziale è che qui la relazione che lega i valori della sequenza non è lineare, e inoltre i valori prodotti in output rappresentano solo una parte dello stato interno del generatore. Osservando l'equazione si vede subito che il valore del seme deve essere coprimo con m, ovvero non deve avere p e q tra i suoi fattori, e ovviamente non deve essere pari a 0 o 1, per evitare che la sequenza degeneri.

Per garantire il periodo massimo, è necessario inoltre che i numeri p e q siano congruenti a $3 \pmod 4$, assunzione fondamentale anche per provarne la sicurezza.

Questo generatore basa la sua sicurezza sull'assunzione di intrattabilità del problema di scomposizione in fattori primi. Viene dimostrato, infatti, che distinguere la sequenza prodotta da una sequenza casuale è difficile quanto la fattorizzazione di m. Stesso discorso per la ricostruzione dei valori precedenti della sequenza a partire da un dato stato x_i. Supponendo quindi l'intrattabilità del problema della fattorizzazione, il BBS soddisfa i criteri illustrati all'inizio della sezione.

È chiaro che il principale svantaggio di questo algoritmo sta nella bassa efficienza implementativa, dovuta alle grandi dimensioni del modulo m. Tuttavia, visto i contesti per i quali è pensato, questo problema può risultare di secondaria importanza.

Conclusioni

La gamma di situazioni in cui è utile, se non fondamentale, avere a disposizione un buon generatore di numeri casuali, è vastissima. Nonostante ciò, molto spesso questa componente viene assunta come una **black box**, implementata chissà come dal produttore del sistema o del linguaggio di programmazione di turno.

A volte è invece fondamentale essere consapevoli delle caratteristiche e dei limiti del generatore che si sta usando. Un esempio lampante viene da una grande quantità di conclusioni scientifiche ottenute negli anni settanta e ottanta, messe poi parzialmente in discussione e talvolta anche smentite quando ci si rese conto delle pessime caratteristiche del già citato generatore `RANDU` implementato dai sistemi IBM su cui quei risultati erano stati ottenuti.

Questo episodio insegna che è importante anteporre a qualsiasi ottimizzazione di carattere implementativo un'analisi dettagliata e attenta per evitare di ottenere effetti indesiderati anche gravi, che nel caso dei PRNG si possono tradurre in risultati scorretti nella migliore delle ipotesi.

Il materiale su cui documentarsi non manca, tuttavia l'argomento non è usualmente incluso nei piani di studio e il materiale a disposizione nella forma di libri di testo scarseggia, ad eccezione di Knuth [9], che però non affronta gli ultimi sviluppi.

I temi riassunti in questo articolo possono dunque servire da punto di partenza più organico per chi volesse interessarsi all'argomento, guidando anche la scelta della fonte di casualità più adatta a seconda delle situazioni.

Riferimenti bibliografici

[1] `http://www.rsa.com/rsalabs/node.asp?id=2152`, visitato il 13/07/2013.

[2] `http://www.stat.fsu.edu/pub/diehard/`, visitato il 10/06/2013.

[3] D. Bini, M. Capovani, and O. Menchi. *Metodi numerici per l'algebra lineare*. Collana di matematica. Testi e manuali. Zanichelli, 1988.

[4] L Blum, M Blum, and M Shub. A simple unpredictable pseudo random number generator. *SIAM J. Comput.*, 15(2):364–383, May 1986.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[6] D.N. Dikranjan and M.S. Lucido. *Aritmetica e algebra*. Liguori Editore, 2007.

[7] F. Fabris. *Teoria dell'informazione, codici, cifrari*. Nuova didattica. Bollati Boringhieri, 2001.

[8] Internet Engineering Task Force. `rfc4086` - randomness requirements for security.

[9] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[10] L. Kuipers and H. Niederreiter. *Uniform Distribution of Sequences*. Dover books on mathematics. Dover Publications, Incorporated, 2006.

[11] Derrick H. Lehmer. Mathematical methods in large-scale computing units. *Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery, Annals of the Computation Laboratory of Harvard University*, 26:141, 1951.

[12] T. G. Lewis and W. H. Payne. Generalized feedback shift register pseudorandom number algorithm. *J. ACM*, 20(3):456–468, July 1973.

[13] Ming Li and Paul M.B. Vitnyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Publishing Company, Incorporated, 3 edition, 2008.

[14] George Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences*, 61(1):25–28, September 1968.

[15] Per Martin-Löf. The definition of random sequences. *Information and Control*, 9(6):602 – 619, 1966.

[16] Makoto Matsumoto and Yoshiharu Kurita. Twisted gfsr generators. *ACM Trans. Model. Comput. Simul.*, 2(3):179–194, July 1992.

[17] Makoto Matsumoto and Yoshiharu Kurita. Twisted gfsr generators ii. *ACM Trans. Model. Comput. Simul.*, 4(3):254–266, July 1994.

[18] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998.

[19] Makoto Matsumoto, Takuji Nishimura, Mariko Hagita, and Mutsuo Saito. Cryptographic mersenne twister and fubuki stream/block cipher. 2005. `http://eprint.iacr.org/`.

[20] Ueli M. Maurer. A universal statistical test for random bit generators. *J. Cryptol.*, 5(2):89–105, March 1992.

[21] François Panneton, Pierre L'Ecuyer, and Makoto Matsumoto. Improved long-period generators based on linear recurrences modulo 2. *ACM Trans. Math. Softw.*, 32(1):1–16, March 2006.

[22] Christos M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[23] S. K. Park and K. W. Miller. Random number generators: good ones are hard to find. *Commun. ACM*, 31(10):1192–1201, October 1988.

[24] K. Pearson. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *Philosophical Magazine Series 5*, 50(302), 1900.

[25] Mutsuo Saito and Makoto Matsumoto. Simd-oriented fast mersenne twister: a 128-bit pseudorandom number generator. In Alexander Keller, Stefan Heinrich, and Harald Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 607–622. Springer Berlin Heidelberg, 2008.

[26] Bruce Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[27] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, July, October 1948.

[28] John von Neumann. Various techniques used in connection with random digits. *J. Res. Nat. Bur. Stand.*, 12:36–38, 1951.

[29] Andrew C. Yao. Theory and application of trapdoor functions. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 80–91, Washington, DC, USA, 1982. IEEE Computer Society.

[30] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theor.*, 24(5):530–536, September 1978.