0.1 Stream

Il prossimo algoritmo che vedremo è quello degli **Stream**. Prima di parlarne, però, dobbiamo capire cosa sono i **Pseudo-Random Number Generator**, perché sono la base degli stream, anche se inizialmente possono sembrare non avere nulla a che fare con essi.

0.1.1 PRNG

Per chi ha già creato qualche programma in **C/C++**, può darsi che abbia dovuto **generare dei numeri casuali**. Tuttavia, è difficile crearli perché i computer sono sistemi **deterministici**, cioè ad ogni input corrisponde sempre lo stesso output. Quindi, per i computer è impossibile generare numeri realmente casuali; per questo motivo sono stati inventati gli **Pseudo-Random Number Generator**, ovvero dei generatori di numeri **pseudo-casuali**, che a noi **sembrano** casuali, ma che in realtà sono deterministici.

Un primo esempio è il **Linear Congruential Generator** (LCG), che parte da alcuni valori iniziali X_0, a, c e m (X_0 è detto **seme** o **seed**), i quali possono essere scelti a piacere. A partire da questi valori, possiamo generare nuovi numeri pseudo-casuali con la seguente formula:

$$X_{n+1} = aX_n + c \pmod{m}$$

Per chi non lo sapesse, il simbolo $\pmod{n}$ indica il **resto della divisione** tra due numeri. Per esempio, $9 \pmod 4$ significa che devo fare la divisione $9/4$ e considerare solo il resto. In questo caso, $9/4 = 2$ con resto 1, quindi $9 \pmod 4 = 1$. Riprenderemo questi concetti nei capitoli dedicati a **RSA** e **Diffie-Hellman**.

Per esempio, se scegliamo come valori iniziali $m = 9$, $a = 4$, $c = 1$ e come seme $X_0 = 3$, otteniamo la seguente sequenza di numeri pseudo-casuali:

$$\begin{aligned} X_1 &= 4 \cdot 3 + 1 \pmod 9 \\ &= 13 \pmod 9 \\ X_1 &= 4 \end{aligned}$$

$$\begin{aligned} X_2 &= 4 \cdot X_1 + 1 \pmod 9 \\ &= 4 \cdot 4 + 1 \pmod 9 \\ &= 17 \pmod 9 \\ X_2 &= 8 \end{aligned}$$

$$\begin{aligned} X_3 &= 4 \cdot 8 + 1 \pmod 9 \\ &= 33 \pmod 9 \\ X_3 &= 6 \end{aligned}$$

Per questa configurazione, i primi valori generati dalla sequenza sono:

$$4, 8, 6, 7, 2, 1, 5, 3, 4...$$

Si nota che, all'apparenza, sembra una sequenza casuale di numeri, ma in realtà deriva da una formula e da alcune variabili fissate inizialmente (a, m, c e X_0).

Un problema di questi generatori è che sono **ciclici**, nel senso che, prima o poi, la sequenza si ripete. Per esempio, se continuiamo la sequenza di prima:

$$\underline{4, 8, 6}, 7, 2, 0, 1, 5, 3, \underline{4, 8, 6}, 7, 2, 0, 1, 5, 3, \underline{4, 8}$$

Notiamo che questa sequenza è lunga 8 numeri, perché poi si ripete. Questo problema è dovuto al fatto che abbiamo scelto numeri **piccoli**; se invece scegliamo numeri più grandi, ad esempio a 10 o 15 cifre, la ciclicità sarà molto più ampia.

Questo sistema di generazione, però, ha un altro problema: dati alcuni output, è possibile ricavare i parametri (a, m e c). Infatti, se prendiamo quattro numeri **successivi**, che chiamiamo X_α, $X_{\alpha+1}$, $X_{\alpha+2}$ e $X_{\alpha+3}$, possiamo metterli a sistema.

$$\begin{cases} X_{\alpha+1} = aX_\alpha + c \pmod m \\ X_{\alpha+2} = aX_{\alpha+1} + c \pmod m \\ X_{\alpha+3} = aX_{\alpha+2} + c \pmod m \end{cases}$$

In questo caso, notiamo che abbiamo un sistema di 3 equazioni con 3 incognite, quindi possiamo risolverlo e calcolare a, m e c. In realtà, il fatto che ci sia il modulo rende tutto più complicato, ma comunque è possibile risolverlo.

Se vi state chiedendo perché è un enorme problema poter dedurre i parametri, la risposta è la seguente: per quanto vedremo tra poco con gli **stream**, è obbligatorio che la sequenza sia imprevedibile, perché quei numeri saranno le **chiavi private** per gli stream. Pertanto, è necessario che sia impossibile generare le chiavi successive (ovvero i numeri successivi nella sequenza).

Per giocare un po' con il **LCG**, vi fornisco due link: uno per generare la sequenza di numeri e un altro link che, dati 4 output, calcola a, m e c.

link sequenza: https://github.com/AlexBro98LoVero/Dispense/blob/main/Giochi/4_1_generaSequenza.py

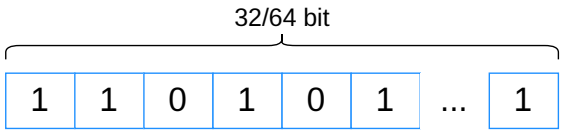
link 4 output: https://github.com/AlexBro98LoVero/Dispense/blob/main/Giochi/4_2_trovareParametri.py

N.B. Può succedere che si trovino più parametri validi; questo è dovuto al fatto che si tratta di equazioni modulari. Chiaramente, per scoprire quali sono i parametri corretti, bisognerebbe avere altri output e verificare per quali parametri funzionano.

Il **LCG** era utilizzato fino a **Java 17** (anno di pubblicazione 2020) come metodo per generare numeri nella libreria standard.

0.1.2 LFSR

Un'altra tecnica per generare numeri casuali è quella del **Linear Feedback Shift Register** (*LFSR*). Questo metodo parte definendo un **registro**, ovvero un **array** o **buffer** di bit, inizializzati a un **valore iniziale**, che fungerà da **chiave privata** del nostro sistema. La lunghezza del registro è arbitraria, ma di solito è di **32 bit** oppure **64 bit**.



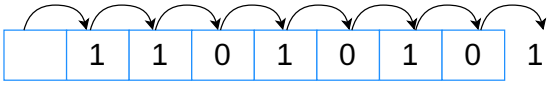
Poi si scelgono dei **tap**, ovvero delle posizioni del buffer. Ad esempio, supponiamo di scegliere le posizioni **2**, **3** e **5**. Il numero di tap può variare. Queste posizioni saranno usate perché i bit in quelle posizioni verranno **xorati** tra loro; il risultato andrà in testa al registro e tutti i bit verranno spostati verso destra (e l'**ultimo** bit andrà perso).

Per esempio, supponiamo di avere un registro a 8 bit con il seguente stato iniziale:

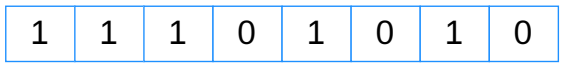
Supponiamo che i **tap** siano: **2**, **3** e **5**. Allora, il numero successivo sarà dato dallo **xor** di questi tre bit.

$$\text{nuovo bit} = \text{bit posizione 2} \oplus \text{bit posizione 3} \oplus \text{bit posizione 5}$$
$$\text{nuovo bit} = 1 \oplus 0 \oplus 0$$
$$\text{nuovo bit} = 1$$

Ora, spostiamo tutti i bit di una posizione verso destra nel registro.



Il bit che "sporge" a destra può essere escluso, e in prima posizione inseriamo il bit che abbiamo calcolato in precedenza.



E questo sarà il nuovo numero casuale, se ne volessi altri basta rieseguire il procedimento su questo nuovo stato del registro.

Una cosa interessante è notare che i **tap** possono essere espressi come **polinomi**. Rimanendo con l'esempio di prima dei tap (**2**, **3**, **5**), possiamo esprimere questa configurazione tramite il seguente polinomio:

$$p(x) = x^2 + x^3 + x^5 + 1$$

Qualsiasi configurazione è esprimibile tramite un polinomio, dove i gradi della variabile x corrispondono ai tap, con l'aggiunta di un termine noto (+1), e tutti i monomi hanno coefficiente 1.

Questo polinomio ha permesso di studiare le configurazioni ottimali per ottenere una generazione di numeri migliore. Infatti, grazie alla rappresentazione polinomiale, si è scoperto che se il polinomio è **primitivo modulo 2**, la generazione di numeri pseudo-casuali sarà massima. Non entrerò nei dettagli, ma considerate che un polinomio primitivo modulo 2 è simile a un **numero primo**, nel senso che non esistono prodotti di polinomi modulo 2 che diano come risultato quel polinomio primitivo.

Per esempio, il polinomio $p(x) = x^2+x+1$ è primitivo, perché non può essere scritto come prodotto di altri polinomi. Invece, il polinomio $p(x) = x^2 + 1$ non è primitivo, perché, in modulo 2, può essere scritto come:

$$\begin{aligned} p(x) &= (x+1)^2 \pmod 2 \\ &= x^2 + 2x + 1 \pmod 2 \\ &= x^2 + 1 \pmod 2 \end{aligned}$$

Inoltre, sempre con questo meccanismo, la generazione sarà massima anche quando le posizioni sono **numeri primi tra loro** e quando il **numero di posti è pari**.

Se non avete compreso bene questa parte, non preoccupatevi: il bello di questo concetto è vedere come una rappresentazione polinomiale di un sistema possa aiutare a migliorarlo e a massimizzarne l'efficienza.

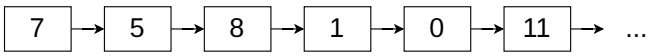
Grazie a questi accorgimenti, si è scoperto che il polinomio più sicuro per un LFSR a 16 bit è:

$$p(x) = x^{11} + x^{13} + x^{14} + x^{16} + 1$$

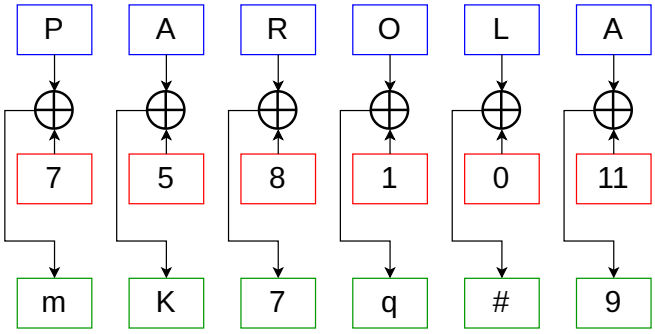
0.1.3 Stream

Fatta la premessa sui **PRNG**, capiamo cosa sono gli stream. Gli stream cercano di replicare la forza della cifratura **OTP**, che, come abbiamo già visto, è l'unico sistema teoricamente impossibile da decifrare. In pratica, gli stream utilizzano un algoritmo **PRNG**, come **LCG** o **LFSR** che abbiamo visto prima, e ogni lettera del messaggio viene cifrata tramite una **cifratura OTP** (cioè mediante un'operazione **XOR**) con uno dei numeri generati dal PRNG.

Quindi, per cifrare con gli stream, si inizia generando una sequenza di numeri:



Successivamente, ogni lettera del messaggio da cifrare viene combinata tramite un'operazione **XOR** con uno dei numeri generati.



N.B. Ricordo che il simbolo $\oplus$ indica l'operazione di **XOR**.

Cosi la stringa "**parola**" è diventata "**mK7q#9**".

Per decifrarlo, sarà sufficiente rigenerare i numeri usando gli stessi parametri e lo stesso seed della cifratura; successivamente, basta eseguire nuovamente l'operazione **XOR** tra il messaggio cifrato e la chiave, proprio come avviene nella cifratura **OTP**.

La **chiave privata** di questo sistema è quindi il **seed** e i **parametri** del PRNG, perchè avendoli si può generare la sequenza di numeri e poi cifrare il messaggio. Anche se in realtà si possono rendere pubblici i parametri e lasciare **solamente il seed come chiave privata**, tanto con valori di paramentri molto altri (10/15 cifre) non rendono facilmente la rottura del cifrario

L'idea alla base degli stream è di replicare la sicurezza dell'**OTP** e di usare chiavi diverse per ogni cifratura, poiché ricordiamo che il problema dell'OTP è che non si può riutilizzare la stessa chiave, altrimenti si può risalire ai messaggi originali.

Ad oggi, però, i cifrari a **stream** non sono molto diffusi, a causa della difficoltà di generare numeri veramente casuali e perché, se si scopre il **seed** (che costituisce la chiave privata del cifrario), è possibile ricavare l'intero messaggio.