**Università degli Studi di Torino**

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea in Matematica


TESI DI LAUREA SPECIALISTICA

# *Applicazioni del Metodo Rho di Pollard al Problema del Logaritmo Discreto e al Problema della Fattorizzazione*

Relatore: Umberto Cerruti

Candidato: Emanuele Bellini
N. matricola: 333442

Anno Accademico 2008-2009

*"L'evoluzione è la legge della vita.*
*Il numero è la legge dell'universo.*
*L'Unità è la legge di Dio"*
Pitagora

# INDICE

# INTRODUZIONE

*The White Rabbit put on his spectacles.*
*"Where shall I begin, please your Majesty ?" he asked.*
*"Begin at the beginning,", the King said, very gravely,*
*"and go on till you come to the end: then stop."*
Lewis Carroll

Il problema della fattorizzazione di un numero è considerato un problema "difficile", nel senso che anche con i macchinari ed i metodi più evoluti, i numeri oltre una certa dimensione richiedono anni, se non l'età dell'universo, per essere fattorizzati. Al problema della fattorizzazione è strettamente collegato il problema del logaritmo discreto. I due problemi infatti, sembrano essere della stessa difficoltà, e negli ultimi decenni, matematici ed esperti di sicurezza informatica hanno sfruttato questa difficoltà per creare sistemi crittografici a chiave pubblica sicuri. Inoltre, spesso metodi che risolvono un problema si rivelano utili per risolvere il secondo e viceversa. L'analisi di uno di questi metodi, il cosiddetto Metodo Rho di Pollard, sarà l'argomento di questa tesi.

Il metodo fu presentato per la prima volta, nel 1975, dal professor John Pollard [7] come metodo per fattorizzare un numero *n*, e tre anni più tardi, dallo stesso autore, uscì l'articolo che presentava le stesse idee applicate al logaritmo discreto [31]. Per quanto riguarda la fattorizzazione, il metodo rimase all'avanguardia per alcuni anni come uno dei migliori algoritmi esponenziali in grado di scomporre un numero composto da due primi di circa le stesse dimensioni (caso utile in crittografia), ma presto venne superato da metodi ed idee innovative basati sulle curve ellittiche e sui campi di numeri che sono in grado di scomporre numeri dello stesso tipo in tempi più brevi. Per quanto riguarda il logaritmo discreto, invece, il Metodo Rho è tuttora uno dei migliori in circolazione, il migliore conosciuto se si lavora in gruppi i cui elementi sono punti di curve ellittiche, come avviene in molti sistemi di sicurezza informatica.

Da quando questi due problemi sono entrati a far parte della vita di tutti i giorni (grazie al loro utilizzo su internet, nelle banche o nei cellulari…) sono stati raggiunti risultati pressoché sorprendenti e soprattutto inaspettati da molti matematici. La

causa di ciò è stata la sviluppo tecnologico dei processori utilizzati per eseguire i calcoli e soprattutto la possibilità di poter utilizzare contemporaneamente migliaia di macchine per risolvere lo stesso problema.

Il fascino del metodo di Pollard (come quello di tanti altri algoritmi in questo campo) sta nel fatto che molte teorie e molti teoremi si intersecano attorno ad esso, dal paradosso dei compleanni, ai cammini pseudorandom, dal piccolo teorema di Fermat ai metodi per la ricerca della ciclicità. Sarà interessante analizzare il metodo da un punto di vista algebrico e da uno statistico. Inoltre si vedrà come a volte in matematica non sia possibile un analisi dettagliata e formale dei risultati, ma ci si debba accontentare di conferme pratiche ed euristiche, dalle quali si vede che il metodo "funziona", anche se non lo si è formalmente dimostrato.

Ecco dunque come sarà suddiviso il lavoro.

Un primo capitolo servirà per spiegare la stretta correlazione tra i due problemi, che sono "quasi" riconducibili l'uno all'altro. Inoltre verrà esposta una dimostrazione da cui estrarremo un algoritmo per risolvere il problema della fattorizzazione riconducendoci al problema del logaritmo discreto.

Nel secondo capitolo verrà presentato il metodo Rho di Pollard per risolvere il problema del logaritmo discreto, prima analizzando il caso in cui si lavora con un singolo computer e poi il modo più conveniente per parallelizzare il metodo.

Nel capitolo tre vedremo come lo stesso metodo si possa applicare alla fattorizzazione, e come il problema della parallelizzazione del metodo si possa risolvere pur presentando alcuni problemi che tutt'oggi restano aperti.

Nel capitolo quattro, come accennato, implementeremo un algoritmo che risolve il problema della fattorizzazione riconducendola al problema del logaritmo discreto, il quale si rivelerà sconveniente da risolvere con il metodo Rho se si devono fattorizzare numeri di piccole dimensioni.

Infine nell'appendice sono riportati gli algoritmi java utilizzati per testare al computer i metodi studiati. In questi algoritmi viene utilizzata la classe BigInteger, grazie alla quale è possibile lavorare con numeri di qualsiasi dimensione (compatibilmente con le capacità del computer).

# Capitolo 1

## RELAZIONE TRA I PROBLEMI DEL LOGARITMO DISCRETO E DELLA FATTORIZZAZIONE

*"God created the natural numbers,*
*all the rest is the work of man"*
Leopold Kronecker

Il problema del Logaritmo Discreto (abbreviato con DLP) è il seguente:

Dati due interi *g* ed *h* relativamente primi ad un altro intero *n*, trovare il numero *x* tale che:

$$g^x \equiv h \bmod n$$

Il problema della Fattorizzazione è il seguente:

Dato un numero intero *n* trovare la sua scomposizione in fattori primi (oppure cercare un divisore primo $p_i$ di *n*)

$$n = p_1^{e_1} \cdot \ldots \cdot p_k^{e_k}$$

Ricordiamo che risolvere un problema in tempo polinomiale significa che il numero medio di operazioni elementari per risolvere il problema nel caso peggiore è limitato da un polinomio nel numero di cifre necessarie per rappresentare l'input.

I due problemi sono così correlati:

1) Se sappiamo risolvere $g^x \equiv h \bmod N$ in tempo polinomiale, allora con alta probabilità possiamo trovare una fattore proprio di *N* in tempo polinomiale. Questa relazione può essere resa deterministica se vale l'Ipotesi di Riemann Estesa.

2) Se una soluzione di $g^x \equiv h \bmod p^e$ (con *p* primo ed *e* intero positivo) esiste, allora posso trovarla in tempo polinomiale da una soluzione di $g^x \equiv h \bmod p$.

**3)** Se siamo in grado di fattorizzare in tempo polinomiale, allora per risolvere in tempo polinomiale $g^x \equiv h \bmod N$ tutto ciò che ci serve è trovare soluzioni modulo i divisori primi di *N*.

## 1.1 Teorema di Miller

Diamo la dimostrazione del primo fatto.

**Teorema (Miller).** Se esiste un algoritmo A di tempo polinomiale per risolvere il Problema del Logaritmo Discreto, allora esiste un algoritmo di tempo polinomiale per risolvere il Problema della Fattorizzazione con una probabilità arbitrariamente alta.

**Dimostrazione.** Assumiamo che *N* sia dispari e non una potenza di un primo. Queste condizioni possono essere testate in maniera efficiente, e se non dovessero valere, condurrebbero immediatamente ad un fattore di *N*.

Consideriamo la struttura di $\left(\frac{\mathbb{Z}}{N\mathbb{Z}}\right)^*$. Se *N* ha fattorizzazione $p_1^{e_1} \cdot \ldots \cdot p_s^{e_s}$ allora

$$\left(\frac{\mathbb{Z}}{N\mathbb{Z}}\right)^* = \left(\frac{\mathbb{Z}}{p_1^{e_1}\mathbb{Z}}\right)^* \times \ldots \times \left(\frac{\mathbb{Z}}{p_s^{e_s}\mathbb{Z}}\right)^*$$

Questo gruppo ha ordine $\phi(N) = \phi_1 \ldots \phi_s$, dove ogni $\phi_i = \phi\left(p_i^{e_i}\right) = p_1^{e_i-1}(p_i - 1)$, ma non è un gruppo ciclico e non c'è nessun elemento di ordine $\phi(N)$. Tuttavia, se eleviamo qualsiasi $a \in \left(\frac{\mathbb{Z}}{N\mathbb{Z}}\right)^*$ ad una potenza *e* tale che $\phi_i | e$, segue che $a^e \equiv 1 \bmod p_i^{e_i}$ per ogni *i*, e quindi $a^e \equiv 1 \bmod N$. Quindi ogni elemento di $\left(\frac{\mathbb{Z}}{N\mathbb{Z}}\right)^*$ avrà ordine che divide $L = \text{mcm}(\phi_1, \ldots, \phi_s)$.

Dimostriamo il nostro risultato in due passi. Prima dimostriamo che data la capacità di trovare l'ordine degli elementi di $\left(\frac{\mathbb{Z}}{N\mathbb{Z}}\right)^*$, possiamo fattorizzare *N*. Poi dimostreremo come calcolare gli ordini a partire dai logaritmi modulo *N*.

(1)

Il nostro primo scopo è dunque quello di fattorizzare *N* determinando l'ordine di un elemento. Abbiamo già stabilito che $a^L \equiv 1 \bmod N$ per tutti gli *x*, ma come prima menzionato, poiché *N* è composto, sappiamo che ci sono radici quadrate di 1 diverse da $\pm 1$. Dunque consideriamo il sottogruppo *K* di $\left(\frac{\mathbb{Z}}{N\mathbb{Z}}\right)^*$:

$$K = \left\{ x \in \left(\frac{\mathbb{Z}}{N\mathbb{Z}}\right)^* : a^{\frac{L}{2}} \equiv \pm 1 \bmod N \right\}$$

Possiamo dimostrare che $K \neq \left(\frac{\mathbb{Z}}{N\mathbb{Z}}\right)^*$ nel modo seguente. Prima di tutto definiamo il simbolo $e_2(k)$ per $k \in \mathbb{Z}$ come l'esponente di 2 nella fattorizzazione prima di *k*, o equivalentemente la maggiore potenza di 2 che divide *k*. Ora ordiniamo i $p_i$ usando questo simbolo, in modo che $e_2(\phi_1) \geq e_2(\phi_i)$ per tutti gli *i*. Segue che $e_2(\phi_1) = e_2(L)$. Poiché ogni gruppo $\left(\frac{\mathbb{Z}}{p_i^{e_i}\mathbb{Z}}\right)^*$ è ciclico, scegliamo i generatori $g_1, \dots, g_s$. Poi siano $(g_1, g_2^2 \dots, g_s^2)$ le coordinate di $a \in \left(\frac{\mathbb{Z}}{N\mathbb{Z}}\right)^*$. Segue che:

$$a^{\frac{L}{2}} = (g_1^{\frac{L}{2}}, g_2^L \dots, g_s^L) = (g_1^{\frac{L}{2}}, 1 \dots, 1)$$

non vale $\pm 1$, poiché ha ordine $\phi_1$, di cui $L/2$ non può essere un multiplo poiché $e_2(L/2) = e_2(\phi_1) - 1$. E quindi $a \notin K$. Poiché *K* è quindi un sottogruppo proprio, segue che almeno metà degli elementi di $\left(\frac{\mathbb{Z}}{N\mathbb{Z}}\right)^*$ stanno fuori da *K*.

Ora supponiamo che $x \in \left(\frac{\mathbb{Z}}{N\mathbb{Z}}\right)^*$ ma che $x \notin K$ e che abbiamo qualche *m* tale che $x^m \equiv 1$. Consideriamo la sequenza $x^m, x^{\frac{m}{2}}, x^{\frac{m}{4}}, \dots, x^{\frac{m}{2^{e_2(m)}}}$. Affermiamo che per qualche *k*, dovrà valere $x^{\frac{m}{2^k}} \equiv 1$, ma $x^{\frac{m}{2^{k+1}}} \not\equiv \pm 1$ . Supponiamo che l'ordine reale di *x* in $\left(\frac{\mathbb{Z}}{N\mathbb{Z}}\right)^*$ sia *r*. Segue che $L = c_0 \cdot r$ per un intero $c_0$. Inoltre, poiché $x \notin K$, abbiamo che $x^{\frac{L}{2}} \not\equiv 1$, e quindi $c_0$ deve essere dispari. Abbiamo anche che $m = c \cdot r$ e possiamo tirare fuori da *c* tutti i 2 per ottenere $m = 2^{e_2(c)} \cdot c_1 \cdot r$ per $c_1$ dispari. Dunque se poniamo $k = e_2(c)$, vediamo che $x^{\frac{m}{2^k}} \equiv x^{c_1 r} \equiv 1$. Consideriamo poi $x^{\frac{m}{2^{k+1}}}$. Osserviamo ora che per ogni *d* dispari, ogni volta che $z^2 \equiv 1 \bmod N$, allora dobbiamo avere che $z \equiv \pm 1 \bmod p_i^{e_i}$ per ogni *i*, e poiché entrambi $\pm 1$ sono la loro stessa *d*_esima potenza, allora $z^d \equiv z \bmod p_i^{e_i}$ e quindi $z^d \equiv z \bmod N$. Segue che $x^{\frac{L}{2}} \equiv x^{\frac{c_0 r}{2}} \equiv x^{\frac{r}{2}}$ (poiché $c_0$ è dispari e $x^{\frac{r}{2}}$ è una radice di 1). Allo stesso modo $x^{\frac{m}{2^{k+1}}} \equiv x^{\frac{c_1 r}{2}} \equiv x^{\frac{r}{2}}$. Dunque

$$x^{\frac{m}{2^{k+1}}} \equiv x^{\frac{L}{2}} \not\equiv \pm 1$$

E ciò prova l'affermazione.

Segue che $x^{\frac{m}{2^{k+1}}}$ è una radice non banale di 1 e dunque possiamo fattorizzare *N* prendendo MCD $\left(x^{\frac{m}{2^{k+1}}} \pm 1, N\right)$.

(2)

Ci rimane solo da dimostrare che un *esponente* (definiamo esponente di $x$ un elemento $y$ tale che $x^y \equiv 1$) di un elemento può essere trovato tramite il calcolo di un logaritmo discreto. Notiamo innanzitutto che non ci basta calcolare il logaritmo discreto di 1 alla base $x$, poiché $m = 0$ è sempre un esponente valido che però non ci conduce ad una sequenza utile nel metodo sopraesposto, poiché 0 non ha un numero ben definito di 2 che lo dividano.

Ma supponiamo che $p$ sia un primo tale che $\mathrm{MCD}\big(p, \phi(N)\big) = 1$. Se è così, allora esiste una soluzione intera $q$ alla congruenza $pq \equiv 1 \bmod \phi(N)$. Allora, per Fermat, $x^{pq} \equiv x \bmod N$, e quindi possiamo risolvere il logaritmo di $x$ in base $x^p$, ottenendo il risultato $q$ e poi prendendo $m = pq - 1$ come esponente di $x$.

Naturalmente noi non conosciamo $\phi(N)$ in anticipo, e quindi non possiamo testare questa condizione su $p$; però, algoritmicamente, possiamo cercare il $\log_{x^p} x$ per $p$ e vedere se il $q$ che otteniamo soddisfa $x^{pq} \equiv x$. Poiché $\phi(N) < N$, possiamo garantire che ci deve essere un primo $p$ relativamente primo a $\phi(N)$ entro i primi $\log N + 1$ primi. E quindi possiamo concludere in tempo polinomiale la ricerca di $p$ tale che $\log_{x^p} x$ esista.

Poiché sappiamo che $x \notin K$ con probabilità $\geq \frac{1}{2}$ per un $x$ scelto casualmente, segue che il nostro algoritmo, se ripetuto $l$ volte, fallirà nell'intento di fornirci un fattore con probabilità $\leq \frac{1}{2^l}$. Quindi abbiamo una probabilità arbitraria di successo.

□

Si può trovare una dimostrazione degli altri due fatti in [11].

Ci occuperemo in questa tesi di risolvere i due problemi tramite un metodo inventato da John Pollard, il cosiddetto Metodo Rho.

# Capitolo 2

## RISOLUZIONE DEL PROBLEMA DEL LOGARITMO DISCRETO CON IL METODO RHO DI POLLARD

*"Anyone who considers arithmetical methods*
*of producing random digits is,*
*of course, in a state of sin"*
John Von Neumann

## 2.1 Paradosso dei Compleanni

I metodi che useremo sono basati su alcuni risultati di teoria delle probabilità. Il primo strumento di cui faremo uso è il cosiddetto Paradosso dei Compleanni, così chiamato poiché da esso si ricava la risposta (per molti paradossale) che per trovare con probabilità del 50% due persone nate lo stesso giorno pescando a caso nella popolazione, ho bisogno di scegliere solamente 23 individui.

**Teorema.** Sia *S* un insieme di *N* elementi. Se scegliamo uniformemente a random gli elementi di *S*, allora il numero atteso di campioni estratti prima di trovare una ripetizione (detta anche match o collisione) è minore di

$$\sqrt{\pi N/2} + 2 \approx 1.253\sqrt{N} + 2$$

**Dim [**in [12] pag. 206**]**

La dimostrazione di [12] da solamente un limite superiore sulla probabilità di una collisione dopo $l$ tentativi. Un limite inferiore è dato in [13] ed è $e^{-\frac{l^2}{2N}-\frac{l^3}{6N^2}}$ per $N \geq 1000$ e $0 \leq l \leq 2N\,ln(N)$. È anche mostrato che il numero atteso di tentativi è $> \sqrt{\pi N/2} - 0.4$.

Ricordiamo il significato del valore atteso. Supponiamo che l'esperimento di selezionare elementi di un insieme *S* di dimensione *N* finché non viene trovata una collisione sia ripetuto *t* volte e che ogni volta contiamo il numero *l* di elementi

estratti. Allora la media di *l* di tutti gli esperimenti tende a $\sqrt{\pi N/2}$ per *t* che va all'infinito.

**Teorema[**esercizio 17.2 pag 206 di [12]**].** Il numero di elementi che dobbiamo selezionare da *S* per avere una ripetizione con probabilità 0.5 è

$$\sqrt{2\ln(2)N} \approx 1.177\sqrt{N}$$

**Dimostrazione.** La probabilità che il secondo elemento estratto sia diverso dal primo è data da

$$\left(\frac{N-1}{N}\right) = \left(1 - \frac{1}{N}\right)$$

La probabilità che il terzo estratto sia diverso dai primi due è

$$\left(\frac{N-2}{N}\right) = \left(1 - \frac{2}{N}\right)$$

Quindi, la probabilità che $q$ elementi estratti da *S*, insieme di *N* elementi, siano tutti diversi è

$$\left(1 - \frac{1}{N}\right)\left(1 - \frac{2}{N}\right)\left(1 - \frac{3}{N}\right)\ldots\left(1 - \frac{q-1}{N}\right)$$

Questa quantità vale circa

$$\approx \left(1 - \frac{q}{2N}\right)^{q-1} = \left(1 - \frac{1}{2N/q}\right)^{q-1} \approx e^{-\frac{q(q-1)}{2N}} = \frac{1}{e^{\frac{q(q-1)}{2N}}}$$

Questa probabilità sarà uguale a ½ se e solo se

$$\frac{q(q-1)}{2N} = \ln 2$$

$$q^2 - q - 2N\ln 2 = 0$$

Da cui

$$q = \frac{1 \pm \sqrt{1 - 8N\ln 2}}{2}$$

E di queste soluzioni escludiamo la soluzione negativa e per *N* grande abbiamo che

$$q \approx \sqrt{2N\ln 2} + \frac{1}{2} \approx 1.177\sqrt{N}$$

□

**Problema [**esercizio 17.3 pag 206 di [12]**].** Uno potrebbe essere interessato nel numero di tentativi richiesti quando si è particolarmente sfortunati. Determiniamo il numero di tentativi necessari per avere una collisione con probabilità 0.99. Fare lo stesso con 0.999.

Seguendo il ragionamento della dimostrazione precedente avremo che la probabilità che $q$ elementi estratti da $S$, insieme di $N$ elementi, siano tutti diversi, vale sempre

$$\frac{1}{e^{\frac{q(q-1)}{2N}}}$$

Questa quantità, questa volta, deve essere uguale a 0.01 (poiché se la probabilità di avere una collisione è 0.99, la probabilità di non averla è $1 - 0.99$), cioè

$$\frac{1}{e^{\frac{q(q-1)}{2N}}} = \frac{1}{100}$$

Questo vale se e solo se

$$\frac{q(q-1)}{2N} = \ln 100$$

Se e solo se

$$q \approx \sqrt{2N \ln 100} + \frac{1}{2} \approx 3.035\sqrt{N}$$

Applicando lo stesso ragionamento alla probabilità 0.001 di non avere collisioni otteniamo

$$q \approx \sqrt{2N \ln 1000} + \frac{1}{2} \approx 3.717\sqrt{N}$$

Cioè, in entrambi i casi, saranno richieste più del doppio delle estrazioni rispetto alla probabilità 0.5.

## 2.2 Metodo Rho di Pollard Seriale

Sia *G* un gruppo e sia *g* un suo elemento di ordine *r*. Consideriamo il problema del logaritmo discreto associato a questo gruppo, cioè dato *h* elemento anch'esso del gruppo trovare un *a* tale che $g^a = h$. Assumiamo anche (come succede spesso nelle applicazioni) che sia già stato verificato che $h \epsilon < g >$.

L'idea dell'algoritmo di Rho è quella di trovare quattro interi ($a_i$ , $b_i$) e ($a_j$ , $b_j$) tali che

$$g^{a_i} h^{b_i} = g^{a_j} h^{b_j}$$

Ciò viene fatto trovando un' autocollisione in un cammino pseudorandom. Una volta trovati questi interi il DLP si risolve come

$$h = g^{(a_i - a_j)(b_j - b_i)^{-1} \bmod r}$$

assumendo che $(b_j - b_i)$ sia invertibile modulo *r*.

Presentiamo ora una versione dell'algoritmo rho che fa uso del metodo di Floyd per trovare la ciclicità; presenteremo in seguito metodi più efficienti.

### 2.2.1 I cammini pseudorandom

Il metodo è motivato da proprietà statistiche dei cammini random negli insiemi. Sia $f : S \rightarrow S$ una funzione random da un insieme di dimensione *N* in se stesso. Per ogni $x_i \in S$ si può definire la sequenza $x_{i+1} = f(x_i)$ per $i$ = 2, 3, … . Chiamiamo questa sequenza cammino deterministico pseudorandom su *G* (deterministico poiché $x_{i+1}$ dipende solo da $x_i$ e non da *i* ).

Poiché *S* è finito prima o poi ci dovrà essere una collisione del tipo $f(x_i) = f(x_j)$ per qualche $1 \leq i < j$ e da qui il cammino diventerà ciclico. Per il paradosso dei compleanni il valore atteso di *j* sarà $\sqrt{\pi N/2}$. Ci si può immaginare il cammino avente una "coda" (parte non ciclica) seguita da un "ciclo" o "testa", ecco perché il nome della lettera rho, ρ. La coda e la testa di questo cammino random ha una lunghezza attesa $\sqrt{\pi N/8}$ ( si veda[14] per una dimostrazione).

Pollard simula una funzione random da *G* in se stesso nel modo seguente.

Il primo passo è scomporre *G* in $n_S$ sottoinsiemi disgiunti di circa uguale dimensione e tali che $G = S_0 \cup ... \cup S_{n_S - 1}$. I libri tradizionali presentano il problema con $n_S$ = 3, ma nella pratica è meglio adoperare i valori 20 o 32.

Gli insiemi $S_i$ sono definiti usando una funzione di selezione $S : G \rightarrow \{0, \dots, n_s - 1\}$ con $S_i = \{g \in G | S(g) = i\}$.

Per esempio, nelle implementazioni al computer di *G*, i suoi elementi sono rappresentati con un'unica stringa binaria *b*(*g*) ed interpretando *b*(*g*) come un intero si può definire $S(g) = b(g) \bmod n_S$ (scegliendo $n_S$ come potenza di 2 i calcoli vengono resi molto semplici). Per ottenere diverse scelte di *S* si potrebbe applicare inizialmente una mappa $\mathbb{F}_2$-lineare *L* a *b*(*g*), in modo che $S(g) = L(b(g)) \bmod n_S$ (in sostanza, calcolando una somma parziale sui bit di *b*(*g*) ).

**Definizione.** I *cammini rho* sono definiti come segue:

precalcoliamo $g_i = g^{u_j} h^{v_j}$ per $0 \leq j \leq n_S - 1$ dove $0 \leq u_j, v_j < r$ sono scelti uniformemente a random.

Poniamo $x_1 = g$ e definiamo i cammini come segue:

il *cammino rho originale* è

$$x_{i+1} = f(x_i) = \begin{cases} x_i^2 & \text{se } S(x_i) = 0 \\ x_i g_j & \text{se } S(x_i) = j, j \in \{1, \dots, n_S - 1\} \end{cases}$$

il *cammino rho additivo* è

$$x_{i+1} = f(x_i) = x_i g_{S(x_i)}$$

Un'importante caratteristica dei cammini è che ad ogni passo richiedono solamente un'operazione nel gruppo.

Anche se la funzione di selezione *S* e gli interi $u_j, v_j$ possono essere scelti uniformemente a random, la funzione *f* stessa non è random, ma è definita in modo compatto; questo è il motivo per cui i cammini rho sono definiti come pseudo casuali.

È necessario tenere traccia della decomposizione $x_i = g^{a_i} h^{b_i}$

I valori di $a_i$ e $b_i$ sono ottenuti ponendo $a_i = 1$ e $b_1 = 0$ e per il cammino rho originale aggiornandoli nel modo seguente:

$$a_{i+1} = \begin{cases} 2a_i \bmod r & \text{se } S(x_i) = 0 \\ a_i + u_{S(x_i)} \bmod r & \text{se } S(x_i) > 0 \end{cases}$$

$$b_{i+1} = \begin{cases} 2b_i \bmod r & \text{se } S(x_i) = 0 \\ b_i + v_{S(x_i)} \bmod r & \text{se } S(x_i) > 0 \end{cases}$$

Mentre per il cammino rho additivo [esercizio 17.6 pag 208 di [12] ] è semplicemente

$$a_{i+1} = a_i + u_{S(x_i)} \bmod r$$
$$b_{i+1} = b_i + v_{S(x_i)} \bmod r$$

**Proprietà**[Esercizio 17.7 pag 208 di [12] ]. Se $b(g) \leq c\log_2(r)$ allora ci sono al massimo $\lceil \log_2(n_S) \rceil r^c$ possibilità per *S*.

### 2.2.2 Trovare la collisione con il metodo di Floyd

Come accennato, il problema del logaritmo discreto viene risolto trovando una collisione $x_i = x_j$. In questa sezione descriviamo un metodo per trovare tale collisione. Potrebbe sembrare che l'unico approccio sia memorizzare tutte le $x_i$ e per ogni nuovo valore $x_j$, controllare che sia nella lista. Quest'approccio richiede un utilizzo eccessivo di memoria e un notevole dispendio di tempo. Se stessimo utilizzando una funzione realmente random allora quella appena citata sarebbe l'unica soluzione. Lo scopo di utilizzare un cammino deterministico che prima o poi diventa ciclico è proprio quello di permettere l'utilizzo di metodi migliori per trovare una collisione.

Sia $l_t$ la lunghezza della coda ed $l_h$ la lunghezza del ciclo, cioè la prima collisione è

$$x_{l_t+l_h} = x_{l_t}$$

Il metodo di Floyd è quello di confrontare $x_i$ con $x_{2i}$. Il seguente risultato mostra che il metodo troverà una collisione in al massimo $l_t + l_h$ passi. Il vantaggio cruciale è che in questo modo è sufficiente memorizzare solo due elementi del gruppo.

**Lemma.** Con le notazioni appena introdotte, allora $x_{2i} = x_i$ se e solo se $l_h \mid i$ e $i \geq l_t$. Inoltre, c'è qualche $l_t \leq i < l_t + l_h$ tale che $x_{2i} = x_i$.

**Dimostrazione.** Se $x_i = x_j$ allora abbiamo che $l_h \mid (i - j)$. Dunque la prima parte del lemma è dimostrata. La seconda segue immediatamente poiché esiste qualche multiplo di $l_h$ tra $l_t$ ed $l_t + l_h$ .

□

Il più piccolo indice $i$ tale che $x_{2i} = x_i$ è chiamato *epact*. In seguito daremo dei risultati euristici per il valore atteso dell'epact.

**Esempio.** Sia $p$ = 809 e consideriamo $g$ = 89 con ordine 101 in $\mathbb{F}_p^*$**.** Sia $h$ =799 appartenente al sottogruppo generato da $g$.

Sia $n_S = 4$. Per definire $S(g)$ scriviamo $g$ nel range $1 \leq g < 809$, rappresentiamolo nell'usuale espansione binaria e poi riduciamolo modulo 4. Scegliamo $(u_1, v_1)$ = (37, 34), $(u_2, v_2)$ = (71, 69), $(u_3, v_3)$ = (76, 18) in modo che $g_1 = 343, g_2 = 676,\ g_3 = 627$. Calcoliamo ora i valori ($x_i$ , $a_i$ , $b_i$ ) come segue:

| $i$ | $x_i$ (mod 809) | $a_i$ mod $r$ | $b_i$ mod $r$ | $S(x_i)$ |
|---|---|---|---|---|
| 1 | 89 | 1 | 0 | 1 |
| 2 | 594 | 38 | 34 | 2 |
| 3 | 280 | 8 | 2 | 0 |
| 4 | 736 | 16 | 4 | 0 |
| 5 | 475 | 32 | 8 | 3 |
| 6 | 113 | 7 | 26 | 1 |
| 7 | 736 | 44 | 60 | 0 |

Segue che $l_t$ = 4 ed $l_h$ = 3 e quindi la prima collisione intercettata con il metodo di Floyd è $x_6 = x_{12}$.

Verifichiamo che il logaritmo discreto vale 50 in questo caso, calcolando

$$(a_4 - a_7)(b_7 - b_4)^{-1} \bmod r = (16 - 44)(60 - 4)^{-1} \bmod 101 = 50$$

Dove 101 è l'ordine di 89 in $\mathbb{Z}_{809}$.

**Esempio.** Sia $p = 347$, $r = 173$, $g = 3$, $h = 11 \in \mathbb{F}_p^*$, $n_S = 3$. Determinare $l_t$ ed $l_h$ a partire dai valori $(u_1, v_1) = (1, 1)$, $(u_2, v_2) = (13, 17)$. Qual è il più piccolo valore di $i$ per cui $x_{2i} = x_i$ ?

Usando l' algoritmo A.2.2 nell'appendice otteniamo il seguente output, dal quale ricaviamo che $l_t = 4$ ed $l_h = 26$.

```
i  --- xi --- ai --- bi --- S(xi)
--------------------------------
1  --> 3   -- 1   -- 0   -- 0
2  --> 9   -- 2   -- 0   -- 0
3  --> 81  -- 4   -- 0   -- 0
4  --> 315 -- 8   -- 0   -- 0
5  --> 330 -- 16  -- 0   -- 0
6  --> 289 -- 32  -- 0   -- 1
7  --> 168 -- 33  -- 1   -- 0
8  --> 117 -- 66  -- 2   -- 0
9  --> 156 -- 132 -- 4   -- 0
10 --> 46  -- 91  -- 8   -- 1
11 --> 130 -- 92  -- 9   -- 1
12 --> 126 -- 93  -- 10  -- 0
13 --> 261 -- 13  -- 20  -- 0
14 --> 109 -- 26  -- 40  -- 1
15 --> 127 -- 27  -- 41  -- 1
16 --> 27  -- 28  -- 42  -- 0
17 --> 35  -- 56  -- 84  -- 2
18 --> 205 -- 69  -- 101 -- 1
19 --> 172 -- 70  -- 102 -- 1
20 --> 124 -- 71  -- 103 -- 1
21 --> 275 -- 72  -- 104 -- 2
22 --> 74  -- 85  -- 121 -- 2
23 --> 136 -- 98  -- 138 -- 1
24 --> 324 -- 99  -- 139 -- 0
25 --> 182 -- 25  -- 105 -- 2
26 --> 25  -- 38  -- 122 -- 1
27 --> 131 -- 39  -- 123 -- 2
28 --> 222 -- 52  -- 140 -- 0
29 --> 10  -- 104 -- 107 -- 1
30 --> 330 -- 105 -- 108 -- 0
```

Il minimo $i$ per cui $x_{2i} = x_i$ è $i = 25$ (non riportiamo l'intera stampata per questioni di spazio).

**Esempio.** Ripetiamo l'esercizio precedente con $g$ = 11, $h$ = 3, $(u_1, v_1)$ = (4, 7), $(u_2, v_2)$ = (23, 5).

Ecco l'output del programma:

```
i  --- xi --- ai --- bi -- S(xi)-- x2i -- a2i - b2i - S(x2i)
-----------------------------------------------------------
1  --> 11  -- 1   -- 0   -- 2 <--> 115 -- 24  -- 5   -- 1
2  --> 115 -- 24  -- 5   -- 1 <--> 117 -- 56  -- 24  -- 0
3  --> 168 -- 28  -- 12  -- 0 <--> 46  -- 51  -- 96  -- 1
4  --> 117 -- 56  -- 24  -- 0 <--> 324 -- 78  -- 108 -- 0
5  --> 156 -- 112 -- 48  -- 0 <--> 10  -- 6   -- 48  -- 1
6  --> 46  -- 51  -- 96  -- 1 <--> 30  -- 14  -- 62  -- 0
7  --> 206 -- 55  -- 103 -- 2 <--> 324 -- 51  -- 129 -- 0
8  --> 324 -- 78  -- 108 -- 0 <--> 10  -- 125 -- 90  -- 1
9  --> 182 -- 156 -- 43  -- 2 <--> 30  -- 133 -- 104 -- 0
10 --> 10  -- 6   -- 48  -- 1 <--> 324 -- 116 -- 40  -- 0
11 --> 256 -- 10  -- 55  -- 1 <--> 10  -- 82  -- 85  -- 1
12 --> 30  -- 14  -- 62  -- 0 <--> 30  -- 90  -- 99  -- 0
il minimo i tale che xi = x2i vale: 12
```

Notiamo che il più piccolo $i$ per cui $x_{2i} = x_i$ è proprio il valore iniziale della testa della rho. Questo è il caso migliore che possa capitare.

### 2.2.3 L'algoritmo di base e la sua analisi euristica

Possiamo ora presentare la base dell'algoritmo rho. Per semplicità assumeremo la presenza di una funzione *walk* $(x_i, a_i, b_i)$ che calcola i cammini random e restituisce le terne $(x_{i+1}, a_{i+1}, b_{i+1})$. In altre parole *walk* calcola $j = S(x_i)$ ed esegue le operazioni

$$x_{i+1} = f(x_i) = \begin{cases} x_i^2 \bmod p & \text{se } S(x_i) = 0 \\ x_i g_j \bmod p & \text{se } S(x_i) = j, j \in \{1, \dots, n_S - 1\} \end{cases}$$

o

$$x_{i+1} = f(x_i) = x_i g_{S(x_i)}$$

e

$$a_{i+1} = \begin{cases} 2a_i \bmod r & \text{se } S(x_i) = 0 \\ a_i + u_{S(x_i)} \bmod r & \text{se } S(x_i) > 0 \end{cases}$$

$$b_{i+1} = \begin{cases} 2b_i \bmod r & \text{se } S(x_i) = 0 \\ b_i + v_{S(x_i)} \bmod r & \text{se } S(x_i) > 0 \end{cases}$$

Notiamo che una volta che la funzione *walk* è definita allora l'algoritmo rho è deterministico. È importante per l'analisi che vi siano diverse possibilità per la funzione walk.

*Algoritmo Rho*

INPUT: $g$, $h \in G$ (se $G = \mathbb{Z}/n\mathbb{Z}$ dovrò dare $n$ in input)
OUTPUT: $a$ tale che $h = g^a$, o $\perp$
1: scegliere a caso una funzione *walk* come spiegato sopra
2: $x_1 = g$, $a_1 = 1$, $b_1 = 0$
3: $(x_2, a_2, b_2)$ = *walk* $(x_1, a_1, b_1)$
4: while $(x_2 \neq x_1)$ do
5: $(x_1, a_1, b_1)$ = *walk* $(x_1, a_1, b_1)$
6: $(x_2, a_2, b_2)$ = *walk* ( *walk* $(x_2, a_2, b_2)$)
7: end while
8: if $b_1 \equiv b_2 \bmod r$ then
9: return $\perp$
10: else
11: return $(a_i - a_j)(b_j - b_i)^{-1} \bmod r$
12: end if

Questo algoritmo termina sempre, ma ci sono casi in cui potrebbe andare male:

- Il valore $(b_j - b_i)$ potrebbe non essere invertibile modulo *r*. Dunque possiamo solo dimostrare che l'algoritmo abbia successo con una certa probabilità
- Il ciclo potrebbe essere molto lungo (quanto r ) e in questo caso l'algoritmo è più lento che la ricerca tramite forza bruta. Dunque, possiamo solo aspettarci di ricavare un tempo atteso per l'algoritmo, ricordando che per tempo atteso in questo caso intendiamo la media, sopra tutte le scelte per la funzione walk, dei peggiori casi di tempi di esecuzione dell'algoritmo su tutti i casi possibili del problema.

È un problema aperto il dare un'analisi rigorosa per il tempo di esecuzione dell'algoritmo. È invece più tradizionale fare l'assunzione di tipo euristico che i cammini pseudorandom definiti sopra si comportino in modo sufficientemente random. Prima di iniziare una precisa analisi euristica determiniamo un'approssimazione del valore atteso dell'epact nel caso di un reale cammino random.

**Euristica.** Sia $x_i$ una sequenza di elementi del gruppo *G* di ordine *r* ottenuta come sopra iterando una funzione random $f : G \rightarrow G$. Ricordiamo che l'epact è il più piccolo intero positivo $i$ tale che $x_{2i} = x_i$. Allora il valore atteso dell'epact è approssimativamente $\frac{\varsigma(2)}{2}\sqrt{\frac{\pi r}{2}} \approx 0.823\sqrt{\frac{\pi r}{2}}$ , dove $\varsigma(2)$ è il valore della funzione zeta di Riemann in 2.

**Argomentazione.** Fissiamo una sequenza $x_i$, e sia $l$ la lunghezza della rho, in modo che $x_{l+1}$ giaccia in $\{x_1, x_2, \dots, x_l\}$. Poiché $x_{l+1}$ può essere una qualsiasi delle $x_i$, la lunghezza del ciclo $l_h$ può essere qualsiasi valore $1 \leq l_h \leq l$ ed ogni possibilità avviene con probabilità $1/l$.

L'epact è il più piccolo multiplo di $l_h$ maggiore di $l_t = l - l_h$. Dunque , se $\frac{l}{2} \leq l_h \leq l$

Allora l'epact sarà $l_h$, se se $\frac{l}{3} \leq l_h \leq \frac{l}{2}$ allora l'epact sarà $2l_h$. In generale se $\frac{l}{k+1} \leq l_h \leq \frac{l}{k}$ allora l'epact sarà $kl_h$.

Il valore atteso dell'epact quando la rho ha lunghezza $l$ è dunque

$$E = \sum_{k=1}^{\infty} \sum_{l_h=1}^{l} k l_h P_l(k, l_h)$$

Dove $P_l(k, l_h)$ è la probabilità che $kl_h$ sia l'epact. Per la discussione di sopra $P_l(k, l_h) = \frac{1}{l}$ se $\frac{l}{k+1} \leq l_h \leq \frac{l}{k}$ o se $(k, l_h) = (1, l)$ e 0 altrimenti.

Dunque

$$E = \frac{1}{l} \sum_{k=1}^{l-1} k \sum_{\substack{\frac{l}{k+1} \leq l_h \leq \frac{l}{k} \\ \text{oppure } (k,l_h)=(1,l)}} l_h$$

Approssimando la sommatoria più interna come $\frac{1}{2}\left(\left(\frac{l}{k}\right)^2 - \left(\frac{l}{k+1}\right)^2\right)$ otteniamo

$$E \approx \frac{l}{2} \sum_{k=1}^{l-1} k \left(\frac{1}{k^2} - \frac{1}{(k+1)^2}\right)$$

Ora,

$$k\left(\frac{1}{k^2} - \frac{1}{(k+1)^2}\right) = \frac{1}{k} - \frac{1}{k+1} + \frac{1}{(k+1)^2}$$

e

$$\sum_{k=1}^{\infty} \left(\frac{1}{k} - \frac{1}{k+1}\right) = 1$$

$$\sum_{k=1}^{\infty} \frac{1}{(k+1)^2} = \varsigma(2) - 1$$

Dunque $E \approx l/2(1 + \varsigma(2) - 1)$. È risaputo che $\varsigma(2) = 1.645$.

Infine scriviamo Pr($e$) per la probabilità che l'epact sia *e*, Pr($l$) per la probabilità che la lunghezza della rho sia *l*, e Pr($e \mid l$ ) per la probabilità condizionata che l'epact sia *e* dato che la lunghezza della rho sia *l*. L'attesa per *e* è allora:

$$\begin{aligned} E(e) &= \sum_{e=1}^{\infty} e\Pr(e) = \sum_{e=1}^{\infty} e \sum_{l=1}^{\infty} Pr(e|l)\Pr(l) \\ &= \sum_{l=1}^{\infty} \Pr(l) \left(\sum_{e=1}^{\infty} e\Pr(e|l)\right) \\ &= \sum_{l=1}^{\infty} \Pr(l)\, E \approx \left(\frac{\varsigma(2)}{2}\right) E(l) \end{aligned}$$

Che completa l'argomentazione.

□

**Euristica.** Supponiamo che $r$ sia sufficientemente grande, che $n_S$ sia sufficientemente grande e che la funzione *walk* sia scelte casualmente. Allora

1- Esiste un $\epsilon > 0$ piccolo tale che il valore atteso per l'epact sia

$$(1+\epsilon)0.823\sqrt{\pi r/2}$$

2- Il valore $\sum_{i=l_t}^{l_t+l_h-1} v_{S(x_i)} \bmod r$ è uniformemente distribuito in $\mathbb{Z}/r\mathbb{Z}$.

L'esperienza pratica mostra che se il numero di partizioni $n_S$ nei cammini pseudorandom è sufficientemente grande e se le coppie $(u_j, v_j)$ (usate per definire i salti $g_j = g^{u_j} h^{v_j}$ ) sono scelti uniformemente a random allora valgono le proprietà enunciate nelle precedenti euristiche.

Teske in [15] mostra che il cammino rho originale con $n_S$ = 3 non raggiunge un tempo di esecuzione ottimale (più avanti daremo qualche risultato euristico sulla dipendenza da $n_S$) e inoltre suggerisce che nella pratica bisognerebbe prendere $n_S \geq 20$, caso in cui la lunghezza attesa della rho è $(1+\epsilon)\sqrt{\pi r/2}$ per $\epsilon < 0.04$.

**Teorema.** Assumiamo le notazioni precedenti e l'ultima euristica. Allora l'algoritmo rho con il metodo di Floyd per la ciclicità ha un tempo di esecuzione atteso di $3.093(1+\epsilon)\sqrt{r}$ operazioni di gruppo. La probabilità che l'algoritmo fallisca è trascurabile.

Da un punto di vista di complessità teorica, se le operazioni del gruppo in $G$ avvengono in tempo polinomiale, allora il tempo di esecuzione dell'algoritmo rho è di $O(\sqrt{r})$ operazioni elementari.

**Dimostrazione.** Il numero di iterazioni del ciclo principale dell'algoritmo è l'epact. Per l'ultima euristica il valore attesto dell'epact è $(1+\epsilon)0.823\sqrt{\pi r/2}$.

L'algoritmo richiama tre volte la funzione *walk* in ogni iterazione. Ogni chiamata a questa funzione richiede una operazione del gruppo e due addizioni modulo $r$ (ignoreremo però queste due addizioni in quanto il loro costo è significativamente minore di un'operazione del gruppo). Dunque il numero atteso di operazioni del gruppo è $(1+\epsilon)0.823\sqrt{\pi r/2} \approx (1+\epsilon)3.093\sqrt{r}$, come enunciato.

L'algoritmo fallisce solamente se $b_{2i} \equiv b_i \bmod r$. Abbiamo che $g^{a_{l_t}} h^{b_{l_t}} = g^{a_{l_t+l_h}} h^{b_{l_t+l_h}}$ da cui segue che $a_{l_t+l_h} = a_{l_t} + u$, $b_{l_t+l_h} = b_{l_t} + v$, dove $g^u h^v = 1$. Precisamente, $v \equiv b_{l_t+l_h} - b_{l_t} \equiv \sum_{i=l_t}^{l_t+l_h+1} v_{S(x_i)} \bmod r$.

Scriviamo $i = l_t + i'$ per qualche $0 \leq i' < l_h$ e $b_i = b_{l_t} + w$. Assumiamo che $l_h \geq 2$ (la probabilità che $l_h = 1$ è trascurabile). Allora $2i = l_t + xl_h + i'$ per qualche intero $1 \leq x < (l_t + 2l_h)/l_h < r$ e quindi $b_{2i} = b_{l_t} + xv + w$. Segue che $b_{2i} \equiv b_i \bmod r$ se e solo se $r|v$.

Per quanto detto nell'ultima euristica il valore *v* è uniformemente distribuito in $\mathbb{Z}/r\mathbb{Z}$ e quindi la probabilità che esso sia zero è $1/r$, che è una quantità trascurabile rispetto alle dimensioni dell'input del problema.

□

**Esempio**[esercizio 17.15 pag 211 di [12]]. Sia *p* = 569 e sia *g* = 262 e *h* = 5, per il quale si può controllare che l'ordine sia 71 modulo *p*. Usare l'algoritmo rho per calcolare il logaritmo discreto di *h* rispetto alla base *g* modulo *p*.

Ecco l'output del programma java con $n_S = 4$ e u[1] = 32, v[1] = 54, u[2] = 76, v[2] = 45, u[3] = 76, v[3] = 65.

```
i --  xi  -- ai -- bi -- S(xi)- x2i -- a2i - b2i - S(x2i)
-------------------------------------------------------------
1 --> 262 -- 1  -- 0  -- 2 <--> 447 -- 6  -- 45 -- 3
2 --> 447 -- 6  -- 45 -- 3 <--> 315 -- 43 -- 22 -- 3
3 --> 209 -- 11 -- 39 -- 1 <--> 488 -- 53 -- 61 -- 0
4 --> 315 -- 43 -- 22 -- 3 <--> 411 -- 40 -- 25 -- 3
5 --> 90  -- 48 -- 16 -- 2 <--> 447 -- 19 -- 38 -- 3
6 --> 488 -- 53 -- 61 -- 0 <--> 315 -- 56 -- 15 -- 3
7 --> 302 -- 35 -- 51 -- 2 <--> 488 -- 66 -- 54 -- 0
8 --> 411 -- 40 -- 25 -- 3 <--> 411 -- 66 -- 11 -- 3
il minimo i tale che xi = x2i vale: 8
il logaritmo cercato vale: 12
```

**Osservazione** [esercizio 17.16 pag 211 di [12]]. La definizione di cammino rho e l'equazione di aggiornamento di $a_i$ e $b_i$ può essere modificata rimpiazzando $g_j$ da $g^{u_j}$ o da $h^{v_j}$ (indipendentemente per ciascuna $j$ ). Mostriamo che ciò permette di risparmiare un'addizione modulare in ogni iterazione dell'algoritmo. Spieghiamo inoltre perché questa ottimizzazione non influisce sul successo dell'algoritmo, sempre che il cammino utilizzi tutti i valori per $S(x_i)$ con circa la stessa probabilità.

La definizione di cammino rho è la seguente:

$$x_{i+1} = f(x_i) = \begin{cases} x_i^2 & \text{se } S(x_i) = 0 \\ x_i g_j & \text{se } S(x_i) = j, j \in \{1, \dots, n_S - 1\} \end{cases}$$

Dove $g_j = g^{u_j} h^{v_j}$. Quando trovo una collisione del tipo $x_i = x_{2i}$ ho che $g^a h^b = g^{a'} h^{b'}$ e quindi trovo il logaritmo discreto ponendo $g^{(a-a')(b'-b)^{-1}} = h$.

Ponendo per esempio

$$x_{i+1} = f(x_i) = \begin{cases} x_i^2 & \text{se } S(x_i) = 0 \\ x_i g^{u_j} & \text{se } S(x_i) \text{ è pari} \\ x_i h^{v_j} & \text{se } S(x_i) \text{ è dispari} \end{cases}$$

e supponendo che il cammino utilizzi tutti i valori per $S(x_i)$ con circa la stessa probabilità, arriviamo comunque all'identità $g^a h^b = g^{a'} h^{b'}$ da cui ricavare il logaritmo. I cammini per arrivare ad *a* e *b* vengono così modificati:

$$a_{i+1} = \begin{cases} 2a_i \bmod r & \text{se } S(x_i) = 0 \\ a_i + u_{S(x_i)} \bmod r & \text{se } S(x_i) \text{ è pari} \end{cases}$$

$$b_{i+1} = \begin{cases} 2b_i \bmod r & \text{se } S(x_i) = 0 \\ b_i + v_{S(x_i)} \bmod r & \text{se } S(x_i) \text{ è dispari} \end{cases}$$

Risparmiando così metà delle somme modulo *r*.

Il metodo di Floyd per trovare la ciclicità non è un metodo molto efficiente. Per quanto ogni metodo per trovare la ciclicità richieda di calcolare almeno $l_t + l_h$ operazioni del gruppo, il metodo di Floyd richiede in media $2.47(l_t + l_h)$ operazioni del gruppo. Inoltre la sequenza "più lenta" $x_i$ ricalcola elementi che sono già stati calcolati dalla sequenza $x_{2i}$. Vedremo più avanti un metodo introdotto da Brent che richiede la memorizzazione di due soli elementi ma un minor numero di operazioni di gruppo. Si può ottenere anche di meglio utilizzando più memoria. La via più efficiente per trovare i cicli (in senso di tempo e non di memoria) è quella di usare *punti distinti*, che vedremo più avanti.

Nella pratica si usa una versione distribuita dell'algoritmo rho, la quale non richiede il calcolo di cicli. Quindi, per i nostri scopi, non c'è bisogno di dare i dettagli di metodi migliori per trovare la ciclicità.

### 2.2.4 Verso un analisi rigorosa di Pollard Rho

L'ultimo teorema enunciato non è soddisfacente poiché l'euristica alla sua base è essenzialmente equivalente all'enunciato "l'algoritmo rho ha un tempo atteso di esecuzione di $\frac{9}{4}(1+\epsilon)\sqrt{\pi r/2}$ operazioni di gruppo". La ragione per cui abbiamo enunciato l'euristica è per chiarire esattamente quali proprietà dei cammini pseudorandom sono richieste. La ragione per credere a questa euristica è che esperimenti con l'algoritmo rho confermano le stime fatte sui tempi di esecuzione.

Poiché l'algoritmo è fondamentale per la comprensione della crittografia basate sulle curve ellittiche (e per i metodi torus/trace) è naturale domandarsi una trattazione completa e rigorosa dell'algoritmo. D'altra parte, gli esperimenti pratici mostrano che le performance dell'algoritmo sono molto vicine alle predizioni teoriche. Inoltre, da un punto di vista crittografico, per il compito di determinare la dimensione della chiave è sufficiente avere un limite inferiore sul tempo di esecuzione dell'algoritmo. Dunque, nella pratica, l'assenza della dimostrazione di tempi di esecuzione non è necessariamente un problema serio.

Diamo ora una breve panoramica di risultati teorici sull'algoritmo rho. I metodi usati per ottenere questi risultati vanno aldilà degli scopi della tesi, perciò non daremo tutti i dettagli. Notiamo che tutti i risultati esistenti sono in un modello ideale dove la selezione della funzione *S* è una funzione random.

I risultati principali per il cammino rho originale (con $n_S = 3$) sono dovuti a Horwitz e Venkatesan [16], Miller e Verkatesan [17], Kim, Montenegro, Peres e Tetali [18,19].

L'idea di base è definire un *grafo rho*, che è un grafo diretto con insieme dei vertici $< g >$ ed un lato da $x_1$ a $x_2$ se $x_2$ è il passo successivo a $x_1$ nel cammino pseudorandom. Fissiamo un intero *n*. Definiamo la distribuzione $D_n$ su $< g >$ ottenuta scegliendo uniformemente a random $x_1 \in < g >$, eseguendo il cammino per *n* passi, e memorizzando il punto finale del cammino. La proprietà cruciale da studiare è il *tempo misto*, che, informalmente, è il più piccolo intero *n* tale che $D_n$ è

"sufficientemente simile" alla distribuzione uniforme. Per questi risultati, l'operazione di elevamento al quadrato presente nel cammino originale è cruciale.

Enunciamo il principale risultato di Miller e Verkatesan [17].

**Teorema.** (teorema 1.1 di [17]). Fissiamo $\epsilon > 0$. Allora l'algoritmo rho usando il cammino originale con $n_S = 3$, trova una collisione nel tempo $O_\epsilon(\sqrt{r}\log(r)^3)$ con probabilità almeno $1 - \epsilon$ , dove la probabilità è presa su tutte le partizioni di $< g >$ nei tre insiemi $S_1$, $S_2$ ed $S_3$. La notazione $O_\epsilon$ significa che la costante implicita in *O* dipende da $\epsilon$.

Kim, Montenegro, Peres e Tetali hanno migliorato questo risultato in [18] al desiderato $O_\epsilon(\sqrt{r})$. Notiamo che tutti questi lavori lasciano la costante in *O* non specificata.

Notiamo anche che il modello ideale con *S* funzione realmente random non è implementabile con occupazione di memoria costante (e nemmeno polinomiale). Dunque questi risultati non possono essere applicati all'algoritmo precedentemente presentato, poiché la nostra funzione *S* sono molto lontane dall'essere scelte uniformemente tra tutte le partizioni possibili dell'insieme $< g >$. Il numero delle partizioni possibili di $< g >$ in tre sottoinsiemi delle stesse dimensioni è (per convenienza poniamo 3 | *r*)

$$\binom{r}{r/3}\binom{2r/3}{r/3}$$

che, usando il fatto che $\binom{a}{b} \geq \left(\frac{a}{b}\right)^b$ , è almeno $6^{r/3}$. Come accennato in precedenza (esercizio 17.7 di [12] ), la nostra costruzione implica tipicamente un numero di scelte di *S* minore di $r^3$.

Notiamo che questi risultati danno il tempo atteso per una collisione. Comunque, una collisione potrebbe non condurre alla soluzione del DLP nel raro caso in cui $b_{2i} \equiv b_i \bmod r$.

Sattler e Schnorr [20] e Teske [15] hanno considerato i cammino rho additivi. Una funzione chiave del loro lavoro è la discussione sul numero di partizioni $n_S$. Sattler e Schnorr mostrano (soggetti a congettura) che se $n_S \geq 8$ allora il tempo atteso di esecuzione dell'algoritmo rho è di $c\sqrt{\pi r/2}$ operazioni del gruppo per una

costante *c* esplicita. Teske mostra invece, usando alcuni risultati di Hildebrand, che il cammino additivo dovrebbe approssimare la distribuzione uniforme dopo meno di $\sqrt{r}$ passi una volta che $n_S \geq 6$. Lei raccomanda di usare il cammino additivo con $n_S \geq 20$ e, quando ciò è fatto, congettura che la lunghezza del ciclo attesa è $\leq 1.3\sqrt{r}$ (paragonato con il teorico $\approx 1.2533\sqrt{r}$ ; il quale da la costante $\epsilon < 0.04$, come menzionato prima).

Ulteriori motivazioni per l'utilizzo di un $n_S$ grande sono date da Brent e Pollard [8], Arney e Bender [21] e Blackburn e Murphy [22]. Essi presentano argomenti euristici per cui la lunghezza del ciclo attesa usando $n_S$ partizioni è $\sqrt{c_{n_S}\pi r/2}$ dove $c_{n_S} = n_S/(n_S - 1)$. Questa euristica è supportata dai risultati sperimentali di Teske [15]. Sia $G =< g >$. La loro analisi considera il grafo diretto formato iterando la funzione *walk* : $G \rightarrow G$ (cioè il grafo che ha *G* come insieme dei vertici e lati da *g* a *walk*(*g*) ). Allora per un grafo di questo tipo scelto casualmente, $n_S/(n_S - 1)$ è la varianza dell'in-degree (numero di frecce entranti in un nodo) per questo grafo, che è lo stesso che il valore atteso di $n(x) = \#\{y \in G: y \neq x, walk(y) = walk(x)\}$.

## 2.3 Pollard Rho Distribuito

In questa sezione spieghiamo come può essere parallelizzato l'algoritmo Pollard Rho. Più che un modello di calcolo parallelo considereremo un *modello di calcolo distribuito*. In questo modello c'è un *server* ed $N_P \geq 1$ *client* (a cui ci riferiremo anche con il nome *processori*). Non c'è condivisione di memoria o comunicazione diretta tra i vari client. Invece, il server può spedire messaggi ai client ed ognuno di essi può mandare messaggi al server. In generale preferiamo minimizzare la quantità di comunicazioni tra server e client.

Per risolvere un problema di logaritmo discreto il server attiverà un numero di client, fornendo ad ognuno i propri dati iniziali. I client eseguiranno i cammini pseudorandom e occasionalmente manderanno i dati indietro al server. Prima o poi il server collezionerà abbastanza informazioni per risolvere il problema, caso in cui spedirà a tutti i client l'istruzione di terminare l'esecuzione.

Più precisamente, i client trovano gli elementi del gruppo $x_i = g^{a_i} h^{b_i}$ di una certa forma "distinta" e inviano la corrispondente terna $(x_i, a_i, b_i)$ al server. Questi attende fino a che riceve lo stesso elemento $x$ del gruppo in due terne diverse $(x, a', b')$ e $(x, a'', b'')$, caso in cui ha $g^{a'} h^{b'} = g^{a''} h^{b''}$ e può risolvere il DLP come nel caso seriale (non parallelo).

Il meglio che ci si può aspettare da un calcolo distribuito è un accelerazione lineare rispetto al caso seriale (poiché se il lavoro totale nel caso distribuito fosse minore di quello del caso seriale allora ciò condurrebbe ad un algoritmo più veloce nel caso seriale). In altre parole, con $N_P$ client speriamo di ottenere un tempo di esecuzione proporzionale a $\sqrt{r}/N_P$.

### 2.3.1 Punti Distinti

L'idea di usare punti distinti in problemi di ricerca è probabilmente di Rivest. La prima applicazione di questa idea per calcolare il logaritmo discreto è di Van Oorshot e Wiener [23].

**Definizione.** Un elemento $g \in G$ si dice *punto distinto* se la sua espansione binaria $b(g)$ soddisfa qualche proprietà facilmente controllabile. Denotiamo con $D \subset G$ l'insieme dei punti distinti. La probabilità che un elemento del gruppo scelto uniformemente sia un punto distinto, cioè $\#D/\#G$, è denotata da $\theta$.

Un tipico esempio è il seguente.

**Esempio.** Sia E una curva ellittica su $\mathbb{F}_p$. Un punto $P = (x_P, y_P) \in E(\mathbb{F}_p)$ è rappresentato come stringa binaria $b(P)$ costituita dalla rappresentazione binaria dell'intero $0 \leq x_P < p$ seguita dalla rappresentazione binaria dell'intero $0 \leq y_P < p$.

Fissiamo un intero $n_D$. Definiamo $D$ come l'insieme dei punti $P \in E(\mathbb{F}_p)$ tali che le $n_D$ cifre meno significative di $x_P$ siano zero. In altre parole

$$D = \{P = (x_P, y_P) \in E(\mathbb{F}_p) : x_P \equiv 0 \bmod 2^{n_D}, \text{dove } 0 \leq x_P < p\}$$

Allora $\theta \approx 1/2^{n_D}$.

Nell'analisi assumiamo di scegliere i punti uniformemente e indipendentemente a random. È importante determinare il numero atteso di passi prima di trovare un punto distinto.

**Lemma.** Sia $\theta$ la probabilità che un punto scelto a caso sia un punto distinto. Allora

1. La probabilità di scegliere $\alpha/\theta$ punti, nessuno dei quali sia distinto, è approssimativamente$e^{-\alpha}$ quando $1/\theta$ è grande.

2. Il numero atteso di punti da scegliere prima di trovare un punto distinto è $1/\theta$.

3. Se si sono già scelti *i* punti, nessuno dei quali è distinto, allora il numero atteso di punti ancora da scegliere prima di trovarne uno distinto è $1/\theta$.

**Dimostrazione.** La probabilità che $i$ punti scelti non siano distinti è $(1-\theta)^i$. Di conseguenza la probabilità di scegliere $\alpha/\theta$ punti, nessuno dei quali sia distinto, è

$$(1-\theta)^{\frac{\alpha}{\theta}} = \left(\left(1-\frac{1}{\frac{1}{\theta}}\right)^{\frac{1}{\theta}}\right)^{\alpha} \approx \mathrm{e}^{-\alpha}$$

Dove $\frac{1}{\theta}$ è grande.

Il secondo enunciato è la formula standard per il valore atteso di una distribuzione geometrica, ma per completezza ne diamo una dimostrazione. Per definizione, il valore atteso è

$$S = \sum_{i=1}^{\infty} i(1-\theta)^{i-1}\theta$$

Che è assolutamente convergente per il test del rapporto. Allora

$$\theta S = S - (1-\theta)S = \theta\left(\sum_{i=1}^{\infty} i(1-\theta)^{i-1} + \sum_{i=1}^{\infty}(i-1)(1-\theta)^{i-1}\right)$$

$$= \theta\sum_{j=1}^{\infty}(1-\theta)^{j} = 1$$

Il risultato segue facilmente.

Per il terzo enunciato, supponiamo che qualcuno abbia già estratto $i$ punti senza trovarne nessuno distinto. Poiché i tentativi sono indipendenti, la probabilità di scegliere altri punti $j$ che non siano distinti rimane $(1-\theta)^j$. Dunque il numero di punti ancora da scegliere è ancora $1/\theta$.

□

**Proprietà** [Esercizio 17.22 pag 214 di [12] ]**.** Se $N_P \leq \theta \sqrt{\frac{\pi r}{2}} / \ln r$ allora non ci si aspetta mai un cammino con più di $\sqrt{\frac{\pi r}{2}} / N_P$ passi prima di colpire un punto distinto.

**Dimostrazione.** Per quanto visto nel lemma il numero atteso di passi per trovare un punto distinto è $\frac{1}{\theta}$. Da $N_P \leq \frac{\theta \sqrt{\frac{\pi r}{2}}}{\ln r}$ ricaviamo che $\frac{1}{\theta} \leq \frac{\sqrt{\frac{\pi r}{2}}}{N_P \ln r}$. Poiché $r$, l'ordine di $g \in G$, è maggiore di $e$, il membro sinistro della disequazione è sempre minore di $\frac{\sqrt{\frac{\pi r}{2}}}{N_P}$.

□

### 2.3.2 L'algoritmo e la sua analisi euristica

Tutti i processori eseguono lo stesso cammino pseudorandom $(x_{i+1}, a_{i+1}, b_{i+1}) = walk(x_i, a_i, b_i)$ come visto nel caso seriale, ma ogni processore inizia da un punto casuale diverso. Ogni volta che un processore intercetta un punto distinto allora spedisce la terna $(x_i, a_i, b_i)$ al server e poi riprende il suo cammino da un nuovo punto casuale $(x_1, a_1, b_1)$. Se capita che un processore visita un punto visitato da un altro processore allora i due cammini da quel punto in poi sono identici ed entrambi finiranno allo stesso punto distinto. Notiamo che questo algoritmo non richiede "auto collisioni" nel cammino e perciò il nome rho non è più giustificato, ma continueremo ad usarlo per motivi storici. Quando il server riceve due triple $(x, a', b')$ e $(x, a'', b'')$ per lo stesso elemento *x* del gruppo ma con $b' \not\equiv b'' \bmod r$, allora il server calcola il logaritmo discreto e invia il segnale d'interruzione a tutti i client. A breve scriveremo l'algoritmo; notiamo che se questo termina, per come è stato costruito, allora la risposta è corretta.

Analizziamo prima la performance di questo algoritmo. Per avere un risultato pulito, supponiamo che nessun client possa avere dei crash, che la comunicazione tra server e client sia perfettamente realizzabile, che tutti i client godano della stessa efficienza computazionale ed eseguano le operazioni in modo continuo (in altre parole, ogni processore calcola lo stesso numero di operazioni elementari in periodi

di tempo uguali). Anche in questo caso è un problema aperto il dare un analisi rigorosa per il metodo rho distribuito. Dunque, enunciamo la seguente euristica per l'analisi dell'algoritmo.

**Euristica.** Supponiamo che *r* sia sufficientemente grande e che la funzione *walk* sia scelta a random. Allora:

1. Il numero atteso di elementi del gruppo che devono essere selezionati prima che lo stesso elemento sia selezionato due volte è $\sqrt{\pi r/2}$.

2. Vale l'enunciato del lemma sui punti distinti, ed ogni cammino raggiunge prima o poi un punto distinto.

Affermiamo che questa euristica vale quando: il numero di partizioni *n* nei cammini pseudorandom è sufficientemente grande; le coppie $(u_j, v_j)$ (usate per definire i salti $g_j = g^{u_j} h^{v_j}$) sono scelte uniformemente a random; i punti iniziali $(a, b)$ di ogni cammino sono scelti uniformemente a random; l'insieme *D* dei punti distinti è distribuito equamente sul gruppo *G*; la probabilità $\theta$ è sufficientemente grande; il numero $N_P$ di client è sufficientemente piccolo (per esempio $N_P < \theta\sqrt{\frac{\pi r}{2}} / \ln r$ ; vedere l'ultimo esercizio al proposito).

*Algoritmo rho distribuito: server side*

INPUT: $g, h \in G$

OUTPUT: $a$ tale che $h = g^a$

1: scegliere a random una funzione cammino $walk(x, a, b)$
2: inizializzare a *null* una struttura *L* con facile algoritmo di ricerca (sorted list, binary tree, ecc.)
3: dare il via a tutti i processori con la funzione *walk*.
4: while (DLP è irrisolto) do
5: ricevere la tripla $(x_i, a_i, b_i)$ da ogni client e inserirla in *L*.
6: if (la prima coordinata della nuova tripla $(x, a'', b'')$ coincide con una tripla esistente $(x, a', b')$ ) then
7: if ($b' \not\equiv b'' \bmod r$) then
8: invia il messaggio di fine esecuzione ai client
9: return $(a'' - a')(b' - b'') \bmod r$
10: end if
11: end if
12: end while

*Algoritmo rho distribuito: client side*

```
INPUT: g,h ∈ G, funzione walk
1:  while (il segnale di fine esecuzione non è ricevuto) do
2:    scegliere uniformemente a random 0 ≤ a_1, b_1 < r
3:    porre: x = g^{a_1} h^{b_1}
4:    while (x ∉ D) do
5:       (x, a_1, b_1) = walk(x, a_1, b_1)
6:    end while
7:    invia (x, a_1, b_1) al server
8:  end while
```

**Teorema.** Assumiamo la solita notazione, in particolare con $N_P$ numero di client e $\theta$ probabilità che l'elemento del gruppo sia un punto distinto. Assumiamo che valga la precedente euristica. Allora, per ogni client, il tempo di esecuzione del metodo rho distribuito è $\frac{\sqrt{\frac{\pi r}{2}}}{N_P} + \frac{1}{\theta}$ operazioni del gruppo.

La memoria necessaria al server è di $\theta\sqrt{\pi r/2} + N_P$ punti.

**Dimostrazione.** L'ultima euristica enunciata dice che ci aspettiamo di selezionare $\sqrt{\pi r/2}$ elementi del gruppo in totale prima di raggiungere una collisione. Poiché questo lavoro è distribuito su $N_P$

client di uguale velocità, segue che ogni client dovrebbe eseguire $\sqrt{\pi r/2}/N_P$ operazioni di gruppo prima di una collisione. Il server non scoverà questa collisione fino a che il secondo client non colpisce un punto distinto, operazione che dovrebbe richiedere $1/\theta$ passi ulteriori per l'euristica (parte 3 del lemma). Dunque ogni client dovrebbe eseguire $\frac{\sqrt{\frac{\pi r}{2}}}{N_P} + \frac{1}{\theta}$ passi nel cammino.

Ovviamente una collisione $g^{a'} h^{b'} = g^{a''} h^{b''}$ può risultare inutile nel senso che $b' \equiv b'' \bmod r$. Una collisione implica che $a' + ab' \equiv a'' + ab'' \bmod r$ dove $h = g^a$; ci sono $r$ coppie$(a'', b'')$ per ogni coppia $(a', b')$. Poiché ogni cammino comincia con i valori uniformemente a random $(a_1, b_1)$, segue che i valori $(a'', b'')$ sono uniformemente distribuiti sulle $r$ possibilità. Dunque, la probabilità che una collisione sia inutile è 1 / $r$ e il numero atteso di collisioni richieste è 1.

Ogni processore esegue per $\frac{\sqrt{\frac{\pi r}{2}}}{N_P} + \frac{1}{\theta}$ passi e quindi ci si aspetta che invii $\theta \frac{\sqrt{\frac{\pi r}{2}}}{N_P} + 1$ punti distinti nel corso dell'esecuzione. Il numero totale di punti da memorizzare è dunque $\theta\sqrt{\pi r/2} + N_P$.

□

Schulte-Geers [24] analizzano la scelta di $\theta$ e mostrano che l'euristica alla base del teorema è valida solamente quando $\theta r = o(\sqrt{r})$ (cioè quando $\theta = c\log(r)/\sqrt{r}$). Poiché $\frac{1}{\theta} = \frac{\sqrt{r}}{c\log(r)}$ è molto improbabile che ci sia un'auto collisione (e dunque un ciclo) prima di scovare un punto distinto. La quantità di memoria attesa è dunque di $\sqrt{\pi/2}c\log(r)$ elementi del gruppo. Un corollario dei risultati di [24] è che i metodi dei punti distinti non sono asintoticamente efficienti se la memoria è ristretta ad un numero costante di elementi del gruppo.

**Proprietà** (Kuhn-Struik [25] ). Supponiamo che siano dati $g, h_1, \dots, h_L$ (dove $L < r^{1/4}$) e che sia richiesto di trovare tutti gli $a_i$ per $1 \le i \le L$ tali che $h_i = g^{a_i}$. Questo può essere fatto in circa $\sqrt{2rL}$ operazioni di gruppo.

## 2.4 Accelerare l'Algoritmo Rho usando le Classi di Equivalenza

Gallant, Lambert e Vanstone [26] e Wiener e Zuccherato [27] hanno mostrato che in alcuni casi si può accelerare il metodo di rho definendo dei cammini pseudorandom non sul gruppo $< g >$, ma su un insieme di classi di equivalenza. Ciò è essenzialmente la stessa cosa che lavorare, anziché in un gruppo algebrico, nel suo quoziente.

Supponiamo che ci sia una relazione di equivalenza su $< g >$. Denotiamo con $\bar{x}$ la classi di equivalenza $x \in < g >$. Sia $N_C$ la dimensione di una generica classe di equivalenza. Richiediamo che valgano le seguenti proprietà:

1. Si deve poter definire un unico rappresentante $\hat{x}$ per ogni classe di equivalenza $\bar{x}$.

2. Dati $(x_i, a_i, b_i)$ tali che $x_i = g^{a_i} h^{b_i}$ allora si devono poter calcolare $(\widehat{x_i}, \widehat{a_i}, \widehat{b_i})$ tali che $\widehat{x_i} = g^{\widehat{a_i}} h^{\widehat{b_i}}$.

Daremo tra breve qualche esempio.

A questo punto si può implementare l'algoritmo rho sulle classi di equivalenza definendo una funzione di cammino pseudorandom $walk(x_i, a_i, b_i)$ come in precedenza. Più precisamente, ponendo $x_1 = g$, $a_1 = 1$ e $b_1 = 0$ e definendo (usando il "cammino originale") la sequenza $x_i$ come:

$$x_{i+1} = f(x_i) = \begin{cases} \widehat{x_i}^2 & \text{se } S(\widehat{x_i}) = 0 \\ \widehat{x_i} g_j & \text{se } S(\widehat{x_i}) = j, j \in \{1, \dots, n_S - 1\} \end{cases}$$

Dove la funzione di selezione $S$ e i valori $g_j = g^{u_j} h^{v_j}$ sono definiti come nel caso semplice. Usando i punti distinti si può definire che una classe di equivalenza sia distinta se l'unico suo rappresentante ha la proprietà di distinzione.

**Teorema.** Sia $G$ un gruppo e $g \in G$ un suo elemento di ordine $r$. Supponiamo che esista una relazione di equivalenza su $< g >$ come sopra. Sia $N_C$ la generica dimensione di una classe di equivalenza. Sia $C_1$ il numero di operazioni elementari necessarie per svolgere un operazione del gruppo in $< g >$, e $C_2$ il numero di operazioni elementari necessario per calcolare l'unico rappresentante $\widehat{x_i}$ di una classe di equivalenza (e per calcolare $\widehat{a_i}, \widehat{b_i}$).

Consideriamo l'algoritmo rho distribuito come sopra (ignorando la possibilità di cicli corti (al riguardo vedere le sezioni successive) ). Sotto un'assunzione euristica per le classi di equivalenza analoga a quella per il problema semplice, il tempo atteso per risolvere il problema del logaritmo discreto è di

$$\left(\frac{\sqrt{\frac{\pi r}{2} N_C}}{N_P} + \frac{1}{\theta}\right)(C_1 + C_2)$$

operazioni elementari.

Calcolare un unico rappresentante per classe di equivalenza di solito implica il dover elencare tutti gli elementi della classe, e quindi $\tilde{O}(N_C)$ operazioni elementari. Dunque, ingenuamente, il tempo di esecuzione sarà di $\tilde{O}\left(\frac{\sqrt{\frac{\pi r}{2} N_C}}{N_P}\right)$ operazioni elementari, che è peggio che eseguire l'algoritmo rho senza le classi di equivalenza. Tuttavia, nella pratica si usa questo metodo solo quando $C_2 < C_1$, caso in cui l'accelerazione è significativa.

### 2.4.1 Esempi di classi di equivalenza

Diamo alcuni esempi di queste classi di equivalenza su alcuni gruppi algebrici.

**Esempio 1.** Per un gruppo *G* con inverso facilmente calcolabile (per esempio le curve ellittiche $E(\mathbb{F}_q)$ o il toro algebrico $T_n$ con $n > 1$, vedere cap 7.3 di [12] ) si può definire la relazione di equivalenza $x \equiv x^{-1}$. In questo caso abbiamo $N_C = 2$ (sebbene alcuni elementi, come l'identità e gli elementi di ordine 2, siano identici al loro inverso e quindi le loro classi di equivalenza abbiano dimensione 1). Se $x_i = g^{a_i} h^{b_i}$ allora chiaramente ${x_i}^{-1} = g^{-a_i} h^{-b_i}$. Si può definire un unico rappresentante per classe $\hat{x}$ per esempio imponendo un ordine lessicografico sulla rappresentazione binaria degli elementi della classe.

Possiamo generalizzare questo esempio come segue.

**Esempio 2.** Sia $G$ un gruppo algebrico definito su $\mathbb{F}_q$ con un gruppo auto morfismo Aut($G$) di dimensione $N_C$. Supponiamo che per $g \in G$ di ordine $r$ si abbia $\psi(g) \in < g >$ per ogni $\psi \in \mathrm{Aut}(G)$. Inoltre assumiamo che per ogni $\psi \in \mathrm{Aut}(G)$ si possano calcolare gli auto valori $\lambda_\psi \in \mathbb{Z}$ tali che $\psi(g) = g^{\lambda_\psi}$. Allora per $x \in G$ si può definire $\bar{x} = \{\psi(x) : \psi \in \mathrm{Aut}(G)\}$.

Nuovamente, si può definire $\hat{x}$ imponendo un ordine lessicografico sulla rappresentazione binaria degli elementi della classe.

### 2.4.2 Problemi con i Cicli

Uno dei problemi che si può incontrare è quello dei cicli inutili.

**Proprietà.** Supponiamo che la relazione di equivalenza sia $x \equiv x^{-1}$. Fissiamo $x_i = \widehat{x_i}$ e sia $x_{i+1} = \widehat{x_i} g$. Supponiamo che $\widehat{x_{i+1}} = x_{i+1}^{-1}$ e che $S(\widehat{x_{i+1}}) = S(\widehat{x_i})$. Allora $x_{i+2} \equiv x_i$ e quindi c'è un ciclo di ordine 2. Supponiamo che le classi di equivalenza abbiano genericamente dimensione $N_C$. Allora, sotto l'assunzione che la funzione $S$ sia perfettamente random e che $\hat{x}$ sia un elemento scelto a caso della classe di equivalenza, la probabilità che un $x_i$ scelto a caso conduca ad un ciclo di ordine 2 è $1/(N_C n_S)$.

Una discussione teorica sui cicli è stata data in [26] e da Duursma, Gaudry e Morain [28]. Come mostra l'esercizio appena citato, i cicli inutili sono molto frequenti e verranno incontrati sovente nell'algoritmo. Con l'aggiunta di una piccola quantità di calcoli e di memoria si possono individuare i cicli piccoli (per esempio memorizzando un numero prefissato di valori consecutivi del cammino, o usando regolarmente un algoritmo per trovare la ciclicità per un piccolo numero di passi). Una via ben definita per aggirare cicli corti è specificare un unico elemento del ciclo e fare un passo (diverso da quello usato nel ciclo) da quel punto; Gallant, Lambert e Vanstone chiamano questo metodo il *collapsing the cicle* (sezione 6 di [26]).

Gallant, Lambert e Vanstone [26] hanno presentato un cammino diverso che in generale non conduce a cicli brevi. Sia $G$ un gruppo algebrico con un endomorfismo

$\psi$ di ordine *m*. Sia $g \in G$ di ordine *r* tale che $\psi(g) = g^{\lambda}$ in modo che $\psi(x) = x^{\lambda}$ per ogni $x \in < g >$. Definiamo le classi di equivalenza $\bar{x} = \{\psi^j(x): 0 \leq j < m\}$. Definiamo una sequenza pseudorandom $x_i = g^{a_i} h^{b_i}$ usando $\hat{x}$ per selezionare un endomorfismo $(1 + \psi^j)$ e poi agendo su $x_i$ con questa mappa. Più precisamente, *j* è una funzione di $\hat{x}$ (per esempio la funzione *S* delle sezioni precedenti) e

$$x_{i+1} = (1 + \psi^j)x_i = x_i \psi^j(x_i) = x_i^{1+\lambda^j}$$

Si può verificare che la mappa sia ben definita sulle classi di equivalenza e che $x_{i+1} = g^{a_{i+1}} h^{b_{i+1}}$, dove $a_{i+1} = (1 + \lambda^j)a_i \bmod r$ e $b_{i+1} = (1 + \lambda^j)b_i \bmod r$.

Insistiamo sul fatto che questo approccio richiede ancora che venga trovato un unico rappresentante per ogni classe di equivalenza allo scopo di definire i passi del cammino in un modo ben definito. Quindi, si possono di nuovo usare i punti distinti definendo che una classe è distinta se il suo rappresentante è distinto.

Un inconveniente dell'idea di Gallant, Lambert e Vanstone è che c'è meno flessibilità nel design del cammino pseudo casuale.

### 2.4.3 Esperienza pratica con l'Algoritmo Rho Distribuito

I calcoli reali non sono così semplici come nell'analisi ideale appena esposta: non si può sapere in anticipo quanti client si avranno a disposizione per i calcoli; non tutti i client hanno le stesse prestazioni e affidabilità; i client potrebbero decidere di ritirarsi dall'esecuzione dei calcoli in qualsiasi momento; la comunicazione tra client e server potrebbe essere inaffidabile, eccetera. Dunque, nella pratica c'è la necessità di scegliere i punti distinti in modo che siano abbastanza comuni da permettere anche al client più debole possa colpirne uno durante l'esecuzione dei calcoli in tempi ragionevoli (magari uno o due giorni). Ciò significa che i client più efficienti trovano molti punti distinti ogni ora.

I più grandi problemi di logaritmo discreto risolto usando il metodo di rho distribuito sono i *Certicon challenge elliptic curve discrete logarithm problems*. I record correnti sono per i gruppi $E(\mathbb{F}_p)$ con $p \approx 2^{109}$ (da una squadra coordinata da Chris Monico nel 2002) e per $E(\mathbb{F}_{2^{109}})$ (di nuovo dalla squadra di Monico nel 2004). In entrambi i casi i calcoli hanno utilizzato la classe di equivalenza $\{P, -P\}$ che ha quasi sempre dimensione 2.

Riassumiamo brevemente i parametri utilizzati per questi grossi calcoli. Per il risultato del 2002 la curva $E(\mathbb{F}_p)$ aveva ordine primo è quindi $r \approx 2^{109}$. Il numero di processori era di oltre 10000, $\theta = 2^{-29}$. Il numero di punti distinti trovati fu di $68228567$ che è circa 1,61 volte il numero atteso di punti da essere collezionati, cioè $\theta\sqrt{\pi 2^{109}/4}$. Dunque, questa computazione è stata piuttosto sfortunata poiché è stata in esecuzione significativamente di più (1,61 volte) di quanto ci si aspettasse. L'intero calcolo richiese circa 18 mesi.

Il risultato del 2004 è dovuto ad una curva con co-fattore 2, e dunque $r \approx 2^{108}$. Il numero di processori era di 2000, $\theta = 2^{-30}$. Il numero di punti distinti trovati fu di $16531676$ che è circa 1,11 volte il numero atteso di punti da essere collezionati, cioè $\theta\sqrt{\pi 2^{108}/4}$. L'intero calcolo richiese circa 17 mesi.

I risultati del tempo reale di esecuzione e delle stime teoriche sono molto vicini. Questo è evidenza del fatto che l'analisi euristica del tempo di esecuzione non è molto lontana dalle prestazioni pratiche. Infine, menzioniamo alcuni recenti lavori sull'algoritmo rho di Pollard. Cheon, Hong e Kim [29] hanno accelerato Pollard rho in $\mathbb{F}_p^*$ usando una strategia "look ahead"; essenzialmente essi determinano in quale partizione si troverà il prossimo valore del cammino, senza eseguire un intera operazione di gruppo. Un'idea simile per le curve ellittiche è stata usata da Bos, Kaihara e Kleinjung [30].

# Capitolo 3

# RISOLUZIONE DEL PROBLEMA DELLA FATTORIZZAZIONE CON IL METODO RHO DI POLLARD

*"I cannot conceive that anybody will require multiplications*
*at the rate of 40,000, or even 4,000 per hour;*
*such a revolutionary change as the octonary scale*
*should not be imposed upon mankind in general*
*for the sake of a few individuals"*
F. H. Wales

## 3.1 Metodo Rho di Pollard seriale

### 3.1.1 L'idea e la teoria

Per la presentazione di questo metodo prendo spunto da [2].

Supponiamo che $p$ sia primo e che S = {0, 1, …, $p$ – 1}. Scegliamo una funzione $f$ da S in S, per esempio $f(x) = x^2 + 1 \bmod p$. Poiché S è un insieme finito, se inizio a calcolare $f^{(1)}(s)$, $f^{(2)}(s)$, … a partire da un elemento $s$ di S avremo prima o poi che $f^{(j)} = f^{(k)}$ per il semplice motivo che S è un insieme finito. Inoltre, se la $f$ è una funzione "abbastanza random" possiamo stimare che si giungerà ad una ripetizione nella sequenza $f^{(i)}$ con $i$ = 1, 2, …, in $O(\sqrt{p})$ passi; questo risultato è già stato discusso nel secondo capitolo.

Supponiamo ora di voler fattorizzare $N$ e che $p$ sia il suo più piccolo fattore primo. Poiché non conosciamo ancora $p$, non possiamo calcolare $f(s)$. Però possiamo calcolare i valori dati dalla funzione $F(x) = x^2 + 1 \bmod N$. Questa funzione è tale che $f(x) = F(x) \bmod p$ e dunque $F^{(j)}(s) \equiv F^{(k)}(s) \bmod p$.

Quindi per trovare $p$ mi basterà calcolare i valori di $F^{(i)}(s)$ partendo da un $s$ casuale e calcolare per ogni coppia il MCD($F^{(j)}(x) - F^{(k)}(x)$, $N$). Quando troviamo la coppia tale che $F^{(j)}(s) \equiv F^{(k)}(s) \bmod p$ allora questo massimo comune divisore sarà un numero divisibile per $p$, e se viene un numero diverso da $N$ vorrà dire che avremo trovato un suo fattore.

Notiamo subito che il procedimento appena illustrato non è affatto conveniente, se infatti dovessimo calcolare il MCD($F^{(j)}(x) - F^{(k)}(x), N$) per tutte le coppie $j$, $k$ con $0 \leq j < k$ fino a circa $\sqrt{p}$, allora avremmo ½ $p$ coppie!

La soluzione è data dal metodo di Floyd per trovare la ciclicità di una funzione random su insieme finito:

### 3.1.2 Floyd cycle-finding method

Sia $l = k - j$ , cioè per ogni $m \geq j$, $F^{(m)}(s) \equiv F^{(m+l)}(s) \equiv F^{(m+2l)}(s) \equiv \ldots \bmod p$.

Ora prendiamo $m$ come il primo multiplo di $l$ che supera $j$, cioè $m = l \cdot \left\lceil \frac{j}{l} \right\rceil$.

Allora $F^{(m)}(s) \equiv F^{(2m)}(s) \bmod p$ e $m \leq k = O(\sqrt{p})$.

Dunque l'idea di base del metodo Rho di Pollard è calcolare la sequenza MCD($F^{(j)}(x) - F^{(2j)}(x), N$) con $j = 1, 2, \ldots$. Questa dovrebbe terminare con un fattore non banale di $N$ in $O(\sqrt{p})$ passi, dove $p$ è il fattore primo più piccolo di $N$.

### 3.1.3 L'Algoritmo

*Pseudocodice*

1. Impostazione dei valori iniziali
   a) Scegliere il tipo di funzione mod $N$ più conveniente (nel caso della funzione $x^2 + a$ basta scegliere un valore casuale di $a$ compreso tra 1 e $N - 3$)
   b) Scegliere un valore casuale $s$ da cui partire compreso tra 0 e $N - 1$
   c) Porre i valori temporanei di $U$ e $V = s$
2. Ricerca dei fattori
   a) Porre
      $U = F(U)$ e $V = F(F(V))$
   b) Continuare fino a che $g = \text{MCD}(U - V, N) = 1$
   c) Se $g = N$ ritornare al passo 1 e cambiare $a$ (o funzione)
   d) Se $g \neq 1$ e da $N$ allora stampa $g$.

### 3.1.4 Osservazioni

Prima di fare osservazioni specifiche notiamo che il metodo di Pollard richiede una piccola quantità di memoria; per ogni ciclo i valori salvati sono quelli temporanei di $U$ e $V$ e quello di $N$.

*Osservazione 1. Riduzione dei calcoli*.

I passi 2.a e 2.b richiedono 3 moltiplicazioni modulari (dei quadrati se la funzione è del tipo $x^2 + a$) ed il calcolo del MCD. In realtà, al costo di una moltiplicazione modulare in più, si può effettuare il calcolo del MCD più raramente. Infatti i numeri $U - V$ possono essere accumulati insieme (cioè moltiplicati) modulo $N$ per $k$ iterazioni, e il MCD basterà cercarlo tra questo prodotto ed $N$. Se per esempio $k = 100$, allora il costo del calcolo del MCD diventa trascurabile ed il costo di un generico ciclo diventa quello di 3 quadrati modulari e di una moltiplicazione modulare.

In questo modo ovviamente è possibile che il MCD, dopo 100 cicli, sia diventato $N$ stesso e per ovviare questo inconveniente sarà necessario memorizzare i valori di $U$ e $V$ all'inizio del ciclo, per poi eventualmente ripartire da essi con calcoli più frequenti del MCD.

*Osservazione 2. La scelta della funzione.*

Ci sono molte scelte per la funzione $F$. Il criterio chiave è che le iterate di $F$ mod $p$ non devono avere delle lunghe ρ, o come vengono chiamate in [2], dei lunghi "epacts" di $p$ rispetto a $F$, dove per epact si intende il più grande $k$ per cui gli elementi della sequenza $F^{(2)}(s)$, $F^{(2)}(s)$, …, $F^{(k)}(s)$ sono tutti distinti.

Una scelta sconveniente di $F$ è del tipo $F(x) = ax + b$, poiché l'epact di un primo $p$ in questo caso è l'ordine moltiplicativo di $a$ (quando $a \not\equiv 1 \bmod p$), di solito un grande divisore di $p - 1$ (quando $a \equiv 1 \bmod p$ e $b \not\equiv 0 \bmod p$ l'epact è $p$).

Anche tra le funzioni quadratiche ci possono essere scelte sconvenienti, per esempio se $a = 0$. Un'altra scelta meno evidente, ma sempre sconveniente, è $x^2 - 2$. Se $x$ può essere rappresentata come $y + y^{-1} \bmod p$, allora la $k$_esima iterata è $y^{2^k} + y^{-2^k} \bmod p$.

Non si sa se l'epact di $x^2 + 1$ per $p$ sia una funzione di $p$ abbastanza lenta a crescere, ma Guy ha congetturato che sia $O(\sqrt{p \ln p})$.

*Osservazione 3. Fattorizzazione di* $F_8 = 2^{2^8} + 1$

Se invece si conoscono delle informazioni sul fattore primo $p$ di $N$ cercato, potrebbe essere utile usare polinomi di grado superiore. Per esempio poiché tutti i fattori primi del numero di Fermat $F_k = 2^{2^k} + 1$ sono congruenti a 1 mod $2^{k+2}$ quando $k \geq 2$ (Eulero dimostrò nel 1732 che ogni eventuale divisore di $F_k$ è del tipo $h \cdot 2^{k+2} + 1$, vedere anche teorema 1.3.5 in [2]) allora, per fattorizzare questo tipo di numeri, si potrebbe utilizzare la funzione $x^{2^{k+2}} + 1$. Ci si può aspettare che l'epact di un fattore primo $p$ di $F_k$ rispetto a $x^{2^{k+2}} + 1$ sia più piccolo di quello di $x^2 + 1$ di un fattore di circa $\sqrt{2^{k+1}}$. A conferma di ciò si può leggere il modello probabilistico dato in [8] .

*Osservazione 4. Metodo di Brent per la ciclicità*

Nel 1980 Richard Brent [5] pubblicò un metodo più veloce rispetto a quello di Floyd per trovare la ciclicità nella successione $f(x_i)$. Il risparmio è dato dal non dover calcolare due volte per ogni ciclo (nel punto 2a) il valore di $f$. L'idea è che un puntatore punti alla posizione $2^i - 1$ (0, 1, 3, 7, 15, …) mentre l'altro scorre fino a $2^{i+1} - 1$ ($i + 1$, $i + 2$, …, $2i + 1$), accumulando in un prodotto $q$ mod $N$ i fattori $(x_{2^i} - x_j)$ (con $j$ che varia tra $2^i + 1$ e $2^{i+1}$) e verificando alla fine del ciclo se MCD$(x_{2^i}, q) \neq 1$. Se MCD = 1 allora il primo puntatore passerà a $2^{i+1}$ e il secondo scorrerà fino a $2^{i+2}$ e così via. Dall' analisi di Brent e Knut [6] si vede che questo accorgimento velocizza la ricerca di un fattore solo del 4% essendo il numero di moltiplicazioni mod $N$ e di valutazioni di $f$ con il metodo di Floyd circa $4{,}1232p^{1/2}$, mentre con il metodo di Brent $3{,}9655p^{1/2}$ . Ciò che rende veramente migliore l'algoritmo è l'omettere i termini $(x_{2^i} - x_j)$ con $j < \frac{3}{2} \cdot 2^i$ nel prodotto $q$. Infatti un fattore non banale di $q$ contenuto in questi termini dev'essere necessariamente contenuto nei termini con $\frac{3}{2} \cdot 2^i \leq j < 2 \cdot 2^i$. In questo modo il numero di moltiplicazioni mod $N$ e di valutazioni di $f$ diventa circa $3.1225p^{1/2}$, migliorando la velocità dell'algoritmo del 24% circa.

Prima di parlare della parallelizzazione del metodo Rho analizziamolo da due diversi punti di vista prendendo spunto da [10]:

### 3.1.5 Analisi del metodo dal punto di vista statistico

Pensiamo alla sequenza $< x_i : i = 1, \ldots, n >$ come sequenza casuale modulo un fattore nascosto $p$ di $N$. Quanti elementi di questa sequenza dovremo pescare prima di trovare una ripetizione nella sequenza (e quindi trovare un fattore di $N$).

Dato un insieme di $P$ elementi, la probabilità che in una sequenza casuale di essi vi sia una "collisione" è data dalla seguente formula:

$$\left(1-\frac{1}{P}\right)\left(1-\frac{2}{P}\right)\ldots\left(1-\frac{n-1}{P}\right)\frac{n}{P}$$

che possiamo approssimare come una particolare funzione densità di Poisson:

$$f(n)=\frac{n}{P}e^{-\frac{n^2}{2P}}$$

Il cui integrale per $n \in (0, \infty)$ è proprio l'unità. Dunque il numero di iterate atteso per una collisione è:

$$< n >= \int_0^{\infty} nf(n)dn = \sqrt{\frac{\pi P}{2}}$$

Risultati euristici confermano che il numero di iterazioni necessarie per scoprire $p$ con il metodo Rho di Pollard sono $< n > \sim c\sqrt{p}$

### 3.1.6 Analisi del metodo dal punto di vista algebrico

Osserviamo che la catena di fattori che si accumula alla $k$-esima iterata è:

$$\begin{aligned} x_{2k}-x_k &= x_{2k-1}^2+a-x_{k-1}^2-a=\cdots \\ &= (x_{2k-1}-x_{k-1})(x_{2k-2}-x_{k-2})\ldots(x_k-x_0)(x_k+x_0) \end{aligned}$$

Quindi la differenza $x_{2k}-x_k$ contiene circa $k$ fattori algebrici (il "circa" è dovuto alle possibili ripetizioni). Ciò significa che se facciamo scorrere l'indice $k$ da 1 ad $n$ accumuliamo circa $\frac{n^2}{2}$ fattori algebrici. È ragionevole (vedere [1]) supporre che $O(n^2)$ fattori casuali possano contenere $p$ quando $n$ è $O(\sqrt{p})$

### 3.1.7 Analisi del caso in cui il fattore cercato $p \equiv 1 \bmod 2K$

Dal punto di vista statistico l'iterazione $x \coloneqq x^{2K} + a \bmod p$ scorre su un insieme ristretto, poiché per la classe $p$ in questione l'insieme delle potenze $2K$-esime ha cardinalità $P = (P - 1)/2K$. dunque possiamo aspettarci una riduzione di operazioni di circa $1/\sqrt{K}$ (in realtà Brent e Pollard in [8] hanno stimato che l'esatto fattore di riduzione sia $1/\sqrt{2K-1}$ ).

Dal punto di vista algebrico invece abbiamo che

$$x_{2k} - x_k = x_{2k-1}^{2K} - x_{k-1}^{2K}$$

E possiamo aspettarci che il membro destro dell'equazione abbia una maggiore probabilità di contenere il fattore $p$ poiché nel campo $F_p[X,Y]$ il binomio $X^{2K} - Y^{2K}$ ha i fattori $X - g^m Y$, dove $g$ è una radice $2K$-esima dell'unità.

Dunque da entrambi i punti di vista la riduzione euristica di tempo totale è $O(\log_2 K/\sqrt{K})$, con il fattore del logaritmo dovuto ai calcoli per elevare alla $2K$ anziché solo al quadrato.

Casualmente, anche se non si hanno informazioni sul fattore nascosto $p$, usando l'iterazione $x \coloneqq x^{2K} + a$ si può dedurre [8] da entrambi i punti di vista che ci si può aspettare che il fattore di riduzione del numero di operazioni nell'anello sia:

$$1/\sqrt{MCD(p-1,2K)-1}$$

## 3.2 Parallelizzazione del metodo Rho di Pollard

### 3.2.1 Macchine indipendenti

Mostriamo ora che l'uso di *m* macchine in parallelo che lavorano indipendentemente (cioè con polinomi e valori iniziali diversi) riduce il tempo di fattorizzazione di solo $\sqrt{m}$.

Denotiamo con $x_i^{(k)}$ l'*i*-esima iterata della *k*-esima macchina di *m* macchine. Assumiamo quindi che la *k*-esima macchina usi l'iterazione $x := x^2 + a_k$ con gli $a_k$ scelti indipendentemente (escludendo 0 e – 2) e che ognuna sia inizializzata con un indipendente $x_0^{(k)}$. La probabilità che nessuna delle macchine abbia, all'*i*-esima iterata, una collisione $x_i^{(k)} = x_j^{(k)}$ per ogni $j < i$ è:

$$\left(1 - \frac{i}{P}\right)^m$$

E quindi arriviamo ad una densità di probabilità approssimata a

$$f(n) = \frac{2mn}{P} e^{-\frac{mn^2}{2p}}$$

che rappresenta la prima collisione all'iterata *n*. Ciò conduce immediatamente a

$$< n > \sim \sqrt{\frac{\pi P}{2m}}$$

In cui appare il fattore di riduzione $1/\sqrt{m}$.

Usando il punto di vista algebrico possiamo arrivare alla stessa deduzione. Come visto poco fa, ogni macchina generava $O(n^2/2)$ fattori algebrici e quindi ora tutte le macchine ne avranno generati $O(mn^2/2)$ e di conseguenza ci vorranno $n \sim \sqrt{p/m}$ passi per scoprire *p*.

### 3.2.2 Macchine con lo stesso parametro *a*

Dunque consideriamo un modello diverso.

Prima di tutto imponiamo alle *m* macchine di avere lo stesso parametro *a*, ma diverso valore iniziale $x_0^{(k)}$. In questo modo possiamo considerare il cosiddetto "prodotto di correlazione"

$$Q = \prod_{i=1}^{n} \prod_{k=0}^{m-1} \prod_{j=0}^{m-1} (x_{2i}^{(k)} - x_i^{(k)})$$

Questo prodotto contiene $O(m^2\ n^2)$ fattori algebrici e quindi le *n* iterate necessarie a scoprire *p* dovrebbero essere $O\left(\frac{\sqrt{p}}{m}\right)$. Dal punto di vista statistico avremmo potuto ottenere la stessa stima poiché la densità di probabilità per la prima collisione diventa

$$f(n) = \frac{2m^2 n}{P} e^{-\frac{m^2 n^2}{2p}}$$

La riduzione di 1/*m* dell'*n* atteso è buona, ma ora si pone il problema di valutare il prodotto *Q*, che in appropriato senso asintotico, richiede un po' più di tempo rispetto ai calcoli del metodo base Rho attraverso l'indice $2n$. In particolare, scegliamo per convenienza *m* che divide *n*, e dunque la macchina μ di *m* macchine può calcolare il prodotto di correlazione "parziale"

$$Q_\mu = \prod_{i=\mu n/m}^{(\mu+1)n/m-1} \prod_{k=0}^{m-1} \prod_{j=0}^{m-1} (x_{2i}^{(k)} - x_i^{(k)})$$

in sole *n*/*m* applicazioni di qualche algoritmo veloce per la valutazione di un polinomio. Infatti, la macchina μ valuterà il polinomio

$$q_i(x) = \prod_{k=0}^{m-1} (x_{2i}^{(k)} - x)$$

ai punti $x \in \left\{x_i^{(j)} : j = 0, \dots, m-1\right\}$ in $O(m \log^2 m)$ operazioni modulari (si veda[2]). Dunque il numero totale di operazioni modulari alla μ-esima macchina sarà:

$$O\left(\frac{n}{m} m \log^2 m\right) = O(n \log^2 m)$$

Questa idea basilare unita all'osservazione che tutti i calcoli, come l'accumulo nel prodotto $Q_\mu$, sono $O(n)$ operazioni modulari conduce alla seguente stima di tempo totale per trovare $p$

$$O\left(\tau_N \frac{\sqrt{p}\log^2 m}{m}\right)$$

Dove $\tau_N$ è il tempo necessario per un'operazione modulare in $\mathbb{Z}_N$.

E dunque abbiamo ottenuto una riduzione effettiva dei calcoli dell'ordine $O(\log^2 m/m)$.

Anche in questo nuovo modello possiamo utilizzare l'iterazione $x \coloneqq x^{2K} + a \; mod \; N$ (supponendo di sapere che un divisore primo di $N$ sia 1 mod $2K$ o che vi sia un sufficientemente grande MCD($p$ – 1, $2K$)). Ciò significa che ogni macchina utilizzerà l'iterazione con il grado $2K$ con $a$ sempre fissato e $x_0^{(k)}$ casuale come prima. Con tutte queste considerazioni, il tempo totale per scoprire un fattore di $N$ dovrebbe essere:

$$O\left(\tau_N \frac{\sqrt{p}}{m} \frac{1 + \log K + \log^2 m}{\sqrt{MCD(p-1,2K) - 1}}\right)$$

### 3.2.3 Problemi riguardanti il nuovo modello e prospettive di ricerca

L'assunzione del parametro *a* fissato conduce ad alcuni problemi. Mentre avere un *a* diverso porta ad avere una varietà sufficiente di cicli iterativi, fissare questo parametro può portare ad averne un numero davvero basso. Per esempio con $p = 257$ e $a = -1$ ci sono solo tre diversi periodi, di lunghezza 12, 7 o 2.

*1. Ottimizzazione dell'uso della memoria*

L'algoritmo accennato richiede un'estensiva condivisione di dati.

Non è ancora stato determinato se un ordine diverso di alcuni passi dell'algoritmo possa far risparmiare memoria. Per esempio, quando la macchina $\mu = 0$ fa le sue $n/m$ valutazioni polinomiali, ognuna di grado $m$, in $m$ punti, forse questo sforzo può essere condiviso con tutte le altre macchine, ed effettuare un'operazione di accumulazione (come un MCD), in modo che gli indici $i \in [0, n/m - 1]$, una volta processati, possano essere ignorati.

*2. Scelta dei valori K*

Non si sa ancora se usando diversi valori di *K* per macchina possa portare dei vantaggi. Sarebbe interessante analizzare la questione sia dal punto di vista statistico che algebrico.

# Capitolo 4

## FATTORIZZAZIONE ATTRAVERSO LA RISOLUZIONE DEL LOGARITMO DISCRETO

*"By definition, when you are investigating the unknown, you do not know what you will find"*

In questo capitolo vogliamo occuparci della risoluzione del problema della fattorizzazione tramite la risoluzione del logaritmo discreto con il metodo Rho di Pollard. Sfrutteremo, il teorema del Capitolo 1, il quale afferma che se si può risolvere il logaritmo discreto in tempo polinomiale, allora si può fare lo stesso per la fattorizzazione. L'algoritmo è il seguente

INPUT: $n$
OUTPUT: fattore $p$ di $n$
1: inizializzo $x = 2, m = 1, p = 1$
2: While $((n \equiv 0 \bmod x)$ e $(p = 1)$ o $(p = n))$
3: $m =$ trovaEsponente$(x, n)$
4: $k = 0$
5: While $(x^{\frac{m}{2^k}} \equiv 1 \bmod n$ e $x^{\frac{m}{2^{k+1}}} \equiv \pm 1 \bmod n)$
6: $k = k + 1$
7: End While
8: $p = MCD(x^{\frac{m}{2^{k+1}}} + 1, n)$
9: $x = x + 1 \bmod n$
10: End While
11: Stampa $p, x$

## 4.1 Esempi

Vediamo un paio di esempi molto semplici per capire il meccanismo.

**Esempio.**

Scegliamo $n = 5 \cdot 7 = 35$

Poniamo $x = 2$

$m = 12$, poiché $2^{12} \equiv 1 \bmod 35$

$k = 0$,

$$2^{\frac{12}{2^0}} \equiv 1 \bmod n$$

$$2^{\frac{12}{2^1}} = 2^6 \equiv 29 \bmod n \not\equiv \pm 1 \bmod n$$

Per cui posso già calcolare $p = MCD\left(2^{\frac{12}{2}} + 1{,}35\right) = MCD(30{,}35) = 5$

**Esempio.**

```
n = 79099 = 83·953
x = 2
m = 5576
k = 0
x^(m/2^k) mod n = 1
x^(m/2^k+1) mod n = 1
k = 1
x^(m/2^k) mod n = 1
x^(m/2^k+1) mod n = 51461
p = MCD(x^(m/2^k+1) mod n + 1, n)
  = MCD(51461, 79099) = 953
```

Come già notato nel primo capitolo, per ogni *x*, l'algoritmo ha probabilità ≥ ½ di avere successo, e se escludiamo il punto 3 dell'algoritmo, cioè la funzione *trovaEsponente*(*x, n*), possiamo notare che il risultato viene raggiunto in pochissimi passaggi.

Passiamo dunque ad analizzare la funzione *trovaEsponente*(*x, n* ) ed il suo algoritmo

INPUT: $x,n$

OUTPUT: esponente di $x$, cioè un numero $e$ tale che $x^e \equiv 1 \bmod n$

1: Inizializzo $q = 1, p = 2$
2: For $i = 1..$numero di cifre di $n$ in base 2
3: If $p$ è un numero primo
4: $q = \log_{x^p} x$
5: If $x^{pq} \equiv x \bmod n$
6: return $pq - 1$
7: End If
8: End If
9: $p = p + 1$
10: End For
11: return $0$

Vediamo l'algoritmo subito in azione in un altro semplice esempio.

**Esempio.**

```
n = 133 = 7·19

Cerco l'esponente di x = 2 mod 133:

p = 2 che e' primo
q = log di 2 in base 2^2 = non esiste

p = 3 che e' primo
q = log di 2 in base 2^3 = non esiste

p = 4 che NON e' primo

p = 5 che e' primo
q = log di 2 in base 2^5 = 11
x^pq mod n = 2^5^11 mod 133 = 2
m = pq-1 = 54

passo alla ricerca del fattore
x = 2
k = 0
x^(m/2^k) mod n = 1
x^(m/2^k+1) mod n = 113
p = MCD(x^(m/2^k+1) mod n + 1, n) = MCD(114, 133) = 19
```

Notiamo che per un calcolo così piccolo sarebbe stata conveniente una fattorizzazione con la forza bruta che ci avrebbe portato alla scoperta del fattore 7 nel giro di poche divisioni. Più crescono i fattori del numero da fattorizzare e più questo metodo conviene rispetto alla forza bruta: il miglioramento lo si può già notare nel seguente (qui, con la forza bruta sarebbero previste al massimo $\sqrt{79099} = 281$ divisioni):

**Esempio.**

```
n = 79099 = 83·953

Cerco l'esponente di x = 2 mod 79099:

p = 2 che e' primo
q = log di 2 in base 2^2 = non esiste

p = 3 che e' primo
q = log di 2 in base 2^3 = 1859
x^pq mod n = 2^3^1859 mod 79099 = 2

m = pq-1 = 5576

passo alla ricerca del fattore:

x = 2
k = 0
x^(m/2^k) mod n = 1
x^(m/2^k+1) mod n = 1
k = 1
x^(m/2^k) mod n = 1
x^(m/2^k+1) mod n = 51461
p = MCD(x^(m/2^k+1) mod n + 1, n) = MCD(51462, 79099) = 953
```

Ricordiamo che il primo $p$ tale che $x^{pq} \equiv x \bmod n$ lo si troverà nei primi $\log N + 1$ primi (parte finale della dimostrazione del capitolo 1), quindi in un tempo polinomiale.

Il problema ora sta nel calcolo del logaritmo discreto, cioè $\log_{x^p} x$, che dev'essere svolto nell'anello $\frac{\mathbb{Z}}{n\mathbb{Z}}$. Questo è il punto critico dell'intero algoritmo di fattorizzazione, nel senso che questo è il punto in cui i calcoli diventano esponenziali (o sub-esponenziali a seconda del metodo usato), non esistendo un algoritmo di tempo polinomiale nelle cifre di *n* che sappia risolvere il problema del logaritmo discreto. Per i semplici esempi illustrati è stato usato il metodo di forza bruta, e successivamente si è provato a risolvere i logaritmi incontrati con il metodo Rho di Pollard, nella versione esposta da Gilbraith in [12].

## 4.2 Risoluzione dei Logaritmi incontrati con il metodo Rho di Pollard: problemi e difficoltà.

Si è tratta di risolvere logaritmi del tipo $\log_{x^p} x$ in $\mathbb{Z}/n\mathbb{Z}$ dove *n* è il numero che deve essere fattorizzato e di cui quindi dobbiamo supporre di non conoscere la fattorizzazione. Questa supposizione è una delle prime difficoltà, poiché spesso per risolvere un logaritmo discreto modulo *n* con il metodo di Pollard, ci si riconduce a risolvere vari logaritmi discreti modulo *p*, dove *p* è un fattore di *n*, e poi si giunge ad una soluzione tramite il teorema cinese dei resti come per esempio dimostrato in [11].

*Problema 1: il calcolo dell ordine di g.*

Il vantaggio del risolvere problemi modulo *p* è che non c'è bisogno di calcolare l'ordine $r$ di $g \in \mathbb{Z}/p\mathbb{Z}$, in quanto possiamo sostituirlo con il suo multiplo $p - 1$ (teorema di Fermat).

Quando invece svolgiamo i calcoli in $\mathbb{Z}/n\mathbb{Z}$, come nel nostro caso, sappiamo che l'ordine *r* di $g \in \mathbb{Z}/n\mathbb{Z}$ è un divisore di $\phi(n)$, valore facilmente calcolabile solo se, di nuovo, si conosce la fattorizzazione di *n*. Conoscere l'ordine $r$ è quindi un problema considerato difficile, che al momento sembra non si possa aggirare poiché la conoscenza dell'ordine di $g$ è fondamentale per il calcolo degli $a_i$ e $b_i$ e per

l'inversione (al punto 11 dell'algoritmo di Gilbraith per il logaritmo discreto) che avviene proprio in $\mathbb{Z}/r\mathbb{Z}$.

*Problema 2: l'inversione* $(b_j - b_i)^{-1} \bmod r$.

Il fallimento di questo calcolo è a detta di molti autori raro, ma nei piccoli casi incontrati è risultato un problema, in quanto l'inverso è spesso inesistente. Vediamo l'esempio in cui $n$ = 133. Ad un certo punto bisogna risolvere il $\log_{2^5} 2$. In questo caso l'ordine di $g = 18$. Le possibili combinazioni per la funzione $S$, per $n_S = 3$, sono date da i valori $u_1, v_1, u_2, v_2$, che possono assumere valori compresi tra 0 e $r = 18$; quindi sono in tutto $18^4 = 104976$. Con un apposito algoritmo sono state calcolate le coppie $(b_j, b_i)$ tali per cui $(b_j - b_i)$ è invertibile modulo $r$, che sono 17064, cioè circa il 16% delle coppie totali raggiunte dall'algoritmo.

*Problema 3: calcoli a vuoto del logaritmo.*

Molto spesso inoltre succede che la soluzione di $\log_{x^p} x$ non esista e ciò comporta un ciclo a vuoto molto dispendioso, dell'ordine di $\sqrt{n} \cdot ord^{n_S+1}$, cioè il tempo necessario a trovare una collisione moltiplicato il numero di possibili funzioni $S$ da testare. Sarebbe utile trovare un metodo veloce che a priori stabilisca se il logaritmo discreto sopra citato sia risolvibile oppure no, cioè se $x \in < x^p > \bmod n$, problema che di solito non si pone nell'esposizione del metodo Rho di Pollard per i Logaritmi Discreti poiché si suppone che $h \in < g >$, fatto non ovvio nel nostro caso.

# APPENDICE – ALGORITMI JAVA

*"I really hate this damn machine, I wish that they would sell it.*
*It never does just what I want, But only what I tell it"*

## A.1 Per la Fattorizzazione

### A.1.1 Pollard Rho Classico

```
import java.util.Scanner;
import java.math.BigInteger;
/*mandando in esecuzione la classe, viene chiesto di inserire il
numero da fattorizzare n, la base a e il limite l fino a cui
eseguire la ricerca.
Stampa il primo fattore trovato (non necessariamente un numero
primo) utilizzando il metodo Rho di Pollar*/
public final class PollardRhoBI {
//restituisce il valore della funzione x^2 + a
public static BigInteger Funzione(BigInteger x,
BigInteger a,BigInteger n){
            x = x.pow(2).add(a).remainder(n);
            return x;
}
//metodo che ritorna il primo fattore trovato di num usando
//il metodo Rho di pollard
//se non è stato trovato alcun fattore allora ritorna num
public  static  BigInteger  calcolaPrimoFattorePollardRho(BigInteger
n,BigInteger a, BigInteger s){
            BigInteger u;
            BigInteger v;
            BigInteger g = new BigInteger("1");
            u = s;
            v = s;
            while (g.compareTo(BigInteger.ONE) == 0){
                  u = Funzione(u,a,n);
                  v = Funzione(Funzione(v,a,n),a,n);
                  g = n.gcd(u.subtract(v).abs());
            }
            return g;
}
```

```
public static void main(String[] args) {
            Scanner input = new Scanner(System.in);
            System.out.println("Inserisci il numero N
                                    da fattorizzare: ");
            BigInteger n = new BigInteger(input.nextLine());
            System.out.println("Inserisci il parametro a: ");
            BigInteger a = new BigInteger(input.nextLine());
            System.out.println("Inserisci un valore s
                              da cui cominciare la ricerca: ");
            BigInteger s = new BigInteger(input.nextLine());
            BigInteger fattoreTrovato =
                              calcolaPrimoFattorePollardRho(n,a,s);
            System.out.println( "il primo fattore trovato di "+ n +
                                          " e' "+ fattoreTrovato);
            while (n.compareTo(fattoreTrovato)==0){
                  System.out.println("Inserisci un nuovo
                                                parametro a: ");
                  a = new BigInteger(input.nextLine());
                  System.out.println("Inserisci un nuovo valore s
                                    da cui cominciare la ricerca: ");
                  s = new BigInteger(input.nextLine());
                  fattoreTrovato =
                              calcolaPrimoFattorePollardRho(n,a,s);
                  System.out.println( "il primo fattore trovato
                              di "+ n + " e' "+ fattoreTrovato);
            }
      }
}
```

### A.1.2 Pollard Rho con metodo di Brent per la ciclicità

La classe java è uguale alla precedente per quanto riguarda il main che quindi non riportiamo.

Il metodo `calcolaPrimoFattorePollardRho` viene sostituito dal metodo `calcolaPrimoFattorePollardBrentRho`

```
//restituisce il valore della funzione x^1024 + a
public static BigInteger funzione(BigInteger x,
BigInteger a,BigInteger n){
            x = x.pow(1024).add(a).remainder(n);
            return x
}
//metodo che ritorna il primo fattore trovato di num
//usando il metodo Rho di Pollard
//se non è stato trovato alcun fattore allora ritorna num
public static BigInteger calcolaPrimoFattorePollardBrentRho
(BigInteger n,BigInteger a, BigInteger x0){
            BigInteger y ;
            BigInteger x ;
            BigInteger r = new BigInteger("1");
            BigInteger q = new BigInteger("1");
            //accumula i fattori |x-y| mod n
            BigInteger k = new BigInteger("0");
            BigInteger m = new BigInteger("1");
            BigInteger i; //contatore
            BigInteger min;
            BigInteger ys; // salva il primo valore di y
            BigInteger MCD; // contiente il MCD

            y = x0;
            do{
                  x = y;
                  //for i = 1..r
                  for(i=BigInteger.ONE;i.compareTo(r)!=1;
                                    i=i.add(BigInteger.ONE)){
                        y = funzione(y,a,n);
                        k = BigInteger.ZERO;
                  }
```

```
                do{
                        ys = y;
                        min = m.min(r.subtract(k));
                        //for i = 1..min(m,r-k)
                        for (i = BigInteger.ONE;i.compareTo(min)!=1;
                                        i=i.add(BigInteger.ONE)){
                                y = funzione(y,a,n);
                                //q = q*|x-y| mod n
                                q =
                                q.multiply(x.subtract(y).abs()).mod(n);
                        }
                        MCD = q.gcd(n);
                        k = k.add(m);
                }while ((k.compareTo(r) == -1)&&
                                (MCD.compareTo(BigInteger.ONE) == 0));
                r = r.add(r); // r = r*2
        }while (MCD.compareTo(BigInteger.ONE) == 0);
        if (MCD.compareTo(n) == 0){
                do{
                        ys = funzione(ys,a,n);
                        MCD = n.gcd(x.subtract(ys).abs());
                        //MCD = MCD(|x-ys|,n)
                }while (MCD.compareTo(BigInteger.ONE)==0);
        }
        return MCD;
}
```

## A.2 Per il Logaritmo Discreto

### A.2.1 Risoluzione Brute Force

```
import java.util.Scanner;
import java.math.BigInteger;
/*risolve il problema del logaritmo discreto, dati g, h ed n trova
l'esponente a tale che g^a=h mod n
INPUT:g, h, n    OUTPUT:a */
public final class logaritmoDiscretoBruteForce {
      public static int trovaLogaritmoDiscreto(BigInteger g,
                                    BigInteger h,BigInteger n){
      //metodo che risolve il logaritmo discreto;e t.c. g^e=h mod n
            int e = 0;
            BigInteger temp = new BigInteger("1");
            //mentre g^e mod n != h allora e = e+1
            while (temp.mod(n).compareTo(h) != 0 &&
                                          (e<n.intValue())) {
                  temp = temp.multiply(g).mod(n);
                  e=e+1;
            }
            return e;
      }
      public static void main(String[] args) {
            Scanner input = new Scanner(System.in);
            System.out.println("Inserisci la base g: ");
            BigInteger g = new BigInteger(input.nextLine());
            System.out.println("Inserisci h: ");
            BigInteger h = new BigInteger(input.nextLine());
            System.out.println("Inserisci n: ");
            BigInteger n = new BigInteger(input.nextLine());
            int a = trovaLogaritmoDiscreto(g,h,n);
            if (g.pow(a).mod(n).compareTo(h)!=0)
                  System.out.println("Il logaritmo cercato non
                  esiste");
            else
                  System.out.println("Il logaritmo di "+h+"
                              in base "+g+" mod "+n+" e': "+a );
      }
}
```

### A.2.2 Pollard Rho Gilbraith Version

```
import java.util.Scanner;
import java.math.BigInteger;

/*risolve il problema del logaritmo discreto, dati g, h ed n trova
l'esponente a tale che g^a=h mod n
funziona SOLO SE n E'UN NUMERO PRIMO!!!!!!!!
INPUT:g, h, n
OUTPUT:a */

public final class logaritmoDiscretoGilbraithVersionModificato {

	public static BigInteger ordine(BigInteger g,BigInteger p){
		BigInteger e = new BigInteger("1");
		while (g.pow(e.intValue()).mod(p)
			.compareTo(BigInteger.ONE)!=0
			){e=e.add(BigInteger.ONE);}
		return e;
	}

	public static BigInteger fx(BigInteger x,BigInteger[] gg,
			BigInteger p,BigInteger ns,BigInteger ord){
	// metodo che restituisce x^2 se S(x) = 0,
	//altrimenti restituisce x*gg[S(x)]
		if (S(x,ns).compareTo(BigInteger.ZERO)==0)
			return x.multiply(x).mod(p);
		else return x.multiply(gg[S(x,ns).intValue()]).mod(p);
	}

	public static BigInteger fa(BigInteger x, BigInteger a,
			BigInteger[] u,BigInteger p,BigInteger ns,
			BigInteger ord){
	// metodo che restituisce 2a se S(x) = 0,
	//altrimenti restituisce a+u[S(x)]
		if (S(x,ns).compareTo(BigInteger.ZERO)==0)
			return a.add(a).mod(ord);
		else return a.add(u[S(x,ns).intValue()]).mod(ord);
	//se nn ho l'ordine devo fare mod(p.subtract(BigInteger.ONE))
	}
```

```
//in realtà non e' necessario costruire la funzione fb,
//poichè e' uguale ad fa ma con i parametri cambiati,
//la inseriamo per completezza
public static BigInteger fb(BigInteger x, BigInteger b,
      BigInteger[] v,BigInteger p,BigInteger ns,BigInteger ord){
// metodo che restituisce 2a se S(x) = 0,
//altrimenti restituisce a+u[S(x)]
            if (S(x,ns).compareTo(BigInteger.ZERO)==0)
                  return b.add(b).mod(ord);
            else return b.add(v[S(x,ns).intValue()]).mod(ord);
}

public static BigInteger S(BigInteger x, BigInteger ns){
      // restituisce il valore di x modulo ns
            return x.mod(ns);
}

public static void main(String[] args) {
            Scanner input = new Scanner(System.in);
            System.out.print("Inserisci la base --- g = ");
            BigInteger g = new BigInteger(input.nextLine());
            System.out.print("Inserisci --- h =  ");
            BigInteger h = new BigInteger(input.nextLine());
            System.out.print("Inserisci un numero primo p = ");
            // dim anello
            BigInteger p = new BigInteger(input.nextLine());
            System.out.print("Inserisci il numero d partizioni = ");
            BigInteger ns = new BigInteger(input.nextLine());
            BigInteger[] u = new BigInteger[ns.intValue()];
            BigInteger[] v = new BigInteger[ns.intValue()];
            BigInteger[] gg;
            //calcolo l'ordine di g in Zp
            BigInteger ord = ordine(g,p);
            BigInteger x = new BigInteger("0");
            x = g;
            BigInteger a = new BigInteger("1");
            BigInteger b = new BigInteger("0");
            BigInteger xx = new BigInteger("0");
            BigInteger A = new BigInteger("1");
            BigInteger B=new BigInteger("0");
            int finito = 1;
            while(finito==1){
```

```
                //inizializzo u[i], v[i] scelti a random
                //t.c. 0<=u[i],v[i]<ord di g
                //questi parametri rendono f una funzione pseudorandom
                      for (int i=1;i<ns.intValue();i++){
                            System.out.print("u["+i+"] = ");
                            u[i] = new BigInteger(input.nextLine());
                            System.out.print("v["+i+"] = ");
                            v[i] = new BigInteger(input.nextLine());
                      }
                      //inizializzo i valori g[i] = g^u[i] * h^v[i]
                      gg = new BigInteger[ns.intValue()];
                      for (int i=1;i<ns.intValue();i++){
                            gg[i]=g.pow(u[i].intValue())
                            .multiply(h.pow(v[i].intValue())).mod(p);
                      }
                      x = g;
                      a = new BigInteger("1");b = new BigInteger("0");
                      xx = fx(x,gg,p,ns,ord);
                      A = fa(x,a,u,p,ns,ord);B = fb(x,b,v,p,ns,ord);
                      int i = 1;
                      while (x.compareTo(xx)!=0){
                            i++;
                            a = fa(x,a,u,p,ns,ord);b=fb(x,b,v,p,ns,ord);
                            x = fx(x,gg,p,ns,ord);
                            A=fa(xx,A,u,p,ns,ord);B=fb(xx,B,v,p,ns,ord);
                            xx = fx(xx,gg,p,ns,ord);
                            //eseguo questo calcolo 2 volte
                            A=fa(xx,A,u,p,ns,ord);B=fb(xx,B,v,p,ns,ord);
                      }
                      System.out.println("min i tc xi = x2i vale: "+i);
                      //calcolo del logaritmo (a-A)(B-b)^(-1) mod ord
                      if (B.subtract(b).gcd(ord)
                            .compareTo(BigInteger.ONE)==0){
                            System.out.println("log="+a.subtract(A)
                .multiply(B.subtract(b).modInverse(ord)).mod(ord));
                            finito = 0;
                      }else System.out.println("inserisci dei nuovi
                                        valori iniziali:");
                }
          }
    }
```

### A.2.3 Algoritmo per statistiche

I metodi sono uguali al precedente, quindi riportiamo solo il main

```
public static void main(String[] args) {
      Scanner input = new Scanner(System.in);

      System.out.print("Inserisci la base --- g = ");
      BigInteger g = new BigInteger(input.nextLine());
      System.out.print("Inserisci --- h =  ");
      BigInteger h = new BigInteger(input.nextLine());
      System.out.print("Inserisci un numero primo ");
      BigInteger p = new BigInteger(input.nextLine());
      System.out.print("Inserisci il numero di partizioni ns = ");
      BigInteger ns = new BigInteger(input.nextLine());
      BigInteger[] u = new BigInteger[ns.intValue()];
      BigInteger[] v = new BigInteger[ns.intValue()];
      BigInteger[] gg;
      //calcolo l'ordine di g in Zp
      BigInteger ord = p.subtract(BigInteger.ONE);
      ord = ordine(g,p);
      System.out.println("l'ordine di "+g+" in Z"+p+" vale: "+ord);
      BigInteger x = new BigInteger("0");
      x = g;
      BigInteger a = new BigInteger("1");
      BigInteger b = new BigInteger("0");
      BigInteger xx = new BigInteger("0");
      BigInteger A = new BigInteger("1");
      BigInteger B = new BigInteger("0");
      int finito = 1;
      int numCoppieInvertibili = 0;
      //per ns = 3
      for (BigInteger k=new BigInteger("1");
            k.compareTo(ord)!=1;k=k.add(BigInteger.ONE)){
       for (BigInteger kk=new BigInteger("1");
             kk.compareTo(ord)!=1;kk=kk.add(BigInteger.ONE)){
        for (BigInteger kkk=new BigInteger("1");
              kkk.compareTo(ord)!=1;kkk=kkk.add(BigInteger.ONE)){
         for (BigInteger kkkk=new BigInteger("1");
              kkkk.compareTo(ord)!=1;kkkk=kkkk.add(BigInteger.ONE)){
            finito = 1;
```

```
while(finito==1){
//inizializzo u[i],v[i] con possibili combin da 1 a ord
      u[1]=k;
      v[1]=kk;
      u[2]=kkk;
      v[2]=kkkk;
      //inizializzo i valori g[i] = g^u[i] * h^v[i]
      gg = new BigInteger[ns.intValue()];
      for (int i=1;i<ns.intValue();i++){
            gg[i] = g.pow(u[i].intValue())
            .multiply(h.pow(v[i].intValue())).mod(p);
      }
      x = g;
      a = new BigInteger("1");
      b = new BigInteger("0");
      xx = fx(x,gg,p,ns,ord);
      A = fa(x,a,u,p,ns,ord);
      B = fb(x,b,v,p,ns,ord);
      int i = 1;
```

```
while (x.compareTo(xx)!=0){

                        i++;
                        a = fa(x,a,u,p,ns,ord);
                        b = fb(x,b,v,p,ns,ord);
                        x = fx(x,gg,p,ns,ord);
                        A = fa(xx,A,u,p,ns,ord);
                        B = fb(xx,B,v,p,ns,ord);
                        xx = fx(xx,gg,p,ns,ord);
                        A = fa(xx,A,u,p,ns,ord);
                        B = fb(xx,B,v,p,ns,ord);
                        xx = fx(xx,gg,p,ns,ord);
                    }
                    if
(B.subtract(b).gcd(ord).compareTo(BigInteger.ONE)==0)
{numCoppieInvertibili++;}
            finito = 0;
            }//fine while
          }//fine for 4
         }//fine for 3
        }//fine for 2
       }//fine for 1
            System.out.println("num coppie invertibili =
"+numCoppieInvertibili);
            System.out.println("Su un totale di:
"+ord.pow(4)+" coppie possibili");
}
```

## A.3 Fattorizzazione attraverso Logaritmo Discreto Brute Force

```
import java.util.Scanner;
import java.math.BigInteger;
/*trova un fattore di n con il metodo del Logaritmo Discreto
n non deve essere pari e non deve essere una potenza di primi,
queste condizioni vanno testate con altri programmi
la ricerca dell'esponente avviene risolvendo log discreto con BRUTE
FORCE
INPUT:n
OUTPUT:p, fattore di n */
Public final class trovaFattoreConLogDiscr3 {
      public static BigInteger trovaLogaritmoDiscreto(BigInteger g,
                                    BigInteger h,BigInteger n){
      // metodo che risolve il logaritmo discreto,
      //trova e tale che g^e=h mod n
      // usa la BRUTE FORCE
            BigInteger e = new BigInteger("0");
            BigInteger temp = new BigInteger("1");
            //mentre g^e mod n != h allora e = e+1
            while (temp.mod(n).compareTo(h) != 0 &&
                                          (e.compareTo(n)<0) ) {
                  temp = temp.multiply(g).mod(n);
                  e=e.add(BigInteger.ONE);
            }
            return e;
      }
      public static BigInteger trovaEspConLogDiscr(
                                    BigInteger x,BigInteger n){
      // metodo che restituisce m tale che x^m = 1 mod n
      // se ritorna 0 allora non è stato trovato nessun esponente
            BigInteger q = new BigInteger("1");
            BigInteger p = new BigInteger("2");
            BigInteger contaOp = new BigInteger("0");
            System.out.println("Cerco l'esponente di "+x+" mod "+n);
```

```
for (int i=0;i< (n.bitLength()+1) ;i++){
      System.out.print("p = "+p);
      if (p.isProbablePrime(100)) {
            System.out.println(" che e' primo");
            q =
            trovaLogaritmoDiscreto(x.pow(p.intValue()),
            x,n);
            //se x^p^q mod n = x mod n
            if (x.pow(p.intValue()).pow(q.intValue())
                  .mod(n).compareTo(x.mod(n))==0 ){
                  System.out.println("q = log di "+x+
                        " in base "+x+"^"+p+" = "+q);
                  System.out.println("x^pq mod n =
                        "+x+"^"+p+"*"+q+"mod "+n+" = "
                        +x.pow(q.multiply(p).intValue())
                        .mod(n) );
                  if (x.pow(q.multiply(p).intValue())
                        .mod(n).compareTo(x)==0){
                        System.out.println("m = pq-1 =
                              "+q.multiply(p)
                              .subtract(BigInteger.ONE));
                  Return
                        (q.multiply(p)
                        .subtract(BigInteger.ONE));
                  }
            }else
                  System.out.println("q = log di "+x+" in
                        base "+x+"^"+p+"= NON ESISTE");
      }
      else System.out.println(" che NON e' primo ");
      p = p.add(BigInteger.ONE);
}
System.out.println("Non e' stato trovato nessun
            esponente perciò restituisco m = 0");
return BigInteger.ZERO;
}
```

```
public static BigInteger trovaFattore(BigInteger n, BigInteger x,
                                      BigInteger m){
            BigInteger p = new BigInteger("1");
            BigInteger k = new BigInteger("0");
            BigInteger temp1 = new BigInteger("1");
            BigInteger temp2 = new BigInteger("1");
            BigInteger due = new BigInteger("2");
            BigInteger menouno = new BigInteger("-1");
            temp1 = x.pow(m.divide(due.pow(k.intValue()))
                                                  .intValue() );
            temp2 =
                  x.pow(m.divide(due.pow(k.add(BigInteger.ONE)
                  .intValue()))).intValue() );
            System.out.println("x = "+x);
            System.out.println("m = "+m);
            System.out.println("k = "+k);
            System.out.println("x^(m/2^k) mod n = "+temp1.mod(n) );
            System.out.println("x^(m/2^k+1) mod n= "+temp2.mod(n) );
            //mentre x^(m/2^(k+1)) mod n != +1 o -1 aumenta k di 1
            while (temp1.mod(n).compareTo(BigInteger.ONE)==0 &&
                  (temp2.mod(n).compareTo(BigInteger.ONE)==0 ||
                   temp2.mod(n).compareTo(menouno)==0)) {
            k = k.add(BigInteger.ONE);
                  temp1 =
                  x.pow(m.divide(due.pow(k.intValue())).intValue());
                  temp2 =
                  x.pow( m.divide(due.pow(k.add(BigInteger.ONE)
                  .intValue()))).intValue());
                  System.out.println("k = "+k);
                  System.out.println("x^(m/2^k) mod n =
                              "+temp1.mod(n) );
                  System.out.println("x^(m/2^k+1) mod n =
                              "+temp2.mod(n) );
            }
            p = temp2.add(BigInteger.ONE).gcd(n);
            System.out.println("p = MCD(x^(m/2^k+1) mod n + 1, n) =
            MCD("+temp2.add(BigInteger.ONE).mod(n)+", "+n+") = +p);
            return p;
      }
```

```
	public static void main(String[] args) {
		Scanner input = new Scanner(System.in);
		System.out.println("Inserisci il numero n
		  (dispari e non potenza di primi) o 0 per uscire dal
		  programma: ");
		BigInteger n = new BigInteger(input.nextLine());
		while (n.compareTo(BigInteger.ZERO)!=0){
			//variabile che contiene l'esponente da restituire
			BigInteger x = new BigInteger("2");
			BigInteger m = new BigInteger("1");
			BigInteger p = new BigInteger("1");
			//mentre n mod x != 0 && p=1
			//questo ciclo non puo' rimanere in loop
			//perche' prima o poi x raggiungera' divisore di n
			while (n.mod(x).compareTo(BigInteger.ZERO)!=0 &&
					(p.compareTo(BigInteger.ONE)==0 ||
					p.compareTo(n)==0)){
				m = trovaEspConLogDiscr(x,n);
				p = trovaFattore(n,x,m);
				//se alla fine del ciclo nn trovo un fattore
				//allora cambio il valore di x con x+1 mod n
				x = x.add(BigInteger.ONE).mod(n);
			}
			System.out.println("Fuori dal ciclo while:");
			System.out.println("x = "+x);
			System.out.println("p = "+p);
			System.out.println("Inserisci il numero n (dispari
				e non potenza di primi) o 0 per uscire dal
				programma: ");
			n = new BigInteger(input.nextLine());
		}
	}
}
```

# UNA BREVE RIFLESSIONE…

La corsa che ha avuto luogo negli ultimi decenni verso la ricerca di fattorizzazioni sempre più sorprendenti assomiglia molto a quella dei centometristi, che con anni di durissimo lavoro riescono ad abbassare i loro record di pochi centesimi di secondo e raggiungendo velocità estremamente più lente di quelle raggiungibili dall'uomo con altri mezzi. Per quanto si sia riusciti a scomporre numeri di seicento cifre binarie in pochi mesi, e per quanto i miglioramenti si siano rivelati sorprendentemente al di sopra delle aspettative grazie ai nuovi metodi, all'avvento del calcolo in parallelo e all'evoluzione straordinaria della tecnologia, i numeri fattorizzati restano un nulla, una molecola nell'universo, un istante nell'eternità, di fronte all'arbitraria grandezza che un numero può assumere. E come in questo campo della matematica particolarmente complicato, così l'uomo, nel corso della sua vita, non potrà che raggiungere spazi e luoghi finiti e compiere un numero finito di azioni. Viene spontaneo allora, chiedersi quale sia il fascino di vivere. Credo che non a tutti sia dato di comprenderlo, ma solo a coloro che avranno il coraggio di giungere fino al confine di tali spazi, che avranno la determinazione e la volontà di rendere le proprie azioni insignificanti imprese. Infatti, in piedi, sulla linea tracciata dal proprio limite è l'unico luogo da cui si può contemplare l'infinito mistero che sovrasta la vita umana e la sua misera condizione. Per la bellezza di questo mistero vale la pena passare qualche notte sui libri, alzarsi preso la mattina, stare un mese dietro un teorema che sembra incomprensibile, fare sacrifici per pagare le tasse universitarie, e addirittura laurearsi in ritardo per dare gli esami solo quando li si comprende veramente.

E poi chissà che un giorno Dio non voglia rivelare all'uomo se c'è un metodo per scomporre un numero in fattori primi in un tempo uguale a quello che ci vuole per comporlo. Perché a volte, osservando e scrutando con attenzione il mistero, qualcuna delle sue idee diventa una nostra intuizione…

# BIBLIOGRAFIA


[1] H.Riesel, *Prime Numbers and Computer Methods for Factorization*, Birkhauser, Boston, seconda edizione, 1994

[2] R.Crandall, C.Pomerance, *Prime Numbers. A Computational Perspective*, Springer-Verlag, Berlin, Heidelberg, New York, seconda edizione, 2005

[3] A.Languasco, A.Zaccagnini, *Introduzione alla Crittografia. Algoritmi – Protocolli – Sicurezza Informatica*, Ulrico Hoepli, Milano, 2004

[4] D.Bishop, *Introduction to Cryptography with Java Applets*, Jones and Barlett, 2003

[5] R.Brent, *An Improved Monte Carlo Factorization Algorithm*, BIT v.20, 1980, pp176 - 184

[6] D.E.Knut, *The Art of Computer Programming*, vol. 2, Addison – Wesley, Reading, Mass., 1969

[7] J.M.Pollard, *A Monte Carlo Method for Factorization*, BIT, v.15, 1975, pp.331 – 334, MR50#6992

[8] R.Brent, J.Pollard, *Factorization of the Eighth Fermat Number*, Math. Comp., 36:627-630, 1981

[9] H.Davenport, *The Higher Aritmetic – An Introduction to the Theory of Numbers*, Cambridge University Press, Cambridge, quinta edizione, 1989

[10] R.Crandall, *Parallelization of Pollard-Rho Factorization*, 1999. http://www.perfsci.com

[11] E.Bach, *Discrete Logartithms and Factoring*, Computer Science Division, University of California, Berkeley, CA 94720

[12] S.Galbraith, *Mathematics of Public Key Cryptography*, ???

[13] M. J. Wiener, *Bounds on birthday attack times*, Cryptology ePrint Archive, Report 2005/318.

[14] P. Flajolet and A. M. Odlyzko, *Random mapping statistics*, EUROCRYPT ’89 (J.-J.Quisquater and J. Vandewalle, eds.), LNCS, vol. 434, Springer, 1990, pp. 329–354.

[15] , E.Teske, *Speeding up Pollard’s rho method for computing discrete logarithms*, ANTS III (J. P. Buhler,ed.), LNCS, vol. 1423, Springer, 1998, pp. 541–554.

[16] J. Horwitz and R. Venkatesan, Random *Cayley digraphs and the discrete logarithm*, ANTS V C. Fieker and D.R. Kohel, eds.), LNCS, vol. 2369, Springer, 2002, pp. 416–430.

[17] S. D. Miller and R. Venkatesan, *Spectral analysis of Pollard rho collisions*, ANTS VII (F.Hess, S. Pauli,and M. E. Pohst, eds.), LNCS, vol. 4076, Springer, 2006, pp. 573–581.

[18] J. H. Kim, R. Montenegro, Y. Peres, and P. Tetali, *A birthday paradox for Markov chains, with an optimal bound for collision in the Pollard rho algorithm for discrete logarithm*, ANTS VIII (A. J.van der Poorten and A. Stein, eds.), LNCS, vol. 5011, Springer, 2008, pp. 402–415.

[19] J. H. Kim, R. Montenegro, and P. Tetali, Near *optimal bounds for collision in Pollard rho for discrete log*, FOCS, IEEE Computer Society, 2007, pp. 215–223.

[20] J. Sattler and C.-P. Schnorr, *Generating random walks in groups*, Ann. Univ. Sci. Budapest. Sect.Comput. 6 (1985), 65–79.

[21] J. Arney and E. D. Bender, *Random mappings with constraints on coalescence and number of origins*, Pacific J. Math. 103 (1982), 269–294.

[22] S. R. Blackburn and S. Murphy, *The number of partitions in Pollard rho*, unpublished manuscript, 1998.

[23] P. C. van Oorschot and M. J.Wiener, *Parallel collision search with cryptanalytic applications*, J. Crypt. 12 (1999), 1–28.

[24] E. Schulte-Geers, *Collision search in a random mapping: Some asymptotic results*, Presentation at ECC 2000, Essen, Germany, 2000.

[25] F. Kuhn and R. Struik, *Random walks revisited: Extensions of Pollard's rho algorithm for computing multiple discrete logarithms*, SAC '01 (S. Vaudenay and A.M. Youssef, eds.), LNCS, vol. 2259, Springer, 2001, pp. 212–229.

[26] R. P. Gallant, R. J. Lambert, and S. A. Vanstone, *Improving the parallelized Pollard lambda search on binary anomalous curves*, Math. Comp. 69 (2000), no. 232, 1699–1705.

[27] M. J. Wiener and R. J. Zuccherato, *Faster attacks on elliptic curve cryptosystems*, SAC '98 (S. E. Tavares and H. Meijer, eds.), LNCS, vol. 1556, Springer, 1998, pp. 190–200.

[28] I. M. Duursma, P. Gaudry, and F. Morain, *Speeding up the discrete log computation on curves with automorphisms*, ASIACRYPT '99 (K.Y. Lam, E. Okamoto, and C. Xing, eds.), LNCS, vol. 1716, Springer, 1999, pp. 103–121.

[29] J. H. Cheon, J. Hong, and M. Kim, *Speeding up the Pollard rho method on prime fields*, ASIACRYPT 2008 (J. Pieprzyk, ed.), LNCS, vol. 5350, Springer, 2008, pp. 471–488.

[30] J. W. Bos, M. E. Kaihara, and T. Kleinjung, *Pollard rho on elliptic curves*, preprint, 2009.

[31] J. Pollard, *Monte Carlo methods for index computation mod p*, Mathematics of Computation, Volume 32, 1978.

## CITAZIONI

*"317 è un numero primo, non perché lo pensiamo noi, o perché la nostra mente è conformata in un modo piuttosto che in un altro, ma perché è così, perché la realtà matematica è fatta così. "*
Godfrey Harold Hardy

*"I numeri primi sono divisibili soltanto per 1 e per se stessi. Se ne stanno al loro posto nell'infinita serie dei numeri naturali, schiacciati come tutti fra due, ma un passo in là rispetto agli altri. Sono numeri sospettosi e solitari …tra i numeri primi ce ne sono alcuni ancora più speciali. I matematici li chiamano primi gemelli: sono coppie di numeri primi che se ne stanno vicini,anzi,quasi vicini, perché fra di loro vi è sempre un numero pari che gli impedisce di toccarsi per davvero. Numeri come l'11 e il 13, come il 17 e il 19, il 41 e il 43. Se si ha la pazienza di andare avanti a contare, si scopre che queste coppie via via si diradano. Ci si imbatte in numeri primi sempre più isolati, smarriti in quello spazio silenzioso e cadenzato fatto solo di cifre e si avverte il presentimento angosciante che le coppie incontrate fino a lì fossero un fatto accidentale, che il vero destino sia quello di rimanere soli. Poi, proprio quando ci si sta per arrendere, quando non si ha più voglia di contare, ecco che ci si imbatte in altri due gemelli, avvinghiati stretti l'uno all'altro. Tra i matematici è convinzione comune che per quanto si possa andare avanti, ve ne saranno sempre altri due, anche se nessuno può dire dove, finché non li si scopre."*
Paolo Giordano

*"Che l'aritmetica sia la base di ogni attività mentale è provato dal fatto che sia l'unica cosa che una macchina possa ottenere"*
Arthur Schopenhauer

*"One, probability is a factor which operates within natural forces. Two, probability is not operating as a factor. Three, we are now within un-, sub- or supernatural forces…The equanimity of your average tosser of coins depends upon a law, or rather a tendency, or let us say a probability, or at any rate a mathematically calculable chance, which ensures that he will not upset himself by losing too much nor upset his opponent by winning too often. This made for a kind of harmony and a kind of confidence. It related the fortuitous and the ordained as nature. The sun came up about as often as it went down, in the long run, and a coin showed heads about as often as it showed tails. Then a messenger arrived. We had been sent for, nothing else happened. Ninety-two coins spun consecutively have come down heads ninety-two consecutive times… and for the last three minutes on the wind of a windless day I have heard the sound of drums and flute…"*
Tom Stoppard

*"«Signore», mi permetta di farle una domanda. Se la Chiesa le dicesse che «due più tre fa dieci» cosa farebbe? «Signore», mi rispose «ci crederei e conterei così: uno, due, tre, quattro, dieci»"*
James Boswell

*"It takes a long time to understand nothing."*
Edward Dahlberg

*Grazie Gesù,*
*a mamma+papà(e ai "soldini x il pranzo?"),*
*al prof.Cerruti (per la libertà, la stima,*
*la gentilezza e la competenza concessemi),*
*a Fra+Irene (x la luce accesa la notte),*
*a Maio, Laura, Michi, Renzo e Andre (dove sarei ora senza voi?*
*A dim l'ipotesi di Riemann con triangoli e circonferenze?...),*
*a Pulzel+Gheiass (sempre a ridere delle mie disgrazie...),*
*al Puddu, al Conte, al numero 3,*
*a ki si batte x ki nn conosce,*
*e a tutti gli amici ke hanno vissuto con me le fatiche di questa laurea...*