Appunti di Crittografia

Pietro Battiston
Dipartimento di Matematica
Università degli Studi di Pisa

8 Febbraio 2007

Capitolo 1

Introduzione

1.1 Introduzione a questi appunti

Questo documento riporta alcuni appunti che ho preso nel corso di *Metodi per la crittografia* del prof. Traverso (corso di laurea in Matematica, Università di Pisa), nel primo semestre dell'anno 2006-2007. Intimo però al(l'eventuale) lettore di non prendere niente per oro colato, dato che ho riarrangiato/interpretato qua e là, qunidi mi assumo la responsabilità di qualsiasi bestialità presente tra queste pagine.

Come tante cose che comincio in grande e dopo un po' lascio in un angolo, questi appunti sono palesemente incompleti (ci sono alcuni algoritmi importanti, ma nessuna descrizione dei crittosistemi trattati nel corso, che puoi comunque trovare descritti abbastanza chiaramente sulla wikipedia inglese); NON penso che svilupperò mai oltre questi appunti (se ti piacerebbe farlo, sono dispostissimo a passarti i sorgenti); ciononostante, se trovi errori, scrivimi pure, sarò felice di correggerli: battiston@mail.dm.unipi.it.

1.2 Introduzione alla crittografia

Da migliaia di anni gli uomini hanno affrontato il problema di codificare un messaggio per renderlo incomprensibile a chi eventualmente lo intercettasse. La prima codifica di cui abbiamo notizia è il cosiddetto *codice di Cesare*, che consiste semplicemente nel sostituire ogni lettera con quella che viene k lettere dopo nell'alfabeto[1], dove k è un numero concordato tra le due parti (una *chiave*), e può essere un qualsiasi intero minore della lunghezza dell'alfabeto utilizzato.
Si dice *variabilità* la dimensione dello spazio delle chiavi possibili. Nel codice di Cesare, la variabilità è data dalla lunghezza dell'alfabeto, ovvero è decisamente bassa (ad esempio 21 in quello italiano); questo significa che il codice non offre una grande resistenza ai cosiddetti *metodi a forza bruta*, ovvero i metodi che si basano sul provare ripetutamente varie possibilità, in modo più o meno casuale, per decodificare un codice. In questo caso il metodo, che infatti chiunque può

[1] Ovviamente l'alfabeto viene visto come ciclico: alla Z segue di nuovo la A

implementare addirittura a mente, è provare una dopo l'altra, sul messaggio codificato, le possibili chiavi (prima provo a sostituire la A con la B, la B con la C e così via... poi provo a sostituire la A con la C, la B con la D e così via... poi la A con la D...), e fermarsi quando si ottiene un messaggio di senso compiuto[2].

Si potrebbe osservare che nel caso del codice di Cesare in realtà una persona qualunque che intercettasse il codice dovrebbe perlomeno sapere, prima di poter applicare il metodo a forza bruta, che quel messaggio è effettivamente codificato con un codice di Cesare: in altre parole, si potrebbe pensare che il modo più sicuro per scambiare informazioni codificate è utilizzare un codice che non dipende da una chiave segreta ma che è in sé segreto. In realtà questo argomento vacilla per vari motivi:

- efficienza: se voglio comunicare in modo segreto con varie persone, e con ogni persona indipendentemente dalle altre, sarà per me più conveniente ricordare e utilizzare diverse chiavi per uno stesso algoritmo di cifratura che diversi algoritmi. Inoltre, se l'algoritmo che uso è pubblicamente conosciuto, per instaurare un dialogo con una nuova persona sarà sufficiente fare in modo di provvederle la chiave, e non tutte le regole della codifica.
- sicurezza: se la mia comunicazione è basata su una chiave, sarà semplicissimo cambiare periodicamente (e di comune accordo con il mio interlocutore) tale chiave in modo da rendere ancora più sicura la comunicazione (ad esempio durante la Seconda Guerra Mondiale le famosissime macchine Enigma, che i Nazisti utilizzavano per comunicare in codice, avevano una chiave che cambiava ogni giorno)
- affidabilità: in generale è difficile dimostrare in modo rigoroso che un codice è difficile da decodificare. Sebbene ci siano effettivamente alcuni strumenti per farlo, la più grande garanzia della sicurezza di un metodo di codifica è sempre sapere che esso è ampliamente utilizzato e che nessuno è (ancora) riuscito a "crackarlo".

È per questo che al giorno d'oggi i metodi crittografici solitamente utilizzati sono definiti da standard pubblici, che permettono a chiunque di realizzare al computer programmi che comunichino con tali codici. Esistono poche eccezioni, principalmente in ambito militare, dove effettivamente vengono utilizzati alcuni codici il cui stesso funzionamento è totalmente segreto.

Tornando al codice di Cesare, possiamo descriverlo più formalmente assegnando ad ogni lettera un numero (la sua posizione nell'alfabeto) e al codice stesso la funzione:

$$\phi(x) = x + k$$

[2]E se un messaggio di senso compiuto potesse essere codificato e poi decodificato con *un'altra chiave* in modo tale da farlo diventare *un altro messaggio* di senso compiuto?! Il dubbio è legittimo ma in realtà è estremamente improbabile che ciò accada; anche se una dimostrazione non sarebbe immediata (e in parte dipende da ciò che si intende esattamente per "senso compiuto") basti osservare che in un alfabeto di 26 lettere le possibili parole di 4 lettere sono $26^4 = 456976$, quelle di senso compiuto non più di 10000 e questo significa che ad esempio in un messaggio di 104 lettere prese a caso ci sarà una media di $104 * \frac{10000}{456976} \approx 2,5$ parole di 4 lettere di senso compiuto: la media scende enormemente per parole di 5 o più lettere, e ci si rende conto che difficilmente un messaggio composto quasi solo da parole di 3 o meno lettere avrà senso computo.

, dove k è ovviamente la chiave. Dato che la debolezza di questo codice è la limitatissima variabilità, potrebbe venire la tentazione di modificarlo, cambiando k da lettera a lettera:

$$\phi(x) = x + k(x)$$

Ma ϕ (e di conseguenza k) non può essere qualunque! Infatti:

- deve essere iniettiva: se per assurdo avessimo due lettere x e y tali che $\phi(x) = \phi(y) = z$, sarebbe impossibile, anche al destinatario del messaggio, stabilire con certezza se una z nel messaggio vada decodificata come x o come y.
- di conseguenza, dato che in questo codice lo ogni lettera dell'alfabeto viene trasformata in una stessa lettera dello stesso alfabeto, deve essere surgettiva

Ovvero ϕ deve essere una *permutazione* degli elementi da 1 a N, dove N è la lunghezza del nostro alfabeto.
Le possibili permutazioni di N elementi sono $N!$ (nell'esempio dell'alfabeto italiano, $21! \approx 10^19$), che è una variabilità più che sufficiente per mettere fuori gioco qualsiasi metodo di forza bruta. Ma anche questo metodo ha una grande debolezza: si può utilizzare criteri statistici sulle frequenze e le posizioni delle lettere nel linguaggio naturale utilizzato, e quindi ricostruire il messaggio originale partendo dale lettere più utilizzate.
C'è un rimedio anche a questo, e consiste nel modificare il messaggio (ad esempio con un algoritmo di compressione come zip, gzip, bzip) *prima* di codificarlo, in modo tale che il messaggio che si codifica non sia più in linguaggio naturale. Resta il problema di fondo che un tale codice non è praticamente più un codice non dipende praticamente più da una chiave, perché la chiave è diventata ϕ, cioé il codice stesso... quindi è praticamente un codice monouso.

Nel Rinascimento nacquero molti nuovi codici, ed il primo che abbia un certo interesse matematico è quello di *Vigenére*, che funziona così: si concorda una chiave (ad esempio un certo passo della Bibbia), si traduce in numeri le lettere (sempre le posizioni nell'alfabeto) del messaggio e della chiave, si fa una somma modulo N, numero per numero, tra chiave e messaggio e si spedisce il risultato. Sebbene questo codice sia notevolmente più robusto di quello di Cesare nei confronti dello studio statistico, non è imbattibile: basandosi sempre sulle lettere più frequenti e sulle adiacenze tra di esse è possibile con una certa rapidità decrittare un messaggio (che non sia troppo corto).
Esiste in realtà un modo per rendere il codice di Vigenére davvero robusto, e consiste nel

1. scrivere i numeri in sistema binario invece che decimale
2. utilizzare come chiave una stringa casuale (ma *della stessa lunghezza del messaggio*, e diversa per ogni messaggio!) di 0 e di 1.

È evidente che però se due individui hanno problemi a passarsi i messaggi con sufficiente sicurezza, avranno altrettanti problemi a passarsi una chiave della stessa lunghezza; perciò questo metodo (che viene detto *codice di Verman*, o *one-time pad*) rappresenta solo una curiosità matematica, essendo assolutamente inadatto ad un utilizzo reale.

1.3 Non solo codifica

Facciamo un passo indietro: sebbene l'utilizzo più noto e più antico della crittografia sia proprio quello di *codificare* dei messaggi, al giorno d'oggi è molto importante anche per:

- *autenticare* un messaggio (riconoscere che proviene da un certo mittente)
- *garantire l'integrità* di un messaggio (riconoscere che non è stato manipolato)

In entrambi i casi il termine che spesso si utilizza è "*firma digitale*" (di una persona o di un messaggio). La firma digitale deve ovviamente essere legata al particolare messaggio (in altre parole, non si firma mai fogli scritti con inchiostro simpatico!).

Il motivo per cui questi tre problemi (codifica, autenticità, integrità) vengono raggruppati tutti nel termine *crittografia* è che i metodi utilizzati per risolverli sono molto simili.

Diamo ora uno schema generale di una possibile comunicazione. Alice vuole mandare un messaggio a Bob, ma il messaggio nel suo tragitto viene intercettato da Caterina, che vuole capire cosa i due si stanno dicendo.
Alice e Bob hanno però a disposizione una macchinetta ϕ che traduce i messaggi in chiaro in messaggi in codice e viceversa.
Abbiamo detto prima che ϕ deve essere iniettiva. Può però avere componenti aleatorie (come la funzione crypt del linguaggio di programmazione php) , ovvero lo stesso messaggio può essere codificato in più di un modo (ma solo quel messaggio sarà codificato in ognuno di quei modi!).
ϕ dovrà avere una funzione inversa, ma scoprire tale inversa dovrà essere un'operazione computazionalmente proibitiva (solitamente ciò significa sia che ϕ^{-1} non è esprimibile in modo sintetico[3], sia che dato un certo y, trovare $\phi^{-1}(y)$ richiede troppi calcoli).
Ovviamente il concetto di "computazionalmente proibitivo" cambia molto con il tempo e con i miglioramenti dei computer, ma già le chiavi a 56 bit (che venivano utilizzate negli anni Settanta) possono rendere la forza bruta quantomeno dispendiosa anche con macchine moderne.
La crittografia prevede anche situazioni apparentemente paradossali, come una funzione $f : A \to B$ con $|A| > |B|$ *computazionalmente iniettiva*, ovvero tale che trovare due elementi con la stessa immagine sia difficile. Talvolta ϕ può essere tale che, data una qualunque chiave k,

$$\phi(\phi(m, k)) = m$$

Le "macchinette" di questo tipo si chiamano *codici simmetrici*, e possono essere utilizzate per garantire l'integrità di un certo messaggio, ma evidentemente non per autenticare (dato che chiunque voglia verificare l'autenticità di un messaggio dovrebbe essere in possesso della chiave di chi l'ha autenticato!). Anche per la garanzia dell'integrità comunque deve essere risolto il problema di far pervenire

[3]Ovvero che l'unico modo per esprimere ϕ^{-1} è una lista di tutti i valori che essa può assumere, ognuno messo in relazione con l'elemento di cui è immagine.

la chiave (che potrebbe essere a sua volta manipolata) al destinatario.
La permutazione delle lettere vista come miglioramento del codice di Cesare è in fondo una forma di crittografia a chiave simmetrica, dato che la chiave che serve per codificare è anche quella che serve per decodificare (anche se usata in modo leggermente diverso, l'informazione necessaria è la stessa). Un altro esempio di codifica simmetrica può essere un anagramma, la cui configurazione è concordata tra mittente e destinatario, delle lettere del messaggio, oppure un *cambiamento di base* della stringa vista come un vettore in $\mathbb{Z}[N]^m$. Si chiama poi *crittografia a chiave pubblica* una configurazione in cui una chiave è accessibile a tutti, e la chiave di cifratura e quella di decifratura sono correlate matematicamente ma in modo tale che ottenere l'una dall'altra è computazionalmente proibitiva. Ad esempio Bob potrebbe voler dare una chiave pubblica che permette a chiunque di codificare messaggi che poi solo lui, con la sua chiave privata, può decodificare: tutti avranno così la possibilità di scrivergli in maniera sicura.

1.4 La matematica nella crittografia

Ma che c'entra la matematica con la crittografia?
Il compito della matematica in questo caso è proprio fornire casi di ϕ tali che ϕ^{-1} esista ma non sia facile da trovare; seguono alcuni esempi:

- RSA: dati $n = pq$, con p, q primi[4]: n si ottiene molto facilmente da p e q, ma non è vero il viceversa.

- *logaritmo discreto*: in $\mathbb{Z}_n$è facile calcolare $b = a^x$ dati a, x, ma non altrettanto calcolare $x : a^x = b$ dati a, b.

- È facile trovare sistemi di equazioni con una soluzione data, ma non altrettanto trovare una soluzione compatibile con un sistema dato.

- Il cosiddetto *problema dello zaino* dato un insieme di numeri naturali, trovarne un sottoinsieme di sommma totale data è un processo complesso: viceversa, dato il sottoinsieme trovare la somma è banale.

- Curve ellittiche: particolari curve i cui insiemi di valori sono sempre gruppi (e su qualsiasi campo).

1.5 Testi consigliati per approfondimenti:

- Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone - Handbook of applied cryptography

- N. Koblitz - A course in number theory and criptography

- N. Koblitz - Algebraic methods of cryptography

[4]Verificare la primalità di un numero è molto meno costoso che fattorizzarlo.

1.6 Crittografia e comunicazione

Se tra gli scopi della crittografia c'è la possibilità di comunicare a distanza avendo garanzie sull'integrità del contenuto, che differenza c'è con i codici correttori? La differenza è che i codici correttori si basano su un modello di perturbazioni casuali di cui è possibile limitare il danno grazie ad una certa ridondanza, invece la crittografia si occupa di intrusioni *intelligenti* (si può vedere come un esempio di teoria dei giochi). Parlando di crittografia non ci porremo il problema di errori di trasmissione accidentali.

Ma come funziona effettivamente una comunicazione crittografata? Ci sono fondamentalmente due possibilità: la codifica *a blocchi*, in cui ogni "pacchetto' è indipendente, e quella *a flusso*, in cui ogni trasmissione si basa in un qualche modo sulla trasmissione precedente.

1.7 La RSA

Vediamo un po' nel dettaglio uno dei più utilizzati metodi di crittografia a chiave pubblica: la RSA.
Bob stabilisce dei numeri $n = p * q$, con p e q primi, e rende pubblico n. Inoltre c'è un altro numero, a che è fissato (tutti utilizzano lo stesso).
La codifica è data dalla funzione $\phi(x) \equiv x^a \pmod{n}$.
La decodifica dovrà mandare x^a in $(x^a)^b \equiv x \pmod{n} \Leftrightarrow x^{ab-1} \equiv 1 \pmod{n}$.
Consideriamo x invertibile (perché se so che non lo è ho automaticamente fattorizzato n, per cui scoprire che non lo è è considerato un problema almeno tanto difficile quanto al fattorizzazione di n).
Sappiamo che $x^{\phi(n)} = 1 = x^{k\phi(n)}$. Ma $ab \equiv 1 \pmod{()\phi(n)}, (a, \phi(n)) = 1 \Rightarrow$ so trovare b.

$$\phi(n) = (p-1)(q-1)$$

Caterina vedrà n ed a, ma non p, q, a, b. Dovrebbe ritrovare b, e perciò le basterebbe conoscere $\phi(n)$, così da conoscere pq e $p + q$...
Tutto va bene se effettivamente le operazioni di Alice e Bob sono *facili* e quelle di Caterina *difficili*.

Osservazione: In matematica si utilizza spesso il concetto di *complessità computazionale* di un problema da risolvere, ma questo concetto si dimostra poco interessante in crittografia. Infatti se un problema ha complessità computazionale, poniamo, esponenziale, significa che *nel caso pessimo* la sua risoluzione ha effettivamente costo esponenziale. Ma a noi non basta che Caterina debba sopportare un costo esponenziale nel caso pessimo: noi vogliamo che per lei le operazioni siano *sempre* difficili, anche nel caso migliore.

Capitolo 2

Test di primalità

Molti sistemi crittografici necessitano di primi grossi. Si pone quindi il problema di come generarli con costo minore possibile.
I metodi deterministici sono poco convenienti, per cui ci si accontenta di utilizzare numeri che *con grande probabilità* sono primi. Se prendo un numero a caso, la probabilità che esso sia primo è circa $\frac{n}{\log n}$. Se prendo un numero a caso che non sia multiplo di 2 o 3, la probabilità che esso sia effettivamente un primo è già ragionevole.
A questo punto effettuerò ripetutamente un test che, se n è in realtà composto, abbia una qualche probabilità di dimostrarmelo, e ripeterò questo test fin tanto che la probabilità di un *falso positivo* (un numero composto che io assumerò primo) sia sufficientemente bassa per le mie necessità.
Riportiamo di seguito due possibili test.

2.1 Criterio di Fermat

Il (piccolo) teorema di Fermat dice che se n è primo, l'equazione:

$$a^n \equiv a \pmod{n}$$

è verificata $\forall a \in \mathbb{Z}_n$.
Il *test di Fermat* consiste quindi, dato un numero n, nel controllare se per un qualche a si ha $a^n \not\equiv a \pmod{n}$: se si trova un tale a, n è necessariamente composto.

Analisi dell'algoritmo: Dato n a caso, è possibile scegliere abbastanza valori di a (ad esempio una buona scelta è solitamente $a = 2$) da rendere la probabilità di un *falso positivo* (ovvero di un numero composto che il test non riconosce come tale) piccola a piacere. Esistono però anche numeri n non primi tali che $a^n \equiv a \pmod{n} \forall a \in \mathbb{Z}_n$.

Operativamente poi di solito si verifica la condizione equivalente (dato che il caso $a = 0$ ovviamente non ci interessa):

$$a^{n-1} \equiv 1 \pmod{n}$$

2.2 Criterio di Miller-Rabin

In $\mathbb{Z}_n$, con n dispari, se esiste a tale che $a^2 = 1$ ma $a \neq \pm 1$, allora n non è primo (questo è vero perché n primo $\Rightarrow \mathbb{Z}_n$ è un campo $\Rightarrow$ un polinomio di grado 2 ha al più 2 radici, e $1, -1$ sono sempre radici[1]).
Sia ora $n - 1 = 2^k r$, con r primo. Dato un intero qualsiasi tra 2 e $p - 2$, eseguiamo le seguenti operazioni

1. calcoliamo $b_0 = a^r \pmod n$. Se $b_0 = \pm 1$, affermiamo che n *sembra* primo, e scegliamo un altro a. Altrimenti:

2. calcoliamo $b_1 = {b_0}^2, b_2 = {b_1}^2 \cdots$ e così via.

3. Ad ogni elevamento al quadrato che effettuiamo, se $b_i = -1$ affermiamo che n *sembra* primo. Se $b_i = 1$, terminiamo l'algoritmo affermando che n *è composto*, in quanto $b_{i-1} \neq \pm 1$ ma ${b_i}^2 = 1$.

4. Se arriviamo a b_k senza avere incontrato né un 1 né un -1, affermiamo che n è composto, perché $b_k = {b_0}^{2^k} = a^{r2^k} = a^{n-1} \neq 1 \Rightarrow$ ci riconduciamo al teorema di Fermat.

Analisi dell'algoritmo: Dato un a qualsiasi, i passi descritti vengono a costare al più ${\log_2}^3 n$, ovvero il costo di un esponenziale di esponente n in $\mathbb{Z}_n$.
È evidente che l'efficienza di questo algoritmo dipende dalla probabilità che un a casuale in $\mathbb{Z}_n$, con n non primo, ci permetta di dimostrare che n effettivamente è composto. Si dimostra che questa probabilità è maggiore[2] di $\frac{3}{4}$, e che perciò dopo avere ad eseguire l'algoritmo per k diversi valori di a senza dimostrare che n è composto abbiamo una probabilità minore di $\left(1 - \frac{3}{4}\right)^k = \frac{1}{4^k}$ che esso in realtà lo sia.

Ci limitiamo a dare qui la dimostrazione che la probabilità che il nostro a sia "buono" è maggiore di $\frac{1}{2}$ nel caso in cui $n = pq$ con p, q primi distinti:

$$\mathbb{Z}_n \cong \mathbb{Z}_p \otimes \mathbb{Z}_q \Rightarrow {\mathbb{Z}_n}^* \cong {\mathbb{Z}_p}^* \otimes {\mathbb{Z}_q}^*$$

.
Siano $p - 1 = r_1 2^{k_1}, q - 1 = r_2 2^{k_2}$ gli ordini di ${\mathbb{Z}_p}^*, {\mathbb{Z}_q}^*$. Allora:

$${\mathbb{Z}_p}^* \cong \mathbb{G}_1 \oplus \mathbb{H}_1 \text{ con } |\mathbb{G}_1| = r_1, |\mathbb{G}_1| = 2^{k_1}$$
$${\mathbb{Z}_p}^* \cong \mathbb{G}_2 \oplus \mathbb{H}_2 \text{ con } |\mathbb{G}_2| = r_2, |\mathbb{G}_2| = 2^{k_2}$$

Prendiamo a caso un elemento in ${\mathbb{Z}_p}^* \otimes {\mathbb{Z}_q}^*$: l'algoritmo descritto consiste nel moltiplicare ripetutamente per 2 un elemento casuale di $\mathbb{G}_1 \oplus \mathbb{G}_2 \oplus \mathbb{H}_1 \oplus \mathbb{H}_2$ (a_1, a_2, b_1, b_2) fino ad ottenere l'elemento $(0, 0, 0, 0)$. Il penultimo elemento ottenuto sarà della forma $(a_1 2^k, 0, b_1 2^k, 0)$, $(0, a_2 2^k, 0, b_2 2^k)$ oppure $(a_1^k, a_2^k, b_1^k, b_2^k)$ (corrispondente a -1 in $\mathbb{Z}_n$). Siano $d_1 = ord((0, a_2, 0, b_2))$ e $d_2 = ((a_1, 0, b_1, 0))$. Se $d_1 = 1$ o $d_2 = 1$, abbiamo che gli elementi della successione sono tutti della forma $(0, x, 0, y)$ oppure $(x, 0, y, 0)$, per cui non otterremo mai $(a_1 2^{k-1}, a_2^{k-1}, b_1^{k-1}, b_2^{k-1})$. Altrimenti, la probabilità che effettivamente si ottenga un -1 è minore o uguale

[1] Stiamo suppondendo solo $nnot = 2$

[2] In realtà per la stragrande maggioranza dei numeri questa probabilità è molto maggiore di $\frac{3}{4}$, ma esiste una successione infinita di numeri per cui è $\approx \frac{3}{4}$.

della probabilità che a_1 e a_2 vadano a 0 contemporaneamente, e questa è minore di $\frac{1}{2}$[3].

2.3 Criterio di Solovay-Strassen:

Dato p primo $\neq 2$ e n qualunque, ricordiamo la definizione del *simbolo di Legendre*:

$$\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}}$$

Eulero ha dimostrato che tale simbolo si può definire equivalentemente nel seguente modo:

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & se \quad p|a \\ 1 & se \quad \exists x : x^2 \equiv a \pmod{p} \\ -1 & se \quad \nexists x : x^2 \equiv a \pmod{p} \end{cases}$$

Il *simbolo di Jacobi* è una generalizzazione del simbolo di Legendre a p intero dispari qualunque, definita nel seguente modo:

- se p è primo, il simbolo di Jacobi coincide con quello di Legendre
- se $p = \prod p_i^{\alpha_i}$, $\left(\frac{a}{p}\right) = \prod \left(\frac{a}{p_i}\right)^{\alpha_i}$

Come quello di Legendre, il simbolo di Jacobi rispetta la *legge di reciprocità quadratica*, e può quindi essere calcolato con facilità. Dato $a \in \mathbb{Z}_p$, è quindi poco costoso verificare se vale l'equazione:

$$a^{\frac{p-1}{2}} = \left(\frac{a}{p}\right)$$

Se troviamo almeno un a per cui essa non vale, ciò significa che il simbolo di Legendre (termine di sinistra) è diverso dal simbolo di Jacobi (termine di destra), e quindi p non è primo.

Analisi dell'algoritmo: Prima che fosse sviluppato del test di Miller-Rabin, questo era il più utilizzato, essendo più efficiente di quello di Fermat.
In particolare, si dimostra che dato n composto, per almeno metà degli $a \in \mathbb{Z}_n$ l'equazione di sopra non è verificata, mentre abbiamo visto che alcuni n composti (*numeri di Carmichael*) non vengono affatto "smascherati" dal test di Fermat.

[3] perché mai?! FIXME

Capitolo 3

Algoritmi di fattorizzazione

3.1 Algoritmo standard

L'algoritmo più intuitivo per fattorizzare un numero composto consiste ovviamente nel provare a dividerlo per ognuno dei numeri più piccoli. Un semplicissimo ragionamento dimostra che possiamo accontentarci di provare con i numeri più piccoli di $\sqrt{n}$.

Analisi dell'algoritmo: L'esecuzione dell'algoritmo richiede il calcolo di al più $\sqrt{n}$ divisioni, e quindi $\sqrt{n}log^2 n$, il che rende il suo utilizzo proibitivo per i numeri utilizzati in crittografia.

3.2 Crivello di Eratostene

Il crivello di Eratostene è certamente il più antico ed intuitivo metodo di fattorizzazione: si prende una lista di tutti i numeri da 2 a $\sqrt{n}$, dove n è il numero da fattorizzare, e si scorre la lista "cancellando" alcuni numeri. In particolare, per ogni numero p_i non "cancellato" controlliamo se $p_i|n$; se sì, abbiamo trovato un fattore, altrimenti cancelliamo tutti i multipli di p_i e passiamo al successivo elemento non cancellato (che sarà esattamente il successivo numero primo).

Analisi dell'algoritmo: Mentre con l'algoritmo standard provavamo a dividere n per ogni numero minore di $\sqrt{n}$, ora ci risparmiamo tutti i numeri composti (quelli cancellati). Siccome i numeri primi minori di un m dato sono all'incirca[1] $\frac{m}{\log_2{}^2 m}$, il costo di questo algoritmo per la fattorizzazione è al più:

$$\frac{\sqrt{n}}{\log \sqrt{n}} \log_2{}^2$$

Sebbene sia meno costoso dell'algoritmo standard, rimane[2] $\tilde{O}(\sqrt{n})$; inoltre richiede richiede un considerevole utilizzo di memoria (dell'ordine di $\sqrt{n}$).

[1] la stima è asintotica

[2] Con la notazione $O(\tilde{f}(x))$, in inglese *"O soft"*, si intende "ordine di $f(x)$ a meno di termini logaritmici".

3.3 Algoritmo di Fermat per la fattorizzazione

Calcoliamo $\sqrt{n}$. Se è intero, ci riconduciamo ovviamente a fattorizzare $\sqrt{n}$. Altrimenti sia $m = \lceil \sqrt{n} \rceil$.
Calcoliamo $k_0 = m^2 - n$ (che sappiamo essere un intero positivo) e controlliamo se k_0 è un quadrato.
Se non lo è, calcoliamo $h_1 = (m+1)^2 - n$ e controlliamo se h_1 è un quadrato, e andiamo avanti finché non troviamo $i : k_i = (m+i)^2 - n$ sia un quadrato.
A quel punto, sia $h^2 = k_i$: abbiamo $h^2 = m^2 - n \Rightarrow n = m^2 - h^2 = (m-h)(m+h)$ e ci riconduciamo quindi a studiare i due fattori $(m-h)$ e $(m+h)$.

Analisi dell'algoritmo: Evidentemente il costo dell'algoritmo dipende dal numero di operazioni necessarie per verificare se k_i è un quadrato.
Il modo più efficiente per implementare questa verifica è il seguente: calcoliamo $\left(\frac{k_i}{2}\right), \left(\frac{k_i}{3}\right), \left(\frac{k_i}{5}\right), \left(\frac{k_i}{7}\right) \ldots \left(\frac{k_i}{p_h}\right)$ per un opportuno valore di h.
Se troviamo un primo $\bar{p}$ tale che $\left(\frac{p_i}{k}\right) = -1$, sappiamo che k non è un quadrato. Altrimenti, ci andiamo a calcolare effettivamente $\sqrt{k}$, e in quel frangente eventualmente ci accorgiamo che in realtà non viene un numeri intero (ma la probabilità è piuttosto bassa).
Per rendere la verifica più efficiente calcoleremo $k_i \pmod{p_k}$ come $k_i + 1 \pmod{p_k}$.
FIXME: e allora che costo viene?!

Osservazione: gli algoritmi visti finora sono particolarmente efficienti nel fattorizzare numeri con fattori *piccoli*; questo invece si comporta meglio quando i fattori sono vicini a $\sqrt{n}$.

3.4 Algoritmo *p-1* di Pollard

Per un opportuno valore di B, calcoliamo:

$$N = mcm\{m : m < B\} = \prod_{p_i < \sqrt{B}} p_i^{\alpha_i}, \text{ dove } \alpha_i = log_{p_i} B$$

Ora supponiamo che per un dato fattore p di n si abbia che $p - i$ sia B-liscio per potenze[3], ovvero divisore di N, e sia $k = N/(p-1)$.
Calcoliamo $a = 2^N \pmod{n}$ e $b = MCD(a-1, n)$.

$$N = (p-1)k \Rightarrow a = 2^N \equiv 1^k \equiv 1 \pmod{p}$$

$$p | n \Rightarrow MCD(a^N - 1, n) > 1$$

Se quindi troviamo che $MCD(a^N - 1, n) \neq n$, abbiamo trovato un fattore non banale .

[3] Un numero $n = \prod p_i^{\alpha_i}$ si dice *B-liscio* se $\forall i, p_i < B$ e N-liscio *per potenze* se $\forall i, p_i^{\alpha_i} < B$

Analisi dell'algoritmo: questo algoritmo è piuttosto specializzato, dato che parte da un'ipotesi piuttosto forte su p (se $p-1$ *non* è B-liscio, avremo semplicemente che $MCD(a^N-1, n) = 1$): non a caso le specifiche della RSA stabiliscono che, data $\{n, p, q\}, n = pq$ la chiave utilizzata, $p-1$ e $q-1$ *non* devono essere B-lisci per valori troppo piccoli di B.
Bisogna fare anche attenzione al rischio che il numero n da fattorizzare sia prodotto di primi p_k tali che $p_k - 1$ sia B-liscio $\forall k$: in questo caso infatti il massimo comune divisore trovato sarà n perché B è stato scelto troppo *grande*!
Ciononostante, l'algoritmo è talvolta utilizzato ed è particolarmente efficiente se si vuole tentare di fattorizzare molti numeri composti.

3.5 Algoritmo ρ di Pollard

Sia $n = \prod_i p_i^{\alpha_i}$ il numero che vogliamo fattorizzare, dove $p_1 < p_2 < p_3 \cdots p_r$ (ovviamente sia r che i vari p_i sono sconosciuti).
Sia f una funzione pseudocasuale (ovvero dal comportamento deterministico ma statisticamente non caratterizzabile) in $\mathbb{Z}_n$ (un buon esempio è $f(x) = x^2 + 1$), e, dato x_0 un elemento qualsiasi di $\mathbb{Z}_n$, sia $\{x_i\}_{i\in\mathbb{N}}$ la successione così definita:

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

L'algoritmo consiste nel ripetere il seguente passo base:

1. calcolare $(x_i, x_{2i}) = (f(x_{i-1}), f(f(x_{2(i-1)}))$
2. se $x_i = x_{2i}$, l'algoritmo ricomincia partendo da un nuovo elemento x_0, altrimenti:
3. calcolare $m_i = MCD(x_i - x_{2i}, n)$
4. se m_i è un fattore non banale di n, l'algoritmo termina con successo
5. se $m_i = 1$, l'algoritmo continua (si torna al passo 1)

Analisi dei costi: Richiamiamo il cosiddetto *paradosso dei compleanni*: in una stanza con m persone, la probabilità che (almeno) due persone compiano gli anni lo stesso giorno supera $\frac{1}{2}$ se $m > 23$.
Enunciato più in generale e più formalmente, il risultato è il seguente: dato un insieme X da cui si estrae casualmente un insieme alla volta (con reimbussolamento), e dato il numero k di estrazioni che è necessario eseguire prima di incontrare due volte uno stesso elemento, si ha:

$$\mathbb{E}[k] = 1.177\sqrt{|X|}$$

Ovvero, in altri termini, la probabilità che sia estratto almeno 2 volte lo stesso elemento supera $\frac{1}{2}$ quando il numero di estrazioni supera $1.177\sqrt{|X|}$.

Nel nostro caso, X sarà l'insieme delle classi di congruenza modulo p_1, e il paradosso dei compleanni ci permette di affermare che prima di incontrare, nella nostra successione di coppie (x_i, x_{2i}), due elementi tali che:

$$x_i \equiv x_{2i} \pmod{p_i} \Rightarrow p_i | MCD(x_i - x_{2i}, n)$$

ci aspettiamo di dovere effettuare circa $1.177\sqrt{p_i} < 1.177\sqrt{\sqrt{n}} = 1.177\sqrt[4]{n}$ controlli.
Questa analisi si basa però sul fatto che la successione x_i abbia effettivamente un comportamento aleatorio. Invece il comportamento di una successione pseudocasuale diventa evidentemente non aleatorio quando si incontra un elemento già incontrato nel passato; infatti se $x_i = x_j, i \neq j$, allora $x_{i+1} = x_{j+1}, x_{i+2} = x_{j+2} \cdots$, ovvero la successione entra in un ciclo (il nome dell'algoritmo deriva appunto dal ciclo preceduto da un gambo, che nel complesso hanno la forma della lettera greca ρ); se in tale ciclo non c'è una coppia (x_i, x_{2i}) che ci permette di fattorizzare n, il nostro algoritmo non termina mai.
È proprio per evitare questo pericolo che si aggiunge il controllo 2: se le successioni x_i e x_{2i} si incontrano (e questo può succedere solo se sono entrate in un ciclo), cambiamo successione.
Infine, se effettivamente le successioni entrano in un ciclo la condizione $x_i = x_{2i}$ avverrà prima che la successione più lenta (il campo sinistro di ogni coppia) abbia effettuato un giro completo del ciclo (questo perché la distanza tra le due successioni quando la più lenta entra nel ciclo sarà minore della lunghezza del ciclo stesso, e diminuirà di 1 ad ogni passo).

Osservazione: $1.177\sqrt[4]{n}$ è una maggiorazione piuttosto ampia del numero di iterazioni che ci aspettiamo di dover fare, perché considera p_i, il più piccolo fattore di n, pari a $\sqrt{n}$ ma soprattutto perché è fatta considerando solo tale fattore, mentre è ovviamente possibile incontrare anche $x_i - x_{2i} \equiv 0 \pmod{p_j}$ con $j \neq 1$; se ad esempio n ha k fattori minori di $\sqrt{n}$ (contati con molteplicità) possiamo maggiorare tale valore atteso con $\frac{1.177\sqrt[4]{n}}{k}$.

Capitolo 4

Algoritmi per la risoluzione del logaritmo discreto

4.1 Algoritmo della *base di fattori* per il logaritmo discreto

Sia a un generatore di ${\mathbb{Z}_p}^*$, e sia $b = a^x$, con x sconoscuito.
Consideriamo l'insieme $I = \{2, 3, 5, \cdots p_h\}$, degli h più piccoli numeri primi.
Prendiamo una successione $\{r_i\}_{i \in \mathbb{Z}}$ di interi distinti compresi tra 0 e $p-1$, e per ogni i proviamo a fattorizzare a^{r_i} come prodotto di elementi di I.
Creiamo una tabella che abbia una colonna per ogni elemento di h e una riga per ogni elemento a^{r_i} che riesco a fattorizzare, e in ogni cella di coordinate (i, j) riportiamo l'esponente che assume p_j nella fattorizzazione di a^{r_j}:

	2	**3**	**5**	**7**	**11**	**13**	**17**	$\cdots$
$a^{r_{k_1}}$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$a^{r_{k_2}}$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$a^{r_{k_3}}$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$a^{r_{k_4}}$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

Arrivato a r_{k_h}, avrò una tabella con più righe che colonne, ovvero avrò necessariamente una riga linearmente dipendente dalle altre (ovviamente può capitare anche prima di avere h righe, in tal caso mi posso fermare prima). A questo punto, risolvendo un semplice sistema lineare otterrò il logaritmo discreto di (quasi) ogni primo in I, ovvero $x_j : a^{x_j} = p_j$. Se quindi riesco a fattorizzare b come $b = \prod_{i<h} p_i^{\alpha_i}$ (eventualmente $\alpha_i = 0$ per alcuni i), potrò ottenere x come:

$$x = \sum_{i<h} \alpha_i x_i$$

Infatti:

$$a^{\sum_{i<h} \alpha_i x_i} = a^{\sum_{i<h} \alpha_i \log_a p_i} = \prod a^{\alpha_i log_a p_i} = \prod_{i<h} p_i^{\alpha_i} = b$$

Analisi dei costi: Non è facile trovare una stima ragionevole del costo medio di questo algoritmo: esso dipende dal valore del parametro h, e il valore ottimale di questo parametro a sua volta non è univocamente determinato, dato per alcuni valori l'algoritmo sarà più veloce ma per altri valori avrà più possibilità di riuscita[1].
È però evidente che la parte più costosa di questo algoritmo è il riempimento della tabella e la risoluzione del sistema lineare. Ma se si effettuare il logaritmo discreto di diversi elementi di ${\mathbb{Z}_p}^*$ queste operazioni vanno effettuate solo una volta, e quindi questo algoritmo è particolarmente efficiente rispetto ai suoi equivalenti se applicato a diversi b.

4.2 Algoritmo

[1]La probabilità che un numero n sia p_h-liscio può essere approssimata (basandosi sulla stima asintotica della densità dei primi fornita dal *Teorema dei numeri primi*) come:

$$1 - \left(\frac{1}{p_h}\right)^{\frac{n \log p_h - p_h \log n}{\log n \log p_h}}$$

Se il numero n effettivamente è p_h-liscio, il costo dell'algoritmo è (basandosi sull'approssimazione appena riportata):

$$\frac{h\sqrt{n}}{1 - p_h^{\frac{p_h \log n - n \log p_h}{\log n \log p_h}}}$$

Mi guardo bene dal riconoscere in ciò un risultato interessante.