Capitolo 5: Fattorizzazione di interi

Trovare i fattori di un numero intero grande è una impresa assai ardua, e può essere impossibile con le risorse oggi disponibili. Non si conoscono metodi polinomiali per la fattorizzazione, come invece accade per i test di primalità.

Infatti i migliori algoritmi di fattorizzazione noti sono subesponenziali ovvero, in generale, i tempi di elaborazione sono dell'ordine di $2^{\sqrt[3]{\ln n}}$. Per essere più precisi i tempi di elaborazione dei migliori algoritmi di fattorizzazione si comportano tutti come $L(a,n)^k = 2^{k(\lg n)^a (\lg\lg n)^{1-a}}$ dove $0 < a \leq 1$ e generalmente vale $\frac{1}{3}$, k è una costante e n o è il numero da fattorizzare o un suo fattore.
Si noti che le migliorie apportate agli algoritmi esistenti vanno ormai a incidere solo sulla costante k.

Dunque, se per fattorizzare un numero di 100 cifre occorre un tempo Q, per fattorizzarne uno di 200 il tempo sale a circa 50.000.000 Q, e per 300 arriviamo a 60.000.000.000.000 Q. Se Q è un secondo, per un intero di 100 cifre occorre 1 secondo, per 200 cifre più di un anno e mezzo, per 300 cifre 2 milioni di anni.

La RSA Security Inc. gestisce una sfida a livello mondiale, ove vengono premiati coloro che fattorizzano gli interi proposti da loro e pubblicati su WEB in una particolare lista.

Recentemente, il 3 dicembre del 2003, è stato fattorizzato il numero
188198812920607963838697239461650439807163563379417382700763356422988859715234665485319060606504743045317388011303396716199692321205734031879550656996221305168759307650257059 (chiamato RSA576 in quanto costituito da 576 bit pari a 174 cifre decimali) prodotto di due primi entrambi costituiti da 87 cifre decimali:
p=3980750864240649373971255005503864911990643623425267084063851895759463889579261768583317
e
q=4727721461074353025362230719730482246329146953020971164598521711305207112563635903975 27

Il link alla pagina dei "factoring challenge" promossi dalla RSA Security Inc. è:
http://www.rsasecurity.com/rsalabs/challenges/factoring/index.html

§.21 Test di primalità

Prima di dare in pasto un numero primo ad un algoritmo di fattorizzazione è opportuno verificare che il numero sia effettivamente composito. A tale scopo esistono vari test detti di primalità che, però, non garantiscono con certezza che il numero testato sia primo.

In altre parole, se il test viene superato il numero è certamente composito, viceversa è probabilmente primo.

Esistono vari test di primalità. Tra tutti, noi vedremo quello che, al momento, discrimina tra primi e compositi con la maggior probabilità di successo.
Tale test, dovuto a Miller e Rabin, si basa sostanzialmente sul seguente

Piccolo Teorema di Fermat: se p è primo allora $a^{p-1} \equiv 1 \bmod p \quad \forall a = 1, \ldots, p-1$

Consideriamo l'equazione:
(88) $$a^{n-1} \equiv 1 \bmod n$$

Diremo che n è *pseudoprimo in base* a se n è composito e soddisfa l'equazione (88) per un certo valore di a.

Il teorema di Fermat implica che se n è primo deve soddisfare (88) $\forall a \in \mathbb{Z}_n^+$.
Quindi se, dato n, riusciamo a trovare almeno un valore a per cui (88) non vale, potremo concludere che n è certamente composito.
Per questa ragione il valore a viene detto "testimone".

Una semplice test di primalità prevede l'uso di $a = 2$ come testimone.
Se (88) non è soddisfatta n viene dichiarato certamente composito, viceversa n è candidato ad essere primo.

Sorprendentemente n non è solo un candidato ma è anche un buon candidato. Infatti i casi in cui un numero pseudoprimo in base 2 risulta poi composito sono veramente rari. Tanto per dare un'idea ci sono solo 22 valori di n inferiori a 10ÿ000 per cui la procedura "sbaglia". I primi quattro valori sono 341, 561, 645 e 1105.
Inoltre è dimostrato che la probabilità di errore su un numero estratto a caso tra i primi N interi tende a zero al crescere di N.

Più precisamente un numero costituito da 512 bit scelto casualmente e dichiarato primo ha una probabilità su 10^{20} di essere soltanto pseudoprimo in base 2 mentre per un numero costituito da 1024 bit tale probabilità scende a una su 10^{41}.

Per quanto, però, questo test possa essere raffinato ripetendolo per altri valori del testimone, non sarà mai possibile ridurre a zero la probabilità di errore perchè esistono interi che soddisfano (88) pur essendo compositi.
Questi interi, detti numeri di Carmichael, sono estremamente rari: si pensi che ne esistono solo 255 inferiori a 100.000.000 i primi tre dei quali sono 561, 1105 e 1729.

L'ostacolo dei numeri di Carmichael può però essere aggirato facendo uso del seguente

Teorema: se $x^2 \equiv 1 \bmod n$ ammette soluzioni non banali allora n è certamente composito.

Poichè i numeri di Carmichael, per motivi che omettiamo, non sfuggono a questa regola, riusciremo a escludere gli errori da essi generati semplicemente inserendo nei cicli di calcolo della potenza $a^{n-1} \bmod n$ la verifica dell'esistenza delle soluzioni non banali dell'equazione $x^2 \equiv 1 \bmod n$.

A questo punto però abbiamo ricondotto i tempi di elaborazione a quelli di un qualsiasi algoritmo di trial division perchè la certezza che n sia primo l'avremo solo dopo aver escluso tutti i possibili testimoni.

Ecco quindi che nel test di Miller Rabin anzichè tutte le basi ne vengono provate solo alcune scelte casualmente.
Questa scelta fa ricadere il test tra gli algoritmi probabilistici ma, rispetto al test base, la probabilità di errore non dipende più da n bensì solo dal numero s di basi provate.

Infatti è possibile dimostrare che per un qualsiasi intero $n > 2$ dispari e un qualsiasi intero positivo s la probabilità di errore non dipende da n ed è al più 2^{-s}.

Dunque una scelta di $s = 50$ è sufficiente per praticamente qualsiasi applicazione immaginabile.

Prima di presentare la procedura concludiamo la nostra analisi (non esaustiva) osservando che il test richiede al più s volte il calcolo di una potenza modulare che, a sua volta, richiede $O(\lg n)$ operazioni aritmetiche e $O(\lg^3 n)$ operazioni binarie. Quindi, complessivamente, il test di Miller Rabin richiede $O(s \lg n)$ operazioni aritmetiche e $O(s \lg^3 n)$ operazioni binarie.

```
Procedura MILLER-RABIN(n,s)  ▷ n>2 dispari e 0<s<n-1
  for j ← 1 to s
        a ← RND(1,n−1)
        if WIT(a,n) then return FALSE  ▷ n è certamente composito
  next
  return TRUE  ▷ n è quasi sicuramente primo
```

```
Procedura RND(a,b)
  ▷ genera un numero casuale con distribuzione uniforme discreta in a ≤ x ≤ b
  return x
```

Procedura WIT(a,n)

$(u,t) = \text{BASE-SHIFT}(n)$

$x_0 = \text{POWERMOD}(a,u,n)$

for $i \leftarrow 1$ **to** t

$x_i \leftarrow x_{i-1}^2 \bmod n$

if $x_i = 1 \wedge x_{i-1} \neq 1 \wedge x_{i-1} \neq n-1$ **then**

return *TRUE*

next

if $x_t \neq 1$ **then return *TRUE***

return *FALSE*

Procedura BASE-SHIFT(n)

▷ *calcola la coppia* (u,t) *tale che* $n-1 = 2^t u$ *dove* $t \geq 1$ *e* u *è pari*

return (u,t)

Procedura POWERMOD(a,b,n)

$c \leftarrow 0$

$d \leftarrow 1$

$(b_k,\ldots,b_0) = \text{BINARY}(b)$

for $i \leftarrow k$ **downto** 0

$c \leftarrow 2c$

$d \leftarrow d^2 \bmod n$

if $b_i = 1$ **then**

$c \leftarrow c+1$

$d \leftarrow ad \bmod n$

return d

Procedura BINARY(b)

▷ *calcola l'espansione binaria del numero intero* b

return $(b_k,\ldots,b_0)$

§.22 Algoritmi di fattorizzazione

Come abbiamo già detto la fattorizzazione di numeri interi molto grandi ci riporta alla teoria (computazionale) dei numeri.

Dovendo fattorizzare un numero intero molto grande, il nostro primo obiettivo è quello di determinare se il numero in questione è certamente composito o probabilmente primo.
Ciò abbiamo visto può essre rivelato da un test di primalità.
Supponiamo quindi di sapere che il nostro numero è certamente composito. Come ci muoviamo ora?
La difficoltà della fattorizzazione consiste (in parte) nel fatto che, a prescindere dal provare uno ad uno tutti i fattori primi, non ci sono altri modi ovvi di procedere.

Vedremo che non esiste un unico algoritmo valido per qualsiasi intero da fattorizzare.
Pertanto la tecnica più efficace consisterà nello scegliere di volta in volta l'algoritmo più opportuno, in funzione della taglia del numero da fattorizzare e delle altre informazioni disponibili in merito a tale numero.

§.23 Algoritmo Trial Division

Il miglior punto di partenza è la fattorizzazione per tentativi: usando una lista di numeri primi generata col crivello di Eratostene si tratta semplicemente di verificare se questi dividono l'intero da fattorizzare.
Poichè l'algoritmo prova a dividere il numero n da fattorizzare per tutti gli interi primi inferiori a $\sqrt{n}$, ricordando che $p(n) \sim \frac{n}{\ln n}$ è la funzione che conta i numeri primi inferiori ad n, il calcolo richiederà, nel caso peggiore, $O\left(p\left(\sqrt{n}\right)\right) = O\left(\frac{2\sqrt{n}}{\ln n}\right)$ operazioni aritmetiche ovvero $O\left(2\sqrt{n}\ln^2 n\right) = O\left(2^{\frac{1}{2}\lg n + 2\lg\lg n}\right)$ operazioni binarie.
Seppur più veloci in generale, nessun algoritmo di fattorizzazione noto è efficiente come il trial division nell'individuare fattori primi relativamente piccoli. D'altro canto però, questo algoritmo diventa inutilizzabile per individuare fattori più grandi di 10^7.

Una delle caratteristiche dell'algoritmo trial division è che dato un fattore primo è possibile calcolare esattamente il tempo necessario ad individuarlo.
In altri termini il trial division è un algoritmo completamente deterministico.

Gli altri algoritmi di fattorizzazione più potenti del trial division si basano, come vedremo, su una certa casualità.

In questo caso, dato un fattore primo, potremo fare previsioni sui tempi medi di elaborazione ma non avremo certezza che i tempi effettivi si mantengano vicini ai tempi medi.

Inoltre questa categoria di algoritmi probabilistici ha come peculiarità il fatto che non vi è alcuna garanzia che arrivino al risultato atteso di trovare un fattore primo.

Tutto ciò che sono in grado di fare è di scomporre l'intero in questione in due fattori più piccoli sui quali è possibile effettuare un test di primalità per poi, nel caso il test non venga superato, ripetere nuovamente la ricerca che si conclude quando i fattori individuati risultano probabilmente primi.

Gli algoritmi probabilistici di fattorizzazione si suddividono in due categorie.

La prima è costituita dagli algoritmi "abili a scovare" fattori primi di un numero a partire dai divisori più piccoli.

I loro tempi di elaborazione dipendono più dalla taglia di questo divisore che dalla taglia del numero da fattorizzare.

In questa categoria rientrano ad esempio l'algoritmo Pollard rho che fu utilizzato per fattorizzare il numero di Fermat F_8 ($F_k = 2^{2^k} - 1$), l'algoritmo Pollard p-1, l'algoritmo Williams p+1 e il metodo delle curve ellittiche usato per fattorizzare F_{10} e F_{11}.
Poichè questi algoritmi sono estremamente efficaci per trovare divisori che abbiano tra 7 e 40 cifre decimali dando il meglio di loro per divisori fino a 20 decimali sarebbe opportuno usarli sempre come seconda linea di attacco.

Quando però si tratta di fattorizzare interi costituiti da 100 o più cifre decimali e gli algoritmi della prima categoria non hanno condotto ad alcun risultato, allora è il momento di passare agli algoritmi della seconda categoria, detta Famiglia di Kraitchik.
Di questa famiglia di algoritmi estremamente complessi da analizzare diremo solo che si basano sulla ricerca casuale di coppie (a,b) tali che $a^2 \equiv b^2 \bmod n$.
I tempi di elaborazione sono essenzialmente indipendenti dalla taglia del più piccolo fattore primo ma dipendono invece dalla taglia del numero da fattorizzare.
Per questa ragione è opportuno passare agli algoritmi di questa famiglia solo dopo aver usato al meglio gli algoritmi di trial division e della prima categoria.
D'altro canto, però, la dipendenza dei tempi di elaborazione dalla taglia del numero e non dei suoi fattori fa si che questi algoritmi siano ideali per la ricerca di fattori molto grandi.

§.24 Algoritmo di Fermat

Trial division non è l'unico algoritmo completamente deterministico. Fermat propose un algoritmo la cui peculiarità consiste nella ricerca di fattori a partire da quelli prossimi alla radice quadrata del numero da fattorizzare. Uno dei vantaggi di questo algoritmo è l'assenza di divisioni (tranne una alla fine).

L'idea di base è la seguente: se n è dispari e composito allora $n = ab$. Inoltre osserviamo che se $x = \frac{a+b}{2}$ e $y = \frac{a-b}{2}$ allora $n = ab = (x+y)(x-y) = x^2 - y^2$.

Dunque se riusciamo a scrivere $n = x^2 - y^2$ come differenza di quadrati allora avremo ricavato una fattorizzazione $n = (x+y)(x-y)$.

L'algoritmo procede cominciando con $x = \lceil \sqrt{n} \rceil$ e $y = 0$. Poi, ogni ciclo incrementerà di uno o x o y a seconda che, rispettivamente, sia $x^2 - y^2 < n$ o $x^2 - y^2 > n$ fino all'individuazione della coppia cercata.

§.25 Algoritmo euristico Pollard's rho

Nel 1975 J. M. Pollard pubblicò i primi due algoritmi di fattorizzazione di categoria 1 oggi noti come Pollard p-1 e Pollard rho il secondo dei quali ancora oggi è utilizzato per la ricerca di fattori primi di taglia compresa tra 7 e 20 cifre decimali.

Sia n l'intero composito da fattorizzare. L'algoritmo, innanzi tutto, genera un numero s a caso in $\mathbb{Z}_n$ e su questo costruisce la successione $\{s_i\}$ definita da $s_0 = s$ e $s_i = f_n(s_{i-1}) = (s_{i-1}^2 + 1) \bmod n$.

Da notare che l'algoritmo funziona anche con la ricorrenza definita da $s_i = (s_{i-1}^2 - c) \bmod n$ per tutti i valori $c \in \pm\mathbb{Z}_n$ (anche se i valori $c = 0$ e $c = 2$ dovrebbero essere evitati per ragioni che non approfondiamo qui). Alcuni testi, ad esempio, riportano la successione con $c = -1$.

Per capire il funzionamento dell'algoritmo vediamolo in un caso particolare.

Sia dunque $n = 59153$ e supponiamo di aver generato casualmente il numero $s = 24712$.

Supponiamo per un attimo di sapere che $n = pq$, con $p = 149$ e $q = 349$ fattori primi di n dunque coprimi tra loro, e vediamo come si comporta la successione $s_i \bmod p$:

i	s(i) mod 149	i	s(i) mod 149	i	s(i) mod 149	i	s(i) mod 149
1	38	5	30	9	131	13	119
2	104	6	7	10	27	14	7
3	89	7	50	11	134	15	50
4	25	8	117	12	77	16	117

Come possiamo notare $s_6 \bmod p$ e $s_{14} \bmod p$ sono uguali e poichè ogni termine della successione è completamente determinato dal precedente avremo che

(89) $$s_{i+8} \bmod p = s_i \bmod p \quad \forall i \geq 6.$$

Dunque dopo un certo numero di termini (detti “coda”) la successione modulo p entra in un ciclo.

Ora ritorniamo a quanto ci è noto ovvero alla successione modulo n e osserviamo che la relazione (89) implica che $p \mid s_{i+8} - s_i \quad \forall i \geq 6$.

Per nostra fortuna q, essendo coprimo con p, non divide $s_{i+8} - s_i$ e da ciò possiamo concludere che $\mathrm{mcd}\left(s_{14} - s_6, n\right) = \mathrm{mcd}\left(11927 - 56180,59153\right) = 149$.

Ovviamente, tornando al caso generale, noi non conosciamo nè la lunghezza della coda nè quella del ciclo.

Tutto ciò che fa l’algoritmo quindi, è andare per tentativi e continuare a calcolare prima la differenza tra due termini della successione s_{i_h} e s_k (con $i_h < k \leq i_{h+1}$) e poi l’mcd tra questa differenza e n nella speranza che non sia banale.

Osserviamo che il criterio di scelta dei termini da sottrarre non è unico.

L’importante è che la successione $\{i_h\}$ degli indici sia monotona crescente e tale che $i_{h+1} - i_h \uparrow +\infty$ per garantire da un lato che, poco alla volta, si esca dalla coda e dall’altro che tutti i possibili periodi del ciclo (o multipli del periodo del ciclo) vengano verificati.

Nella nostra implementazione adotteremo $i_h = 2^h$ e inoltre calcoleremo l’mcd ad ogni passo anche se, per velocizzare l‘algoritmo, sarebbe possibile raggruppare più differenze tra loro e calcolare in un solo colpo un unico mcd.

Infine useremo la ricorsione $f_n(x) = \left(x^2 - 1\right) \bmod n$ ottenuta ponendo $c = 1$.

E’ fondamentale osservare che l’algoritmo non stampa mai risultati errati: ogni numero stampato è effettivamente un fattore di n. Però in quanto euristico, non è assolutamente detto che l’algoritmo produca un risultato.

Vi sono due motivi per cui questo algoritmo potrebbe non comportarsi come ci si aspetta.

Innanzi tutto l’analisi euristica del tempo di esecuzione non è rigorosa ed è possibile che il ciclo possa essere più grande del previsto. In questo caso l’algoritmo si comporta correttamente ma molto più lentamente del previsto.

In secondo luogo i divisori di n prodotti da questo algoritmo potrebbero essere solo quelli banali. In tal caso, se necessario, si può rilanciare la procedura con un diverso valore di c nella ricorrenza.

Procedura POLRHO(n)

$i \leftarrow 1$
$x_i \leftarrow \text{RND}(0, n-1)$
$y \leftarrow x_1$
$k \leftarrow 2$
while TRUE ▷ *l'algoritmo non termina mai!*
    $i \leftarrow i+1$
    $x_i \leftarrow \left(x_{i-1}^2 - 1\right) \bmod n$
    $d \leftarrow \text{mcd}\left(y - x_i, n\right)$
    if $d \neq 1 \wedge d \neq n$ **then print** d
    if $i = k$ **then**
        $y \leftarrow x_i$
        $k \leftarrow 2k$

Ora occupiamoci dei tempi di elaborazione.

Supponiamo che sia $n = pq$ con p e q coprimi.
Allora la successione $\{s_i\}$ induce una corrispondente successione $\{s_i'\}$ modulo p dove $s_i' = s_i \bmod p$.
Osserviamo che la successione modulo p è una versione "ridotta" della successione originale modulo n (e altrettanto dicasi per la successione modulo q). Infatti:

$$\begin{aligned} s_{i+1}' &= s_{i+1} \bmod p \\ &= f_n(s_i) \bmod p \\ &= \left(\left(s_i^2 + 1\right) \bmod n\right) \bmod p \\ &= \left(s_i^2 + 1\right) \bmod p \\ &= \left(\left(s_i \bmod p\right)^2 + 1\right) \bmod p \\ &= \left(\left(s_i'\right)^2 + 1\right) \bmod p \\ &= f_p(s_i') \end{aligned}$$

Quindi, anche se non stiamo effettivamente calcolando la successione indotta, questa è ben definita e e segue le stesse regole della successione iniziale.

Ci sono valide ragioni (certamente note a chi ha dimestichezza col calcolo delle probabilità e conosce il paradosso dei compleanni) per aspettarsi che se esiste un divisore p allora le lunghezze del ciclo e della coda modulo p saranno pari a $\Theta\left(\sqrt{p}\right)$.

In altri termini ci potremo ragionevolmente attendere una ripetizione dopo $\Theta\left(\sqrt{p}\right)$ passi ovvero che l'algoritmo stampi un fattore p di n approssimativamente dopo $\sqrt{p}$ iterazioni del ciclo while.

Se p è piccolo rispetto a n la successione modulo p potrà dunque ripetersi molto più rapidamente di quella modulo n.

Di conseguenza possiamo attenderci che l'algoritmo individui sufficienti divisori per garantire la completa fattorizzazione di n dopo circa $n^{\frac{1}{4}}$ aggiornamenti poichè ogni fattore primo di n, eccetto eventualmente il più grande, è inferiore a $\sqrt{n}$.

Complessivamente, dunque, l'algoritmo richiederà $O\left(2^{\frac{1}{4}\ln n+2\lg\lg n}\right)$ operazioni binarie.

Concludiamo l'argomento precisando in che modo l'algoritmo può fallire presentando solo divisori banali di n.

Osserviamo che $p \mid s_i - s_{i+k}$ e $q \mid s_i - s_{i+k}$ implicano $pq \mid s_i - s_{i+k}$ e, di conseguenza $\mathrm{mcd}\left(s_i - s_{i+k}, n\right) = pq = n$.

Ciò capita costantemente se le lunghezze di coda e ciclo delle successioni modulo p e modulo q sono identiche.