La formula

Nell’esempio che segue utilizzerò variabili elevate a potenza per rendere più chiaro il concetto e facilitare la comprensione. Inoltre, quando presenterò esempi con numeri molto grandi, questo approccio permetterà di esprimerli in forma più compatta e leggibile.

L’equazione assumerà quindi la seguente forma:

- **a** elevato a un esponente Ae.
- **b** elevato a un esponente Be.

*Il prodotto **n** verrà calcolato aggiungendo 1 ai coefficienti*

L'aggiunta di 1 ai due coefficienti è fondamentale. Senza questa aggiunta verrebbe restituito il valore zero nella ricerca dell’intervallo

$$n=\left(a^{Ae}+1\right)*\left(b^{Be}+1\right)$$

*La chiave, stando a quanto scritto sopra, verrà presa sottraendo 1 al coefficiente **b***

$$c = b^{Be} - 1$$

Questa chiave, e il prodotto calcolato ***n****, ci forniranno un intervallo numerico nel quale poter eseguire la fattorizzazione*

$$I = [\frac{c}{n \bmod c}] * 2$$

Per capire meglio cosa significa questo intervallo, pensatelo come a un "contenitore" di numeri entro cui la chiave ci permette di trovare facilmente due valori distinti, uno per x e uno per y, senza doverli cercare uno per uno. ***Questo meccanismo funziona sempre allo stesso modo, ma è importante sottolineare che l'intervallo cresce in modo esponenziale all'aumentare dei valori dei due coefficienti, rendendo la ricerca ancora più efficiente per numeri più grandi.*** *Sebbene non conosco ancora esattamente il motivo matematico di questa regolarità, questa scoperta apre la strada a nuove domande e potenziali sviluppi interessanti.*

Dopo aver definito l'intervallo **I***, possiamo chiedere al computer di selezionare, in modo* ***totalmente casuale e indipendente****, un numero entro il limite del suo valore da assegnare a x e y.*

Questo può essere fatto in due modi:

1. **Utilizzando funzioni separate** per studiare il comportamento della distribuzione:
   x=RND(1,I)
   y=RND(1,I)

$$n=\left(a^{Ae}+x\right)*\left(b^{Be}+y\right)$$

2. **Utilizzando una forma compatta**, utile quando interessa solo il risultato finale

$$n=\left(a^{Ae}+RND\left(1,I\right)\right)*\left(b^{Be}+RND\left(1,I\right)\right)$$

A questo punto **n,** che prima rappresentava solo il prodotto dei due coefficienti noti incrementati di 1, ora rappresenta il prodotto dei due coefficienti noti, **a** e **b**, a cui si sommano le due incognite **x** e **y**. L'equazione potrà essere risolta, tramite la proprietà della chiave, per qualsiasi numero uscito da x e y, a partire da **n**

$$MCD\left(n,n\bmod c\right)=a^{Ae}+x$$

Il calcolo di **n** appena eseguito fornisce un numero ai due coefficienti che non sappiamo se sono numeri primi o composti. Ciò significa che, sebbene questa struttura sia **accettabile nel contesto delle equazioni diofantee**, non è sufficiente quando vogliamo lavorare **esclusivamente con numeri primi**. Infatti, le incognite **x** e **y** non assicurano che le somme **a+x** e **b+y** siano effettivamente numeri primi.

Per risolvere questo problema, introduciamo un nuovo elemento, affinando il nostro approccio in modo che il risultato di queste somme corrisponda esattamente a due numeri primi.

introduciamo la lettera **S** per indicare che il valore che stiamo per calcolare è un numero ***semiprimo***, ovvero un numero composto ottenuto dal prodotto di due numeri primi, che chiameremo **p** e **q**.

Utilizzando questa nuova notazione e supportandoci con algoritmi probabilistici, come il test di Miller-Rabin, possiamo ora formulare la seguente equazione:

$$p = Next\ Prime\left(a^{Ae} + RND\left(1, I\right)\right)$$

$$q = Next\ Prime\left(b^{Be} + RND\left(1, I\right)\right)$$

$$S = \left(p * q\right)$$

La funzione ***NextPrime*** individua il numero primo immediatamente successivo alla somma dei coefficienti noti alle loro rispettive incognite **x** e **y**. Questo garantisce, con una probabilità prossima al 100%, che i due numeri ottenuti siano primi.

Nota: ✅

La funzione `nextprime(x),nextprime(y)` restituisce **il primo numero primo maggiore del numero dato**, ma **non ti dice nulla su quanto vicino sia al numero primo successivo o precedente**.

In altre parole:

- I **numeri primi** non sono equidistanti tra loro.
- Non esiste una regola semplice che dica: "Ogni numero primo dista 2 o 6 o 10 unità da quello precedente o da quello successivo".
- Quindi:

  `nextprime(numero dato)` ti dà il **prossimo numero primo dopo il numero dato**, ma **non garantisce** che sia il **più vicino possibile a `n` in termini assoluti** (es. non è detto che sia distante solo +2 o +4, potrebbe anche essere distante +10, +50, +1000...).

Utilizzando la stessa funzione di MCD, utilizzata sopra, otterremo lo stesso risultato ma stavolta riuscendo a fattorizzare un numero semiprimo

$$MCD\left(S, S \bmod c\right) = p$$

Questo dimostra che la proprietà della chiave non distingue tra numeri composti in generale e semiprimi, che pur essendo anch'essi composti, hanno esattamente due fattori primi. In entrambi i casi, la difficoltà di fattorizzazione rimane invariata, poiché la chiave si applica con la stessa efficacia, indipendentemente dalla struttura interna del numero.

Prima di proseguire con un esempio reale, numerico, vorrei spendere due parole sulla funzione MCD

(Massimo Comune Divisore)

Per chi non è familiare con questo tipo di operazione, il Massimo Comune Divisore (MCD) è un concetto fondamentale in matematica che viene utilizzato per trovare il più grande

divisore comune tra due numeri. Esistono diversi metodi per calcolare MCD, uno dei quali è l'algoritmo **euclideo**, noto per la sua efficienza, specialmente con numeri grandi.

A differenza del metodo tradizionale, che richiede la fattorizzazione completa di ogni numero nei suoi fattori primi, l'algoritmo euclideo utilizza un sistema di divisione modulare.

Ecco come funziona:

1. **Passo Iniziale**:
    - Si prendono due numeri, A e B.
2. **Calcolo del Resto**:
    - Si esegue la divisione modulare: R=A Mod (B)
    - Questo calcolo restituisce il resto R della divisione di A per B.
3. **Aggiornamento dei Valori**:
    - Se R è maggiore di zero, si aggiornano i valori: A assume il valore di B, e B assume il valore di R,
4. **Ripetizione**:
    - Questo processo viene ripetuto fino a quando il resto R non diventa zero.

5. **Risultato**:
    - Quando R è zero, il valore corrente di B è il massimo comune divisore

Vantaggi dell'Algoritmo Euclideo

Questo tipo di confronto è molto veloce e particolarmente adatto per numeri grandi perché evita la complessità della fattorizzazione completa. Invece di cercare tutti i divisori di entrambi i numeri fino a raggiungere i loro fattori primi, l'algoritmo punta direttamente al divisore comune, rendendolo estremamente efficiente. L'algoritmo euclideo MCD è una tecnica potente e rapida per determinare il massimo comune divisore tra due numeri. La sua capacità di gestire grandi numeri senza la necessità di una fattorizzazione completa lo rende ideale per applicazioni che richiedono efficienza e velocità come nel caso dell'algoritmo GC57

Passiamo ora a un esempio numerico di ciò che ho mostrato prima

Cercherò di utilizzare dei numeri piccoli, per quanto possibile, per rendere l'esempio comprensibile senza che l'occhio si perda in una marea di numeri.

Impostiamo i coefficienti a e b, con questi valori:

- a=13
- b=19
- Ae=14
- Be=20

Con lo stesso sistema appena descritto ma con valori numerici reali, otteniamo

$n=(13^{14}+1)*(19^{20}+1)=$

148005873830804548381842924327290408922580

$c=19^{20}-1 = 375899734575459581933556 00$

$I=[\frac{c}{n \bmod c}]*2 = 9546959644$

Prendiamo da questo intervallo le due incognite con la funzione random.

x=RND(1,I) = 571736589

y=RND(1,I) = 6164196145

Individuando nel proseguo i due numeri primi

p=NextPrime(13^{14}+571736589) = 3937376957435893

q=NextPrime(19^{20}+6164196145) =

375899734575459643575517 99

Ottenendo il semiprimo

$S=(p\cdot q)$ =

1480058953223683041300978453582019693215 07

Applicando la proprietà della chiave, questa individuerà all'istante il fattore primo ***p***

$MCD(s, s \bmod c)=p$ = 3937376957435893

Riporto di seguito alcuni esempi presi dal mio programma tester, che ogni volta che entra in esecuzione seleziona in modo casuale le incognite x e y. Gli esempi riportati mostrano che la chiave è sempre la medesima, e così anche l'intervallo, mentre

il semiprimo creato è sempre diverso. Viene poi mostrato il fattore primo trovato, mentre i due **True** che compaiono certificano che entrambi i divisori trovati sono numeri primi.

---

chiave 19^{20}-1 = 375899734575459581933556 00

I = 9546959644

Test divisore 1: True

Test divisore 2: True

divisore : 3937379268972229

Semiprimo analizzato:

1480059822129578054864364881252532270644 17

---

chiave 19^{20}-1 = 375899734575459581933556 00

I = 9546959644

Test divisore 1: True

Test divisore 2: True

divisore : 3937384951430207

Semiprimo analizzato:

1480061958164023772502222999225921782781 29

---

chiave 19^{20}-1 = 375899734575459581933556 00

I = 9546959644

Test divisore 1: True

Test divisore 2: True

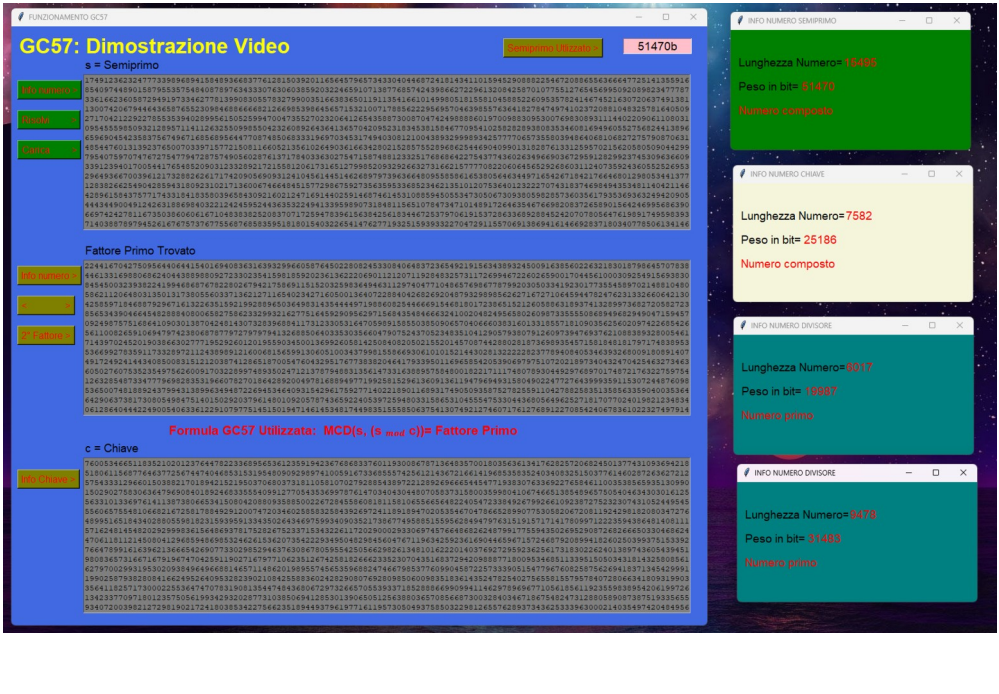
divisore : 3937380346250413

Semiprimo analizzato:

1480060227078161674365113117205742675783 13

Grandi Numeri

Come ho già accennato all'inizio, questa proprietà agisce su numeri a lunghezza arbitraria, mantenendo la stessa velocità di esecuzione indipendentemente dalla dimensione dei numeri utilizzati, purché vengano rispettate le specifiche illustrate. Per dimostrare questa capacità, introdurrò uno screenshot che mostra l'algoritmo GC57 alle prese con numeri di dimensioni estreme. Si tratta di numeri talmente grandi che, se dovessi stamparli per intero, occuperebbero molte pagine, rendendo difficile seguire il ragionamento.



Per garantire la massima chiarezza, riporto qui le caratteristiche dei numeri coinvolti:

- Semiprimo: Lunghezza 15495 cifre (51470 bit)
- Chiave: Lunghezza 7582 cifre (25186 bit)
- Numero primo p: Lunghezza 6017 cifre (19987 bit)
- Numero primo q: Lunghezza 9478 cifre (31483 bit)

Faccio una piccola parentesi per mostrare la chiave che ho utilizzato in alcuni dei mie programmi di criptazione e, come ho già accennato, questo tipo di chiave, più piccola rispetto a ***B-1,*** *produce un intervallo più piccolo. L'ho sperimentata su semiprimi non molto grandi, 8.000, 13.000 bit, perché quando si va su semiprimi oltre i 14.000 bit, l'operazione logaritmica comincia a dare dei problemi di memoria. Al contrario, la chiave* ***B-1****, non dà nessun problema anche su semiprimi oltre i 20.000 bit*

$$n = a^{Ae} * b^{Be}$$

$$Ec = [\frac{(\log_b(n))}{2}]$$

$$c = b^{Ec}$$

$$n = (a^{Ae} + 1) * (b^{Be} + 1)$$

$$I = [\frac{c}{n \bmod c}]$$

*La chiave in questo caso viene presa da **b,** elevato alla potenza intermedia del logaritmo a base **b** di **n. (Le parentesi quadre rappresentano l'intero della divisone)***

Vediamo un esempio numerico:

(coefficiente b)

907772904243865444884463929857279843314925894581

(Esponente Ec)

39

Questa chiave b^{39} risolve dei semiprimi di 12.464 bit, numeri composti da 3.753 cifre, il cui intervallo di risoluzione è di 2^{1715}

I Contenitori: Gli Scomparti degli Intervalli

Passiamo ora a un concetto fondamentale: gli scomparti, ovvero contenitori di intervalli.

Abbiamo già definito l'intervallo come la regione numerica in cui GC57 è in grado di fattorizzare senza alcuna implementazione aggiuntiva. Ora, possiamo immaginare questi intervalli come contenitori organizzati in ordine crescente.

Struttura dei Contenitori

- *Ogni contenitore occupa una posizione precisa nella sequenza numerica.*
- *Il primo intervallo parte da zero e si estende fino al valore massimo calcolato.*
- *Il secondo intervallo parte dalla fine del primo e si estende del valore trovato, e così via.*
- *L'espansione degli intervalli non è infinita, ma è limitata dalla grandezza del semiprimo.*
- *La fine dell'espansione è determinata dalla relazione tra a e b, e dalla loro distanza in cifre.*

Per visualizzare meglio questo concetto, immaginiamo la seguente struttura dove I è l'intervallo trovato:

Intervallo 0	Intervallo 1	Intervallo 2	Intervallo 3
Da 0 a I	Da I a (I*2)	Da (I*2) a (I*3)	Da (I*3) a (I*4)

Con l'esempio che segue proverò a semplificare il concetto di "contenitore" :

Supponiamo ora di aver trovato un intervallo di 2^{100}. Secondo la tabella riportata sopra, il contenitore (intervallo 0), conterrà una numerazione da 0 a 2^{100}, essendo 0 un numero nullo, diciamo da 1 a 2^{100}. Di conseguenza i contenitori successivi conterranno la stessa quantità numerica, che sarà da 2^{100} a $2^{100}*2$, e così via.

Introduciamo ora il parametro **K**, che consente di accedere a un **contenitore specifico**, indipendentemente da dove si trovi nella sequenza numerica.

Se vogliamo fattorizzare numeri **all'interno di un contenitore diverso da quello iniziale**, basta modificare leggermente l'algoritmo, indicando il contenitore desiderato.

Innanzi tutto devo determinare quanto è grande l'intervallo del contenitore zero, e questo calcolo rimane invariato

$$I = [\frac{c}{n \bmod c}] * 2$$

Eseguo la ricerca dei numeri primi in questo contenitore

$$p = Next\ Prime(a^{Ae} + RND(I*3, I*4))$$

$$q = Next\ Prime(b^{Be} + RND(I*3, I*4))$$

$$S = (p*q)$$

Eseguo la proprietà della chiave, aggiungendo il contenitore, facendomi restituire il fattore primo ***p***

$$MCD\left(S,\left(S\,mod\,c\right)+c*k\,3\right)=p$$

Perché i Contenitori Sono Importanti?

L'introduzione degli **scomparti** estende **drasticamente** la capacità della proprietà, permettendo:

- **Una gestione più ampia delle equazioni diofantee.**
- **Una maggiore sicurezza nell'uso di numeri primi per la crittografia.**
- **Un nuovo livello di studio sulla distribuzione dei numeri primi.**

Conclusione

Il metodo presentato in questo saggio non nasce da teorie accademiche complesse, ma da un'**intuizione logica e sperimentale**, maturata attraverso l'esperienza diretta con i numeri e la programmazione.
Il cuore del sistema non è affidato a un algoritmo pubblicamente noto, ma a una **combinazione unica di**

semiprimi pre-generati e **chiavi personali custodite in modo fisico** (tramite una pen drive USB nominativa), rendendo **praticamente impossibile decifrare un messaggio senza il possesso della chiave originale**.

Questa struttura, pur nella sua semplicità, permette di creare un sistema di cifratura:

- **Autonomo** (nessun server centrale)
- **Distribuito** (chiavi gestite individualmente)
- **Flessibile** (adattabile a vari livelli di sicurezza)
- **Sicuro** (basato sulla conoscenza privata dei fattori primi)

La possibilità di studiare e analizzare liberamente il codice, unita alla gestione concreta delle chiavi, fa di questo approccio **uno strumento didattico, pratico e innovativo**, adatto a chi vuole approfondire davvero il legame tra **logica, matematica e sicurezza**.

Non serve essere esperti per capire tutto subito. Serve solo curiosità, pazienza e il desiderio di andare in profondità.

Licenza e condizioni d’uso

Il presente saggio, comprensivo della formula GC57, è distribuito liberamente per finalità di studio, ricerca e

divulgazione. La formula GC57 è concessa in forma **open source**, ma resta vincolata alle seguenti condizioni irrevocabili:

- **Non è consentita alcuna modifica** al contenuto originale, né parziale né totale.
- **È obbligatoria la citazione dell'autore** ("Claudio Govi") e della formula **GC57** in ogni forma di utilizzo, sia essa pubblica, privata, accademica, sperimentale o editoriale.
- **È vietato attribuire altro nome** o marchio alla formula o alle sue implementazioni: essa può essere menzionata solo come **"GC57"**.
- L'opera **non è soggetta a copyright restrittivo**: può essere condivisa, stampata o distribuita, **anche a pagamento**, purché vengano rispettate le condizioni sopra esposte.
- **Qualsiasi uso commerciale o accademico** deve comunque mantenere il riferimento integrale alla formula GC57 e al suo autore.