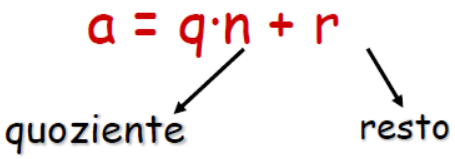
7.0 ARITMETICA MODULARE

Dati a ∈ Z, n ∈ Z esiste un'unica coppia (q,r) con $0 \leq r \leq n-1$ tale che:



r è indicato con **a mod n**, significa che il valore a lo divido per n, ottengo un quoziente q e ottengo un resto. **a mod n** è il resto della divisione per n

Vengono mostrati degli esempi:

a=11, n=7, 11 = 1·7 + 4 → r = 11 mod 7 = 4

a=-11, n=7, -11 = (-2)·7 + 3 → r = -11 mod 7 = 3

a e b sono congruenti mod n (a≡b mod n) se: a mod n = b mod n, e cioè che questi due valori sono congrui modulo n se hanno lo stesso resto una volta divisi per n. Ad esempio:

$73 \equiv 4 \bmod 23$

- 73 mod 23 = 4 mod 23

$21 \equiv -9 \bmod 10$

- 21 mod 10 = -9 mod 10

7.1 L'INSIEME Zn

Z_n sono tutti i numeri che vanno da 0 a n-1 → Z_n={0,.., n-1}. In realtà questi valori possono essere interpretati come resti delle divisioni di numeri per un fissato n. Ad esempio 0, non è il singolo valore 0 ma è come se fosse l'insieme di tutti i valori che danno, diviso un certo n, resto 0.

Esempio:

Esempio: $Z_4 = \{ [0]_4, [1]_4, [2]_4, [3]_4 \}$

- $[0]_4$={…, -12, -8, -4, 0, 4, 8, 12, …}
- $[1]_4$={…, -11, -7, -3, 1, 5, 9, 13, …}
- $[2]_4$={…, -10, -6, -2, 2, 6, 10, 14, …}
- $[3]_4$={…, -9, -5, -1, 3, 7, 11, 15, …}

Se considero Z_4, i valori sarebbero 0,1,2,3. Il valore 0 non è niente altro che l'insieme di tutti i valori mostrati nell'immagine. Tutti questi valori, quando divisi per 4, danno valore 0 come resto della divisione. Dunque il resto 0, rappresenta tutti i valori che hanno quel resto. Per gli scopi pratici Z_n è semplicemente l'insieme di valori che vanno da 0 a n-1.

Su questi valori, posso andare a definire 2 operazioni: **Addizione e Moltiplicazione.**

7.1.0 ARITMETICA MODULARE

L'addizione modulo n è definita come: (a mod n) + (b mod n) = (a+b) mod n

Ad esempio si consideri l'operazione di addizione in modulo 8, sulle righe e sulle colonne ci sono tutti i numeri rappresentabili in Z_8, ossia da 0 a 7. Naturalmente le addizioni, vengono effettuate in mod 8.

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6

Naturalmente l'addizione modulare gode di proprietà commutativa, associativa, identità (in questo caso 0) ed esistenza inversa additiva (l'inversa additiva di x è y tale che $x+y \equiv 0 \bmod n$). Questo implica che (Z_n, +) è un gruppo additivo mod n.

7.2 MASSIMO COMUNE DIVISORE (gcd)

Definita nel seguente modo: d è il **massimo comune divisore** di a e n se:

- d è un divisore di a e n.
- Ogni divisore di a e n è un divisore di d.

Un'altra definizione possibile per il massimo comune divisore è che è il più piccolo intero positivo che può essere scritto nella forma **d= a × x + n × y**

Gode di varie proprietà ma quella per noi importante è che se gcd(a,n)=1, a e n sono **relativamente primi**, dunque non hanno fattori comuni aldilà di 1.

7.3 L'INSIEME Zn*

Corrisponde agli elementi che sono relativamente primi rispetto al modulo.

Ad esempio:

Esempio: $Z_4^*=\{ [1]_4, [3]_4\}$

- $[1]_4=\{..., -11, -7, -3, 1, 5, 9, 13, ...\}$
- $[3]_4=\{..., -9, -5, -1, 3, 7, 11, 15, ...\}$

Esempio: $Z_8^*=\{ [1]_8, [3]_8, [5]_8, [7]_8\}$

- $[1]_8=\{..., -23, -15, -7, 1, 9, 17, 25, ...\}$
- $[3]_8=\{..., -21, -13, -5, 3, 11, 19, 30, ...\}$

In Z_4^* sono presenti 1 e 3, poiché il gcd(2,4) è 2 e il gcd(4,4) è 4.

In Z_8^* sono presenti 1,3,5,7 per motivi analoghi al caso precedente.

La moltiplicazione modulo n, definita come: (a mod n) × (b mod n)= (a × b) mod n, gode delle stesse proprietà dell'addizione: commutatività, associatività, distributività, identità, esistenza inversa moltiplicativa (l'inversa moltiplicativa di x è y tale che x × y ≡ 1 mod n). Questo implica che $(Z_n^*, \times)$ è un gruppo moltiplicativo mod n. Si nota come sia stato inserito Z_n^* e analizziamo ora il perché, analizzando l'aritmetica mod 8:

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6

Addizione

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	4	7	2	5
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

Moltiplicazione

Analizziamo la tabella moltiplicativa del 3, si osserva che ha l'inverso moltiplicativo (ossia un numero che moltiplicato per 3, da risultato 1 in mod 8), ed è proprio se stesso. Invece il 2 non ha un inverso moltiplicativo, in quanto non è relativamente primo rispetto ad 8. Il 5 ha un inverso moltiplicativo, il 6 non ha un inverso moltiplicativo. Da tutto ciò ne deriva che:

Non tutti gli interi in Z_8 hanno inversa moltiplicativa, ma quelli in $Z_8^*= \{ [1]_8 , [3]_8 , [5]_8 , [7]_8 \}$ ce l'hanno, ossia quei numeri che sono relativamente primi rispetto al modulo, cioè 8.

Viene mostrato un ragionamento analogo per la moltiplicazione in mod 7, dove $Z_7^*= \{ [1]_7 , [2]_7 , [3]_7 , [4]_7 , [5]_7 , [6]_7\}$: poiché in Z_7^* sono presenti 1,2,3,4,5,6, allora i seguenti numeri hanno un'inversa moltiplicativa.

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

Vediamo ora degli algoritmi per il calcolo del Massimo Comune Divisore:

7.4 ALGORITMO DI EUCLIDE

Non avendo i valori in forma fattorizzata, si può applicare il seguente algoritmo, che non dipende dalla fattorizzazione dei numeri.

L'idea di questo algoritmo è legata al **Teorema della ricorsione del gcd** che dice che: Per tutti gli intero a≥0 e n>0 gcd(a,n) = gcd(n, a mod n). Questo procedimento mi consente di fare il gcd di due valori che sono più piccoli rispetto a quelli a sinistra dell'uguale.

Da questa idea deriva l'algoritmo di Euclide:

Euclide(a,n)
if n=0 then return a
else return **Euclide(n, a mod n)**

Esaminiamo ora qualche esempio:

Euclide (30, 21) = Euclide (21, 9)
= Euclide (9, 3)
= Euclide (3, 0) = 3

Euclide (4864, 3458) = Euclide (3458, 1406)
= Euclide (1406, 646)
= Euclide (646, 114)
= Euclide (114, 76)
= Euclide (76, 38)
= Euclide (38, 0) = 38

Quanto tempo occorre per questo algoritmo? Il numero massimo di chiamate ricorsive che si può fare è log n. Analizzando l'RSA, considerando che al giorno d'oggi si ha a che fare con numeri di 2048 bit, questo implica che con l'algoritmo di euclide si fanno ben 2048 chiamate ricorsive al massimo.

Essendo un valore relativo alle chiamate, chiaramente deve essere fatto il modulo che viene fatto per ogni singola chiamata in tempo O ((log a)^2) operazioni su bit. Al massimo si hanno O ((log a)^3) operazioni su bit.

Esiste una versione dell'algoritmo detta **Algoritmo di Euclide Esteso**, in cui, oltre a computare d= gcd(a,n) computa anche due interi x e y, tali che **d= gcd(a,n) = a*x+n*y**, avendo lo stesso running time asintotico di **Euclide(a,n)**.

Euclide-esteso (a,n)
if n = 0 then return (a, 1, 0)
(d´, x´, y´) ← **Euclide-esteso** (n, a mod n)
(d, x, y) ← (d´, y´, x´- ⌊a/n⌋ y´)
return (d, x, y)

Ad esempio:

3= gcd(99,78) = -11*99+14*78, si osserva quindi che viene calcolato il gcd che è 3, ma contemporaneamente vengono calcolati i due valori x e y,

Vediamo ora come calcolare l'inversa moltiplicativa, importante nell'RSA in quanto, una volta calcolato ***e***, bisogna calcolare ***d***, tale che ***e*******d*** ≡ 1 mod (***e***-1)(***d***-1)

Analizziamo questo caso: ax≡1 (mod 8), si osserva come abbiamo il valore a, abbiamo il modulo e si vuole calcolare il valore x, tale che ax≡1 mod 8. Si vuole dunque calcolare l'inversa moltiplicativa di a.

Si riesce a trovare una soluzione se e solo se **gcd(a,8)=1.** Quindi, analizzando la tabella in Z_8, se ho il valore 3, allora il valore x risultante sarà 3.

Si osserva che se si fa girare l'algoritmo di Euclide esteso, e cioè: **Euclice-estero(a,n)** → 1=a*x' + n*y, genera il gcd e contemporaneamente da x' e uno y interi tale che vale l'equazione. Una volta che si ha questa equazione si può calcolare l'inversa moltiplicativa di a mod n, cioè ax≡1 (mod n), che è proprio x'. La complessità di questa operazione è la stessa del Massimo Comune Divisore.

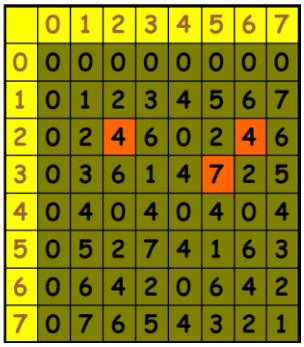
	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	4	7	2	5
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

Esempi:

3x≡7 (mod 8) soluzione 5

2x≡4 (mod 8) soluzione 2,6

4x≡2 (mod 8) nessuna soluzione



Per verificare se ci sono soluzioni, l'idea è la seguente: effettuo l'algoritmo di **Euclide-Esteso(a,n)** –> g= a*x' + n*y. Ricordando l'equazione che voglio soddisfare e cioè ax≡b (mod n), se g divide b allora ci sono esattamente g distinte soluzioni mod n. Se g non divide b allora non ci sono soluzioni.

7.5 TEOREMA CINESE DEL RESTO (SOLO LETTURA)

Dati:

- $M_1, m_2, \ldots, m_t$ interi positivi tali che gcd $(m_i, m_j) = 1$, $i \neq j$
- $M = m_1 * m_2 * \ldots * m_t$
- $a_1, a_2, \ldots, a_t$ interi

Allora esiste una sola soluzione modulo M al sistema di congruenze

$$\begin{cases} X \equiv a_1 \bmod m_1 \\ X \equiv a_2 \bmod m_2 \\ \ldots \\ X \equiv a_t \bmod m_t \end{cases} \qquad X = \sum_{i=1}^{t} a_i \cdot M_i \cdot y_i \bmod M$$

$M_i = M/m_i \qquad y_i = M_i^{-1} \bmod m_i$

7.6 ELEVAZIONE A POTENZA MODULARE

Consente il calcolo di x^y mod z. Esistono 3 modi per effettuare il seguente calcolo (Si noti che nell'RSA questa è un'operazione che viene fatta per cifrare e decifrare il messaggio):

7.6.1 ELEVAZIONE A POTENZA MODULARE – METODO NAIVE

Per fare un'elevazione a potenza, viene moltiplicato il numero per se stesso un numero finito di volte in modulo z.

Calcolo di x^y mod z:

```
Potenza_Modulare_naive (x, y, z)
a ← 1
for i = 1 to y do
        a ← (a * x) mod z
return a
```

Ad esempio se y vale 5, moltiplico il mio a per se stesso per 4 volte, questo mi consente di calcolare un numero a, elevato alla 5.

Un metodo di questo genere nell'RSA implicherebbe la seguente situazione: Se ad esempio y è di 512 bit, occorrono circa 2^{512} operazioni che è un numero stratosferico, cioè non riuscirei mai a risolvere il problema. Questo metodo è perciò adatto per numeri piccoli.

Per compensare questi problemi, si usano 2 metodi, in cui si usa la rappresentazione binaria di y, e, sfruttando le proprietà tipiche dell'esponenziazione, si riesce a calcolare x^y mod z in tempo proporzionale alla lunghezza di y. Questi due metodi sono: **left-to-right** in cui l'algoritmo procede dai bit più significativi ai meno significativi e viceversa, che corrisponde al metodo **right-to-left**.

7.6.2 ELEVAZIONE A POTENZA MODULARE – METODO LEFT-TO-RIGHT

Si procede dai bit più significativi ai bit meno significativi

L'idea è che se si vuole calcolare x^y mod z, posso scrivere y nel seguente modo: $y=y_0 2^0 + y_1 2^1 + \dots + y_t 2^t$ cioè come rappresentazione binaria. L'idea è di poter scrivere y in questa forma: $y = y_0+2(y_1+ 2(y_2+\dots +2(y_{t-1}+2y_t)))))$, cioè, prima si effettua $2y_t + y_{t-1}$. Questo valore viene moltiplicato per 2 e gli aggiungo y_{t-2} e così via. Quindi ogni volta aggiungo il precedente, moltiplico per 2 etc… fino all'inizio.

Ad esempio:

$$40 = 0\cdot 2^0 + 0\cdot 2^1 + 0\cdot 2^2 + 1\cdot 2^3 + 0\cdot 2^4 + 1\cdot 2^5$$
$$40 = 0 + (2\,(0 + 2\,(0 + (2\,(1 + 2\,(0 + 2\cdot 1))))))$$

Il 40 può essere scritto come somma di potenze di 2 e successivamente scrivendolo nella forma descritta precedentemente. Naturalmente la formula restituirà 40.

Ritorniamo alla seguente forma scritta in precedenza: $y = y_0+2(y_1+ 2(y_2+\dots +2(y_{t-1}+2y_t)))))$, questa può essere scritta, sfruttando la proprietà degli esponenziali nei seguenti modi:

$$y = y_0+2(y_1+ 2(y_2+\dots +2(y_{t-1}+2y_t)))))$$

$$\begin{aligned} x^y &= x^{y_0+2(y_1+2(y_2+\dots+2(y_{t-1}+2y_t)))))} \\ &= x^{y_0} x^{2(y_1+2(y_2+\dots+2(y_{t-1}+2y_t)))))} \\ &= x^{y_0}\left(x^{y_1+2(y_2+\dots+2(y_{t-1}+2y_t)))))}\right)^2 \\ &= x^{y_0}\left(x^{y_1}\left(x^{y_2+\dots+2(y_{t-1}+2y_t)))))}\right)^2\right)^2 \\ &= x^{y_0}\left(x^{y_1}\left(\dots\left(x^{y_{t-1}}\left(x^{y_t}\right)^2\right)^2\dots\right)^2\right. \end{aligned}$$

Fino a che assume l'ultima forma, che ci è utile per mostrare questo esempio:

Esempio: x^{40}

$$40 = 0\cdot 2^0 + 0\cdot 2^1 + 0\cdot 2^2 + 1\cdot 2^3 + 0\cdot 2^4 + 1\cdot 2^5$$
$$40 = 0 + (2\,(0 + 2\,(0 + (2\,(1 + 2\,(0 + 2\cdot 1))))))$$
$$x^{40} = x^0\ (\ x^0\ \ (x^0\ \ \ (x^1\ \ (x^0\,(x^1)^2)^2)^2)^2)^2$$

Esprimiamo 40 come mostrato nel precedente esempio, e da questa, ricavo x^{40} tenendo conto della proprietà descritta precedentemente. Per calcolare x^{40} con quest'ultima formula, svolgo le operazioni dalla parentesi più interna cioè x^1 che moltiplicato per $x^0=1$ è come se non ci fosse e quindi non li considero. Continuo i quadrati, fino ad ottenere x^{40}.

La complessità di questo algoritmo dipende da quante volte devo effettuare il quadrato, che dipende dalla lunghezza dell'esponente y.

L'algoritmo è il seguente:

```
Potenza_Modulare (x, y, z)
a ← 1
for i = t downto 0 do
      a ← (a * a) mod z
      if y_i = 1 then a ← (a · x) mod z
return a
```

Se y è di 512, occorrono circa 512 operazioni e no 2^{512} come prima, quindi risulta essere efficiente per l'elevazione a potenza.

7.6.3 ELEVAZIONE A POTENZA MODULARE – METODO RIGHT-TO-LEFT

Si procede dai bit meno significativi a quelli più significativi.

L'idea, anche in questo metodo, è che se si vuole calcolare $x^y \bmod z$, posso scrivere y nel seguente modo: $y=y_0 2^0 + y_1 2^1 + \dots + y_t 2^t$ cioè come rappresentazione binaria. X^y, con una serie di ragionamenti logici può essere espresso nei seguenti modi:

$$x^y = x^{2^0 y_0 + 2^1 y_1 \dots + 2^t y_t}$$
$$= x^{2^0 y_0} \cdot x^{2^1 y_1} \cdot \dots \cdot x^{2^t y_t}$$
$$= (x^{2^0})^{y_0} \cdot (x^{2^1})^{y_1} \cdot \dots \cdot (x^{2^t})^{y_t}$$

Viene esplicitata la rappresentazione binaria di y, e utilizzando le proprietà dell'elevazione a potenza, ogni volta che c'è il + all'esponente, equivale a un prodotto. Successivamente, l'ultimo step, scrivo i vari prodotti in maniera un po' diversa, cioè come un doppio elevamento, e così via fino alla fine.

Facciamo un esempio di calcolo:

Esempio: x^{40}

$$40 = 0\cdot 2^0 + 0\cdot 2^1 + 0\cdot 2^2 + 1\cdot 2^3 + 0\cdot 2^4 + 1\cdot 2^5$$
$$x^{40} = (x^1)^0 \cdot (x^2)^0 \cdot (x^4)^0 \cdot (x^8)^1 \cdot (x^{16})^0 \cdot (x^{32})^1$$

40 viene scritto in rappresentazione binaria da cui ne deriva la rappresentazione di x^{40}, con la formula precedente, in cui gli 0 e gli 1 come esponenti sono dati dai coefficienti dei prodotti. Per calcolare questa espressione, devo calcolare prima i contenuti interni e poi elevarli a 0/1 a seconda di com'è rappresentato. Osservo in questo calcolo che, per calcolare un valore x, mi basta prendere il precedente ed elevarlo al quadrato, quindi li calcolo tutti in tempo proporzionale alla lunghezza di 40, quindi al logaritmo di 32, ed è molto veloce.

Una volta calcolate tutte queste potenze osservo che le variabili x, elevate alla 0, non danno alcun contributo per il calcolo, in quanto, matematicamente un numero elevato alla 0 da 1. Quindi per rappresentare x^{40} usufruisco di x^8 e x^{32} (che sono gli unici 2 che corrispondono all'1 della rappresentazione binaria di 40), in quanto $x^8 * x^{32} = x^{8+32} = x^{40}$

Potenza_Modulare (x, y, z)
if y = 0 then return 1
X ← x; P ← 1
if y_0 = 1 then P ← x
for i = 1 to t do
    X ← X ·X mod z
    if y_i<=1 then P ← P ·X mod z
return P

7.7 NUMERI PRIMI

Un intero p>1 è un numero primo se e solo se i suoi unici divisori sono 1 e sé stesso (Esempio: 2,3,5,7,11,13…).

Teorema fondamentale dell'aritmetica: Ogni intero composto n > 1 può essere scritto in modo unico come prodotto di potenze di primi.
$n= p_1^{e_1} * p_2^{e_2} *\dots* p_k^{e_k}$.

Analizziamo la **Funzione di Eulero**:

Φ(n) = cardinalità di Z_n^* (funzione di Eulero), la lettera è PHI, e ci dice la cardinalità del gruppo moltiplicativo. Notiamo delle cose:

- $\Phi(p)= p-1$ se p è primo (ad esempio $\Phi(7)= 6$ e questo implica che ci sono 6 numeri relativamente primi rispetto a 7)
- $\Phi(pq)= (p-1)(q-1)$ se p,q sono primi (analizzato nella generazione dei parametri dell'RSA)
- $\Phi(n)= n*(1- 1/p_1) *(1- 1/p_2) \dots *(1- 1/p_k)$ se $n= p_1^{e1} p_2^{e2} \dots p_k^{ek}$ p_i primo, $e_i > 0$

Esempio:

$\phi(35) = 6 \bullet 4 = 24$

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25
26	27	28	29	30
31	32	33	34	35

I valori in giallo sono i valori relativamente primi rispetto a 35, mentre quelli in grigio non lo sono.

Osserviamo l'enunciato del **Teorema di Eulero**:

Per ogni a ∈ Z_n^*, $a^{\Phi(n)}= 1 \bmod n$

Esempi:

a=3, n=10, Φ(10)=4 → $3^4 \bmod 10= 1$

a=2, n=11, Φ(11)=10 → $2^{10} \bmod 11= 1$

Un altro teorema che utilizzeremo è il **Teorema di Fermat** che è un caso particolare del caso precedente che ci dice che:

Se p è primo, per ogni a $\in Z_p^*$, $\mathbf{a^{p-1} = 1 \bmod p}$, cioè preso un p primo, a elevato alla p-1 modulo p da proprio 1.

Equivalentemente $\mathbf{a^p = a \bmod p}$

Esempi:

a=7, p=19 → $7^{18} \bmod 19 = 1$

a=10, p=5 → $10^5 \bmod 5 = 0$

Esempio di potenze in Z_{19}*:

a	a^2	a^3	a^4	a^5	a^6	a^7	a^8	a^9	a^{10}	a^{11}	a^{12}	a^{13}	a^{14}	a^{15}	a^{16}	a^{17}	a^{18}
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	4	8	16	13	7	14	9	18	17	15	11	3	6	12	5	10	1
3	9	8	5	15	7	2	6	18	16	10	11	14	4	12	17	13	1
4	16	7	9	17	11	6	5	1	4	16	7	9	17	11	6	5	1
5	6	11	17	9	7	16	4	1	5	6	11	17	9	7	16	4	1
6	17	7	4	5	11	9	16	1	6	17	7	4	5	11	9	16	1
7	11	1	7	11	1	7	11	1	7	11	1	7	11	1	7	11	1
8	7	18	11	12	1	8	7	18	11	12	1	8	7	18	11	12	1
9	5	7	6	16	11	4	17	1	9	5	7	6	16	11	4	17	1
10	5	12	6	3	11	15	17	18	9	14	7	13	16	8	4	2	1
11	7	1	11	7	1	11	7	1	11	7	1	11	7	1	11	7	1
12	11	18	7	8	1	12	11	18	7	8	1	12	11	18	7	8	1
13	17	12	4	14	11	10	16	18	6	2	7	15	5	8	9	3	1
14	6	8	17	10	7	3	4	18	5	13	11	2	9	12	16	15	1
15	16	12	9	2	11	13	5	18	4	3	7	10	17	8	6	14	1
16	9	11	5	4	7	17	6	1	16	9	11	5	4	7	17	6	1
17	4	11	16	6	7	5	9	1	17	4	11	16	6	7	5	9	1
18	1	18	1	18	1	18	1	18	1	18	1	18	1	18	1	18	1

Come si vede, a sinistra ci sono tutti gli elementi più piccoli di 19 (1 a 18), e per ogni colonna l'elevamento a potenza fino a 18 che è proprio p-1. Tutti gli elementi della prima colonna, elevati alla 18 danno sempre 1. Quindi vale il teorema di Fermat.

Vediamo come si fa a generare un numero primo "grande". Questo lo analizziamo perché nell'RSA dobbiamo generare due numeri primi grandi. L'algoritmo è il seguente:

1. Genera a caso un dispari (naturalmente, perché se fosse pari, sicuramente non è primo) p di grandezza appropriata
2. Testa se p è primo
3. Se p è composto, go to **1**.

Questo ciclo viene effettuato in continuazione fintantoché non trovo un numero primo. Se è primo allora esco dal ciclo. Per un algoritmo di questo tipo, potrebbe succedere che l'algoritmo non termini mai. Per valutare un algoritmo del genere, mi baso sul caso medio, quindi mi devo domandare, per la generazione di un p primo, quante volte in media quest'algoritmo girerà? Poiché è un tempo medio, qualche volta l'algoritmo ci metterà un po' di più, qualche volta un po' di meno, però, dal momento che effettuo delle scelte casuali, la probabilità che l'algoritmo ci metta tanto tempo è molto bassa.

Da cosa dipende il numero di volte medie che gira quest'algoritmo? Se ad esempio tra tutti i 1024 bit, ce ne sono solo 10 in totale, la probabilità con cui l'algoritmo trova il numero primo è $10/2^{1024}$. Il numero di volte in cui l'algoritmo girerà è l'inverso di p e quindi $2^{1024}/10$, quindi un numero molto grande. Se invece il numero dei primi fosse la metà di tutti i numeri dispari di 1024 bit, allora la probabilità di successo sarebbe ½ e in media l'algoritmo, per trovare un numero primo, girerà l'inverso della probabilità di successo, quindi 2.

Quindi, sapere quante volte gira quest'algoritmo dipende da quanti numeri primi ci sono tra tutti i numeri dispari di quella lunghezza, quindi, quanti sono i numeri dispari di 1024 bit? Se ce ne sono parecchi, allora l'algoritmo cicla poche volte, se ce ne sono pochi l'algoritmo cicla un numero altissimo di volte.

La seconda cosa importante è verificare se un numero è primo o no, senza fattorizzare. Non esistono algoritmi deterministici efficienti che calcolano se un numero è primo o no, ma esistono dei buoni algoritmi efficienti di tipo probabilistico, ad esempio un algoritmo che ha una bassa probabilità di errore e con cui posso avere una buona confidenza che il risultato che mi fornisce è quello giusto.

Quanti numeri primi ci sono di una stessa grandezza? Per rispondere a questa domanda c'è un famoso teorema nell'ambito dei numeri primi, il **Teorema dei numeri primi** che ci dice che:

Presa $\pi(x)$= numero di primi in [2,x], ossia tutti i numeri primi fino a x, il teorema ci dice che $\lim_{x\to\infty} \frac{\pi(x)}{x/\ln x} = 1$

, cioè $\boldsymbol{\pi(x)}$ **è circa x/ln x**, buona approssimazione.

Esempio: $\pi(10^{23})$= 1.925.320.391.606.803.968.923

$\pi(10^{23})/(10^{23}/\ln 10^{23}) \sim 1{,}020$ (cioè 2% in meno)

Esistono limitazioni, ad esempio, per x> 355.990:

$$\frac{x}{\ln x}\left(1+\frac{1}{\ln x}\right) < \pi(x) < \frac{x}{\ln x}\left(1+\frac{1}{\ln x}+\frac{2{,}51}{(\ln x)^2}\right)$$

Scelto un numero su 512 bit, qual è la probabilità che il numero scelto sia primo?

I primi sono $2^{512}/\log 2^{512}$, dunque la probabilità che sia primo è 1 diviso $\ln 2^{512}$, cioè 1 su 354.89. In media verranno fatti, affinché riesca a trovare un numero primo, l'inverso della probabilità di successo, quindi 354,89. Questo è un calcolo fatto su un intero fino a 512 bit. Il nostro obiettivo è invece calcolare un numero di 512 bit, quindi questo concetto deve essere raffinato.

Innanzitutto, devono essere scelti solo numeri dispari, poiché un numero pari sicuramente non è primo, quindi il numero medio di tentativi sarà $\sim$ 177,44. Se adesso si scelgono dispari in $[2^{511},2^{512}]$ la probabilità è:

$$\approx \left(\frac{2^{512}}{\ln 2^{512}} - \frac{2^{511}}{\ln 2^{511}} \right) \frac{1}{2^{511}/2} \approx \frac{1}{177{,}79}$$

In media ci metterò quindi 177 tentativi.

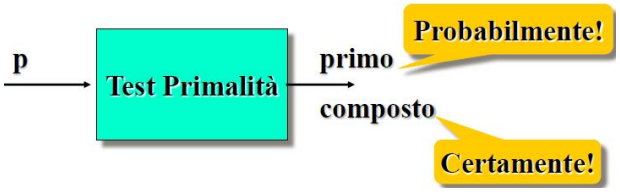
Come faccio a scegliere un numero di esattamente 512 bit che sia dispari? Per essere sicuro che sia dispari inserisco un 1 finale, per essere sicuro che sia di 512 bit, si mette un 1 iniziale, i restanti 510 bit verranno scelti a caso. In questo modo ottengo un numero di 512 bit dispari scelto a caso tra tutti i numeri che hanno queste caratteristiche. La probabilità che questo numero generato sia primo è di 1/177,79.

Se invece voglio trovare un numero di 1024 bit che sia primo, vengono eseguite le stesse azioni precedenti, cambiando opportunamente i valori. Una volta eseguiti tutti gli accorgimenti, mi troverò che la probabilità che il numero generato sia primo è di 1/355,23. Anche in questo caso viene inserito il bit 1 all'inizio e alla fine e 1022 bit scelti a caso.

Tornando all'algoritmo descritto in alto di come si **genera un numero primo**, dopo queste osservazioni, vediamo che non dovrà girare un numero troppo alto di volte per trovarlo. Se il numero è 1024, infatti, girerà intorno ai 1/355,23 volte. Dal punto di vista computazionale posso quindi usare quell'algoritmo.

Abbiamo quindi analizzato la generazione del numero primo.

Ora dobbiamo analizzare il **test di primalità** del numero. Esistono 2 tipi di test che ci dicono se un numero è primo o meno: test probabilistici e test deterministici (dato in input in valore, in maniera deterministica, verrà restituito primo o non primo). Il problema dei test deterministici, è che non esistono test deterministici efficienti per la determinazione di un numero primo. Tutti i test che ci sono di tipo probabilistico, cioè, data una risposta, non è detto che la risposta sia corretta. Analizziamo quanto detto sui test probabilistici:



Preso in input p, viene effettuato il test di primalità, al cui interno ci saranno delle scelte casuali, e in base a questi valori casuali (insieme di testimoni, dopo sono descritti), l'algoritmo dice primo/composto. Attenzione, se il risultato è composto, allora con certezza si può dire che il numero è composto. Se dice primo l'algoritmo, potrebbe essere vero che è primo, ma l'algoritmo potrebbe anche aver composto un errore. In genere quest'errore è più piccolo del 50%.

Se quest'algoritmo lo usassi così com'è, non avrò mai la certezza se il mio numero è primo. La caratteristica di questi test è che li posso iterare una seconda volta. Iterandolo 2 volte, assumiamo che la prima volta il risultato è che è primo, mentre la seconda volta è composto, cosa posso dire? Sicuramente che è composto, perché una volta assunto composto lo sarà sempre. Assumiamo invece che per 2 volte, l'algoritmo ci dice che il numero p è primo, il numero potrebbe essere veramente primo, ma potrebbe anche non esserlo, e la probabilità che l'algoritmo fallisca in questo caso è di ½ la prima volta, ½ la seconda volta, quindi ½ * ½= ¼, cioè il 25%. Supponiamo di farlo girare 3 volte e che per 3 volte so che è primo, in questo caso la probabilità di errore ½* ½ * ½ , cioè 1/8, ossia 12.5% di probabilità di errore.

In generale, la probabilità di errore dopo t iterazioni dell'algoritmo è $(1/2)^t$, si osserva che più volte lo itero, più volte so che è primo, e che quindi posso essere sempre più sicuro che il numero sia effettivamente primo.

L'idea di questi algoritmi di basa sul seguente concetto: Sia W(p) dove p è un numero primo, esiste un insieme W, che è un insieme di testimoni:

- Dato a ∈ [1,p-1] è facile verificare se a ∈ W(p), quindi composto.
- Se p è primo allora W(p) allora W(p) è vuoto.
- Se p è composto allora $|W(p)| \geq p/2$

Se un numero a ∈ W(p) sicuramente è composto, poiché se era primo W(p) era vuoto. Ecco perché posso avere la certezza che se è composto allora realmente lo è, perché ci sono un insieme di testimoni che lo affermano. Al contrario, se primo, o W(p) è vuoto oppure era al di fuori di W(p), perché metà valori appartengono a W(p).

Gli algoritmi probabilistici differiscono tra loro su come viene definito questo W(p).

Un algoritmo con probabilità $(1/2)^t$ è quello di Solovay-Strassen. Ma quello utilizzato è quello di Miller-Rabin (non viene spiegato) in quanto la probabilità di errore è $(1/4)^t$, iterando più volte, l'algoritmo converge molto più velocemente rispetto a quello di Solovay-Strassen.

Un algoritmo deterministico è nato nel 2002, chiamato **AKS**, ma è meno efficiente di Miller-Rabin a causa della potenza del polinomio 12 troppo grande.

SI TORNI ORA AL SUCCESSIVO DOCUMENTO SULL'RSA DOVE CI ERAVAMO FERMATI……