# RSA
# Clase 2

Criptografía
2019

Instituto de Computación
Facultad de Ingeniería
Universidad de la República

# Contenido

# Test de Fermat: Carmichael

- La clase anterior vimos que los números de Carmichael son números compuestos que no tienen ningún testigo de Fermat.
- Vimos que el Test para estos números responde "siempre" (amén de una probabilidad despreciable) que son "Posiblemente Primos"
- Entonces, ¿qué podemos modificar en el Test?

Pomerance (1990) pleads: "*Using the Fermat congruence is so simple that it seems a shame to give up on it just because there are a few counter-examples!*"

# Test de Fermat: Carmichael

- El nuevo test no solo va a distinguir a los números de Carmichael sino que permitirá hallar un factor de estos números.
- Entonces, en tiempo polinomial hallaríamos un factor de estos números.
- En consecuencia, estos números se terminan volviendo peligrosos.

Pomerance (1990) pleads: "*Using the Fermat congruence is so simple that it seems a shame to give up on it just because there are a few counter-examples!*"

# Strong Pseudoprimality Test

El test se basa en la siguiente propiedad:

- Si $N$ es primo y escribimos $N-1=2^e m$ con $m$ impar se cumple que $\forall a \in N, mcd(a,N)=1$
- entonces ó bien
  - $a^m \equiv 1 \ mod(n)$
  - $\exists j \in [0, e-1] \ / \ a^{2jm} \equiv -1 \ mod(n)$

# Strong Pseudoprimality Test

La definición del Test está motivado por lo siguiente:

- $x^{N-1} - 1 \equiv 0 \ mod(N)$
- pero dado que $N$ es impar puede ser escrito como $N = 2 * s + 1$ y $x^{2s} - 1 = (x^s - 1)(x^s + 1) \equiv 0 \ mod(N)$
- Si $N$ es primo, $N$ debe dividir uno de estos factores, entonces $x^s \equiv \pm 1 \ mod(N)$
- si ahora escribimos $N = 2^e m + 1$ obtenemos que: $x^{N-1} - 1 = (x^m - 1)(x^m + 1)(x^{2m} + 1)...(x^{2^{e-1}m} + 1)$

# Strong Pseudoprimality Test

ALGORITHM 3.7. Strong pseudoprimality test.

Input: An odd integer $N \in \mathbb{Z}$ with $N \geq 3$.
Output: Either "composite" or "probably prime".

1. Write $N-1 = 2^e m$, where $e$ and $m$ are (uniquely determined) positive integers with $m$ odd.
2. $x \xleftarrow{\boxdot} \{1, \ldots, N-1\}$.
3. If $\gcd(x, N) \neq 1$ then return "composite".
4. $y \leftarrow x^m \in \mathbb{Z}_N$.
5. If $y = 1$, then return "probably prime"
6. For $i$ from 0 to $e-1$ do steps 7–8
7. If $y = -1$ then return "probably prime"
8. else $y \leftarrow y^2$ in $\mathbb{Z}_N$.
9. Return "composite".

# Strong Pseudoprimality Test

EXAMPLE 3.9. To illustrate the strong pseudoprimality test, we consider the choice $x = 113$ in the three examples of Table 3.1. Step 4 calculates $y_0$, and $y_1, y_2, \ldots$ denote the values successively computed in Step 8.

| $N$ | $N-1 = 2^e \cdot m$ | $y_0$ | $y_1$ | $y_2$ | $y_3$ | $y_4$ |
|---|---|---|---|---|---|---|
| 553 | $2^3 \cdot 69$ | 407 | 302 | 512 | 22 | |
| 557 | $2^2 \cdot 139$ | 556 | 1 | | | |
| $561 = 3 \cdot 11 \cdot 17$ | $2^4 \cdot 35$ | 56 | 331 | 116 | 67 | 1 |

- El test retorna "compuesto" para 553 y 561.
- El test retorn "posiblemente primo" para 557.

# Strong Pseudoprimality Test

Teorema 3.17. El test tiene las siguientes propiedades:

- Si $N$ es primo, el test retorna *probablemente primo*.
- Si $N$ es compuesto, el test retorna *compuesto* con probabilidad de al menos $1/2$
- Para una entrada $N$ de $n$ bits, el test usa $O(n^3)$ bit operaciones.

# Strong Pseudoprimality Test

- La probabilidad de que el test responda incorrectamente "probablemente primo" para un número compuesto $N$ es como mucho $1/2$.
- Si usamos el test $t$ veces independientes, la probabilidad de error es como mucho $2^{-t}$

# Strong Pseudoprimality Test

Teorema 3.18. El test repetido $t$ veces de forma independiente, tiene las siguientes propiedades para una entrada $N$:

- si responde *compuesto*, entonces $N$ es compuesto.
- si responde "posiblemente primo", luego $N$ es primo con probabilidad de al menos $1 - 2^{-t}$

# Pseudoprimalidad

- Si el test responde “probablemente primo”, $N$ es llamado “pseudoprimo”
- $N$ es primo o no, el nombre “pseudoprimo” responde a la probabilidad de las elecciones aleatorias que se hicieron en el algoritmo.
- Si se ejecuta el test 1000 veces respondiendo “probablemente primo”, significa que si $N$ no es primo, entonces me equivoque con probabilidad como mucho $2^{-1000}$
- Si estas volando en un avión cuya seguridad depende de la primalidad de este número *pseudoprimo* no hay de que preocuparse, dado que otros eventos son mucho más probables a fallar.

# Pseudoprimalidad

¿Qué se hace en la práctica?

- Primero se testea si $N$ tiene un factor primo pequeño. Para ello, se computa el *mcd* de $N$ y los primeros números primos, digamos hasta $log(N)$.
- Luego se aplica el test. A menos que se requiera una seguridad deterministica.

# Encontrar números primos

- El método clásico para encontrar los números primos menores a cierto número $x$ es la *Criba de Eratóstenes*. (Erathosthenes 276-194 AC).
- Se comienza con la lista $L = 2, 3, 4, 5, ....$ El 2 se declara primo y se tacha en la lista todos los múltiplos de este. Luego se sigue el procedimiento con el siguiente número no tachado, en este caso 3 y así sucesivamente.
- El tiempo de ejecución es de $O(xlogx)$

# Encontrar números Psuedoprimos

El siguiente algoritmo busca un número pseudoprimo grande en el rango de RSA.

ALGORITHM 3.19. Finding a pseudoprime.

Input: An integer $n$ and a confidence parameter $t$.
Output: A pseudoprime number $N$ in the range from $2^{(n-1)/2}$ to $2^{n/2}$.

1. $y \longleftarrow 2^{(n-1)/2}$.
2. Repeat steps 3 and 4 until some $N$ is returned.
3. $N \xleftarrow{\boxdot} \{\lceil y \rceil, \ldots, \lfloor \sqrt{2}\, y \rfloor\}$.
4. Call the strong pseudoprimality test with input $N$ for $t$ independently chosen $x \xleftarrow{\boxdot} \{1, \ldots, N-1\}$. Return $N$ if and only if all these tests return "probably prime".

# Encontrar números Psuedoprimos

- Para encontrar números primos, vamos seleccionando candidatos y les aplicamos el test de primalidad hasta tener éxito.
- Pero si no existieran números primos en el rango que estamos buscando, entraríamos en loop infinito en el algoritmo anterior.
- Otro problema es si existieran pocos números primos en el rango, entonces el algoritmo tardaría mucho en responder éxito.
- Afortunadamente, veremos que en el rango deseado existen muchos números primos.

# Teorema de los números primos

- El teorema nos hablará sobre la densidad de números primos.
- Definimos $\pi(x)$ como la cantidad de números primos $p$ menores o iguales a $x$.
- Definimos $p_n$ como el n-ésimo número primo. Ejemplo, $p_3 = 5$.
- Definimos $\upsilon(x) = \sum_{p \leq x} ln(p) = ln \prod_{p \leq x} p$

# Teorema de los números primos

PRIME NUMBER THEOREM 3.21. *We have approximately*

$$\pi(x) \approx \frac{x}{\ln x}, \quad \vartheta(x) \approx x, \quad p_n \approx n \ln n,$$

*and more precisely*

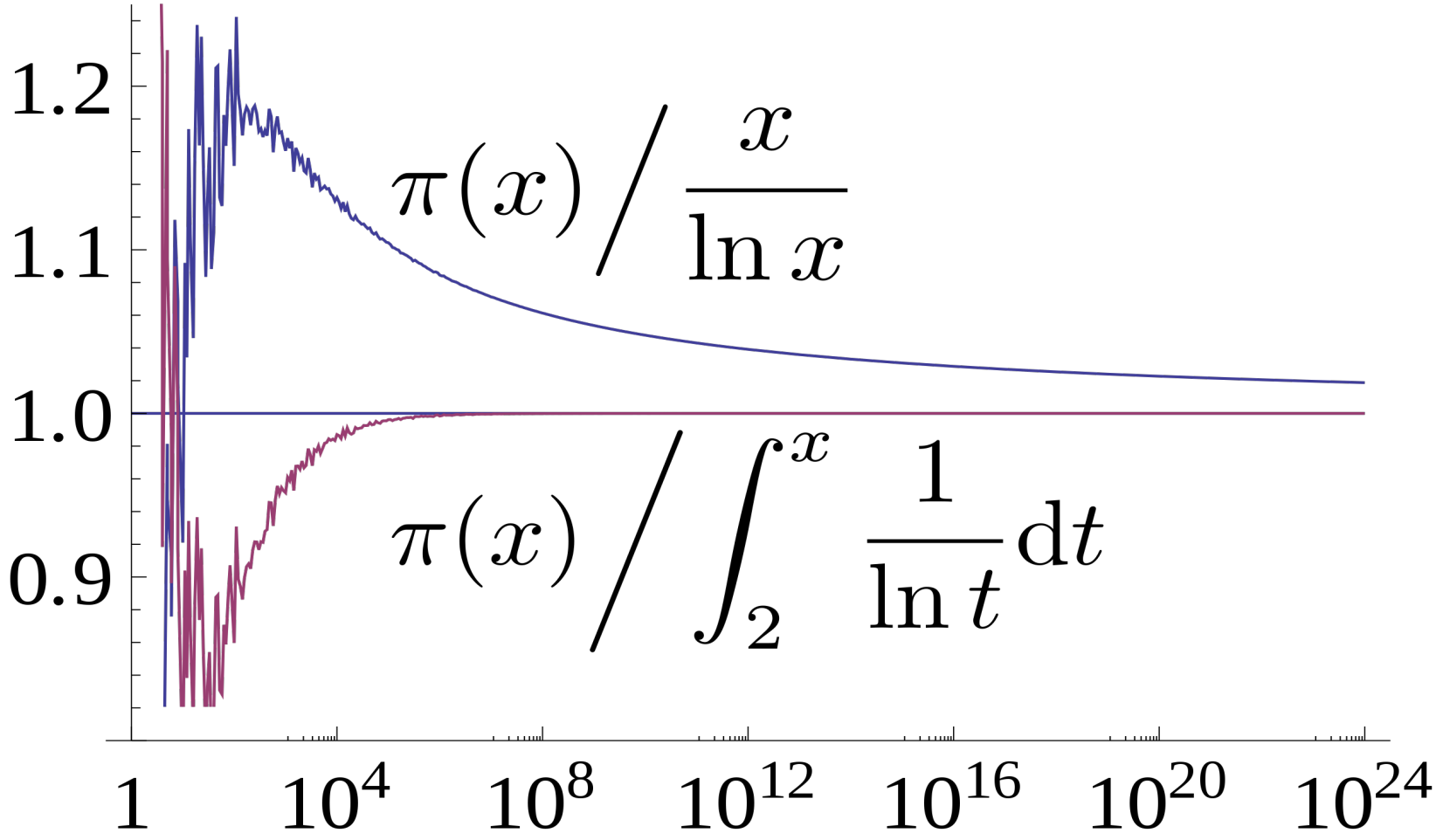
$$\frac{x}{\ln x}\Big(1 + \frac{1}{2\ln x}\Big) < \pi(x) < \frac{x}{\ln x}\Big(1 + \frac{3}{2\ln x}\Big) \text{ for } x \geq 59,$$

$$\frac{3x}{5\ln x} < \pi(2x) - \pi(x) < \frac{7x}{5\ln x} \text{ for } x \geq 21,$$

$$n\Big(\ln n + \ln\ln n - \frac{3}{2}\Big) < p_n < n\Big(\ln n + \ln\ln n - \frac{1}{2}\Big) \text{ for } n \geq 20,$$

$$x\Big(1 - \frac{1}{2\ln x}\Big) < \vartheta(x) < x\Big(1 + \frac{1}{2\ln x}\Big) \text{ for } x \geq 563. \qquad \square$$

# Teorema de los números primos



# Teorema de los números primos

Teorema 3.22

- Para una entrada $n \geq 23$ y $t$, la salida del algoritmo 3.19 es primo con probabilidad de al menos $1 - 2^{-t+1}n$.
- Ejecuta en el orden de $O(tn^4)$ bit operaciones.

# RSA: el parámetro t

Para RSA,

- si escribimos $t = log_2(2n) + s$, el algoritmo retorna un número primo con probabilidad de al menos $1 - 2^{-s}$
  y utiliza del orden $O((s + log(n))n^4)$ bit operaciones.
- En la práctica $t = 50$ es suficiente.

# RSA: determinar e

- Recordemos que el exponente $e$ se debe seleccionar aleatoriamente coprimo con $L = \phi(N)$.
- Se espera que se verifique como mucho $O(logn)$ valores aleatorios.
- Cada test de coprimalidad con $(p-1)$ y $(q-1)$ toma $O(n^2)$ bit operaciones. Ver algoritmo 15.24.

# RSA: Análisis de Eficiencia

Eficiencia de RSA utilizando números pseudoprimos:

| | |
|---|---|
| Find $n/2$-bit pseudoprimes at random | $O(n^4 \log n)$, |
| Find $e$ | $O(n^2 \log n)$, |
| Calculate $N$ and $d$ | $O(n^2)$, |
| Calculate powers modulo $N$ | $O(n^3)$. |

COROLLARY 3.23. *The key generation in RSA can be done in time* $O(n^4 \log n)$, *and the encryption or decryption of one* $n$*-bit plaintext block with* $O(n^3)$ *bit operations.*

# Seguridad de RSA

- Consideramos un adversario $A$ como un algoritmo en tiempo polinomial, posiblemente probabilistico.
- El adversario $A$ conoce la clave pública $p_k = (N, e)$ y el mensaje cifrado $y = enc_{pk}(x)$.
- Tenemos muchas nociones en cuanto a "quebrar el criptosistema".
- El adversario puede tener el objetivo de computar:
  - $B_1$: el mensaje plano $x$
  - $B_2$: el exponente secreto $d$
  - $B_3$: el valor de la función $\phi(N)$
  - $B_4$: un factor $p$ de $N$

# Reducción

Sean $A$ y $B$ dos problemas computacionales,

- Una reducción en tiempo polinomial de $A$ a $B$ es un algoritmo en tiempo polinomial para $A$ el cual puede hacer llamadas a una subrutina de $B$.
- Si existe esta reducción, decimos que $A$ es reductible a $B$ en tiempo polinomial.
- Escribimos, $A \leq_p B$
- Si también existe $B \leq_p A$, decimos que $A$ y $B$ son equivalentes: $A \equiv_p B$

# Reducción

La existencia de la reducción $A \leq_p B$ tiene dos consecuencias:

- Si $B$ es fácil, entonces también $A$ es fácil.
- Si $A$ es dificil, entonces también $B$ es dificil.

# Reducción

Asumimos que cierto problema $X$ es dificil de resolver (primitiva criptográfica)

- Entonces para cierto criptosistema $S$, quebrar $S$ implica quebrar $X$
- $X \leq_p S$ ($X$ es mi axioma de seguridad)

Se considera que el adversario puede ser exitoso en el ataque con suficiente probabilidad.

# RSA: Reducciones

Hasta ahora vimos que *Bob* es un algoritmo en tiempo polinomial, sus computos muestran que:

$$B_1 \leq_p B_2 \leq_p B_3 \leq_p B_4$$

- $B_1$: el mensaje plano $x$
- $B_2$: el exponente secreto $d$
- $B_3$: el valor de la función $\phi(N)$
- $B_4$: un factor $p$ de $N$

¿qué pasa con el otro sentido?

# Reducción: $B_4 \leq_p B_3$

$$B_4 \leq_p B_3.$$

¿qué quiero demostrar?

- que puedo construir un algoritmo en tiempo polinomial llamando a subrutinas de $B_3$ que resuelvan $B_4$.
- en otras palabras, construir un algoritmo en tiempo polinomial que conociendo $\phi(N)$ logre factorizar $N$

# Reducción: $B_4 \leq_p B_3$

LEMMA 3.28. $B_4 \leq_p B_3$.

PROOF. We know $N = p \cdot q$, and with one call to a subroutine for $B_3$ we find $\varphi(N) = (p-1)(q-1) = pq - (p+q) + 1$. We substitute $q = N/p$ and multiply up the denominator $p$ to find a quadratic equation in $p$:

$$p^2 - (N + 1 - \varphi(N)) \cdot p + N = 0.$$

We solve it, and have solved $B_4$. □

# Reducción: $B_3 \leq_p B_2$

$$B_3 \leq_p B_2.$$

¿qué quiero demostrar?

- que puedo construir un algoritmo en tiempo polinomial llamando a subrutinas de $B_2$ que resuelvan $B_3$.
- en otras palabras, sabemos resolver (dado $N$) el secreto $d$. Entonces, construyo un algoritmo en tiempo polinomial que haciendo llamadas a subrutinas de $B_2$ logre computar $\phi(N)$

# Reducción: $B_3 \leq_p B_2$

If we want to show $B_3 \leq_p B_2$, we may proceed as follows. We keep $N$ fixed, and call the subroutine for $B_2$ with several random values $e_1, e_2, \ldots$ for $e$. Each time we get a value $d_1, d_2, \ldots$ with $e_i d_i \equiv 1 \bmod \varphi(N)$. In other words, $\varphi(N)$ is a divisor of each $e_i d_i - 1$, and hence also of $\gcd(e_1 d_1 - 1, e_2 d_2 - 1, \ldots)$.

Now if we could show that our random $e_1, e_2, \ldots$ generate random quotients $(e_1 d_1 - 1)/\varphi(N), (e_2 d_2 - 1)/\varphi(N), \ldots$, then we could conclude that a few choices are already likely to give us $\varphi(N)$ as gcd. This works quite well in practice.

# Reducción: $B_4 \leq_p B_1$

¿qué sucede con: $B_4 \leq_p B_1$?

- es decir, ¿puedo construir un algoritmo en tiempo polinomial con llamadas a $B_1$ (averiguar $x$) que resuelvan $B_4$ (factorizar $N$)?
- Si la respuesta fuera que **SI** estoy diciendo que:
  - $B_1$ es fácil de quebrar $\rightarrow$ $B_4$ es fácil de quebrar:
    si es fácil averiguar $x \rightarrow$ es fácil averiguar algún factor $p$ o $q$
  - $B_4$ es difícil de quebrar $\rightarrow$ $B_1$ es difícil de quebrar:
    si es difícil averiguar algún factor $p$ o $q \rightarrow$ es difícil de averiguar $x$

# Reducción: $B_4 \leq_p B_1$

Esto último es la **RSA assumption!**

- RSA es seguro, dado que factorizar el módulo $N$ es difícil.
- Este es el problema abierto: aún no se conoce algoritmo eficiente capaz de factorizar un número grande.

# RSA: Conjetura de Seguridad

- No existe algoritmo probabilistico en tiempo polinomial $A$ el cual con entrada $(N, e, y)$ tenga como salida $X$ con probabilidad no despreciable.
- Utilizamos el parametro de seguridad $n$ para referirnos a los conceptos de "probabilidad despreciable" y "tiempo polinomial".
- Qué el problema de la factorización sea difícil en el caso general, no quiere decir que no existan factores de $N$ que lo permitan factorizar facilmente.
  Por ejemplo, si $p$ y $q$ se seleccionan muy cercanos, entonces $N$ puede ser factorizado facilmente (ejercicio).

The end.