

UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA
FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS
ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

SEGURIDAD EN COMPUTACIÓN



TEMA:

Problemas matemáticos en Criptografía

Docente:

Mg. Rolando Jesús Cárdenas Talavera

Semestre:

VIII

Presentado por:

Philco Puma, Josue Samuel

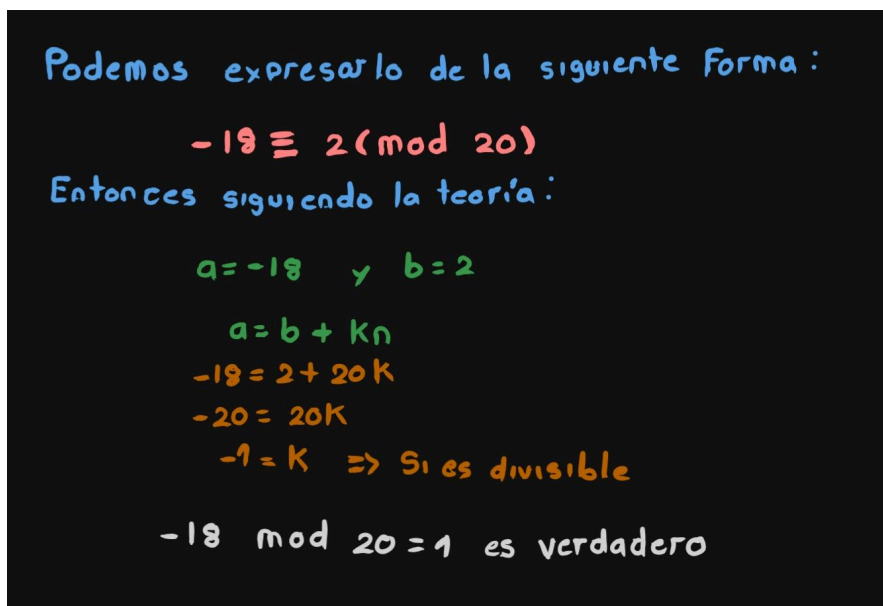
Arequipa - Perú

2025

1. Hallar la congruencia correcta (demostrar) a

En las congruencias, se tiene que las denotaciones $a \equiv b \pmod{n}$ que es llamada también **Aritmética modular**, significa que a y b deben dejar el mismo al momento de dividirse por n , o que de forma equivalente su diferencia es divisible por n : $a = b + kn$ para algún $k \in \mathbb{Z}$. Entonces con esta teoría pasemos a demostrar la congruencia correcta de los siguientes 4 ejemplos:

a) $-18 \pmod{20} = 2$ (Verdadero)



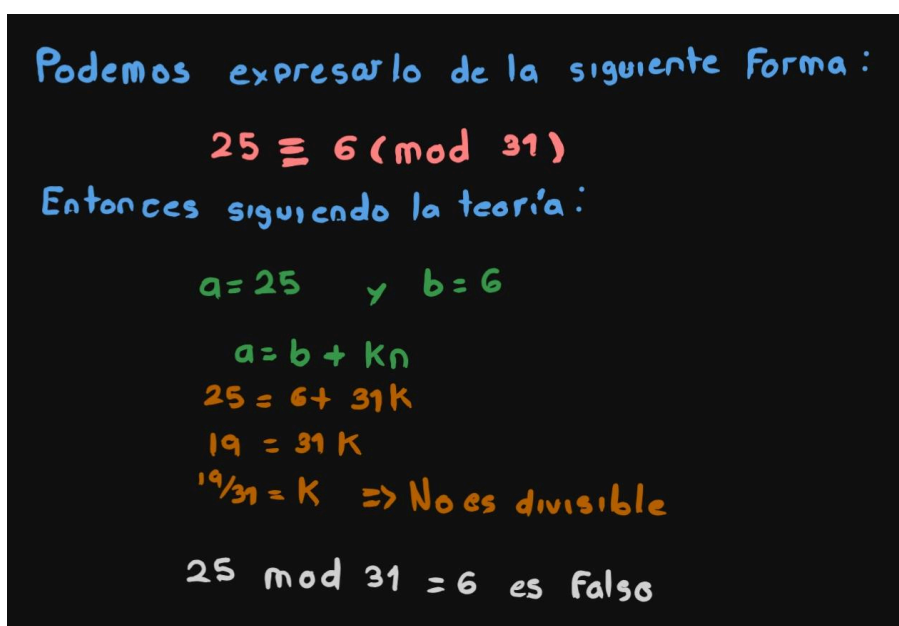
Podemos expresarlo de la siguiente forma:

$$-18 \equiv 2 \pmod{20}$$

Entonces siguiendo la teoría:

$$a = -18 \quad y \quad b = 2$$
$$a = b + kn$$
$$-18 = 2 + 20k$$
$$-20 = 20k$$
$$-1 = k \Rightarrow \text{Si es divisible}$$
$$-18 \pmod{20} = 2 \text{ es verdadero}$$

b) $25 \pmod{31} = 6$ (Falso)



Podemos expresarlo de la siguiente forma:

$$25 \equiv 6 \pmod{31}$$

Entonces siguiendo la teoría:

$$a = 25 \quad y \quad b = 6$$
$$a = b + kn$$
$$25 = 6 + 31k$$
$$19 = 31k$$
$$19/31 = k \Rightarrow \text{No es divisible}$$
$$25 \pmod{31} = 6 \text{ es falso}$$

c) $33 \pmod{10} = 4$ (Falso)

Podemos expresarlo de la siguiente forma:

$$33 \equiv 4 \pmod{10}$$

Entonces siguiendo la teoría:

$$a = 33 \quad y \quad b = 4$$

$$a = b + kn$$

$$33 = 4 + 10k$$

$$29 = 10k$$

$$29/10 = k \Rightarrow \text{No es divisible}$$

$$33 \bmod 10 = 4 \text{ es falso}$$

d) $-9 \bmod 6 = 2$ (Falso)

Podemos expresarlo de la siguiente forma:

$$-9 \equiv 2 \pmod{6}$$

Entonces siguiendo la teoría:

$$a = -9 \quad y \quad b = 2$$

$$a = b + kn$$

$$-9 = 2 + 6k$$

$$-11 = 6k$$

$$-11/6 = k \Rightarrow \text{No es divisible}$$

$$-9 \bmod 6 = 2 \text{ es falso}$$

Entonces, si lo llevamos a código en Python quedaría de la siguiente forma:

Python

```
def are_congruents(a, b, n):  
    return a % n == b % n
```

```
list_cases = [  
    (-18, 2, 20),  
    (25, 6, 31),  
    (33, 4, 10),  
    (-9, 2, 6)
```

```
]
```

```
for a, b, n in list_cases:  
    print(f"{a} ≡ {b} (mod {n}) ? -> {are_congruents(a, b, n)}")
```

Lo que hace es seguir la teoría de $a \equiv b \pmod{n}$ donde es $a \bmod n = b \bmod n$, entonces la salida que nos da es la siguiente:

```
⇒ -18 ≡ 2 (mod 20) ? -> True  
   25 ≡ 6 (mod 31) ? -> False  
   33 ≡ 4 (mod 10) ? -> False  
   -9 ≡ 2 (mod 6) ? -> False
```

2. Hallar el inverso multiplicativo de 5 en módulo 27

El **Inverso Multiplicativo** de x módulo n es y tal que $xy = 1 \pmod{n}$. Este inverso existe si y sólo si $\gcd(x, n) = 1$, es decir, que x y n deben ser coprimos.

Entonces hallemos el inverso multiplicativo del ejercicio y marcar la alternativa correcta:

Debemos comprobar si existe inverso:

$$\gcd(5, 27) = 1$$

5	27		1
5	27		1
⋮	⋮		⋮

Si existe inverso comprobamos:

$$\begin{aligned} x=1 &: (5 \times 1) \% 27 = 5 \\ x=2 &: (5 \times 2) \% 27 = 10 \\ &\vdots \\ x=11 &: (5 \times 11) \% 27 = 55 \% 27 = 1 \end{aligned}$$

El inverso multiplicativo es 11

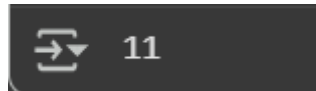
- a) 10
- b) 7
- c) 4
- d) 11

En código quedaría de la siguiente forma:

```
Python
def inverse_multiplicative(a, n):
    for x in range(1, n):
        if (a * x) % n == 1:
            return x
    return None

print(inverse_multiplicative(5, 27))
```

Lo que va a hacer es recorrer a todos los números posibles hasta $n - 1$ chequeando si $(a \cdot x) \bmod n = 1$, si encuentra retorna el valor encontrado, sino dará None.



3. El valor de x en la expresión $2^x \bmod 19 = 7$

Calcular $y = a^x \bmod p$ es un problema de **logaritmo discreto**, es sencillamente calcularlo ya que crece linealmente con el exponente x , pero el problema inverso que es hallar x conociendo los valores de a , y y p , para resolverlo se pueden hacer en potencias de a en módulos pequeños.

Entonces resolvamos la expresión:

Por definición del logaritmo discreto tendríamos:

$$y = a^x \bmod p$$
$$x = \log_a y \bmod p$$
$$x = \log_2 7 \bmod 19$$

Entonces podemos resolverlo por fuerza bruta que es con potencias en a :

$$\begin{aligned} x=1 : 2^1 \bmod 19 &= 2 & \Rightarrow \frac{2}{19} = 0.105 \\ x=2 : 2^2 \bmod 19 &= 4 & \Rightarrow \frac{4}{19} = 0.2105 \\ x=3 : 2^3 \bmod 19 &= 8 \\ &\vdots \\ x=6 : 2^6 \bmod 19 &= 64 \bmod 19 = \frac{64}{19} = 3 \times 19 + 7 \end{aligned}$$

$x = 6$

- a) 6
- b) 5
- c) 4

d) 3

En código quedaría de la siguiente forma:

```
Python
def found_value(a, m):
    for x in range(1, m):
        if (a ** x) % m == 7:
            return x
    return None

print(found_value(2, 19))
```

Lo que va a hacer es recorrer por fuerza bruta todos los números posibles que van a cumplir con la condición del módulo, dando como resultado lo siguiente:



4. Luego de investigar el concepto de raíz primitiva de un primo, responde: Si α es una raíz primitiva del primo p , es porque se cumple que:

En raíces primitivas tenemos un p que es un número primo va a ser un entero g tal que las potencias de g módulo p generan todos los números enteros desde 1 hasta $p - 1$. Veamos un ejemplo matemático:

Sea $p = 7$ y $\alpha = 3$

- Factorizamos $p-1$

$$6 = 2 \times 3$$

\therefore Los factores primos son 2 y 3

- Probar las potencias de los primos

$$\alpha^{(p-1)/2} \not\equiv 1 \pmod{7}$$
$$\alpha = 3 \Rightarrow 3^{6/2} = 3^3 \pmod{7}$$
$$= 27 \pmod{7}$$
$$= 27 \div 7$$
$$= 3 \times 7 + 6$$
$$\Rightarrow 3^{6/2} = 3^3 \pmod{7}$$
$$= 6 \pmod{7}$$
$$= 1 \times 7 + 6$$

- Probar todas las potencias

$$3^1 \equiv 3 \pmod{7} = 7 \times 0 + 3$$
$$3^2 \equiv 2 \pmod{7} = 7 \times 1 + 2$$
$$3^3 \equiv 6 \pmod{7} = 7 \times 3 + 6$$
$$3^4 \equiv 4 \pmod{7} = 7 \times 1 + 4$$
$$3^5 \equiv 5 \pmod{7} = 7 \times 3 + 5$$
$$3^6 \equiv 1 \pmod{7} = 7 \times 10 + 1$$

En código sería de la siguiente forma:

```
Python
def is_prime(p):
    if p < 2:
        return False
    for i in range(2, int(p ** 0.5) + 1):
        if p % i == 0:
            return False
    return True

def gcd(a, p):
    if p == 0:
        return a
    return gcd(p, a % p)

def is_primitive_root(a, p):
    if not is_prime(p) or gcd(a, p) != 1:
        return False

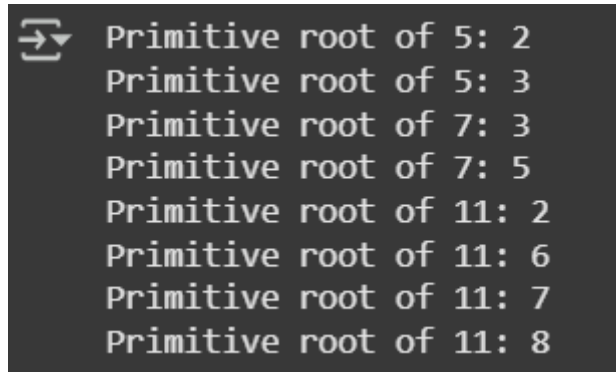
    phi = p - 1
    factors = []
    n = phi
    d = 2
    while d * d <= n:
        if n % d == 0:
            factors.append(d)
            while n % d == 0:
                n //= d
            d += 1
    if n > 1:
        factors.append(n)

    for q in factors:
        if pow(a, phi // q, p) == 1:
            return False
    return True

primes = [5, 7, 11]
for p in primes:
    for a in range(1, p):
        if is_primitive_root(a, p):
            print(f"Primitive root of {p}: {a}")
```

Este código primero verifica si p es un número primo ya que se aplica al grupo multiplicativo de los enteros módulo p . Luego se comprueba que a debe ser raíz primitiva si existe que $\gcd(a, p) = 1$. Por último se factoriza en $p - 1$ para obtener

todos sus factores primos y aplica la condición de que si un valor es 1, entonces el orden de α es más pequeño que $p - 1$ y no cumple para ser raíz primitiva, caso contrario que ninguno sea 1, significa que el orden de α es exactamente igual a $p - 1$ y es raíz primitiva.



```
⇒ Primitive root of 5: 2
    Primitive root of 5: 3
    Primitive root of 7: 3
    Primitive root of 7: 5
    Primitive root of 11: 2
    Primitive root of 11: 6
    Primitive root of 11: 7
    Primitive root of 11: 8
```

Entonces si vemos las alternativas dadas:

- a) Es número impar y genera todos los restos del primo p
- b) Es número par y genera todos los restos del primo p
- c) Es número par o impar y genera todos los restos del primo p**
- d) Es un número que no genera todos los restos del primo p

Nuestra respuesta correcta para este ejercicio es que se va a cumplir que: **Es un número par o impar y genera todos los restos del primo p** (Letra c).

Referencia de raíz primitiva

[1] “Primitive root - algorithms for competitive programming”. Main Page - Algorithms for Competitive Programming. Accedido el 14 de septiembre de 2025. [En línea]. Disponible: <https://cp-algorithms.com/algebra/primitive-root.html>

[2] “Finding a primitive root of a prime number”. Mathematics Stack Exchange. Accedido el 14 de septiembre de 2025. [En línea]. Disponible: https://math.stackexchange.com/questions/124408/finding-a-primitive-root-of-a-prime-number#comment287482_124408