

Universidad Nacional Autónoma de México
Facultad de Ciencias

Criptografía y Seguridad

Semestre: 2025-1

Tarea 2

García Ponce José Camilo 319210536

Equipo Pingüicoders
Arrieta Mancera Luis Sebastian
Cruz Cruz Alan Josué
García Ponce José Camilo
Matute Cantón Sara Lorena

1. Investiga qué es el cifrado RC5. Da la descripción de su funcionamiento y sus debilidades.

El cifrado RC5 fue diseñado por Ronald Rivest en 1994. Es un cifrado por bloques y es clasificado como un criptosistema con clave secreta. El cifrado puede usar bloques de tamaño 32, 64 o 128 bits, la clave usada tiene longitud de 0 a 2040 bits (pero la recomendada es de 128 bits). RC5 hace uso de operaciones como sumas modulares, operaciones XOR y rotaciones dependientes.

Ahora veamos cómo funciona el cifrado.

Tendremos las siguientes objetos/variables: una palabra de longitud variable w , un número de vueltas variable v y una clave de longitud variable b .

Las operaciones que se realizarán (para cifrado y descifrado) son: suma modular $+$, resta modular $-$, XOR bit a bit \oplus , rotación de bits a la izquierda $x \ll y$ (desplazar un bit de x a la izquierda y veces) y rotación de bits a la derecha $x \gg y$ (desplazar un bit de x a la derecha y veces).

Antes de empezar a cifrar se realiza lo siguiente, la clave es expandida para poder llenar un arreglo S de tamaño $2r + 2$ palabras.

Ahora veamos el algoritmo, para esto tendremos que (A, B) representan dos palabras del bloque de entrada (de 32 bits cada una).

- 1) Lo primero es realizar las siguientes sumas $A = A + S[0]$ y $B = B + S[1]$
- 2) Despues se itera una variable i de 1 a r , en cada iteración se realiza lo siguiente:
 - 2.1) Se combinan A y B usando XOR, después al resultado lo rotamos a B veces a la izquierda y por último al resultado se le suma la subclave $S[2 * i]$ y actualizamos el valor de A . De esta manera se vería todo $A = ((A \oplus B) \ll B) + S[2 * i]$
 - 2.2) Se combinan B y A usando XOR, después al resultado lo rotamos a A veces a la izquierda y por último al resultado se le suma la subclave $S[2 * i + 1]$ y actualizamos el valor de B . De esta manera se vería todo $B = ((B \oplus A) \ll A) + S[2 * i + 1]$
- 3) Despues de terminar de iterar a i , obtenemos el bloque cifrado (A, B) o $(A[r], B[r])$

En pseudocódigo se vería algo así

$$A = A + S[0]$$

$$B = B + S[1]$$

$$i = 1$$

while $i \leq r$:

$$A = ((A \oplus B) \ll B) + S[2 * i]$$

$$B = ((B \oplus A) \ll A) + S[2 * i + 1]$$

$$i = i + 1$$

return A, B

Ahora veamos como funciona el descifrado.

Para este proceso vamos a realizar las operaciones que para el cifrado solo usaremos las operaciones inversas y en orden inverso.

Entonces los pasos serían:

- 1) Primero dividir el texto cifrado en dos mitades iguales A y B

2) Luego se itera una variable i de r a 1 donde en cada iteración se realiza lo siguiente:

2.1) Primero a B se le resta la subclave $S[2 * i + 1]$, después al resultado lo rotamos a A veces a la derecha y por último al resultado se le aplica XOR con A y actualizamos el valor de B . De esta manera se vería todo

$$B = ((B - S[2 * i + 1]) \gg A) \oplus A$$

2.1) Después a A se le resta la subclave $S[2 * i]$, después al resultado lo rotamos a B veces a la derecha y por último al resultado se le aplica XOR con A y actualizamos el valor de A . De esta manera se vería todo

$$A = ((A - S[2 * i]) \gg B) \oplus B$$

3) Luego de iterar i se realizan las restas $B = B - S[1]$ y $A = A - S[0]$

4) Al final tenemos el bloque descifrado (A, B) o $(A[0], B[0])$

En pseudocódigo se vería algo así

$i = r$

while $i \geq 1$:

$$B = ((B - S[2 * i + 1]) \gg A) \oplus A$$

$$A = ((A - S[2 * i]) \gg B) \oplus B$$

$$i = i - 1$$

$$B = B - S[1]$$

$$A = A - S[0]$$

return A, B

Por último, veamos las debilidades de este cifrado.

Lo primero que podemos observar que representa como una debilidad es que investigamos que RC5 tiene claves débiles, las cuales pueden hacer que el cifrado sea vulnerable a cierto tipo de ataques, la razón de esto surge del proceso en el cual se expanden las claves, cuando se generan las subclaves, ya que las llaves débiles generan subclaves con propiedades estadísticas malas o pobres.

Otra vulnerabilidad es que los criptoanálisis diferenciales y lineales pueden ser muy efectivos contra este cifrado, principalmente cuando el número de rondas es pequeño. Incrementar el número de rondas ayuda a mitigar esta debilidad pero aumenta el costo computacional para cifrar.

También RC5 no ha sido tan adaptado generalmente a comparación de otros cifrados. Otros problemas son que el tamaño de los bloques está limitado a 64 bits.

Además RC5 ha sido reemplazado por algoritmos de cifrado más nuevos y modernos, como AES que ofrece una mejor seguridad y rendimiento.

Fuentes usadas para este ejercicio:

RC5 Encryption Algorithm. (2018, June 29). GeeksforGeeks.

<https://www.geeksforgeeks.org/rc5-encryption-algorithm/>

www.naukri.com. (2024). Code 360 by Coding Ninjas. Naukri.com.

<https://www.naukri.com/code360/library/rc5-encryption-algorithm>

Rams Muñoz. (2024). Tarea 7 Cifrados de bloque (RC5). Blogspot.com.
<https://criptografiaverm1.blogspot.com/2013/07/tarea-7-cifrados-de-bloque-rc5.html>

2. Ejercicio 2

- Presenta una implementación de DES. Tu rutina debe ser capaz de cifrar y descifrar

El código de se encuentra en des.py

La implementación usada es basada en la tarea moral de DES, por lo tanto la parte del cifrado de uso con ayuda de ChatGPT, ya que la tarea moral lo permitía.

Para esto se le pidió a ChatGPT algunas funciones para poder manejar bits y también las tablas usadas para DES.

Luego se fueron haciendo funciones para cada parte del proceso de DES. Las funciones importantes son cifrar_bloque y descifrar_bloque

Las funciones usadas para el cifrado son las siguientes:

Función para aplicar la permutación inicial

```
def permutacion_inicial(bits):  
  
    # Permutacion inicial  
    tabla_IP = [  
        [58, 50, 42, 34, 26, 18, 10, 2],  
        [60, 52, 44, 36, 28, 20, 12, 4],  
        [62, 54, 46, 38, 30, 22, 14, 6],  
        [64, 56, 48, 40, 32, 24, 16, 8],  
        [57, 49, 41, 33, 25, 17, 9, 1],  
        [59, 51, 43, 35, 27, 19, 11, 3],  
        [61, 53, 45, 37, 29, 21, 13, 5],  
        [63, 55, 47, 39, 31, 23, 15, 7]  
    ]  
  
    # Asegúrate de que la cadena de bits tenga 64 bits  
    if len(bits) != 64:  
        raise ValueError("La cadena de bits debe tener 64 bits.")  
  
    # Aplicamos la permutación inicial usando la tabla_IP  
    permutacion = [bits[pos - 1] for fila in tabla_IP for pos in fila]  
    return ''.join(permutacion)
```

Función para la expansión de bits

```
def expandir_bloque(bloque_32bits):  
  
    # Expacion  
    tabla_E_bit_selection = [  
        [32, 1, 2, 3, 4, 5],  
        [4, 5, 6, 7, 8, 9],  
        [8, 9, 10, 11, 12, 13],  
        [12, 13, 14, 15, 16, 17],  
        [16, 17, 18, 19, 20, 21],  
        [20, 21, 22, 23, 24, 25],  
        [24, 25, 26, 27, 28, 29],  
        [28, 29, 30, 31, 32, 1]  
    ]  
  
    # Aseguramos que el bloque de entrada tenga 32 bits  
    if len(bloque_32bits) != 32:  
        raise ValueError("El bloque debe ser de 32 bits.")  
  
    # Creamos el nuevo bloque de 48 bits aplicando la tabla  
    bloque_48bits = [bloque_32bits[pos - 1] for fila in tabla_E_bit_selection for pos in fila]  
  
    # Convertimos la lista de bits en una cadena  
    return ''.join(bloque_48bits)
```

Función para aplicar XOR entre la llave y un bloque de bits

```
def aplicar_xor_48bits(bloque_48bits, llave_48bits):
    # Aseguramos que tanto el bloque como la llave tengan 48 bits
    if len(bloque_48bits) != 48 or len(llave_48bits) != 48:
        raise ValueError("Tanto el bloque como la llave deben ser de 48 bits.")

    # Realizamos el XOR bit a bit entre el bloque y la llave
    resultado_xor = ''.join(
        str(int(bloque_48bits[i]) ^ int(llave_48bits[i])) for i in range(48)
    )

    return resultado_xor
```

Función para aplicar las cajas (las cajas completas están en el código)

```
def aplicar_cajas_s(bloque_48bits):

    # Cajas S
    tabla_S1 = [
        [14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
        [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
        [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
        [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]
    ]
    tabla_S2 = [
        [15, 1, 0, 14, 6, 11, 3, 4, 9, 7, 2, 12, 13, 8, 5, 10]
    ]

    # Tablas S1 a S8
    tablas_S = [tabla_S1, tabla_S2, tabla_S3, tabla_S4, tabla_S5, tabla_S6, tabla_S7, tabla_S8]

    # Asegúrate de que el bloque tenga 48 bits
    if len(bloque_48bits) != 48:
        raise ValueError("El bloque debe tener 48 bits.")

    # Función auxiliar para convertir un número decimal a una cadena binaria de longitud n
    def decimal_a_binario(n, longitud):
        return bin(n)[2:].zfill(longitud)

    # Dividimos el bloque de 48 bits en 8 subbloques de 6 bits
    subbloques = [bloque_48bits[i:i+6] for i in range(0, 48, 6)]

    resultado = ""

    # Procesamos cada subbloque y aplicamos la caja S correspondiente
    for i, subbloque in enumerate(subbloques):
        # Seleccionamos la tabla S correspondiente
        tabla_s = tablas_S[i]

        # Obtenemos el primer y último bit, los convertimos a decimal
        fila = int(subbloque[0]) + subbloque[5], 2

        # Obtenemos los 4 bits del medio, los convertimos a decimal
        columna = int(subbloque[1:5], 2)

        # Buscamos el valor en la tabla S correspondiente
        valor_s = tabla_s[fila][columna]

        # Convertimos el valor obtenido en binario de 4 bits y lo concatenamos al resultado
        resultado += decimal_a_binario(valor_s, 4)

    # Regresamos el bloque de 32 bits resultante
    return resultado
```

Función para permutar el bloque de 32 bits

```

def permutar_bloque_32bits(bloque_32bits):
    tabla_P = [
        [16, 7, 20, 21],
        [29, 12, 28, 17],
        [1, 15, 23, 26],
        [5, 18, 31, 10],
        [2, 8, 24, 14],
        [32, 27, 3, 9],
        [19, 13, 30, 6],
        [22, 11, 4, 25]
    ]

    # Aseguramos que el bloque de entrada tenga 32 bits
    if len(bloque_32bits) != 32:
        raise ValueError("El bloque debe ser de 32 bits.")

    # Aplicamos la permutación utilizando la tabla P
    permutacion = [bloque_32bits[pos - 1] for fila in tabla_P for pos in fila]

    # Devolvemos el bloque permutado como una cadena de 32 bits
    return ''.join(permutacion)

```

Función para aplicar XOR a bloques de 32 bits

```

def aplicar_xor_32bits(bloque1, bloque2):
    # Aseguramos que ambos bloques tengan 32 bits
    if len(bloque1) != 32 or len(bloque2) != 32:
        raise ValueError("Ambos bloques deben ser de 32 bits.")

    # Realizamos la operación XOR bit a bit entre los dos bloques
    resultado_xor = ''.join(
        str(int(bloque1[i]) ^ int(bloque2[i])) for i in range(32)
    )

    return resultado_xor

```

Función para aplicar la función f (de la explicación de DES)

```

def funcion_f(bloque_32bits, llave_48bits):
    # Aplicamos la expansión
    expansion = expandir_bloque(bloque_32bits)
    B = aplicar_xor_48bits(expansion, llave_48bits)
    C = aplicar_cajas_s(B)
    permutacion = permutar_bloque_32bits(C)
    return permutacion

```

Función para aplicar la permutación inversa

```

def permutacion_inversa(bits):
    # Permutacion inicial inversa
    tabla_IP_inversa = [
        [40, 8, 48, 16, 56, 24, 64, 32],
        [39, 7, 47, 15, 55, 23, 63, 31],
        [38, 6, 46, 14, 54, 22, 62, 30],
        [37, 5, 45, 13, 53, 21, 61, 29],
        [36, 4, 44, 12, 52, 20, 60, 28],
        [35, 3, 43, 11, 51, 19, 59, 27],
        [34, 2, 42, 10, 50, 18, 58, 26],
        [33, 1, 41, 9, 49, 17, 57, 25]
    ]

    # Aseguramos que la cadena de bits tenga 64 bits
    if len(bits) != 64:
        raise ValueError("La cadena de bits debe tener 64 bits.")

    # Aplicamos la permutación inversa usando la tabla_IP_inversa
    permutacion = [bits[pos - 1] for fila in tabla_IP_inversa for pos in fila]

    return ''.join(permutacion)

```

Función para revisar los bits de paridad del bloque e ignorarlos

```

def ignorar_bits_paridad(bloque_64):
    # Verificamos que el bloque tenga exactamente 64 bits
    if len(bloque_64) != 64 or not all(b in '01' for b in bloque_64):
        raise ValueError("El bloque debe ser una cadena de 64 bits de '0' y '1'.")

    bloques_procesados = []

    # Dividimos en 8 bloques de 8 bits
    for i in range(0, 64, 8):
        bloque_8 = bloque_64[1:i+8]

        # Eliminamos el bit de paridad (último bit)
        bloque_7 = bloque_8[:-1]

        # Contamos la cantidad de 1's en el bloque de 7 bits
        cantidad_1s = bloque_7.count('1')

        # Agregamos 0 si la cantidad de 1's es impar, 1 si es par
        if cantidad_1s % 2 == 0:
            bloque_7 += '1' # Par, agregamos 1
        else:
            bloque_7 += '0' # Impar, agregamos 0

        # Agregamos el bloque procesado a la lista
        bloques_procesados.append(bloque_7)

    return ''.join(bloques_procesados)

```

Función para aplicar la caja PC-1

```

def aplicar_PC_1(bloque_64bits):
    tabla_PC_1 = [
        [57, 49, 41, 33, 25, 17, 9],
        [1, 58, 50, 42, 34, 26, 18],
        [10, 2, 59, 51, 43, 35, 27],
        [19, 11, 3, 60, 52, 44, 36],
        [63, 55, 47, 39, 31, 23, 15],
        [7, 62, 54, 46, 38, 30, 22],
        [14, 6, 61, 53, 45, 37, 29],
        [21, 13, 5, 28, 20, 12, 4]
    ]

    # Asegúrate de que la cadena de bits tenga 64 bits
    if len(bloque_64bits) != 64:
        raise ValueError("La cadena de bits debe tener 64 bits.")

    # Aplicamos la permutación inicial usando la tabla_PC_1
    permutacion = [bloque_64bits[pos - 1] for fila in tabla_PC_1 for pos in fila]
    return ''.join(permutacion)

```

Función para obtener la llave inicial

```
def llave_inicial(llave_64bits):
    return aplicar_PC_1(ignorar_bits_paridad(llave_64bits))
```

Función para desplazar bits a la izquierda

```
def desplazar_bits_izquierda(cadena_56bits, numero_desplazamientos):
    # Verificamos que la cadena sea de 56 bits
    if len(cadena_56bits) != 56 or not all(b in '01' for b in cadena_56bits):
        raise ValueError("La cadena debe ser una cadena de 56 bits de '0' y '1'.")  

    # Si el valor es 1, 2, 9 o 16, se desplaza solo 1 posición a la izquierda
    if numero_desplazamientos in [1, 2, 9, 16]:
        desplazamiento = 1
    else:
        desplazamiento = 2  

    # Desplazamiento circular a la izquierda
    cadena_desplazada = cadena_56bits[desplazamiento:] + cadena_56bits[:desplazamiento]
    return cadena_desplazada
```

Función para aplicar la caja PC-2

```
def aplicar_PC_2(cadena_56bits):
    tabla_PC_2 = [
        [14, 17, 11, 24, 1, 5],
        [3, 28, 15, 6, 21, 10],
        [23, 19, 12, 4, 26, 8],
        [16, 7, 27, 20, 13, 2],
        [41, 52, 31, 37, 47, 55],
        [30, 40, 51, 45, 33, 48],
        [44, 49, 39, 56, 34, 53],
        [46, 42, 50, 36, 29, 32]
    ]  

    # Verificamos que la cadena sea de 56 bits
    if len(cadena_56bits) != 56 or not all(b in '01' for b in cadena_56bits):
        raise ValueError("La cadena debe ser una cadena de 56 bits de '0' y '1'.")  

    # Aplicamos la permutación de la tabla PC-2
    permutacion = [cadena_56bits[pos - 1] for fila in tabla_PC_2 for pos in fila]  

    # Retornamos la cadena permutada de 48 bits
    return ''.join(permutacion)
```

Función para pasar de hexadecimal a binario

```
def hex_a_bin(hexadecimal):
    # Convertir cada carácter hexadecimal a su representación binaria de 4 bits
    return bin(int(hexadecimal, 16))[2:]:.zfill(len(hexadecimal) * 4)
```

Función para cifrar un bloque de 64 bits (sería el proceso que cifra usando DES)

```

def cifrar_bloque(bloque_64bits, llave_64bits):
    # Aplicamos la permutación
    bloque_permutacion_inicial = permutacion_inicial(bloque_64bits)

    # Dividimos el bloque
    mitad = len(bloque_permutacion_inicial) // 2
    l = bloque_permutacion_inicial[:mitad]
    r = bloque_permutacion_inicial[mitad:]

    # Obtenemos los bloques iniciales c_0d_0 de la llave
    c_id_i = llave_inicial(llave_64bits)

    for i in range(16):
        c_id_i = desplazar_bits_izquierda(c_id_i, i+1)
        k_i = aplicar_PC_2(c_id_i)
        l, r = r, aplicar_xor_32bits(l, funcion_f(r, k_i))

    y = permutacion_inversa(r+l)

    return y

```

Función para descifrar un bloque de 64 bits (sería el proceso que descifra usando DES)

```

def descifrar_bloque(bloque_64bits, llave_64bits):
    # Aplicamos la permutación
    bloque_permutacion_inicial = permutacion_inicial(bloque_64bits)

    # Dividimos el bloque
    mitad = len(bloque_permutacion_inicial) // 2
    l = bloque_permutacion_inicial[:mitad]
    r = bloque_permutacion_inicial[mitad:]

    # Obtenemos los bloques iniciales c_0d_0 de la llave
    c_id_i = llave_inicial(llave_64bits)

    # Obtenemos las llaves
    subllaves = []
    for i in range(16):
        c_id_i = desplazar_bits_izquierda(c_id_i, i+1)
        k_i = aplicar_PC_2(c_id_i)
        subllaves.append(k_i)

    # Proceso inverso
    for k_i in reversed(subllaves):
        l, r = r, aplicar_xor_32bits(l, funcion_f(r, k_i))

    # Aplicamos la permutación inversa y regresamos el bloque descifrado
    y = permutacion_inversa(r + l)

    return y

```

Funciones auxiliares para convertir mensajes a bits y bits a mensajes

```

def mensaje_a_bits(mensaje):
    bits = ''
    for letra in mensaje:
        bits += format(ord(letra), '08b')
    return bits

def bits_a_mensaje(bits):
    step = 8
    mensaje = ''
    for i in range(0, len(bits), step):
        bloque = bits[i:i+step]
        mensaje += chr(int(bloque, 2))
    return mensaje

```

Función auxiliar para dividir una cadena en bloques

```
def dividir_en_bloques(cadena, k, x):
    """
        Divide una cadena en bloques

    Parameters :
    -----
        cadena:
            cadena a dividir
        k:
            longitud de los bloques
        x:
            relleno de los bloques cuando no sean de longitud k

    Returns :
    -----
        lista con el mensaje dividido
    """
    bloques = []
    for i in range(0, len(cadena), k):
        if (len(cadena) - i < k):
            bloques.append(cadena[i:] + x * (k - (len(cadena) - i)))
        else:
            bloques.append(cadena[i:i+k])
    return bloques
```

Función para poder cifrar un mensaje dada una llave (es la aplicación de cifrar con DES, solo que facilitada para poder usar solo una función)

```
def cifrar(mensaje, llave):
    # Dividimos el mensaje en bloques de 8 letras (64 bits)
    bloques = dividir_en_bloques(mensaje, 8, '\x0e') # \x0e es un carácter no imprimible
    # Convertimos los bloques a bits
    bits = ''.join([mensaje_a_bits(bloque) for bloque in bloques])
    # Ciframos el mensaje
    mensaje_cifrado = ''
    for bloque in dividir_en_bloques(bits, 64, ''):
        mensaje_cifrado += cifrarBloque(bloque, llave)
    return bits_a_mensaje(mensaje_cifrado)
```

Función para poder descifrar un mensaje dada una llave (es la aplicación de descifrar con DES, solo que facilitada para poder usar solo una función)

```
def descifrar(mensaje, llave):
    if (len(mensaje) * 8) % 64 != 0:
        raise ValueError('La longitud del mensaje en bits no es múltiplo de 64')
    # Convertimos el mensaje a bits
    bits = mensaje_a_bits(mensaje)
    # Desciframos el mensaje
    mensaje_descifrado = ''
    for bloque in dividir_en_bloques(bits, 64, ''):
        mensaje_descifrado += descifrarBloque(bloque, llave)
    return bits_a_mensaje(mensaje_descifrado)
```

Ya que la tarea moral sólo abarcaba el cifrado, se tuvo que investigar como hacer el descifrado y se encontró que solo es el proceso inverso. El proceso para cifrar es: dada una cadena X se le aplica la permutación inicial, obteniendo X_0 , y se obtienen dos partes de la cadena resultante $X = L_0 R_0$, luego de hacen 16 iteraciones donde se calculan las L_i y R_i , estás siendo $L_i = R_{i-1}$ y $R_i = f(K_i, R_{i-1}) \oplus L_{i-1}$,

con f siendo la función dada en clase y K_i la i -ésima (es generada) y por último se aplica la permutación inversa a $R_{16} L_{16}$ y se obtiene el mensaje cifrado.

El proceso para descifrar solo es seguir los pasos anteriores solo en el orden contrario, primero aplicar la permutación inicial, luego aplicar las llaves pero en el orden inverso y por último aplicar la permutación inversa.

Las funciones auxiliares (para el manejo de los mensajes) fueron extraídas/obtenidas de trabajos y prácticas anteriores.

Algo importante de notar es que si el mensaje que se quiere cifrar no se puede dividir en bloques de 64 bits entonces lo que hace nuestro código es llenar el mensaje con el carácter “\x0e” (esto a sugerencia de ChatGPT ya que segun es un ejemplo en la codificación PKCS7), también notamos que este carácter no es imprimible.

Un ejemplo de la ejecución del programa es la siguiente (recordemos que el mensaje descifrado puede tener caracteres extras, los usados para llenar)

```

488 |     mensaje = "El patito miraba el cielo en busca de respuestas pero cuando el cielo le respondio el patito ya no estaba"
489 |     clave = hex_a_bin("0123456789ABCDEF")
490 |     print("Mensaje original:", mensaje)
491 |     mensaje_cifrado = cifrar(mensaje, clave)
492 |     print("Mensaje cifrado:", mensaje_cifrado)
493 |     mensaje_descifrado = descifrar(mensaje_cifrado, clave)
494 |     print("Mensaje descifrado:", mensaje_descifrado)
495 |

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    COMMENTS

● camilo@womi:~/CC/septimoSemestre/Crip/Cripto-2025-1$ /bin/python /home/camilo/CC/septimoSemestre/Crip/Cripto-2025-1/Tarea/Tarea2/Ejercicio2/des.py
Mensaje original: El patito miraba el cielo en busca de respuestas pero cuando el cielo le respondio el patito ya no estaba
Mensaje cifrado: %C Ú'N±ÉÙ&T2'Í\Ù·ä/e:E§CðåíÙTç«,ÚoCCýür&hjAdø°ðAáé*ÁDòWAlV`òxtÀë4JB¶µ-Ö
Mensaje descifrado: El patito miraba el cielo en busca de respuestas pero cuando el cielo le respondio el patito ya no estaba
○ camilo@womi:~/CC/septimoSemestre/Crip/Cripto-2025-1$ 
```

- Usando tu implementación, usa las llaves débiles y semi débiles . ¿Por qué es mala idea usar estas llaves?

En el archivo des.py y ejercicio2.py están las llaves débiles y semi débiles, además de algunas ejecuciones para las pruebas.

```

# Prueba llaves debiles
for k in llaves_debiles:
    mensaje_cifrado = cifrar(mensaje, k)
    mensaje_descifrado = descifrar(mensaje_descifrado, k)

# Prueba llaves semidebiles
for k in llaves_semidebiles:
    mensaje_cifrado = cifrar(mensaje, k)
    mensaje_descifrado = descifrar(mensaje_descifrado, k)

# Prueba llaves posiblemente debiles
for k in llaves_posiblemente_debiles:
    mensaje_cifrado = cifrar(mensaje, k)
    mensaje_descifrado = descifrar(mensaje_descifrado, k)

```

Además en mi entrega de la tarea moral relacionada DES puse unos archivos donde se ven las llaves generadas para cada llave débil, semi

débil y posiblemente débiles, se recomienda revisar esos archivos pero como son algo largos se omitirán de esta tarea para no aumentar tanto el tamaño de los archivos.

Para las llaves débiles podemos notar que las dos primeras (estamos usando el orden de las llaves dadas en la tarea moral) no son buenas debido a que generar sub llaves todas iguales (puros 1s y puros 0s), lo cual no es muy bueno ya que los procesos van a ser muy similares al ser las mismas sub llaves, y las últimas dos llaves tiene un gran cantidad de 0s (o 1s) seguidos luego de la mitad de subllave, lo cual tampoco es tan bueno ya que hace que las sub llaves demasiado similares.

Por ejemplos estas son las sub llaves generadas por la primera llave (imagen de los archivos adjuntos en la tarea moral, favor de revisarlos), podemos notar que las sub llaves tienen grandes partes con puros 0s, algo similar pasa con las demás llaves débiles, excepto en el caso de la última llave débil donde algunas sub llaves se repiten.

```
Llave en hexadecimal: 1f1f01010e0e0101
Llave en binario: 00011110001111000000010000000010000111000001110000000100000001
Texto original: 00000001001000110100010101100111100010011010101110011011101111
Texto cifrado: 00010011101101111010110111100100000110101101100011011010111011
Subllaves:
K_1: 000000000000000000000000000000001011101000001010111000
K_2: 000000000000000000000000000000001001001001100010011111010
K_3: 000000000000000000000000000000001101101100110110000101
K_4: 00000000000000000000000000000000100100100110010011111010
K_5: 000000000000000000000000000000001101101100110110000101
K_6: 00000000000000000000000000000000100100100110011111010
K_7: 000000000000000000000000000000001101101100110110000101
K_8: 00000000000000000000000000000000100100100110010011111010
K_9: 00000000000000000000000000000000110100010111110101000111
K_10: 000000000000000000000000000000001011101000001010111000
K_11: 00000000000000000000000000000000110100010111110101000111
K_12: 000000000000000000000000000000001011101000001010111000
K_13: 00000000000000000000000000000000110100010111110101000111
K_14: 000000000000000000000000000000001011101000001010111000
K_15: 00000000000000000000000000000000110100010111110101000111
K_16: 000000000000000000000000000000001101101100110110000101
```

Entonces notamos que estas llaves débiles son malas de usar ya que generan cifrados inseguros y débiles, otra cosa interesante de estas llaves es que si le aplicamos una llave débil a un mensaje y luego al mensaje cifrado le aplicamos la misma llave obtenemos el mensaje original, esto debido a que las sub llaves generadas son iguales por lo tanto aplicando la segundo vez el algoritmo de cifrado sería como aplicar el algoritmo de descifrado, y esto pasa para las llaves débiles, generando que no sean buenas y tengan poca seguridad. Es decir si tenemos la llave débil K_i , pasara esto $E_{k_i}(E_{k_i}(M)) = M$, con M un mensaje y $E_{K_n}(M)$ es cifrar con la llave K_n .

Para las llaves semi débiles podemos notar que varias de las llaves siguen generando sub llaves con una gran cantidad de 0s (o 1s) seguidos y también algunas sub llaves que son las mismas (por ejemplo con fe01fe01fe01fe01 (imagen de los archivos adjuntos en la

tarea moral, favor de revisarlos), tenemos que las sub llaves 2 a 8 y las 16 son la misma) y en otras llaves como 1fe01fe01fe01fe0 podemos observar que la estructura de las sub llaves generadas son muy similares entre ellas, solo con pequeños cambios en algunos dígitos.

```
Llave en hexadecimal: fe01fe01fe01fe01
Llave en binario: 1111111000000001111111100000000111111110000000011111111000000001
Texto original: 000000010010011010001010110011100010011010101110011011101111
Texto cifrado: 1111111001000100111000110011000100001111110110011000011001001111
Subllaves:
K_1: 0110111010101100000110101111001110011001000010
K_2: 10010001010100111100101010000110001100110111101
K_3: 10010001010100111100101010000110001100110111101
K_4: 10010001010100111100101010000110001100110111101
K_5: 10010001010100111100101010000110001100110111101
K_6: 10010001010100111100101010000110001100110111101
K_7: 10010001010100111100101010000110001100110111101
K_8: 10010001010100111100101010000110001100110111101
K_9: 0110111010101100000110101111001110011001000010
K_10: 0110111010101100000110101111001110011001000010
K_11: 0110111010101100000110101111001110011001000010
K_12: 0110111010101100000110101111001110011001000010
K_13: 0110111010101100000110101111001110011001000010
K_14: 0110111010101100000110101111001110011001000010
K_15: 0110111010101100000110101111001110011001000010
K_16: 10010001010100111100101010000110001100110111101
```

En general podemos notar que estas llaves solo producen dos subclaves diferentes. Además investigamos que en las llaves semi débiles podemos encontrar pares de llaves tales que aplicadas dos veces (primero una llave al mensaje y luego otra al mensaje cifrado) obtenemos el mensaje original, esto pasa ya que las sub llaves que generan ambas llaves de la pareja son las mismas, entonces pasa algo similar con las llaves débiles, donde aplicar el proceso de cifrado con las mismas sub llaves vamos a descifrar. Es decir si tenemos las pareja de llaves semi débiles K_i y K_j , pasara esto $E_{k_j}(E_{k_i}(M)) = M$, con

M un mensaje y $E_{K_n}(M)$ es cifrar con la llave K_n .

En la tarea moral también se mencionan llaves posiblemente débiles, así que daremos una breve explicación, investigamos que generan patrones que se repiten y también solamente se generan cuatro diferentes sub llaves de las 16, por lo tanto van a generar resultados que no son tan seguros.

Fuentes usadas para este ejercicio:

Clases

Amata, J. (2024, May 15). Symmetric vs. asymmetric encryption: Practical Python examples. Snyk. <https://snyk.io/blog/symmetric-vs-asymmetric-encryption-python/>

de, C. (2006, April 17). PKCS. Wikipedia.org; Wikimedia Foundation, Inc. <https://es.wikipedia.org/wiki/PKCS>

KonradDos. (2018, October 30). DES encrypt file less than 64 bytes. Stack Overflow. <https://stackoverflow.com/questions/53073550/des-encrypt-file-less-than-64-bytes>

Xandros. (2012, August 30). Python “\x0e” in character by character XOR encryption. Stack Overflow.

<https://stackoverflow.com/questions/12198826/python-x0e-in-character-by-character-xor-encryption>

<https://eitca.org/cybersecurity/eitc-is-ccf-classical-cryptography-fundamentals/des-block-cipher-cryptosystem/data-encryption-standard-des-key-schedule-and-decryption/examination-review-data-encryption-standard-des-key-schedule-and-decryption/describe-the-process-of-decrypting-a-ciphertext-using-the-des-algorithm/>

Weak key. (2024, January 15). Wikipedia.

https://en.wikipedia.org/wiki/Weak_key#Weak_keys_in_DES

SSLeay 0.9.0b docs. (2024). Umich.edu.

<https://public.websites.umich.edu/~x509/ssleay/des-weak.html>

Can you explain “weak keys” for DES? (n.d.). Cryptography Stack Exchange.

<https://crypto.stackexchange.com/questions/12214/can-you-explain-weak-keys-for-ds>

all. (2018, November 13). DES encryption - what happens if all sub-keys are the same? Computer Science Stack Exchange.

<https://cs.stackexchange.com/questions/100017/des-encryption-what-happens-if-all-sub-keys-are-the-same>

DATA ENCRYPTION ALGORITHM. (2024). Umsl.edu.

https://www.umsl.edu/~siegelj/information_theory/projects/des.netau.net/improvements%20in%20des.html

3. Implementa en código el algoritmo visto para el Símbolo de Jacobi y de Legendre.

El código se encuentran simbolos.py

El código implementado es el siguiente:

Algoritmo jacobi para el Símbolo de Jacobi (si p es primo, entonces es el Símbolo de Legendre)

```

# Algoritmo del Simbolo de Jacobi
# Si n es primo, es el simbolo de Legendre
def jacobi(a, n):
    if n < 3:
        raise ValueError(f"{n} debe ser mayor o igual a 3")
    if a < 0 or a >= n:
        a = a % n # Hacemos esto para que a este en el rango [0, n)
    # Paso 1, si a = 0, el simbolo es 0
    if a == 0:
        return 0
    # Paso 2, si a = 1, el simbolo es 1
    if a == 1:
        return 1
    # Paso 3, describir a como  $2^e * m$ , con m impar
    e, m = descomponer(a)
    # Paso 4, si e es par, s es 1
    if e % 2 == 0:
        s = 1
    # Paso 4, si n es congruente con 1 o 7 modulo 8, s es 1
    elif n % 8 == 1 or n % 8 == 7:
        s = 1
    # Paso 4, si n es congruente con 3 o 5 modulo 8, s es -1
    elif n % 8 == 3 or n % 8 == 5:
        s = -1
    # Paso 5, si n es congruente con 3 modulo 4 y m es congruente con 3 modulo 4, s es -s
    if n % 4 == 3 and m % 4 == 3:
        s = -s
    # Paso 6, q es n mod m
    q = n % m
    # Paso 7, si m es 1, el simbolo es s
    if m == 1:
        return s
    # Paso 7, en otro caso, el simbolo es s * jacobi(q, m)
    return s * jacobi(q, m)

```

Y una función auxiliar usada es:

Función para descomponer un entero en $2^e * m$ con m impar

```

# Metodo para describir un entero a como  $2^e * m$ , con m impar
def descomponer(a):
    e = 0
    while a % 2 == 0:
        a = a // 2
        e += 1
    return e, a

```

Fuentes usadas para este ejercicio:

Ayudantías

4. Determina si a es residuo cuadrático módulo n. Muestra tu procedimiento.

$$a = 6007, n = 1902$$

Notamos que 1902 no es primo, entonces usamos el Símbolo de Jacobi $\frac{6007}{1902}$, pero como 6007 es mayor a 1902, usaremos el Símbolo

de Jacobi $\frac{301}{1902}$, ya que 6007 módulo 1902 es 301, entonces el Símbolo de Jacobi $\frac{301}{1902}$ es -1, entonces. Pero nosotros buscamos cómo comprobar si es residuo cuadrático o no, esto lo hacemos iterando i de 0 a 1901 y revisando cuando es i al cuadrado módulo 1902 es 301, notando que 139 al cuadrado módulo 1902 nos da 301, por lo tanto 6007 si es un residuo cuadrático módulo 1902, por las propiedades vistas en la ayudantía.

```
12 # Ejercicio 1
13 # a = 6007, n = 1902
14 # Primero usamos el simbolo de Jacobi de 6007/1902, ya que 1902 no es primo
15 #print(jacobi(6007, 1902))
16 # Pero 6007 es mayor que 1902, por lo que calculamos el residuo de 6007 mod 1902
17 #print(6007 % 1902) # 301
18 # Ahora calculamos el simbolo de Jacobi de 301/1902
19 print(jacobi(301, 1902)) # -1
20 # Como el simbolo de Jacobi es -1, entonces creeriamos que 6007 no es un residuo cuadratico modulo 1902
21 # Pero notamos que 301 es un residuo cuadratico modulo 1902, ya que  $139^2$  es congruente con 301 modulo 1902
22 # 139 lo encontramos de la siguiente manera
23 for i in range(0, 1901):
24     if pow(i, 2, 1902) == 301:
25         print(i)
26 # Por lo tanto tenemos que 139 esta en  $\mathbb{Z}_{1902}$  y  $139^2$  es congruente con 301 modulo 1902 y como 301 es congruente con 6007 modulo 1902
27 #print(pow(139, 2, 1902)) # 301
28 # Entonces 6007 es un residuo cuadratico modulo 1902
29
```

b. $\alpha = 83$, $n = 593$

Notamos que 1902 es primo (revisando una lista de números primos), entonces usamos el Símbolo de Legendre $\frac{83}{593}$, que es 1, entonces sabemos que 83 está en Q_{593} , Q_{593} es el conjunto de los residuos cuadráticos módulo 593. Pero nosotros buscamos como comprobar esto, esto lo hacemos iterando i de 0 a 592 y revisando cuando es i^2 al cuadrado módulo 593 es 83, notando que 26 al cuadrado módulo 593 nos da 83, por lo tanto 83 si es un residuo cuadrático módulo 593.

```
31 # Ejercicio 2
32 # a = 83, n = 593
33 # Primero usamos el simbolo de Legendre de 83/593, ya que 593 es primo https://byjus.com/math/prime-numbers-from-1-to-1000/
34 print(jacobi(83, 593)) # 1
35 # Como el simbolo de Legendre es 1, entonces sabemos que 83 esta en Q_593, Q_593 es el conjunto de los residuos cuadraticos modulo 593
36 # Entonces 83 es un residuo cuadratico modulo 593, pero nos falta encontrar el x tal que x^2 es congruente con 83 modulo 593
37 # Lo encontramos de la siguiente manera
38 for i in range(0, 592):
39     if pow(i, 2, 593) == 83:
40         print(i)
41 # Por lo tanto tenemos que 26 esta en Z_593 y 26^2 es congruente con 83 modulo 593
42 #print(pow(26, 2, 593)) # 83
43 # Entonces 83 es un residuo cuadratico modulo 593
44
```

c. $\alpha = 3677176$, $n = 4568731$

Notamos que 4568731 es primo (revisando una lista de números primos), entonces usamos el Símbolo de Legendre $\frac{3677176}{4568731}$, que es 1,

entonces sabemos que 3677176 está en $Q_{4568731}$, $Q_{4568731}$ es el conjunto de los residuos cuadráticos módulo 4568731. Pero nosotros buscamos como comprobar esto, esto lo hacemos iterando i de 0 a 4568730 y revisando cuando es i al cuadrado módulo 4568731 es 3677176, notando que 23783 al cuadrado módulo 4568731 nos da 3677176, por lo tanto 3677176 si es un residuo cuadrático módulo 4568731.

```

46 # Ejercicio 3
47 # a = 3677176, n = 4568731
48 # Primero usamos el simbolo de Jacobi de 3677176/4568731, ya que 4568731 es primo https://www.bigprimes.net/archive/prime/320401 https://www.bigprimes
49 | print(jacobi(3677176, 4568731)) # 1
50 # Como el simbolo de Jacobi es 1, entonces sabemos que 3677176 esta en Q_4568731, Q_4568731 es el conjunto de los residuos cuadraticos modulo 4568731
51 # Entonces 3677176 es un residuo cuadratico modulo 4568731, pero nos falta encontrar el x tal que x^2 es congruente con 3677176 modulo 4568731
52 # Lo encontramos de la siguiente manera
53 | for i in range(0, 4568730):
54 |   if pow(i, 2, 4568731) == 3677176:
55 |     print(i)
56 # Por lo tanto tenemos que 23783 esta en Z_4568731 y 23783^2 es congruente con 3677176 modulo 4568731
57 | #print(pow(23783, 2, 4568731)) # 3677176
58 # Entonces 3677176 es un residuo cuadratico modulo 4568731
59 |

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```

● camilo@wowi:~/CC/septimoSemestre/Crip/Cripto-2025-1$ /bin/python /home/camilo/CC/septimoSemestre/Crip/Cripto-2025-1/Tarea/Tarea2/Ejercicio4/residuo.py
1
23783
4544948
○ camilo@wowi:~/CC/septimoSemestre/Crip/Cripto-2025-1$ 

```

d. $\alpha = 4568723$, $n = 4568731$

Notamos que 4568731 es primo (revisando una lista de números primos), entonces usamos el Símbolo de Legendre $\frac{4568723}{4568731}$, que es 1, entonces sabemos que 4568723 está en $Q_{4568731}$, $Q_{4568731}$ es el conjunto de los residuos cuadráticos módulo 4568731. Pero nosotros buscamos como comprobar esto, esto lo hacemos iterando i de 0 a 4568730 y revisando cuando es i al cuadrado módulo 4568731 es 4568723, notando que 2237208 al cuadrado módulo 4568731 nos da 4568723, por lo tanto 4568723 si es un residuo cuadrático módulo 4568731.

```

61 # Ejercicio 4
62 # a = 4568723, n = 4568731
63 # Primero usamos el simbolo de Jacobi de 4568723/4568731, ya que 4568731 es primo https://www.bigprimes.net/archive/prime/320401 https://www.bigprimes
64 | print(jacobi(4568723, 4568731)) # 1
65 # Como el simbolo de Jacobi es 1, entonces sabemos que 4568723 esta en Q_4568731, Q_4568731 es el conjunto de los residuos cuadraticos modulo 4568731
66 # Entonces 4568723 es un residuo cuadratico modulo 4568731, pero nos falta encontrar el x tal que x^2 es congruente con 4568723 modulo 4568731
67 # Lo encontramos de la siguiente manera
68 | for i in range(0, 4568730):
69 |   if pow(i, 2, 4568731) == 4568723:
70 |     print(i)
71 # Por lo tanto tenemos que 2237208 esta en Z_4568731 y 2237208^2 es congruente con 4568723 modulo 4568731
72 | #print(pow(2237208, 2, 4568731)) # 4568723
73 # Entonces 4568723 es un residuo cuadratico modulo 4568731
74 |

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```

● camilo@wowi:~/CC/septimoSemestre/Crip/Cripto-2025-1$ /bin/python /home/camilo/CC/septimoSemestre/Crip/Cripto-2025-1/Tarea/Tarea2/Ejercicio4/residuo.py
1
2237208
2331523
○ camilo@wowi:~/CC/septimoSemestre/Crip/Cripto-2025-1$ 

```

Todos estas llamadas a funciones y una pequeña explicación se encuentran en el archivo residuo.py, notemos que usamos las funciones del ejercicio anterior.

Fuentes usadas para este ejercicio:

Ayudantías

Prime Numbers from 1 to 1000 - Complete list. (n.d.). BYJUS.

<https://byjus.com/math/prime-numbers-from-1-to-1000/>

BigPrimes. (2024). Bigprimes.net. <https://www.bigprimes.net/cruncher/4568731>

BigPrimes. (2024). Bigprimes.net. <https://www.bigprimes.net/archive/prime/320401>

5. Implementa los algoritmos para calcular la raíz cuadrada de a módulo n.

Los algoritmos se encuentran `raices.py`

Los algoritmos implementados son los siguientes:

Algoritmo `raiz_cuadrada_p`

```
# Algoritmo para encontrar la raiz cuadrado modulo p, p primo
def raiz_cuadrada_p(a, p):
    if not es_primo(p):
        raise ValueError(f"{p} no es primo")
    if a < 0 or a >= p:
        a = a % p
    if p == 2:
        raise ValueError(f"{p} no es impar")
    # Paso 1, sacar el simbolo de Legendre de a/p
    s = jacobi(a, p)
    # Paso 1, si s = -1, no existe la raiz cuadrada, regresamos a
    if s == -1:
        #return a
        raise ValueError(f"No existe la raiz cuadrada de {a} modulo {p}")
    # Paso 2, seleccionar un entero b, tal que 1 <= b < p y el simbolo de Legendre de b/p = -1
    b = 1
    while jacobi(b, p) != -1:
        b = random.randint(1, p - 1)
    # Paso 3, buscar un s tal que p-1 = 2^s * t, con t impar y s >= 1
    s, t = descomponer(p - 1)
    # Paso 4, encontrar a^-1 mod p
    a_inv = inverso(a, p)
    # Paso 5, calculamos c = b^t mod p
    c = pow(b, t, p)
    # Paso 5, calculamos r = a^{(t+1)/2} mod p
    exp = (t + 1) // 2
    r = pow(a, exp, p)
    # Paso 6, iterar i de 1 a s-1
    for i in range(1, s):
        # Paso 6.1, calculamos d = (r^2 * a^-1)^2^(s-i-1) mod p
        exp = pow(2, s - i - 1)
        interno = pow(r, 2, p) * a_inv
        d = pow(interno, exp, p)
        # Paso 6.2, si d es congruente con -1 mod p, hacemos r = r * c mod p
        if d % p == p - 1:
            r = (r * c) % p
        # Paso 6.3, calculamos c = c^2 mod p
        c = pow(c, 2, p)
    # Paso 7, regresamos r mod p y -r mod p
    return (r % p, (-r) % p)
```

Algoritmo `raiz_3`

```

# Algoritmo para encontrar la raiz cuadrado modulo p, p primo y p = 3 mod 4
def raiz_3(a, p):
    if not es_primo(p):
        raise ValueError(f"{p} no es primo")
    if a < 0 or a >= p:
        a = a % p
    if p % 4 != 3:
        raise ValueError(f"{p} no es congruente con 3 modulo 4")
    # Paso 1, calculamos r = a^{(p+1)/4} mod p
    exp = (p + 1) // 4
    r = pow(a, exp, p)
    # Paso 2, regresamos r mod p y -r mod p
    return (r % p, (-r) % p)

```

Algoritmo raiz_5

```

# Algoritmo para encontrar la raiz cuadrado modulo p, p primo y p = 5 mod 8
def raiz_5(a, p):
    if not es_primo(p):
        raise ValueError(f"{p} no es primo")
    if a < 0 or a >= p:
        a = a % p
    if p % 8 != 5:
        raise ValueError(f"{p} no es congruente con 5 modulo 8")
    if p == 2:
        raise ValueError(f"{p} no es impar")
    # Paso 1, calculamos d = a^{(p-1)/4} mod p
    exp = (p - 1) // 4
    d = pow(a, exp, p)
    # Paso 2, si d = 1, calculamos r = a^{(p+3)/8} mod p
    if d == 1:
        exp = (p + 3) // 8
        r = pow(a, exp, p)
    # Paso 2, si d = p-1, calculamos r = 2 * a * (4a)^{(p-5)/8} mod p
    elif d == p - 1:
        exp = (p - 5) // 8
        r = 2 * a * pow(4 * a, exp, p) % p
    # Paso 3, regresamos r mod p y -r mod p
    return (r % p, (-r) % p)

```

Algoritmo raiz_grande

```

# Algoritmo para encontrar la raiz cuadrada modulo p, p primo, cuando la s en p-1 = 2^s * t es grande
def raiz_grande(a, p):
    if not es_primo(p):
        raise ValueError(f"{p} no es primo")
    if a < 0 or a >= p:
        a = a % p
    if p == 2:
        raise ValueError(f"{p} no es impar")
    # Paso 1, escogemos un entero b en Z_p, tal que el simbolo de Legendre de (b^2 - 4a)/p = -1
    b = 0
    while jacobi(pow(b, 2) - 4 * a, p) != -1:
        b = random.randint(0, p - 1)
    # Paso 2, sea f el polinomio x^2 - bx + a
    f = [1, -b, a]
    # Paso 3, calculamos r = x^(p+1)/2 mod f
    exp = (p + 1) // 2
    poli = [0] * (exp + 1)
    poli[0] = 1
    r = mod_poly(poli, f, p)
    r = r[0]
    return (r % p, (-r) % p)

```

Algoritmo raiz_cuadrada_n

```

# Algoritmo para encontrar la raiz cuadrada modulo n, n = p * q, con p y q primos
def raiz_cuadrada_n(a, n):
    if a < 0 or a >= n:
        a = a % n
    # Paso 0, factorizamos n
    if factorizar(n) == 0:
        raise ValueError(f"{n} debe ser un n = p * q, con p y q primos")
    p, q = factorizar(n)
    # Paso 1, encontramos las raices cuadradas de a modulo p
    r_1, r_2 = raiz_cuadrada_p(a, p)
    # Paso 2, encontramos las raices cuadradas de a modulo q
    s_1, s_2 = raiz_cuadrada_p(a, q)
    # Paso 3, encontramos c y d tal que c * q + d * p = 1
    _, c, d = euclides_extendido(p, q)
    # Paso 4, calculamos x = (r_1 * d * q + s_1 * c * p) mod n
    x = (r_1 * d * q + s_1 * c * p) % n
    # Paso 4, calculamos y = (r_1 * d * q - s_1 * c * p) mod n
    y = (r_1 * d * q - s_1 * c * p) % n
    # Paso 5, regresamos x mod n, -x mod n, y mod n, -y mod n
    return (x % n, (-x) % n, y % n, (-y) % n)

```

Algoritmo raiz_cuadrada_n_gen (este es una generalización de raiz_cuadrada_n)

```

# Algoritmo para encontrar las raices cuadradas módulo n para un n compuesto con múltiples factores primos
def raiz_cuadrada_n_gen(a, n):
    if a < 0 or a >= n:
        a = a % n
    if es_primo(n):
        return raiz_cuadrada_p(a, n)
    # Paso 0: Factorizamos n
    factores = factorizar_n(n)
    # Paso 1: Calculamos las raices cuadradas de a módulo cada factor primo
    raices_factor = []
    for p in factores:
        if p == 2:
            raices_factor.append(raiz_2(a))
        elif p == 3:
            raices_factor.append(raiz_3(a))
        else:
            raices_factor.append(raiz_cuadrada_p(a, p))
    # Paso 2: Iteramos sobre todas las combinaciones de las raices generadas modulo cada factor primo
    raices = []
    for combinacion in itertools.product(*raices_factor):
        # Paso 2.1: Calculamos una x que satisfaga las congruencias
        x = 0
        # Paso 2.2: Iteramos sobre cada factor primo
        for i, p in enumerate(factores):
            # Paso 2.2.1: Calculamos m = n // p
            m = n // p
            # Paso 2.2.2: Calculamos c = m^(-1) mod p
            c = inverso(m, p)
            # Paso 2.2.3: Agregamos a x el valor de la combinacion modulo p multiplicado por c y m
            # Para que x cumpla con la congruencia
            x += combinacion[i] * c * m
        # Paso 2.3: Agregamos x modulo n y -x modulo n a la lista de raices
        raices.append(x % n)
        raices.append((-x) % n)
    # Paso 3: Eliminamos duplicados y regresamos los resultados únicos
    return list(set(raices))

```

Y las funciones auxiliares usadas son (las relacionadas a polinomios se encuentran en poli.py):

Función para ver si un número es primo

```

# Funcion para ver si un numero es primo
def es_primo(n):
    # Encontramos los primos menores o iguales a n
    primos = primos_menores_iguales(n)
    # Si el numero es primo, regresamos True
    return n in primos

```

Función para obtener los primos menores o iguales de un número

```

# Funcion para encontrar los primos menores o iguales a n, Criba de Eratostenes
def primos_menores_iguales(n):
    # Paso 1, creamos una lista de booleanos de tamaño n
    primos = [True] * (n + 1)
    # Paso 2, iteramos sobre los numeros menores a n
    for i in range(2, n + 1):
        # Paso 2.1, si el numero es primo
        if primos[i]:
            # Paso 2.1.1, marcamos los multiplos de i como no primos
            for j in range(i * i, n + 1, i):
                primos[j] = False
    # Paso 3, regresamos los numeros primos
    return [i for i in range(2, n + 1) if primos[i]]

```

Función para obtener el mcd de dos números

```

def mcd(a,b):
    """
    Regresa el maximo comun divisor de dos numeros.
    """
    while b != 0:
        a = a % b
        # Swap
        aux = a
        a = b
        b = aux
    return a

```

Función para reducir un número a módulo n

```

def reduce(x,n):
    """
    Regresa un numero reducido a modulo n
    """
    negativo = x < 0
    numero_reducido = abs(x) % n
    return n - numero_reducido if negativo else numero_reducido

```

Función para hacer Euclides extendido

```

def euclides_extendido(a, b):
    """
    Regresa el maximo comun divisor de 'a' y 'b' junto con los coeficientes de la combinación lineal.
    """
    # Combinacion lineal
    # r = sa + tb
    s_0 = 1      # Coeficiente s
    s_1 = 0      # Carga con el divisor
    t_0 = 0      # Coeficiente t
    t_1 = 1      # Carga con el dividendo
    while b != 0:
        # Algoritmo de la division
        # a = bq + r
        q = int(a / b) # Cociente
        r = a % b       # Residuo
        # Swap
        a = b
        b = r
        # Actualizamos los coeficientes s y t (Swap)
        s_0, s_1 = s_1, s_0 - q * s_1
        t_0, t_1 = t_1, t_0 - q * t_1
    # Imprimimos la combinacion lineal: r = as + bt
    # print(f"{{q}} = {{x}}{{{s_0}}} + {{y}}{{{t_0}}}")
    # print(a,b,s_0,s_1,t_0,t_1)
    return a, s_0, t_0

```

Función para obtener el inverso de un número en un módulo

```

def inverso(x,n):
    """
    Regresa el inverso multiplicativo de un numero (si lo hay).
    """
    # Un numero tiene inverso multiplicativo si x,n son primos relativos (coprimos)
    if mcd(x,n) == 1:
        a, s, t = euclides_extendido(x,n)
        return reduce(s,n)
    else:
        raise ValueError(f"El inverso multiplicativo de {x} mod {n} no existe")

```

Función para factorizar un número en dos primos

```

# Funcion para factorizar un numero n en dos primos p y q
def factorizar(n):
    # Paso 1, obtenemos los primos menores a n
    primos = primos_menores(n)
    # Paso 2, iteramos sobre los primos
    for p in primos:
        for q in primos:
            # Paso 2.1, si p * q = n, regresamos p y q
            if p * q == n:
                return p, q
    # Paso 3, si no encontramos p y q, regresamos 0
    return 0

```

Función para obtener los primos menores de un número

```

# Funcion para encontrar los primos menores a n, Criba de Eratostenes
def primos_menores(n):
    # Paso 1, creamos una lista de booleanos de tamaño n
    primos = [True] * n
    # Paso 2, iteramos sobre los numeros menores a n
    for i in range(2, n):
        # Paso 2.1, si el numero es primo
        if primos[i]:
            # Paso 2.1.1, marcamos los multiplos de i como no primos
            for j in range(i * i, n, i):
                primos[j] = False
    # Paso 3, regresamos los numeros primos
    return [i for i in range(2, n) if primos[i]]

```

Función para factorizar un número en una lista de primos

```

# Funcion para factorizar un numero n en una lista de factores primos
# https://stackoverflow.com/questions/32871539/integer-factorization-in-python
def factorizar_n(n):
    factores = []
    factor = 2
    while factor <= n:
        if n % factor == 0:
            factores.append(factor)
            n = n // factor
        else:
            factor += 1
    return factores

```

Función para encontrar raíces cuadradas en módulo 2

```

# Algoritmo para encontrar la raiz cuadrado modulo 2
def raiz_2(a):
    if a < 0 or a >= 2:
        a = a % 2
    if a == 0:
        return (0, 0)
    return (1, 1)

```

Función para encontrar raíces cuadradas en módulo 3

```
# Algoritmo para encontrar la raiz cuadrado modulo 3
def raiz_3(a):
    if a < 0 or a >= 3:
        a = a % 3
    if a == 0:
        return (0, 0)
    if a == 1:
        return (1, 2)
    raise ValueError(f"No existe la raiz cuadrada de {a} modulo 3")
```

Función para sumar dos polinomios

```
# Funcion para sumar dos polinomios en F_{p^n}
def suma_poli(f, g, p_n):
    if f == [0]:
        return g
    if g == [0]:
        return f
    l_1 = len(f)
    l_2 = len(g)
    if l_1 < l_2:
        f = [0] * (l_2 - l_1) + f
    elif l_2 < l_1:
        g = [0] * (l_1 - l_2) + g
    resultado = [(f[i] + g[i]) % p_n for i in range(max(l_1, l_2))]
    while resultado and resultado[0] == 0:
        resultado.pop(0)
    if resultado == []:
        resultado = [0]
    return resultado
```

Función para restar dos polinomios

```
# Funcion para restar dos polinomios en F_{p^n}
def resta_poli(f, g, p_n):
    if g == [0]:
        return f
    l_1 = len(f)
    l_2 = len(g)
    if l_1 < l_2:
        f = [0] * (l_2 - l_1) + f
    elif l_2 < l_1:
        g = [0] * (l_1 - l_2) + g
    resultado = [(f[i] - g[i]) % p_n for i in range(max(l_1, l_2))]
    while resultado and resultado[0] == 0:
        resultado.pop(0)
    if resultado == []:
        resultado = [0]
    return resultado
```

Función para multiplicar dos polinomios

```

# Funcion para multiplicar dos polinomios en F_{p^n}
def mult_poli(f, g, p_n):
    l_1 = len(f)
    l_2 = len(g)
    if f == [0] or g == [0]:
        return [0]
    if f == [1]:
        return g
    if g == [1]:
        return f
    resultado = [0] * (l_1 + l_2 - 1)
    for i in range(l_1):
        for j in range(l_2):
            resultado[i + j] += f[i] * g[j]
    resultado = [x % p_n for x in resultado]
    while resultado and resultado[0] == 0:
        resultado.pop(0)
    if resultado == []:
        resultado = [0]
    return resultado

```

Función para dividir dos polinomios

```

# Funcion para dividir dos polinomios en F_{p^n}, retorna el cociente y el residuo
# https://en.wikipedia.org/wiki/Polynomial_long_division#Pseudocode
def div_poli(f, g, p_n):
    if g == [0]:
        raise ValueError("No se puede dividir por 0")
    q = [0]
    r = f[:]
    while r != [0] and len(r) >= len(g):
        lead_r = r[0]
        lead_g = g[0]
        if lead_g == 0:
            raise ValueError("No se puede dividir por 0")
        coef = lead_r * pow(lead_g, -1, p_n) % p_n
        t = [coef] + [0] * (len(r) - len(g))
        while t and t[0] == 0:
            t.pop(0)
        if t == []:
            t = [0]
        q = suma_poli(q, t, p_n)
        aux = mult_poli(t, g, p_n)
        r = resta_poli(r, aux, p_n)
    return q, r

```

Función para exponentiar un polinomio

```

# Funcion para exponentiar un polinomio a la k en F_{p^n}
def exp_poli(f, k, p_n):
    if k == 0:
        return [1]
    if k == 1:
        return f
    resultado = f[:]
    for i in range(2, k + 1):
        resultado = mult_poli(resultado, f, p_n)
    return resultado

```

Función para hacer un polinomio modulo otro polinomio

```
# Funcion para realizar f(x) mod g(x) en F_{p^n}
def mod_poli(f, g, p_n):
    if g == [0]:
        raise ValueError("No se puede hacer modulo 0")
    residuo = f[:]
    l_1 = len(residuo)
    l_2 = len(g)
    if l_1 >= l_2:
        _, residuo = div_poli(residuo, g, p_n)
    while residuo and residuo[0] == 0:
        residuo.pop(0)
    if residuo == []:
        residuo = [0]
    return residuo
```

Fuentes usadas para este ejercicio:

Ayudantías

Integer factorization in python. (n.d.). Stack Overflow.
<https://stackoverflow.com/questions/32871539/integer-factorization-in-python>

6. Usa tus implementaciones para calcular las raíces de α módulo n.

- $\alpha = 55, n = 103$ (primo)

Para esto usaremos la función `raiz_cuadrada_p`, ya que 103 es un primo, entonces realizamos la llamada `raiz_cuadrada_p(55, 103)` que nos regresó 63, 40. Por lo tanto tenemos que las raíces son 63 y 40.

Para comprobar esto usamos `pow(63, 2, 103)` y `pow(40, 2, 103)`, que ambos nos regresaron 55.

Una observación es que pudimos usar la función `raiz_3`, ya que 103 es congruente con 3 módulo 4, y también regresa que las raíces son 63 y 40.

```
8 # Ejercicio 1
9 # a = 55, n = 103 (primo)
10 # Usaremos el metodo de raiz_cuadrada_p
11 print(raiz_cuadrada_p(55, 103)) # 63, 40
12 # Verificamos que 63^2 es congruente con 55 modulo 103
13 #print(pow(63, 2, 103)) # 55
14 # Verificamos que 40^2 es congruente con 55 modulo 103
15 #print(pow(40, 2, 103)) # 55
16 # Entonces 55 tiene las raices 63 y 40 en modulo 103
17 # Tambien notamos que 103 es congruente con 3 modulo 4, por lo que podemos usar el metodo de raiz_3
18 #print(raiz_3(55, 103)) # 63, 40
19 # Regresando los mismos resultados
20

PROBLEMS (2) OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

camilo@wowi:~/CC/septimoSemestre/Crip/Cripto-2025-1$ /bin/python /home/camilo/CC/septimoSemestre/Crip/Cripto-2025-1/Tarea/Tarea2/Ejercicio6/calc_raices.py
(63, 40)
camilo@wowi:~/CC/septimoSemestre/Crip/Cripto-2025-1$ 
```

- $\alpha = 76, n = 102$

Para esto usaremos la función `raiz_cuadrada_n_gen`, ya que 102 es un número compuesto y su factorización es 2 por 3 por 17, entonces realizamos la llamada `raiz_cuadrada_n_gen(76, 102)` que nos regresó 56, 80, 46, 22. Por lo tanto tenemos que las raíces son 22, 46, 56 y 80. Para comprobar esto usamos `pow(56, 2, 102)`, `pow(80, 2, 102)`, `pow(46, 2, 102)` y `pow(22, 2, 102)`, que todos nos regresaron 76.

```
21
22 # Ejercicio 2
23 # a = 76, n = 102
24 # Usaremos el metodo de raiz_cuadrada_n_gen, ya que 102 se factoriza en 2 * 3 * 17
25 print(raiz_cuadrada_n_gen(76, 102)) # 56, 80, 46, 22
26 # Verificamos que 56^2 es congruente con 76 modulo 102
27 print(pow(56, 2, 102)) # 76
28 # Verificamos que 80^2 es congruente con 76 modulo 102
29 print(pow(80, 2, 102)) # 76
30 # Verificamos que 46^2 es congruente con 76 modulo 102
31 print(pow(46, 2, 102)) # 76
32 # Verificamos que 22^2 es congruente con 76 modulo 102
33 print(pow(22, 2, 102)) # 76
34 # Entonces 76 tiene las raices 22, 46, 56 y 80 en modulo 102
35
```

c. $\alpha = 161$, $n = 211$ (primo)

Para esto usaremos la función `raiz_cuadrada_p`, ya que 211 es un primo, entonces realizamos la llamada `raiz_cuadrada_p(161, 211)` que nos regresó 43, 168. Por lo tanto tenemos que las raíces son 43 y 168. Para comprobar esto usamos `pow(43, 2, 211)` y `pow(168, 2, 211)`, que ambos nos regresaron 161.

```
32
33 # Ejercicio 3
34 # a = 161, n = 211 (primo)
35 # Usaremos el metodo de raiz_cuadrada_p
36 print(raiz_cuadrada_p(161, 211)) # 43, 168
37 # Verificamos que  $43^2$  es congruente con 161 modulo 211
38 #print(pow(43, 2, 211)) # 161
39 # Verificamos que  $168^2$  es congruente con 161 modulo 211
40 #print(pow(168, 2, 211)) # 161
41 # Entonces 161 tiene las raices 43 y 168 en modulo 211
42
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
camilo@woowi:~/CC/septimoSemestre/Crip/Cripto-2025-1$ /bin/python /home/camilo/CC/septimoSemestre/Crip/Cripto-2025-1/Tarea/Tarea2/Ejercicio6/calc_raices.py
(43, 168)
camilo@woowi:~/CC/septimoSemestre/Crip/Cripto-2025-1$
```

d. $\alpha = 133$, $n = 177$

Para esto usaremos la función `raiz_cuadrada_n`, ya que 177 es un número compuesto y su factorización es 3 por 59, entonces realizamos la llamada `raiz_cuadrada_n(133, 177)` que nos regresó 88, 89, 148, 29. Por lo tanto tenemos que las raíces son 29, 88, 89 y 148.

Para comprobar esto usamos `pow(88, 2, 177)`, `pow(89, 2, 177)`, `pow(148, 2, 177)` y `pow(29, 2, 177)`, que todos nos regresaron 133.

```

44  # Ejercicio 4
45  # a = 133, n = 177
46  # Usaremos el metodo de raiz_cuadrada_n
47  print(raiz_cuadrada_n(133, 177)) # 88, 89, 148, 29
48  # Verificamos que 88^2 es congruente con 133 modulo 177
49  #print(pow(88, 2, 177)) # 133
50  # Verificamos que 89^2 es congruente con 133 modulo 177
51  #print(pow(89, 2, 177)) # 133
52  # Verificamos que 148^2 es congruente con 133 modulo 177
53  #print(pow(148, 2, 177)) # 133
54  # Verificamos que 29^2 es congruente con 133 modulo 177
55  #print(pow(29, 2, 177)) # 133
56  # Entonces 133 tiene las raices 29, 88, 89 y 148 en modulo 177
57
58

```

PROBLEMS 2 DEBUG CONSOLE TERMINAL PORTS COMMENTS

- camilo@wofi:~/CC/septimoSemestre/Crip/Cripto-2025-1\$ /bin/python /home/camilo/CC/septimoSemestre/Crip/Cripto-2025-1/Tarea/Tarea2/Ejercicio6/calc_raices.py
- (88, 89, 148, 29)

Todos estas llamadas a funciones y una pequeña explicacion se encuentran en el archivo calc_raices.py

Fuentes usadas para este ejercicio:

Ayudantías

7. Encuentra el máximo común divisor de los polinomios

$$f(x) = x^6 + x^5 + x^4 + 1 \text{ y } h(x) = x^5 + x^3 + x^2 + 1 \text{ en } Z_2[x]$$

Para este ejercicio usamos unas funciones que programamos para realizar operaciones en polinomios, estas se encuentran en poli.py (en la carpeta Ejercicio 5).

Las funciones que usamos son los siguientes:

Función para encontrar el maximo comun divisor

```
# Funcion para encontrar el maximo comun divisor de dos polinomios en F_{p^n}
def mcd_poli(f, g, p_n):
    while g != [0]:
        f, g = g, div_poli(f, g, p_n)[1]
    return f
```

Función para dividir dos polinomios

```

# Funcion para dividir dos polinomios en F_{p^n}, retorna el cociente y el residuo
# https://en.wikipedia.org/wiki/Polynomial_long_division#Pseudocode
def div_poli(f, g, p_n):
    if g == [0]:
        raise ValueError("No se puede dividir por 0")
    q = [0]
    r = f[:]
    while r != [0] and len(r) >= len(g):
        lead_r = r[0]
        lead_g = g[0]
        if lead_g == 0:
            raise ValueError("No se puede dividir por 0")
        coef = lead_r * pow(lead_g, -1, p_n) % p_n
        t = [coef] + [0] * (len(r) - len(g))
        while t and t[0] == 0:
            t.pop(0)
        if t == []:
            t = [0]
        q = suma_poli(q, t, p_n)
        aux = mult_poli(t, g, p_n)
        r = resta_poli(r, aux, p_n)
    return q, r

```

Función para multiplicar dos polinomios

```

# Funcion para multiplicar dos polinomios en F_{p^n}
def mult_poli(f, g, p_n):
    l_1 = len(f)
    l_2 = len(g)
    if f == [0] or g == [0]:
        return [0]
    if f == [1]:
        return g
    if g == [1]:
        return f
    resultado = [0] * (l_1 + l_2 - 1)
    for i in range(l_1):
        for j in range(l_2):
            resultado[i + j] += f[i] * g[j]
    resultado = [x % p_n for x in resultado]
    while resultado and resultado[0] == 0:
        resultado.pop(0)
    if resultado == []:
        resultado = [0]
    return resultado

```

Función para restar dos polinomios

```
# Funcion para restar dos polinomios en F_{p^n}
def resta_poli(f, g, p_n):
    if g == [0]:
        return f
    l_1 = len(f)
    l_2 = len(g)
    if l_1 < l_2:
        f = [0] * (l_2 - l_1) + f
    elif l_2 < l_1:
        g = [0] * (l_1 - l_2) + g
    resultado = [(f[i] - g[i]) % p_n for i in range(max(l_1, l_2))]
    while resultado and resultado[0] == 0:
        resultado.pop(0)
    if resultado == []:
        resultado = [0]
    return resultado
```

Función para sumar dos polinomios

```
# Funcion para sumar dos polinomios en F_{p^n}
def suma_poli(f, g, p_n):
    if f == [0]:
        return g
    if g == [0]:
        return f
    l_1 = len(f)
    l_2 = len(g)
    if l_1 < l_2:
        f = [0] * (l_2 - l_1) + f
    elif l_2 < l_1:
        g = [0] * (l_1 - l_2) + g
    resultado = [(f[i] + g[i]) % p_n for i in range(max(l_1, l_2))]
    while resultado and resultado[0] == 0:
        resultado.pop(0)
    if resultado == []:
        resultado = [0]
    return resultado
```

Entonces lo primero fue representar los polinomios para que puedan ser usados en nuestras funciones.

Obteniendo $f = [1, 1, 1, 0, 0, 0, 1]$ y $g = [1, 0, 1, 1, 0, 1]$

Después realizamos la llamada `mcd_poli(f, g, 2)`, que nos regresó $[1, 1]$, por lo cual significa que el máximo común divisor es $x + 1$.

```
12
13 # Primero transformamos los polinomios a su representacion en listas para poder usar las funciones
14 f = [1, 1, 1, 0, 0, 0, 1]
15 g = [1, 0, 1, 1, 0, 1]
16 # Usamos mcd_poli para encontrar el maximo comun divisor
17 print(mcd_poli(f, g, 2)) # [1, 1]
18 # Transformamos el resultado a su representacion en polinomios
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
> camilo@wowi:~/CC/septimoSemestre/Crip/Cripto-2025-1$ /bin/python /home/camilo/CC/septimoSemestre/Crip/Cripto-2025-1/Tarea/Tarea2/Ejercicio7/mcd_poli.py
[1, 1]
> camilo@wowi:~/CC/septimoSemestre/Crip/Cripto-2025-1$
```

Otra manera que podemos usar para encontrar el máximo común divisor es hacer divisiones entre los polinomios hasta llegar a un residuo 0.

Las llamadas fueron las siguientes

La primera llamada fue `div_poli(f, g, 2)`, que nos regreso $([1, 1], [1, 1, 0])$ lo que significa que el cociente es $x + 1$ y el residuo es $x^2 + x + 1$.

La segunda llamada fue `div_poli(g, [1, 1, 0], 2)`, que nos regreso $([1, 1, 0, 1], [1, 1])$ lo que significa que el cociente es $x^3 + x^2 + 1$ y el residuo es $x + 1$.

La última llamada fue `div_poli([1, 1, 0], [1, 1], 2)`, que nos regreso `([1, 0], [0])` lo que significa que el cociente es x y el residuo es 0, por lo tanto ya llegamos a un residuo 0 y tenemos que el mcd es $x + 1$.

Entonces el maximo comun divisor de $f(x) = x^6 + x^5 + x^4 + 1$ y $h(x) = x^5 + x^3 + x^2 + 1$ en $Z_2[x]$ es $x + 1$.

Todos estas llamadas a funciones y una pequeña explicacion se encuentran en el archivo `mcd_poli.py`

Fuentes usadas para este ejercicio:

Wikipedia Contributors. (2024, October 22). Polynomial long division. Wikipedia; Wikimedia Foundation.

https://en.wikipedia.org/wiki/Polynomial_long_division#Pseudocode

8. Completa la tabla generadora de $GF(2^4)$ usando $m = x^4 + x + 1$

$GF(2^4)$ con $m = x^4 + x + 1$

Representación en Potencia	Representación Polinomial	Representación Binaria	Representación Decimal
0	0	0000	0
g^0	1	0001	1
g^1	x^1	0010	2
g^2	x^2	0100	4
g^3	x^3	1000	8
g^4	$x^1 + 1$	0011	3
g^5	$x^2 + x^1$	0110	6
g^6	$x^3 + x^2$	1100	12
g^7	$x^3 + x^1 + 1$	1011	11
g^8	$x^2 + 1$	0101	5
g^9	$x^3 + x^1$	1010	10
g^{10}	$x^2 + x^1 + 1$	0111	7
g^{11}	$x^3 + x^2 + x^1$	1110	14
g^{12}	$x^3 + x^2 + x^1 + 1$	1111	15
g^{13}	$x^3 + x^2 + 1$	1101	13
g^{14}	$x^3 + 1$	1001	9

Esta tabla se realizó mediante los siguientes cálculos para la parte de los polinomios, para la parte binaria usamos los polinomios solo tomando si tienen o no coeficientes en cada posición, y para la parte entera solo usamos los números binarios.

$$m = x^4 + x + 1$$

$$g^4 + g + 1 = 0 \Rightarrow g^4 = -g - 1 = g + 1 ?$$

$$g^5 = g^4 * g = (g+1)g = g^2 + g$$

$$g^6 = g^5 * g = (g^2 + g)g = g^3 + g^2$$

$$g^7 = g^6 * g = (g^3 + g^2)g = g^4 + g^3 = g^3 + g + 1$$

$$g^8 = g^7 * g = (g^3 + g + 1)g = g^4 + g^2 + g = g^2 + g + g + 1 = g^2 + 1$$

$$g^9 = g^8 * g = (g^2 + 1)g = g^3 + g$$

$$g^{10} = g^9 * g = (g^3 + g)g = g^4 + g^2 = g^2 + g + 1$$

$$g^{11} = g^{10} * g = (g^2 + g + 1)g = g^3 + g^2 + g$$

$$g^{12} = g^{11} * g = (g^3 + g^2 + g)g = g^4 + g^3 + g^2 + g = g^3 + g^2 + g + g + 1 = g^3 + g^2 + 1$$

$$g^{13} = g^{12} * g = (g^3 + g^2 + g + 1)g = g^4 + g^3 + g^2 + g = g^3 + g^2 + g + g + 1 = g^3 + g^2 + 1$$

$$g^{14} = g^{13} * g = (g^3 + g^2 + 1)g = g^4 + g^3 + g = g^3 + g + g + 1 = g^3 + 1$$

$$g^{15} = g^{14} * g = (g^3 + 1)g = g^4 + g = g + g + 1 = 1 = g^0$$

$$0 = 0$$

$$g^9 = g^3 + g$$

$$g^{10} = g^2 + g + 1$$

$$g^1 = g^1$$

$$g^{\prime\prime} = g^3 + g^2 + g$$

$$g^2 = g^2$$

$$g^{\prime\prime\prime} = g^3 + g^2 + g + 1$$

$$g^3 = g^3$$

$$g^{\prime\prime\prime\prime} = g^3 + g^2 + 1$$

$$g^4 = g^4 + 1$$

$$g^{\prime\prime\prime\prime\prime} = g^3$$

$$g^5 = g^2 + g$$

$$g^{\prime\prime\prime\prime\prime\prime} = g^0$$

$$g^6 = g^3 + g^2$$

$$g^{\prime\prime\prime\prime\prime\prime\prime} = g^7$$

$$g^7 = g^3 + g + 1$$

$$g^8 = g^2 + 1$$

Fuentes usadas para este ejercicio:

Clases

9. Factoriza los siguientes números mediante el algoritmo que se pide:

a. 256961 mediante Rho de Pollard

Para este ejercicio usamos la siguiente función(se encuentra en Ejercicio9.py)

Algoritmo Rho de Pollard

```
def Rho_Pollard(n):
    #Paso 1 - Declarar a=2 y b=2
    a = 2
    b = 2
    #Paso 2 - Desde 1 hasta que ya no se pueda (xd) hacer lo siguiente
    for i in range(1,100000000):
        #2.1- Calcular a = a^2+1 mod n, b = b^2+1 mod n y otra vez b = b^2+1 mod n
        a = (pow(a,2)+1) % n
        print(f"Calculando a={a}^2+1 mod {n}={a}")
        b = (pow(b,2)+1) % n
        print(f"Calculando b={b}^2+1 mod {n}={b}")
        b = (pow(b,2)+1) % n
        print(f"Calculando b={b}^2+1 mod {n}={b}")
        # 2.2 Calcular d = gcd(a-b,n)
        d = gcd(a-b,n)
        print(f"\n gcd({a}-{b},{n}) = {d}")
        #2.3 Si d > 1 devolver d y terminar con éxito
        if(1<d<n):
            print(f"Se cumple que 1 < {d} < {n} \n Tenemos un factor d = {d}")
            e = n/d
            print(f" El otro factor es e = {n} / {d} = {e}")
            print(f"\n ÉXITO, se contraron los factores de {n}\n Los factores son es d = {d}, e = {e}\n n = d*e = {d * e}")
            break
        #2.4 Si d = n terminar el algoritmo con fracaso
        if(d == n):
            print(f"FRACASO\n No se encontraron factores para {n}")
            break
```

Y la siguiente función auxiliar:

Función para encontrar el máximo común divisor de dos números

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

Nuestro código nos dio el siguiente resultado.

```
33
34  if __name__ == "__main__":
35      Rho_Pollard(256961)
36

PROBLEMS     OUTPUT     DEBUG CONSOLE     TERMINAL     PORTS     COMMEN
```

Calculando b=47467^2+1 mod 256961=47467
Calculando b=82042^2+1 mod 256961=82042

gcd(122769-82042,256961) = 293
Se cumple que 1 < 293 < 256961
Tenemos un factor d = 293
El otro factor es e = 256961 / 293 = 877

ÉXITO, se contraron los factores de 256961
Los factores son es d = 293, e = 877
n = d*e = 256961
camilo@wowi:~/CC/septimoSemestre/Crip/Cripto-2025-1\$

Por lo tanto 256961 se factorizar como $293 * 877 = 256961$.

b. 8746 mediante Criba Cuadrática

Para este ejercicio usamos la siguiente función(se encuentra en Ejercicio9.py), notemos que el algoritmo no es el mismo visto en la clase, si no usamos una combinación rara de lo que encontramos, esto debido a que el visto en la clase está muy complicado y no entendimos cómo realizarlo.

Algoritmo Criba Cuadrática

```
def criba_cuadratica(n):
    # Paso 1, obtener el nivel de suavidad
    b = int(n ** 0.5)
    # Paso 2, obtener los primos menores a b
    p_b = primos_menores(b)
    # Paso 3, obtener F, la base de factores
    f = []
    # Paso 3, iteramos sobre los primos menores a b
    for primo in p_b:
        # Paso 4, calcular z = n^{(p-1)/2} mod p
        z = pow(n, (primo - 1) // 2, primo)
        # Paso 5, si z es 1, agregar p a f
        if z == 1:
            f.append(primo)
    # Paso 6, buscar números suaves
    suaves = []
    i = b
    # Paso 7, mientras no se tengan suficientes números suaves
    while len(suaves) < len(f) + 1:
        # Paso 8, calculamos i^2 mod n
        posible = pow(i, 2, n)
        # Paso 9, revisamos si es b-smooth
        if es_b_smooth(posible, f):
            # Paso 10, si es b-smooth, lo agregamos a suaves
            suaves.append(i)
        # Paso 11, aumentamos i
        i += 1

    # Paso 12, iteramos sobre los números suaves, intentando factorizar n
    for suave in suaves:
        # Paso 13, factorizamos el número suave
        factores = factorizar_n(suave)
        factores_exp = agrupar_factores(factores)
        # Paso 14, revisar el producto de los factores impares, los que deben contribuir al factor
        producto = 1
        for factor, exponente in factores_exp.items():
            if exponente % 2 == 1:
                producto *= factor
        # Paso 15, si el producto de los factores impares no es el número suave ni n
        if producto != suave and producto != n:
            # Paso 16, calcular el factor candidato que es la diferencia entre el producto y el número suave
            candidato = producto - suave
            if candidato < 0:
                candidato = -candidato
            # Paso 17, calcular el mcd del factor candidato y n
            numero = mcd(candidato, n)
            # Paso 18, si el mcd es un divisor no trivial de n, es decir mayor a 1 y menor a n
            if 1 < numero < n:
                # Paso 19, regresar el divisor y el cociente
                return numero, n // numero
    # Paso 20, regresar None
    return None
```

Y las siguientes funciones auxiliares:

Función para encontrar los primos menores a un número

```

# Algoritmo para obtener los primos menores a n
def primos_menores(n):
    # Paso 1, creamos una lista de booleanos de tamaño n
    primos = [True] * n
    # Paso 2, iteramos sobre los numeros menores a n
    for i in range(2, n):
        # Paso 2.1, si el numero es primo
        if primos[i]:
            # Paso 2.1.1, marcamos los multiplos de i como no primos
            for j in range(i * i, n, i):
                primos[j] = False
    # Paso 3, regresamos los numeros primos
    return [i for i in range(2, n) if primos[i]]

```

Función para ver si un número es b-suave

```

# Algoritmo para ver si un numero n es b-smooth usando los primos menores a b
def es_b_smooth(n, primos):
    # Paso 1, iteramos sobre los primos menores a n
    for primo in primos:
        # Paso 2, mientras n sea divisible por el primo
        while n % primo == 0:
            # Paso 3, actualizamos n
            n = n // primo
    # Paso 4, si n es 1 regresamos True
    if n == 1:
        return True
    # Paso 5, regresamos False
    return False

```

Función para factorizar un número

```

# Funcion para factorizar un numero n en una lista de factores primos
# https://stackoverflow.com/questions/32871539/integer-factorization-in-python
def factorizar_n(n):
    factores = []
    factor = 2
    while factor <= n:
        if n % factor == 0:
            factores.append(factor)
            n = n // factor
        else:
            factor += 1
    return factores

```

Función para agrupar los factores de un número

```

# Funcion para agrupar los factores primos y asi tener un exponente
def agrupar_factores(factores):
    factores_agrupados = {}
    for factor in factores:
        if factor in factores_agrupados:
            factores_agrupados[factor] += 1
        else:
            factores_agrupados[factor] = 1
    return factores_agrupados

```

Función para encontrar el máximo común divisor de dos números

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

Nuestro código nos dio el siguiente resultado.

```
149
150 if __name__ == "__main__":
151     #Rho_Pollard(256961)
152     print(criba_cuadratica(8746))
153
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
● camilo@owwi:~/CC/septimoSemestre/Crip/Cripto-2025-1$ /bin/python /home/camilo/CC/septimoSemestre/Crip/Cripto-2025-1/Tarea/Tarea2/Ejercicio9/Ejercicio9.py
(2, 4373)
○ camilo@owwi:~/CC/septimoSemestre/Crip/Cripto-2025-1$ 
```

Por lo tanto 8746 se factorizar como $2 * 4373 = 8746$.

Fuentes usadas para este ejercicio:

Ayudantías

Menezes, A., Paul Van Oorschot, & Vanstone, S. (1996). Handbook of Applied Cryptography. Crc Press. Páginas 91

https://micsymposium.org/mics_2011_proceedings/mics2011_submission_28.pdf

https://en.wikipedia.org/wiki/Quadratic_sieve

<https://www.math.unl.edu/~mbrittenham2/classwk/445f08/dropbox/landquist.quadratic.sieve.pdf>

10. Determina si los siguientes números son primos utilizando el algoritmo que se pide:

- a. 131317 mediante el Test de Fermat

Para este ejercicio usamos la siguiente función (se encuentra en Ejercicio10.py)

Algoritmo Test de Fermat

```

# Test de Fermat 1
def Fermat_Test(p):
    es_primo = True
    print("-----TEST DE FERMAT-----")
    print(f"Determinar si {p} es primo.")
    # Paso 1, iteramos a de 1 a p-1
    for a in range(1,p):
        # Paso 1.1, calculamos resultado = a^p-a mod p
        resultado = (pow(a,p,p)-a)% p
        # Paso 1.2, si resultado es 0, entonces a^p-a es congruente a 0 mod p
        if(resultado == 0):
            print(f"{a}^{p}-{a} es congruente a 0 mod {p}")
        # Paso 1.3, si resultado es distinto de 0, entonces p es compuesto
        else: es_primo = False
        if (es_primo == False):
            print(f"{a}^{p}-{a} mod {p} = {resultado} es distinto a 0 por lo que {p} es un número compuesto")
            break
    # Paso 2, entonces p es primo
    if(es_primo):
        print(f"Todos los valores de 1 a {p-1} cumplen con la propiedad por tanto {p} es primo")

```

(Se trató de implementar el algoritmo 4.9 como viene en el libro pero se encontró que es algo tardado si se selecciona una t grande, pero se encontró que se puede hacer eficiente/rápido si la condición que se cumple de 1 a p en vez de ser $a^{p-1} \equiv 1 \pmod{p}$ tenemos que sea la

condición $a^p - a \equiv 0 \pmod{p}$ y esto se puede debido a lo obtenido al desarrollar la expresión del Pequeño Teorema de Fermat, de esta manera

$$a^{p-1} \equiv 1 \pmod{p} \rightarrow a^{p-1} * a \equiv 1 * a \pmod{p}$$

$$\rightarrow a^p \equiv a \pmod{p} \rightarrow a^p - a \equiv 0 \pmod{p}$$

por lo tanto se hizo de esta forma)

Nuestro código nos dio el siguiente resultado.

```

123
124 | if __name__ == "__main__":
125 |     Fermat_Test(131317)
126 |     #Miller_Test(193394587,479)
127 |     #Solovay_Test(1346459137, 10)
128
129
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
131302^131317-131302 es congruente a 0 mod 131317
131303^131317-131303 es congruente a 0 mod 131317
131304^131317-131304 es congruente a 0 mod 131317
131305^131317-131305 es congruente a 0 mod 131317
131306^131317-131306 es congruente a 0 mod 131317
131307^131317-131307 es congruente a 0 mod 131317
131308^131317-131308 es congruente a 0 mod 131317
131309^131317-131309 es congruente a 0 mod 131317
131310^131317-131310 es congruente a 0 mod 131317
131311^131317-131311 es congruente a 0 mod 131317
131312^131317-131312 es congruente a 0 mod 131317
131313^131317-131313 es congruente a 0 mod 131317
131314^131317-131314 es congruente a 0 mod 131317
131315^131317-131315 es congruente a 0 mod 131317
131316^131317-131316 es congruente a 0 mod 131317
Todos los valores de 1 a 131316 cumplen con la propiedad por tanto 131317 es primo
camilo@wowl:~/CC/septimoSemestre/Crip/Cripto-2025-1$
```

Por lo tanto 131317 si es un primo.

También lo realizamos con el algoritmo 4.9 (el visto en clase).

Algoritmo Test de Fermat 2

```

# Test de Fermat
def Fermat_Test2(n, t):
    # Paso 1, iteramos i de 1 a t
    for i in range(1,t):
        # Paso 1.1, seleccionamos un a aleatorio entre 2 y n-2
        a = random.randrange(2,n-2)
        # Paso 1.2, calculamos r = a^(n-1) mod n
        r = pow(a,n-1,n)
        # Paso 1.3, si r != 1, entonces n es compuesto
        if(r != 1):
            print(f"{n} es compuesto")
            return
    # Paso 2, entonces n es primo
    print(f"{n} es primo")
```

Y nos dio el mismo resultado.

```

139
139 | if __name__ == "__main__":
140 |     #Fermat_Test(131317)
141 |     Fermat_Test2(131317,10000)
142 |     #Miller_Test(193394587,479)
143 |     #Solovay_Test(1346459137, 10)
144
144
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
camilo@wowl:~/CC/septimoSemestre/Crip/Cripto-2025-1$ /bin/python /home/camilo/CC/septimoSemestre/Crip/Cripto-2025-1/Tarea/Tarea2/Ejercicio10/Ejercicio10.py
131317 es primo
camilo@wowl:~/CC/septimoSemestre/Crip/Cripto-2025-1$
```

- b. 193394587 mediante el Test de Solovay-Strassen

Para este ejercicio usamos la siguiente función (se encuentra en Ejercicio10.py)

Algoritmo Test de Solovay-Strassen

```
# Test de Solovay-Strassen
def Solovay_Test(n,t):
    print("-----TEST DE SOLOVAY-TRASSEN -----\\n")
    es_Primo=True
    # Paso 1, iteramos i de 1 a t
    for i in range(1,t):
        # Paso 1.1, seleccionamos un a aleatorio entre 2 y n-2
        a = random.randrange(2,n-2)
        print(f"\"a={a}\"")
        # Paso 1.2, calculamos r = a^{(n-1)/2} mod n
        r = pow(a,(n-1)//2 , n)
        print(f"\"r = {a}^{((n-1)/2)} mod {n} = {r}\"")
        # Paso 1.3, si r != 1 y r != n-1, entonces n es compuesto
        if(r !=1 and r != (n-1)):
            print(f"\"{r} distinto de 1 y {n-1}\"")
            es_Primo = False
            print(f"\"{n} es compuesto \"")
            break
        print("\nCalculando Símbolo de Jacobi")
        # Paso 1.4, calculamos el símbolo de Jacobi s = (a/n)
        s = Jacobi_Symbol(a,n)
        print(f"\"Símbolo de Jacobi s = {s}\"")
        # Paso 1.5, si r no es congruente con s, entonces n es compuesto
        if((r % n) != (s % n)):
            print(f"\"{r%n}---s = {s}---{n} es compuesto 2\"")
            es_Primo = False
            break
    # Paso 2, entonces n es primo
    if(es_Primo):
        print(f"\"{n} es primo\"")
```

Y la siguiente función auxiliar:

Función para obtener el Símbolo de Jacobi

```

def Jacobi_Symbol(a,n):
    s=0
    #Paso 1 - si a=0 regresa 0
    if(a == 0):
        return 0
    #Paso 2 - si a=1 regresa 1
    if(a == 1):
        return 1
    #Paso 3 - describimos a = 2^e * a_1 con a_1 impar
    ai, e = descomponer(a)
    #Paso 4 - Si es es par entonces s = 1
    if(e % 2 == 0):
        s = 1
    #En otro caso s = 1 si n congruente 1 o 7 módulo 8
    # 0 s = -1 si n congruente a 3 o 5 mod 8
    else:
        if(n%8 == 1 or n%8 == 7):
            s = 1
        if(n%8 == 3 or n%8 == 5):
            s = -1
    #Paso 5 -si a_1 congruente a n concuentre a 3 mod 4 entonces s=-s
    if( n%4 == 3 and ai%4 ==3):
        s = -s
    #Paso 6 - Hacemos n_1= n mod a_1
    ni = n % ai
    #Paso 7 - Si a_1 entonces return s, en otro caso return s * jacobi (n1, a1)
    if(ai == 1):
        return(s)
    else:
        return(s * Jacobi_Symbol(ni,ai))

```

Nuestro código nos dio el siguiente resultado.

```

145
146 if __name__ == "__main__":
147     #Fermat_Test(131317)
148     #Fermat_Test2(131317,10000)
149     Solovay_Test(193394587, 10)
150     #Miller_Test(1346459137,479)
151

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

36/2^3=4.5
La división ya no es entera, así que los valores son
r = 9
s = 2

Encontrar n-1=2^sr
2/2^1=1.0
2/2^2=0.5
La división ya no es entera, así que los valores son
r = 1
s = 1

Simbolo de Jacobi s = 1
193394587 es primo
camilo@wofi:~/CC/septimoSemestre/Crip/Cripto-2025-1$ 

```

Por lo tanto 193394587 si es un primo.

c. 1346459137 mediante el Test de Miller-Rabin

Para este ejercicio usamos la siguiente función (se encuentra en Ejercicio10.py)

Algoritmo Test de Miller-Rabin

```
# Test de Miller-Rabin
def Miller_Test(n, t):
    print("-----TEST DE MILLER-RABIN-----\n")
    # Paso 1, escribimos a n-1 = 2^s * r con r impar
    r, s = descomponer(n-1)
    es_Primo = True
    # Paso 2, iteramos i de 1 a t
    for i in range (1,t):
        # Paso 2.1, seleccionamos un a aleatorio entre 2 y n-2
        a = random.randrange(2,n-2)
        # Paso 2.2, calculamos y = a^r mod n
        y = pow(a,r,n)
        print(f" a = {a}")
        print(f"y = {a}^{r} mod {n} = {y}")
        # Paso 2.3, si y != 1 y y != n-1, entonces hacemos lo siguiente
        if((y !=1)and(y!=(n-1))):
            print(f"\{y} distinto de 1 y \{n}-1")
            # Paso 2.3.1, hacemos j = 1
            j = 1
            # Paso 2.3.2, mientras j <= s-1 y y != n-1, hacemos lo siguiente
            while((j <= (s-1))and(y != n-1)):
                # Paso 2.3.2.1, calculamos y = y^2 mod n
                y = pow(y,2,n)
                print(f"y = {y}")
                # Paso 2.3.2.2, si y = 1, entonces n es compuesto
                if(y==1):
                    print(f"\{n} es compuesto")
                    es_Primo=False
                    break
                # Paso 2.3.2.3, incrementamos j en 1
                j=j+1
            # Paso 2.3.3, si y != n-1, entonces n es compuesto
            if(y!=n-1):
                print(f"\{n} es compuesto")
                es_Primo = False
                break
        # Paso 3, entonces n es primo
        if(es_Primo):
            print(f"\n{n} es primo")
```

Y la siguiente función auxiliar:

Función para descomponer un entero en $2^s * r$ con r impar

```

def descomponer(n):
    print("Encontrar n-1=2^sr")
    bandera = True
    i = 1
    r=0
    while(bandera):
        div = n / pow(2,i)
        print(f"{n}/2^{i}={div}")
        if((div % 1) == 0 ):
            i = i+1
        else:
            print("La división ya no es entera, así que los valores son")
            r = n // pow(2,i-1)
            bandera = False

    s = i-1
    print(f" r = {r} \n s = {s}\n")
    return r, s

```

Nuestro código nos dio el siguiente resultado.

```

166  if __name__=="__main__":
167      #Fermat_Test(131317)
168      #Fermat_Test2(131317,100)
169      #Solovay_Test(193394587, 10)
170      Miller_Test(1346459137,479)
171

```

PROBLEMS	OUTPUT	DEBUG CONSOLE	<u>TERMINAL</u>	PORTS	COMMENTS
----------	--------	---------------	-----------------	-------	----------

```

y = 342516149^2629803 mod 1346459137 = 1113609390
1113609390 distinto de 1 y 1346459137-1
y = 889791655
y = 747027815
y = 42717049
y = 616541535
y = 720324740
y = 28576378
y = 763436302
y = 1346459136

1346459137 es primo
> camilo@wofi:~/CC/septimoSemestre/Crip/Cripto-2025-1$ 

```

Por lo tanto 1346459137 si es un primo.

Fuentes usadas para este ejercicio:

Ayudantías

Menezes, A., Paul Van Oorschot, & Vanstone, S. (1996). Handbook of Applied Cryptography. Crc Press. Páginas 136, 138 y 139

Burn, B. (2002). Fermat's Little Theorem: Proofs That Fermat Might Have Used. The Mathematical Gazette, 86(507), 415–422. <https://doi.org/10.2307/3621133>