Universidad Del Valle de Guatemala — Departamento de Computación

Lab 2. Esquemas de detección y corrección de errores Parte 1 — Sección 20

Angel Sebastian Castellanos Pineda — 21700

Javier Alejandro Azurdia Arrecis — 21242

Diego Alejandro Morales Escobar — 21146

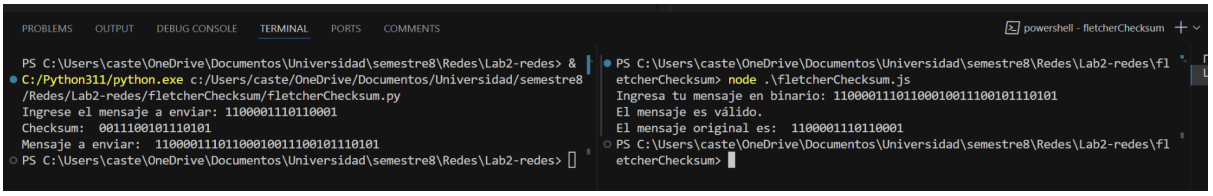
Mensajes utilizados

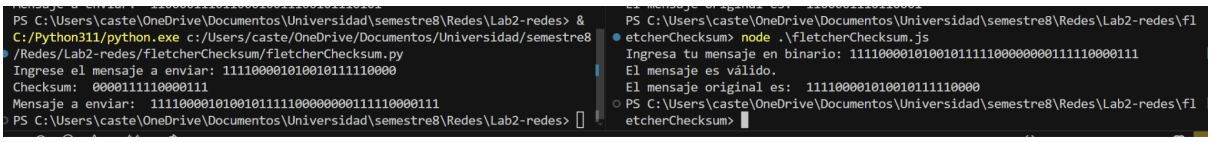
- 1100001110110001
- 11110000101001011111000000
- 1111000010100101111100000000111110000111

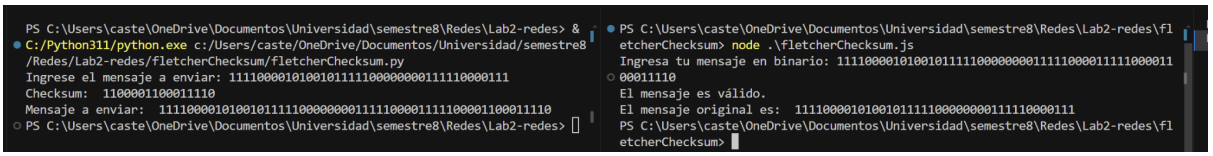
Algoritmo Fletcher checksum 16

Escenarios de prueba

Envió de los 3 mensajes sin error

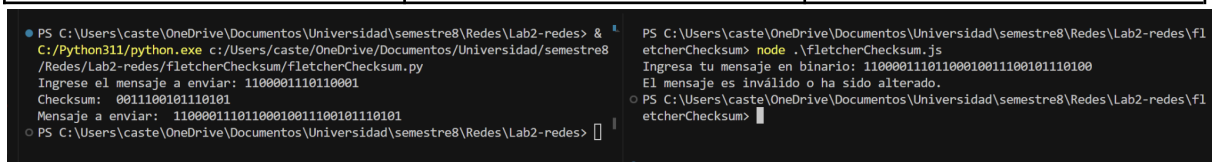






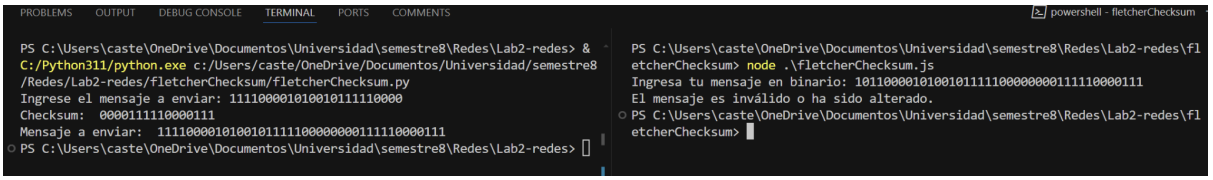
Envió de los 3 mensajes con errores

Mensaje original	Resultado	Mensaje modificado
• 1100001110110001	• 1100001110110001001110101110101	• 1100001110110001001110101110100

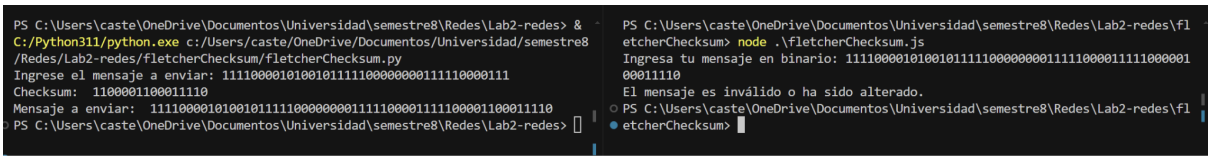


Mensaje original	Resultado	Mensaje modificado

• 11110000101001011110000	• 11110000101001011110000000000111110000111	• 10110000101001011110000000000111110000111

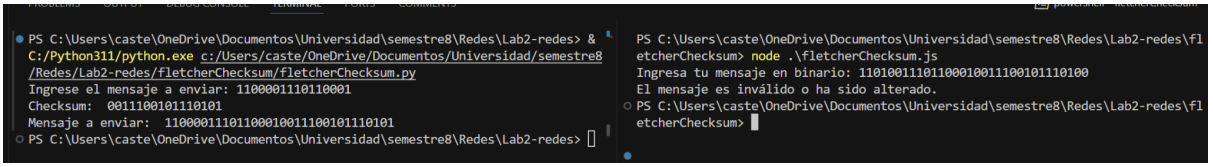


Mensaje original	Resultado	Mensaje modificado
• 11110000101001011110000000000111110000111	• 11110000101001011110000000000111110000111100001111000011000011110	• 11110000101001011110000000000111110000111100001111000001000011110

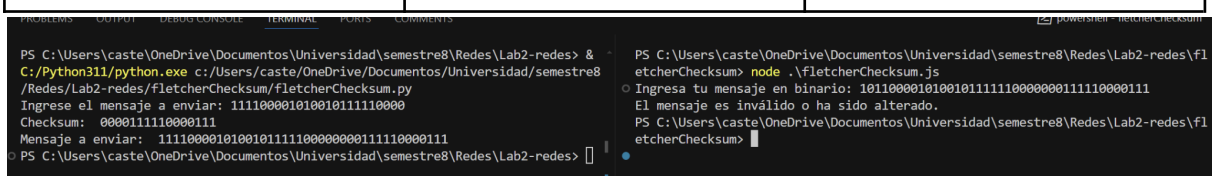


Envió de los 3 mensajes con dos errores

Mensaje original	Resultado	Mensaje modificado
• 1100001110110001	• 11000011101100010011100101110101	• 11010011101100010011100101110100

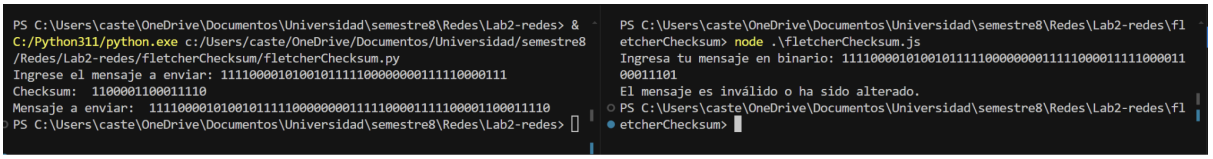


Mensaje original	Resultado	Mensaje modificado
• 11110000101001011110000	• 11110000101001011110000000000111110000111	• 10110000101001011111100000000111110000111



Mensaje original	Resultado	Mensaje modificado

• 11110000101001011110000000000111110000111	• 11110000101001011110000000000111110000011110	• 11110000101001011110000000000111110000011101

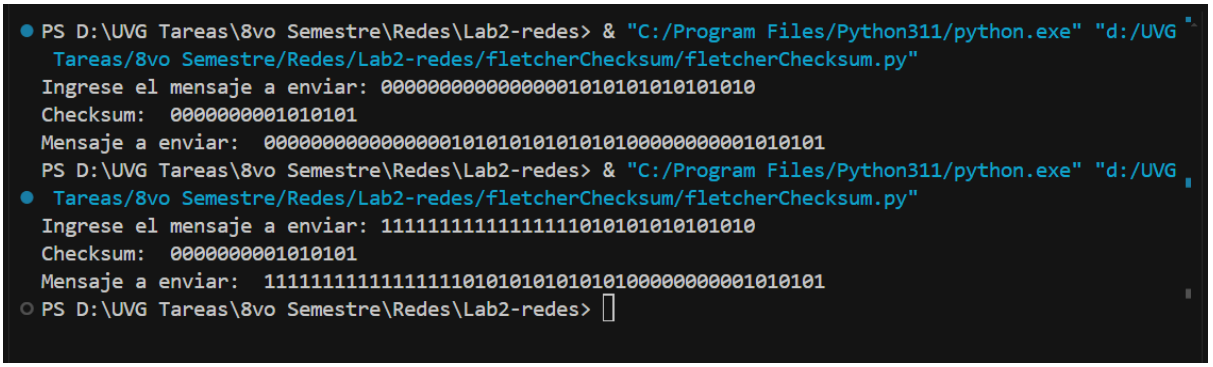


¿Es posible manipular los bits de tal forma que el algoritmo seleccionado no sea capaz de detectar el error? ¿Por qué sí o por qué no?

Si es posible, ya que si los errores en los datos son tales que las alteraciones en valores de los bytes resultan en el mismo valor para la primera suma y la segunda al calcular el checksum, entonces el algoritmo no detectará la alteración. Esto es posible si se alteran varios bits de tal manera que los cambios se cancelen entre sí en la suma total. Es decir, la comprobación no puede distinguir entre bloques con todos los bits a 0 y bloques con todos los bits a 1. (*Funciones Resumen I*, s/f)

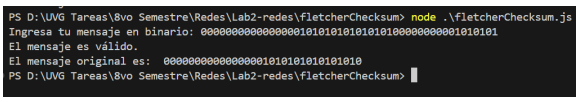
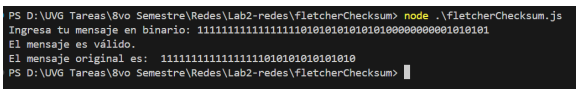
Por ejemplo con los siguientes mensajes se obtiene el mismo resultado para el checksum de comprobación y el mensaje puede ser considerado como correcto cuando en realidad es distinto y ha sido alterado.

Mensaje	Checksum
00000000000000001010101010101010	0000000001010101
11111111111111111010101010101010	0000000001010101

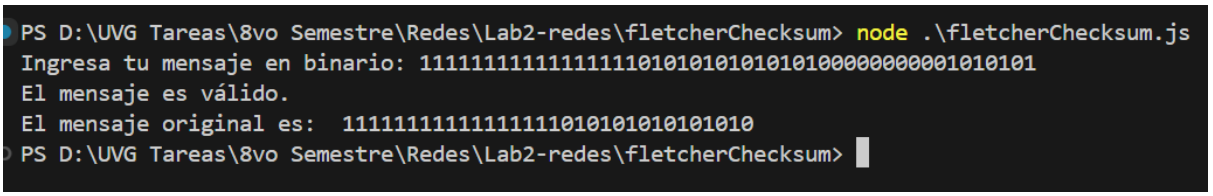


Al cambiar la primera parte del mensaje de 0 a 1, y dejando sin modificar el resto del mensaje, se logra obtener el mismo resultado, con mensajes distintos.

Mensaje	Resultado

00000000000000001010101010101010000000001010101	
11111111111111111010101010101010000000001010101	

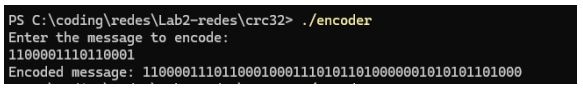
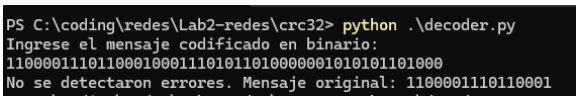
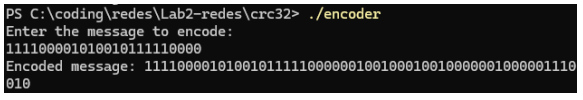
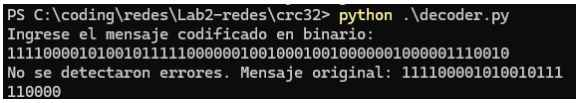
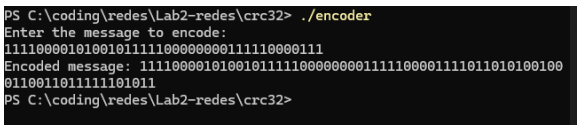
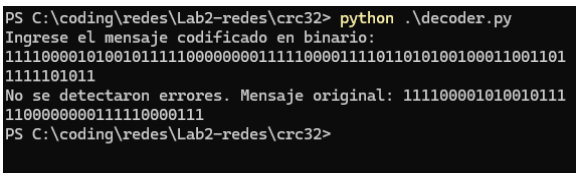
Ahora suponiendo que se quiere enviar el mensaje 00000000000000001010101010101010000000001010101, sin embargo por ruidos o atacantes el mensaje que llega es el siguiente 11111111111111111010101010101010000000001010101, el algoritmo lo detectara como válido y lo dejará pasar.



Algoritmo CRC-32

Escenarios de prueba

Envió de los 3 mensajes sin error

Envió de los 3 mensajes con errores

Mensaje original	Resultado	Mensaje modificado

• 1100001110110001	• 11000011101100010 00111010110100000 01010101101000	• 11000011101100010 00111010100100000 01010101101000

```
PS C:\coding\redes\Lab2-redes\crc32> python .\decoder.py
Ingrese el mensaje codificado en binario:
11000011101100010001110101101000000101010110100 0
No se detectaron errores. Mensaje original: 1100001110110001
PS C:\coding\redes\Lab2-redes\crc32>
```

```
PS C:\coding\redes\Lab2-redes\crc32> python .\decoder.py
Ingrese el mensaje codificado en binario:
11000011101100010001110101001000000101010110100 0
Se detectaron errores.
PS C:\coding\redes\Lab2-redes\crc32>
```

Mensaje original	Resultado	Mensaje modificado
• 11110000101001011 1110000	• 11110000101001011 11100000010010001 00100000010000011 10010	• 11110000101001011 11100010010010001 00100000010000011 10010

```
PS C:\coding\redes\Lab2-redes\crc32> python .\decoder.py
Ingrese el mensaje codificado en binario:
1111000010100101111100000010010001001000000100000111001 0
No se detectaron errores. Mensaje original:
11110000101001011110000
PS C:\coding\redes\Lab2-redes\crc32>
```

```
PS C:\coding\redes\Lab2-redes\crc32> python .\decoder.py
Ingrese el mensaje codificado en binario:
1111000010100101111100000010010001001000000100000111001 0
Se detectaron errores.
PS C:\coding\redes\Lab2-redes\crc32>
```

Mensaje original	Resultado	Mensaje modificado
• 11110000101001011 11100000000111110 000111	• 11110000101001011 11100000000111110 00011110110101001 00011001101111110 1011	• 11110000101001011 11100000000111110 00011100110101001 00011001101111110 1011

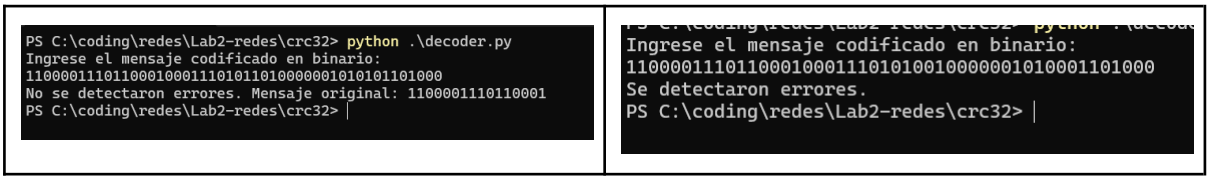
```
PS C:\coding\redes\Lab2-redes\crc32> python .\decoder.py
Ingrese el mensaje codificado en binario:
1111000010100101111100000000111110000111010101001000110011011111101011 0
No se detectaron errores. Mensaje original:
111100001010010111110000000011111000011
PS C:\coding\redes\Lab2-redes\crc32>
```

```
PS C:\coding\redes\Lab2-redes\crc32> python .\decoder.py
Ingrese el mensaje codificado en binario:
1111000010100101111100000000111110000111001101010010001100110111111101011 0
Se detectaron errores.
PS C:\coding\redes\Lab2-redes\crc32>
```

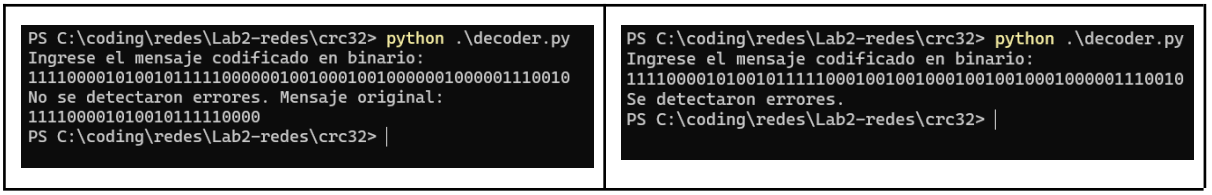
Envió de los 3 mensajes con dos errores

Mensaje original	Resultado	Mensaje modificado

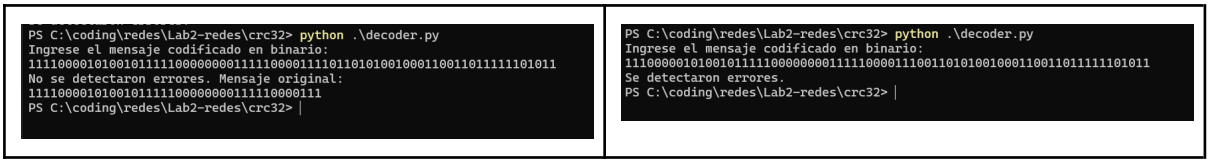
• 1100001110110001	• 11000011101100010 00111010110100000 01010101101000	• 11000011101100010 00111010100100000 01010001101000



Mensaje original	Resultado	Mensaje modificado
• 11110000101001011 1110000	• 11110000101001011 11100000010010001 00100000010000011 10010	• 11110000101001011 11100010010010001 00100100010000011 10010



Mensaje original	Resultado	Mensaje modificado
• 11110000101001011 11100000000111110 000111	• 11110000101001011 11100000000111110 00011110110101001 00011001101111110 1011	• 11100000101001011 11100000000111110 00011100110101001 00011001101111110 1011



¿Es posible manipular los bits de tal forma que el algoritmo seleccionado no sea capaz de detectar el error? ¿Por qué sí o por qué no? En caso afirmativo, demuestrelo con su implementación.

Sí, es posible manipular los bits de tal forma que el algoritmo CRC32 no sea capaz de detectar el error. Esto se debe a la posibilidad de colisiones en el algoritmo CRC32. Una colisión ocurre cuando dos entradas diferentes generan el mismo valor de CRC32.

Para demostrar lo anterior, se realizó un script adicional en python, buscando dos valores que generarán el mismo valor CRC32. Se decidió a realizar una prueba de fuerza bruta ya que no hay una forma estándar de encontrar dos valores con el mismo valor CRC32.

El script es el siguiente:

crc32 > search_collisions.py > ...

```python
import random
import string

def crc32_manual(data):
    crc = 0xFFFFFFFF
    polynomial = 0xEDB88320

    for i in range(0, len(data), 8):
        byte = int(data[i:i+8], 2)
        crc ^= byte
        for _ in range(8):
            if crc & 1:
                crc = (crc >> 1) ^ polynomial
            else:
                crc >>= 1

    return crc ^ 0xFFFFFFFF

def string_to_binary(s):
    return ''.join(format(ord(c), '08b') for c in s)

def generate_random_string(length):
    return ''.join(random.choices(string.ascii_letters + string.digits, k=length))

def main():
    length = 5  # Longitud de las cadenas aleatorias
    num_trials = 1000000000000  # Número de pruebas a realizar

    hashes = {}

    for _ in range(num_trials):
        random_str = generate_random_string(length)
        binary_str = string_to_binary(random_str)
        crc = crc32_manual(binary_str)

        if crc in hashes and hashes[crc] != random_str:
            print(f"Colisión encontrada: '{random_str}' y '{hashes[crc]}' tienen el mismo CRC32: {crc:08X}")
            break
        else:
            hashes[crc] = random_str
    else:
        print(f"No se encontraron colisiones en {num_trials} pruebas")

if __name__ == "__main__":
    main()
```

Y se obtuvo como resultado que las cadenas '3DA58' y 'Cx00h' generan el mismo valor CRC32.

Es posible apreciar el valor en la implementación original:

```
PS C:\coding\redes\Lab2-redes\crc32> ./encoder
Ingrese el mensaje: 3DA58
Encoded message: 0011001101000100010000010011010100111000110011111011111001100111111111110
PS C:\coding\redes\Lab2-redes\crc32> ./encoder
Ingrese el mensaje: Cx00h
Encoded message: 0100001101111000001100000011000001101000110011111011111001100111111111110
PS C:\coding\redes\Lab2-redes\crc32> ./encoder
```

```
PS C:\coding\redes\Lab2-redes\crc32> python .\decoder.py
Ingrese el mensaje codificado en binario:
0011001101000100010000010011010100111000110011111011111001100111111111110
Mensaje: 00110011010001000100000100110101001110000
CRC32: 11001111101111100110011111111110
No se detectaron errores. Mensaje original:
3DA58
PS C:\coding\redes\Lab2-redes\crc32> python .\decoder.py
Ingrese el mensaje codificado en binario:
0100001101111000001100000011000001101000110011111011111001100111111111110
Mensaje: 01000011011110000011000000110000001101000
CRC32: 11001111101111100110011111111110
No se detectaron errores. Mensaje original:
Cx00h
PS C:\coding\redes\Lab2-redes\crc32>
```

La parte resaltada en verde es el valor crc de ambas cadenas, que son iguales.

Algoritmo Hamming

Escenarios de prueba

Envió de los 3 mensajes sin error

Mensaje 1:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  COMMENTS

PS C:\Users\DIEAL> & C:/Users/DIEAL/AppData/Local/Programs/Python/Python311/python.exe c:/Users/DIEAL/OneDrive/Documents/Programas/Redes/Lab2-redes/Hamming/Hamming.py
Ingrese el mensaje a codificar: 1100001110110001
Mensaje codificado:  11000001101110001110
PS C:\Users\DIEAL>
```

```
Ingrese el mensaje a decodificar:
110000011101110001110
No hay errores
Mensaje decodificado: 1100001110110001
```

Mensaje 2:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  COMMENTS

PS C:\Users\DIEAL> & C:/Users/DIEAL/AppData/Local/Programs/Python/Python311/python.exe c:/Users/DIEAL/OneDrive/Documents/Programas/Redes/Lab2-redes/Hamming/Hamming.py
Ingrese el mensaje a codificar: 111100001010010111110000
Mensaje codificado:  111100001010001011111000000000
PS C:\Users\DIEAL>
```

```
Ingrese el mensaje a decodificar:
111100001010001011111000000000
No hay errores
Mensaje decodificado: 111100001010010111110000

C:\Users\DIEAL\OneDrive\Documents\Programas\Redes\Lab2-redes\Hamming\Hamminng\Hamminng\bin\
oceso 17592) se cerró con el código 0
```

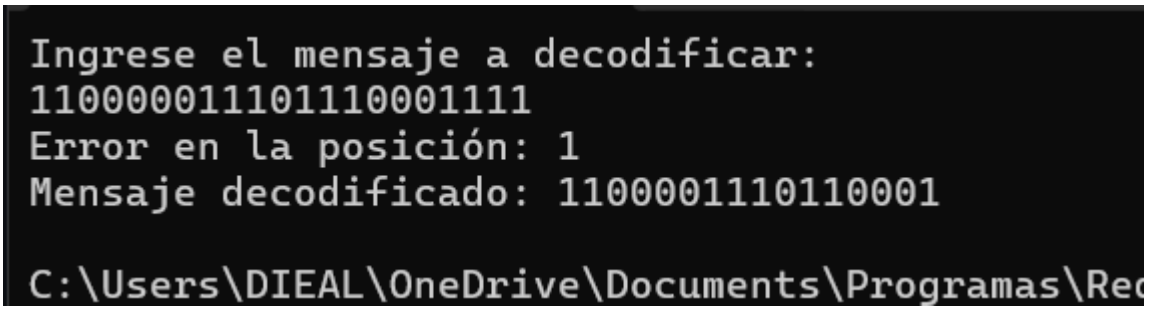
Mensaje 3;

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  COMMENTS

PS C:\Users\DIEAL> & C:/Users/DIEAL/AppData/Local/Programs/Python/Python311/python.exe c:/Users/DIEAL/OneDrive/Documents/Programas/Redes/Lab2-redes/Hamming/H
Ingrese el mensaje a codificar: 11110000101001011111000000001111110000111
Mensaje codificado:  11110000101001101111100000000101111000101101000
PS C:\Users\DIEAL>
```

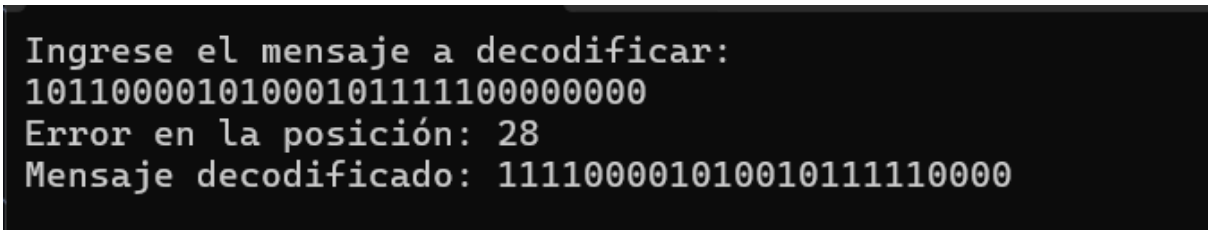
```
Consola de depuración de Mi

Ingrese el mensaje a decodificar:
11110000101001101111100000000101111000101101000
No hay errores
Mensaje decodificado: 111100001010010111110000000111111000011 1

C:\Users\DIEAL\OneDrive\Documents\Programas\Redes\Lab2-redes\Hamming\Hamm
oceso 27348) se cerró con el código 0
```

Envió de los 3 mensajes con errores

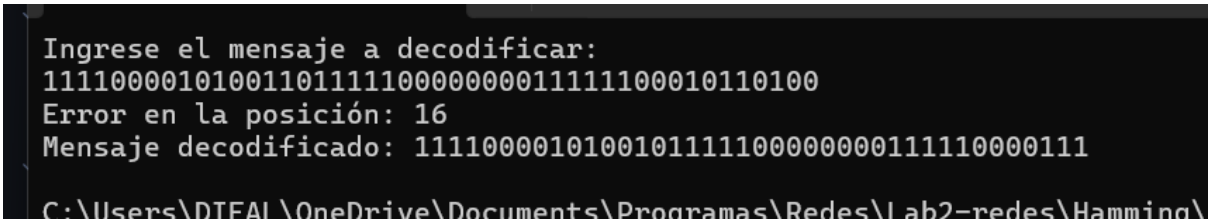
Mensaje original	Resultado	Mensaje modificado
• 1100001110110001	• 110000011101110001110	• 110000011101110001111



Mensaje original	Resultado	Mensaje modificado
• 11110000101001011110000	• 11110000101000101111100000000	• 10110000101000101111100000000

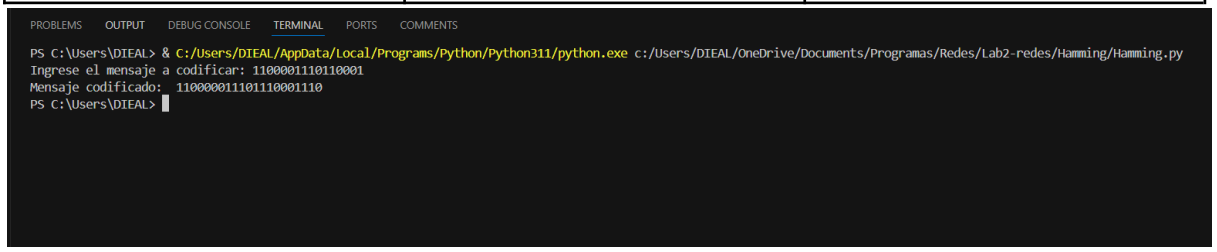


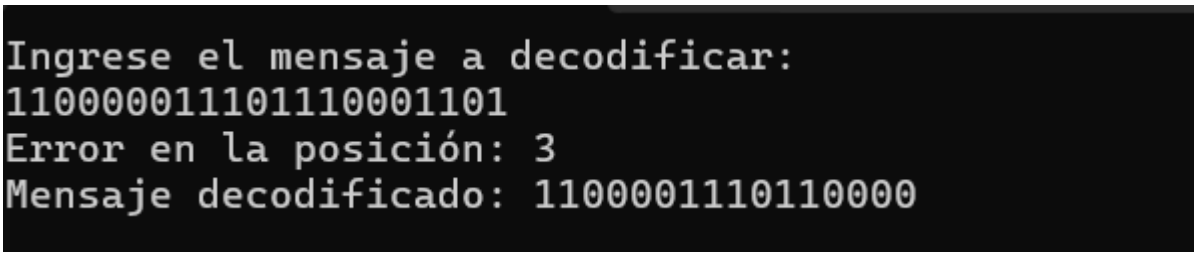
Mensaje original	Resultado	Mensaje modificado
• 11110000101001011111000000000111110000111	• 11110000101001101111000000001011100010110100	• 11110000101001101111000000001111100010110100



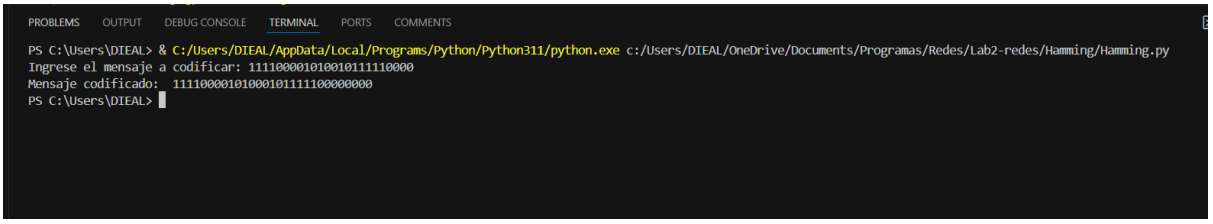
Envió de los 3 mensajes con dos errores

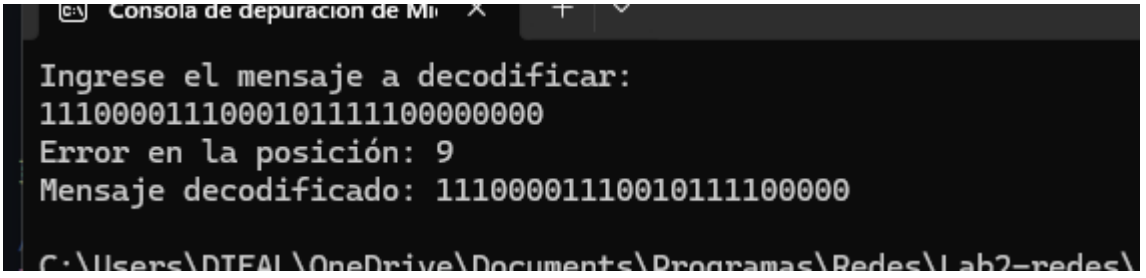
Mensaje original	Resultado	Mensaje modificado
• 1100001110110001	• 1100000111011100011110	• 1100000111011100011101



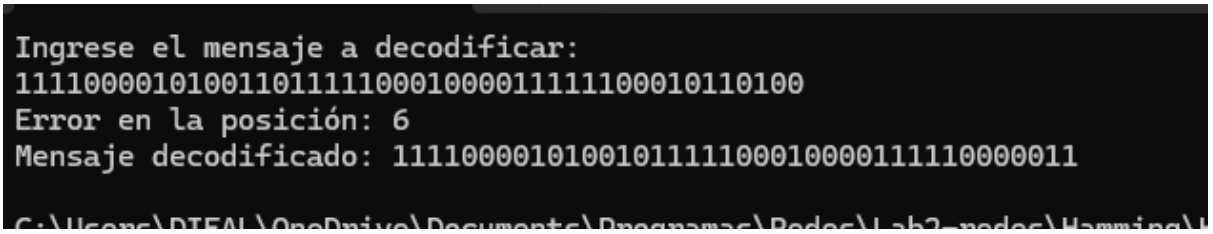


Mensaje original	Resultado	Mensaje modificado
• 111100001010010111110000	• 111100001010001011111000000000	• 111100001110001011111000000000





Mensaje original	Resultado	Mensaje modificado
• 11110000101001011110000000000111110000000111	• 11110000101001101111000000001011100010110100	• 11110000101001101111000100001111100010110100



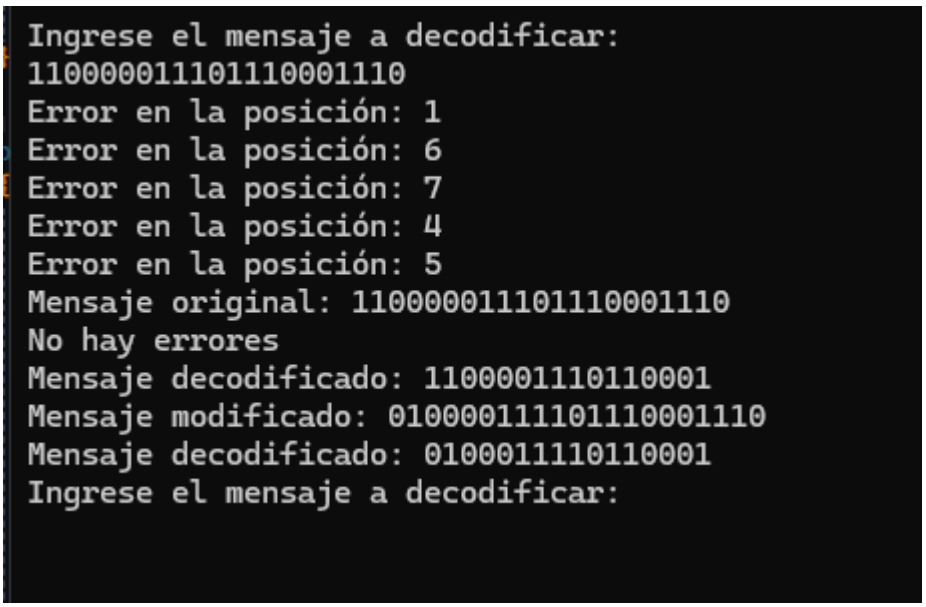
¿Es posible manipular los bits de tal forma que el algoritmo seleccionado no sea capaz de detectar el error? ¿Por qué sí o por qué no?

Sí, es posible manipular los bits de tal manera que el algoritmo de Hamming no sea capaz de detectar el error, pero esto ocurre bajo ciertas condiciones:

- Errores de dos bits: Aunque el algoritmo de Hamming puede detectar errores de dos bits, no puede corregirlos. Además, en algunos casos específicos, dos errores de bit pueden anularse mutuamente, lo que hace que el algoritmo de Hamming no detecte ningún error.
- Errores de más de dos bits: El algoritmo de Hamming no está diseñado para manejar errores de más de dos bits. En estos casos, puede no detectar el error o incluso corregir incorrectamente el dato.
- Manipulación intencionada: Si alguien manipula los datos y los bits de paridad de una manera muy específica y cuidadosa, es posible que el error pase desapercibido.

(Codifica.me, 2011)

Para demostrar esto, se realizó un script en el que se modifica la cadena original hasta encontrar una combinación en la que no se detecten errores en el mensaje. se realizó la prueba con la cadena **110000011101110001110**, en donde se tuvo este resultado:



Se encontró que con la cadena modificada **01000011101110001110** no se detectaron errores con el algoritmo de Hamming.

En base a las pruebas que realizó, ¿qué ventajas y desventajas posee cada algoritmo con respecto a los otros dos?

- La gran ventaja de Hamming Code frente a los otros dos algoritmos usados, es que puede detectar y corregir errores de un solo bit. El resto solo puede detectar, más no corregir. Sin embargo, es significativamente más complicado de implementar y requiere de una mayor capacidad de procesamiento y bits de paridad, lo que hace que aumente el tamaño del mensaje.
- Checksum16 tiene como ventaja ser un algoritmo simple y rápido de implementar, lo que hace que sea ideal para sistemas con recursos limitados. Sin embargo, de los tres implementados, es el más susceptible a colisiones, además de ser el que menos capacidad de detección de errores tiene.
- El algoritmo CRC32 es robusto en la detección de errores. Por eso suele ser usado en aplicaciones críticas como Ethernet. Puede detectar muchos errores sin fallar. Sin embargo, su implementación es más compleja.

En resumen, Hamming es ideal para la corrección de errores, Checksum16 para la simplicidad y eficiencia, y CRC32 para la detección robusta en aplicaciones críticas.

Referencias

Codifica.me. (2011, August 11). *Código Hamming | Detectar errores por paridad*. Codifica.me | Desarrollo web. Retrieved July 25, 2024, from https://www.codifica.me/codigo-hamming-detectar-errores-por-paridad/

fuchsia. (2023). *CRC-32*. CRC-32. Retrieved July 25, 2024, from https://fuchsia.googlesource.com/third_party/wuffs/+/HEAD/std/crc32/README.md

ilkkachu. (2010, April 6). *c - How is a CRC32 checksum calculated?* Stack Overflow. Retrieved July 25, 2024, from https://stackoverflow.com/questions/2587766/how-is-a-crc32-checksum-calculated