Laboratorio 2

Autores:

Kristopher Javier Alvarado López	| Carné 21188
Mario Antonio Guerra Morales	| Carné 21008
David Jonathan Aragón Vásquez	| Carné 21053

Curso:
Redes
Catedrático:
Miguel Novella Linares
Sección:
10

Universidad del Valle de Guatemala
11 calle 15-79 Zona 15 Vista Hermosa lll
Guatemala, C. A.
Facultad de Ingeniería

Laboratorio 2 - Esquemas de Detección y Corrección

Repositorio de GitHub:

https://github.com/AragonD19/Lab2Redes

Corrección de Errores:

Códigos de Hamming:

Escenarios de pruebas:

- Sin errores:

```
"C:\Users\acer\OneDrive - Univers
Enter a binary message:1101
 Hamming code: 1010101
```

```
Enter the received Hamming code: 1010101
No errors detected.
Original message: 1101
```

```
Enter a binary message:000
 Hamming code: 000000
```

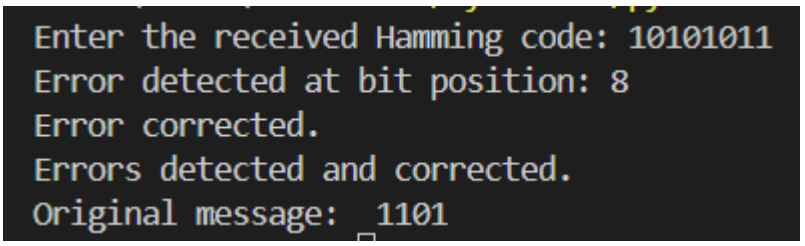
```
Enter the received Hamming code: 000000
No errors detected.
Original message: 000
```

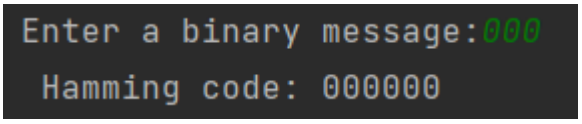
```
Enter a binary message:00000
 Hamming code: 000000000
```

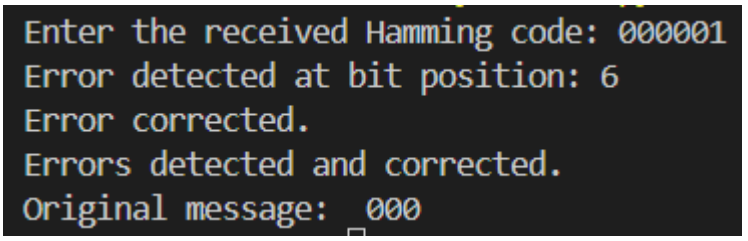
```
Enter the received Hamming code: 000000000
No errors detected.
Original message: 00000
```

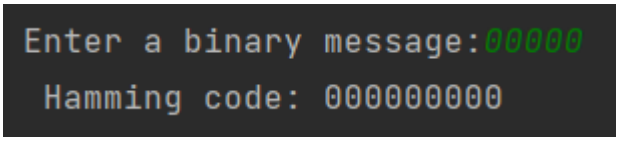
- Un error:

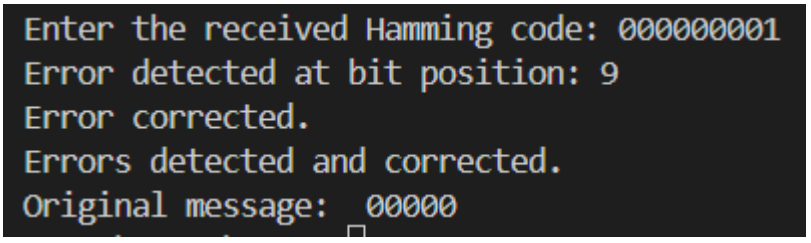
```
"C:\Users\acer\OneDrive - Univers
Enter a binary message:1101
 Hamming code: 1010101
```



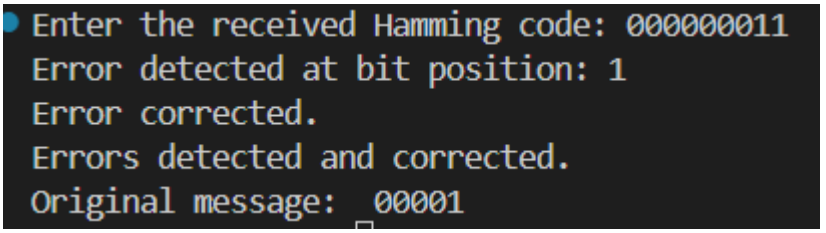








- Dos o más errores:



El código de Hamming no puede corregir dos o más errores simultáneos porque está diseñado para detectar y corregir únicamente un único bit erróneo en un bloque de datos. Cuando se produce un único error, los bits de paridad calculados permiten identificar la posición exacta del error y corregirlo. Sin embargo, si hay dos o más errores, las posiciones calculadas de los bits de paridad pueden llevar a un diagnóstico incorrecto, resultando en la identificación errónea de la posición del error o en la incapacidad de detectar que existen múltiples errores.

- ¿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 de Hamming no detecte el error. Esto ocurre cuando los errores introducidos afectan a los bits de manera que

la combinación de los bits de paridad no cambie lo suficiente como para ser detectada como anómala. Por ejemplo, si dos bits se corrompen y la combinación resultante aún parece válida según los bits de paridad calculados, el algoritmo de Hamming podría no detectar el error.

- En base a las pruebas que realizó, ¿qué ventajas y desventajas posee cada algoritmo con respecto a los otros dos? Tome en cuenta complejidad, velocidad, redundancia (overhead), etc.

  Los códigos de Hamming son simples y tienen bajo overhead, ideales para corrección de errores de un solo bit, pero no manejan múltiples errores. Los códigos convolucionales, utilizando el algoritmo de Viterbi, ofrecen una robusta corrección de errores en condiciones ruidosas a costa de mayor complejidad y overhead. El CRC-32 es eficiente para detectar errores en bloques grandes de datos y es rápido, pero solo detecta errores sin corregirlos y tiene un overhead mayor comparado con los códigos de Hamming. En resumen, Hamming es adecuado para corrección básica con bajo overhead, códigos convolucionales para corrección avanzada en entornos ruidosos, y CRC-32 para detección eficaz de errores en grandes bloques de datos.

Códigos Convolucionales (Algoritmo de Viterbi):

Escenarios de pruebas:

- Sin errores:

```
C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>Viterbi_emisor.exe
Introduce el mensaje en binario: 1101
Mensaje codificado: 11011011

C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>python3 Viterbi_receptor.py
Introduce el mensaje codificado en binario: 11011011
No se detectaron errores. Mensaje original: 1101
```

```
C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>Viterbi_emisor.exe
Introduce el mensaje en binario: 000
Mensaje codificado: 000000

C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>python3 Viterbi_receptor.py
Introduce el mensaje codificado en binario: 000000
No se detectaron errores. Mensaje original: 000
```

```
C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>Viterbi_emisor.exe
Introduce el mensaje en binario: 00000
Mensaje codificado: 0000000000

C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>python3 Viterbi_receptor.py
Introduce el mensaje codificado en binario: 0000000000
No se detectaron errores. Mensaje original: 00000
```

- Un error:

```
C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>Viterbi_emisor.exe
Introduce el mensaje en binario: 1101
Mensaje codificado: 11011011

C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>python3 Viterbi_receptor.py
Introduce el mensaje codificado en binario: 11111011
Se detectaron errores. Mensaje corregido: 1101
```

```
C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>Viterbi_emisor.exe
Introduce el mensaje en binario: 000
Mensaje codificado: 000000

C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>python3 Viterbi_receptor.py
Introduce el mensaje codificado en binario: 010000
Se detectaron errores. Mensaje corregido: 000
```

```
C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>Viterbi_emisor.exe
Introduce el mensaje en binario: 00000
Mensaje codificado: 0000000000

C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>python3 Viterbi_receptor.py
Introduce el mensaje codificado en binario: 1000000000
Se detectaron errores. Mensaje corregido: 00000
```

- Dos o más errores:

```
C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>Viterbi_emisor.exe
Introduce el mensaje en binario: 1101
Mensaje codificado: 11011011

C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>python3 Viterbi_receptor.py
Introduce el mensaje codificado en binario: 11011110
Se detectaron errores. Mensaje corregido: 1110
```

```
C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>Viterbi_emisor.exe
Introduce el mensaje en binario: 000
Mensaje codificado: 000000

C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>python3 Viterbi_receptor.py
Introduce el mensaje codificado en binario: 000110
Se detectaron errores. Mensaje corregido: 010
```

```
C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>Viterbi_emisor.exe
Introduce el mensaje en binario: 00000
Mensaje codificado: 0000000000

C:\Users\mague\OneDrive\Documentos\Lab2Redes\Mario>python3 Viterbi_receptor.py
Introduce el mensaje codificado en binario: 0011111100
Se detectaron errores. Mensaje corregido: 01100
```

No es posible en el algoritmo de Viterbi lograr resolver dos o más errores actualmente, esto puede deberse al tener una tasa de código 2:1.

- ¿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.

No es posible con este algoritmo hacerlo. Ya que, depende de ciertos patrones que se detecten conforme vaya leyendo los bits de entrada por el usuario. Lo cual impide que haya patrones iguales para un mismo número binario.

- En base a las pruebas que realizó, ¿qué ventajas y desventajas posee cada algoritmo con respecto a los otros dos? Tome en cuenta complejidad, velocidad, redundancia (overhead), etc.

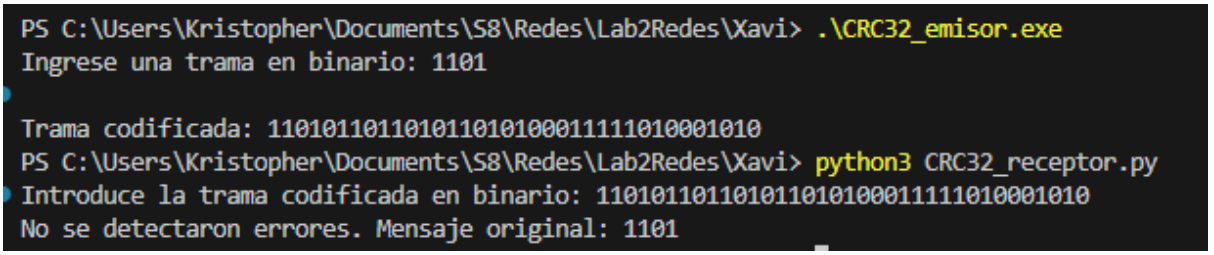
  Veo como ventajas con el algoritmo de Viterbi que utiliza una mayor cantidad de bits de acuerdo a cómo se defina la tasa de código para poder codificar de una manera más segura los mensajes que se le envíen a alguien más. Además de ser un algoritmo no muy complejo al momento de utilizarlo, siendo principalmente, algoritmos de búsqueda para encontrar un mejor camino para decodificar un mensaje. Mientras que, una desventaja como tal es depender de esta misma tasa de código para poder resolver cierta cantidad de errores, ya que en implementaciones con tasas pequeñas no será totalmente comprobable que pueda verificar varios errores y poder corregirlos.

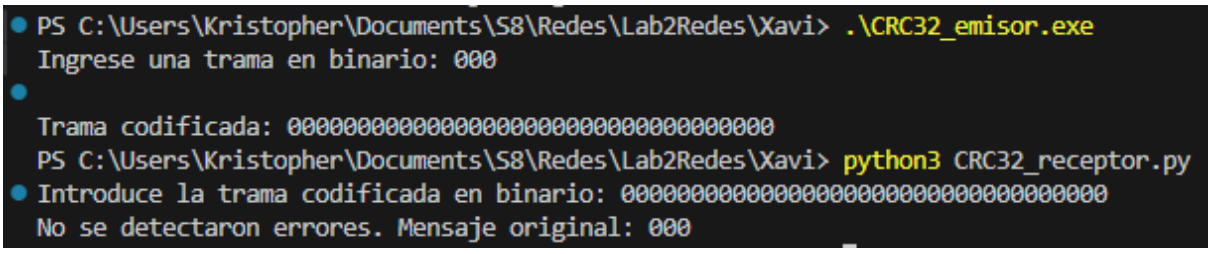
Detección de Errores:

CRC-32:

Escenarios de pruebas:

- Sin errores:





```
PS C:\Users\Kristopher\Documents\S8\Redes\Lab2Redes\Xavi> .\CRC32_emisor.exe
Ingrese una trama en binario: 00000

Trama codificada: 00000000000000000000000000000000
PS C:\Users\Kristopher\Documents\S8\Redes\Lab2Redes\Xavi> python3 CRC32_receptor.py
Introduce la trama codificada en binario: 00000000000000000000000000000000
No se detectaron errores. Mensaje original: 00000
```

- Un error:

  Trama original: 1101
  Trama codificada: 11010110110101101010001111101000101010
  Trama modificada: 11010110110101101010001111101000101011

```
PS C:\Users\Kristopher\Documents\S8\Redes\Lab2Redes\Xavi> .\CRC32_emisor.exe
Ingrese una trama en binario: 1101

Trama codificada: 11010110110101101010001111101000101010
PS C:\Users\Kristopher\Documents\S8\Redes\Lab2Redes\Xavi> python3 CRC32_receptor.py
Introduce la trama codificada en binario: 11010110110101101010001111101000101011
Se detectaron errores. El mensaje se descarta.
```

  Trama original: 000
  Trama codificada: 00000000000000000000000000000000
  Trama modificada: 00000000000000010000000000000000

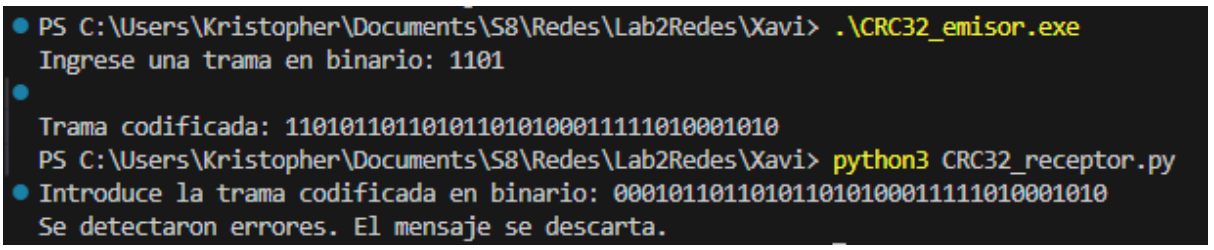
```
PS C:\Users\Kristopher\Documents\S8\Redes\Lab2Redes\Xavi> .\CRC32_emisor.exe
Ingrese una trama en binario: 000

Trama codificada: 00000000000000000000000000000000
PS C:\Users\Kristopher\Documents\S8\Redes\Lab2Redes\Xavi> python3 CRC32_receptor.py
Introduce la trama codificada en binario: 00000000000000010000000000000000
Se detectaron errores. El mensaje se descarta.
```

  Trama original: 00000
  Trama codificada: 00000000000000000000000000000000
  Trama modificada: 01000000000000000000000000000000

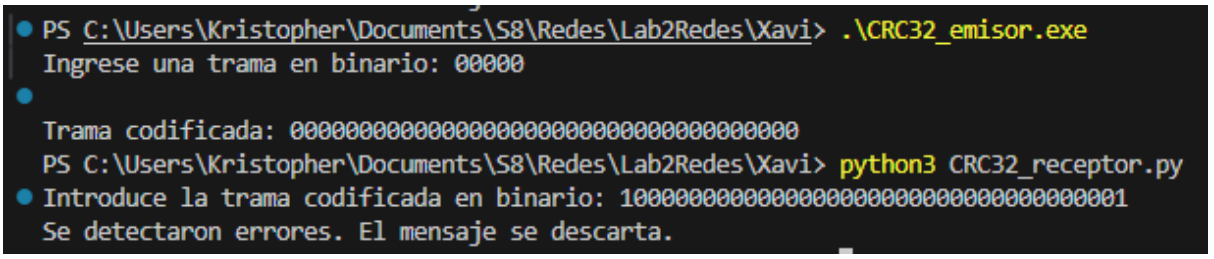
```
PS C:\Users\Kristopher\Documents\S8\Redes\Lab2Redes\Xavi> .\CRC32_emisor.exe
Ingrese una trama en binario: 00000

Trama codificada: 00000000000000000000000000000000
PS C:\Users\Kristopher\Documents\S8\Redes\Lab2Redes\Xavi> python3 CRC32_receptor.py
Introduce la trama codificada en binario: 01000000000000000000000000000000
Se detectaron errores. El mensaje se descarta.
```

- Dos o más errores:

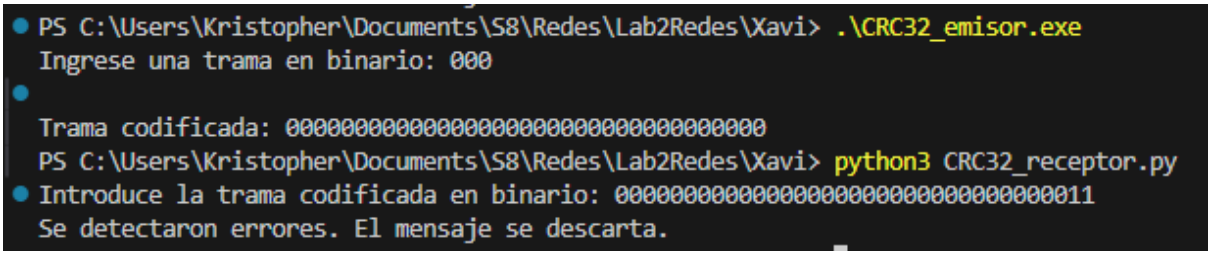
  Trama original: 1101
  Trama codificada: 11010110110101101010001111101000101O
  Trama modificada: 00010110110101101010001111101000101O



  Trama original: 00000
  Trama codificada: 00000000000000000000000000000000000
  Trama modificada: 10000000000000000000000000000000001



  Trama original: 000
  Trama codificada: 00000000000000000000000000000000000
  Trama modificada: 10000000000000000000000000000000011



- ¿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.

  En la implementación realizada CRC32 siempre detecta los errores en las pruebas, ya que es muy eficaz para detectar errores comunes. Sin embargo, la teoría subyacente demuestra que hay ciertos patrones específicos de errores múltiples que podrían no ser

detectados, aunque replicar estos patrones puede ser difícil y requiere ciertos escenarios muy específicos del polinomio generador y de la estructura del mensaje.

- En base a las pruebas que realizó, ¿qué ventajas y desventajas posee cada algoritmo con respecto a los otros dos? Tome en cuenta complejidad, velocidad, redundancia (overhead), etc.

  En este caso, en la implementación del algoritmo CRC32, me di cuenta que es eficiente y rápido para la detección de errores con bajo overhead, pero no corrige errores, siendo adecuado para aplicaciones de tiempo real. El código Hamming es simple y rápido, capaz de corregir errores de un bit y detectar errores de dos bits, aunque añade más bits de redundancia y es menos efectivo en entornos con alto ruido. El algoritmo de Viterbi es más potente en la corrección de múltiples errores, siendo ideal para sistemas de comunicación digital. La elección entre estos algoritmos depende de la necesidad de balancear velocidad, complejidad, overhead y capacidades de detección/corrección de errores en la aplicación específica.