UNIVERSIDAD DEL VALLE DE GUATEMALA

Laboratorio 2

Parte 1: Algoritmos de detección

Josue Say - 22801
José Prince – 22087

Redes

Código de Hamming

Pruebas realizadas

- Sin errores:
    - Prueba 1:
        - Emisor: 1011001
        - Receptor: 101001110010
            - Bits totales: 12
            - Bits de datos: 7
            - Resultado: 1011001
    - Prueba 2:
        - Emisor: 00110
        - Receptor: 1000011001
            - Bits totales: 12
            - Bits de datos: 7
            - Resultado: 00110
    - Prueba 3:
        - Emisor: 1110
        - Receptor: 00101101
            - Bits totales: 8
            - Bits de datos: 4
            - Resultado: 1110
- 1 error:
    - Prueba 1:
        - Emisor: 0100101
            - Respuesta: 110110001010
            - Procolo: (12, 7)
        - Receptor: 110110101010
            - Protocolo: (12,7)
            - Resultado: 0100101
    - Prueba 2:
        - Emisor: 00110
            - Respuesta: 1000011001
            - Procolo: (10, 5)
        - Receptor: 1001011001
            - Protocolo: (10,5)
            - Resultado: 00110
    - Prueba 3:
        - Emisor: 1110

                - Respuesta: 00101101
                - Procolo: (8, 4)
            - Receptor: 01101101
                - Protocolo: (8, 4)
                - Resultado: 1110
- 2 o más errores:
    - Prueba 1:
        - Emisor: 0100101
            - Respuesta: 110110001010
            - Procolo: (12, 7)
        - Receptor: 110111101010
            - Protocolo: (12, 7)
            - Resultado: RETRANSMITIR
    - Prueba 2:
        - Emisor: 00110
            - Respuesta: 1000011001
            - Procolo: (10, 5)
        - Receptor: 0101111001
            - Protocolo: (10, 5)
            - Resultado: RETRANSMITIR
    - Prueba 3:
        - Emisor: 1110
            - Respuesta: 00101101
            - Procolo: (8, 4)
        - Receptor: 10111110
            - Protocolo: (8, 4)
            - Resultado: RETRANSMITIR

Para ver el detale de cada uno de estos ejemplos se puede ejecutar el comando siguiendo las instrucciones que se encuentran en el repositorio. Al ejecutar tanto el emisor como el receptor se puede generar un reporte y detalle de todas las acciones que pasaron por parte del emisor y/o receptor, por ejemplo, usando la primera prueba para el segundo caso:

Emisor:

```
═══ DETALLE DE LA CODIFICACIÓN HAMMING ═══

Construcción del mensaje codificado:
- Posición 1: reservado para bit de paridad Hamming (r)
- Posición 2: reservado para bit de paridad Hamming (r)
- Posición 3: bit de datos '0' asignado desde msg[0]
- Posición 4: reservado para bit de paridad Hamming (r)
- Posición 5: bit de datos '1' asignado desde msg[1]
- Posición 6: bit de datos '0' asignado desde msg[2]
- Posición 7: bit de datos '0' asignado desde msg[3]
- Posición 8: reservado para bit de paridad Hamming (r)
- Posición 9: bit de datos '1' asignado desde msg[4]
- Posición 10: bit de datos '0' asignado desde msg[5]
- Posición 11: bit de datos '1' asignado desde msg[6]
- Posición 12: reservado para bit de paridad global (rg)

r1 cubre posiciones [1, 3, 5, 7, 9, 11] → valores: [1=None, 3=0, 5=1, 7=0, 9=1, 11=1]. Total de 1s: 3. Paridad par usada → bit de paridad: 1
r2 cubre posiciones [2, 3, 6, 7, 10, 11] → valores: [2=None, 3=0, 6=0, 7=0, 10=0, 11=1]. Total de 1s: 1. Paridad par usada → bit de paridad: 1
r4 cubre posiciones [4, 5, 6, 7] → valores: [4=None, 5=1, 6=0, 7=0]. Total de 1s: 1. Paridad par usada → bit de paridad: 1
r8 cubre posiciones [8, 9, 10, 11] → valores: [8=None, 9=1, 10=0, 11=1]. Total de 1s: 2. Paridad par usada → bit de paridad: 0
Paridad extendida cubre todas las posiciones excepto la final → valores: [1=1, 2=1, 3=0, 4=1, 5=1, 6=0, 7=0, 8=0, 9=1, 10=0, 11=1]. Total de 1s: 6. Paridad par usada → bit global: 0

═══ FIN DEL DETALLE ═══
```

```
Hamming   Windows PowerShell

(Hamming)
[22:17] Shell main ≡ ~2
D:\repositorios\UVG\2025\Lab2-Redes\Hamming
❯ python .\transmitter.py
═══ TRANSMISOR HAMMING ═══

Se utilizará la configuración definida en el archivo: 'D:\repositorios\UVG\2025\Lab2-Redes\Hamming\protocol.yaml'

Configuración cargada:
- Paridad: par
- Paridad extendida: sí

        Hint: Si deseas modificar el tipo de paridad (even/odd) o activar/desactivar la paridad extendida (true/false), edita el archivo 'protocol.yaml'

Ingrese los datos para codificar con Hamming:

Ingrese el mensaje binario (solo 0s y 1s): 0100101
Codificación final del mensaje: 110110001010
Protocolo de data: (12,7)

        Reporte generado en: 'D:\repositorios\UVG\2025\Lab2-Redes\Hamming\reports\t_hamming_report.txt'
        Detalle de los pasos seguidos, generado en: 'D:\repositorios\UVG\2025\Lab2-Redes\Hamming\reports\t_hamming_detail.txt'
```

Receptor:

r_hamming_detail.txt M

Hamming > reports > r_hamming_detail.txt

You, hace 3 horas | 1 author (You)

```
═══ DETALLE DE LA CODIFICACIÓN HAMMING ═══

Construcción del mensaje recibido:
  - Posición 1: bit recibido '1' (paridad Hamming, r)
  - Posición 2: bit recibido '1' (paridad Hamming, r)
  - Posición 3: bit recibido '0' (dato, d)
  - Posición 4: bit recibido '1' (paridad Hamming, r)
  - Posición 5: bit recibido '1' (dato, d)
  - Posición 6: bit recibido '0' (dato, d)
  - Posición 7: bit recibido '1' (dato, d)
  - Posición 8: bit recibido '0' (paridad Hamming, r)
  - Posición 9: bit recibido '1' (dato, d)
  - Posición 10: bit recibido '0' (dato, d)
  - Posición 11: bit recibido '1' (dato, d)
  - Posición 12: bit recibido '0' (paridad global, rg)

Inicializando msgBitsCalculated:
  - Bit de redundancia tipo 'r' inicializado en posición 1 con valor -1
  - Bit de redundancia tipo 'r' inicializado en posición 2 con valor -1
  - Bit de dato en posición 3 con valor 0 copiado
  - Bit de redundancia tipo 'r' inicializado en posición 4 con valor -1
  - Bit de dato en posición 5 con valor 1 copiado
  - Bit de dato en posición 6 con valor 0 copiado
  - Bit de dato en posición 7 con valor 1 copiado
  - Bit de redundancia tipo 'r' inicializado en posición 8 con valor -1
  - Bit de dato en posición 9 con valor 1 copiado
  - Bit de dato en posición 10 con valor 0 copiado
  - Bit de dato en posición 11 con valor 1 copiado
  - Bit de redundancia tipo 'rg' inicializado en posición 12 con valor -1

Calculando y asignando bits de redundancia:      You, hace 4 horas • feat: finalización receptro

r1 cubre posiciones [1, 3, 5, 7, 9, 11] → valores: [1=-1, 3=0, 5=1, 7=1, 9=1, 11=1]. Total de 1s: 4. Paridad par usada → bit de paridad: 0
  - Bit de redundancia r1 asignado con valor 0
r2 cubre posiciones [2, 3, 6, 7, 10, 11] → valores: [2=-1, 3=0, 6=0, 7=1, 10=0, 11=1]. Total de 1s: 2. Paridad par usada → bit de paridad: 0
  - Bit de redundancia r2 asignado con valor 0
r4 cubre posiciones [4, 5, 6, 7] → valores: [4=-1, 5=1, 6=0, 7=1]. Total de 1s: 2. Paridad par usada → bit de paridad: 0
  - Bit de redundancia r4 asignado con valor 0
r8 cubre posiciones [8, 9, 10, 11] → valores: [8=-1, 9=1, 10=0, 11=1]. Total de 1s: 2. Paridad par usada → bit de paridad: 0
  - Bit de redundancia r8 asignado con valor 0
Paridad extendida cubre todas las posiciones excepto la final → valores: [1=0, 2=0, 3=0, 4=0, 5=1, 6=0, 7=1, 8=0, 9=1, 10=0, 11=1]. Total de 1s: 4. Paridad par usada → bit global: 0
  - Bit de redundancia extendida rg asignado con valor 0

═══ DETECCIÓN DE ERRORES ═══
Comparación de bits de paridad recibidos vs calculados:
Bits de paridad (posRedundancyBits): [1, 2, 4, 8]
 - Posición 1: recibido = 1, calculado = 0
   → Diferencia detectada en bit de paridad r1
 - Posición 2: recibido = 1, calculado = 0
   → Diferencia detectada en bit de paridad r2
 - Posición 4: recibido = 1, calculado = 0
   → Diferencia detectada en bit de paridad r4
 - Posición 8: recibido = 0, calculado = 0
→ Se detectaron errores de paridad.
 - Posición decimal del error estimado: 7

Verificando bit de paridad extendida:
 - Paridad extendida recibida: 0
 - Paridad extendida calculada: 0
→ Bit global coincide → Se asume un único error, se puede corregir.
═══ FIN DE DETECCIÓN DE ERRORES ═══

═══ PROCESO DE CONSTRUCCIÓN DEL MENSAJE FINAL ═══

→ Se detectó un error en el mensaje.
→ Es un único error. Se intentará corregir.
 - Posición del error: 7
 - Bit antes de la corrección: 1
 - Bit después de la corrección: 0
 - Mensaje final (solo bits de datos): 0100101
═══ FIN DEL PROCESO DE CONSTRUCCIÓN ═══

═══ FIN DEL DETALLE ═══
```

Hamming | Windows PowerShell

```
[INFO] ── maven-resources-plugin:2.6:resources (default-resources) @ hamming-receiver ──
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] Copying 0 resource
[INFO]
[INFO] ── maven-compiler-plugin:3.1:compile (default-compile) @ hamming-receiver ──
[INFO] Changes detected - recompiling the module!
[WARNING] File encoding has not been set, using platform encoding UTF-8, i.e. build is platform dependent!
[INFO] Compiling 3 source files to /mnt/d/repositorios/UVG/2025/Lab2-Redes/Hamming/target/classes
[INFO]
[INFO] ── exec-maven-plugin:3.1.0:java (default-cli) @ hamming-receiver ──
═══ RECEPTOR HAMMING ═══

Se utilizará la configuración definida en el archivo: '/mnt/d/repositorios/UVG/2025/Lab2-Redes/Hamming/./protocol.yaml'

Configuración cargada:
- Paridad: par
- Paridad extendida: sí

        Hint: Si deseas modificar la configuración, edita el archivo 'protocol.yaml'

Ingrese los datos para decodificar con Hamming:

Ingrese la trama binaria (solo 0s y 1s): 110110001010
Ingrese el total de bits: 12
Ingrese la cantidad de bits de datos: 7

Resultado de la decodificación:
- Mensaje final decodificado: 0100101
- Protocolo de data: (12,7)

        Reporte generado en: '/mnt/d/repositorios/UVG/2025/Lab2-Redes/Hamming/./reports/r_hamming_report.txt'
        Detalle de los pasos seguidos, generado en: '/mnt/d/repositorios/UVG/2025/Lab2-Redes/Hamming/./reports/r_hamming_detail.txt'
[INFO] ────────────────────────────────────────────────────────────────────────
[INFO] BUILD SUCCESS
[INFO] ────────────────────────────────────────────────────────────────────────
[INFO] Total time:  7.098 s
[INFO] Finished at: 2025-07-23T22:17:52-06:00
[INFO] ────────────────────────────────────────────────────────────────────────
```

Fletcher checksum

Para la implementación del algoritmo "Fletcher checksum" se realizó el emisor del mensaje mediante el lenguaje de programación “Lua” en donde como input se le pasa una trama y la configuración de fletcher que se desea utilizar y devuelve el mensaje codificado. En el caso del receptor se utilizó el lenguaje "Python" que recibe el mensaje codificado y devuelve si el mensaje recibido es aceptado, en caso de detectar errores indica en cuál de las sumas se encontró una desigualdad.

Pruebas realizadas

- Sin errores:
- 0111001101101001

En este caso se puede ver que para esta trama de 16 bit siempre detecta correctamente que el mensaje es válido, habiéndolo probado con todas las configuraciones del algoritmo.

```
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 0111001101101001
Insert fletcher type (16|32|64): 16
Checksum: 0111001101101001010100001101110O
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 0111001101101001010100001101110O
Insert fletcher type (16|32|64): 16
Valid Message
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 0111001101101001
Insert fletcher type (16|32|64): 32
Checksum: 01110011011010010111001101101001011100110110100l
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 01110011011010010111001101101001011100110110100l
Insert fletcher type (16|32|64): 32
Valid Message
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 0111001101101001
Insert fletcher type (16|32|64): 64
Checksum: 0111001101101001000000000000000011100110110100100000000000000001110011011010010000000000000000
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 0111001101101001000000000000000011100110110100100000000000000001110011011010010000000000000000
Insert fletcher type (16|32|64): 64
Valid Message
[akice@Akice-PC Lab2-Redes]$
```

- 01100010

Para este caso de una trama de 8 bits vemos que identifica correctamente el mensaje cuando no encuentra ningún error.

```
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 01100010
Insert fletcher type (16|32|64): 16
Checksum: 01100010011000100110010
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 01100010011000100110010
Insert fletcher type (16|32|64): 16
Valid Message
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 01100010
Insert fletcher type (16|32|64): 32
Checksum: 01100010000000000110001000000000110001000000000
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 01100010000000000110001000000000110001000000000
Insert fletcher type (16|32|64): 32
Valid Message
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 01100010
Insert fletcher type (16|32|64): 64
Checksum: 0110001000000000000000000000000011000100000000000000000000000011000100000000000000000000000000
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 0110001000000000000000000000000011000100000000000000000000000011000100000000000000000
00000000
Insert fletcher type (16|32|64): 64
Valid Message
[akice@Akice-PC Lab2-Redes]$
```

- 0011000100

De igual forma acá para una trama de 10 bits se ve que para cada una de las configuraciones se ve que detecta correctamente que ningún mensaje tiene errores.

```
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 0011000100
Insert fletcher type (16|32|64): 16
Checksum: 0011000100000000110001000110001
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 0011000100000000110001000110001
Insert fletcher type (16|32|64): 16
Valid Message
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 0011000100
Insert fletcher type (16|32|64): 32
Checksum: 00110001000000000110001000000000011000100000000
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 00110001000000000110001000000000011000100000000
Insert fletcher type (16|32|64): 32
Valid Message
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 0011000100
Insert fletcher type (16|32|64): 64
Checksum: 0011000100000000000000000000000011000100000000000000000000000011000100000000000000000000000000
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 0011000100000000000000000000000011000100000000000000000000000011000100000000000000000000
00000000
Insert fletcher type (16|32|64): 64
Valid Message
[akice@Akice-PC Lab2-Redes]$
```

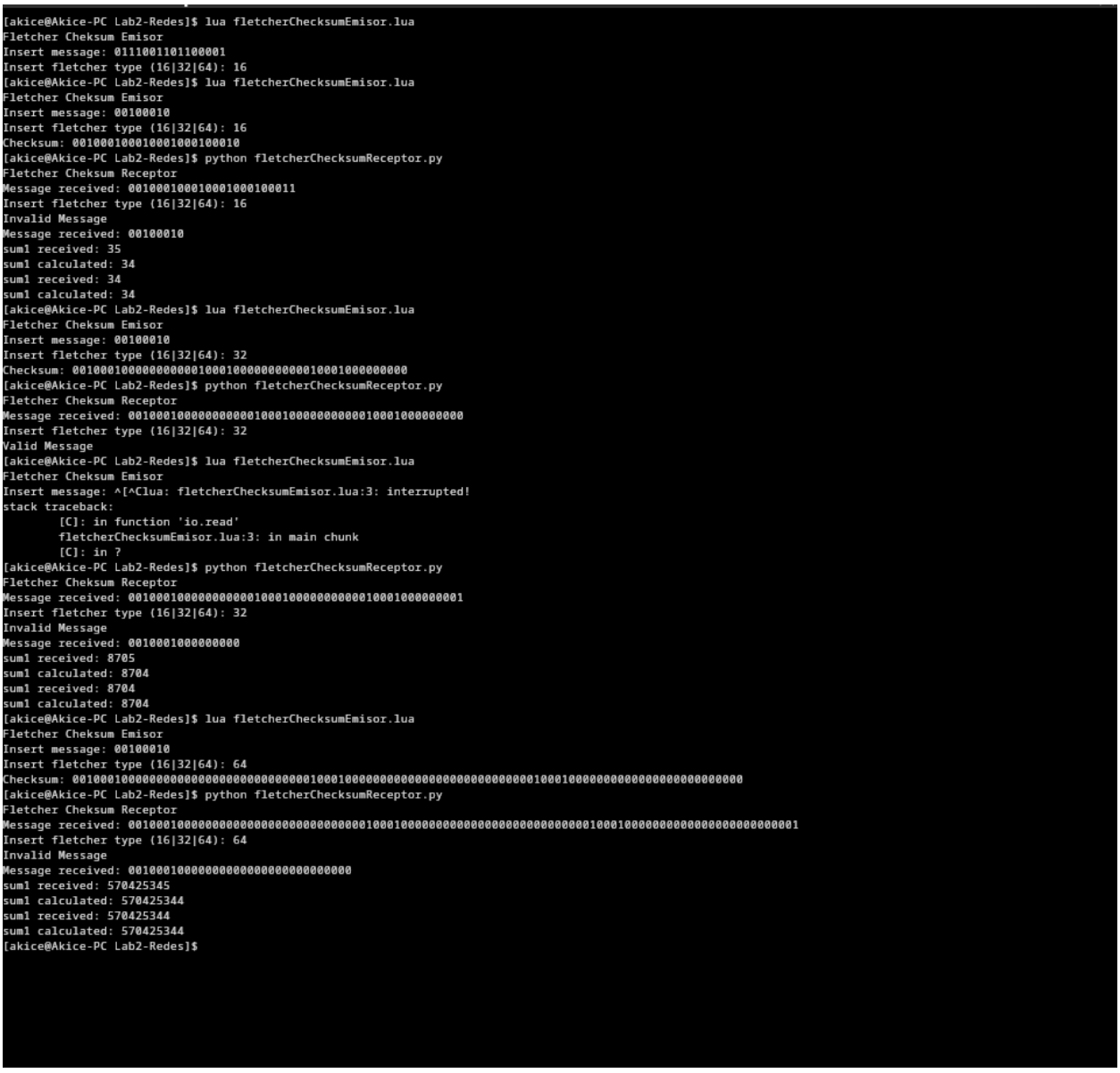
- 1 error:
- 0111001101100001

En este caso se modifica el último bit de cada mensaje codificado. Para cada configuración de fletcher se ve que detecta correctamente, en este caso se ve que detecta que una de las sumas es incorrecta en este caso como solo se modifica el último bit indica que solo una de las sumas es errónea con respecto a la suma del mensaje enviado.

```
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 0111001101100001
Insert fletcher type (16|32|64): 16
Checksum: 0111001101100001010010001101010 0
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 0111001101100001010010001101011 0
Insert fletcher type (16|32|64): 16
Invalid Message
Message received: 0111001101100001
sum1 received: 214
sum1 calculated: 212
sum1 received: 72
sum1 calculated: 72
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: ^Clua: fletcherChecksumEmisor.lua:3: interrupted!
stack traceback:
        [C]: in function 'io.read'
        fletcherChecksumEmisor.lua:3: in main chunk
        [C]: in ?
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 0111001101100001
Insert fletcher type (16|32|64): 32
Checksum: 0111001101100001011100110110000101110011011000 01
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 0111001101100001011100110110000101110011011000 011
Insert fletcher type (16|32|64): 32
Invalid Message
Message received: 0111001101100001
sum1 received: 29539
sum1 calculated: 29537
sum1 received: 29537
sum1 calculated: 29537
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 0111001101100001
Insert fletcher type (16|32|64): 64
Checksum: 0111001101100001000000000000000001110011011000010000000000000000011100110110000100000000000000000 01
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 0111001101100001000000000000000001110011011000010000000000000000011100110110000100000000000000001 01
Insert fletcher type (16|32|64): 64
Invalid Message
Message received: 0111001101100001000000000000000 0
sum1 received: 1935736833
sum1 calculated: 1935736832
sum1 received: 1935736832
sum1 calculated: 1935736832
[akice@Akice-PC Lab2-Redes]$
```

- 00100010

Ahora para la trama de 8 bits se tiene un comportamiento similar que, con la trama de 16 bits, viendo que identifica que una de las sumas es errónea después de comparar las sumas del mensaje enviado con el recibido.



- 0011000110

Para las tramas de 10 bits se ve que para cada configuración de fletcher también detecta que hay un error en una de las sumas realizadas para comprobar la integridad del mensaje.

```
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 0011000110
Insert fletcher type (16|32|64): 16
Checksum: 00110001100000001110001010110001
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 00110001100000001110001010110000
Insert fletcher type (16|32|64): 16
Invalid Message
Message received: 0011000110000000
sum1 received: 176
sum1 calculated: 177
sum1 received: 226
sum1 calculated: 226
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 0011000110
Insert fletcher type (16|32|64): 32
Checksum: 00110001100000000011000110000000011000110000000
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 00110001100000000011000110000000011000110000001
Insert fletcher type (16|32|64): 32
Invalid Message
Message received: 0011000110000000
sum1 received: 12673
sum1 calculated: 12672
sum1 received: 12672
sum1 calculated: 12672
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 0011000110
Insert fletcher type (16|32|64): 64
Checksum: 0011000110000000000000000000000011000110000000000000000000000011000110000000000000000000000
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 0011000110000000000000000000000011000110000000000000000000000011000110000000000000
00000001
Insert fletcher type (16|32|64): 64
Invalid Message
Message received: 00110001100000000000000000000000
sum1 received: 830472193
sum1 calculated: 830472192
sum1 received: 830472192
sum1 calculated: 830472192
[akice@Akice-PC Lab2-Redes]$
```

- 2 o más errores:
- 1111000101101001

Ahora con la detección de 2 o más errores se ve que sigue detectando los errores en este caso solo se detecta errores en solo una de las sumas ya que los bits cambiados (en todas las configuraciones) fueron los últimos bits.

```
Insert message: 1111000101101001
Insert fletcher type (16|32|64): 16
Checksum: 1111000101101001010011010101011
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 1111000101101001010011010101011000
Insert fletcher type (16|32|64): 16
Invalid Message
Message received: 1111000101101001
sum1 received: 88
sum1 calculated: 91
sum1 received: 77
sum1 calculated: 77
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: ^CTraceback (most recent call last):
  File "/home/akice/Documents/Repositories/Lab2-Redes/fletcherChecksumReceptor.py", line 2, in <module>
    msg = input("Message received: ")
KeyboardInterrupt

[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 1111000101101001
Insert fletcher type (16|32|64): 32
Checksum: 11110001011010011111000101101001111100010110100l
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 11110001011010011111000101101001111100010110111O
Insert fletcher type (16|32|64): 32
Invalid Message
Message received: 1111000101101001
sum1 received: 61806
sum1 calculated: 61801
sum1 received: 61801
sum1 calculated: 61801
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 1111000101101001
Insert fletcher type (16|32|64): 64
Checksum: 1111000101101001000000000000000011110001011010010000000000000000111100010110100100000000000000
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 1111000101101001000000000000000011110001011010010000000000000000111100010110100100000000000001111
Insert fletcher type (16|32|64): 64
Invalid Message
Message received: 1111000101101001000000000000000
sum1 received: 4050190351
sum1 calculated: 4050190336
sum1 received: 4050190336
sum1 calculated: 4050190336
[akice@Akice-PC Lab2-Redes]$
```

- 01110000

En este caso de igual forma se detectaron los errores en los mensajes, incluso cuando para cada configuración de fletcher se fue aumentando la cantidad de errores , siempre al final del mensaje.

```
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 01110000
Insert fletcher type (16|32|64): 16
Checksum: 01110000111000001110000
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 01110000111000001110011
Insert fletcher type (16|32|64): 16
Invalid Message
Message received: 01110000
sum1 received: 115
sum1 calculated: 112
sum1 received: 112
sum1 calculated: 112
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 01110000
Insert fletcher type (16|32|64): 32
Checksum: 01110000000000001110000000000001110000000000000
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 01110000000000001110000000000001110000000000111
Insert fletcher type (16|32|64): 32
Invalid Message
Message received: 0111000000000000
sum1 received: 28679
sum1 calculated: 28672
sum1 received: 28672
sum1 calculated: 28672
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 01110000
Insert fletcher type (16|32|64): 64
Checksum: 0111000000000000000000000000000011100000000000000000000000000011100000000000000000000000000000
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 0111000000000000000000000000000011100000000000000000000000000011100000000000000000000
00001111
Insert fletcher type (16|32|64): 64
Invalid Message
Message received: 01110000000000000000000000000000
sum1 received: 1879048207
sum1 calculated: 1879048192
sum1 received: 1879048192
sum1 calculated: 1879048192
[akice@Akice-PC Lab2-Redes]$
```

- 1011001111

Para una trama de 10 bits también detecta los errores para cada una de las configuraciones, al igual que con las pruebas anteriores se fue incrementando la cantidad de errores fue aumentando con cada configuración y se ve que sigue detectando los errores.

```
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 1011001111
Insert fletcher type (16|32|64): 16
Checksum: 10110011110000000010100001110100
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 10110011110000000010100001110111
Insert fletcher type (16|32|64): 16
Invalid Message
Message received: 1011001111000000
sum1 received: 119
sum1 calculated: 116
sum1 received: 40
sum1 calculated: 40
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 1011001111
Insert fletcher type (16|32|64): 32
Checksum: 10110011110000001011001111000000101100111100000
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 10110011110000001011001111000000101100111100000111
Insert fletcher type (16|32|64): 32
Invalid Message
Message received: 1011001111000000
sum1 received: 46023
sum1 calculated: 46016
sum1 received: 46016
sum1 calculated: 46016
[akice@Akice-PC Lab2-Redes]$ lua fletcherChecksumEmisor.lua
Fletcher Cheksum Emisor
Insert message: 1011001111
Insert fletcher type (16|32|64): 64
Checksum: 1011001111000000000000000000000010110011110000000000000000000000101100111100000000000000000000
[akice@Akice-PC Lab2-Redes]$ python fletcherChecksumReceptor.py
Fletcher Cheksum Receptor
Message received: 1011001111000000000000000000000010110011110000000000000000000001011001111000000000000000
00011111
Insert fletcher type (16|32|64): 64
Invalid Message
Message received: 10110011110000000000000000000000
sum1 received: 3015704607
sum1 calculated: 3015704576
sum1 received: 3015704576
sum1 calculated: 3015704576
[akice@Akice-PC Lab2-Redes]$
```

Preguntas

¿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, demuéstrelo con su implementación.

Hamming

Sí, es posible manipular múltiples bits y que el algoritmo no detecte los errores, o peor aún, que asuma que solo hay un error y aplique una corrección incorrecta.

En una de las pruebas realizadas, modificamos todos los bits del mensaje original (1000011001 → 1111111111), es decir, más de un bit fue alterado. Sin embargo, el algoritmo asumió que solo un bit estaba en error (el bit de paridad en la posición 8) y aplicó una "corrección". El resultado fue que el mensaje final decodificado fue "11110", el cual coincidía con el mensaje original del emisor: 00110.

Esto ocurre porque:

- La suma de paridades Hamming coincidía en todas menos una, haciendo que el algoritmo estime un solo bit de error.
- La paridad global (bit extendido) también coincidía, lo cual reforzó la falsa suposición de un único error.
- Al corregir ese único bit (posición 8), el mensaje reconstruido resultó igual al original por coincidencia, aunque en realidad el mensaje estaba completamente corrupto.

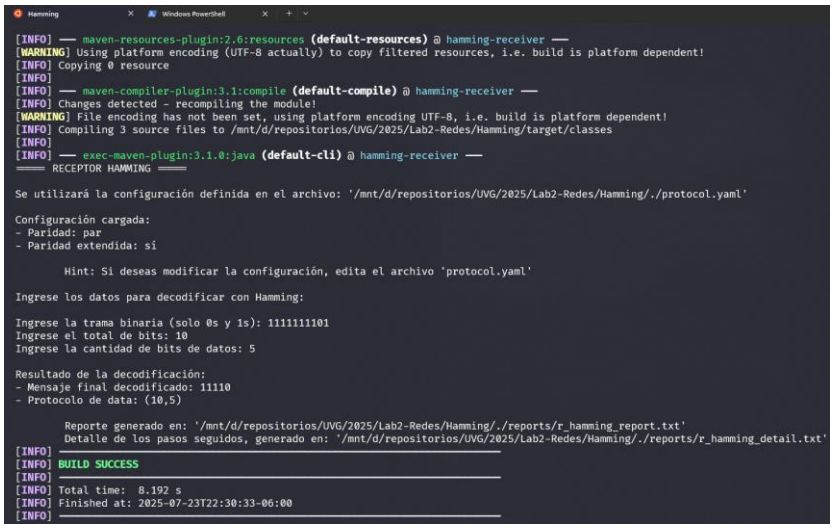
Ejemplo:

Emisor:

Entrada: 00110

Receptor:

1. Mensaje recibido por emisor: 1000011001
2. Entrada (error en todos los bits 0's): 1111111111
3. Bits totales (n): 10
4. Bits de datos (m): 5



r_hamming_report.txt M

Hamming > reports > r_hamming_report.txt

You, hace 3 minutos | 1 author (You)

```
═══ HAMMING TRANSMITTER REPORT ═══

Configuración:
- Protocolo de data: (10,5)
- Paridad usada: par
- Paridad extendida: sí
- Bits de datos: 5
- Bits de redundancia Hamming: 4      You, hace 4 horas
- Bit de redundancia global (extendido): 1
- Bits totales (recibidos): 10
- Posiciones de redundancia Hamming: [1, 2, 4, 8]
- Posición de bit de redundancia global (extendido): 10

═══ Información RECIBIDA ═══
Mensaje recibido (bits de datos):
1111111101

Bits recibidos (posiciones y tipos):
Posición 1: Tipo = r
Posición 1: Tipo = r
Posición 1: Tipo = d
Posición 1: Tipo = r
Posición 1: Tipo = d
Posición 1: Tipo = d
Posición 1: Tipo = d
Posición 1: Tipo = r
Posición 0: Tipo = d
Posición 1: Tipo = rg

═══ Información CALCULADA ═══
Mensaje codificado (msgBitsCalculated):
Posición  Bit Tipo
1   1 r
2   1 r
3   1 d
4   1 r
5   1 d
6   1 d
7   1 d
8   1 r
9   0 d
10    1 rg

Cobertura de cada bit de paridad Hamming:
r1 (pos 1) → cubre: [1, 3, 5, 7, 9]
r2 (pos 2) → cubre: [2, 3, 6, 7]
r4 (pos 4) → cubre: [4, 5, 6, 7]
r8 (pos 8) → cubre: [8, 9]

Mensaje binario final (para transmisión):
11110

═══ FIN DEL REPORTE ═══
```

r_hamming_detail.txt M

Hamming > reports > r_hamming_detail.txt

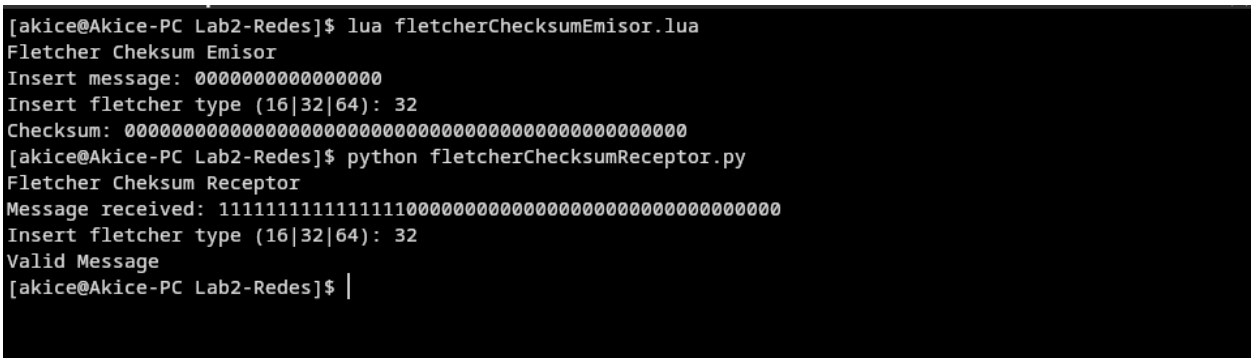
You, hace 3 minutos | 1 author (You)

```
═══ DETALLE DE LA CODIFICACIÓN HAMMING ═══

Construcción del mensaje recibido:
  - Posición 1: bit recibido '1' (paridad Hamming, r)
  - Posición 2: bit recibido '1' (paridad Hamming, r)
  - Posición 3: bit recibido '1' (dato, d)
  - Posición 4: bit recibido '1' (paridad Hamming, r)
  - Posición 5: bit recibido '1' (dato, d)
  - Posición 6: bit recibido '1' (dato, d)
  - Posición 7: bit recibido '1' (dato, d)
  - Posición 8: bit recibido '1' (paridad Hamming, r)
  - Posición 9: bit recibido '0' (dato, d)
  - Posición 10: bit recibido '1' (paridad global, rg)

Inicializando msgBitsCalculated:
  - Bit de redundancia tipo 'r' inicializado en posición 1 con valor -1
  - Bit de redundancia tipo 'r' inicializado en posición 2 con valor -1
  - Bit de dato en posición 3 con valor 1 copiado
  - Bit de redundancia tipo 'r' inicializado en posición 4 con valor -1
  - Bit de dato en posición 5 con valor 1 copiado      You, hace 4 horas • feat: finalización receptro
  - Bit de dato en posición 6 con valor 1 copiado
  - Bit de dato en posición 7 con valor 1 copiado
  - Bit de redundancia tipo 'r' inicializado en posición 8 con valor -1
  - Bit de dato en posición 9 con valor 0 copiado
  - Bit de redundancia tipo 'rg' inicializado en posición 10 con valor -1

Calculando y asignando bits de redundancia:

r1 cubre posiciones [1, 3, 5, 7, 9] → valores: [1=-1, 3=1, 5=1, 7=1, 9=0]. Total de 1s: 3. Paridad par usada → bit de paridad: 1
    - Bit de redundancia r1 asignado con valor 1
r2 cubre posiciones [2, 3, 6, 7] → valores: [2=-1, 3=1, 6=1, 7=1]. Total de 1s: 3. Paridad par usada → bit de paridad: 1
    - Bit de redundancia r2 asignado con valor 1
r4 cubre posiciones [4, 5, 6, 7] → valores: [4=-1, 5=1, 6=1, 7=1]. Total de 1s: 3. Paridad par usada → bit de paridad: 1
    - Bit de redundancia r4 asignado con valor 1
r8 cubre posiciones [8, 9] → valores: [8=-1, 9=0]. Total de 1s: 0. Paridad par usada → bit de paridad: 0
    - Bit de redundancia r8 asignado con valor 0
Paridad extendida cubre todas las posiciones excepto la final → valores: [1=1, 2=1, 3=1, 4=1, 5=1, 6=1, 7=1, 8=0, 9=0]. Total de 1s: 7. Paridad par usada → bit global: 1
    - Bit de redundancia extendida rg asignado con valor 1

═══ DETECCIÓN DE ERRORES ═══
Comparación de bits de paridad recibidos vs calculados:
Bits de paridad (posRedundancyBits): [1, 2, 4, 8]
  - Posición 1: recibido = 1, calculado = 1
  - Posición 2: recibido = 1, calculado = 1
  - Posición 4: recibido = 1, calculado = 1
  - Posición 8: recibido = 1, calculado = 0
    → Diferencia detectada en bit de paridad r8
→ Se detectaron errores de paridad.
  - Posición decimal del error estimado: 8

Verificando bit de paridad extendida:
  - Paridad extendida recibida: 1
  - Paridad extendida calculada: 1
→ Bit global coincide → Se asume un único error, se puede corregir.
═══ FIN DE DETECCIÓN DE ERRORES ═══

═══ PROCESO DE CONSTRUCCIÓN DEL MENSAJE FINAL ═══

→ Se detectó un error en el mensaje.
→ Es un único error. Se intentará corregir.
  - Posición del error: 8
  - Bit antes de la corrección: 0
  - Bit después de la corrección: 1
  - Mensaje final (solo bits de datos): 11110
═══ FIN DEL PROCESO DE CONSTRUCCIÓN ═══

═══ FIN DEL DETALLE ═══
```

Fletcher checksum

Sí es posible manipular los bits de un mensaje de tal forma que el algoritmo de Fletcher no detecter el error. Esto se debe a que Fletcher no es un código de detección perfecto. Esto es porque el algoritmo de Fletcher se basa en dos sumas acumulativas sobre los bytes del mensaje. Si se alteran ciertos bits de forma que las sumas acumulativas se mantengan iguales, el checksum será el mismo, y el recepto no detectará el error.

Por ejemplo, en este caso extremo tenemos la siguiente trama de 16 bits: 0000000000000000, en el caso de que todos los bits se flipearan dando una trama de 16 1's entonces el algoritmo fallaría y lo podemos ver a continuación:



En este caso se flipearon todos los bits y se ve que para la configuración de fletcher-32 este caso falla detectando los errores en el mensaje.

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

Hamming

Ventajas:

- Permite detectar y corregir automáticamente un único bit con error, lo cual está bien si se sabe que solo vendrá un error
- La paridad extendida permite detectar casos con múltiples errores, lo cual reduce el riesgo de corrección errónea.

Desventajas:

- No es infalible ante múltiples errores: si las paridades engañan al algoritmo, puede hacer una corrección equivocada (como se demostró en el caso donde todos los bits fueron modificados).
- Tiene un overhead de bits de redundancia mayor que otros algoritmos (como la simple paridad o Fletcher).

- Su implementación es más compleja que la de un simple checksum o bit de paridad.

Incluso con más de un bit alterado, el algoritmo en un caso devolvió el mensaje correcto por coincidencia matemática de las paridades, lo que representa una debilidad estructural.

Al intentar validar los resultados del código Hamming con tramas de prueba generadas por otro compañero, notamos que las cadenas no coincidían en estructura ni longitud. Esto se debió a que el otro compañero usó el algoritmo Fletcher checksum, por ejemplo, Las cadenas de Fletcher eran de 16 o más bits, con longitudes y patrones diferentes a los usados por Hamming (que utiliza bits de paridad en potencias de 2 y tamaños como 7, 12, 15 bits).

Fletcher checksum

Ventajas:

- Es capaz de detectar múltiples errores en las diferentes partes del mensaje codificado, detectando errores de manera eficaz en la mayoría de los casos.
- Su implementación es bastante simple y no involucra una complejidad alta computacionalmente.
- Admite diferentes configuraciones (16, 32 y 64 bits), permitiendo que este algoritmo se adapte al nivel de robustez que se requiera.

Desventajas:

- Solo detecta los errores, no los corrige.
- Puede fallar en detectar algunos patrones de error, casos como en la cancelación de sumas modulares.
- Tampoco permite saber dónde o cuántos errores se obtuvieron, solo indica que hubo uno o varios errores en el mensaje, indicando al comprobar que las sumas son diferentes.

Conclusiones

- Se puede ver que el algoritmo de Fletcher este es un algoritmo que sirve para la detección de errores. Este algoritmo ofrece una implementación sencilla a cambio de que no se indica donde es que están los errores del mensaje, solo se muestra que hubo errores (incluso se desconoce la cantidad de errores). No es 100% efectivo, ya que se demostro que hay casos en donde este algoritmo falla detectando errores. En conclusión, el algoritmo de Fletcher checksum es fácil y

rápido de implementar a cambio de presentar fallas que son resueltas por otros algoritmos, tanto de detección como de corrección de errores.

- No es posible usar las mismas entradas para ambos algoritmos, aunque la intención era comparar los algoritmos usando los mismos mensajes, esto no fue factible porque:
    - Hamming requiere posiciones de paridad específicas (1, 2, 4, 8...).
    - Fletcher opera sobre bloques completos (8, 16 bits) y agrega un checksum que no tiene relación directa con la estructura de Hamming.
    - Por tanto, las salidas codificadas no son compatibles ni pueden ser validadas por el otro algoritmo.
- En una prueba extrema donde se invirtieron todos los bits del mensaje codificado de Hamming, el receptor asumió erróneamente que había un solo error (ya que la paridad extendida coincidía), aplicó una corrección y el mensaje final resultó ser igual al original por coincidencia, aunque el mensaje estaba completamente corrupto.
    - Esto demostró una debilidad del algoritmo en escenarios con múltiples errores y paridades engañosas.
- El receptor de Hamming solo funciona correctamente si se le proporciona:
    - La trama binaria codificada.
    - La cantidad exacta de bits (n) y bits de datos (m).
    - Usar una entrada incorrecta o incompleta puede llevar a fallos en la detección, errores de cálculo o falsas correcciones.