Memoria Práctica 2

Carlos Moreno Antorrena
Marcos Malandro Pérez
Pablo Ruiz Calvo

Ejercicio 1.

Diseñe una función que simule un canal ruidoso. La función admitirá un vector binario de longitud L
bytes y devolverá un vector de la misma longitud donde la probabilidad de que un bit haya cambiado de valor será f. Ambos vectores binarios se pasarán empaquetados en un tipo binario
mayor como por ejemplo el unsigned char
La signatura de la función void noisyChannel(unsigned char *in, unsigned char *out, int l, float f);

```C/C++
/*
 * Práctica 2.1 - Simulación de un canal ruidoso
 *
 * Esta función simula un canal ruidoso para comunicaciones binarias.
 * Recibe un vector binario de entrada, una longitud L en bytes, y una probabilidad de error f.
 * Devuelve un vector de salida donde cada bit tiene una probabilidad f de haber cambiado su valor.
 */

#include <Arduino.h>

/**
 * Función que simula un canal ruidoso.
 * @param in Vector binario de entrada empaquetado en unsigned char
 * @param out Vector binario de salida empaquetado en unsigned char
 * @param l Longitud del vector en bytes
 * @param f Probabilidad de que un bit cambie de valor (entre 0 y 1)
 */
void noisyChannel(unsigned char *in, unsigned char *out, int l, float f) {
  // Inicializar la semilla para la generación de números aleatorios
  // En una implementación real, esto debería hacerse solo una vez al inicio del programa
  randomSeed(analogRead(0));

  // Recorrer cada byte del vector de entrada
  for (int i = 0; i < l; i++) {
       // Inicializar el byte de salida con el valor del byte de entrada
       out[i] = in[i];

       // Procesar cada bit del byte actual
       for (int j = 0; j < 8; j++) {
```

```
        // Generar un número aleatorio entre 0 y 1
        float r = random(0, 100) / 100.0;

        // Si el número aleatorio es menor que la probabilidad de error f,
        // invertir el bit correspondiente en el byte de salida
        if (r < f) {
        // Invertir el bit j-ésimo usando XOR con una máscara
        // La máscara tiene un 1 en la posición j y 0 en las demás posiciones
        out[i] ^= (1 << j);
        }
        }
  }
}

// Función para imprimir un vector de bytes en formato binario
void printBinaryVector(unsigned char *vec, int length) {
  for (int i = 0; i < length; i++) {
        for (int j = 7; j >= 0; j--) {
        // Extraer y mostrar cada bit, empezando por el más significativo
        Serial.print((vec[i] >> j) & 1);
        }
        Serial.print(" ");
  }
  Serial.println();
}

void setup() {
  // Inicializar comunicación serial
  Serial.begin(9600);
  while (!Serial) {
        ; // Esperar a que el puerto serial se conecte
  }

  // Ejemplo de uso de la función noisyChannel
  const int length = 3; // Longitud del vector en bytes
  unsigned char input[length] = {0b10101010, 0b11110000, 0b00001111};
  unsigned char output[length];
  float errorProb = 0.2; // Probabilidad de error del 20%

  Serial.println("Vector de entrada:");
  printBinaryVector(input, length);

  // Aplicar el canal ruidoso
  noisyChannel(input, output, length, errorProb);

  Serial.println("Vector de salida (después del canal ruidoso):");
  printBinaryVector(output, length);
```

```
  // Contar bits que han cambiado
  int changedBits = 0;
  for (int i = 0; i < length; i++) {
       unsigned char diff = input[i] ^ output[i]; // XOR para detectar bits
diferentes
       for (int j = 0; j < 8; j++) {
       if ((diff >> j) & 1) {
       changedBits++;
       }
       }
  }

  Serial.print("Bits cambiados: ");
  Serial.print(changedBits);
  Serial.print(" de ");
  Serial.print(length * 8);
  Serial.print(" (");
  Serial.print((float)changedBits / (length * 8) * 100);
  Serial.println("%)");
}

void loop() {
  // No se requiere código en el loop para esta práctica
}
```

Ejercicio 2.

Diseñar un codificador de repetición que grado Rn donde n es el número de bits que se repiten en
cada bloque. A la función se le pasa como parámetros el mensaje a codificar en binario empaquetado en caracteres, la longitud l en bytes del mensaje, n el grado del codificador de repetición y un vector vacío suficientemente grande como para almacenar el resultado.
La signatura de la función void reptitionCoder(unsigned char *in, unsigned char *out, int l, int n);

```
C/C++
/*
 * Práctica 2.2 - Codificador de repetición
 *
 * Esta función implementa un codificador de repetición de grado Rn donde n
es el número
 * de bits que se repiten en cada bloque. Recibe un mensaje binario
empaquetado en bytes,
 * y genera una versión codificada donde cada bit se repite n veces.
 */
```

```
#include <Arduino.h>
/*
 * Función que implementa un codificador de repetición.
 * @param in Vector binario de entrada empaquetado en unsigned char
 * @param out Vector binario de salida empaquetado en unsigned char
 * @param l Longitud del vector de entrada en bytes
 * @param n Grado del codificador de repetición (número de veces que se
repite cada bit)
 */
void repetitionCoder(unsigned char *in, unsigned char *out, int l, int n) {
  int outBitIndex = 0; // Índice del bit actual en el vector de salida

  // Recorrer cada byte del vector de entrada
  for (int i = 0; i < l; i++) {
       // Procesar cada bit del byte actual
       for (int j = 7; j >= 0; j--) {
       // Extraer el bit j-ésimo del byte i-ésimo
       unsigned char bit = (in[i] >> j) & 1;

       // Repetir el bit n veces
       for (int k = 0; k < n; k++) {
       // Calcular el índice del byte de salida donde se colocará el bit
repetido
       int outByteIndex = outBitIndex / 8;
       // Calcular la posición del bit dentro del byte de salida
       int outBitPosition = 7 - (outBitIndex % 8);

       // Si estamos en un nuevo byte, inicializarlo a 0
       if (outBitPosition == 7) {
       out[outByteIndex] = 0;
       }

       // Colocar el bit en la posición correspondiente del byte de salida
       out[outByteIndex] |= (bit << outBitPosition);

       // Incrementar el índice del bit de salida
       outBitIndex++;
       }
       }
  }
}

// Función para imprimir un vector de bytes en formato binario
void printBinaryVector(unsigned char *vec, int length) {
  for (int i = 0; i < length; i++) {
       for (int j = 7; j >= 0; j--) {
       // Extraer y mostrar cada bit, empezando por el más significativo
       Serial.print((vec[i] >> j) & 1);
```

```
        }
        Serial.print(" ");
    }
    Serial.println();
}

void setup() {
    // Inicializar comunicación serial
    Serial.begin(9600);
    while (!Serial) {
        ; // Esperar a que el puerto serial se conecte
    }

    // Ejemplo de uso del codificador de repetición
    const int length = 2; // Longitud del vector de entrada en bytes
    unsigned char input[length] = {0b10101010, 0b11110000};
    int n = 6; // Grado del codificador (cada bit se repite 3 veces)

    // Calcular el tamaño necesario para el vector de salida
    // Cada bit de entrada genera n bits de salida
    int outputLength = (length * 8 * n + 7) / 8; // Redondeo hacia arriba
    unsigned char output[outputLength];

    Serial.println("Vector de entrada:");
    printBinaryVector(input, length);

    // Aplicar el codificador de repetición
    repetitionCoder(input, output, length, n);

    Serial.println("Vector de salida (después del codificador de
repetición):");
    printBinaryVector(output, outputLength);

    // Mostrar información sobre la codificación
    Serial.print("Grado de repetición: ");
    Serial.println(n);
    Serial.print("Bits de entrada: ");
    Serial.println(length * 8);
    Serial.print("Bits de salida: ");
    Serial.println(outputLength * 8);
}

void loop() {
}
```

Ejercicio 3.

Diseñar un decodificador de repetición de grado Rn donde n es el número de bits que se repiten en cada bloque. A la función se le pasa como parámetros el mensaje a decodificar en binario empaquetado en caracteres, la longitud l en bytes del mensaje, n el grado del decodificador de repetición y un vector vacío suficientemente grande como para almacenar el resultado.
La signatura de la función void reptitionDecoder(unsigned char *in, unsigned char *out, int l, int n);

```C/C++
/*
 * Práctica 2.3 - Decodificador de repetición
 *
 * Esta función implementa un decodificador de repetición de grado Rn donde
n es el número
 * de bits que se repiten en cada bloque. Recibe un mensaje codificado donde
cada bit original
 * se ha repetido n veces, y recupera el mensaje original mediante un
sistema de votación por mayoría.
 */
#include <Arduino.h>

/**
 * Función que implementa un decodificador de repetición.
 * @param in Vector binario de entrada empaquetado en unsigned char (mensaje
codificado)
 * @param out Vector binario de salida empaquetado en unsigned char (mensaje
decodificado)
 * @param l Longitud del vector de entrada en bytes
 * @param n Grado del decodificador de repetición (número de veces que se
repitió cada bit)
 */
void repetitionDecoder(unsigned char *in, unsigned char *out, int l, int n)
{
  int inBitIndex = 0;  // Índice del bit actual en el vector de entrada
  int outBitIndex = 0; // Índice del bit actual en el vector de salida

  // Inicializar el vector de salida a ceros
  int outLength = (l * 8) / n; // Número total de bits en el mensaje
original
  int outBytes = (outLength + 7) / 8; // Número de bytes necesarios para
almacenar el mensaje original

  for (int i = 0; i < outBytes; i++) {
       out[i] = 0;
  }
```

```
  // Procesar cada grupo de n bits repetidos
  while (inBitIndex < l * 8) {
       int countOnes = 0; // Contador de unos en el grupo actual

       // Contar cuántos unos hay en el grupo de n bits
       for (int k = 0; k < n && inBitIndex < l * 8; k++) {
       // Calcular el índice del byte de entrada y la posición del bit
       int inByteIndex = inBitIndex / 8;
       int inBitPosition = 7 - (inBitIndex % 8);

       // Extraer el bit de la posición correspondiente
       unsigned char bit = (in[inByteIndex] >> inBitPosition) & 1;

       // Incrementar el contador si el bit es 1
       if (bit == 1) {
       countOnes++;
       }

       // Avanzar al siguiente bit de entrada
       inBitIndex++;
       }

       // Determinar el bit original mediante votación por mayoría
       unsigned char decodedBit = (countOnes > n / 2) ? 1 : 0;

       // Calcular el índice del byte de salida y la posición del bit
       int outByteIndex = outBitIndex / 8;
       int outBitPosition = 7 - (outBitIndex % 8);

       // Colocar el bit decodificado en la posición correspondiente del byte
de salida
       out[outByteIndex] |= (decodedBit << outBitPosition);

       // Avanzar al siguiente bit de salida
       outBitIndex++;
  }
}

// Función para imprimir un vector de bytes en formato binario
void printBinaryVector(unsigned char *vec, int length) {
  for (int i = 0; i < length; i++) {
       for (int j = 7; j >= 0; j--) {
       // Extraer y mostrar cada bit, empezando por el más significativo
       Serial.print((vec[i] >> j) & 1);
       }
       Serial.print(" ");
  }
  Serial.println();
```

```
}

void setup() {
  // Inicializar comunicación serial
  Serial.begin(9600);
  while (!Serial) {
       ; // Esperar a que el puerto serial se conecte
  }

  // Ejemplo de uso del decodificador de repetición
  const int n = 3; // Grado del codificador/decodificador (cada bit se
repite 3 veces)

  // Mensaje original para demostración
  const int originalLength = 2; // Longitud del mensaje original en bytes
  unsigned char original[originalLength] = {0b10101010, 0b11110000};

  // Calcular el tamaño necesario para el mensaje codificado
  int codedLength = (originalLength * 8 * n + 7) / 8; // Redondeo hacia
arriba
  unsigned char coded[codedLength];

  // Codificar el mensaje (simulando el uso del codificador de repetición)
  int outBitIndex = 0;
  for (int i = 0; i < originalLength; i++) {
       for (int j = 7; j >= 0; j--) {
       unsigned char bit = (original[i] >> j) & 1;
       for (int k = 0; k < n; k++) {
       int outByteIndex = outBitIndex / 8;
       int outBitPosition = 7 - (outBitIndex % 8);
       if (outBitPosition == 7) {
       coded[outByteIndex] = 0;
       }
       coded[outByteIndex] |= (bit << outBitPosition);
       outBitIndex++;
       }
       }
  }

  // Mensaje decodificado
  unsigned char decoded[originalLength];

  Serial.println("Mensaje original:");
  printBinaryVector(original, originalLength);

  Serial.println("Mensaje codificado:");
  printBinaryVector(coded, codedLength);
```

```
  // Aplicar el decodificador de repetición
  repetitionDecoder(coded, decoded, codedLength, n);

  Serial.println("Mensaje decodificado:");
  printBinaryVector(decoded, originalLength);

  // Mostrar información sobre la decodificación
  Serial.print("Grado de repetición: ");
  Serial.println(n);
  Serial.print("Bits del mensaje codificado: ");
  Serial.println(codedLength * 8);
  Serial.print("Bits del mensaje decodificado: ");
  Serial.println(originalLength * 8);
}

void loop() {
  // No se requiere código en el loop para esta práctica
}
```

Ejercicio 4.

Diseñar un codificador de bloque de código Hamming (7,4). A la función se le pasa como parámetros el mensaje a codificar en binario empaquetado en caracteres, la longitud l en bytes del mensaje y un
vector vacío suficientemente grande como para almacenar el resultado. Tenga en cuenta que cada byte de entrada tiene dos paquetes de 4 bit a codificar
La signatura de la función void hammingCoder(unsigned char *in, unsigned char *out, int l)

```
C/C++
/*
 * Práctica 2.4 - Codificador de bloque Hamming (7,4)
 *
 * Esta función implementa un codificador de bloque Hamming (7,4) que toma
bloques de 4 bits
 * y genera palabras código de 7 bits añadiendo 3 bits de paridad.
 * Cada byte de entrada contiene dos bloques de 4 bits que deben codificarse
por separado.
 */
#include <Arduino.h>
/*
 * Función que implementa un codificador de bloque Hamming (7,4).
 * @param in Vector binario de entrada empaquetado en unsigned char
 * @param out Vector binario de salida empaquetado en unsigned char
```

```
 * @param l Longitud del vector de entrada en bytes
 */
void hammingCoder(unsigned char *in, unsigned char *out, int l) {
  int outBitIndex = 0; // Índice del bit actual en el vector de salida

  // Recorrer cada byte del vector de entrada
  for (int i = 0; i < l; i++) {
       // Procesar cada bloque de 4 bits del byte actual (primero los 4 bits
más significativos)
       for (int nibbleIndex = 0; nibbleIndex < 2; nibbleIndex++) {
       // Extraer el bloque de 4 bits
       unsigned char nibble;
       if (nibbleIndex == 0) {
       // Extraer los 4 bits más significativos (posiciones 7-4)
       nibble = (in[i] >> 4) & 0x0F;
       } else {
       // Extraer los 4 bits menos significativos (posiciones 3-0)
       nibble = in[i] & 0x0F;
       }

       // Extraer los bits de datos individuales (d1, d2, d3, d4)
       unsigned char d1 = (nibble >> 3) & 1; // Bit más significativo
       unsigned char d2 = (nibble >> 2) & 1;
       unsigned char d3 = (nibble >> 1) & 1;
       unsigned char d4 = nibble & 1;          // Bit menos significativo

       // Calcular los bits de paridad según las ecuaciones de Hamming (7,4)
       unsigned char p1 = d1 ^ d2 ^ d4;        // p1 = d1 + d2 + d4 (XOR)
       unsigned char p2 = d1 ^ d3 ^ d4;        // p2 = d1 + d3 + d4 (XOR)
       unsigned char p3 = d2 ^ d3 ^ d4;        // p3 = d2 + d3 + d4 (XOR)

       // Formar la palabra código de 7 bits: p1, p2, d1, p3, d2, d3, d4
       unsigned char resultado[7] = {p1, p2, d1, p3, d2, d3, d4};

       // Colocar los 7 bits de la palabra código en el vector de salida
       for (int j = 0; j < 7; j++) {
       // Calcular el índice del byte de salida donde se colocará el bit
       int outByteIndex = outBitIndex / 8;
       // Calcular la posición del bit dentro del byte de salida
       int outBitPosition = 7 - (outBitIndex % 8);

       // Si estamos en un nuevo byte, inicializarlo a 0
       if (outBitPosition == 7) {
       out[outByteIndex] = 0;
       }

       // Colocar el bit en la posición correspondiente del byte de salida
       out[outByteIndex] |= (resultado[j] << outBitPosition);
```

```
        // Incrementar el índice del bit de salida
        outBitIndex++;
        }
        }
  }
}

// Función para imprimir un vector de bytes en formato binario
void printBinaryVector(unsigned char *vec, int length) {
  for (int i = 0; i < length; i++) {
        for (int j = 7; j >= 0; j--) {
        // Extraer y mostrar cada bit, empezando por el más significativo
        Serial.print((vec[i] >> j) & 1);
        }
        Serial.print(" ");
  }
  Serial.println();
}

void setup() {
  // Inicializar comunicación serial
  Serial.begin(9600);
  while (!Serial) {
        ; // Esperar a que el puerto serial se conecte
  }
  // Ejemplo de uso del codificador Hamming (7,4)
  const int length = 2; // Longitud del vector de entrada en bytes
  unsigned char input[length] = {0b10101010, 0b11110000};

  // Calcular el tamaño necesario para el vector de salida
  // Cada byte tiene 2 bloques de 4 bits, y cada bloque genera 7 bits
  int outputBits = length * 2 * 7; // Número total de bits en la salida
  int outputLength = (outputBits + 7) / 8; // Redondeo hacia arriba para
obtener bytes
  unsigned char output[outputLength];

  Serial.println("Vector de entrada:");
  printBinaryVector(input, length);

  // Aplicar el codificador Hamming (7,4)
  hammingCoder(input, output, length);

  Serial.println("Vector de salida (después del codificador Hamming):");
  printBinaryVector(output, outputLength);

  // Mostrar información detallada sobre la codificación
  Serial.println("\nPruebas de codificación Hamming (7,4):");
```

```
  Serial.print("Bytes de entrada: ");
  Serial.println(length);
  Serial.print("Bits de entrada: ");
  Serial.println(length * 8);
  Serial.print("Bytes de salida: ");
  Serial.println(outputLength);
  Serial.print("Bits de salida: ");
  Serial.println(outputLength * 8);
  Serial.print("Tasa de código: ");
  Serial.println("4/7");
  Serial.print("Bits de redundancia añadidos: ");
  Serial.println((outputLength * 8) - (length * 8));
  Serial.println("TENER EN CUENTA QUE SE REDONDEAN LOS BYTES");
}

void loop() {
  // No se requiere código en el loop para esta práctica
}
```

Ejercicio 5.

Diseñar un decodificador de bloque de código Hamming (7,4). A la función se le pasa como parámetros el mensaje a decodificar en binario empaquetado en caracteres, la longitud l en bytes del mensaje, n el grado del decodificador de repetición y un vector vacío suficientemente grande como para almacenar el resultado.
La signatura de la función void hammingDecoder(unsigned char *in, unsigned char *out, int l, int n);

```
C/C++
/*
 * Práctica 2.5 - Decodificador de bloque Hamming (7,4)
 *
 * Esta función implementa un decodificador de bloque Hamming (7,4) que toma
palabras código de 7 bits,
 * detecta y corrige errores de un solo bit, y recupera los bloques
originales de 4 bits.
 */
#include <Arduino.h>

/**
 * Función que implementa un decodificador de bloque Hamming (7,4).
 * @param in Vector binario de entrada empaquetado en unsigned char (mensaje
codificado)
```

```
 * @param out Vector binario de salida empaquetado en unsigned char (mensaje
decodificado)
 * @param l Longitud del vector de entrada en bytes
 * @param n Grado del decodificador de repetición (no se utiliza en este
caso, pero se mantiene por compatibilidad)
 */
void hammingDecoder(unsigned char *in, unsigned char *out, int l, int n) {
  int inBitIndex = 0;   // Índice del bit actual en el vector de entrada
  int outBitIndex = 0;  // Índice del bit actual en el vector de salida

  // Inicializar el vector de salida a ceros
  int outBytes = (l * 8 * 4) / 7; // Aproximación del número de bytes
necesarios
  outBytes = (outBytes + 7) / 8;  // Redondeo hacia arriba

  for (int i = 0; i < outBytes; i++) {
       out[i] = 0;
  }

  // Procesar cada palabra código de 7 bits
  while (inBitIndex + 6 < l * 8) { // Asegurarse de que hay al menos 7 bits
disponibles
       // Extraer los 7 bits de la palabra código
       unsigned char codeword[7];

       for (int j = 0; j < 7; j++) {
       // Calcular el índice del byte de entrada y la posición del bit
       int inByteIndex = inBitIndex / 8;
       int inBitPosition = 7 - (inBitIndex % 8);

       // Extraer el bit de la posición correspondiente
       codeword[j] = (in[inByteIndex] >> inBitPosition) & 1;

       // Avanzar al siguiente bit de entrada
       inBitIndex++;
       }

       // Extraer bits de paridad y datos de la palabra código
       unsigned char p1 = codeword[0]; // Bit de paridad 1
       unsigned char p2 = codeword[1]; // Bit de paridad 2
       unsigned char d1 = codeword[2]; // Bit de dato 1
       unsigned char p3 = codeword[3]; // Bit de paridad 3
       unsigned char d2 = codeword[4]; // Bit de dato 2
       unsigned char d3 = codeword[5]; // Bit de dato 3
       unsigned char d4 = codeword[6]; // Bit de dato 4

       // Calcular el síndrome para detectar errores
       unsigned char s1 = p1 ^ d1 ^ d2 ^ d4; // Verificar paridad 1
```

```
        unsigned char s2 = p2 ^ d1 ^ d3 ^ d4; // Verificar paridad 2
        unsigned char s3 = p3 ^ d2 ^ d3 ^ d4; // Verificar paridad 3

        // Determinar la posición del error (si existe)
        unsigned char errorPos = (s3 << 2) | (s2 << 1) | s1;

        // Corregir el error si se detecta (errorPos != 0)
        if (errorPos != 0) {
        // Posiciones de error y su correspondencia en el array codeword
        // 1 -> p1 (índice 0)
        // 2 -> p2 (índice 1)
        // 3 -> d1 (índice 2)
        // 4 -> p3 (índice 3)
        // 5 -> d2 (índice 4)
        // 6 -> d3 (índice 5)
        // 7 -> d4 (índice 6)

        // Corregir el bit erróneo
        if (errorPos >= 1 && errorPos <= 7) {
        // Invertir el bit en la posición del error
        codeword[errorPos - 1] = codeword[errorPos - 1] ^ 1;

        // Actualizar los bits de datos si fueron corregidos
        if (errorPos == 3) d1 = codeword[2];
        else if (errorPos == 5) d2 = codeword[4];
        else if (errorPos == 6) d3 = codeword[5];
        else if (errorPos == 7) d4 = codeword[6];
        }
        }

        // Reconstruir el nibble (4 bits) original
        unsigned char nibble = (d1 << 3) | (d2 << 2) | (d3 << 1) | d4;

        // Determinar si este nibble va en la parte alta o baja del byte de
salida
        if (outBitIndex % 8 == 0) {
        // Parte alta del byte (bits 7-4)
        out[outBitIndex / 8] |= (nibble << 4);
        } else {
        // Parte baja del byte (bits 3-0)
        out[outBitIndex / 8] |= nibble;
        }

        // Avanzar 4 bits en el índice de salida (un nibble)
        outBitIndex += 4;
    }
}
```

```
// Función para imprimir un vector de bytes en formato binario
void printBinaryVector(unsigned char *vec, int length) {
  for (int i = 0; i < length; i++) {
       for (int j = 7; j >= 0; j--) {
       // Extraer y mostrar cada bit, empezando por el más significativo
       Serial.print((vec[i] >> j) & 1);
       }
       Serial.print(" ");
  }
  Serial.println();
}

void setup() {
  // Inicializar comunicación serial
  Serial.begin(9600);
  while (!Serial) {
       ; // Esperar a que el puerto serial se conecte
  }

  // Ejemplo de uso del decodificador Hamming (7,4)
  const int length = 2; // Longitud del vector de entrada en bytes
  unsigned char original[length] = {0b10101010, 0b11110000};

  // Codificar el mensaje original usando Hamming (7,4)
  // Cada byte tiene 2 bloques de 4 bits, y cada bloque genera 7 bits
  int codedBits = length * 2 * 7; // Número total de bits en la salida
codificada
  int codedLength = (codedBits + 7) / 8; // Redondeo hacia arriba para
obtener bytes
  unsigned char coded[codedLength];

  // Aplicar el codificador Hamming (7,4) - simulación
  int outBitIndex = 0;
  for (int i = 0; i < length; i++) {
       // Procesar cada bloque de 4 bits del byte actual
       for (int nibbleIndex = 0; nibbleIndex < 2; nibbleIndex++) {
       // Extraer el bloque de 4 bits
       unsigned char nibble;
       if (nibbleIndex == 0) {
       // Extraer los 4 bits más significativos (posiciones 7-4)
       nibble = (original[i] >> 4) & 0x0F;
       } else {
       // Extraer los 4 bits menos significativos (posiciones 3-0)
       nibble = original[i] & 0x0F;
       }

       // Extraer los bits de datos individuales (d1, d2, d3, d4)
       unsigned char d1 = (nibble >> 3) & 1; // Bit más significativo
```

```
      unsigned char d2 = (nibble >> 2) & 1;
      unsigned char d3 = (nibble >> 1) & 1;
      unsigned char d4 = nibble & 1;          // Bit menos significativo

      // Calcular los bits de paridad según las ecuaciones de Hamming (7,4)
      unsigned char p1 = d1 ^ d2 ^ d4;        // p1 = d1 + d2 + d4 (XOR)
      unsigned char p2 = d1 ^ d3 ^ d4;        // p2 = d1 + d3 + d4 (XOR)
      unsigned char p3 = d2 ^ d3 ^ d4;        // p3 = d2 + d3 + d4 (XOR)

      // Formar la palabra código de 7 bits: p1, p2, d1, p3, d2, d3, d4
      unsigned char resultado[7] = {p1, p2, d1, p3, d2, d3, d4};

      // Colocar los 7 bits de la palabra código en el vector de salida
      for (int j = 0; j < 7; j++) {
      // Calcular el índice del byte de salida donde se colocará el bit
      int outByteIndex = outBitIndex / 8;
      // Calcular la posición del bit dentro del byte de salida
      int outBitPosition = 7 - (outBitIndex % 8);

      // Si estamos en un nuevo byte, inicializarlo a 0
      if (outBitPosition == 7) {
      coded[outByteIndex] = 0;
      }

      // Colocar el bit en la posición correspondiente del byte de salida
      coded[outByteIndex] |= (resultado[j] << outBitPosition);

      // Incrementar el índice del bit de salida
      outBitIndex++;
      }
      }
}

// Introducir un error en el mensaje codificado para probar la corrección
// Invertir el bit en la posición 10 (por ejemplo)
int errorBitIndex = 10;
int errorByteIndex = errorBitIndex / 8;
int errorBitPosition = 7 - (errorBitIndex % 8);
coded[errorByteIndex] ^= (1 << errorBitPosition);

// Decodificar el mensaje
unsigned char decoded[length];

Serial.println("Mensaje original:");
printBinaryVector(original, length);


Serial.println("Mensaje codificado con Hamming (7,4):");
```

```
  printBinaryVector(coded, codedLength);


  Serial.println("Mensaje codificado con un error introducido:");
  printBinaryVector(coded, codedLength);


  // Aplicar el decodificador Hamming (7,4)
  hammingDecoder(coded, decoded, codedLength, 0); // El parámetro n no se
utiliza

  Serial.println("Mensaje decodificado (después de corregir el error):");
  printBinaryVector(decoded, length);


  // Mostrar información sobre la decodificación
  Serial.println("\nPruebas de decodificación Hamming (7,4):");
  Serial.print("Bytes del mensaje codificado: ");
  Serial.println(codedLength);
  Serial.print("Bits del mensaje codificado: ");
  Serial.println(codedBits);
  Serial.print("Bytes del mensaje decodificado: ");
  Serial.println(length);
  Serial.print("Bits del mensaje decodificado: ");
  Serial.println(length * 8);
  Serial.print("Tasa de código: ");
  Serial.println("4/7");
  Serial.print("Capacidad de corrección: ");
  Serial.println("1 bit por palabra código");
}

void loop() {
}
```

Ejercicio 6.

Diseñe una clase NoisyChannel que tenga los métodos necesarios para enviar un paquete y recibir un paquete a través del canal. El constructor indicará el porcentaje de ruido aleatorio que introducirá la clase al mensaje.

C/C++

```
/*
 * Práctica 2.6 - Clase NoisyChannel
 *
```

```
 * Esta clase simula un canal ruidoso para comunicaciones binarias.
 * Permite enviar y recibir paquetes a través de un canal con ruido
aleatorio.
 */

#include <Arduino.h>

/**
 * Clase que simula un canal ruidoso para comunicaciones binarias.
 * Permite configurar el porcentaje de ruido que se introducirá en los
mensajes.
 */
class NoisyChannel {
private:
  float noisePercentage; // Porcentaje de ruido (entre 0 y 1)

  // Inicializa la semilla para la generación de números aleatorios
  void initRandomSeed() {
       randomSeed(analogRead(0));
  }

public:
  /**
   * Constructor de la clase NoisyChannel.
   * @param noise Porcentaje de ruido que se introducirá en los mensajes
(entre 0 y 1)
   */
  NoisyChannel(float noise) {
       // Asegurar que el porcentaje de ruido esté entre 0 y 1
       noisePercentage = constrain(noise, 0.0, 1.0);
       initRandomSeed();
  }

  /**
   * Envía un paquete a través del canal ruidoso.
   * @param input Vector binario de entrada empaquetado en unsigned char
   * @param output Vector binario de salida empaquetado en unsigned char
   * @param length Longitud del vector en bytes
   */
  void sendPacket(unsigned char *input, unsigned char *output, int length) {
       // Recorrer cada byte del vector de entrada
       for (int i = 0; i < length; i++) {
       // Inicializar el byte de salida con el valor del byte de entrada
       output[i] = input[i];

       // Procesar cada bit del byte actual
       for (int j = 0; j < 8; j++) {
       // Generar un número aleatorio entre 0 y 1
```

```
        float r = random(0, 100) / 100.0;

        // Si el número aleatorio es menor que la probabilidad de error,
        // invertir el bit correspondiente en el byte de salida
        if (r < noisePercentage) {
        // Invertir el bit j-ésimo usando XOR con una máscara
        output[i] ^= (1 << j);
        }
        }
        }
  }

  /**
   * Recibe un paquete a través del canal ruidoso.
   * Esta función es idéntica a sendPacket, ya que el canal es simétrico.
   * @param input Vector binario de entrada empaquetado en unsigned char
   * @param output Vector binario de salida empaquetado en unsigned char
   * @param length Longitud del vector en bytes
   */
  void receivePacket(unsigned char *input, unsigned char *output, int
length) {
        // En un canal simétrico, enviar y recibir son operaciones
equivalentes
        sendPacket(input, output, length);
  }

  /**
   * Obtiene el porcentaje de ruido configurado en el canal.
   * @return Porcentaje de ruido (entre 0 y 1)
   */
  float getNoisePercentage() {
        return noisePercentage;
  }

  /**
   * Establece un nuevo porcentaje de ruido para el canal.
   * @param noise Nuevo porcentaje de ruido (entre 0 y 1)
   */
  void setNoisePercentage(float noise) {
        noisePercentage = constrain(noise, 0.0, 1.0);
  }
};

// Función para imprimir un vector de bytes en formato binario
void printBinaryVector(unsigned char *vec, int length) {
  for (int i = 0; i < length; i++) {
        for (int j = 7; j >= 0; j--) {
        // Extraer y mostrar cada bit, empezando por el más significativo
```

```
      Serial.print((vec[i] >> j) & 1);
      }
      Serial.print(" ");
  }
  Serial.println();
}

void setup() {
  // Inicializar comunicación serial
  Serial.begin(9600);
  while (!Serial) {
      ; // Esperar a que el puerto serial se conecte
  }

  // Ejemplo de uso de la clase NoisyChannel
  const int length = 3; // Longitud del vector en bytes
  unsigned char input[length] = {0b10101010, 0b11110000, 0b00001111};
  unsigned char output[length];

  // Crear un canal con 20% de ruido
  NoisyChannel channel(0.2);

  Serial.println("Vector de entrada:");
  printBinaryVector(input, length);

  // Enviar el paquete a través del canal ruidoso
  channel.sendPacket(input, output, length);

  Serial.println("Vector de salida (después del canal ruidoso):");
  printBinaryVector(output, length);

  // Contar bits que han cambiado
  int changedBits = 0;
  for (int i = 0; i < length; i++) {
      unsigned char diff = input[i] ^ output[i]; // XOR para detectar bits
diferentes
      for (int j = 0; j < 8; j++) {
      if ((diff >> j) & 1) {
      changedBits++;
      }
      }
  }

  Serial.print("Bits cambiados: ");
  Serial.print(changedBits);
  Serial.print(" de ");
  Serial.print(length * 8);
  Serial.print(" (");
```

```
  Serial.print((float)changedBits / (length * 8) * 100);
  Serial.println("%)");

  // Ejemplo de cambio de porcentaje de ruido
  Serial.println("\nCambiando porcentaje de ruido a 50%");
  channel.setNoisePercentage(0.5);

  // Reiniciar el vector de entrada
  unsigned char input2[length] = {0b10101010, 0b11110000, 0b00001111};
  unsigned char output2[length];

  Serial.println("Vector de entrada:");
  printBinaryVector(input2, length);

  // Enviar el paquete a través del canal ruidoso con el nuevo porcentaje
  channel.sendPacket(input2, output2, length);

  Serial.println("Vector de salida (después del canal ruidoso):");
  printBinaryVector(output2, length);

  // Contar bits que han cambiado
  changedBits = 0;
  for (int i = 0; i < length; i++) {
       unsigned char diff = input2[i] ^ output2[i]; // XOR para detectar bits
diferentes
       for (int j = 0; j < 8; j++) {
       if ((diff >> j) & 1) {
       changedBits++;
       }
       }
  }

  Serial.print("Bits cambiados: ");
  Serial.print(changedBits);
  Serial.print(" de ");
  Serial.print(length * 8);
  Serial.print(" (");
  Serial.print((float)changedBits / (length * 8) * 100);
  Serial.println("%)");
}

void loop() {
  // No se requiere código en el loop para esta práctica
}
```

Ejercicio 7.

Diseñe una clase RepetitionCode que incluya los métodos usados para codificar y decodificar un mensaje utilizando este mecanismo de codificación.

```C/C++
/*
 * Práctica 2.7 - Clase RepetitionCode
 *
 * Esta clase implementa un codificador y decodificador de repetición de
grado Rn donde n es el número
 * de bits que se repiten en cada bloque. Encapsula la funcionalidad de
codificación y decodificación
 * por repetición en una clase reutilizable.
 */
#include <Arduino.h>

/**
 * Clase que implementa un codificador y decodificador de repetición.
 * Permite codificar mensajes repitiendo cada bit n veces y decodificarlos
 * mediante un sistema de votación por mayoría.
 */
class RepetitionCode {
private:
  int repetitionDegree; // Grado de repetición (número de veces que se
repite cada bit)

  /**
   * Método auxiliar para imprimir un vector de bytes en formato binario
   * @param vec Vector a imprimir
   * @param length Longitud del vector en bytes
   */
  void printBinaryVector(unsigned char *vec, int length) {
      for (int i = 0; i < length; i++) {
      for (int j = 7; j >= 0; j--) {
      // Extraer y mostrar cada bit, empezando por el más significativo
      Serial.print((vec[i] >> j) & 1);
      }
      Serial.print(" ");
      }
      Serial.println();
  }

public:
  /**
   * Constructor de la clase
   * @param n Grado de repetición (número de veces que se repite cada bit)
   */
```

```
  RepetitionCode(int n) {
       repetitionDegree = n;
  }

  /**
   * Método para establecer el grado de repetición
   * @param n Nuevo grado de repetición
   */
  void setRepetitionDegree(int n) {
       repetitionDegree = n;
  }

  /**
   * Método para obtener el grado de repetición actual
   * @return Grado de repetición
   */
  int getRepetitionDegree() {
       return repetitionDegree;
  }

  /**
   * Método para calcular la longitud del mensaje codificado en bytes
   * @param originalLength Longitud del mensaje original en bytes
   * @return Longitud del mensaje codificado en bytes
   */
  int getEncodedLength(int originalLength) {
       return (originalLength * 8 * repetitionDegree + 7) / 8; // Redondeo
hacia arriba
  }

  /**
   * Método para codificar un mensaje utilizando repetición
   * @param in Vector binario de entrada empaquetado en unsigned char
   * @param out Vector binario de salida empaquetado en unsigned char
   * @param length Longitud del vector de entrada en bytes
   */
  void encode(unsigned char *in, unsigned char *out, int length) {
       int outBitIndex = 0; // Índice del bit actual en el vector de salida

       // Recorrer cada byte del vector de entrada
       for (int i = 0; i < length; i++) {
       // Procesar cada bit del byte actual
       for (int j = 7; j >= 0; j--) {
       // Extraer el bit j-ésimo del byte i-ésimo
       unsigned char bit = (in[i] >> j) & 1;

       // Repetir el bit n veces
       for (int k = 0; k < repetitionDegree; k++) {
```

```
      // Calcular el índice del byte de salida donde se colocará el bit
repetido
      int outByteIndex = outBitIndex / 8;
      // Calcular la posición del bit dentro del byte de salida
      int outBitPosition = 7 - (outBitIndex % 8);

      // Si estamos en un nuevo byte, inicializarlo a 0
      if (outBitPosition == 7) {
            out[outByteIndex] = 0;
      }

      // Colocar el bit en la posición correspondiente del byte de salida
      out[outByteIndex] |= (bit << outBitPosition);

      // Incrementar el índice del bit de salida
      outBitIndex++;
      }
      }
      }
  }

  /**
   * Método para decodificar un mensaje codificado con repetición
   * @param in Vector binario de entrada empaquetado en unsigned char
(mensaje codificado)
   * @param out Vector binario de salida empaquetado en unsigned char
(mensaje decodificado)
   * @param length Longitud del vector de entrada en bytes
   */
  void decode(unsigned char *in, unsigned char *out, int length) {
      int inBitIndex = 0;  // Índice del bit actual en el vector de entrada
      int outBitIndex = 0; // Índice del bit actual en el vector de salida

      // Inicializar el vector de salida a ceros
      int outLength = (length * 8) / repetitionDegree; // Número total de
bits en el mensaje original
      int outBytes = (outLength + 7) / 8; // Número de bytes necesarios para
almacenar el mensaje original

      for (int i = 0; i < outBytes; i++) {
      out[i] = 0;
      }

      // Procesar cada grupo de n bits repetidos
      while (inBitIndex < length * 8) {
      int countOnes = 0; // Contador de unos en el grupo actual

      // Contar cuántos unos hay en el grupo de n bits
```

```
        for (int k = 0; k < repetitionDegree && inBitIndex < length * 8; k++)
{
        // Calcular el índice del byte de entrada y la posición del bit
        int inByteIndex = inBitIndex / 8;
        int inBitPosition = 7 - (inBitIndex % 8);

        // Extraer el bit de la posición correspondiente
        unsigned char bit = (in[inByteIndex] >> inBitPosition) & 1;

        // Incrementar el contador si el bit es 1
        if (bit == 1) {
        countOnes++;
        }

        // Avanzar al siguiente bit de entrada
        inBitIndex++;
        }

        // Determinar el bit original mediante votación por mayoría
        unsigned char decodedBit = (countOnes > repetitionDegree / 2) ? 1 : 0;

        // Calcular el índice del byte de salida y la posición del bit
        int outByteIndex = outBitIndex / 8;
        int outBitPosition = 7 - (outBitIndex % 8);

        // Colocar el bit decodificado en la posición correspondiente del byte
de salida
        out[outByteIndex] |= (decodedBit << outBitPosition);

        // Avanzar al siguiente bit de salida
        outBitIndex++;
        }
  }

  /**
   * Método para mostrar información sobre un mensaje y su versión
codificada
   * @param original Mensaje original
   * @param coded Mensaje codificado
   * @param originalLength Longitud del mensaje original en bytes
   * @param codedLength Longitud del mensaje codificado en bytes
   */
  void printInfo(unsigned char *original, unsigned char *coded, int
originalLength, int codedLength) {
        Serial.println("Mensaje original:");
        printBinaryVector(original, originalLength);

        Serial.println("Mensaje codificado:");
```

```
        printBinaryVector(coded, codedLength);

        Serial.print("Grado de repetición: ");
        Serial.println(repetitionDegree);
        Serial.print("Bits del mensaje original: ");
        Serial.println(originalLength * 8);
        Serial.print("Bits del mensaje codificado: ");
        Serial.println(codedLength * 8);
    }
};

void setup() {
    // Inicializar comunicación serial
    Serial.begin(9600);
    while (!Serial) {
        ; // Esperar a que el puerto serial se conecte
    }

    // Crear una instancia de RepetitionCode con grado 3
    RepetitionCode repCode(3);

    // Mensaje original para demostración
    const int originalLength = 2; // Longitud del mensaje original en bytes
    unsigned char original[originalLength] = {0b10101010, 0b11110000};

    // Calcular el tamaño necesario para el mensaje codificado
    int codedLength = repCode.getEncodedLength(originalLength);
    unsigned char coded[codedLength];

    // Codificar el mensaje
    repCode.encode(original, coded, originalLength);

    // Mensaje decodificado
    unsigned char decoded[originalLength];

    // Mostrar información sobre el mensaje original y codificado
    repCode.printInfo(original, coded, originalLength, codedLength);

    // Decodificar el mensaje
    repCode.decode(coded, decoded, codedLength);

    Serial.println("Mensaje decodificado:");
    for (int i = 0; i < originalLength; i++) {
        for (int j = 7; j >= 0; j--) {
        Serial.print((decoded[i] >> j) & 1);
        }
        Serial.print(" ");
    }
```

```
  Serial.println();

  // Verificar si la decodificación fue correcta
  bool correct = true;
  for (int i = 0; i < originalLength; i++) {
       if (original[i] != decoded[i]) {
       correct = false;
       break;
       }
  }

  Serial.print("Decodificación correcta: ");
  Serial.println(correct ? "Sí" : "No");

  // Ejemplo con otro grado de repetición
  Serial.println("\nCambiando el grado de repetición a 5:");
  repCode.setRepetitionDegree(5);

  // Recalcular el tamaño del mensaje codificado
  codedLength = repCode.getEncodedLength(originalLength);
  unsigned char coded2[codedLength];

  // Codificar y decodificar con el nuevo grado
  repCode.encode(original, coded2, originalLength);
  repCode.decode(coded2, decoded, codedLength);

  // Mostrar información
  repCode.printInfo(original, coded2, originalLength, codedLength);

  Serial.println("Mensaje decodificado:");
  for (int i = 0; i < originalLength; i++) {
       for (int j = 7; j >= 0; j--) {
       Serial.print((decoded[i] >> j) & 1);
       }
       Serial.print(" ");
  }
  Serial.println();
}

void loop() {
  // No se requiere código en el loop para esta práctica
}
```

Ejercicio 8.

Haga lo mismo para HammingCode

C/C++

```
/*
 * Práctica 2.8 - Clase HammingCode
 *
 * Esta clase implementa un codificador y decodificador Hamming (7,4) que
encapsula
 * la funcionalidad de codificación y decodificación Hamming en una clase
reutilizable.
 * Permite codificar bloques de 4 bits en palabras código de 7 bits y
decodificarlos
 * con capacidad de corrección de errores de un solo bit.
 */
#include <Arduino.h>

// Función auxiliar global para imprimir un vector de bytes en formato
binario
void printBinaryVector(unsigned char *vec, int length) {
  for (int i = 0; i < length; i++) {
       for (int j = 7; j >= 0; j--) {
       // Extraer y mostrar cada bit, empezando por el más significativo
       Serial.print((vec[i] >> j) & 1);
       }
       Serial.print(" ");
  }
  Serial.println();
}

/**
 * Clase que implementa un codificador y decodificador Hamming (7,4).
 * Permite codificar mensajes usando el código Hamming (7,4) y
decodificarlos
 * con capacidad de corrección de errores de un solo bit.
 */
class HammingCode {
public:
  /**
   * Constructor de la clase
   */
  HammingCode() {
       // No se requieren parámetros para Hamming (7,4)
  }

  /**
   * Método para calcular la longitud del mensaje codificado en bytes
   * @param originalLength Longitud del mensaje original en bytes
   * @return Longitud del mensaje codificado en bytes
   */
  int getEncodedLength(int originalLength) {
```

```
        // Cada byte tiene 2 bloques de 4 bits, y cada bloque genera 7 bits
        int codedBits = originalLength * 2 * 7; // Número total de bits en la
salida codificada
        return (codedBits + 7) / 8; // Redondeo hacia arriba para obtener
bytes
    }

    /**
     * Método para codificar un mensaje utilizando Hamming (7,4)
     * @param in Vector binario de entrada empaquetado en unsigned char
     * @param out Vector binario de salida empaquetado en unsigned char
     * @param length Longitud del vector de entrada en bytes
     */
    void encode(unsigned char *in, unsigned char *out, int length) {
        int outBitIndex = 0; // Índice del bit actual en el vector de salida

        // Recorrer cada byte del vector de entrada
        for (int i = 0; i < length; i++) {
        // Procesar cada bloque de 4 bits del byte actual (primero los 4 bits
más significativos)
        for (int nibbleIndex = 0; nibbleIndex < 2; nibbleIndex++) {
        // Extraer el bloque de 4 bits
        unsigned char nibble;
        if (nibbleIndex == 0) {
        // Extraer los 4 bits más significativos (posiciones 7-4)
        nibble = (in[i] >> 4) & 0x0F;
        } else {
        // Extraer los 4 bits menos significativos (posiciones 3-0)
        nibble = in[i] & 0x0F;
        }

        // Extraer los bits de datos individuales (d1, d2, d3, d4)
        unsigned char d1 = (nibble >> 3) & 1; // Bit más significativo
        unsigned char d2 = (nibble >> 2) & 1;
        unsigned char d3 = (nibble >> 1) & 1;
        unsigned char d4 = nibble & 1;          // Bit menos significativo

        // Calcular los bits de paridad según las ecuaciones de Hamming (7,4)
        unsigned char p1 = d1 ^ d2 ^ d4;        // p1 = d1 + d2 + d4 (XOR)
        unsigned char p2 = d1 ^ d3 ^ d4;        // p2 = d1 + d3 + d4 (XOR)
        unsigned char p3 = d2 ^ d3 ^ d4;        // p3 = d2 + d3 + d4 (XOR)

        // Formar la palabra código de 7 bits: p1, p2, d1, p3, d2, d3, d4
        unsigned char resultado[7] = {p1, p2, d1, p3, d2, d3, d4};

        // Colocar los 7 bits de la palabra código en el vector de salida
        for (int j = 0; j < 7; j++) {
        // Calcular el índice del byte de salida donde se colocará el bit
```

```
        int outByteIndex = outBitIndex / 8;
        // Calcular la posición del bit dentro del byte de salida
        int outBitPosition = 7 - (outBitIndex % 8);

        // Si estamos en un nuevo byte, inicializarlo a 0
        if (outBitPosition == 7) {
              out[outByteIndex] = 0;
        }

        // Colocar el bit en la posición correspondiente del byte de salida
        out[outByteIndex] |= (resultado[j] << outBitPosition);

        // Incrementar el índice del bit de salida
        outBitIndex++;
        }
        }
        }
  }

  /**
   * Método para decodificar un mensaje codificado con Hamming (7,4)
   * @param in Vector binario de entrada empaquetado en unsigned char
(mensaje codificado)
   * @param out Vector binario de salida empaquetado en unsigned char
(mensaje decodificado)
   * @param length Longitud del vector de entrada en bytes
   */
  void decode(unsigned char *in, unsigned char *out, int length) {
        int inBitIndex = 0;   // Índice del bit actual en el vector de entrada
        int outBitIndex = 0;  // Índice del bit actual en el vector de salida

        // Inicializar el vector de salida a ceros
        int outBytes = (length * 8 * 4) / 7; // Aproximación del número de
bytes necesarios
        outBytes = (outBytes + 7) / 8;  // Redondeo hacia arriba

        for (int i = 0; i < outBytes; i++) {
        out[i] = 0;
        }

        // Procesar cada palabra código de 7 bits
        while (inBitIndex + 6 < length * 8) { // Asegurarse de que hay al
menos 7 bits disponibles
        // Extraer los 7 bits de la palabra código
        unsigned char codeword[7];

        for (int j = 0; j < 7; j++) {
        // Calcular el índice del byte de entrada y la posición del bit
```

```
int inByteIndex = inBitIndex / 8;
int inBitPosition = 7 - (inBitIndex % 8);

// Extraer el bit de la posición correspondiente
codeword[j] = (in[inByteIndex] >> inBitPosition) & 1;

// Avanzar al siguiente bit de entrada
inBitIndex++;
}

// Extraer bits de paridad y datos de la palabra código
unsigned char p1 = codeword[0]; // Bit de paridad 1
unsigned char p2 = codeword[1]; // Bit de paridad 2
unsigned char d1 = codeword[2]; // Bit de dato 1
unsigned char p3 = codeword[3]; // Bit de paridad 3
unsigned char d2 = codeword[4]; // Bit de dato 2
unsigned char d3 = codeword[5]; // Bit de dato 3
unsigned char d4 = codeword[6]; // Bit de dato 4

// Calcular el síndrome para detectar errores
unsigned char s1 = p1 ^ d1 ^ d2 ^ d4; // Verificar paridad 1
unsigned char s2 = p2 ^ d1 ^ d3 ^ d4; // Verificar paridad 2
unsigned char s3 = p3 ^ d2 ^ d3 ^ d4; // Verificar paridad 3

// Determinar la posición del error (si existe)
unsigned char errorPos = (s3 << 2) | (s2 << 1) | s1;

// Corregir el error si se detecta (errorPos != 0)
if (errorPos != 0) {
// Posiciones de error y su correspondencia en el array codeword
// 1 -> p1 (índice 0)
// 2 -> p2 (índice 1)
// 3 -> d1 (índice 2)
// 4 -> p3 (índice 3)
// 5 -> d2 (índice 4)
// 6 -> d3 (índice 5)
// 7 -> d4 (índice 6)

// Corregir el bit erróneo
if (errorPos >= 1 && errorPos <= 7) {
// Invertir el bit en la posición del error
codeword[errorPos - 1] = codeword[errorPos - 1] ^ 1;

// Actualizar los bits de datos si fueron corregidos
if (errorPos == 3) d1 = codeword[2];
else if (errorPos == 5) d2 = codeword[4];
else if (errorPos == 6) d3 = codeword[5];
else if (errorPos == 7) d4 = codeword[6];
```

```
        }
        }

        // Reconstruir el nibble (4 bits) original
        unsigned char nibble = (d1 << 3) | (d2 << 2) | (d3 << 1) | d4;

        // Determinar si este nibble va en la parte alta o baja del byte de
salida
        if (outBitIndex % 8 == 0) {
        // Parte alta del byte (bits 7-4)
        out[outBitIndex / 8] |= (nibble << 4);
        } else {
        // Parte baja del byte (bits 3-0)
        out[outBitIndex / 8] |= nibble;
        }

        // Avanzar 4 bits en el índice de salida (un nibble)
        outBitIndex += 4;
        }
  }

  /**
   * Método para mostrar información sobre un mensaje y su versión
codificada
   * @param original Mensaje original
   * @param coded Mensaje codificado
   * @param originalLength Longitud del mensaje original en bytes
   * @param codedLength Longitud del mensaje codificado en bytes
   */
  void printInfo(unsigned char *original, unsigned char *coded, int
originalLength, int codedLength) {
        Serial.println("Mensaje original:");
        ::printBinaryVector(original, originalLength);

        Serial.println("Mensaje codificado con Hamming (7,4):");
        ::printBinaryVector(coded, codedLength);

        Serial.print("Bits del mensaje original: ");
        Serial.println(originalLength * 8);
        Serial.print("Bits del mensaje codificado: ");
        Serial.println(codedLength * 8);
        Serial.print("Tasa de código: ");
        Serial.println("4/7");
  }
};

void setup() {
  // Inicializar comunicación serial
```

```
Serial.begin(9600);
while (!Serial) {
     ; // Esperar a que el puerto serial se conecte
}

// Crear una instancia de HammingCode
HammingCode hammingCode;

// Mensaje original para demostración
const int originalLength = 2; // Longitud del mensaje original en bytes
unsigned char original[originalLength] = {0b10101010, 0b11110000};

// Calcular el tamaño necesario para el mensaje codificado
int codedLength = hammingCode.getEncodedLength(originalLength);
unsigned char coded[codedLength];

// Codificar el mensaje
hammingCode.encode(original, coded, originalLength);

// Mensaje decodificado
unsigned char decoded[originalLength];

// Mostrar información sobre el mensaje original y codificado
hammingCode.printInfo(original, coded, originalLength, codedLength);

// Introducir un error en el mensaje codificado para probar la corrección
// Invertir el bit en la posición 10 (por ejemplo)
int errorBitIndex = 10;
int errorByteIndex = errorBitIndex / 8;
int errorBitPosition = 7 - (errorBitIndex % 8);
coded[errorByteIndex] ^= (1 << errorBitPosition);

Serial.println("\nMensaje codificado con un error introducido:");
printBinaryVector(coded, codedLength);

// Decodificar el mensaje
hammingCode.decode(coded, decoded, codedLength);

Serial.println("Mensaje decodificado (después de corregir el error):");
for (int i = 0; i < originalLength; i++) {
     for (int j = 7; j >= 0; j--) {
     Serial.print((decoded[i] >> j) & 1);
     }
     Serial.print(" ");
}
Serial.println();

// Verificar si la decodificación fue correcta
```

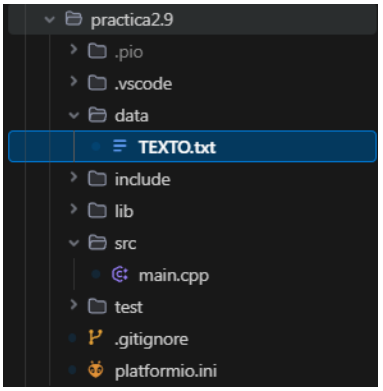
```
  bool correct = true;
  for (int i = 0; i < originalLength; i++) {
       if (original[i] != decoded[i]) {
       correct = false;
       break;
       }
  }

  Serial.print("Decodificación correcta: ");
  Serial.println(correct ? "Sí" : "No");
}

void loop() {
  // No se requiere código en el loop para esta práctica
}
```

Ejercicio 9.

Analice las ventajas e inconvenientes de cada uno de los dos sistemas de codificación por bloques que hemos analizado hasta ahora. Tome fichero de texto plano no comprimido y hágalo pasar por el sistema codificador – canalRuidoso – decodificador de bloque con una probabilidad de error de 0.05. Utilice una codificación de Hamming y una codificación R4 y compare los resultados obtenidos entre ambos sistemas de codificación. https://es.wikipedia.org/wiki/Windows_bitmap en función de la tasa de transmisión conseguida versus el porcentaje de error de la imagen resultado.



En este ejercicio tendremos que añadir un archivo de texto plano en la estructura del proyecto tal que así:

```
C/C++
/*
 * Práctica 2.9 - Comparación de códigos de repetición y Hamming
 *
 * Este programa compara el rendimiento de los códigos de repetición y
Hamming
 * al transmitir datos a través de un canal ruidoso con probabilidad de
error 0.05.
 * Analiza las ventajas e inconvenientes en términos de tasa de transmisión
y
 * porcentaje de error para cada sistema de codificación.
```

```
 */

#include <Arduino.h>
#include <string.h>

// Probabilidad de error del canal ruidoso
const float ERROR_PROBABILITY = 0.05;

/**
 * Función que simula un canal ruidoso.
 * @param in Vector binario de entrada empaquetado en unsigned char
 * @param out Vector binario de salida empaquetado en unsigned char
 * @param l Longitud del vector en bytes
 * @param f Probabilidad de que un bit cambie de valor (entre 0 y 1)
 */
void noisyChannel(unsigned char *in, unsigned char *out, int l, float f) {
  // Inicializar la semilla para la generación de números aleatorios
  randomSeed(analogRead(0));

  // Recorrer cada byte del vector de entrada
  for (int i = 0; i < l; i++) {
       // Inicializar el byte de salida con el valor del byte de entrada
       out[i] = in[i];

       // Procesar cada bit del byte actual
       for (int j = 0; j < 8; j++) {
       // Generar un número aleatorio entre 0 y 1
       float r = random(0, 100) / 100.0;

       // Si el número aleatorio es menor que la probabilidad de error f,
       // invertir el bit correspondiente en el byte de salida
       if (r < f) {
       // Invertir el bit j-ésimo usando XOR con una máscara
       out[i] ^= (1 << j);
       }
       }
  }
}

// Función para imprimir un vector de bytes en formato binario
void printBinaryVector(unsigned char *vec, int length) {
  for (int i = 0; i < length; i++) {
       for (int j = 7; j >= 0; j--) {
       // Extraer y mostrar cada bit, empezando por el más significativo
       Serial.print((vec[i] >> j) & 1);
       }
       Serial.print(" ");
  }
```

```
  Serial.println();
}

/**
 * Clase que implementa un codificador y decodificador de repetición.
 * Permite codificar mensajes repitiendo cada bit n veces y decodificarlos
 * mediante un sistema de votación por mayoría.
 */
class RepetitionCode {
private:
  int repetitionDegree; // Grado de repetición (número de veces que se
repite cada bit)

public:
  /**
   * Constructor de la clase
   * @param n Grado de repetición (número de veces que se repite cada bit)
   */
  RepetitionCode(int n) {
       repetitionDegree = n;
  }

  /**
   * Método para establecer el grado de repetición
   * @param n Nuevo grado de repetición
   */
  void setRepetitionDegree(int n) {
       repetitionDegree = n;
  }

  /**
   * Método para obtener el grado de repetición actual
   * @return Grado de repetición
   */
  int getRepetitionDegree() {
       return repetitionDegree;
  }

  /**
   * Método para calcular la longitud del mensaje codificado en bytes
   * @param originalLength Longitud del mensaje original en bytes
   * @return Longitud del mensaje codificado en bytes
   */
  int getEncodedLength(int originalLength) {
       return (originalLength * 8 * repetitionDegree + 7) / 8; // Redondeo
hacia arriba
  }
```

```
  /**
   * Método para codificar un mensaje utilizando repetición
   * @param in Vector binario de entrada empaquetado en unsigned char
   * @param out Vector binario de salida empaquetado en unsigned char
   * @param length Longitud del vector de entrada en bytes
   */
  void encode(unsigned char *in, unsigned char *out, int length) {
      int outBitIndex = 0; // Índice del bit actual en el vector de salida

      // Recorrer cada byte del vector de entrada
      for (int i = 0; i < length; i++) {
      // Procesar cada bit del byte actual
      for (int j = 7; j >= 0; j--) {
      // Extraer el bit j-ésimo del byte i-ésimo
      unsigned char bit = (in[i] >> j) & 1;

      // Repetir el bit n veces
      for (int k = 0; k < repetitionDegree; k++) {
      // Calcular el índice del byte de salida donde se colocará el bit
repetido
      int outByteIndex = outBitIndex / 8;
      // Calcular la posición del bit dentro del byte de salida
      int outBitPosition = 7 - (outBitIndex % 8);

      // Si estamos en un nuevo byte, inicializarlo a 0
      if (outBitPosition == 7) {
            out[outByteIndex] = 0;
      }

      // Colocar el bit en la posición correspondiente del byte de salida
      out[outByteIndex] |= (bit << outBitPosition);

      // Incrementar el índice del bit de salida
      outBitIndex++;
      }
      }
      }
  }

  /**
   * Método para decodificar un mensaje codificado con repetición
   * @param in Vector binario de entrada empaquetado en unsigned char
(mensaje codificado)
   * @param out Vector binario de salida empaquetado en unsigned char
(mensaje decodificado)
   * @param length Longitud del vector de entrada en bytes
   */
  void decode(unsigned char *in, unsigned char *out, int length) {
```

```
        int inBitIndex = 0;  // Índice del bit actual en el vector de entrada
        int outBitIndex = 0; // Índice del bit actual en el vector de salida

        // Inicializar el vector de salida a ceros
        int outLength = (length * 8) / repetitionDegree; // Número total de
bits en el mensaje original
        int outBytes = (outLength + 7) / 8; // Número de bytes necesarios para
almacenar el mensaje original

        for (int i = 0; i < outBytes; i++) {
        out[i] = 0;
        }

        // Procesar cada grupo de n bits repetidos
        while (inBitIndex < length * 8) {
        int countOnes = 0; // Contador de unos en el grupo actual

        // Contar cuántos unos hay en el grupo de n bits
        for (int k = 0; k < repetitionDegree && inBitIndex < length * 8; k++)
{
        // Calcular el índice del byte de entrada y la posición del bit
        int inByteIndex = inBitIndex / 8;
        int inBitPosition = 7 - (inBitIndex % 8);

        // Extraer el bit de la posición correspondiente
        unsigned char bit = (in[inByteIndex] >> inBitPosition) & 1;

        // Incrementar el contador si el bit es 1
        if (bit == 1) {
        countOnes++;
        }

        // Avanzar al siguiente bit de entrada
        inBitIndex++;
        }

        // Determinar el bit original mediante votación por mayoría
        unsigned char decodedBit = (countOnes > repetitionDegree / 2) ? 1 : 0;

        // Calcular el índice del byte de salida y la posición del bit
        int outByteIndex = outBitIndex / 8;
        int outBitPosition = 7 - (outBitIndex % 8);

        // Colocar el bit decodificado en la posición correspondiente del byte
de salida
        out[outByteIndex] |= (decodedBit << outBitPosition);

        // Avanzar al siguiente bit de salida
```

```
      outBitIndex++;
      }
  }
};

/**
 * Clase que implementa un codificador y decodificador Hamming (7,4).
 * Permite codificar mensajes usando el código Hamming (7,4) y
decodificarlos
 * con capacidad de corrección de errores de un solo bit.
 */
class HammingCode {
public:
  /**
   * Constructor de la clase
   */
  HammingCode() {
      // No se requieren parámetros para Hamming (7,4)
  }

  /**
   * Método para calcular la longitud del mensaje codificado en bytes
   * @param originalLength Longitud del mensaje original en bytes
   * @return Longitud del mensaje codificado en bytes
   */
  int getEncodedLength(int originalLength) {
      // Cada byte tiene 2 bloques de 4 bits, y cada bloque genera 7 bits
      int codedBits = originalLength * 2 * 7; // Número total de bits en la
salida codificada
      return (codedBits + 7) / 8; // Redondeo hacia arriba para obtener
bytes
  }

  /**
   * Método para codificar un mensaje utilizando Hamming (7,4)
   * @param in Vector binario de entrada empaquetado en unsigned char
   * @param out Vector binario de salida empaquetado en unsigned char
   * @param length Longitud del vector de entrada en bytes
   */
  void encode(unsigned char *in, unsigned char *out, int length) {
      int outBitIndex = 0; // Índice del bit actual en el vector de salida

      // Recorrer cada byte del vector de entrada
      for (int i = 0; i < length; i++) {
      // Procesar cada bloque de 4 bits del byte actual (primero los 4 bits
más significativos)
      for (int nibbleIndex = 0; nibbleIndex < 2; nibbleIndex++) {
      // Extraer el bloque de 4 bits
```

```
        unsigned char nibble;
        if (nibbleIndex == 0) {
        // Extraer los 4 bits más significativos (posiciones 7-4)
        nibble = (in[i] >> 4) & 0x0F;
        } else {
        // Extraer los 4 bits menos significativos (posiciones 3-0)
        nibble = in[i] & 0x0F;
        }

        // Extraer los bits de datos individuales (d1, d2, d3, d4)
        unsigned char d1 = (nibble >> 3) & 1; // Bit más significativo
        unsigned char d2 = (nibble >> 2) & 1;
        unsigned char d3 = (nibble >> 1) & 1;
        unsigned char d4 = nibble & 1;          // Bit menos significativo

        // Calcular los bits de paridad según las ecuaciones de Hamming (7,4)
        unsigned char p1 = d1 ^ d2 ^ d4;        // p1 = d1 + d2 + d4 (XOR)
        unsigned char p2 = d1 ^ d3 ^ d4;        // p2 = d1 + d3 + d4 (XOR)
        unsigned char p3 = d2 ^ d3 ^ d4;        // p3 = d2 + d3 + d4 (XOR)

        // Formar la palabra código de 7 bits: p1, p2, d1, p3, d2, d3, d4
        unsigned char resultado[7] = {p1, p2, d1, p3, d2, d3, d4};

        // Colocar los 7 bits de la palabra código en el vector de salida
        for (int j = 0; j < 7; j++) {
        // Calcular el índice del byte de salida donde se colocará el bit
        int outByteIndex = outBitIndex / 8;
        // Calcular la posición del bit dentro del byte de salida
        int outBitPosition = 7 - (outBitIndex % 8);

        // Si estamos en un nuevo byte, inicializarlo a 0
        if (outBitPosition == 7) {
              out[outByteIndex] = 0;
        }

        // Colocar el bit en la posición correspondiente del byte de salida
        out[outByteIndex] |= (resultado[j] << outBitPosition);

        // Incrementar el índice del bit de salida
        outBitIndex++;
        }
        }
        }
}

/**
 * Método para decodificar un mensaje codificado con Hamming (7,4)
```

```
   * @param in Vector binario de entrada empaquetado en unsigned char
(mensaje codificado)
   * @param out Vector binario de salida empaquetado en unsigned char
(mensaje decodificado)
   * @param length Longitud del vector de entrada en bytes
   */
  void decode(unsigned char *in, unsigned char *out, int length) {
       int inBitIndex = 0;   // Índice del bit actual en el vector de entrada
       int outBitIndex = 0;  // Índice del bit actual en el vector de salida

       // Inicializar el vector de salida a ceros
       int outBytes = (length * 8 * 4) / 7; // Aproximación del número de
bytes necesarios
       outBytes = (outBytes + 7) / 8;  // Redondeo hacia arriba

       for (int i = 0; i < outBytes; i++) {
       out[i] = 0;
       }

       // Procesar cada palabra código de 7 bits
       while (inBitIndex + 6 < length * 8) { // Asegurarse de que hay al
menos 7 bits disponibles
       // Extraer los 7 bits de la palabra código
       unsigned char codeword[7];

       for (int j = 0; j < 7; j++) {
       // Calcular el índice del byte de entrada y la posición del bit
       int inByteIndex = inBitIndex / 8;
       int inBitPosition = 7 - (inBitIndex % 8);

       // Extraer el bit de la posición correspondiente
       codeword[j] = (in[inByteIndex] >> inBitPosition) & 1;

       // Avanzar al siguiente bit de entrada
       inBitIndex++;
       }

       // Extraer bits de paridad y datos de la palabra código
       unsigned char p1 = codeword[0]; // Bit de paridad 1
       unsigned char p2 = codeword[1]; // Bit de paridad 2
       unsigned char d1 = codeword[2]; // Bit de dato 1
       unsigned char p3 = codeword[3]; // Bit de paridad 3
       unsigned char d2 = codeword[4]; // Bit de dato 2
       unsigned char d3 = codeword[5]; // Bit de dato 3
       unsigned char d4 = codeword[6]; // Bit de dato 4

       // Calcular el síndrome para detectar errores
       unsigned char s1 = p1 ^ d1 ^ d2 ^ d4; // Verificar paridad 1
```

```
        unsigned char s2 = p2 ^ d1 ^ d3 ^ d4; // Verificar paridad 2
        unsigned char s3 = p3 ^ d2 ^ d3 ^ d4; // Verificar paridad 3

        // Determinar la posición del error (si existe)
        unsigned char errorPos = (s3 << 2) | (s2 << 1) | s1;

        // Corregir el error si se detecta (errorPos != 0)
        if (errorPos != 0) {
        // Posiciones de error y su correspondencia en el array codeword
        // 1 -> p1 (índice 0)
        // 2 -> p2 (índice 1)
        // 3 -> d1 (índice 2)
        // 4 -> p3 (índice 3)
        // 5 -> d2 (índice 4)
        // 6 -> d3 (índice 5)
        // 7 -> d4 (índice 6)

        // Corregir el bit erróneo
        if (errorPos >= 1 && errorPos <= 7) {
        // Invertir el bit en la posición del error
        codeword[errorPos - 1] = codeword[errorPos - 1] ^ 1;

        // Actualizar los bits de datos si fueron corregidos
        if (errorPos == 3) d1 = codeword[2];
        else if (errorPos == 5) d2 = codeword[4];
        else if (errorPos == 6) d3 = codeword[5];
        else if (errorPos == 7) d4 = codeword[6];
        }
        }

        // Reconstruir el nibble (4 bits) original
        unsigned char nibble = (d1 << 3) | (d2 << 2) | (d3 << 1) | d4;

        // Determinar si este nibble va en la parte alta o baja del byte de
salida
        if (outBitIndex % 8 == 0) {
        // Parte alta del byte (bits 7-4)
        out[outBitIndex / 8] |= (nibble << 4);
        } else {
        // Parte baja del byte (bits 3-0)
        out[outBitIndex / 8] |= nibble;
        }

        // Avanzar 4 bits en el índice de salida (un nibble)
        outBitIndex += 4;
        }
    }
};
```

```
/**
 * Función para contar bits diferentes entre dos vectores
 * @param vec1 Primer vector
 * @param vec2 Segundo vector
 * @param length Longitud de los vectores en bytes
 * @return Número de bits diferentes
 */
int countDifferentBits(unsigned char *vec1, unsigned char *vec2, int length)
{
  int count = 0;
  for (int i = 0; i < length; i++) {
       unsigned char diff = vec1[i] ^ vec2[i]; // XOR para detectar bits
diferentes
       for (int j = 0; j < 8; j++) {
       if ((diff >> j) & 1) {
       count++;
       }
       }
  }
  return count;
}

/**
 * Función para leer un archivo de texto y convertirlo en un vector de bytes
 * @param filename Nombre del archivo
 * @param buffer Buffer donde se almacenarán los datos
 * @param maxLength Tamaño máximo del buffer
 * @return Número de bytes leídos
 */
int readTextFile(const char *filename, unsigned char *buffer, int maxLength)
{
  // En Arduino, simulamos la lectura del archivo con un texto predefinido
  const char *text = "The quick brown fox jumps over the lazy dog";
  int length = strlen(text);

  // Asegurarse de no exceder el tamaño del buffer
  if (length > maxLength) {
       length = maxLength;
  }

  // Copiar el texto al buffer
  memcpy(buffer, text, length);

  return length;
}

void setup() {
```

```
// Inicializar comunicación serial
Serial.begin(9600);
while (!Serial) {
     ; // Esperar a que el puerto serial se conecte
}

Serial.println("Comparación de códigos de repetición y Hamming");
Serial.println("============================================\n");

// Leer el archivo de texto
const int MAX_LENGTH = 100;
unsigned char originalData[MAX_LENGTH];
int dataLength = readTextFile("TEXTO.txt", originalData, MAX_LENGTH);

Serial.print("Texto original: ");
for (int i = 0; i < dataLength; i++) {
     Serial.print((char)originalData[i]);
}
Serial.println();
Serial.print("Longitud: ");
Serial.print(dataLength);
Serial.print(" bytes (");
Serial.print(dataLength * 8);
Serial.println(" bits)");
Serial.println();

// Crear instancias de los codificadores
RepetitionCode repCode(3); // Código de repetición con grado 3
HammingCode hammingCode;   // Código Hamming (7,4)

// Calcular tamaños de los mensajes codificados
int repCodedLength = repCode.getEncodedLength(dataLength);
int hammingCodedLength = hammingCode.getEncodedLength(dataLength);

// Crear buffers para los mensajes codificados
unsigned char repCoded[repCodedLength];
unsigned char hammingCoded[hammingCodedLength];

// Codificar los mensajes
repCode.encode(originalData, repCoded, dataLength);
hammingCode.encode(originalData, hammingCoded, dataLength);

// Crear buffers para los mensajes después del canal ruidoso
unsigned char repNoisyChannel[repCodedLength];
unsigned char hammingNoisyChannel[hammingCodedLength];

// Simular el canal ruidoso
```

```
  noisyChannel(repCoded, repNoisyChannel, repCodedLength,
ERROR_PROBABILITY);
  noisyChannel(hammingCoded, hammingNoisyChannel, hammingCodedLength,
ERROR_PROBABILITY);

  // Crear buffers para los mensajes decodificados
  unsigned char repDecoded[dataLength];
  unsigned char hammingDecoded[dataLength];

  // Decodificar los mensajes
  repCode.decode(repNoisyChannel, repDecoded, repCodedLength);
  hammingCode.decode(hammingNoisyChannel, hammingDecoded,
hammingCodedLength);

  // Calcular estadísticas para el código de repetición
  int repBitsTransmitted = repCodedLength * 8;
  int repBitsOriginal = dataLength * 8;
  float repRate = (float)repBitsOriginal / repBitsTransmitted;
  int repErrorsChannel = countDifferentBits(repCoded, repNoisyChannel,
repCodedLength);
  int repErrorsFinal = countDifferentBits(originalData, repDecoded,
dataLength);
  float repErrorPercentChannel = (float)repErrorsChannel /
repBitsTransmitted * 100;
  float repErrorPercentFinal = (float)repErrorsFinal / repBitsOriginal *
100;

  // Calcular estadísticas para el código Hamming
  int hammingBitsTransmitted = hammingCodedLength * 8;
  int hammingBitsOriginal = dataLength * 8;
  float hammingRate = (float)hammingBitsOriginal / hammingBitsTransmitted;
  int hammingErrorsChannel = countDifferentBits(hammingCoded,
hammingNoisyChannel, hammingCodedLength);
  int hammingErrorsFinal = countDifferentBits(originalData, hammingDecoded,
dataLength);
  float hammingErrorPercentChannel = (float)hammingErrorsChannel /
hammingBitsTransmitted * 100;
  float hammingErrorPercentFinal = (float)hammingErrorsFinal /
hammingBitsOriginal * 100;

  // Mostrar resultados para el código de repetición
  Serial.println("CÓDIGO DE REPETICIÓN (R3)");
  Serial.println("-------------------------");
  Serial.print("Tasa de código: ");
  Serial.println(repRate);
  Serial.print("Bits transmitidos: ");
  Serial.println(repBitsTransmitted);
  Serial.print("Errores en el canal: ");
```

```
Serial.print(repErrorsChannel);
Serial.print(" (");
Serial.print(repErrorPercentChannel);
Serial.println("%)");
Serial.print("Errores después de decodificar: ");
Serial.print(repErrorsFinal);
Serial.print(" (");
Serial.print(repErrorPercentFinal);
Serial.println("%)");
Serial.println();

// Mostrar resultados para el código Hamming
Serial.println("CÓDIGO HAMMING (7,4)");
Serial.println("--------------------");
Serial.print("Tasa de código: ");
Serial.println(hammingRate);
Serial.print("Bits transmitidos: ");
Serial.println(hammingBitsTransmitted);
Serial.print("Errores en el canal: ");
Serial.print(hammingErrorsChannel);
Serial.print(" (");
Serial.print(hammingErrorPercentChannel);
Serial.println("%)");
Serial.print("Errores después de decodificar: ");
Serial.print(hammingErrorsFinal);
Serial.print(" (");
Serial.print(hammingErrorPercentFinal);
Serial.println("%)");
Serial.println();

// Comparación de resultados
Serial.println("COMPARACIÓN DE RESULTADOS");
Serial.println("------------------------");
Serial.println("Ventajas del código de repetición:");
Serial.println("- Implementación más simple");
if (repErrorPercentFinal < hammingErrorPercentFinal) {
    Serial.println("- Menor tasa de error final");
}

Serial.println("\nVentajas del código Hamming:");
Serial.println("- Mayor eficiencia en la tasa de código");
if (hammingErrorPercentFinal < repErrorPercentFinal) {
    Serial.println("- Menor tasa de error final");
}

Serial.println("\nConclusiones:");
if (hammingRate > repRate) {
```

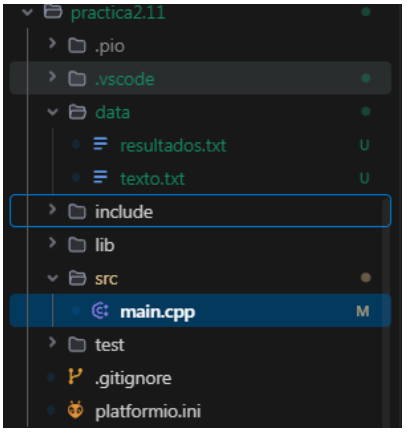
```
      Serial.println("- Hamming es más eficiente en términos de tasa de
transmisión");
  } else {
      Serial.println("- Repetición es más eficiente en términos de tasa de
transmisión");
  }

  if (hammingErrorPercentFinal < repErrorPercentFinal) {
      Serial.println("- Hamming tiene mejor capacidad de corrección de
errores");
  } else if (repErrorPercentFinal < hammingErrorPercentFinal) {
      Serial.println("- Repetición tiene mejor capacidad de corrección de
errores");
  } else {
      Serial.println("- Ambos códigos tienen similar capacidad de corrección
de errores");
  }
}

void loop() {
  // No se requiere código en el loop para esta práctica
}
```

Ejercicio 10.

Una forma de mejorar los sistemas de codificación es serializar dos códigos diferentes para minimizar la posibilidad de error. Diseñe una clase que aproveche las dos clases previas RepetitionCode y HammingCode para crear un nuevo código llamado HammingRepetition.
Para esto también tenemos que crear dos archivos en la estructura del programa:



C/C++

```
/*
 * Práctica 2.10 - Clase HammingRepetition
 *
 * Esta clase implementa un codificador y decodificador que combina los
códigos Hamming y Repetición
 * en serie. Primero aplica el código Hamming (7,4) y luego el código de
repetición de grado Rn.
 * Esta combinación aprovecha la capacidad de corrección de errores de un
bit de Hamming junto con
```

```
 * la robustez adicional que proporciona la repetición.
 */
#include <Arduino.h>

// Función auxiliar global para imprimir un vector de bytes en formato
binario
void printBinaryVector(unsigned char *vec, int length) {
  for (int i = 0; i < length; i++) {
       for (int j = 7; j >= 0; j--) {
       // Extraer y mostrar cada bit, empezando por el más significativo
       Serial.print((vec[i] >> j) & 1);
       }
       Serial.print(" ");
  }
  Serial.println();
}

/**
 * Clase que implementa un codificador y decodificador Hamming (7,4).
 * Permite codificar mensajes usando el código Hamming (7,4) y
decodificarlos
 * con capacidad de corrección de errores de un solo bit.
 */
class HammingCode {
public:
  /**
   * Constructor de la clase
   */
  HammingCode() {
       // No se requieren parámetros para Hamming (7,4)
  }

  /**
   * Método para calcular la longitud del mensaje codificado en bytes
   * @param originalLength Longitud del mensaje original en bytes
   * @return Longitud del mensaje codificado en bytes
   */
  int getEncodedLength(int originalLength) {
       // Cada byte tiene 2 bloques de 4 bits, y cada bloque genera 7 bits
       int codedBits = originalLength * 2 * 7; // Número total de bits en la
salida codificada
       return (codedBits + 7) / 8; // Redondeo hacia arriba para obtener
bytes
  }

  /**
   * Método para codificar un mensaje utilizando Hamming (7,4)
   * @param in Vector binario de entrada empaquetado en unsigned char
```

```
   * @param out Vector binario de salida empaquetado en unsigned char
   * @param length Longitud del vector de entrada en bytes
   */
  void encode(unsigned char *in, unsigned char *out, int length) {
      int outBitIndex = 0; // Índice del bit actual en el vector de salida

      // Recorrer cada byte del vector de entrada
      for (int i = 0; i < length; i++) {
      // Procesar cada bloque de 4 bits del byte actual (primero los 4 bits
más significativos)
      for (int nibbleIndex = 0; nibbleIndex < 2; nibbleIndex++) {
      // Extraer el bloque de 4 bits
      unsigned char nibble;
      if (nibbleIndex == 0) {
      // Extraer los 4 bits más significativos (posiciones 7-4)
      nibble = (in[i] >> 4) & 0x0F;
      } else {
      // Extraer los 4 bits menos significativos (posiciones 3-0)
      nibble = in[i] & 0x0F;
      }

      // Extraer los bits de datos individuales (d1, d2, d3, d4)
      unsigned char d1 = (nibble >> 3) & 1; // Bit más significativo
      unsigned char d2 = (nibble >> 2) & 1;
      unsigned char d3 = (nibble >> 1) & 1;
      unsigned char d4 = nibble & 1;          // Bit menos significativo

      // Calcular los bits de paridad según las ecuaciones de Hamming (7,4)
      unsigned char p1 = d1 ^ d2 ^ d4;        // p1 = d1 + d2 + d4 (XOR)
      unsigned char p2 = d1 ^ d3 ^ d4;        // p2 = d1 + d3 + d4 (XOR)
      unsigned char p3 = d2 ^ d3 ^ d4;        // p3 = d2 + d3 + d4 (XOR)

      // Formar la palabra código de 7 bits: p1, p2, d1, p3, d2, d3, d4
      unsigned char resultado[7] = {p1, p2, d1, p3, d2, d3, d4};

      // Colocar los 7 bits de la palabra código en el vector de salida
      for (int j = 0; j < 7; j++) {
      // Calcular el índice del byte de salida donde se colocará el bit
      int outByteIndex = outBitIndex / 8;
      // Calcular la posición del bit dentro del byte de salida
      int outBitPosition = 7 - (outBitIndex % 8);

      // Si estamos en un nuevo byte, inicializarlo a 0
      if (outBitPosition == 7) {
            out[outByteIndex] = 0;
      }

      // Colocar el bit en la posición correspondiente del byte de salida
```

```
        out[outByteIndex] |= (resultado[j] << outBitPosition);

        // Incrementar el índice del bit de salida
        outBitIndex++;
        }
        }
        }
  }

  /**
   * Método para decodificar un mensaje codificado con Hamming (7,4)
   * @param in Vector binario de entrada empaquetado en unsigned char
(mensaje codificado)
   * @param out Vector binario de salida empaquetado en unsigned char
(mensaje decodificado)
   * @param length Longitud del vector de entrada en bytes
   */
  void decode(unsigned char *in, unsigned char *out, int length) {
        int inBitIndex = 0;   // Índice del bit actual en el vector de entrada
        int outBitIndex = 0;  // Índice del bit actual en el vector de salida

        // Inicializar el vector de salida a ceros
        int outBytes = (length * 8 * 4) / 7; // Aproximación del número de
bytes necesarios
        outBytes = (outBytes + 7) / 8;  // Redondeo hacia arriba

        for (int i = 0; i < outBytes; i++) {
        out[i] = 0;
        }

        // Procesar cada palabra código de 7 bits
        while (inBitIndex + 6 < length * 8) { // Asegurarse de que hay al
menos 7 bits disponibles
        // Extraer los 7 bits de la palabra código
        unsigned char codeword[7];

        for (int j = 0; j < 7; j++) {
        // Calcular el índice del byte de entrada y la posición del bit
        int inByteIndex = inBitIndex / 8;
        int inBitPosition = 7 - (inBitIndex % 8);

        // Extraer el bit de la posición correspondiente
        codeword[j] = (in[inByteIndex] >> inBitPosition) & 1;

        // Avanzar al siguiente bit de entrada
        inBitIndex++;
        }
```

```
        // Extraer bits de paridad y datos de la palabra código
        unsigned char p1 = codeword[0]; // Bit de paridad 1
        unsigned char p2 = codeword[1]; // Bit de paridad 2
        unsigned char d1 = codeword[2]; // Bit de dato 1
        unsigned char p3 = codeword[3]; // Bit de paridad 3
        unsigned char d2 = codeword[4]; // Bit de dato 2
        unsigned char d3 = codeword[5]; // Bit de dato 3
        unsigned char d4 = codeword[6]; // Bit de dato 4

        // Calcular el síndrome para detectar errores
        unsigned char s1 = p1 ^ d1 ^ d2 ^ d4; // Verificar paridad 1
        unsigned char s2 = p2 ^ d1 ^ d3 ^ d4; // Verificar paridad 2
        unsigned char s3 = p3 ^ d2 ^ d3 ^ d4; // Verificar paridad 3

        // Determinar la posición del error (si existe)
        unsigned char errorPos = (s3 << 2) | (s2 << 1) | s1;

        // Corregir el error si se detecta (errorPos != 0)
        if (errorPos != 0) {
        // Posiciones de error y su correspondencia en el array codeword
        // 1 -> p1 (índice 0)
        // 2 -> p2 (índice 1)
        // 3 -> d1 (índice 2)
        // 4 -> p3 (índice 3)
        // 5 -> d2 (índice 4)
        // 6 -> d3 (índice 5)
        // 7 -> d4 (índice 6)

        // Corregir el bit erróneo
        if (errorPos >= 1 && errorPos <= 7) {
        // Invertir el bit en la posición del error
        codeword[errorPos - 1] = codeword[errorPos - 1] ^ 1;

        // Actualizar los bits de datos si fueron corregidos
        if (errorPos == 3) d1 = codeword[2];
        else if (errorPos == 5) d2 = codeword[4];
        else if (errorPos == 6) d3 = codeword[5];
        else if (errorPos == 7) d4 = codeword[6];
        }
        }

        // Reconstruir el nibble (4 bits) original
        unsigned char nibble = (d1 << 3) | (d2 << 2) | (d3 << 1) | d4;

        // Determinar si este nibble va en la parte alta o baja del byte de
salida
        if (outBitIndex % 8 == 0) {
        // Parte alta del byte (bits 7-4)
```

```
        out[outBitIndex / 8] |= (nibble << 4);
        } else {
        // Parte baja del byte (bits 3-0)
        out[outBitIndex / 8] |= nibble;
        }

        // Avanzar 4 bits en el índice de salida (un nibble)
        outBitIndex += 4;
        }
    }
};

/**
 * Clase que implementa un codificador y decodificador de repetición.
 * Permite codificar mensajes repitiendo cada bit n veces y decodificarlos
 * mediante un sistema de votación por mayoría.
 */
class RepetitionCode {
private:
    int repetitionDegree; // Grado de repetición (número de veces que se
repite cada bit)

public:
    /**
     * Constructor de la clase
     * @param n Grado de repetición (número de veces que se repite cada bit)
     */
    RepetitionCode(int n) {
        repetitionDegree = n;
    }

    /**
     * Método para establecer el grado de repetición
     * @param n Nuevo grado de repetición
     */
    void setRepetitionDegree(int n) {
        repetitionDegree = n;
    }

    /**
     * Método para obtener el grado de repetición actual
     * @return Grado de repetición
     */
    int getRepetitionDegree() {
        return repetitionDegree;
    }

    /**
```

```
   * Método para calcular la longitud del mensaje codificado en bytes
   * @param originalLength Longitud del mensaje original en bytes
   * @return Longitud del mensaje codificado en bytes
   */
  int getEncodedLength(int originalLength) {
      return (originalLength * 8 * repetitionDegree + 7) / 8; // Redondeo
hacia arriba
  }

  /**
   * Método para codificar un mensaje utilizando repetición
   * @param in Vector binario de entrada empaquetado en unsigned char
   * @param out Vector binario de salida empaquetado en unsigned char
   * @param length Longitud del vector de entrada en bytes
   */
  void encode(unsigned char *in, unsigned char *out, int length) {
      int outBitIndex = 0; // Índice del bit actual en el vector de salida

      // Recorrer cada byte del vector de entrada
      for (int i = 0; i < length; i++) {
      // Procesar cada bit del byte actual
      for (int j = 7; j >= 0; j--) {
      // Extraer el bit j-ésimo del byte i-ésimo
      unsigned char bit = (in[i] >> j) & 1;

      // Repetir el bit n veces
      for (int k = 0; k < repetitionDegree; k++) {
      // Calcular el índice del byte de salida donde se colocará el bit
repetido
      int outByteIndex = outBitIndex / 8;
      // Calcular la posición del bit dentro del byte de salida
      int outBitPosition = 7 - (outBitIndex % 8);

      // Si estamos en un nuevo byte, inicializarlo a 0
      if (outBitPosition == 7) {
            out[outByteIndex] = 0;
      }

      // Colocar el bit en la posición correspondiente del byte de salida
      out[outByteIndex] |= (bit << outBitPosition);

      // Incrementar el índice del bit de salida
      outBitIndex++;
      }
      }
      }
  }
```

```
  /**
   * Método para decodificar un mensaje codificado con repetición
   * @param in Vector binario de entrada empaquetado en unsigned char
(mensaje codificado)
   * @param out Vector binario de salida empaquetado en unsigned char
(mensaje decodificado)
   * @param length Longitud del vector de entrada en bytes
   */
  void decode(unsigned char *in, unsigned char *out, int length) {
       int inBitIndex = 0;  // Índice del bit actual en el vector de entrada
       int outBitIndex = 0; // Índice del bit actual en el vector de salida

       // Inicializar el vector de salida a ceros
       int outLength = (length * 8) / repetitionDegree; // Número total de
bits en el mensaje original
       int outBytes = (outLength + 7) / 8; // Número de bytes necesarios para
almacenar el mensaje original

       for (int i = 0; i < outBytes; i++) {
       out[i] = 0;
       }

       // Procesar cada grupo de n bits repetidos
       while (inBitIndex < length * 8) {
       int countOnes = 0; // Contador de unos en el grupo actual

       // Contar cuántos unos hay en el grupo de n bits
       for (int k = 0; k < repetitionDegree && inBitIndex < length * 8; k++)
{
       // Calcular el índice del byte de entrada y la posición del bit
       int inByteIndex = inBitIndex / 8;
       int inBitPosition = 7 - (inBitIndex % 8);

       // Extraer el bit de la posición correspondiente
       unsigned char bit = (in[inByteIndex] >> inBitPosition) & 1;

       // Incrementar el contador si el bit es 1
       if (bit == 1) {
       countOnes++;
       }

       // Avanzar al siguiente bit de entrada
       inBitIndex++;
       }

       // Determinar el bit original mediante votación por mayoría
       unsigned char decodedBit = (countOnes > repetitionDegree / 2) ? 1 : 0;
```

```
        // Calcular el índice del byte de salida y la posición del bit
        int outByteIndex = outBitIndex / 8;
        int outBitPosition = 7 - (outBitIndex % 8);

        // Colocar el bit decodificado en la posición correspondiente del byte
de salida
        out[outByteIndex] |= (decodedBit << outBitPosition);

        // Avanzar al siguiente bit de salida
        outBitIndex++;
        }
    }
};

/**
 * Clase que implementa un codificador y decodificador que combina Hamming y
Repetición en serie.
 * Primero aplica el código Hamming (7,4) y luego el código de repetición de
grado Rn.
 */
class HammingRepetition {
private:
    HammingCode hammingCoder; // Codificador/decodificador Hamming
    RepetitionCode repetitionCoder; // Codificador/decodificador de repetición

public:
    /**
     * Constructor de la clase
     * @param repetitionDegree Grado de repetición (número de veces que se
repite cada bit)
     */
    HammingRepetition(int repetitionDegree) :
repetitionCoder(repetitionDegree) {
        // Inicialización de objetos en la lista de inicialización
    }

    /**
     * Método para establecer el grado de repetición
     * @param n Nuevo grado de repetición
     */
    void setRepetitionDegree(int n) {
        repetitionCoder.setRepetitionDegree(n);
    }

    /**
     * Método para obtener el grado de repetición actual
     * @return Grado de repetición
     */
```

```
  int getRepetitionDegree() {
       return repetitionCoder.getRepetitionDegree();
  }

  /**
   * Método para calcular la longitud del mensaje codificado en bytes
   * @param originalLength Longitud del mensaje original en bytes
   * @return Longitud del mensaje codificado en bytes
   */
  int getEncodedLength(int originalLength) {
       // Primero calculamos la longitud después de aplicar Hamming
       int hammingLength = hammingCoder.getEncodedLength(originalLength);
       // Luego calculamos la longitud después de aplicar repetición
       return repetitionCoder.getEncodedLength(hammingLength);
  }

  /**
   * Método para codificar un mensaje utilizando Hamming seguido de
repetición
   * @param in Vector binario de entrada empaquetado en unsigned char
   * @param out Vector binario de salida empaquetado en unsigned char
   * @param length Longitud del vector de entrada en bytes
   */
  void encode(unsigned char *in, unsigned char *out, int length) {
       // Calcular la longitud del mensaje codificado con Hamming
       int hammingLength = hammingCoder.getEncodedLength(length);

       // Buffer temporal para almacenar el resultado de la codificación
Hamming
       unsigned char *hammingOut = new unsigned char[hammingLength];

       // Aplicar codificación Hamming
       hammingCoder.encode(in, hammingOut, length);

       // Aplicar codificación por repetición al resultado de Hamming
       repetitionCoder.encode(hammingOut, out, hammingLength);

       // Liberar memoria del buffer temporal
       delete[] hammingOut;
  }

  /**
   * Método para decodificar un mensaje codificado con repetición seguido de
Hamming
   * @param in Vector binario de entrada empaquetado en unsigned char
(mensaje codificado)
   * @param out Vector binario de salida empaquetado en unsigned char
(mensaje decodificado)
```

```
   * @param length Longitud del vector de entrada en bytes
   */
  void decode(unsigned char *in, unsigned char *out, int length) {
      // Calcular la longitud del mensaje después de decodificar la
repetición
      int hammingLength = (length * 8) /
repetitionCoder.getRepetitionDegree();
      hammingLength = (hammingLength + 7) / 8; // Redondeo hacia arriba para
obtener bytes

      // Buffer temporal para almacenar el resultado de la decodificación de
repetición
      unsigned char *repetitionOut = new unsigned char[hammingLength];

      // Aplicar decodificación de repetición
      repetitionCoder.decode(in, repetitionOut, length);

      // Aplicar decodificación Hamming al resultado
      hammingCoder.decode(repetitionOut, out, hammingLength);

      // Liberar memoria del buffer temporal
      delete[] repetitionOut;
  }

  /**
   * Método para mostrar información sobre un mensaje y su versión
codificada
   * @param original Mensaje original
   * @param coded Mensaje codificado
   * @param originalLength Longitud del mensaje original en bytes
   * @param codedLength Longitud del mensaje codificado en bytes
   */
  void printInfo(unsigned char *original, unsigned char *coded, int
originalLength, int codedLength) {
      Serial.println("Mensaje original:");
      printBinaryVector(original, originalLength);

      Serial.println("Mensaje codificado con Hamming+Repetición:");
      printBinaryVector(coded, codedLength);

      Serial.print("Grado de repetición: ");
      Serial.println(getRepetitionDegree());
      Serial.print("Bits del mensaje original: ");
      Serial.println(originalLength * 8);
      Serial.print("Bits del mensaje codificado: ");
      Serial.println(codedLength * 8);

      // Calcular y mostrar la tasa de código
```

```
        float rate = (float)(originalLength * 8) / (codedLength * 8);
        Serial.print("Tasa de código: ");
        Serial.println(rate, 4);
    }
};

/**
 * Función que simula un canal ruidoso.
 * @param in Vector binario de entrada empaquetado en unsigned char
 * @param out Vector binario de salida empaquetado en unsigned char
 * @param l Longitud del vector en bytes
 * @param f Probabilidad de que un bit cambie de valor (entre 0 y 1)
 */
void noisyChannel(unsigned char *in, unsigned char *out, int l, float f) {
    // Inicializar la semilla para la generación de números aleatorios
    randomSeed(analogRead(0));

    // Recorrer cada byte del vector de entrada
    for (int i = 0; i < l; i++) {
        // Inicializar el byte de salida con el valor del byte de entrada
        out[i] = in[i];

        // Procesar cada bit del byte actual
        for (int j = 0; j < 8; j++) {
        // Generar un número aleatorio entre 0 y 1
        float r = random(0, 100) / 100.0;

        // Si el número aleatorio es menor que la probabilidad de error f,
        // invertir el bit correspondiente en el byte de salida
        if (r < f) {
        // Invertir el bit j-ésimo usando XOR con una máscara
        out[i] ^= (1 << j);
        }
        }
    }
}

void setup() {
    // Inicializar comunicación serial
    Serial.begin(9600);
    while (!Serial) {
        ; // Esperar a que el puerto serial se conecte
    }

    // Mensaje original para demostración
    const int originalLength = 2; // Longitud del mensaje original en bytes
    unsigned char original[originalLength] = {0b10101010, 0b11110000};
```

```
  // Probabilidad de error del canal ruidoso
  const float ERROR_PROBABILITY = 0.05;

  // Crear una instancia de HammingRepetition con grado de repetición 3
  HammingRepetition hammingRepCode(3);

  // Calcular el tamaño necesario para el mensaje codificado
  int codedLength = hammingRepCode.getEncodedLength(originalLength);

  // Crear vectores para almacenar el mensaje codificado y el mensaje
recibido con ruido
  unsigned char coded[codedLength];
  unsigned char received[codedLength];

  // Codificar el mensaje original
  hammingRepCode.encode(original, coded, originalLength);

  // Mostrar información sobre el mensaje original y el codificado
  Serial.println("=== Demostración de HammingRepetition ===");
  hammingRepCode.printInfo(original, coded, originalLength, codedLength);

  // Simular la transmisión a través de un canal ruidoso
  Serial.println("\nSimulando transmisión a través de un canal ruidoso...");
  Serial.print("Probabilidad de error del canal: ");
  Serial.println(ERROR_PROBABILITY);

  noisyChannel(coded, received, codedLength, ERROR_PROBABILITY);

  Serial.println("\nMensaje recibido (con ruido):");
  printBinaryVector(received, codedLength);

  // Decodificar el mensaje recibido
  unsigned char decoded[originalLength];
  hammingRepCode.decode(received, decoded, codedLength);

  // Mostrar el mensaje decodificado
  Serial.println("\nMensaje decodificado:");
  printBinaryVector(decoded, originalLength);

  // Verificar si la decodificación fue exitosa
  bool success = true;
  for (int i = 0; i < originalLength; i++) {
       if (original[i] != decoded[i]) {
       success = false;
       break;
       }
  }
```

```
  Serial.print("\nDecodificación ");
  if (success) {
       Serial.println("exitosa! El mensaje se recuperó correctamente.");
  } else {
       Serial.println("fallida. El mensaje contiene errores.");
  }

  // Comparar con solo Hamming y solo Repetición
  Serial.println("\n=== Comparación con otros códigos ===");

  // Crear instancias de los codificadores individuales
  HammingCode hammingCode;
  RepetitionCode repetitionCode(3);

  // Calcular longitudes codificadas
  int hammingLength = hammingCode.getEncodedLength(originalLength);
  int repetitionLength = repetitionCode.getEncodedLength(originalLength);

  Serial.println("Longitud en bytes del mensaje original: " +
String(originalLength));
  Serial.println("Longitud en bytes con Hamming (7,4): " +
String(hammingLength));
  Serial.println("Longitud en bytes con Repetición (R3): " +
String(repetitionLength));
  Serial.println("Longitud en bytes con Hamming+Repetición: " +
String(codedLength));

  Serial.println("\nTasa de código Hamming (7,4): 4/7 = " +
String((float)4/7, 4));
  Serial.println("Tasa de código Repetición (R3): 1/3 = " +
String((float)1/3, 4));
  Serial.println("Tasa de código Hamming+Repetición: " +
String((float)(originalLength * 8) / (codedLength * 8), 4));
}

void loop() {
  // No se requiere funcionalidad en el bucle principal
}
```

Ejercicio 11.

Escriba un programa que tome un fichero de texto plano de cualquier longitud y devuelva un fichero con el porcentaje relativo (en tanto por uno) de cada una de las letras del alfabeto

dentro del fichero, normalizado a mayúsculas. (utilice un fichero de texto de al menos 1 millón de caracteres.

```C/C++
/* Plantilla proyectos arduino */
#include "SPIFFS.h"
#include <ctype.h>

void processFile();

void setup(){
       Serial.begin(115200);
       if(!SPIFFS.begin(true)){
       Serial.println("Error montando SPIFFS");
       return;
       }

       // Listar archivos en SPIFFS
       File root = SPIFFS.open("/");
       Serial.println("Contenido SPIFFS:");
       while(File file = root.openNextFile()){
       Serial.print("  ");
       Serial.println(file.name());
       }
       root.close();

       processFile();
}

void loop(){}

void processFile(){
       File input = SPIFFS.open("/data/texto.txt");
       if(!input){
       Serial.println("Error abriendo archivo: /data/texto.txt");
       return;
       }
       Serial.print("Tamaño archivo: ");
       Serial.print(input.size());
       Serial.println(" bytes");

       long counts[26] = {0};
       long total = 0;

       while(input.available()){
       char c = input.read();
       if(isalpha(c)){ // Corregir nombre función
```

```
        Serial.print("Carácter válido detectado: "); Serial.println(c);
              c = toupper(c);
              counts[c - 'A']++;
              total++;
        }
        }

        if(total == 0){
        Serial.println("Archivo contiene " + String(input.size()) + " bytes");
        Serial.println("Caracteres válidos detectados: 0");
        Serial.println("Posible causa: Archivo vacío o formato incorrecto");
        input.close();
        return;
        }
        input.close();


        File output = SPIFFS.open("/data/resultados.txt", FILE_WRITE);
        if(!output){
        Serial.println("Error creando archivo de resultados");
        return;
        }

        for(int i = 0; i < 26; i++){
        float porcentaje = counts[i] * 1.0f / total;
        char buffer[50];
        snprintf(buffer, sizeof(buffer), "%c: %.4f\n", 'A' + i, porcentaje);
        output.print(buffer);
        }
        output.close();
        Serial.println("Análisis completado. Resultados guardados en
/data/resultados.txt");
}
```