ITC Homework Assignment
Block code and Cyclic codes
Name - Anmol Agrawal
Roll No - 122CS0300

1. Implementation of Block code encoding and decoding

```
#include <iostream>
#include <vector>
#include <bitset>
#include <string>

// Implementation of (7,4) Hamming block  Code

class HammingProcessor {
private:
    // Stores bit positions for parity calculations
    struct ParityCheck {
        std::vector<int> p1_positions = {0, 2, 4, 6};
        std::vector<int> p2_positions = {1, 2, 5, 6};
        std::vector<int> p3_positions = {3, 4, 5, 6};
    } parity_map;

public:
    // Takes 4-bit message and returns 7-bit Hamming encoded message
    std::vector<int> generateCode(const std::vector<int>& msg_bits) {
        // Validate input
        if (msg_bits.size() != 4) {
            throw std::invalid_argument("Message must be exactly 4 bits");
```

```
        }

        // Initialize the 7-bit codeword (all zeros)
        std::vector<int> result(7, 0);

        // Place data bits in their positions
        // Data bits go at positions 3, 5, 6, 7 (using 1-based
indexing)
        // Which corresponds to indices 2, 4, 5, 6 (using
0-based indexing)
        result[2] = msg_bits[0];
        result[4] = msg_bits[1];
        result[5] = msg_bits[2];
        result[6] = msg_bits[3];

        // Calculate parity bits using XOR of appropriate
positions
        result[0] = calcParityBit(result,
parity_map.p1_positions);
        result[1] = calcParityBit(result,
parity_map.p2_positions);
        result[3] = calcParityBit(result,
parity_map.p3_positions);

        return result;
    }

    // Processes received code, detects and corrects single-bit
errors
    std::vector<int> processReceivedCode(std::vector<int>
received) {
        // Print the received codeword
        printBits("Received codeword", received);

```

```
        // Calculate syndrome bits
        int syndrome_bit1 = calcParityBit(received,
parity_map.p1_positions);
        int syndrome_bit2 = calcParityBit(received,
parity_map.p2_positions);
        int syndrome_bit3 = calcParityBit(received,
parity_map.p3_positions);

        // Convert syndrome to position (binary to decimal)
        int error_position = (syndrome_bit3 << 2) |
(syndrome_bit2 << 1) | syndrome_bit1;

        // Handle error correction
        if (error_position > 0) {
            std::cout << "! Error detected at position " <<
error_position << std::endl;
            // Flip the erroneous bit (0-indexed)
            received[error_position - 1] =
!received[error_position - 1];
            printBits("After correction", received);
        } else {
            std::cout << "√ No errors detected in transmission"
<< std::endl;
        }

        // Extract and return original message bits
        std::vector<int> original_msg = {
            received[2], received[4], received[5], received[6]
        };

        printBits("Decoded message", original_msg);
        return original_msg;
    }
```

```
private:
    // Helper to calculate parity bits using XOR
    int calcParityBit(const std::vector<int>& bits, const
std::vector<int>& positions) {
        int result = 0;
        for (int pos : positions) {
            result ^= bits[pos];
        }
        return result;
    }

    // Helper to print bit vectors nicely
    void printBits(const std::string& label, const
std::vector<int>& bits) {
        std::cout << label << ": ";
        for (int bit : bits) {
            std::cout << bit;
        }
        std::cout << std::endl;
    }
};

// Test function that demonstrates Hamming code with error
introduction
void runHammingDemo() {
    HammingProcessor hamming;

    // Test message - can be changed to any 4-bit combination
    std::vector<int> test_message = {1, 0, 1, 1};

    std::cout << "----- HAMMING CODE DEMONSTRATION -----" <<
std::endl;
    std::cout << "Original message: ";
    for (int bit : test_message) {
```

```
        std::cout << bit;
    }
    std::cout << std::endl;

    // Encode the message
    std::vector<int> encoded =
hamming.generateCode(test_message);

    std::cout << "Encoded message:  ";
    for (int bit : encoded) {
        std::cout << bit;
    }
    std::cout << std::endl;

    // Introduce an error (flip bit at position 3)
    std::cout << std::endl << "Simulating transmission error..."
<< std::endl;
    int error_pos = 2; // 0-based index
    encoded[error_pos] ^= 1; // Flip the bit

    // Decode and correct if needed
    std::cout << std::endl << "RECEIVER SIDE:" << std::endl;
    std::vector<int> decoded =
hamming.processReceivedCode(encoded);

    // Verify correctness
    bool match = true;
    for (size_t i = 0; i < test_message.size(); i++) {
        if (test_message[i] != decoded[i]) {
            match = false;
            break;
        }
    }
```

```cpp
    std::cout << std::endl << "Verification: "
              << (match ? "SUCCESS - Original message recovered"
: "FAILED - Could not recover message")
              << std::endl;
}

// Entry point
int main() {
    try {
        runHammingDemo();
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

```
```

Output:

Output

```
----- HAMMING CODE DEMONSTRATION -----
Original message: 1011
Encoded message:  0110011

Simulating transmission error...

RECEIVER SIDE:
Received codeword: 0100011
! Error detected at position 3
After correction: 0110011
Decoded message: 1011

Verification: SUCCESS - Original message recovered


=== Code Execution Successful ===
```

2. Implementation of cyclic code encoding and decoding

```

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

// Cyclic Codes Implementation
// A polynomial division based approach for detecting
transmission errors
```
```

```cpp
class CyclicCodeProcessor {
private:
    std::vector<int> generator_poly;
    int data_length;
    int checksum_size;

public:
    CyclicCodeProcessor(const std::vector<int>& gen_polynomial)
:
        generator_poly(gen_polynomial),
        checksum_size(gen_polynomial.size() - 1) {
        // Validate generator polynomial (must start with 1)
        if (gen_polynomial.empty() || gen_polynomial[0] != 1) {
            throw std::runtime_error("Invalid generator
polynomial");
        }
    }

    // Performs polynomial division in GF(2)
    // Returns the remainder of dividend / divisor
    std::vector<int> calculateRemainder(const std::vector<int>&
dividend) const {
        // Create a working copy of the dividend
        std::vector<int> work_buffer = dividend;
        int dividend_size = work_buffer.size();
        int divisor_size = generator_poly.size();

        // Perform polynomial long division in GF(2)
        for (int i = 0; i <= dividend_size - divisor_size; i++)
{
            // If current bit is 1, perform XOR with generator
            if (work_buffer[i]) {
                for (int j = 0; j < divisor_size; j++) {
                    // XOR operation (addition in GF(2))
```

```
                    work_buffer[i + j] = work_buffer[i + j] ^
generator_poly[j];
                }
            }
        }

        // Extract remainder (last checksum_size bits)
        std::vector<int> remainder;
        remainder.reserve(checksum_size);
        for (int i = dividend_size - checksum_size; i <
dividend_size; i++) {
            remainder.push_back(work_buffer[i]);
        }

        return remainder;
    }

    // Creates a codeword from message by appending checksum
    std::vector<int> createCodeword(const std::vector<int>& msg)
{
        data_length = msg.size();

        // Step 1: Create dividend by appending zeros
        std::vector<int> padded_msg = msg;
        for (int i = 0; i < checksum_size; i++) {
            padded_msg.push_back(0);
        }

        // Step 2: Calculate remainder
        std::vector<int> checksum =
calculateRemainder(padded_msg);

        // Step 3: Create codeword = message + checksum
        std::vector<int> result = msg;
```

```
        result.insert(result.end(), checksum.begin(),
checksum.end());

        return result;
    }

    // Validates a received codeword and extracts original
message if valid
    bool validateAndExtract(const std::vector<int>&
received_word, std::vector<int>& extracted_msg) {
        // Calculate syndrome (remainder after division)
        std::vector<int> syndrome =
calculateRemainder(received_word);

        // Check if syndrome is all zeros (indicating no errors)
        bool is_valid = true;
        for (int bit : syndrome) {
            if (bit != 0) {
                is_valid = false;
                break;
            }
        }

        // Extract original message if valid
        if (is_valid && data_length > 0) {
            extracted_msg.clear();
            for (int i = 0; i < data_length; i++) {
                extracted_msg.push_back(received_word[i]);
            }
        }

        return is_valid;
    }
```

```
    // Debug helper to print bit arrays
    static void printBits(const std::string& label, const
std::vector<int>& bits) {
        std::cout << label;
        for (int bit : bits) {
            std::cout << bit;
        }
        std::cout << std::endl;
    }
};

// Demo function showing cyclic code error detection
void demonstrateCyclicCode() {
    // Define a generator polynomial (x³ + x + 1)
    std::vector<int> generator = {1, 0, 1, 1};

    // Sample message to encode
    std::vector<int> message = {1, 0, 1, 1};

    // Create processor with our generator polynomial
    CyclicCodeProcessor processor(generator);

    std::cout << "=== CYCLIC CODE ERROR DETECTION DEMO ===" <<
std::endl;
    processor.printBits("Generator polynomial: ", generator);
    processor.printBits("Original message:     ", message);

    // Encode the message
    std::vector<int> encoded =
processor.createCodeword(message);
    processor.printBits("Encoded codeword:     ", encoded);

    // Testing error-free transmission
    std::vector<int> clean_received = encoded;
```

```cpp
    std::vector<int> extracted_msg;

    std::cout << "\n--- TEST 1: ERROR-FREE TRANSMISSION ---" <<
std::endl;
    processor.printBits("Received codeword:    ",
clean_received);

    bool is_valid = processor.validateAndExtract(clean_received,
extracted_msg);
    if (is_valid) {
        std::cout << "Status: VALID - No errors detected" <<
std::endl;
        processor.printBits("Extracted message:    ",
extracted_msg);
    } else {
        std::cout << "Status: INVALID - Errors detected" <<
std::endl;
    }

    // Testing with an error
    std::cout << "\n--- TEST 2: TRANSMISSION WITH ERROR ---" <<
std::endl;
    std::vector<int> corrupted = encoded;

    // Introduce an error at position 5
    int error_pos = 5;
    corrupted[error_pos] = corrupted[error_pos] ^ 1;

    processor.printBits("Corrupted codeword:   ", corrupted);

    is_valid = processor.validateAndExtract(corrupted,
extracted_msg);
    if (is_valid) {
```

```
        std::cout << "Status: VALID - No errors detected" << std::endl;
        processor.printBits("Extracted message:    ", extracted_msg);
    } else {
        std::cout << "Status: INVALID - Errors detected" << std::endl;
        std::cout << "Message cannot be reliably decoded" << std::endl;
    }
}

int main() {
    try {
        demonstrateCyclicCode();
    }
    catch (const std::exception& e) {
        std::cerr << "ERROR: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

Output

```
=== CYCLIC CODE ERROR DETECTION DEMO ===
Generator polynomial: 1011
Original message:     1011
Encoded codeword:     1011000

--- TEST 1: ERROR-FREE TRANSMISSION ---
Received codeword:    1011000
Status: VALID - No errors detected
Extracted message:    1011

--- TEST 2: TRANSMISSION WITH ERROR ---
Corrupted codeword:   1011010
Status: INVALID - Errors detected
Message cannot be reliably decoded


=== Code Execution Successful ===
```