Министерство образования Республики Беларусь
Учреждение образования
«Брестский государственный технический университет»
Кафедра ИИТ

Лабораторная работа №1

За седьмой семестр

По дисциплине: «Современные методы защиты компьютерных систем»

Тема: «Проектирование классов»

Выполнила:
Студентка 4 курса
Группы ИИ-21(II)
Соболева П.С.

Проверила:
Хацкевич А. С.

Брест 2024

Цель: приобретение практических навыков кодирования/декодирования двоичных данных при использовании кода Хемминга.

Ход работы:

Вариант 11

Задание:

1. Составить код Хемминга (классический алгоритм) (M+r, M), допустить ошибку в одном из разрядов и отыскать её по алгоритму.
2. Составить код Хемминга (расширенный алгоритм) (11,7), допустить 2 или более ошибок в разрядах и отыскать их по алгоритму.

Код:

```
def decimal_to_binary(n):
    """Перевод десятичного числа в двоичное представление."""
    return bin(n)[2:]

def calculate_parity_bits_positions(r):
    """Определение позиций проверочных битов в коде Хемминга."""
    return [2 ** i for i in range(r)]

def generate_hamming_code(data_bits, r):
    """Генерация кода Хемминга с проверочными битами."""
    m = len(data_bits)  # Количество информационных битов
    positions = calculate_parity_bits_positions(r)  # Позиции проверочных битов
    code_length = m + r  # Общая длина кодового слова
    code = [0] * code_length  # Инициализация кодового слова нулями

    j = 0  # Индекс для прохода по информационным битам
    # Заполнение кодового слова информационными битами
    for i in range(1, code_length + 1):
        if i in positions:
            continue  # Пропуск позиций, отведенных под проверочные биты
        if j < m:  # Проверяем, что индекс не выходит за пределы data_bits
            try:
                code[i - 1] = int(data_bits[j])  # Заполняем информационные биты
            except IndexError:
                print(f"Ошибка индекса при добавлении информационного бита: i = {i}, j =
{j}")
            j += 1

    # Вычисление значений проверочных битов
    for position in positions:
        parity = 0
        for i in range(1, code_length + 1):
            if i & position:
                parity ^= code[i - 1]
        code[position - 1] = parity

    return code

def introduce_error(code):
    """Внесение ошибки в случайный бит кодового слова."""
```

```
    import random
    if not code:
        raise ValueError("Код Хемминга пуст. Ошибка при генерации кода.")
    error_position = random.randint(0, len(code) - 1)
    code[error_position] ^= 1  # Инвертируем бит для внесения ошибки
    print(f"Ошибка внесена в позицию: {error_position + 1}")
    return code, error_position + 1

def detect_and_correct_error(code, r):
    """Обнаружение и исправление ошибки в коде Хемминга."""
    positions = calculate_parity_bits_positions(r)
    error_position = 0

    # Определение позиции ошибки по проверочным битам
    for position in positions:
        parity = 0
        for i in range(1, len(code) + 1):
            if i & position:
                parity ^= code[i - 1]
        if parity != 0:
            error_position += position

    if error_position == 0:
        print("Ошибок не обнаружено.")
    else:
        print(f"Обнаружена ошибка в позиции: {error_position}")
        code[error_position - 1] ^= 1
        print("Ошибка исправлена.")

    return code

# Исходные данные
M = 590
r = 4

# 1. Перевод в двоичное представление
binary_data = decimal_to_binary(M)
print(f"Двоичное представление M ({M}): {binary_data}")

# 2. Генерация кода Хемминга
hamming_code = []  # Инициализация переменной перед использованием
try:
    hamming_code = generate_hamming_code(binary_data, r)
    if not hamming_code:
        raise ValueError("Не удалось сгенерировать код Хемминга.")
    print(f"Код Хемминга: {''.join(map(str, hamming_code))}")
except IndexError as e:
    print(f"Ошибка при генерации кода Хемминга: {e}")
except ValueError as e:
    print(e)

# 3. Внесение ошибки (проверяем, что код сгенерирован)
if hamming_code:
    try:
        code_with_error, error_position = introduce_error(hamming_code.copy())
        print(f"Код с ошибкой: {''.join(map(str, code_with_error))}")
    except ValueError as e:
```

```
        print(e)

# 4. Обнаружение и исправление ошибки
if hamming_code:
    corrected_code = detect_and_correct_error(code_with_error, r)
    print(f"Исправленный код: {''.join(map(str, corrected_code))}")
```

Результат:

```
Двоичное представление M (590): 1001001110
Код Хемминга: 01110011001110
Ошибка внесена в позицию: 5
Код с ошибкой: 01111011001110
Обнаружена ошибка в позиции: 5
Ошибка исправлена.
Исправленный код: 01110011001110
```

Код:

```
import random


def decimal_to_binary(n, length):
    """Перевод десятичного числа в двоичное представление с фиксированной длиной."""
    binary_str = bin(n)[2:]  # Получаем двоичное представление без префикса '0b'
    return binary_str.zfill(length)  # Дополняем до нужной длины нулями слева


def calculate_parity_bits_positions(r):
    """Определение позиций проверочных битов в коде Хемминга."""
    return [2 ** i for i in range(r)]


def generate_hamming_code_11_7(data_bits):
    """Генерация кода Хемминга (11,7) с проверочными битами."""
    m = len(data_bits)  # Количество информационных битов (7)
    r = 4  # Количество проверочных битов для кода (11,7)
    positions = calculate_parity_bits_positions(r)  # Позиции проверочных битов: 1, 2, 4,
8
    code_length = m + r  # Общая длина кодового слова: 11
    code = [0] * code_length  # Инициализация кодового слова нулями

    j = 0  # Индекс для прохода по информационным битам
    # Заполнение кодового слова информационными битами
    for i in range(1, code_length + 1):
        if i in positions:
            continue  # Пропуск позиций, отведенных под проверочные биты
        code[i - 1] = int(data_bits[j])  # Заполняем информационные биты
        j += 1

    # Вычисление значений проверочных битов
    for position in positions:
        parity = 0
        for i in range(1, code_length + 1):
            if i & position:  # Проверяем, участвует ли бит в проверке для данного
проверочного бита
                parity ^= code[i - 1]  # Вычисляем четность
        code[position - 1] = parity  # Устанавливаем значение проверочного бита
```

```
    return code


def introduce_two_errors(code):
    """Внесение двух ошибок в случайные биты кодового слова."""
    if len(code) < 2:
        raise ValueError("Код слишком короткий для внесения двух ошибок.")
    error_positions = random.sample(range(len(code)), 2)  # Две случайные позиции для
ошибок
    for pos in error_positions:
        code[pos] ^= 1  # Инвертируем бит для внесения ошибки
    print(f"Ошибки внесены в позиции: {[pos + 1 for pos in error_positions]}")
    return code, [pos + 1 for pos in error_positions]


def detect_errors(code, r):
    """Обнаружение всех ошибок в коде Хемминга (11,7)."""
    positions = calculate_parity_bits_positions(r)
    error_position = 0

    # Определение позиции ошибки по проверочным битам
    for position in positions:
        parity = 0
        for i in range(1, len(code) + 1):
            if i & position:
                parity ^= code[i - 1]
        if parity != 0:
            error_position += position

    return error_position


def find_two_errors(code, r):
    """Попытка обнаружения двух ошибок с помощью перебора."""
    original_error_position = detect_errors(code, r)
    if original_error_position == 0:
        print("Ошибок не обнаружено.")
        return []

    # Пробуем найти комбинации ошибок
    possible_errors = []
    for i in range(len(code)):
        for j in range(i + 1, len(code)):
            modified_code = code.copy()
            modified_code[i] ^= 1
            modified_code[j] ^= 1
            if detect_errors(modified_code, r) == 0:
                possible_errors.append((i + 1, j + 1))

    if possible_errors:
        print(f"Найдены возможные позиции двух ошибок: {possible_errors}")
    else:
        print("Не удалось найти две ошибки.")
    return possible_errors


def correct_errors(code, error_positions):
```

```
    """Исправление двух ошибок."""
    if not error_positions:
        print("Ошибки не обнаружены для исправления.")
        return code

    for pos in error_positions:
        code[pos - 1] ^= 1  # Исправляем ошибки
        print(f"Ошибка исправлена в позиции: {pos}")

    return code


# Исходные данные
M = 590  # Пример числа (в двоичном виде: 1011010)
binary_data = decimal_to_binary(M, 7)  # Приведение длины до 7 бит (информационные биты)

print(f"Двоичное представление числа M ({M}): {binary_data}")

# Генерация кода Хемминга (11,7)
hamming_code = generate_hamming_code_11_7(binary_data)
print(f"Код Хемминга (11,7): {''.join(map(str, hamming_code))}")

# Внесение двух ошибок
code_with_two_errors, error_positions = introduce_two_errors(hamming_code.copy())
print(f"Код с двумя ошибками: {''.join(map(str, code_with_two_errors))}")

# Обнаружение двух ошибок
found_errors = find_two_errors(code_with_two_errors, 4)

# Исправление ошибок
if found_errors:
    corrected_code = correct_errors(code_with_two_errors, found_errors[0])
    print(f"Код после исправления: {''.join(map(str, corrected_code))}")
else:
    print("Исправление не удалось.")
```

Результат:

```
Двоичное представление числа M (590): 1001001110
Код Хемминга (11,7): 01110011001110
Ошибки внесены в позиции: [4, 1]
Код с двумя ошибками: 11100011001110
Найдены возможные позиции двух ошибок: [(1, 4), (2, 7), (3, 6), (8, 13), (9, 12), (11,
14)]
Ошибка исправлена в позиции: 1
Ошибка исправлена в позиции: 4
Код после исправления: 01110011001110
```

Вывод: в ходе лабораторной работы изучила кодирование/декодирование двоичных данных при использовании кода Хемминга.