Data Communication and Networks

Lab experiment 04

Name - Naman Dixit

Sap ID - 500125539

Batch - 7 Data Science

# Write a program to implement Hamming code concept. Make a test run and verify the result.

Use formula - 2 ^ r >= m + r + 1

Sol->

import java.util.Scanner;

public class HammingCode {

// Method to calculate the number of redundant bits needed

private static int calculateRedundantBits(int m) {

int r = 0;

while (Math.pow(2, r) < (m + r + 1)) {

r++;

}

return r;

}

// Method to position redundant bits in the data

private static int[] insertRedundantBits(int[] data, int r) {

int[] hammingCode = new int[data.length + r];

int j = 0;

for (int i = 0; i < hammingCode.length; i++) {

if (Math.pow(2, j) - 1 == i) {

// Position for redundant bit

hammingCode[i] = 0;

j++;

} else {

hammingCode[i] = data[i - j];

}

}

return hammingCode;

}

// Method to calculate parity bits

private static void calculateParityBits(int[] hammingCode, int r) {

for (int i = 0; i < r; i++) {

int parityBitPosition = (int) Math.pow(2, i) - 1;

int parity = 0;

for (int j = parityBitPosition; j < hammingCode.length; j += (parityBitPosition + 1) \* 2) {

for (int k = j; k < j + parityBitPosition + 1 && k < hammingCode.length; k++) {

parity ^= hammingCode[k];

}

}

hammingCode[parityBitPosition] = parity;

}

}

// Method to detect and correct errors

private static int detectAndCorrectError(int[] hammingCode, int r) {

int errorPosition = 0;

for (int i = 0; i < r; i++) {

int parityBitPosition = (int) Math.pow(2, i) - 1;

int parity = 0;

for (int j = parityBitPosition; j < hammingCode.length; j += (parityBitPosition + 1) \* 2) {

for (int k = j; k < j + parityBitPosition + 1 && k < hammingCode.length; k++) {

parity ^= hammingCode[k];

}

}

if (parity != 0) {

errorPosition += parityBitPosition + 1;

}

}

if (errorPosition != 0) {

hammingCode[errorPosition - 1] ^= 1; // Correct the error

}

return errorPosition;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of data bits: ");

int m = scanner.nextInt();

int[] data = new int[m];

System.out.print("Enter the data bits (space-separated): ");

for (int i = 0; i < m; i++) {

data[i] = scanner.nextInt();

}

int r = calculateRedundantBits(m);

int[] hammingCode = insertRedundantBits(data, r);

calculateParityBits(hammingCode, r);

System.out.println("Hamming code after adding parity bits: ");

for (int bit : hammingCode) {

System.out.print(bit + " ");

}

System.out.println();

System.out.print("Introduce a 1-based error position (or 0 for no error): ");

int errorPosition = scanner.nextInt();

if (errorPosition != 0) {

hammingCode[errorPosition - 1] ^= 1; // Introduce an error

}

System.out.println("Received Hamming code: ");

for (int bit : hammingCode) {

System.out.print(bit + " ");

}

System.out.println();

int detectedErrorPosition = detectAndCorrectError(hammingCode, r);

if (detectedErrorPosition != 0) {

System.out.println("Error detected and corrected at position: " + detectedErrorPosition);

} else {

System.out.println("No error detected.");

}

System.out.println("Corrected Hamming code: ");

for (int bit : hammingCode) {

System.out.print(bit + " ");

}

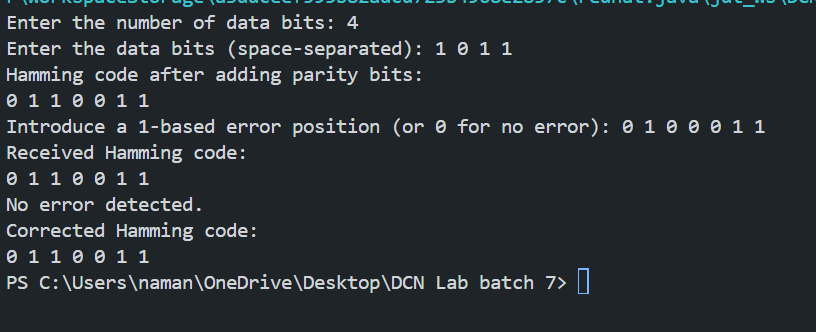
System.out.println();

scanner.close();

}

}

Output:-



### Algorithm to Implement Hamming Code:-

#### Input:

1. Number of data bits m.
2. The actual data bits D[] (binary values: 0 or 1).

#### Output:

1. The complete Hamming code with redundant (parity) bits.
2. Error detection and correction (if applicable).

Steps:

#### 1. Calculate the Number of Redundant Bits (r):

* Use the formula 2^r >= m+r+1, where:
  + m = number of data bits.
  + r = number of redundant bits.
  + m+r+1 = total number of bits (data + redundant bits + error bit).
* Initialize r=0. Increment r until 2^r≥m+r+1 is satisfied.
* Output: The total number of redundant bits, r.

#### 2. Insert Redundant Bits into the Data:

* Create a new array H[] to represent the Hamming Code (length m+r).
* Redundant bits are placed at positions 2^i (1-based index), i.e., positions 1,2,4,8,…1, 2, 4, 8, leaving other positions for data bits.
* Copy the data bits from D[] into the remaining positions of H[].
* Output: A partially-filled Hamming Code array with redundant bit positions initialized to 0.

3. Calculate Parity Bits:

* For each redundant bit at position 2^i, calculate its value based on the XOR of specific data and redundant bits:
  + Parity Calculation Rule: The bit at position 2^i affects all positions where the binary representation of the position has the i-th bit set to 1.
  + Traverse the Hamming Code array and perform XOR for all affected bits.
  + Update the redundant bit with the calculated parity value (0 or 1).
* Output: The complete Hamming Code with parity bits.

#### 4. Simulate Error Introduction (Optional):

* Allow the user to manually introduce an error by flipping a bit at a specific 1-based index.
* Flip the bit at the given position (0 → 1 or 1 → 0).

#### 5. Detect and Correct Errors:

* Initialize errorPosition = 0.
* For each redundant bit r*i*th at position 2^i, calculate parity using the same rule as in Step 3.
* If the calculated parity is incorrect (not 0), add the position 2^i to errorPosition.
* Case Handling:
  + If errorPosition = 0: No error is detected.
  + If errorPosition > 0: An error is detected at the bit corresponding to the 1-based index errorPosition. Flip the bit to correct it.

#### 6. Display Results:

* Print the Hamming Code after adding parity bits.
* Print the received Hamming Code (after simulating an error, if any).
* Display the position of the detected error (if any) and the corrected Hamming Code.

Illustrative Example:

#### Input:

* Number of data bits m=4.
* Data bits D[]=[1,0,1,1].

#### Execution:

1. Calculate Redundant Bits:
   * 2r≥m+r+1.
   * r=3, as 2^3=8≥4+3+1=8.
2. Insert Redundant Bits:
   * Total Hamming Code length = m+r=4+3=7.
   * Positions: 1,2,4 → redundant bits (initially 0).
   * Positions: 3,5,6,7 → data bits.
   * Partially-filled Hamming Code: [0,0,1,0,1,1,1].
3. Calculate Parity Bits:
   * Redundant bit at position 1: XOR bits 1,3,5,7 → 0XOR1XOR1XOR1=1.
   * Redundant bit at position 2: XOR bits 2,3,6,7 → 0XOR1XOR1XOR1=1.
   * Redundant bit at position 4: XOR bits 4,5,6,7 → 0XOR1XOR1XOR1=1.
   * Complete Hamming Code: [1,1,1,1,1,1,1].
4. Simulate Error:
   * Flip bit at position 4: [1,1,1,0,1,1,1].
5. Detect and Correct Error:
   * Recalculate parities for positions 1,2,4:
     + P1=1 (incorrect)
     + P2 = 1 (incorrect)
     + P4=0 (incorrect).
   * Error position: 1+2+4=7.
   * Correct the error by flipping bit 7: [1,1,1,1,1,1,1].

Output:

* Original Hamming Code: [1,1,1,1,1,1,1].
* Received Code (with error): [1,1,1,0,1,1,1].
* Error detected and corrected at position 7.
* Corrected Code: [1,1,1,1,1,1,1]..

### Time Complexity:

1. Redundant bit calculation: O(r)O(r)O(r).
2. Parity calculation: O(r⋅(m+r))O(r \cdot (m + r))O(r⋅(m+r)).
3. Error detection: O(r⋅(m+r))O(r \cdot (m + r))O(r⋅(m+r)).