Beni-Suef University
Communications and Electronics Engineering 4rd.

Final Project

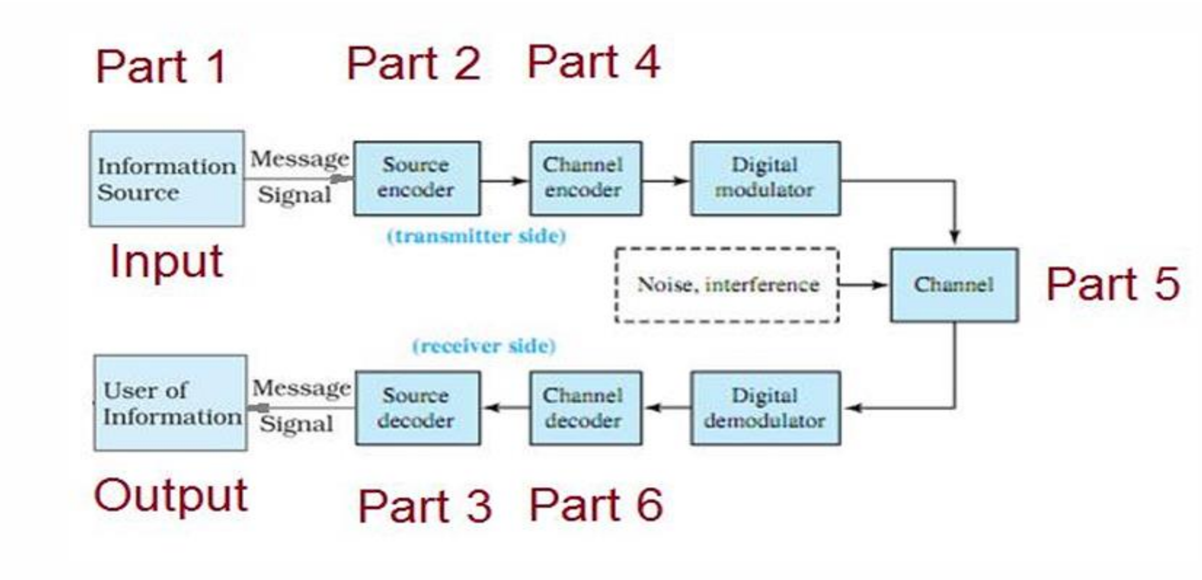
Information Theory

1- *Ahmed Ragab Mogoda*
2- *Abdallah Ashraf Abdallah*
3- *Mohamed Abdo Tawfig*
4- *Amar Ragab Youssef*

Dr/ Mohamed Faisal

Communication system

Project structure



Project Overview: Shannon-Fano Encoding with Hamming Code for Error Detection and Correction

This project involves **data compression** and **error correction** using two main algorithms:

1. **Shannon-Fano Encoding**: Used for data compression by encoding text into a binary sequence.
2. **Hamming Code (7,4)**: Used to add error detection and correction capabilities to the encoded binary sequence.

The project is divided into **six parts**, starting from analyzing text input to encoding, adding errors, and correcting errors. Below is a detailed explanation of the project.

Purpose of the Project

1. **Data Compression**: Efficiently reduce the size of text data using Shannon-Fano encoding.
2. **Error Detection and Correction**: Add robustness to the binary data using Hamming (7,4) code to handle transmission errors.

The project demonstrates a real-world workflow for compressing text data, simulating errors, and fixing errors during data transmission.

Part 1: Calculate Probabilities:

Purpose:

Analyse a text file, count the frequency of each symbol, and calculate its probability of occurrence.

Input:

- input_file: The name of the text file containing the input data.

```
input.txt - Notepad
File  Edit  Format  View  Help
Hellow world!! how are you?
```

Output:

- output_file containing each symbol, its frequency, and probability.

```
Symbol probabilities:
' ' (ASCII 32): Count = 4, Probability = 0.148148
'!' (ASCII 33): Count = 2, Probability = 0.074074
'?' (ASCII 63): Count = 1, Probability = 0.037037
'H' (ASCII 72): Count = 1, Probability = 0.037037
'a' (ASCII 97): Count = 1, Probability = 0.037037
'd' (ASCII 100): Count = 1, Probability = 0.037037
'e' (ASCII 101): Count = 2, Probability = 0.074074
'h' (ASCII 104): Count = 1, Probability = 0.037037
'l' (ASCII 108): Count = 3, Probability = 0.111111
'o' (ASCII 111): Count = 4, Probability = 0.148148
'r' (ASCII 114): Count = 2, Probability = 0.074074
'u' (ASCII 117): Count = 1, Probability = 0.037037
'w' (ASCII 119): Count = 3, Probability = 0.111111
'y' (ASCII 121): Count = 1, Probability = 0.037037
Probabilities have been calculated and saved to symbol_probabilities.txt.
symbolprobabilities text written to: symbol_probabilities.txt
```

How it Works:

Calculate the probability of each symbol using the formula:

$$\text{Probability} = \frac{\text{Count of Symbol}}{\text{Total Characters in File}}$$

Part 2: Encode Text

Purpose:

Encode the input text using the Shannon-Fano coding algorithm. Generate binary codes for each symbol based on their probabilities.

Input:

- input_file: The name of the text file containing the original text.

Output:

- output_file: Contains the text encoded as a binary sequence.

```
Contents of symbol_codes.txt:
o 00
  010
l 011
w 100
r 1010
! 1011
e 1100
? 11010
h 11011
H 11100
u 11101
d 11110
a 111110
y 111111
```

- codes_file: Contains the mapping of symbols to binary codes.

```
Contents of encoded.txt:
111001100011011001000101000010100111111010111011010110110010001011111010101100010111111001110111010
 encoded text written to: encoded.txt
```

How it Works:

1. Calculate symbol frequencies (uses Part 1 internally).
2. Use the Shannon-Fano algorithm to generate binary codes:
   - Sort symbols in descending order of probabilities.
   - Recursively split symbols into two groups with roughly equal probabilities.
   - Assign "0" to the first group and "1" to the second group at each level.
3. Replace each character in the input text with its corresponding binary code.
4. Write the encoded binary sequence to `output file`.
5. Save the symbol-to-binary mapping table in `codes_file`.

Part 3: Decode Text

Purpose:

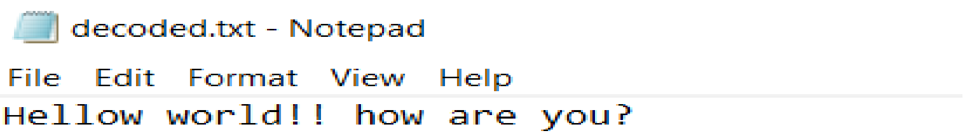
Decode the binary sequence (generated in Part 2) back into the original text using the Shannon-Fano symbol-to-code mapping.

Input:

- input_file: The file containing the binary-encoded sequence.
- codes_file: The file containing the symbol-to-binary mapping table.

Output:

- output_file: Contains the decoded original text.



How it Works:

1. Load the symbol-to-binary mapping table from codes_file.
2. Read the binary sequence from input_file bit by bit.
3. Compare the accumulated bits with the binary codes in the mapping table:
    - If a match is found, write the corresponding symbol to output_file.
    - Reset the buffer and continue matching subsequent bits.

Part 4: add hamming code

Purpose:

Add Hamming (7,4) error-correcting code to the binary sequence to prepare it for error detection and correction.

Input:

- input_file: The file containing the binary-encoded sequence.

Output:

- output_file: Contains the binary sequence with Hamming (7,4) codes applied.

```
==========================================
Step 4: Hamming Code Encoding
==========================================
Encoding binary data with Hamming (7,4) code...
Hamming Encoded Sequence:
0010110 1100110 1000011 1100110 1001100 0100101 0000000 1011010 0001111 0010110 0110011 0110011 0100101 0110011 0101010
0101010 1111111 1011010 0110011 1101001 0001111 0010110 0001111 0001111 1001100
Hamming encoding complete. Output saved to hamming_encoded.txt.
Hamming encoded binary written to: hamming_encoded.txt
```

How it Works:

1. Split the binary sequence into blocks of 4 bits.
2. For each 4-bit block, calculate 3 parity bits (P1, P2, P4) using the Hamming (7,4) rules:
   - Parity bits are calculated based on specific bit positions.
3. Combine the 4 data bits with the 3 parity bits to form a 7-bit Hamming code.
4. Write the Hamming-encoded sequence to output_file

Part 5: add errors

Purpose:

Introduce artificial errors into the Hamming-encoded binary sequence to simulate transmission errors.

Input:

- input_file: The file containing the binary sequence with Hamming codes.

Output:

- output_file: Contains the binary sequence with bit errors.

```
Contents of error_file.txt:
0010110111011010000011100110110110001000010000000000110100000111001011101100110100011010011
1011001100010100101110111111100110100111011110100000011110000110000110100011111101100
Binary with errors written to: error_file.txt
```

How it Works:

1. Read the binary sequence from input_file.
2. Introduce errors by flipping bits (changing 0 to 1 or 1 to 0) at fixed intervals (e.g., every 10th bit).
3. Write the modified sequence to output_file.

Part 6: DecodeHammingCodeAndFixErrors

Purpose:

Detect and correct single-bit errors in the Hamming-encoded binary sequence using the Hamming (7,4) error correction method.

Input:

- input_file: The file containing the binary sequence with errors.

Output:

- output_file: Contains the corrected binary sequence (errors fixed).

```
Contents of corrected_file.txt:
1110011000110110010001010000101001111110101110110101101100100010111110101011000101111110011101110100
Corrected binary written to: corrected_file.txt
```

How it Works:

1. Read the binary sequence in blocks of 7 bits (Hamming-encoded words).
2. Recalculate the parity bits (P1, P2, P4) for each block and compare them with the original parity bits.
3. If there's a mismatch, determine the position of the erroneous bit using the parity bits.
4. Correct the erroneous bit by flipping it (from 0 to 1 or vice versa).
5. Extract the original 4 data bits from each corrected 7-bit block.
6. Write the corrected sequence to output_file.

Real-World Applications

1. **Data Compression**: Shannon-Fano encoding reduces file sizes for efficient storage and transmission.
2. **Error Correction**: Hamming codes ensure reliable data transmission in communication systems.
3. **Simulation of Transmission Errors**: Useful for testing error detection and correction algorithms.