ELEN090-2 Information And Coding Theory Project 2: Source coding, data compression and channel coding

Sébastien Laurent (s201561) - Duy Vu Dinh (s2401627)

1 Implementation

1.1 Question 1

Explanation of implementation

The main steps of our implementation of the binary Huffman coding for a given probability distribution are shown below:

1. `Node` class: A `Node` class is defined to represent each node in the Huffman tree. Each node contains:
    - `symbol` represents the actual character,
    - `freq` represents the probability of the symbol,
    - `left` and `right` represent pointers to child nodes representing the '0' and '1' branches of the Huffman tree.
2. `build_huffman_tree`: A priority queue is used to build the Huffman tree efficiently. In each iteration, the two nodes with the smallest probabilities are merged into a new internal node, whose probability is the sum of the two children.
3. `generate_huffman_codes`: Once the tree is built, it is traversed recursively. The traversal assigns a binary digit to each branch: `'0'` for the left child and `'1'` for the right child. A codeword is generated for each symbol as the path from the root to its corresponding leaf.

Example verification (Exercise 5, TP2)

We verified our implementation using Exercise 5 from TP2. The source is memoryless and emits six different symbols with the following probability distribution: $P(S) = [0.05, 0.10, 0.15, 0.15, 0.20, 0.35]$ with symbols `A, B, C, D, E, F`, respectively.

The Huffman tree is shown in Figure 1 and the resulting Huffman codebook is:

```
{'A': '000', 'B': '001', 'E': '01', 'C': '100', 'D': '101', 'F': '11'}
```

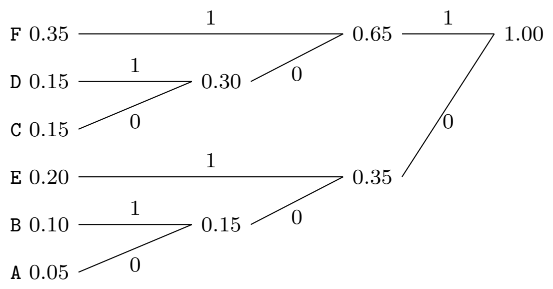


Figure 1: Binary Huffman tree for $P(S)$ built by our implementation.

Although the Huffman code obtained from our implementation (Figure 1) is not identical to the one from the TP2 session, this is expected: when multiple pairs of nodes have the same lowest probabilities, different merge orders can lead to different (but still optimal) Huffman trees. All such results are valid as long as:

- The code is prefix-free,
- The average codeword length is minimal.

Our code satisfies both properties, and the constructed tree in Figure 1 illustrates one such valid optimal coding.

Extend the function to generate a Huffman code of any (output) alphabet size

- `Node` class: Instead of using binary `left` and `right` children, we define a `children` list attribute in each node. This allows internal nodes to have up to D children ($D \geq 2$), depending on the arity of the output alphabet.
- `build_huffman_tree`: In the standard binary Huffman algorithm, the two nodes with the smallest probabilities are merged at each step. In the generalized version, we merge the D lowest-probability nodes instead. This continues until only one root node remains.
  - However, for $D > 2$, not all input sets naturally form a complete D-ary tree. To guarantee correctness and optimality, zero-probability *dummy symbols* will be added to ensure that the total number of nodes satisfies the required condition:

    $$(\text{Total number of symbols} - 1) \mod (D-1) = 0$$

    This condition ensures that all internal nodes can have exactly D children.

- `generate_huffman_codes`: During the tree traversal, each child is assigned a digit between 0 and $D-1$, constructing codewords in base D (instead of binary). The code generation is recursive, just like in the binary case.

Using the same probability distribution from Exercise TP2, and setting $D = 3$ (ternary Huffman code), our implementation produces the following codebook:

```
{'E': '0', 'F': '1', 'C': '20', 'D': '21', 'A': '221', 'B': '222'}
```

The Huffman tree for ternary code is shown in Figure 2.

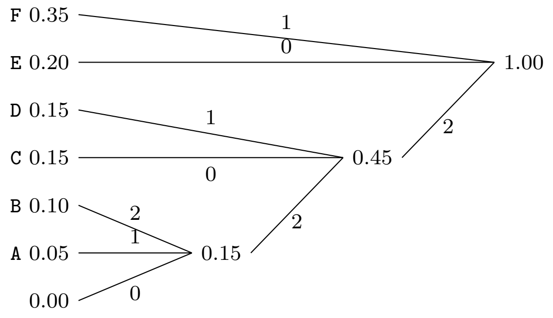


Figure 2: Ternary Huffman tree for $P(S)$ built by our extended implementation.

1.2 Question 2

We applied our implementation to the binary sequence: $T =$ `1011010100010`

The resulting encoded sequence is: $U =$ `100011101100001000010`

The dictionary entries created during the process are listed in Table 1.

Table 1: on-line Lempel-Ziv dictionary

Word	**Index**	**Encoded** (binary address + last bit)
λ	0	
1	1	1
0	2	00
11	3	011
01	4	101
010	5	1000
00	6	0100
10	7	0010

1.3 Question 3

Comparison between basic and on-line Lempel-Ziv algorithm

Both algorithms build a dictionary incrementally and emit output as address-symbol pairs. However, they differ in how the address (i.e., index of the prefix) is encoded and how adaptively they respond to the growth of the dictionary. While the basic algorithm uses a fixed-length or predefined number of bits for all prefix addresses, the on-line one uses a dynamic number of bits: $\lceil \log_2(n) \rceil$, where n is the dictionary size.

- **Codeword representation**
  - Basic: Index is encoded using a fixed bit-length (e.g., 4, 6, 8 bits).
  - On-line: Index uses the minimum required bit-length based on dictionary size.
- **Bit-length behavior**
  - Basic: Remains constant throughout the encoding.
  - On-line: Grows logarithmically with dictionary size.

Basic Lempel-Ziv algorithm

- Advantages:
  - Suitable for offline compression where the dictionary size is known or fixed.
  - Allows for easier decoding when bit-length is uniform.
- Drawbacks:
  - Inefficient in early stages when the dictionary is small, fixed address length leads to wasted bits.
  - Less adaptive to varying or unknown data sources.

On-line Lempel-Ziv algorithm

- Advantages:
  - Dynamically adjusts address size, improving compression in early phases.
  - Robust and effective for real-time and streaming applications.
  - No need to know dictionary size in advance.
- Drawbacks:
  - Slightly more complex to implement due to bit-length recalculation.
  - Can lead to variable-length codewords, making decoding more complex.
  - May reach optimal compression only for very long inputs, as asymptotic performance is achieved when the dictionary contains a significant fraction of sufficiently long typical messages.

1.4 Question 4

Replication of example

We implemented the LZ77 compression algorithm using a sliding window approach, as described in the theoretical course and visualized in Figure 2 in the statement. Our implementation supports customizable window sizes and accurately performs prefix matching between the search buffer and look-ahead buffer.

Using our implementation with:

- Source sequence: `abracadabrad`
- Window size: 7

we obtain the following encoded sequence:

```
(0,0,a)(0,0,b)(0,0,r)(3,1,c)(2,1,d)(7,4,d)
```

This output matches the structure expected from the example in that Figure 2. Each tuple (d, p, c) represents:

- d: the offset (i.e., how far back in the window to start the match),
- p: the length of the matched prefix,
- c: the next unmatched symbol following the prefix.

The encoding identifies repeated substrings in the already-seen portion of the input, and replaces them with backward references plus one new symbol. The match $(7, 4, d)$ at the end represents the repeated substring `abra` (length 4), followed by `d`.

Decoding principle of LZ77

The LZ77 decoding process works by reversing the encoded triples (d, p, c) as follows:

1. For each triple, go back d positions from the current end of the decoded sequence.
2. Copy p characters starting from that position.
3. Append the new character c directly.

This process is guaranteed to reconstruct the original input exactly because each codeword refers only to previously decoded symbols. The implicit dictionary (the already-decoded text) grows with each step, just as the encoder's sliding window does.

The decoder thus maintains a buffer of decoded symbols and updates it iteratively by resolving each backward reference and new character, fully reconstructing the original message.

2 Source coding and reversible (lossless) data compression

In the following questions, the compression rate is computed as follows

$$\text{Compression rate} = \frac{\text{Original length}}{\text{Encoded length}} \cdot \frac{\log_2(\text{Original alphabet size})}{\log_2(2)}$$

2.1 Question 5

The total length of the encoded English text is 239,008 bits and the compression rate is equal to 1.16.

2.2 Question 6

The expected average length is equal to 4.10 bits and the empirical average length is also equal to 4.10 bits. This was expected.

Let us define:

- n_i the number of symbols i in the English text
- p_i the marginal probability we computed for the symbol i in question 5
- $|i|$ the number of bits composing the codeword of the symbol i
- $\mathcal{A}$ the alphabet of the English text
- T the number of symbols composing the English text
- T_E the number of bits composing the encoded English text
- μ_E the expected average length
- μ_P the empirical average length

We computed p_i in question 5 in the following way

$$p_i = \frac{n_i}{\sum_{i \in \mathcal{A}} n_i} = \frac{n_i}{T}$$

The expected average length and empirical average length are respectively computed in the following ways

$$\mu_E = \sum_{i \in \mathcal{A}} p_i \cdot |i|$$

$$\mu_P = \frac{T_E}{T}$$

We notice that $T_E = \sum_{i\in\mathcal{A}} n_i \cdot |i|$, so

$$\mu_P = \frac{T_E}{T} = \frac{\sum_{i\in\mathcal{A}} n_i \cdot |i|}{T} = \sum_{i\in\mathcal{A}} \frac{n_i}{T} \cdot |i| = \sum_{i\in\mathcal{A}} p_i \cdot |i| = \mu_E$$

We can conclude that the expected average length is exactly equal to the empirical average length, because we compute the marginal probability distribution using the English text.

Now, if we define our source S using the probability distribution computed in question 5, we can verify the theoretical bounds for optimal codes

$$\frac{H(S)}{\log_2(q)} \leq \mu_E < \frac{H(S)}{\log_2(q)} + 1$$

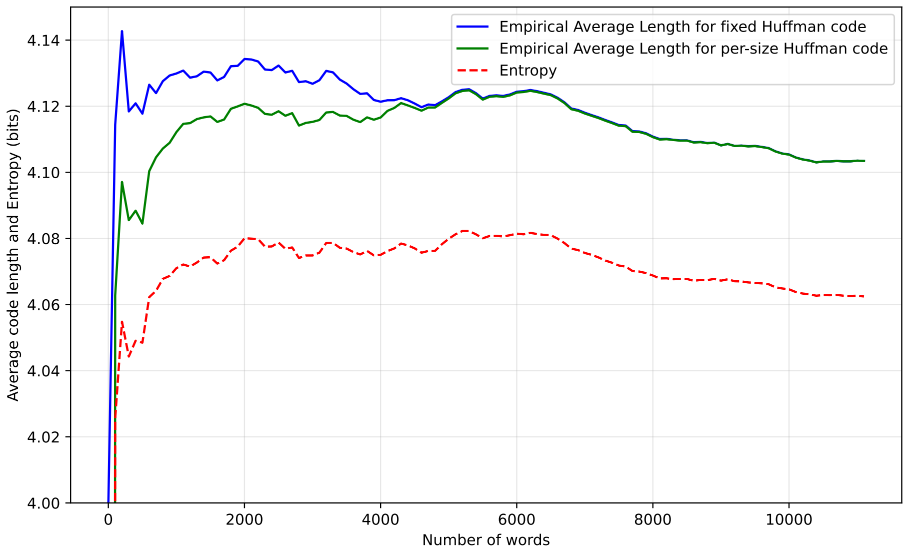
where $q = 2$, $H(S) = 4.06$. And indeed

$$4.06 \leq 4.10 < 5.06$$

This was expected, because Huffman coding is an optimal coding strategy.

2.3 Question 7

We plot the empirical average length of encoded text subparts using a Huffman code computed on the entire English text (in blue), the empirical average length using Huffman codes computed on individual text subparts (in green), and the entropy of the marginal probability distribution corresponding to text subparts (in red). In this case, because the alphabet size of the encoded message is equal to 2, the theoretical lower bound for the average length is equal to the entropy of the text subpart.



As expected, both empirical average lengths are always greater than the entropy. Additionally, the green curve is consistently lower than the blue curve. This is also expected, since the Huffman codes used to draw the green curve are computed to be optimal for the specific text subparts, whereas the Huffman code used to draw the blue curve is optimized for the full text. It is therefore logical that the green curve converges to the blue curve as the size of the text subparts increases and approaches the size of the full text. Finally, we can clearly see that there is a correlation between the entropy and both empirical average lengths.

2.4 Question 8

The on-line Lempel-Ziv algorithm applied on the English text produces a mix of binary symbols and non-binary symbols (the alphabet and the space character). To convert all non-binary symbols to binary symbols, we decide to adopt a very simple encoding of non-binary symbols:

Non-binary symbol	**5-bit Binary Encoding**
SPACE	00000
a	00001
b	00010
...	...
y	11001
z	11010

We decided to encode each symbol using 5 bits, because we have 27 different symbols and we know that $2^4 < 27 < 2^5$. This might not be optimal, but this strategy ensures that codes remain decodable and we will explore more complex encoding of the non-binary symbols in the following questions. The encoded text length is equal to 217,013 bits and the compression rate is equal to 1.28.

2.5 Question 9

The LZ77 algorithm produces triplets of non-binary symbols:

1. The first part of the triplets represents the distance to the start of the prefix. If we denote by m the maximum value (with $m \leq$ `window_size`) taken by this part across all triplets in the message, then it can be encoded using $\lfloor \log_2 m \rfloor + 1$ bits. This number m must be known by the receiver or sent along with the message. In our case, the size of the binary representation of the first part is equal to 3 bits.

2. The second part of the triplets is the length of the prefix. If we denote by m the maximum value (with $m <$ Message length) taken by this part across all triplets in the message, then it can be encoded using $\lfloor \log_2 m \rfloor + 1$ bits. This number m must be known by the receiver or sent along with the message. In our case, the size of the binary representation of the second part is equal to 3 bits.

3. The third part of the triplets is a symbol from the English text. It can be encoded in the same way as described in the previous question.

This encoding might not be optimal, but this strategy ensures that codes remain decodable and we will explore more complex encoding of the non-binary symbols in the following questions. The length of the encoded text is 441,320 bits and the compression rate is 0.63 (indicating the opposite effect of compression).

2.6 Question 10

To combine the LZ77 algorithm with the Huffman algorithm, we propose two solutions:

- Applying the LZ77 algorithm followed by the Huffman algorithm. This is the approach used in the DEFLATE algorithm. The LZ77 algorithm eliminates redundancy in the source text by replacing repeated patterns with references, and the Huffman algorithm then efficiently encodes the resulting non-binary symbols into binary symbols.

- Applying the Huffman algorithm followed by the LZ77 algorithm. This is not the best approach for multiple reasons:

    1. After applying the LZ77 algorithm, the distance to the start of the prefixes and the lengths of the prefixes in the triplets are still not encoded as binary symbols, so they must be further encoded into binary form.

2. After applying the Huffman algorithm, the same redundancies from the original text still exist in the encoded text, but they require a larger `window_length` to be detected. This is because each symbol of the original text is now represented by a sequence of multiple bits. This results in larger values for the distance to the start of the prefixes and the lengths of the prefixes, which in turn require longer binary representations.

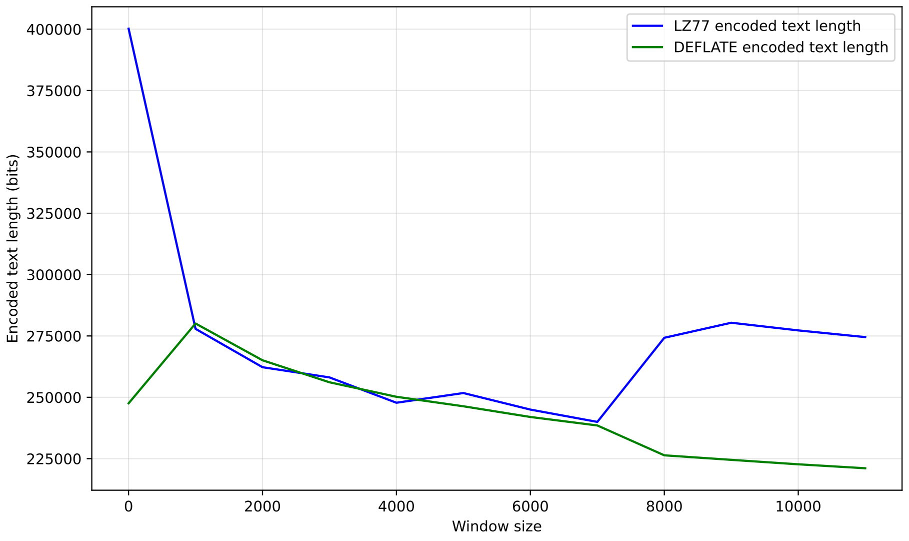
We will implement the first approach that we described for the following questions.

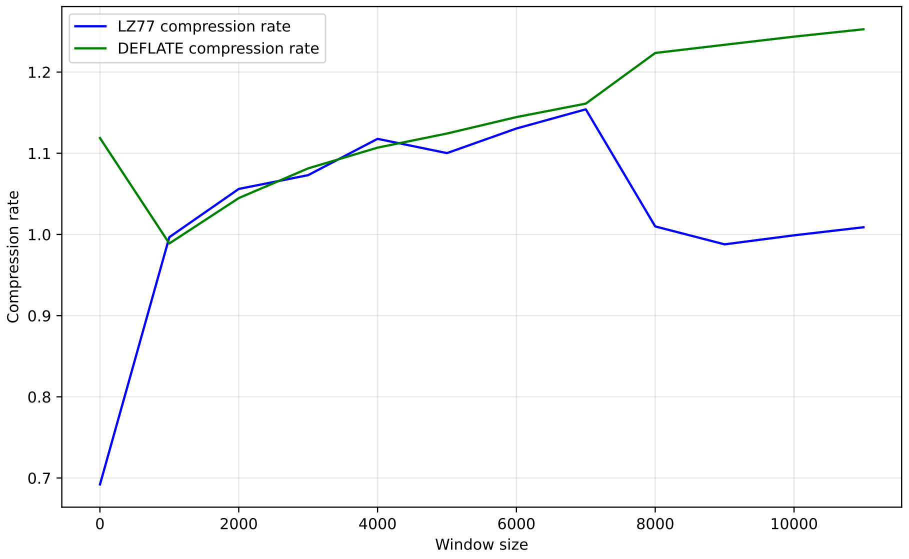
2.7 Question 11

For this question, we apply the LZ77 algorithm followed by the Huffman algorithm, which is computed based on the marginal probability distribution of the characters in the triplets. We choose to ignore zeros in the first and second parts of the triplets when computing and applying the Huffman code. In other words, instead of encoding `(0,0,c)`, we directly encode `c`. This approach works (codes remain decodable) because the Huffman code is instantaneous, and because the set of symbols in the first and second parts of the triplets is disjoint from the set of symbols in the third part, thanks to the fact that there are no integers in the English text. If there were integers in the text, we would have to distinguish between integers originating from the English text and those in the first and second parts of the triplets by assigning them different codes. The length of the encoded text is 322,199 bits and the compression rate is 0.86 (indicating the opposite effect of compression).

2.8 Question 12

We plot the lengths and compression rates of messages encoded using either LZ77 or the combination of LZ77 and Huffman coding, which we call DEFLATE.





2.8.1 LZ77 performance analysis

The compression rate of the LZ77 algorithm starts at a low value of around 0.7 and increases with the window size up to 7000. When the window size is too small, the algorithm causes overhead due to the triplet format of its output, without being able to eliminate much redundancy. When the window size becomes larger than 7000, the size of the encoding of the first and second parts of the triplet keeps increasing, without the algorithm being able to eliminate much more redundancy than with a lower window size, which makes the compression rate worse. This is due to the fact that we use a very simple encoding, with its size scaling with the maximum value taken by the first and second parts of the triplets. We will not encounter this kind of problem with the Huffman encoding used in the DEFLATE algorithm.

2.8.2 DEFLATE performance analysis

The compression rate of the DEFLATE algorithm starts at a value of around 1.12, before dropping to a value of around 0.99 for a window size of 1000. When the window size is equal to 1, there is a large number of triplets with zeros as their first and second parts, so this makes the optimization made in question 11 very useful (we encode `c` instead of `(0,0,c)`). However, when the window size is equal to 1000, there are far fewer triplets with zeros as their first and second parts compared to when the window size is equal to 1, so we just get the expected poor performance of LZ77 for "small" window sizes. In fact, if all triplets had zeros as their first and second parts, we would get the performance of just using Huffman encoding (a compression rate of 1.16). After a window size of 1000, the compression rate monotonically increases, because the LZ77 algorithm succeeds in eliminating more redundancies.

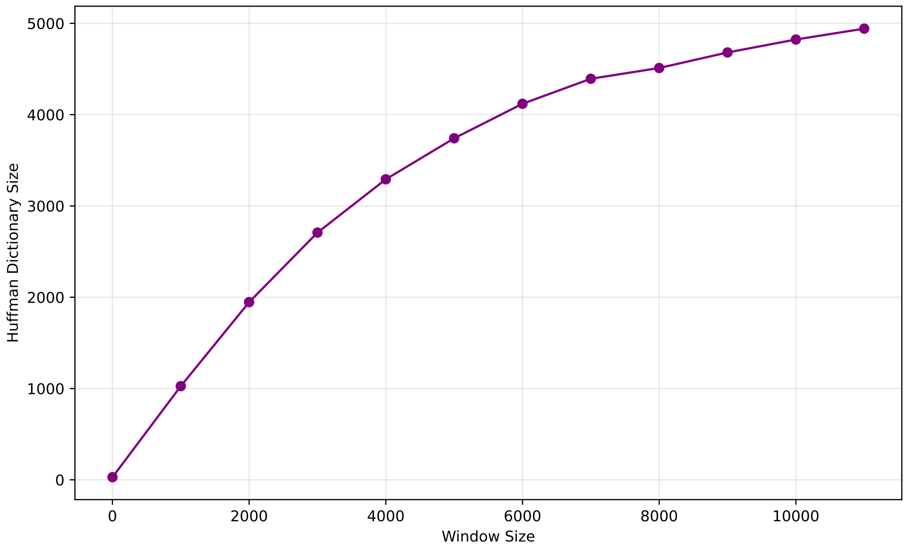
2.8.3 Comparison between algorithms

Algorithm	**Length (bits)**	**Compression rate**
LZ77	239,976	1.15
DEFLATE	221,090	1.25
On-line LZ	217,013	1.28

Table 2: Length and compression rate of the encoded message using LZ77, DEFLATE, and On-line LZ. For LZ77 and DEFLATE, we chose the performance of the best window size.

As we can see, the DEFLATE algorithm performs better than the LZ77 algorithm, and the On-line LZ algorithm performs better than the DEFLATE algorithm.

The DEFLATE algorithm performs better than the LZ77 algorithm, thanks to the improved encoding of binary symbols provided by the Huffman encoding. However, because the DEFLATE algorithm uses Huffman encoding, the Huffman tree must be sent to the receiver along with the encoded message, so that the receiver is able to properly decode the message. For a window size of 11000, the size of the dictionary of the Huffman code is around 5000. For the sake of simplicity, let us say that each key of the dictionary is an 8-bit character (This is not possible, but we keep things simple), then the Huffman encoding adds, at least, an overhead of $5000 \cdot 8 = 40000$ bits, which already makes it worse than basic LZ77. This is the reason why the RFC of the DEFLATE algorithm allows the possibility of encoding the output of the Lempel-Ziv algorithm using a fixed Huffman code defined in the RFC.



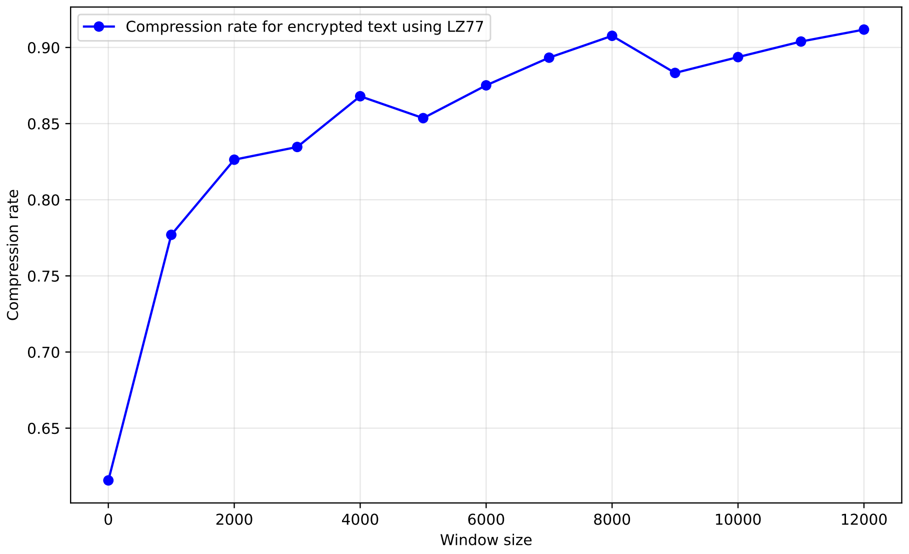
Moreover, the on-line LZ algorithm outperforms both the LZ77 and DEFLATE algorithms while being significantly faster. This is because it eliminates redundancy across the entire text, while not relying on the triplet format used in LZ77, which introduces considerable overhead.

2.9 Question 13

Because the online LZ algorithm is able to eliminate redundancy across the entire text, it is best suited for text encoding (in comparison to other algorithms), where repetitions are assumed to occur over long distances. For this question, we combine the online LZ algorithm with Huffman encoding to determine whether this leads to improved compression rates. Indeed, we achieve superior performance: an encoded text length of 209,170 and a compression rate of 1.32. Moreover, although the Huffman tree must be transmitted along with the encoded message, the dictionary of the Huffman encoding contains only 27 symbols (the symbols of the English text).

2.10 Question 14

The compression rate for the Huffman code on the encrypted text is 1.0, and we plot the compression rate of the LZ77 algorithm with respect to window size.



2.11 Question 15

Algorithms	Compression rate	
	Text	Encrypted text
Huffman code	1.16	1.0
LZ77	1.15	0.91

Table 3: Compression rate of the English text and encrypted text using both Huffman encoding and the LZ77 algorithm. For the LZ77 algorithm, we chose the performance of the best window size.

As observed, both Huffman encoding and the LZ77 algorithm achieve significantly better compression rates on the English text compared to the encrypted text. Encryption increases the entropy of the text (the entropy of the text is 4.06, whereas that of the encrypted text is 4.68), making it less predictable. This makes the task of compression more difficult. As the complexity of the key increases (through factors such as key size, the number of different characters, and the irregularity of patterns in the key), the encrypted text becomes increasingly unpredictable, entropy rises, and the task of compression becomes harder. Therefore, we can expect worse compression rates as key complexity increases.

2.12 Question 16

Encryption makes compression worse while leaving the size of the text unchanged. Therefore, we advise Alice to first compress the text and then encrypt the encoded text. This way, she will benefit from a better compression rate than if she were to encrypt the text first.

3 Channel coding

3.1 Question 17

The function performs the following steps:

- `img = Image.open("image.jpg")`: The image is loaded using the `Pillow` library.
- `img_gray = img.convert("L")`: It is immediately converted to grayscale using mode `"L"`, which maps each pixel to a single intensity value in the range $[0, 255]$.
- `img_array = np.array(img_gray, dtype=np.uint8)`: The grayscale image is then transformed into a two-dimensional NumPy array of type `uint8`, preserving both pixel precision and spatial resolution. This array is returned by the function.

Additionally, an optional `display` flag enables visualization of the image using `matplotlib`, with the grayscale colormap applied and pixel intensities explicitly scaled between 0 (black) and 255 (white). The Figure 3 is the original image.

The resulting NumPy array has a resolution of 1365×2048 pixels. Each pixel is represented by an integer value between 0 and 255, corresponding to its grayscale intensity.

3.2 Question 18

Each pixel in the grayscale image takes an integer value in the range $[0, 255]$, corresponding to $2^8 = 256$ distinct levels of intensity. These values can be interpreted as 256 unique symbols from a finite alphabet.

Because the task requires a fixed-length binary encoding, all symbols must be represented using codewords of equal length. The smallest integer k such that $2^k \geq 256$ is $k = \log_2(256) = 8$, meaning that 8 bits are both necessary and sufficient to represent all possible grayscale values without ambiguity or redundancy. Any encoding using fewer than 8 bits would fail to represent some pixel values, while using more would introduce unnecessary overhead. Thus, 8-bit fixed-length encoding is optimal in this context.

The encoding follows natural binary order, as shown in Table 4.

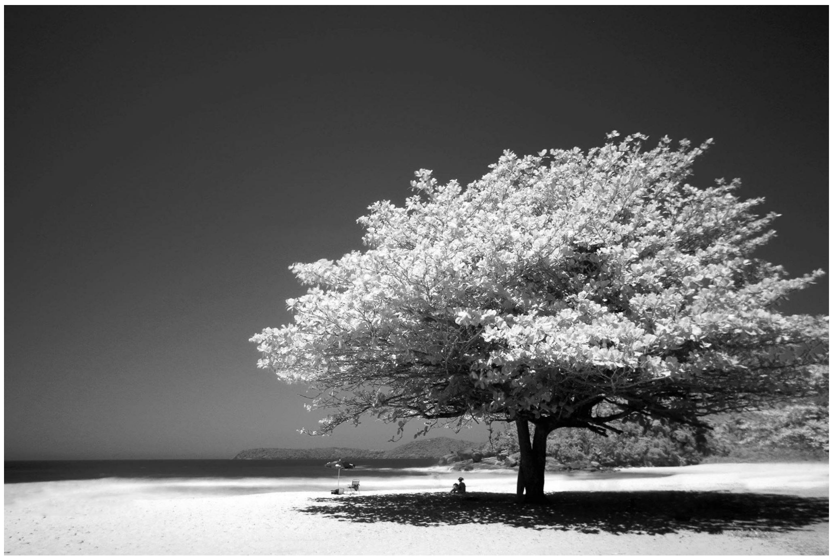


Figure 3: Original grayscale image.

Table 4: Fixed-length binary encoding of grayscale pixel values using 8-bit codewords.

Symbol	**8-bit encode**
0	`00000000`
1	`00000001`
2	`00000010`
⋮	⋮
254	`11111110`
255	`11111111`

Let the image have dimensions $H \times W$, where H is the height and W is the width. Since each pixel is represented by exactly 8 bits, the total number of bits required to encode the entire image is:

$$\text{Total bits} = 8 \times H \times W$$

In our case, the image size is 1365×2048, resulting in:

$$8 \times 1365 \times 2048 = 22{,}364{,}160 \text{ bits.}$$

In the implementation, the image array is first flattened into a one-dimensional array of pixel values. Each pixel is then encoded using `np.binary_repr(value, width=8)`, which ensures uniform 8-bit binary strings.

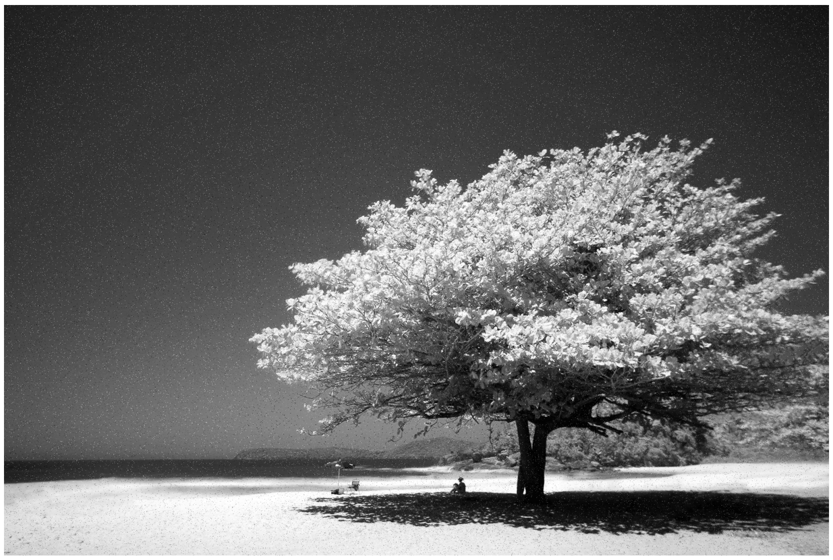


Figure 4: Decoded image after BSC simulation with $p = 0.01$.

3.3 Question 19

We simulated the transmission of the encoded binary image signal through a Binary Symmetric Channel (BSC) with a bit-flip probability of $p = 0.01$. This models the effect of a noisy communication channel in which each bit has a 1% chance of being independently flipped from 0 to 1 or vice versa.

Interpretation:

- A bit '0' is received as '0' with 99% probability and as '1' with 1% probability.
- A bit '1' is received as '1' with 99% probability and as '0' with 1% probability.

Observations: Figure 4 shows the decoded image. Although the bit-flip probability is low, each pixel is composed of 8 bits, so the probability that *at least one bit* is flipped in a pixel is approximately:

$$1 - (1 - p)^8 = 1 - 0.99^8 \approx 7.7\%$$

This theoretical estimate aligns well with the observed results:

- Bit error rate: 1.00%
- Pixel error rate: 7.71%

Thus, about 7.7% of the pixels are affected. These changes appear as mild, randomly distributed noise—small intensity shifts that do not significantly distort the overall structure. There are no burst errors or structured artifacts, as the BSC introduces independent noise. While the image remains easily recognizable, fine details may appear slightly degraded.

3.4 Question 20

The Hamming (7,4) code transforms each block of 4 signal bits $s_1s_2s_3s_4$ into a 7-bit codeword $s_1s_2s_3s_4p_1p_2p_3$ by adding 3 parity bits (p_1, p_2, p_3) to enable error detection and correction. The parity bits are computed as follows:

- $p_1 = (s_1 + s_2 + s_4) \bmod 2$
- $p_2 = (s_1 + s_3 + s_4) \bmod 2$
- $p_3 = (s_2 + s_3 + s_4) \bmod 2$

In our case, each pixel value is originally represented using an 8-bit fixed-length binary code. To apply Hamming (7,4) encoding, we split each 8-bit pixel into two 4-bit blocks. Each of these is then independently encoded into a 7-bit Hamming codeword, resulting in a total of 14 bits per symbol.

For example, Table 5 shows the encoding of the first three symbols from the image:

Table 5: Hamming (7,4) encoding applied to the first three symbols of the signal.

Position	**Pixel**	**8-bit binary encode**	**Hamming (7,4) encoded**
1	31	00011111	0001111 1111111
2	31	00011111	0001111 1111111
3	32	00100000	0010011 0000000
Total		24 bits	42 bits

Since each 4-bit block is expanded into 7 bits, the encoded sequence length increases from 22,364,160 to 39,137,280 bits.

3.5 Question 21

To improve transmission reliability, the binary image signal is encoded using the Hamming (7,4) error-correcting code, which introduces redundancy by mapping each 4-bit block $s_1s_2s_3s_4$ to a 7-bit codeword that includes 3 parity bits p_1, p_2, p_3. These parity bits are computed such that each one covers a subset of the data bits. The relationships are illustrated using a Venn diagram in Figure 5.

We then simulated the effect of a BSC with a bit-flip probability of $p = 0.01$ on the Hamming-encoded sequence. After receiving the corrupted signal, we applied a syndrome-based decoding procedure that detects and corrects single-bit errors in each 7-bit block. Because Hamming code can only correct single-bit errors, we assume the error is limited to one bit, as this is the most probable case. The syndrome can be computed as follows

$$\text{Syndrome} = (p_R + p_C) \bmod 2$$

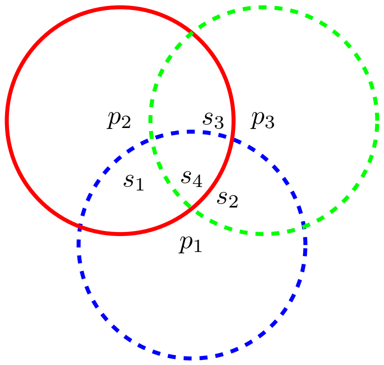


Figure 5: Venn diagram representation of the Hamming (7,4) code where parity bits p_1, p_2, and p_3 cover different subsets of data bits s_1 through s_4.

Where p_R are the received parity bits, p_C are the parity bits computed using the received data bits, and where the addition and the modulo are bitwise operations. If the syndrome is equal to zero, we assume that there is no error in the 7-bit block, as no single-bit error can result in a zero syndrome. If the syndrome has only one bit equal to 1, we assume that the error occurred on the corresponding parity bit, as no single-bit error on the four data bits can produce such a syndrome, so no correction is needed for the data bits. Finally, if the syndrome contains two or three bits equal to 1, we apply the following rules:

- If the syndrome is equal to $(1, 1, 0)$, it indicates that the error is on s_1, as it is the only bit common in the formulas for both p_1 and p_2. We need to flip s_1 to correct the error.
- If the syndrome is equal to $(1, 0, 1)$, the error is on s_2, since it is the only bit shared between the formulas for p_1 and p_3. We need to flip s_2 to correct the error.
- If the syndrome is equal to $(0, 1, 1)$, the error is on s_3, as it is the only bit common in the formulas for p_2 and p_3. We need to flip s_3 to correct the error.
- If the syndrome is equal to $(1, 1, 1)$, the error is on s_4, as it is the only bit present in all three parity bit formulas. We need to flip s_4 to correct the error.

After decoding the data bits and reconstructing the image (Figure 6), we compared the result with the original to evaluate the effectiveness of error correction. The measured error rates are:

- Bit error rate after correction: 0.09%
- Pixel error rate after correction: 0.41%

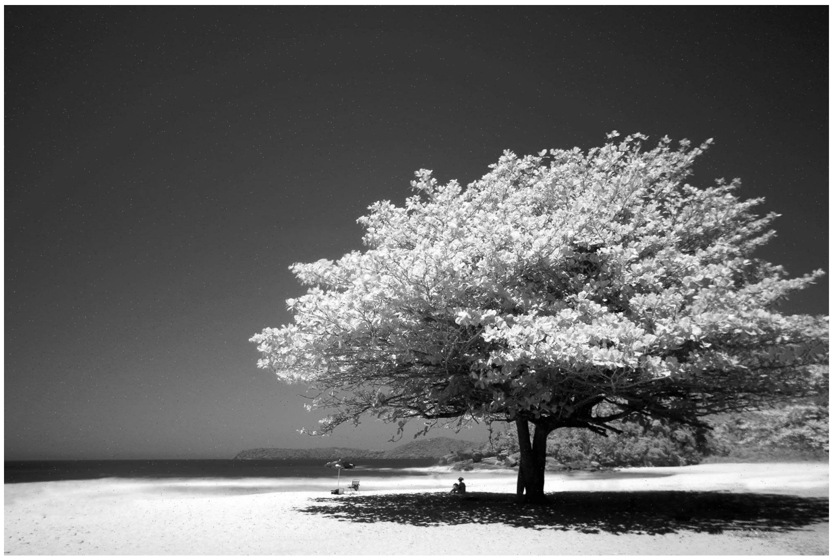


Figure 6: Decoded image after BSC simulation with $p = 0.01$ using Hamming (7,4) encode.

Observations: Compared to the results from Question 19 (1.00% bit error rate and 7.71% pixel error rate), the use of Hamming (7,4) significantly improved signal integrity. Most single-bit errors introduced by the BSC were corrected. The remaining minor pixel-level distortion likely stems from the multi-bit errors within a single 7-bit block, which cannot be corrected by Hamming (7,4). In other words, if at least two bits are flipped in the same 7-bit codeword, the resulting syndrome could be invalid or misleading. The visual quality of the image after decoding is near-original, and most structural and fine details are preserved.

3.6 Question 22

To further reduce information loss and improve the communication rate, we propose two complementary strategies: applying lossless compression before channel encoding, and using longer block codes with improved error protection efficiency.

Apply lossless compression before encoding

Compressing the binary image signal using methods like Huffman or LZ77 reduces data size without affecting quality. This allows for more efficient use of bandwidth and creates space to apply stronger error correction if needed.

Justification: Compression reduces redundancy, increasing the effective data rate. It is especially useful when combined with error-correcting codes, as fewer bits need protection.

Use longer and more efficient block codes

Instead of relying on short codes like Hamming (7,4), which introduces a large amount of overhead (75% increase), more efficient error-correcting codes with longer block lengths should be considered. Examples include:

- Hamming (15,11): corrects single-bit errors with only 4 parity bits per 11 data bits.
- BCH (31,26): corrects up to two errors per 31-bit codeword.

Instead of Hamming (7,4), codes like Hamming (15,11) or BCH (31,26) might offer better protection with less relative overhead. These longer codes correct more errors while maintaining a higher data-to-parity ratio.

Justification: Longer block codes benefit from better coding efficiency due to their lower relative overhead. They also enable more accurate syndrome decoding and are better suited for scenarios with burst errors or where stronger protection is needed. Although they may introduce slightly higher decoding latency, this is acceptable in many image transmission settings where throughput and integrity are prioritized.

Therefore, by first compressing the data and then encoding it with a more efficient block code, the system can significantly reduce both the amount of data transmitted and the likelihood of visual degradation due to noise. These strategies offer a balanced trade-off between robustness and communication efficiency.