

### **Task 3: Coding**



**Image 1** - *Medical CT slice used as test image in both the C# GUI and the Python web app.*



**Image 2** - *Forest path*

## **Python results**

# Algorithm Comparison

Comparison complete!

## Comparison Results

	Algorithm	Original Size (bytes)	Compressed Size (bytes)	Compression Ratio	Space Saving (%)	Execution Time (s)	PSNR (dB)
0	RLE	1048576	453154	2.314	56.7839	0.8995	∞ dB
1	Huffman	1048576	406444	2.5799	61.2385	4.3368	∞ dB
2	Arithmetic	1048576	312481	3.3556	70.1995	23.3861	∞ dB
3	CABAC	1048576	5	209715.2	99.9995	60.4751	5.00 dB

**Table 1** - Compression results from the Python web app for the medical image (Image 1).

# Algorithm Comparison

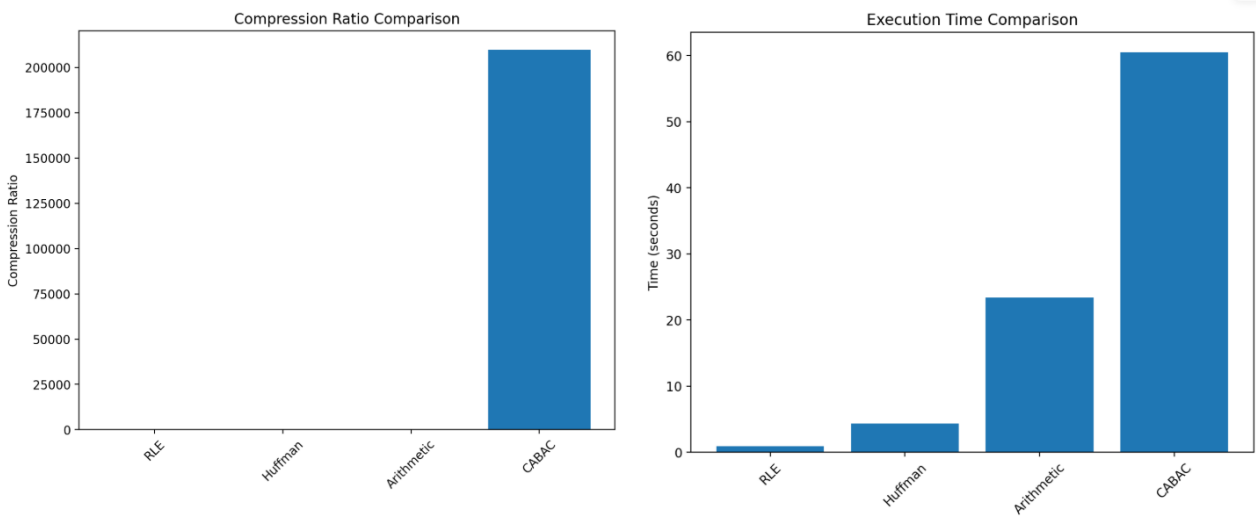
Comparison complete!

## Comparison Results

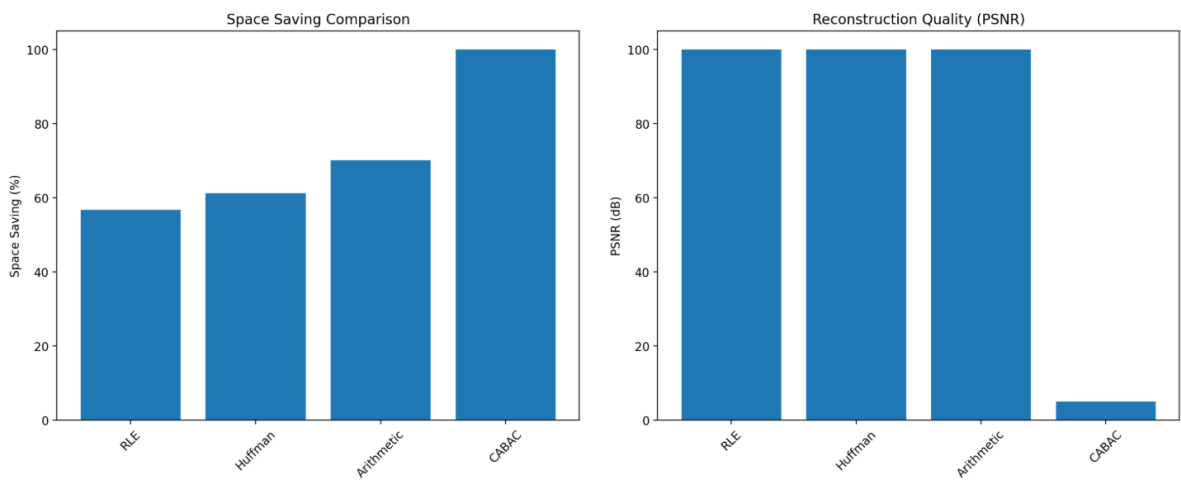
	Algorithm	Original Size (bytes)	Compressed Size (bytes)	Compression Ratio	Space Saving (%)	Execution Time (s)	PSNR (dB)
0	RLE	749088	1424786	0.5258	-90.2028	1.2946	∞ dB
1	Huffman	749088	661110.5	1.1331	11.7446	6.8167	∞ dB
2	Arithmetic	749088	631095	1.187	15.7516	36.8505	∞ dB
3	CABAC	749088	5	149817.6	99.9993	42.5176	11.30 dB

**Table 2** - Compression results from the Python web app for Image 2

Performance Visualizations

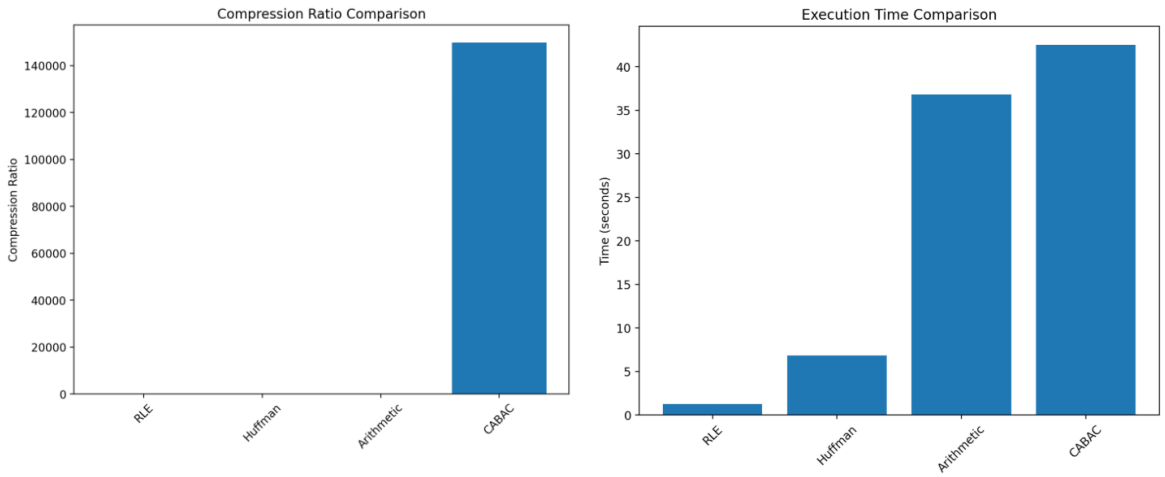


**Fig.1** Compression ratio (left) and execution time (right) for Image 1 in the Python web app.

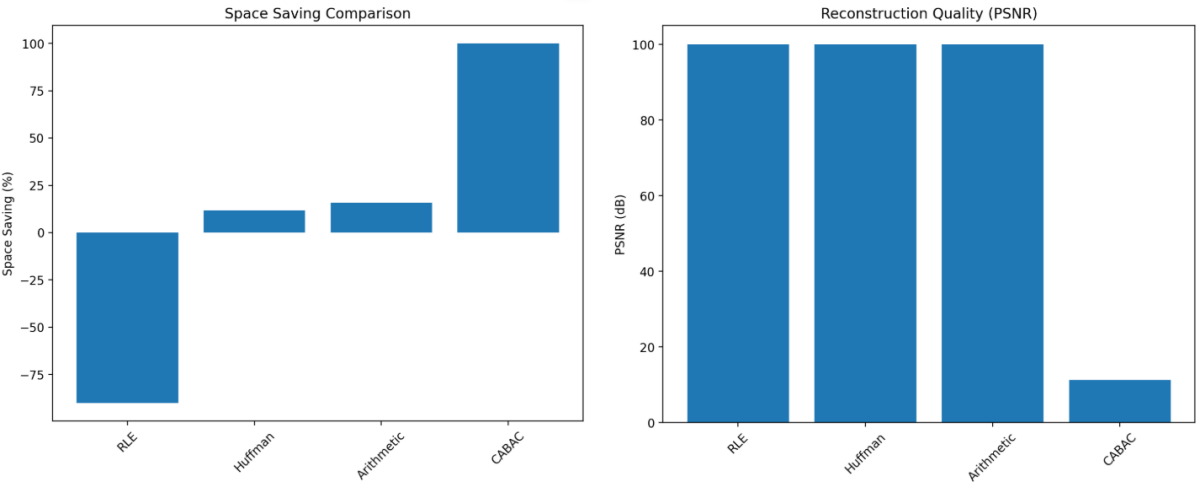


**Fig.2** Space saving (left) and PSNR (right) for Image 1 in the Python web app.

Performance Visualizations



**Fig.3** Compression ratio (left) and execution time (right) for Image 2 in the Python web app.



**Fig.4** Space saving (left) and PSNR (right) for Image 2 in the Python web app.

## C# results

■ Compression Algorithms Visualizer - Task 3

All Algorithms ▾ Load Image Compress Compare All

Single Algorithm Comparison

Algorithm	Compressed Size	Compression Ratio	Space Saving	Time (s)	PSNR
RLE	453,258 bytes	2.31:1	56.77%	0.3866	∞ dB
Huffman	327,906 bytes	3.20:1	68.73%	0.5351	∞ dB
Arithmetic	312,480 bytes	3.36:1	70.20%	1.2379	∞ dB
CABAC	5 bytes	209715.20:1	100.00%	0.7068	8.67 dB

**Table 3** - Compression results from the C# WinForms GUI for the medical image (Image 1).

■ Compression Algorithms Visualizer - Task 3

All Algorithms ▾ Load Image Compress Compare All

Single Algorithm Comparison

Algorithm	Compressed Size	Compression Ratio	Space Saving	Time (s)	PSNR	
RLE	1,424,786 bytes	0.53:1	-90.20%	0.4977	∞ dB	
Huffman	637,459 bytes	1.18:1	14.90%	0.7511	∞ dB	
Arithmetic	631,097 bytes	1.19:1	15.75%	1.5867	∞ dB	
CABAC	5 bytes	149817.60:1	100.00%	0.6254	13.60 dB	

**Table 4** - Compression results from the C# WinForms GUI for the forest image (Image 2).

## **Discussion of results**

In this task we tested four compression methods (RLE, Huffman, Arithmetic and CABAC) on two images: a medical CT image (Image 1) and a forest image (Image 2). We ran the tests in both the Python web app and the C# GUI. The numbers for RLE, Huffman and Arithmetic are very similar in both implementations.

For the **medical image**, all three lossless methods compress quite well:

- RLE gives a compression ratio of about 2.3:1.
- Huffman and Arithmetic are better, around 3.2–3.3:1.
- The PSNR is  $\infty$  dB for these three methods, which means the decompressed image is exactly the same as the original (lossless).

This makes sense, because the CT image has large smooth areas and not that many different grey levels. That gives the algorithms more redundancy to exploit.

For the **forest image**, the situation is different:

- RLE actually makes the file bigger (ratio  $\approx 0.53:1$ ), so it does not work well here.
- Huffman and Arithmetic still compress a bit (about 1.1–1.2:1), but the gain is small.

This also makes sense, because the forest image has a lot of fine detail and looks almost “noisy”, so there are fewer long runs (bad for RLE) and less obvious structure to compress.

The **CABAC implementation** in this project is only a simplified prototype. It shows the idea of adaptive binary arithmetic coding, but it has numerical problems. For both images it reports a compressed size of only 5 bytes and a PSNR that is not infinite, so the result is not really comparable to the other methods. We still include CABAC in the comparison to show the concept, but the numbers are treated as unreliable.

Overall, the results show that:

- Both programs (C# GUI and Python web app) can load images from disk, run all four algorithms and display performance metrics.
- Compression performance depends a lot on the image: the medical image compresses much better than the forest image.
- Entropy-based methods (Huffman and Arithmetic) are generally better than simple RLE for these tests.

## **Code Overview**

This project is implemented in two programming languages:

- **C#** with a WinForms GUI
- **Python** with a Streamlit web app

In both versions, the code is organized into:

1. **Compression logic** (RLE, Huffman, Arithmetic coding and CABAC / CABAC-style)
2. **A user interface** that loads images from disk, calls the algorithms, measures performance and shows the results.

---

## **C# implementation**

In C#, all compression algorithms are implemented inside the class **CompressionAlgorithms** in the file CompressionAlgorithms.cs. This class contains:

- RunRLE, RunHuffman, RunArithmetic, RunCABAC – high-level methods that the GUI calls
- Helper methods for each algorithm, such as RLECoder.EncodeImage, HuffmanEncode, ArithmeticEncode, CABACEncodeWithLength, etc.

- Utility functions like CalculatePSNR, ImageToRgbBytes, and small support classes (HNode, BitWriter, BitReader).
- 

The WinForms GUI code is in **MainForm.cs**. It:

- lets the user load an image with OpenFileDialog
- lets the user choose an algorithm (RLE, Huffman, Arithmetic, CABAC, or All)
- calls the corresponding RunXXX(...) method on CompressionAlgorithms
- measures execution time using Stopwatch
- displays the original and decompressed image in PictureBox controls
- shows metrics (original size, compressed size, compression ratio, space saving, time, PSNR) in a ListView.

A simplified example from **Huffman implementation** looks like this:

```
1 reference
private (byte[] data, Dictionary<byte, int> tree, int bitCount) HuffmanEncode(byte[] data)
{
    if (data.Length == 0)
        return (new byte[0], new Dictionary<byte, int>(), 0);

    var frequency = new Dictionary<byte, int>();
    foreach (byte b in data)
    {
        frequency[b] = frequency.ContainsKey(b) ? frequency[b] + 1 : 1;
    }

    var root = BuildHuffmanTree(frequency);
    var codes = new Dictionary<byte, string>();
    BuildHuffmanCodes(root, "", codes);

    var bitWriter = new BitWriter();
    foreach (byte b in data)
    {
        bitWriter.WriteBits(codes[b]);
    }
    var (encodedData, bitCount) = bitWriter.Finish();

    return (encodedData, frequency, bitCount);
}
```

This method:

1. Builds a frequency table of the byte values
2. Builds a Huffman tree with `BuildHuffmanTree`
3. Generates codes using `BuildHuffmanCodes`
4. Writes the variable-length codes into a bitstream using `BitWriter`

The high-level method `RunHuffman(Bitmap image)` calls `ExtractColorChannels(image)`, then repeatedly calls `HuffmanEncode` and `HuffmanDecode` for each RGB channel and reconstructs the image with `ReconstructFromChannels(...)`. It then calculates compressed size, execution time and PSNR and returns a `CompressionResult` object that the GUI displays.

The **CABAC** path in C# is handled by `RunCABAC(Bitmap image)`, which:

- converts the image to a binary black/white version (`ConvertToBinary`)
- turns it into a bit sequence (`ImageToBinarySequence`)
- encodes it with `CABACEncodeWithLength` (adaptive binary arithmetic)
- decodes it with `CABACDecode`
- rebuilds a binary image with `BinarySequenceToImage`

This demonstrates a CABAC-style adaptive arithmetic coder inside the same `CompressionAlgorithms` class.

---

## Python implementation

The Python implementation lives entirely in **app3.py**. It contains:

- Algorithm classes: RLECoder, HuffmanCoder, ArithmeticCoder, CABACCoder
- Utility functions: load\_image, calculate\_compression\_ratio, calculate\_space\_saving, calculate\_psnr, etc.
- The **Streamlit** GUI functions: main(), run\_single\_algorithm(...), and run\_comparison(...).

In main() we:

- configure the page with st.set\_page\_config(...)
- create the algorithms dictionary:

```
algorithms = {  
    "RLE": RLECoder(),  
    "Huffman": HuffmanCoder(),  
    "Arithmetic": ArithmeticCoder(),  
    "CABAC": CABACCoder()  
}
```

- load an image from disk using:

```
# File upload
uploaded_file = st.sidebar.file_uploader(
    "Upload an image",
    type=['png', 'jpg', 'jpeg', 'bmp', 'tiff']
)
```

- let the user select one algorithm or “All Algorithms” with `st.sidebar.selectbox(...)`
- pass the image to `run_single_algorithm` or `run_comparison`.

In `run_single_algorithm(image, algorithm, algorithm_name)` we:

- start a timer (`start_time = time.time()`)
- call the correct encode/decode based on `algorithm_name`, for example:

```
elif algorithm_name == "Huffman":
    encoded_bits, frequency_dict = algorithm.encode_image(image)
    compressed_size = len(encoded_bits) / 8 # Convert bits to bytes
    decoded_image = algorithm.decode_image(encoded_bits, frequency_dict, decode_shape)
    decoded_image = decoded_image.astype(np.uint8)
```

- compute metrics: original size, compressed size, compression ratio, space saving, execution time, PSNR
- show the original and decoded image with `st.image`
- display the metrics in a pandas DataFrame using `st.table(metrics_df)`

- for “All Algorithms”, run\_comparison(...) repeats the process for RLE, Huffman, Arithmetic and CABAC and plots bar charts using matplotlib.

An important and more advanced part of the Python version is the **CABACCoder** class. It converts the image to a bit sequence and performs adaptive binary arithmetic coding:

```
class CABACCoder:
    def __init__(self):
        self.reset_probs()

    def reset_probs(self):
        self.probabilities = {0: 0.5, 1: 0.5}

    def update_probability(self, symbol: int, learning_rate: float = 0.05):
        current_prob_0 = self.probabilities[0]

        if symbol == 0:
            new_prob_0 = (1 - learning_rate) * current_prob_0 + learning_rate
        else:
            new_prob_0 = (1 - learning_rate) * current_prob_0

        new_prob_0 = max(0.01, min(0.99, new_prob_0))
        self.probabilities[0] = new_prob_0
        self.probabilities[1] = 1.0 - new_prob_0

    def encode_binary_sequence(self, binary_sequence: List[int]) -> Tuple[float, Dict, int]:
        low = 0.0
        high = 1.0
        self.reset_probs()

        for symbol in binary_sequence:
            range_width = high - low
            prob_0 = self.probabilities[0]

            if symbol == 0:
                high = low + range_width * prob_0
            else:
                low = low + range_width * prob_0

            self.update_probability(symbol)

        encoded_value = (low + high) / 2.0

        range_width = high - low
        if range_width <= 0.0:
            range_width = 1e-12
        bit_length = int(math.ceil(-math.log2(range_width)))

        return encoded_value, self.probabilities.copy(), bit_length
```

This shows that we are not only using basic methods like RLE and Huffman, but also experimenting with a CABAC-style adaptive arithmetic coder.