Overview of the Quantum PAQJP_6 Project

- **Purpose**: Compress and decompress files using a combination of transformations and compression algorithms, embedding file creation time metadata and supporting various file types (e.g., JPEG, TEXT, DEFAULT).

- **Key Components**:

  - **Transformations**: Preprocessing steps (1-11 and 12-255) that modify data to make it more compressible.

  - **Compression Methods**: PAQ (dictionary-based), zlib (DEFLATE, dictionary-based), and a custom Huffman coding implementation.

  - **Metadata**: Encodes file creation time in 6 bytes, prepended with a 1-byte method marker.

  - **Losslessness**: All transformations and compression methods are lossless, ensuring perfect data recovery.

- **Filetype Handling**: Detects filetypes (JPEG, TEXT, DEFAULT) to prioritize transformations for specific formats.

Transformations (1-11 and 12-255)

The transformations are applied before compression to restructure the data, aiming to reduce entropy or exploit patterns that improve compression ratios. Each transformation is paired with a reverse transformation to ensure losslessness during decompression. Below is a detailed description of each transformation, as implemented in the provided code.

Transformations 1-11

1. **Transformation 1 (`transform_01`, marker 1)**:

   - **Operation**: XORs every third byte with a value derived from prime numbers (`PRIMES`), where `xor_val = prime` (for prime = 2) or `ceil(prime * 4096 / 28672)` otherwise, repeated 100 times.

   - **Reverse**: Identical to `transform_01` (XOR is self-inverse: `x ^ k ^ k = x`).

   - **Lossless**: Yes, as XOR is reversible, and the prime-based values are deterministic.

   - **Purpose**: Introduces periodicity to exploit patterns in data with regular structures.

2. **Transformation 2 (Mapped to `transform_01`, marker 2)**:

  - **Operation**: Uses `transform_01` (likely a typo in the code, as no `transform_02` is defined).

  - **Reverse**: Uses `reverse_transform_01`.

  - **Lossless**: Yes, identical to transformation 1.

  - **Note**: If a unique transformation was intended, it would need to be defined. The current implementation is lossless but redundant.

3. **Transformation 3 (`transform_03`, marker 3)**:

  - **Operation**: XORs each byte in chunks of size 4 with 0xFF.

  - **Reverse**: Identical to `transform_03` (XOR with 0xFF is self-inverse).

  - **Lossless**: Yes, as XOR is reversible.

  - **Purpose**: Inverts bits in fixed-size chunks to potentially simplify data patterns.

4. **Transformation 4 (`transform_04`, marker 1)**:

  - **Operation**: Subtracts `(i % 256)` from each byte (where `i` is the byte index), modulo 256, repeated 100 times.

  - **Reverse**: Adds `(i % 256)` modulo 256, repeated 100 times.

  - **Lossless**: Yes, modular arithmetic ensures reversibility: `(byte - n * (i % 256) + n * (i % 256)) % 256 = byte`.

  - **Purpose**: Introduces position-dependent changes to align data with compression algorithms.

5. **Transformation 5 (`transform_05`, marker 5)**:

  - **Operation**: Rotates each byte left by 3 bits (circular shift).

  - **Reverse**: Rotates right by 3 bits.

  - **Lossless**: Yes, bit rotation is a bijective operation for 8-bit bytes.

  - **Purpose**: Reorganizes bit patterns to potentially reduce entropy.

6. **Transformation 6 (`transform_06`, marker 6)**:

  - **Operation**: Applies a random permutation of bytes (0-255) using a fixed seed (42).

  - **Reverse**: Applies the inverse permutation using the same seed.

  - **Lossless**: Yes, the permutation is a bijection, and the fixed seed ensures consistency.

  - **Purpose**: Shuffles byte values to potentially create more compressible sequences.

7. **Transformation 7 (`transform_07`, marker 7)**:

  - **Operation**: XORs each byte with `len(data) % 256`, then with base-256 mapped pi digits (shifted by `len(data) % pi_length`), repeated based on data size (in KB, capped at 10 cycles).

  - **Reverse**: Reverses the XORs and pi digit shift.

  - **Lossless**: Yes, XOR is reversible, and the pi digits and size byte are deterministic.

  - **Purpose**: Uses pi digits to introduce mathematical structure, prioritized for JPEG/TEXT files.

8. **Transformation 8 (`transform_08`, marker 8)**:

  - **Operation**: XORs each byte with a prime number derived from `len(data) % 256`, then with pi digits, with a circular shift.

  - **Reverse**: Reverses the XORs and pi shift.

  - **Lossless**: Yes, similar to transformation 7, with deterministic parameters.

  - **Purpose**: Combines prime numbers and pi digits for complex patterns, suitable for JPEG/TEXT.

9. **Transformation 9 (`transform_09`, marker 9)**:

  - **Operation**: XORs each byte with a prime and seed table value, then with pi digits XORed with position, with a pi digit shift.

  - **Reverse**: Reverses the XORs and pi shift.

  - **Lossless**: Yes, all operations are reversible with deterministic inputs.

  - **Purpose**: Enhances transformation 8 with seed table randomness, optimized for JPEG/TEXT.

10. **Transformation 10 (`transform_10`, marker 10)**:

   - **Operation**: XORs each byte with `n`, computed from the count of 'X1' sequences (`(count * 2 + 1) // 3 * 3 % 256`), stored as the first byte, repeated based on data size.

   - **Reverse**: XORs with the stored `n`.

   - **Lossless**: Yes, XOR is reversible, and `n` is stored for accurate reversal.

   - **Purpose**: Uses data-specific patterns ('X1') to tailor the transformation, prioritized for JPEG/TEXT.

11. **Transformation 11 (`transform_11`, marker 11)**:

   - **Operation**: Adds `(y + 1) % 256` to each byte, repeated 100 times, with `y` (1-255) chosen to minimize compressed size and stored as the first byte.

   - **Reverse**: Subtracts `(y + 1) % 256` using the stored `y`, repeated 100 times.

   - **Lossless**: Yes, modular addition/subtraction is reversible: `(byte + n * (y + 1) - n * (y + 1)) % 256 = byte`.

   - **Purpose**: Optimizes for compression efficiency by testing multiple `y` values, prioritized for JPEG/TEXT.

Transformations 12-255 (`generate_transform_method`)

- **Operation**: For each marker (12-255), XORs each byte with `(data_size % scale_factor) % 256 + (i % 256)) % 256`, where `scale_factor = max(2000, min(256000, data_size))`, repeated 1000 times.

- **Reverse**: Identical to the forward transformation (XOR is self-inverse).

- **Lossless**: Yes, XOR is reversible, and `scale_factor` is deterministic based on data size: `(byte ^ k) ^ k = byte`.

- **Purpose**: Applies a size-dependent transformation to exploit data length patterns, with a high repeat count (1000) to ensure thorough mixing. The scaling factor (2000 to 256000) adjusts the transformation based on data size, making each marker unique in its effect.

Huffman Coding

- **Implementation**: The `compress_data_huffman` and `decompress_data_huffman` methods implement a custom Huffman coding scheme.

- **Process**:

  - **Compression**:

    - Converts input bytes to a binary string (bit sequence).

    - Calculates bit frequencies (`0` and `1`).

    - Builds a Huffman tree using a priority queue (`heapq`).

    - Generates variable-length codes for each bit (`0` and `1`), ensuring at least a default code (`0` or `1`) if missing.

    - Encodes the binary string using Huffman codes.

    - Converts the encoded bit string to bytes.

  - **Decompression**:

    - Converts compressed bytes to a binary string.

    - Rebuilds the Huffman tree from the compressed data's bit frequencies.

    - Decodes the binary string using the reversed Huffman codes.

    - Converts the decoded bit string back to bytes.

- **Lossless**: Yes. Huffman coding is inherently lossless, as each symbol (bit) is assigned a unique prefix-free code, and the frequency-based tree ensures accurate reconstruction. The code handles edge cases (e.g., empty input or missing symbols) by returning empty strings or default codes.

- **Usage**: Applied when the input size is below `HUFFMAN_THRESHOLD` (1024 bytes) in `compress_with_best_method`. If the Huffman-compressed size is smaller than other methods, it uses marker 4.

- **Purpose**: Efficient for small files or data with skewed bit distributions, complementing PAQ and zlib.

Dictionary-Based Compression

- **PAQ (`paq_compress`, `paq_decompress`)**:

  - **Description**: Uses the `paq` library for context-mixing compression, which builds a statistical model of the data to predict and encode symbols efficiently.

  - **Lossless**: Yes, PAQ is designed for lossless compression, using arithmetic coding and context modeling to reconstruct the original data perfectly.

  - **Role**: Primary compression method for transformed data, tested for each transformation in `compress_with_best_method`.

  - **Strengths**: Excellent for data with complex patterns, especially after transformations that reduce entropy (e.g., 7-11 for JPEG/TEXT).

- **zlib (`compress_data_zlib`, `decompress_data_zlib`)**:

  - **Description**: Uses the zlib library's DEFLATE algorithm, combining LZ77 (dictionary-based matching) and Huffman coding.

  - **Lossless**: Yes, DEFLATE is lossless, using a sliding window to find repeated patterns and encoding them with Huffman codes.

  - **Role**: Secondary compression method, tested alongside PAQ for each transformation. Used as a fallback in decompression if PAQ fails.

  - **Strengths**: Fast and effective for general-purpose compression, especially for data with moderate repetition.

Integration in the Project

- **Compression Workflow** (`compress_with_best_method`):

  - **Input**: Reads the input file as bytes and detects its filetype (JPEG, TEXT, DEFAULT).

  - **Transformations**: Applies transformations 1-11 and 12-255, prioritizing 7-11 for JPEG/TEXT files. For each transformation:

    - Transforms the data.

    - Compresses the result using PAQ and zlib.

    - Tracks the smallest compressed size and corresponding marker/method.

  - **Huffman Option**: For inputs < 1024 bytes, applies Huffman coding (marker 4) and compares its size.

  - **Output**: Writes `[marker][datetime_bytes][compressed_data]`, where `datetime_bytes` (6 bytes) encodes the file's creation time.

  - **Lossless**: Ensured by lossless transformations and compression methods. The marker identifies the transformation for decompression.

- **Decompression Workflow** (`decompress_with_best_method`):

  - **Input**: Reads the compressed file, extracting the marker (1 byte), datetime (6 bytes), and compressed data.

  - **Datetime**: Decodes the datetime for setting the output file's creation time.

  - **Decompression**:

    - If marker = 4, uses Huffman decompression.

    - Otherwise, attempts PAQ decompression, falling back to zlib if PAQ fails.

    - Applies the reverse transformation (1-11 or 12-255) based on the marker.

  - **Output**: Writes the decompressed data to the output file, restoring the original extension (JPEG/TEXT) for markers 7-11 and 12-255 if applicable.

  - **Lossless**: Ensured by reversible transformations and lossless decompression methods.

- **Filetype Handling**:

  - Prioritizes transformations 7-11 for JPEG/TEXT in compression to optimize for their data patterns.

  - Restores .jpg/.jpeg or .txt extensions during decompression for markers 7-11 and 12-255, based on the input file's extension.

Losslessness Confirmation

As analyzed previously:

- **Transformations 1-11**: All are lossless due to reversible operations (XOR, modular arithmetic, bit rotation, permutation). Reverse transformations use the same parameters (e.g., stored `y` for 11, `n` for 10, or deterministic size/pi digits).

- **Transformations 12-255**: Lossless, as they use self-inverse XOR operations with a deterministic `size_mod` based on data length.

- **Compression Methods**: PAQ, zlib, and Huffman are inherently lossless, ensuring no data loss during encoding/decoding.

- **Metadata**: The marker and datetime bytes are separate from the data, ensuring they do not affect losslessness.

Project Features and Notes

- **Pi Digits**: Uses 3 base-10 pi digits (mapped to 0-255) in transformations 7-9, loaded from or saved to `pi_digits.txt`. Fallback digits (3, 1, 4) are used if generation fails.

- **Datetime Encoding**: Embeds file creation time in 6 bytes, supporting years 0-4095, and restores it during decompression.

- **Error Handling**: Robustly handles empty inputs, invalid files, and compression/decompression failures, logging warnings/errors.

- **Performance Consideration**: Testing 255 transformations (1-11, 12-255) for each compression method (PAQ, zlib) may be computationally intensive. Limiting transformations (e.g., 12-50) could improve speed for large files.

- **Potential Improvement**: The redundant mapping of marker 2 to `transform_01` could be clarified or replaced with a unique transformation if intended.

Conclusion

The PAQJP_6 project combines a rich set of lossless transformations (1-11 and 12-255) with dictionary-based (PAQ, zlib) and Huffman coding methods to achieve flexible and efficient compression. Transformations 1-11 leverage diverse techniques (XOR, modular arithmetic, bit rotation, pi digits, seed tables), while 12-255 provide a scalable, size-dependent XOR approach. The custom Huffman coding is ideal for small files, and PAQ/zlib handle complex data patterns. All components are lossless.