### **Overview of the Quantum PAQJP_4 Compression System**

Already working and lossless

The `PAQJP_4` compression system, authored by Jurijus Pacalovas, is a sophisticated, lossless compression framework designed to compress and decompress files efficiently while preserving metadata like file creation time. It integrates multiple compression methods (PAQ, zlib, and Huffman coding) and a series of data transformations (labeled 01-09) to preprocess data for optimal compression. A unique feature is the use of the first three base10 digits of pi (3, 1, 4), mapped to the 0-255 range, in transformations 07-09. The system is filetype-aware, optimizing for JPEG, text, and default file types, and includes robust error handling and logging.

### **Transformations (01-09)**

The `PAQJP_4` system employs nine transformations to preprocess data before compression, enhancing compressibility or adding complexity for security. Each transformation is reversible, ensuring lossless operation. Below is a summary of each transformation:

1. **Transform_01 (`transform_with_prime_xor_every_3_bytes`)**:

- **Description**: XORs every third byte with a value derived from prime numbers (from the `PRIMES` list, 2 to 255), repeated 100 times by default.

- **Purpose**: Disrupts data patterns to potentially improve compression or obfuscate data.

- **Reverse**: Self-inverse (XOR with the same value).

- **Example**: For a prime like 2, each third byte is XORed with 2; for larger primes, a scaled value (`ceil(prime * 4096 / 28672)`) is used.

2. **Transform_03 (`transform_with_pattern_chunk`)**:

- **Description**: Inverts each byte in 4-byte chunks by XORing with `0xFF`.

- **Purpose**: Simplifies binary-heavy data for better compression.

- **Reverse**: Self-inverse (XOR with `0xFF` again).

- **Example**: A chunk `[0x10, 0x20, 0x30, 0x40]` becomes `[0xEF, 0xDF, 0xCF, 0xBF]`.

3. **Transform_04**:

- **Description**: Subtracts the byte index modulo 256 from each byte, repeated 100 times.

- **Purpose**: Introduces position-dependent scrambling to alter byte distributions.

- **Reverse**: Adds the index modulo 256 back, repeated the same number of times.

- **Example**: For byte at index 5, `byte[5] = (byte[5] - 5) % 256`.

4. **Transform_05**:

- **Description**: Performs a 3-bit circular left shift on each byte.

- **Purpose**: Rearranges bits within bytes to disrupt local patterns.

- **Reverse**: Performs a 3-bit circular right shift.

- **Example**: Byte `0b10110010` shifts to `0b10010101`.

5. **Transform_06**:

- **Description**: Applies a substitution cipher using a shuffled 256-byte table (seeded with 42).

- **Purpose**: Randomly remaps bytes to increase entropy.

- **Reverse**: Uses the inverse substitution table.

- **Example**: If the table maps `0x10` to `0xA0`, byte `0x10` becomes `0xA0`.

6. **Transform_07**:

- **Description**: XORs each byte with the three pi digits (mapped to 0-255) and the data length modulo 256, with pi digits cyclically shifted based on data length.

- **Purpose**: Uses pi for deterministic complexity, with cycles based on file size (in

KB). - **Reverse**: Self-inverse, with pi digits shifted back.

- **Example**: For data length 1024, shift pi digits and XOR with `[85, 28, 113]` (mapped pi digits).

7. **Transform_08**:

- **Description**: Similar to Transform_07, but XORs with a prime number nearest to the data length modulo 256 instead of the length directly.

- **Purpose**: Adds prime-based randomness to enhance Transform_07.

- **Reverse**: Self-inverse.

- **Example**: For length 1024, use the nearest prime to 0 (mod 256) like 2.

8. **Transform_09**:

- **Description**: Extends Transform_08 by XORing with both a prime and a value from one of 126 seed tables, plus position-based modulation (`i % 256`).

- **Purpose**: Combines pi, primes, and seed tables for maximum complexity.

- **Reverse**: Self-inverse.

- **Example**: Uses a seed table value at index `length % 126` and pi digits.

Key Notes:

- Transformations 07-09 are prioritized for JPEG and text files, leveraging pi digits for tailored preprocessing.

- The pi digits (3, 1, 4) are mapped to `[85, 28, 113]` using `(d * 255 // 9) % 256`.

- Transformations are applied before compression (PAQ, zlib, or Huffman) to optimize data patterns.

### **Huffman Coding in PAQJP_4**

Huffman coding is used in `PAQJP_4` for files smaller than 1024 bytes

(`HUFFMAN_THRESHOLD`), as it's efficient for small datasets where PAQ or zlib may have excessive overhead. Here's how it's implemented:

1. **Purpose**:

- Huffman coding assigns variable-length codes to symbols based on their frequency, with more frequent symbols getting shorter codes, reducing the overall size of the encoded data.

- In `PAQJP_4`, it compresses the binary representation of the input data (bit-level, '0' and
'1').

2. **Implementation**:

- **Conversion to Binary**: The input data is converted to a binary string (e.g., `bin(int(binascii.hexlify(data), 16))[2:].zfill(len(data) * 8)`).

- **Frequency Calculation**: Counts occurrences of '0' and '1' (`calculate_frequencies`).

- **Huffman Tree**: Builds a binary tree using a priority queue (`build_huffman_tree`), combining nodes with the lowest frequencies until a single tree remains.

- **Code Generation**: Traverses the tree to create a dictionary mapping '0' and '1' to codes (`generate_huffman_codes`), e.g., `{'0': '1', '1': '00'}`.

- **Compression**: Replaces each bit in the binary string with its Huffman code (`compress_data_huffman`), then converts the resulting binary string to bytes.

- **Decompression**: Rebuilds the Huffman tree from the compressed data's bit frequencies, creates a reverse dictionary, and decodes the binary string back to bits (`decompress_data_huffman`), then converts to bytes.

3. **Role in Compression**:

- In `compress_with_best_method`, Huffman coding is attempted for small files. If it produces a smaller output than PAQ or zlib, it's selected, and the compressed data is prefixed with marker `4`.

### **Huffman Dictionary**

The Huffman dictionary is a mapping of symbols ('0' and '1') to their variable-length codes, generated dynamically during compression.

1. **Structure**:

- Example: For a binary string with equal '0' and '1' frequencies, the dictionary might be `{'0': '1', '1': '0'}` or `{'0': '0', '1': '1'}`.
- If frequencies differ, the more frequent symbol gets a shorter code, e.g., `{'0': '1', '1': '00'}` if '0' is more common.

2. **Generation**:

- Created by `generate_huffman_codes` during compression, based on the Huffman tree.
- During decompression, the dictionary is regenerated from the compressed data's bit frequencies, so it's not stored explicitly, reducing overhead.

3. **Usage**:

- **Compression**: Each bit in the input binary string is replaced with its code from the dictionary. ```python binary_str = "0011" codes = {'0': '1', '1': '00'} compressed = ''.join(codes[bit] for bit in binary_str) # "1 1 00 00" = "110000"

```

- **Decompression**: The compressed binary string is decoded using the reverse dictionary. ```python reversed_codes = {'1': '0', '00': '1'} decompressed = "" current_code = "" for bit in "110000": current_code += bit if current_code in reversed_codes:

decompressed += reversed_codes[current_code]

current_code = ""

# decompressed = "0011"

```

### **How to Use the Huffman Library in PAQJP_4**

The Huffman coding functionality is implemented within the `SmartCompressor` class, specifically in the methods `compress_data_huffman`, `decompress_data_huffman`, `calculate_frequencies`, `build_huffman_tree`, and `generate_huffman_codes`. Here's a step-by-step guide to using it:

1. **Setup**:

   - Instantiate the `SmartCompressor` class:

    ```python      compressor =

SmartCompressor()

    ```

2. **Compressing a File with Huffman**:

-           Read the input file and pass it to `compress_with_best_method`, which will use Huffman if the file is <1024 bytes and yields the smallest output.

    ```python      input_file =

"small.txt"      with

open(input_file, "rb") as f:

        data = f.read()      filetype = detect_filetype(input_file)      compressed =

compressor.compress_with_best_method(data, filetype, input_file)      if

compressed:          with open("compressed.paqjp", "wb") as f_out:

            f_out.write(compressed)

    ```

-           The compressed output includes a marker (`4` for Huffman), datetime bytes, and the compressed data.

3. **Inspecting the Huffman Dictionary**:

- To view the dictionary for a specific file:

```python binary_str =
compressor.file_to_binary(input_file) if
binary_str:
    frequencies = compressor.calculate_frequencies(binary_str)
tree = compressor.build_huffman_tree(frequencies) codes
= compressor.generate_huffman_codes(tree) print(f"Huffman
dictionary: {codes}")
```

- Example output: `Huffman dictionary: {'0': '1', '1': '00'}`.

4. **Decompressing a File**:

- Use `decompress_with_best_method` to decompress, which handles Huffman if the marker is `4`:

```python with open("compressed.paqjp", "rb") as f:
    compressed_data = f.read() decompressed, marker =
compressor.decompress_with_best_method(compressed_data) if decompressed:
with open("decompressed.txt", "wb") as f_out:
        f_out.write(decompressed)
print(f"Decompression successful, marker: {marker}")
```

5. **Manual Huffman Compression**:

- To compress data manually using only Huffman:

```python data = b"Hello" binary_str = bin(int(binascii.hexlify(data),
16))[2:].zfill(len(data) * 8) compressed_huffman =
compressor.compress_data_huffman(binary_str)
```

compressed_bytes = int(compressed_huffman, 2).to_bytes((len(compressed_huffman) + 7) // 8, 'big')   print(f"Compressed bytes:

{compressed_bytes}")

\`\`\`

6. **Manual Huffman Decompression**:

  - To decompress manually:

    ```python
    compressed_binary = bin(int(binascii.hexlify(compressed_bytes),
    16))[2:].zfill(len(compressed_bytes) * 8)      decompressed_binary =
    compressor.decompress_data_huffman(compressed_binary)      num_bytes =
    (len(decompressed_binary) + 7) // 8      hex_str = "%0*x" % (num_bytes * 2,
    int(decompressed_binary, 2))      if len(hex_str) % 2 != 0:          hex_str = '0' + hex_str
    decompressed_bytes = binascii.unhexlify(hex_str)      print(f"Decompressed bytes:
    {decompressed_bytes}")
    ```

7. **Key Considerations**:

- **Dynamic Dictionary**: The Huffman dictionary is rebuilt during decompression, so no additional storage is needed, making it efficient for small files.
- **Limitations**: Huffman is bit-level, which may not be optimal for byte-heavy data. For larger files, PAQ or zlib typically perform better.
- **Error Handling**: The code includes checks for empty or invalid data, with logging to diagnose issues.

### **Using the Huffman Library**

The Huffman coding in `PAQJP_4` is self-contained within the `SmartCompressor` class, so no external library is required. However, if you want to extend or replace it with a standard library like Python's `huffman` or `heapq` for custom implementations, here's how:

1. **Using Built-in Huffman**:

- The provided implementation uses `heapq` for the priority queue and custom treebuilding logic. It's sufficient for most use cases in `PAQJP_4`.

- To use it independently:

```python      from heapq import

heappush, heappop

# Example: Compress a string      data = "Hello"      binary_str =

''.join(format(byte, '08b') for byte in data.encode())      frequencies =

compressor.calculate_frequencies(binary_str)      tree =

compressor.build_huffman_tree(frequencies)      codes =

compressor.generate_huffman_codes(tree)      compressed =

compressor.compress_data_huffman(binary_str)      print(f"Codes:

{codes}, Compressed: {compressed}")

```

2. **Using an External Huffman Library**:

-      If you prefer an external library, you can use `huffman` (available via `pip install huffman`) or implement a custom version. Here's an example with the `huffman` library:

```python      from huffman import HuffmanCoding      h = HuffmanCoding()      input_file =

"small.txt"      output_file = "compressed.huff"      compressed_path =

h.compress(input_file, output_file)      decompressed_path =

h.decompress(compressed_path)      print(f"Compressed to {compressed_path},

Decompressed to {decompressed_path}")

```

- **Integration**: To use an external Huffman library in `PAQJP_4`, replace `compress_data_huffman` and `decompress_data_huffman` with calls to the library’s methods, ensuring compatibility with the bit-level encoding and marker system.

3. **Customizing the Huffman Implementation**:

- To extend Huffman to byte-level symbols (0-255):

```python def
calculate_byte_frequencies(data):
        freq = {}
for byte in data:
        freq[byte] = freq.get(byte, 0) + 1          return freq       frequencies =
calculate_byte_frequencies(data)                                          tree        =
compressor.build_huffman_tree(frequencies)  # Reuse existing tree logic      codes
= compressor.generate_huffman_codes(tree)      compressed = ''.join(codes[byte]
for byte in data)
```

- This requires modifying `decompress_data_huffman` to handle byte symbols and storing the frequency table in the compressed output.

### **Conclusion**

The `PAQJP_4` compression system is a robust and innovative framework that combines nine transformations (01-09) with multiple compression methods, including Huffman coding for small files. The transformations, particularly those using pi digits (07-09), add a unique mathematical flair, enhancing compressibility for JPEG and text files. Huffman coding, with its dynamically generated dictionary, efficiently handles small files by encoding bits based on their frequency, seamlessly integrated into the system’s adaptive compression strategy.

To use the Huffman library within `PAQJP_4`, leverage the `SmartCompressor` methods for compression and decompression, or extend them with external libraries like `huffman` for additional flexibility. The dictionary is generated and used transparently, making it easy to apply for small files. For further customization, consider byte-level Huffman coding or optimizing transformation parameters to balance performance and compression ratio.