Portfolio of Transformations (01-09)

The `PAQJP_4` system includes a set of transformations (labeled 01 through 09) that preprocess data before compression to enhance compressibility or add complexity for security. Each transformation modifies the input data in a reversible way, ensuring the original data can be recovered during decompression. Here's a detailed breakdown of each transformation:

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

   - **Description**: Applies an XOR operation with prime numbers (from the `PRIMES` list) to every third byte of the data, repeated `repeat` times (default 100).

   - **Purpose**: Introduces randomness by XORing with prime-based values, which can disrupt patterns in the data to improve compression for certain data types or add a layer of obfuscation.

   - **Reverse**: The transformation is self-inverse because XOR is its own inverse (`a ^ b ^ b = a`).

   - **Implementation**: For each prime, a value is computed (`prime` or `ceil(prime * 4096 / 28672)`), and this value is XORed with every third byte.

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

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

   - **Purpose**: Flips bits in a predictable pattern, which can simplify certain data structures (e.g., binary-heavy data) for better compression.

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

   - **Implementation**: Processes data in 4-byte chunks, applying `byte ^ 0xFF`.

3. **Transform_04**:

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

   - **Purpose**: Introduces a position-dependent transformation to scramble data, potentially aiding compression by altering byte distributions.

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

   - **Implementation**: `byte[i] = (byte[i] - (i % 256)) % 256` for forward, and `byte[i] = (byte[i] + (i % 256)) % 256` for reverse.

4. **Transform_05**:

   - **Description**: Performs a circular bit shift (left by `shift=3` bits) on each byte.

   - **Purpose**: Rearranges bits within each byte to disrupt local patterns, which may improve compressibility for certain data.

   - **Reverse**: Performs a circular right shift by the same number of bits.

   - **Implementation**: `byte = ((byte << shift) | (byte >> (8 - shift))) & 0xFF` for forward, and the reverse for backward.

5. **Transform_06**:

   - **Description**: Applies a substitution cipher using a shuffled table (based on a seed, default 42).

   - **Purpose**: Randomly remaps byte values to increase entropy, which can aid compression or serve as lightweight encryption.

   - **Reverse**: Uses the inverse of the substitution table to map bytes back.

   - **Implementation**: A 256-byte substitution table is created with `random.shuffle`, and each byte is replaced with its mapped value.

6. **Transform_07**:

   - **Description**: XORs each byte with the 3 base-10 digits of pi (mapped to 0-255) and the data length modulo 256, with a circular shift of pi digits based on data length.

   - **Purpose**: Uses pi digits for a deterministic yet complex transformation, with cycle count based on file size (in KB) for adaptability.

   - **Reverse**: Self-inverse due to XOR, with the pi digits shifted back.

   - **Implementation**: Applies `byte[i] ^= pi_digit[i % pi_length]` and `byte[i] ^= size_byte` for `cycles` iterations.

7. **Transform_08**:

   - **Description**: Similar to Transform_07 but uses a prime number (nearest to data length modulo 256) instead of the length directly.

   - **Purpose**: Enhances Transform_07 by incorporating a prime-based XOR for added complexity.

   - **Reverse**: Self-inverse, with the same prime and pi digit shift logic.

   - **Implementation**: `byte[i] ^= size_prime` followed by pi-based XOR.

8. **Transform_09**:

   - **Description**: Extends Transform_08 by incorporating a seed table (126 tables of 256 values) and XORing with both a prime and a seed value, plus position-based modulation.

   - **Purpose**: Combines multiple sources of randomness (pi, primes, seed tables) for a highly complex transformation.

   - **Reverse**: Self-inverse, reversing the XOR operations and restoring the pi digit shift.

   - **Implementation**: `byte[i] ^= size_prime ^ seed_value` and `byte[i] ^= pi_digit ^ (i % 256)`.

Key Features Across Transformations:

- **Pi Integration**: Transformations 07, 08, and 09 use the first three base-10 digits of pi (3, 1, 4), mapped to 0-255 (`(d * 255 // 9) % 256`), stored in `PI_DIGITS`. These digits are cycled based on data length for dynamic behavior.

- **Reversibility**: All transformations are designed to be reversible, ensuring lossless compression.

- **Filetype Optimization**: JPEG and TEXT files prioritize transformations 07, 08, and 09 to leverage pi-based randomness, which may better suit their data patterns.

Huffman Coding in PAQJP_4

Huffman coding is used as one of the compression methods when the input data size is below `HUFFMAN_THRESHOLD` (1024 bytes). Here's how it works in the system:

1. **Purpose**:

   - Huffman coding is a variable-length prefix code that assigns shorter codes to more frequent symbols, optimizing compression for small files where PAQ or zlib may be less efficient.

   - It's particularly effective for data with skewed frequency distributions.

2. **Implementation** (`compress_data_huffman` and `decompress_data_huffman`):

   - **Frequency Calculation**: The input data is converted to a binary string (each byte to 8 bits), and the frequency of '0' and '1' bits is calculated (`calculate_frequencies`).

   - **Huffman Tree**: A binary tree is built using a priority queue (`build_huffman_tree`), where nodes with lower frequencies are combined into a parent node until a single tree remains.

   - **Code Generation**: The tree is traversed to assign codes (`generate_huffman_codes`), with left edges as '0' and right edges as '1'. Each bit ('0' or '1') gets a unique code.

   - **Compression**: The binary string is encoded by replacing each bit with its Huffman code, resulting in a compressed binary string. This is converted to bytes for storage.

   - **Decompression**: The compressed binary string is decoded by traversing the Huffman tree bit by bit, emitting the corresponding symbol ('0' or '1') when a leaf is reached. The resulting binary string is converted back to bytes.

3. **Usage in Compression**:

   - In `compress_with_best_method`, Huffman coding is attempted for small files (<1024 bytes). If the compressed size is smaller than other methods (PAQ or zlib), it's selected, and the output is prefixed with marker `4`.

4. **Dictionary**:

   - The "dictionary" in Huffman coding refers to the mapping of bits ('0' and '1') to their variable-length codes (e.g., `'0': '1'`, `'1': '00'`).

- This dictionary is generated dynamically for each compression based on the bit frequencies in the input data.

- During compression, the dictionary is not stored explicitly in the output; instead, the decompressor rebuilds it by analyzing the frequency of bits in the compressed data (`decompress_data_huffman`).

5. **How to Use the Dictionary**:

  - **Compression**: The dictionary is used to map each bit in the input binary string to its Huffman code. For example, if `'0': '1'` and `'1': '00'`, the string `"0011"` becomes `"1 00 1 00" = "100100"`.

  - **Decompression**: The reverse dictionary (code to symbol) is used to decode the compressed binary string by matching sequences of bits to their corresponding symbols.

  - **Dynamic Generation**: Since the dictionary is rebuilt during decompression based on the compressed data's bit frequencies, no additional storage is needed, making it efficient for small files.

Table Count Minus

The phrase "table count minus" is ambiguous in the context of your code, but based on the provided code, it likely refers to one of the following concepts:

1. **State Table (`StateTable` and `nex`)**:

  - The `StateTable` class defines a state machine with 252 states, used for context modeling or prediction in the compression process (possibly in the PAQ component, though not explicitly used in the provided code).

  - Each state has transitions (`nex(state, sel)`) based on a selector (`sel`), with columns representing next states for different inputs.

  - **Interpretation of "Table Count Minus"**: This could refer to reducing the number of states or transitions in the table for optimization. For example, if you want to count the number of valid transitions minus invalid ones:

- The table has 252 rows and 4 columns, so there are `252 * 4 = 1008` total transitions.

- To "count minus," you might exclude transitions where the next state is 0 or invalid. However, in the provided table, all transitions seem valid (non-zero), so the count minus invalid transitions would still be 1008.

- **Calculation**: If you meant to count transitions where the state decreases (`next_state < current_state`):

```python
decrease_count = sum(1 for state in range(len(nex.table)) for sel in range(4)
          if nex.table[state][sel] < state)
print(f"Transitions where state decreases: {decrease_count}")
```

This would need to be computed explicitly, but it's not directly used in the compression logic.

2. **Seed Tables (`generate_seed_tables`)**:

  - The `seed_tables` in `SmartCompressor` consist of 126 tables, each with 256 random values (5 to 255).

  - **Interpretation of "Table Count Minus"**: This could mean reducing the number of seed tables or their size. For example:

   - Current count: 126 tables.

   - "Minus" could imply using fewer tables (e.g., 126 - N) for optimization or testing.

   - **Calculation**: If you want to reduce the number of tables:

   ```python
   num_tables = 126
   reduced_tables = num_tables - 10  # Example: reduce by 10
   print(f"Reduced seed tables: {reduced_tables}")
   ```

   This would require modifying `generate_seed_tables` to generate fewer tables.

3. **Pi Digits or Transformation Count**:

- The code uses 3 pi digits (`PI_DIGITS`). "Table count minus" might refer to reducing the number of pi digits used in transformations 07-09.

  - **Calculation**: Current count is 3 digits. Reducing by 1 would leave 2 digits:

  ```python
  new_pi_digits = PI_DIGITS[:-1]  # Remove last digit
  print(f"Reduced pi digits: {new_pi_digits}")
  ```

  This would require updating `generate_pi_digits` and the transformation logic.

4. **Huffman Tree Nodes**:

   - In Huffman coding, the "table" could refer to the frequency table or the number of nodes in the Huffman tree.

   - **Interpretation of "Table Count Minus"**: This might mean counting the number of nodes minus leaf nodes or vice versa.

   - **Calculation**: For a binary string with symbols '0' and '1', the Huffman tree has:

   - 2 leaf nodes (one for '0', one for '1').

   - 1 internal node (the root for a 2-symbol tree).

   - Total nodes = 3. "Minus" leaf nodes: `3 - 2 = 1` (the internal node).

   - For dynamic frequencies:

   ```python
   frequencies = compressor.calculate_frequencies(binary_str)
   num_symbols = len(frequencies)
   num_nodes = 2 * num_symbols - 1
   num_leaves = num_symbols
   print(f"Total nodes: {num_nodes}, Non-leaf nodes: {num_nodes - num_leaves}")
   ```


Most Likely Interpretation:

- Given the context, "table count minus" most likely refers to the **seed tables** in `SmartCompressor` (126 tables) or the **state table** transitions. If you meant reducing the number of seed tables, you could modify `generate_seed_tables` to use fewer tables (e.g., 126 - 10 = 116). If you meant the state table, it could involve counting transitions where the state decreases or excluding certain states.

How to Compute:

- For seed tables:

```python
original_count = len(compressor.seed_tables)  # 126
reduced_count = original_count - 10  # Example reduction
print(f"Original seed tables: {original_count}, Reduced: {reduced_count}")
```

- For state table transitions where state decreases:

```python
decrease_count = sum(1 for state in range(len(nex.table)) for sel in range(4)
            if nex.table[state][sel] < state)
print(f"State transitions where state decreases: {decrease_count}")
```

How to Use the Huffman Dictionary

The Huffman dictionary is a mapping of symbols ('0' and '1') to variable-length codes, generated dynamically during compression. Here's how to use it effectively:

1. **Generating the Dictionary**:
   - During compression (`compress_data_huffman`):

- Convert the input data to a binary string (e.g., `bin(int(binascii.hexlify(data), 16))[2:].zfill(len(data) * 8)`).

- Calculate bit frequencies (`calculate_frequencies`).

- Build the Huffman tree (`build_huffman_tree`) and generate codes (`generate_huffman_codes`).

- Example:

```python
binary_str = "00110011"

frequencies = compressor.calculate_frequencies(binary_str)  # {'0': 4, '1': 4}

tree = compressor.build_huffman_tree(frequencies)

codes = compressor.generate_huffman_codes(tree)  # e.g., {'0': '1', '1': '0'}

print(f"Huffman codes: {codes}")
```

2. **Using in Compression**:
   - Replace each bit in the binary string with its Huffman code:

   ```python
   compressed = ''.join(codes[bit] for bit in binary_str)

   print(f"Compressed binary: {compressed}")
   ```

   - Convert the compressed binary string to bytes:

   ```python
   compressed_bytes = int(compressed, 2).to_bytes((len(compressed) + 7) // 8, 'big')
   ```

3. **Using in Decompression**:
   - Rebuild the Huffman dictionary from the compressed data's bit frequencies:

   ```python

```python
    compressed_binary = bin(int(binascii.hexlify(compressed_data),
16))[2:].zfill(len(compressed_data) * 8)

    frequencies = compressor.calculate_frequencies(compressed_binary)

    tree = compressor.build_huffman_tree(frequencies)

    codes = compressor.generate_huffman_codes(tree)

    reversed_codes = {code: symbol for symbol, code in codes.items()}
```

- Decode the compressed binary string by matching codes:

```python
decompressed = ""

current_code = ""

for bit in compressed_binary:

    current_code += bit

    if current_code in reversed_codes:

        decompressed += reversed_codes[current_code]

        current_code = ""
```

- Convert the decompressed binary string back to bytes:

```python
num_bytes = (len(decompressed) + 7) // 8

hex_str = "%0*x" % (num_bytes * 2, int(decompressed, 2))

if len(hex_str) % 2 != 0:

    hex_str = '0' + hex_str

decompressed_bytes = binascii.unhexlify(hex_str)
```

4. **Practical Usage**:

   - The dictionary is not stored in the compressed file; it's regenerated during decompression, making the process self-contained.

- To inspect the dictionary for a specific file:

```python
with open("input_file", "rb") as f:
    data = f.read()
binary_str = bin(int(binascii.hexlify(data), 16))[2:].zfill(len(data) * 8)
frequencies = compressor.calculate_frequencies(binary_str)
tree = compressor.build_huffman_tree(frequencies)
codes = compressor.generate_huffman_codes(tree)
print(f"Huffman dictionary: {codes}")
```

5. **Limitations**:
   - Huffman coding is only used for files < 1024 bytes due to its overhead for larger files.
   - The dictionary is minimal (only '0' and '1'), so it's efficient but limited to bit-level compression.

Example Usage of the System

To demonstrate how to use the transformations and Huffman dictionary, here's an example:

```python
compressor = SmartCompressor()

# Compress a small file with Huffman coding
with open("small.txt", "rb") as f:
    data = f.read()
```

```python
compressed = compressor.compress_with_best_method(data, Filetype.TEXT, "small.txt")
with open("compressed.paqjp", "wb") as f_out:
    f_out.write(compressed)

# Decompress
with open("compressed.paqjp", "rb") as f:
    compressed_data = f.read()
decompressed, marker = compressor.decompress_with_best_method(compressed_data)
with open("decompressed.txt", "wb") as f_out:
    f_out.write(decompressed)

# Inspect Huffman dictionary for a file
binary_str = compressor.file_to_binary("small.txt")
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}")
```

Table Count Minus Example

Assuming "table count minus" refers to reducing the number of seed tables:

```python
# Original number of seed tables
```

```python
original_count = len(compressor.seed_tables)  # 126

print(f"Original seed tables: {original_count}")


# Reduce by 10 tables

reduced_count = original_count - 10

compressor.seed_tables = compressor.seed_tables[:reduced_count]

print(f"Reduced seed tables: {len(compressor.seed_tables)}")
```


For state table transitions where the state decreases:


```python
decrease_count = sum(1 for state in range(len(nex.table)) for sel in range(4)

            if nex.table[state][sel] < state)

print(f"State transitions where state decreases: {decrease_count}")
```

Recommendations

1. **Clarify "Table Count Minus"**:

   - If you meant something specific (e.g., reducing seed tables, state table transitions, or pi digits), please provide more details, and I can tailor the explanation or code.

   - If it's about optimizing the state table, consider analyzing the table's structure to remove redundant states.

2. **Enhance Huffman Usage**:

- For larger files, consider extending Huffman coding to byte-level symbols (0-255) instead of bits to improve compression ratios, though this would increase dictionary overhead.

   - Store the Huffman tree or frequencies explicitly for faster decompression if needed.

3. **Optimize Transformations**:

   - Transformations 07-09 are computationally intensive due to multiple cycles. Consider reducing `repeat` or `cycles` for faster processing.

   - Test transformations on different file types to determine which ones yield the best compression ratios.

4. **Error Handling**:

   - The code already includes robust error handling (e.g., logging, fallback pi digits). Ensure input files are validated for size and content to avoid edge cases.