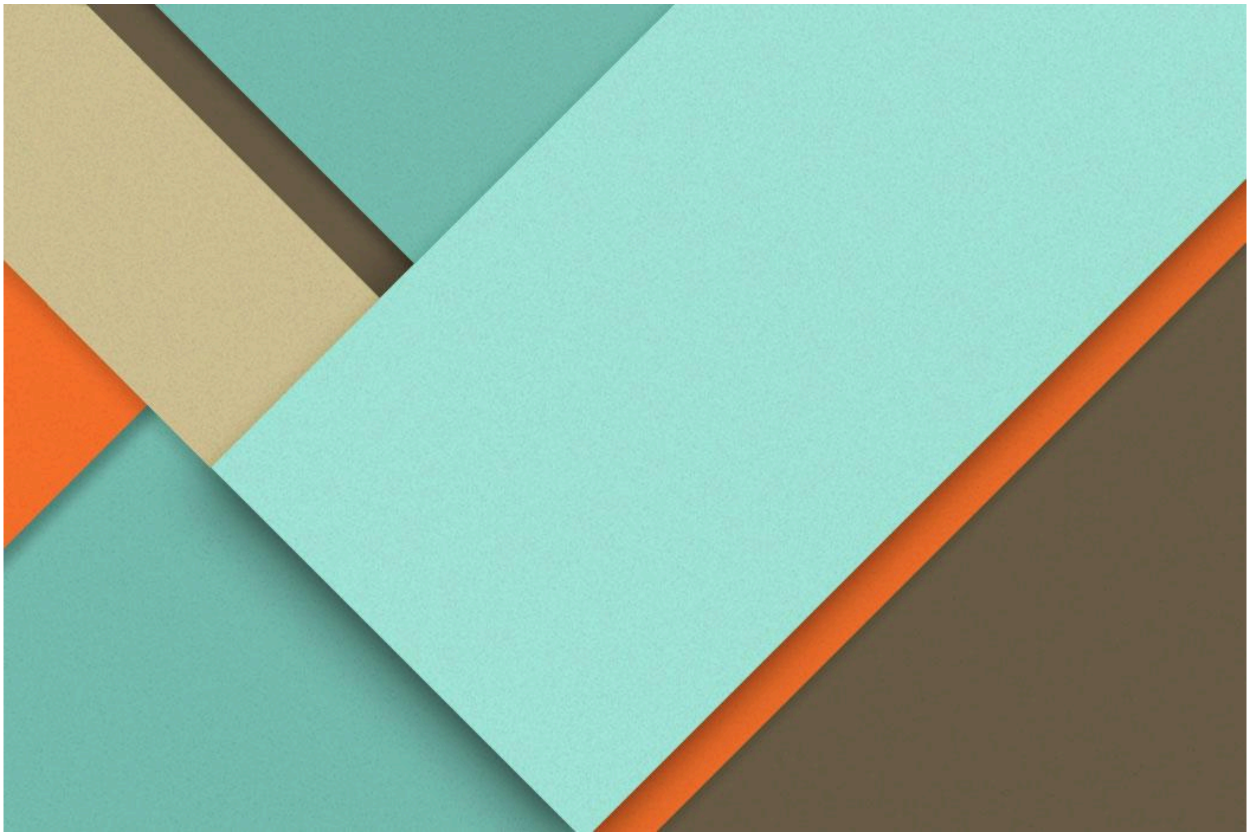

Burrows-Wheeler Compression

Source code:

Transform function: SuffixArray ,

Compress , Decompress : minimum heap

Suffix Array and MinHeap Analysis:

While the provided code snippets don't reveal the exact implementation details of `SuffixArray` and `MinHeap` classes, we can discuss their functionalities and general approaches relevant to Burrows-Wheeler Transform (BWT) and Huffman Coding.

1. Suffix Array:

- **Purpose:**

  - A Suffix Array is a data structure that stores the indexes of all suffixes of a string in lexicographical order (alphabetical order).
- **Functionality in BWT:**
  - The BWT algorithm utilizes the Suffix Array to efficiently compute the BWT transformation.
  - By iterating through the Suffix Array, BWT can directly access the character preceding each suffix in the original string.
- **Implementation Approaches:**
  - There are various algorithms for constructing a Suffix Array, with varying time and space complexities.
    - Some common approaches include:
      - **SAIS (String Algorithm Induced Sorting):** Efficient O(n log n) time complexity with low memory overhead.
      - **Manacher's Algorithm:** Linear time complexity (O(n)) but requires more space compared to SAIS.

2. MinHeap:

- **Purpose:**
  - A MinHeap is a specialized tree-based data structure where the root node always holds the minimum value among all elements.
- **Functionality in Huffman Coding:**
  - Huffman Coding utilizes a MinHeap to efficiently build the Huffman tree.
  - The MinHeap is used to prioritize nodes (characters) based on their frequencies during tree construction.
  - Nodes with lower frequencies (higher priority) are extracted first, allowing for the creation of an optimal tree for data compression.
- **Implementation:**
  - MinHeaps are typically implemented using a binary tree data structure.
  - Specific operations like `Insert` (add a new element) and `ExtractMin` (remove the minimum element) involve maintaining the heap property (minimum at the root) through efficient operations like swapping and sifting elements up or down the tree.

Complexity Analysis (General):

- Suffix Array construction algorithms generally have a time complexity of O(n log n) with space complexity ranging from O(n) to O(n log n) depending on the specific algorithm.
- MinHeap operations (`Insert` and `ExtractMin`) typically have a logarithmic time complexity (O(log n)) due to efficient heap manipulation techniques.

Analysis of the whole code :

1-Burrows-Wheeler Transform (BWT):

A- Transform function :

Purpose: Performs the Burrows-Wheeler Transform (BWT) on a byte array representing text data.

Functionality:

- Appends a unique End-of-File (EOF) character to the input for proper BWT handling.
- Converts the byte array to a short array (assuming `SuffixArray` works with shorts).
- Utilizes the **assumed** `SuffixArray.Construct(shortString)` function (not provided) to construct the suffix array for the input data.
- Iterates through the suffix array to compute BWT indices and creates a new byte array containing the transformed data.

Complexity:

- Time complexity depends on the `SuffixArray.Construct` function (assumed O(n log n) for efficient algorithms like SAIS).
- Space complexity is O(n) for storing the transformed data and potentially O(n) for temporary data structures used by `SuffixArray.Construct`.

Overall Time Complexity: O(n log(n)) ,

Overall Space Complexity: O(n).

Detailed Time complexity:

Appending EOF Character and Conversion to Short Array (Loops): O(n),

Suffix Array Construction : O(n log n),

BWT Calculation (Loop): O(n).

Detailed Space complexity:

Input with EOF (Array): O(n),

Short Array: O(n).

Suffix Array: O(n).

BWT Array: O(n).

B- Inverse function :

Purpose: Performs the inverse BWT to recover the original data from the transformed byte array.

Functionality:

- Constructs a dictionary to count the occurrences of each byte in the transformed data.
- Builds the "first column" by iterating through the count dictionary and adding each byte based on its frequency.
- Creates a dictionary (`nextIndex`) to store queues for each byte, representing the next position where that byte appears in the transformed data.
- Populates the `next` array based on the order of bytes in the transformed data and the `nextIndex` dictionary.
- Iteratively uses the `next` array to reconstruct the original byte sequence, starting from the first byte in the "first column."
- Removes the appended EOF character during reconstruction.

Complexity:

- Time complexity is dominated by loop iterations and potentially O(n) for building data structures.

- Space complexity is O(n) for storing dictionaries, queues, and the `next` array.

Overall Time Complexity: O(n) ,

Overall Space Complexity: O(n).

Detailed time complexity:

Counting Character Occurrences (Loop): O(n),

Building First Column (Loop): O(n),

Building Next Array (Loops): O(n),

Reconstructing Original Data (Loop): O(n).

Detailed space complexity:

Character Count Dictionary: O(n),

Next Index Dictionary and Queue: O(n),

Next Array and Original Data Array: O(n).

Move-to-front function

- **Encode:**
    - **Purpose:** Implements the Move-to-Front (MTF) encoding algorithm for data compression.
    - **Functionality:**
        - Initializes a table containing all possible byte values (0 to 255).
        - Iterates through the input data, finding the index of each byte in the table.
        - The index becomes the encoded output for that byte.
        - Removes the encoded byte from its original position in the table and inserts it at the beginning, effectively moving it to the front.
    - **Complexity:**
        - Time complexity is O(n) for iterating through the input data and performing table operations (insertion/removal).
        - Space complexity is O(1) as a constant size table is used.
-

- **Decode:**
    - **Purpose:** Performs the inverse MTF decoding to recover the original data from the encoded byte array.
    - **Functionality:**
        - Follows the same logic as encoding but utilizes the encoded indices to retrieve the corresponding byte values from the table.
        - After retrieving a byte value, it's moved to the front of the table for subsequent lookups.
    - **Complexity:**
        - Time complexity is O(n) for iterating through the encoded data and performing table operations.
        - Space complexity is O(1) due to the constant size table.

Overall Time Complexity: O(n) ,

Overall Space Complexity: O(n).

Detailed Time complexity:

Encoding/Decoding Loop (Iteration): O(n),

IndexOf and RemoveAt/InsertAt (List Operations): O(n).

Detailed Space complexity:

Symbol Table: O(n).

Huffman function:

Combined Complexity Analysis

Time Complexity

Encoding (Compression):

1.Frequency Calculation: O(n)

2.Building the Huffman Tree: O(m log m)

3.Generating Huffman Codes: O(m)

4.Encoding Tree Structure and Data: O(n log m)

Overall Encoding Time Complexity: O(n + m log m + n log m) = O(n log m)

Decoding (Decompression):

1.Reconstructing the Huffman Tree: O(m)

2.Decoding the Data: O(n)

Overall Decoding Time Complexity: O(n + m)

Combined Time Complexity:

The total time complexity for both encoding and decoding will be:

O(encoding)+O(decoding)=O(nlogm)+O(n+m)O(encoding)+O(decoding)=O(nlogm)+O(n+m)

Given that m≤nm≤n (the number of unique characters mm cannot exceed the length of the input nn), the combined complexity can be simplified to:

O(nlogm)+O(n+m)≈O(nlogm)+O(n)=O(nlogm)O(nlogm)+O(n+m)≈O(nlogm)+O(n)=O(nlogm)

Therefore, the overall time complexity for encoding followed by decoding is:

O(nlogm)O(nlogm)

Space Complexity

Encoding (Compression):

1.Frequency Map: O(m)

2.Huffman Tree: O(m)

3.Encoded Data and Tree Structure: O(n + m)

Overall Encoding Space Complexity: $O(n + m)$

Decoding (Decompression):

1. Reconstructed Huffman Tree: $O(m)$
2. Decoded Data: $O(n)$

Overall Decoding Space Complexity: $O(n + m)$

Combined Space Complexity:

The total space complexity for both encoding and decoding will be:

O(encoding)+O(decoding)=O(n+m)+O(n+m)=2 · O(n+m)=O(n+m)O(encoding)+O(decoding)=O(n+m)+O(n+m)=2 · O(n+m)=O(n+m)

Therefore, the overall space complexity for encoding followed by decoding is:

O(n+m)O(n+m)

Conclusion

Combining the steps of encoding and decoding, the overall computational complexity is as follows:

Time Complexity: O(nlogm)O(nlogm)

Space Complexity: O(n+m)O(n+m)

Overall Time Complexity: O(n log m) :where m is number of unique character ,

Overall Space Complexity: trivial .

Compression ratio of “large cases” :

Equation : 1 - (size of file after compression / size of original file)

Case 1: chromosome11-human : 0.713 = 71%

Case2: dickens: 0.387 = 38%

Case3: pi-10million: 0.562 = 56%

Case 4 : world192: 0.325 = 32%

Execution time of compression and decompression for “large cases” :