Programming Assignment 3

Department of Computer Science, University of Wisconsin – Whitewater
Algorithms in the Real World (CS 738)

Instructions For Submissions

- **This assignment is to be complete individually.**
- Submission is via Canvas as a single zip file. Submit code.
- **Your choice of programming languages are limited to C++ or Java.** Make sure you cite everything. If you use online code resources, please make sure that it is not a verbatim copy; you really should avoid this in general.

---

1 Data Description and Testing

A skeleton code has been provided for both C++ and Java. That will be your starting point for the assignment.

Small function snippets have been written in the `Test` file for testing each of the following sections, code for which you have to write. The file also contains test correctness methods for Huffman, Huffman IO (writing & reading the Huffman object from the file), ManberMyers, BruteForce Suffix Array construction, and LZ77; you do not have to do anything for them. The expected output for these correctness methods have been provided.

The input data is provided in the assignment folder `Data`. You will notice that under this folder there are a few empty folders – the compressed files that the code creates will be created here.

The compressed files that you are supposed to obtain are given in the `Compressed Files` folder; check the appropriate programming language and algorithms. Notice the use of the extensions:

- In LZ77, for the delta array, the files with the `.delta.huffman.encoding` extension store the Huffman output in a bit-packed manner, and the files with the `delta.huffman.mapping` extension store the Huffman mapping table.

  Similar remarks hold for the extensions involving `length` and `next`.

- In LZSS, the extensions are similar to LZ77. The files with the `.identifier` extension store the identifier array in a bit-packed manner.

- In BWT, the files with the `.alpha` extension store the alphabet to understand what the MTF should be decoded into. The files with the `.huffman.encoding` extension store the Huffman output in a bit-packed manner, and the files with the `.huffman.mapping` extension store the Huffman mapping table.

- In LZ78, the characters are stored in Huffman compressed form under the files with the following extensions: `.chars.huffman.encoding` and `.chars.huffman.mapping`. The files with the `.nodes` extension store the nodes in a bit-packed manner.

You will also find an excel that shows the compression ratio, compression time, and decompression time for each algorithm and each programming language.[1]

2 Task 0: Understand the structure of the code

There are files that you may need to use, but you do not have to write any code in them.

- `FilePaths`

  There is no need to modify anything here other than DATA_DIRECTORY path (as long as you copy the `Data` folder provided by me as is).

  All extensions for compressed files can be found in this class.

- `HelperFunctions`

  This contains functions that you may find useful both for implementing and debugging. I am not documenting the role of the functions. For the most part, they are self-explanatory; the missing details can be figured out by running LZ77 test-code provided to you.

- `Huffman`, `HuffmanElement`, and `HuffmanIOHelper`

  In this project, you will need to Huffman encode both characters (such as the next characters of LZ77) and integers (such as MTF). Since characters can be mapped to integers via ASCII, we have implemented Huffman to work on a sequence of integers.

  `Huffman` encodes a sequence of integers into a `HuffmanElement` object. `HuffmanElement` contains two fields:

  - the encoding, which is the encoded string of zeroes and ones representing the input
  - the mapping table, which maps the an integer (in the input) to its equivalent codeword

  `Huffman` also decodes a `HuffmanElement` object back into the original sequence of integers.

  `HuffmanIOHelper` writes an `HuffmanElement` object into two files:

  - the mapping part into a file as multiple lines, where each line is *integer* : *codeword*
  - the encoding part as a bit-packed byte array into another file.

  Notice the use of the file extensions and the naming format while writing to these files. `HuffmanIOHelper` also reads back from these files to create the `HuffmanElement` object.

- `IOHelper`

  Contains functions for reading the data, reading/writing a byte array/vector from/to a file, reading/writing a hash-map from/to a file, reading/writing the alphabet from/to a file, and computing the size of a file (in bytes).

  The function for reading the data appends the EOF '\0' character at the end.

- `LZ77`, and `LZ77Element`

  `LZ77` encodes a sequence of characters into an LZ77 encoding, which is captured by an `LZ77Element` object. The encoder assumes that the sequence ends in the unique EOF '\0'.

  Given an `LZ77Element` object, `LZ77` also decodes it back to the original sequence of characters.

[1] C++ outputs are slower. Trying to optimize may make the structure of the code more cumbersome/vastly different from the Java variant; that's why I ignored it.

- `LZSSElement`, and `LZ78Element`

  You will need to use this to represent the encoding for LZSS and LZ78.

- `Master`

  This is the project driver. For each of the compression schemes, the data file is read, compressed, and written to the disk. Then, the compressed data is read back, decompressed, and compared to the data to ensure that the compression algorithm has been correctly implemented. Statistics (such as compression ratio and time) are displayed for each algorithm.

- `MTFElement`

  You will need to use this for the MTF implementation.

- `SuffixArray`

  This contains implementations of ManberMyers and a brute-force algorithm (based on hybrid sorting); toggle the flag to use the algorithm you want.

- `Test`

  Contains code for testing. Initially, you should:

  - Most of the code in the `loadTest()` function in `Test` is commented out.
  - Starting from the `testBitPacker` section, the code in the `testCorrectness()` function in `Test` is commented out.

Now, you will start your implementation. Each function (check the next few sections) that you implement has placeholder statements to ensure that the project compiles.

3 Task 1: Bit Packing

In C++, a byte is a char. So, whenever we say byte here, the underlying representation for those using C++ is a char.

- In the `pack` method of the BitPacker class, you have to pack the bits into a byte array/vector.The argument is a string of zeroes & ones. Sketchy ideas on how to do this:

  - Blocks of 8 bits at a time will be packed; the last block may have < 8 bits. The total number of bytes needed is $1 + \lceil \frac{bits.length}{8.0} \rceil$. Last byte will store length of the last block.
  - Convert each block into a decimal number (a function is given to you in `HelperFunctions`) and store it in the byte array/vector.
  - At the end, remember to store the length of the last block in the last cell of the byte array/vector.

  So, if there are 43 bits, then the byte array will be of length 7. There are 6 blocks – first 5 blocks (i.e., bit-strings) are of length 8 each and the last one is of length 3. The decimal equivalent of these bit-strings will be stored in the first 6 bytes. The last byte will store the length of the last block, i.e., 3 in this case.

- In the `unpack` method of the BitPacker class, the argument is a byte array/vector. The last cell contains the length of the last block during the packing stage. We will now convert the byte array/vector back to a string of bits.

- You will now convert a decimal number back to a binary string of a particular length. Notice that the binary equivalent of the decimal may need to be padded with zeroes to obtain the desired length (a function is given to you in `HelperFunctions`).
- So, read each byte from the argument and convert into a binary string. Remember that each bit string (block) is of length 8, except possibly the last one, whose length is stored in the last cell of the byte array.
- When you read a byte, some of them can be negative (because bytes range from -128 to 127). To convert the negative numbers to the appropriate binary string, remember to first add 256 to get the unsigned variant.

Once you have implemented the two functions, you can uncomment the code in `testCorrectness()` for testing the `BitPacker` class. You can also uncomment the code for correctness testing as well as load testing of `LZ77`.

4 Task 2: LZSS

You should be able to borrow pretty much the entire code from LZ77 and modify a little bit following the logic of LZSS. Check the notes for more details. For encoding, as in LZ77, assume that the sequence of characters ends in '\0'. Notice the usage of `LZSSElement` for both functions.

Once you have implemented the two functions, you can uncomment the code in `testCorrectness()` for testing the `LZSS` class.

5 Task 3: BWT

This has the following sub-tasks:

- For the `encode` function, assume that the argument character array ends in the unique EOF character '\0'. The `compute` function in the `SuffixArray` class returns you the suffix array; use that. Build the Burrows-Wheeler Transform (BWT) from the suffix array, also as a sequence of characters.
- Write code in the `decode` function for converting a BWT back to the original string. There are multiple ways to do this; we must use LF mapping for achieving the desired efficiency. Here's a sketchy outline on what can be done.
  - Create an array $R[\,]$, where $R[i]$ will store `rank`$(i, BWT[i])$. In C++, use a vector.
  - Create a character-integer map called C, which is the same as the C-structure used for LF mapping.
  - To populate R:
    - First create a character-integer TreeMap/map called `freq` that will store the frequency of each character in the BWT in sorted order of character.
    - Scan through BWT using a loop. Within the loop:
      - If `freq` contains $BWT[i]$, then increment `freq` of $BWT[i]$ by one, else set `freq` of $BWT[i]$ to one.
      - set $R[i]$ to `freq` of $BWT[i]$
  - To populate C, initialize a *counter* to 0. For each $< key, value >$ in `freq`:

    - Add $< key, counter >$ to C
    - Increment *counter* by *value*
  - Find the position of '\0' in BWT. Starting from this position, apply LF mapping (computed by using R and C) repeatedly to create the original string from the BWT. Check notes for more details.

    Caution: Notice that the function returns the string as an ArrayList/vector. Do not append at the beginning of the ArrayList/vector when creating it. Performance will be awful if you do that. So, add at the end and reverse.

Once you have implemented the two functions, you can uncomment the code in `testCorrectness()` for testing the `BWT` class.

6 Task 4: MTF

This has the following sub-tasks:

- The `encode` function will work pretty much in the same way as we discussed (in notes), except that the MTF returned will also contain a header information: *the size σ of the alphabet* and then *the alphabet in sorted order*. This will be followed by the Move-to-Front encoding itself. The header will be needed when you decode.

  For example, consider the string *spsssmmipissssiiii*. We will return the alphabet $[i, m, p, s]$ and the following move-to-front-code

  $$[3, 3, 1, 0, 0, 3, 0, 3, 3, 1, 3, 0, 0, 0, 1, 0, 0, 0]$$

  Therefore, this function will work as follows:

  - First obtain the sorted alphabet for the input sequence of characters (a function is given to you in `HelperFunctions`)
  - Now scan the input to create the move-to-front encoding and append that to MTF
  - Return the alphabet and the MTF you formed within an `MTFElement` object.
- For the `decode` function, first obtain the alphabet from the `MTFElement` object and then sort it. Now, use Move-to-Front encoding in the object and the logic in the notes for converting the Move-to-Front code back to the original string.

Once you have implemented the two functions, you can uncomment the code in `testCorrectness()` for testing the `MTF` class. You can also uncomment the code for load testing `BWT`.

7 Task 5: LZ78

This has the following sub-tasks:

- In the `encode` method, obtain the LZ78 encoding for the given argument. Use the encoding technique described in the notes. Assume that the sequence of characters ends in '\0'. Notice that the method returns an LZ78Element.[2]

[2]To create a TrieNode with a particular id, just call the TrieNode constructor with that id. C++ programmers remember to use dynamic allocation for creating a TrieNode pointer.

For any *node*, use `getChild`(c) to retrieve the *child* such that edge from *node* to *child* is labeled by the character c. Likewise, use `addChild`(id, c) to create a child with the desired id and edge-label.

C++ programmers should ideally call *delete root* at the end to prevent a memory leak.

- In the `decode` method convert the LZ78 pair back to the original string. Use either of the two decoding techniques in the notes; it is suggested that you use the improved version for better performance.

Once you have implemented the two functions, you can uncomment the code in `testCorrectness()` for testing the `LZ78` class.

We are done with LZ78 (and everything). At this point, uncomment the calls in `loadTest()` for LZ78 to check what output you are getting.

8 Task 6: Verify Statistics

Check the excel that I have provided to make sure the results that you are obtaining are consistent with what I have observed. Ideally, the compression ratios should not vary at all for BWT and LZ78; it may vary for LZ77 and LZSS if you are using different parameters for WINDOW, LOOKAHEAD, and THRESHOLD. The compression/decompression time can vary slightly.