COMP 261 (2021) - Assignment 5

Goal

In this assignment, you will implement two string search and two compression algorithms which have been discussed in lectures.
The assignment has four parts, each worth 25% (see the mark sheet linked below for a more detailed breakdown of marks).

Resources

starter code: code.zip
the data: data.zip
mark sheet: Marking schedule (from the ECS marking system)

To Submit

Each part needs to go into the specific files provided so you will upload back the Java files for each of the four parts that you complete. You can add additional files if that helps your implementation, but please avoid using packages and keep all the files in the same folder so that simple command-line `javac *.java` results in compiling everything required and produces the class files with the main methods as currently provided.
Each file has the main method that accepts arguments (such as the file name and if needed the search string) and we will test your programs using a script that will execute them on some test data --- if your program correctly returns the right results, you will get marks for each test that will translate to your final grades for this assignment.
NB! It is important not to modify the behaviour of the program and the arguments it expects. The output is always the result of the compression or decompression that we will assess either using the size or the content depending on the part of the assignment.
Please **don't** upload or submit the data files, as they are quite large!
The submission link can be found on the left.

Sample Data

Several data files have been provided to search and compress.
War and Peace. A novel by Tolstoy that is often used as sample text in machine learning. It is roughly 3 MB.
Taisho. A ninth century dictionary from the Chinese Buddhist Canon, a collection of texts written in Classical Chinese. It uses a very large alphabet and is about 3 MB in size.
Pi. The first million digits of Pi, totalling about 1 MB.
Lenna. A hexdump of the famous image of Lenna, often used as an example in image processing. It is small, only 300 kB. This makes it good for quickly testing your algorithms.
Apollo. A text version of the ``the eagle has landed'' sound recording, from Apollo 11. There are two channels, and hence two numbers at each time step. It is about 6 MB.

Part 1: String Search

In this section, your task is to implement the KMP string search algorithms to enable searching.
A method stub is provided in the `Search` class which you should fill in. The `search` method takes two arguments: the text to search through and the string to search for. The method returns an integer as follows:
The starting index of the first match in the text if one exists.
-1 if no match exists.
Note that KMP consists of two stages: computing the match table for input string, and performing the string search itself. You should write a separate method for each of these stages and structure the code appropriately.
Once you have the algorithm implemented, experiment with the provided files.
Use `KMP.java`. For example: `java KMP ../data/war_and_peace.txt astronomy`.

Part 2: Huffman coding

Your task in this part is to implement the Huffman coding and decoding algorithm, as described in lectures.
A full implementation does three things:
Create a tree of binary codes for each character in the input text.
Encode an input text using that tree.
Decode, using that tree, some encoded text.
You will need to write methods to do all three of these steps for a particular text. Remember, you need to dynamically generate the tree from a given input text, and *not* use a fixed tree that you supply manually. This means you need to analyse the text to create a frequency table for the characters used in the text.
Use the following assumptions when implementing the Huffman Tree:
Put 0 on the left branches and 1 on the right branches.
If the two nodes have the same frequency in the priority queue, pick first the nodes with the smallest characters alphabetically. Use Character.compare(..., ...) to compare them.
Here are some further implementation notes:
The `HuffmanCoding` class has three methods that you should fill in.
The `encode` method should return a *binary string*, i.e. a string containing only `1` 's and `0` 's. Similarly, `decode` takes a binary string as its argument.
You could store the binary codes for each character in a `Map<Character, String>` : useful for encoding (but don't use the dictionary to decode - use the tree, as per lecture, as that is much faster/more efficient).
One way to debug your code is to manually create an encoding tree and then generate a text using its frequencies.
Use `HuffmanCoding.java`. For example: `java HuffmanCoding ../data/war_and_peace.txt 0` (0 to print the tree, or use 1 to encode, or use 2 to encode/decode).
NB! Late update: When picking two subtrees with the same frequency, find the smallest symbol in each alphabetically, and pick the node with the smaller one as higher priority.

Examples

Below are a some small examples that you can use to check that your code works as expected. Be aware that the file-loading code will append a \n to the end of each line, so each file should be considered to have an extra \n at the end of it.
The output of the 0 (print tree) and 1 (encode) options are included. Option 2 should return the same text as in the input.
Files: test1.txt, test2.txt, test3.txt, test4.txt

test1.txt:

```
ABCDEF
```

A very simple example that checks that your nodes are ordered properly when they all have the same frequency.

```
0: {A=011, B=100, C=101, D=110, E=111, F=00, \n=010}
1: 01110010111011100010
```

test2.txt:

```
AABBCCCCDD\n
```

A slightly harder example that checks that you're searching the sub-trees properly for the lowest-valued character when two nodes have the same frequency.

```
0: {A=101, B=110, C=0, D=111, \n=100}
1: 10110111011000001111111100100
```

test3.txt:

```
施氏食獅史
```

Now to check that your code works for non-English characters.

```
0: {史=101, 獅=00, \n=100, 施=110, 食=01, 氏=111}
1: 1101110100101100
```

test4.txt:

```
Happy pride month (on platforms that support these emoji, anyway)!
🏳️‍🌈 🏳️‍⚧️
```

Aside from it being pride month, I picked these because they're ZWJ sequences. Each flag is actually three characters: the white flag emoji (U+1F3F3), zero-width-join (or "zwidge") (U+200D), and then a third code-point that specifies what the flag is. On systems that don't support ZWJs, you will see the first and last characters instead (zwidge doesn't display).
The below was conducted on the university systems (which, alas, does not have emoji support in the terminal):

```
0: {H=000100,\uDF08=010110, \n=010111, \u200D=01001, \uFE0F=11111,
  =011, !=000000, a=1000, d=000101, e=0010, f=000110, ♂=010001,
 (=000001, h=10100, )=000010, i=00110, j=000111, ,=000011,
 l=001110, m=10101, n=10110, o=1001, p=1100, r=10111, \uDFF3=01010,
 s=11010, t=1110, u=001111, w=010000, y=11011, \uD83C=11110 }

1:
000100100011001100110110111100101110011000010100100111010110011011011101010001100
001100110110011110000111010001110000110100110111101011101001111101010010001110011
101000111111001100100110111111001111101010000101101000100110010101011001000111001
000001101110001011011011010000100011011000010000000010111111100101011111010011111
010110011111100101011111010010100011111101011
```

Part 3: Lempel-Ziv compression

In this part, your task is to implement the Lempel-Ziv 77 compression and decompression algorithms, as described in lecture.
NB! Use the window size of 100 characters for your implementation.
Implementation notes:
The `LempelZiv` class has two methods you should fill in.
None of the provided data files include the characters `[`, `]`, or `|`. This means you can use them to start, end, and delimit your tuples respectively.
To debug your code, you could make some small files containing carefully constructed strings (all one character, one repetition, etc.) and check you get the expected result.
Use `LempelZivCompress.java` and `LempelZivDecompress.java`. For example: `java LempelZivCompress ../data/war_and_peace.txt > war_and_peace.short`.

Part 4: Challenge --- Boyer-Moore

Implement the Boyer-Moore string search algorithm, which is faster but more complex than KMP. It improves efficiency by doing a more complex search (building two tables, not just one) and starting from the end of a pattern instead of the start.
Use `BM.java`. For example: `java BM ../data/war_and_peace.txt astronomy`.

Writing it yourself

Make sure that you write the code for the data structures yourself -- you will not learn what you need to learn if you use code from somewhere else. You can build on code examples from somewhere else, but do not simply copy large segments of code, and make sure that you acknowledge the source appropriately. If we identify any plagiarism, we *will* penalise it!