FIT3155 S1/2023: Assignment 2
(Due night 11:55pm on Fri 05 May 2023)

[Weight: $20 = 8 + 8 + 4$ marks.]

Your assignment will be marked on the *performance/efficiency* of your program. You must write all the code yourself, and should not use any external library routines, except those that are considered standard.

Follow these procedures while submitting this assignment:

The assignment should be submitted online via moodle strictly as follows:

- All your scripts MUST contain your name and student ID.
- Use `gzip` or `Winzip` to bundle your work into an archive which uses your student ID as the file name.
  - Your archive should extract to a directory which is your student ID.
  - It should contain 3 subdirectories `q1/`, `q2/`, and `q3/`. Read each question's specification carefully and place your attempts in their corresponding subdirectories.
- Submit your archive electronically via Moodle.
- Strictly adhere to the specifications listed in each question.
- Do NOT hard-code input filenames in your solutions. These should be passed from the command line.

Academic integrity, plagiarism and collusion

Monash University is committed to upholding high standards of honesty and academic integrity. As a Monash student your responsibilities include developing the knowledge and skills to avoid plagiarism and collusion. Read carefully the material available at https://www.monash.edu/students/academic/policies/academic-integrity to understand your responsibilities. As per FIT policy, all submissions will be scanned via MOSS or JPLAG.

Generative AI not allowed!

This unit fully restricts you from availing/using generative AI to answer these assessed questions.

Question 1: Suffix tree to suffix array [8 Marks]

Given a string `str`$[1 \ldots n]$, write a program that computes its suffix array by first constructing its suffix tree and then generating a suffix array from it. You are free to assume that all input strings come from the restricted ASCII range $[37, 126]$.

Strictly follow the following specification to address this task:

Program name: `st2sa.py`

Argument to your program: A filename, whose contents are the input string `str`$[1 \ldots n]$.

- It is safe to assume that there are no line breaks in the input file, and that all characters `str`$[1 \ldots n]$ are lexicographically larger than the terminal character, \$, which you will append to `str`$[1 \ldots n]$ after reading it from the file.

Command line usage of your script:

```
python st2sa.py <filename of the input file containing str[1...n]>
```

Output file name: `output_sa.txt`

- Output format: The output file should contain the indexes corresponding to the suffix array of `str`$[1 \ldots n]$\$, one per line.

Example: If `str`$[1 \ldots n]$\$ = `mississippi`\$, then the output would be (in 1-based indexing):

```
12
11
8
5
2
1
10
9
7
4
6
3
```

Question 2: Burrows-Wheeler Transform (BWT) based compression (encoder/decoder) [8 Marks]

This question deals with compressing and uncompressing text using its Burrows-Wheeler Transform.

Specifically, you will first write an encoder (bwtzip) that will read an input file containing a string `str`$[1 \ldots n]$. (As in the previous question, you are free to assume that all input strings come from the ASCII range of $[37, 126]$.) This program will compute the input string's Burrows-Wheeler Transform `bwt`$[1 \ldots n+1]$ (which will include within it the special terminal character

$). After computing the BWT[1] of the input string, the program (encoder) will then perform a **runlength binary encoding** on the BWT string using Huffman codewords for encoding characters and Elias (Omega) codewords for encoding their runlengths. (See details below. Also, refer to your week 5 lecture slides for details of these coding schemes.)

You will also write a decoder (bwtunzip) to complement the above encoder, which reads the output generated from your bwtzip program and is then able to decode the runlength encoded BWT string losslessly and then invert the BWT string to recover the original string $\texttt{str}[1 \ldots n]$.

What is runlength encoding? Runlength encoding of any string (here, a BWT string) is an encoding based on the number of repeats of individual characters that appear successively when scanning the string from left to right. The observed characters and their corresponding runlengths can be encoded as tuples of the form $\langle \texttt{character}, \texttt{runlength} \rangle$. For example, the runlength encoding of `bbbcbbacc` results in the tuples:

$\langle \texttt{b}, 3 \rangle$, i.e. `b` appears over a run of length=3, followed by
$\langle \texttt{c}, 1 \rangle$, i.e. `c` appears over a run of length=1, followed by
$\langle \texttt{b}, 2 \rangle$, i.e. `b` appears over a run of length=2, followed by
$\langle \texttt{a}, 1 \rangle$, i.e. `a` appears over a run of length=1, followed by
$\langle \texttt{c}, 2 \rangle$, i.e. `c` appears over a run of length=2.

Note: each character in the tuple is encoded using its computed variable-length Huffman codeword (in bits) whereas each positive integer is encoded using its computed variable-length Elias code (in bits). The collection of bits from this encoding process defines a continuous bit/binary stream. In other words, the goal is to write out the entire encoding in binary (i.e. as bits, and not as '0' and '1' characters).

To generate a fully-decodable bit stream, bwtzip needs to encode (as a continuous stream of bits) the following pieces of information in the enumerated order:

1. The length $n + 1$ of the BWT string encoded using Elias codeword for integers.
2. The number of **distinct** characters in the BWT string, and this number again is encoded using its corresponding Elias codeword.
3. For each distinct character in the BWT string (in any order of these characters):
   - the 7-bit ASCII code word of that character,
   - the length of its constructed Huffman code word, where the length is encoded using its Elias integer codeword, and
   - the Huffman codeword you computed for that character.

   (Note: The Huffman codewords for a given text string is not unique and can vary depending on the decisions made during the Huffman code tree construction.)
4. For each runlength encoded tuple derived on the BWT string (in left-to-right order):
   - the Huffman codeword of the character being encoded, and
   - the Elias codeword of its runlength.

Note that the collective information from items 1, 2 and 3 form the necessary 'header' of the encoding for it to be decodable, whereas the information in item 4 forms the 'data' part where compression is gained, especially when the text grows longer.

[1] See note at the start of page 5.

Strictly follow the following specification to address this question.

ENCODER SPEC:

Program name: `bwtzip.py`

Argument to your program: A filename, whose contents are the input string `str`$[1 \ldots n]$.

Command line usage of your script:

```
python bwtzip.py <filename of the input file containing the string>
```

Output file name: `bwtencoded.bin`

- Output format: The output is a **binary** stream of bits concatenating component codewords for all pieces of information enumerate in the previous page.

Encoding example:

```
Let INPUT FILE  contain the string str[1...6] = banana.
(Its BWT string would be bwt[1...7] = annb$aa.)

The OUTPUT FILE will contain the following stream of bits:
000111000100110000110110001001111011011100101001001000111110110010110111110010

The above binary stream encodes the following pieces of information
---------------------------  HEADER PART  --------------------------------
Length(bwt) = 7 => EliasCode(7) = 000111
nUniqChars(bwt) = 4 (i.e., 'a','b', 'n' '$') => EliasCode(4) = 000100
Encoding unique characters in the BWT and their constructed...
 ...Huffman codewords: 'a' = 0, 'b' = 110, 'n' = 10, '$' = 111:
 ASCII('a') = 1100001 EliasCode(codelen=1) = 1   HuffmanCode('a') = 0
 ASCII('b') = 1100010 EliasCode(codelen=3) = 011 HuffmanCode('b') = 110
 ASCII('n') = 1101110 EliasCode(codelen=2) = 010 HuffmanCode('n') = 10
 ASCII('$') = 0100100 EliasCode(codelen=3) = 011 HuffmanCode('$') = 111
-----------------------------  DATA PART  --------------------------------
Runlength encoded tuples of the BTW string annb$aa
 <'a',1> =>  HuffmanCode('a') = 0    EliasCode(runlen=1) = 1
 <'n',2> =>  HuffmanCode('n') = 10   EliasCode(runlen=2) = 010
 <'b',1> =>  HuffmanCode('b') = 110  EliasCode(runlen=1) = 1
 <'$',1> =>  HuffmanCode('$') = 111  EliasCode(runlen=1) = 1
 <'a',2> =>  HuffmanCode('a') = 0    EliasCode(runlen=2) = 010
---------------------------ENCODING COMPLETE!---------------------------

Illustration of the packed (binary) representation of the OUTPUT:
|--------|--------|--------|--------|--------|--------|--------|--------|
|Byte-1  |Byte-2  |Byte-3  |Byte-4  |Byte-5  |Byte-6  |Byte-7  |Byte-8  |
|--------|--------|--------|--------|--------|--------|--------|--------|
|00011100|01001100|00110110|00100111|10110111|00101001|00100011|11101100|
|--------|--------|--------|--------|--------|--------|--------|--------|
|--------|--------| Note: This illustrates the bit stream packed into bytes.
|Byte-9  |Byte-10 | Based on the length of the stream, you may have to pad
|--------|--------| additional '0' bits so that the final length is a perfect
|10110111|11001000| multiple of 8. The example here shows a stream of 78 bits.
|--------|--------| So, the last byte (Byte-10) is padded with two extra '0's.
```

Note about generating a BWT: In principle, you ought to be able to generate a BWT of any string in linear ($O(n)$) time through a proper implementation of Question 1. However, for Question 2, you are free to use any method to generate a BWT for the encoding task (including a naïve explicit suffix sorting approach using inbuilt sort, should you wish to do so) – the BWT generation part will be ignored from efficiency considerations. However, once the BWT is generated, you must focus on encoding the BWT correctly and efficiently, and (for the decoder – see below) you should focus on efficiently decoding the encoded BWT and inverting it efficiently using LF-mapping to retrieve the original string.
Note about writing/reading the binary file: On the previous page, the byte-wise representation of packed bits from the stream of variable-length codewords generated from the encoding process has been illustrated. For this question you will have to implement the logic of 'packing' bits into bytes before writing each fully packed byte to a file yourself, and cannot use inbuilt functions or other libraries to achieve this. (Similarly for decoding, you will have to handle reading bytes from the binary file and unpack them into bits to be able to decode the BWT.)

DECODER SPEC:

The decoder is a separate program to decode and recover the original string `str`$[1 \dots n]$ from a binary encoded file (produced by the encoder above).

Program name: `bwtunzip.py`

Arguments to your program: Output file generated by the encoder (bwtzip).

Command line usage of your script:

```
python bwtunzip.py <binary file generated by the encoder, bwtzip>
```

Output file name: `recovered.txt`

- If the input binary file contained the following bit stream:
  000111000100110000110110001001111011011100101001001000111110110010110111110010
- then, the output `recovered.txt` file will contain:
  banana

Question 3 Prove Elias Omega code is prefix-free [4 Marks]

This is a 'proof' question (and no programming is required for this, beyond submitting your proof in a pdf format.)

Elias Omega code (see Week 5 lecture slides) is a decodable code over all possible positive integers $\mathbb{Z}^+ = \{1, 2, 3, \dots \infty\}$. The goal of this question is for you to mathematically prove that no Elias codeword of any $n \in \mathbb{Z}^+$ can be a prefix of any other codeword for $m \neq n \in \mathbb{Z}^+$.

Strictly follow the following specification to address this task: You are required to type (not scan) your proof with clear arguments and submit a PDF file with the file name `proof.pdf`.

Make sure you typeset your proof either in LaTeX or Google/Word Doc before generating a PDF from it. In other words, do not scan or paste pictures of a handwritten proof, but rather typeset it.

```
-=o0o=-
  END
-=o0o=-
```