FIT3155 S1/2019: Assignment 3
(Due midnight 11:59pm on Sunday 2 June 2019)

[Weight: $20 = 10 + 5 + 5$ 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. The usual input/output and other unavoidable routines are exempted.

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. (STRICTLY AVOID UPLOADING `.rar` ARCHIVES!)
  - Your archive should extract to a directory which is your student ID.
  - This directory should contain a subdirectory for each of the three questions, named as `q1/, q2/, and q3/`.
  - Your corresponding scripts and work should be tucked within those subdirectories.
- Submit your zipped file electronically via Moodle.

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.

Assignment Questions

1. Write an **encoder** that implements Lempel-Ziv-Storer-Szymanski (LZSS) variation of LZ77 algorithm with the following specifications.

   Strictly follow the specification below to address this question:

ENCODER SPEC:

Program name: `lzss_encoder.py`

Arguments to your program: (a) An input `ASCII` text file.

(b) Search window size (integer) W

(c) Lookahead buffer size (integer) L

Command line usage of your script:

`lzss_encoder.py <input_text_file> <W> <L>`

Output file name: `output_lzss_encoder.bin`

- Output format: The output is a **binary** stream of bits that losslessly encodes the input text file over two parts: (i) the `header` part, and (ii) the `data` part. The information encoded in each of these two parts is given below:

  `Information encoded in the header part:`

  - Encode using variable-length Elias integer code (see slide 29 in lecture 9), the number of **unique** `ASCII` characters in the input text.
  - For each **unique** character in the text:
    - Encode using fixed-length 8-bit ASCII code the unique `character`.
    - Encode using variable-length Elias code the `length` of the Huffman code assigned to that unique `character`.
    - Concatenate the variable-length Huffman codeword assigned to that unique `character`.

  `Information encoded in the data part:`

  - Encode using variable-length Elias code the `total number` of Format-`0`/`1` fields (see slide 36 in lecture 9 slides) required to encode the input text.
  - Successively encode information in each Format-`0`/`1` field as:

    For Format-`0`: $\langle$`0`-bit, `offset`, `length`$\rangle$, where `offset` and `length` are each encoded using the variable-length Elias code.

    For Format-`1`: $\langle$`1`-bit, `character`$\rangle$, where `character` is encoded using its assigned variable-length Huffman code defined in the header.

Example: This example is a truncation of the example on slide 37 of week 9 lecture. Assume $W = 6, L = 4$.

Assume that the input file contained the following text:

$$aacaacabcaba$$

Note, there are 3 unique characters in the text, $\{a, b, c\}$. A feasible set of Huffman codewords for $\{a, b, c\}$ are $\{\texttt{1}, \texttt{00}, \texttt{01}\}$ respectively. Using LZSS approach we get the following Format-`0`/`1` fields:

$\langle \texttt{1}, a\rangle$, $\langle \texttt{1}, a\rangle$, $\langle \texttt{1}, c\rangle$, $\langle \texttt{0}, 3, 4\rangle$, $\langle \texttt{1}, b\rangle$, $\langle \texttt{0}, 3, 3\rangle$, and $\langle \texttt{1}, a\rangle$.

The <u>`header`</u> part will contain:

- the number of unique characters, 3 in this example, encoded using Elias code as `011`
- ASCII code of each unique character followed by the Elias code of the length of its assigned Huffman codeword, followed by the statement of the Huffman codeword:
  - Statement of a with Huffman codeword '1' of length 1: `01100001`, followed by `1`, followed by `1`
  - Statement of b with Huffman codeword '00' of length 2: `01100010`, followed by `010`, followed by `00`
  - Statement of c with Huffman codeword of '01' of length 2 : `01100011`, followed by `010`, followed by `01`

Thus, concatenating all of the above codes, the header part is encoded as:

`01101100001110110001001000011000110100 1`

The <u>`data`</u> part will contain:

- The encoding of the `total number` of Format-`0`/`1` fields. In this example, it is 7, encoded using Elias code as 000111.
- The encoded information of successive Format-`0`/`1` fields:
  - $\langle 1, a\rangle$ encoded as `11`,
  - $\langle 1, a\rangle$ encoded as `11`,
  - $\langle 1, c\rangle$ encoded as `101`,
  - $\langle 0, 3, 4\rangle$ encoded as `0011000100`,
  - $\langle 1, b\rangle$ encoded as `100`,
  - $\langle 0, 3, 3\rangle$ encoded as `0011011`, and finally
  - $\langle 1, a\rangle$ encoded as `11`.

Thus concatenating all codes in the data part, we get the encoding:

`000111111110100110001001000011011111`

Finally, concatenating the `header` and `data` parts gives the lossless encoding of the input text to be written out in binary:

`011011000011101100010010000110001101001000111111110100110001001000011011111`

2. Write a **decoder** that decodes the output of the above encoder.
   Strictly follow the specification below to address this question:

DECODER SPEC:

Program name: `lzss_decoder.py`

Arguments to your program: (a) Output file from your encoder program implemented in the question above.

Command line usage of your script:

`lzss_decoder.py <output_lzss_encoder.txt>`

Output file name: `output_lzss_decoder.txt`

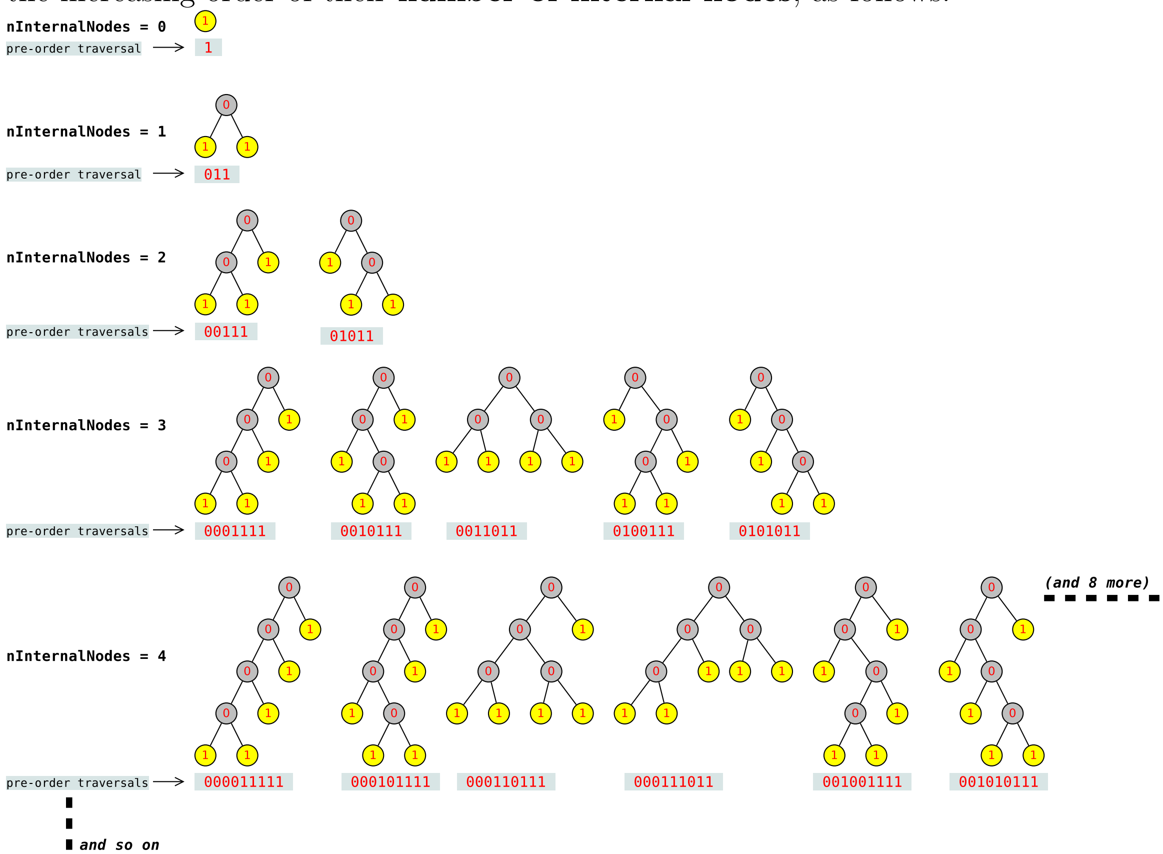
- Output format: The output is the decoded ASCII text.
- Example: If the input binary encoded file contained this bit stream:

  01101100001110110001001000011000110100100011111111010011000100100001101111

  the output file will decode the above as:

  aacaacabcaba

3. **Background for question 3:** A *full* binary tree is a binary tree where each **internal** node has **exactly** two children. Full binary trees can be enumerated systematically in the increasing order of their **number of internal nodes**, as follows:



The number of full binary trees with 0 internal nodes is 1 (see first row of the illustration above). The number of full binary trees with 1 internal node is also 1 (see second row). The number of full binary trees with 2 internal nodes is 2 (see third row). The number of full binary trees with 3 internal nodes is 5 (see fourth row). The number of full binary trees with 4 internal nodes is 14 (see fifth row, which shows the first 6 of 14). In general, the number of full binary trees with N internal nodes is given by the formula $\frac{(2N)!}{(N+1)!N!}$.

One could uniquely associate a **variable-length** bit string with each full binary tree, based on its **pre-order** traversal. In such a traversal, an internal node is associated with bit 0, and a leaf node is associated with bit 1. The illustration above gives the bit string underneath each tree corresponding to the pre-order traversal of that tree.

Furthermore, in the illustration above, in each row, notice that the bit strings corresponding to trees containing the *same* **number of internal nodes** are of the same length and appear in a lexicographically **sorted order**.

Based on this background, the goal of this exercise is as follows. Given some N, enumerate the full binary trees (represented by their traversal-based bit strings) containing

$0, 1, \ldots, k, \ldots, N$ intermediate nodes. Note again, for each $0 \leq k \leq N$, the bit strings (i.e., full binary trees) are enumerated lexicographically.

Strictly follow the specification below to address this question:

Program name: `enumerate.py`

Argument to your program: N (Assume N comes from the range $[0, 15]$).

Command line usage of your script:

```
enumerate.py <N>
```

Output file name: `output_enumerate.txt`

- Output format of each line of the output:

  ```
  <tree number>  <bit string associated with its pre-order traversal>
  ```

- Example output for $N = 4$ (meaning, we are enumerating trees with $\{0, 1, 2, 3, 4\}$ internal nodes):

  ```
   1      1
   2      011
   3      00111
   4      01011
   5      0001111
   6      0010111
   7      0011011
   8      0100111
   9      0101011
  10      000011111
  11      000101111
  12      000110111
  13      000111011
  14      001001111
  15      001010111
  16      001011011
  17      001100111
  18      001101011
  19      010001111
  20      010010111
  21      010011011
  22      010100111
  23      010101011
  ```

- Note: When submitting files on moodle, include any output file corresponding to a value of $N \leq 10$. Anything higher, the output file will be very large.

-=o0o=-
END
-=o0o=-

Non-examinable: Notice that as $N \to \infty$, the enumeration above maps positive integers $[1, \infty]$ to a new set of (tree-based) variable-length prefix-free integer codewords. This is another integer encoding scheme beyond the one you have learnt in Lecture 9. An obvious exercise that emerges from question 3 is, given any tree-based encoding of an integer, decode its corresponding positive integer. Hopefully some of you will try this problem after your exams, as an intellectual challenge.