FIT3155 S1/2020: Assignment 3
(Due midnight 11:59pm on Sunday 14 June 2020)

[Weight: $20 = 5 + 15$ 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 permitted as stated within the tasks.

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 two questions, named as `task1/ and task2/`.
  - 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

Task 1 In this task you are writing a program to generate a random prime number that is k-bits long, where k is the argument to your program. The recipe to generate such a prime number is given below:

1. Uniformly pick any random number $n \in [2^{k-1}, 2^k - 1]$
2. Test primality of n using Miller-Rabin's randomized algorithm.

3. If n is prime, print to the terminal the decimal value of n and stop. Else repeat from 1.

To undertake this task, you will have to implement Miller-Rabin primality testing algorithm introduced in Week 8. For uniform random number generation, you are allowed to '`import random`' in your script and use its functions.

Before you implement, you may want to read this (non-examinable) theoretical consideration: Slide 21 of your week 8 lecture mentions the prime number distribution function, $\pi(n)$, that gives the number of prime numbers $\leq n$. Asymptotically, this function tends to $n/\log(n)$. Using this approximation for $\pi(n)$, you can estimate the probability of picking a prime number n between $[2^{k-1}, 2^k - 1]$. This will reveal to you a good estimate of the *expected* number of iterations before finding a prime number that terminates your program.

Strictly follow the specification below to address this question:

Program name: `genPrime.py`

Argument to your program: k

Command line usage of your script:

```
genPrime.py <k>
```

Output: Output to terminal the value of prime n in decimal (which would take k-bits to represent in binary).

- When, $k = 4$, the program will output either 11 or 13. Since the choices are random, the output is not deterministic.

Task 2 Write an **encoder** and **decoder** that implements Lempel-Ziv-Storer-Szymanski (LZSS) variation of LZ77 algorithm with the following specifications. For this task you are free to use Python's `bytearray` or `bitarray`, whichever suits your needs.

Strictly follow the specification below to address this question:

ENCODER SPEC: [10 marks]

Program name: `encoder_lzss.py`

Arguments to your program:
1. An input `ASCII` text file.
2. Search window size (integer) W
3. Lookahead buffer size (integer) L

Command line usage of your script:

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

Output file name: `output_encoder_lzss.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 the number of **unique** `ASCII` characters in the input text using variable-length Elias ω integer code (see slides 24-30 in lecture 9).

- For each **unique** character in the text:
  - Encode the unique `character` using the fixed-length 7-bit ASCII code. (All input characters will have ASCII values < 128.)
  - Then encode the `length` of the Huffman code assigned to that unique `character` using variable-length Elias ω code.
  - To the above, append the variable-length Huffman codeword assigned to that unique `character`.

`Information encoded in the data part:`

- Encode using variable-length Elias ω code the `total number` of Format-0/1 fields (see slide 38 in lecture 9 slides) required to encode the input text.
- Successively encode information in each Format-0/1 field as follows:

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

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

Encoding example: This example is a truncation of the example on slide 39 of your 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 $\{1, 00, 01\}$ respectively. Using LZSS approach we get the following Format-0/1 fields:
$\langle 1, a\rangle$, $\langle 1, a\rangle$, $\langle 1, c\rangle$, $\langle 0, 3, 4\rangle$, $\langle 1, b\rangle$, $\langle 0, 3, 3\rangle$, and $\langle 1, a\rangle$.
The `header` part will contain:

- the number of unique characters, 3 in this example, encoded using Elias ω code as 011
- ASCII code of each unique character followed by the Elias ω 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: 1100001, followed by 1, followed by 1
  - Statement of b with Huffman codeword '00' of length 2: 1100010, followed by 010, followed by 00
  - Statement of c with Huffman codeword of '01' of length 2 : 1100011, followed by 010, followed by 01

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

$$01111000011111000100100011000110100 1$$

The `data` part will contain:

- The encoding of the `total number` of Format-0/1 fields. In this example, it is 7, encoded using Elias ω 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:

011110000111110001001000110001101001000111111110100110001001000011011111

The above has to be written out as a binary stream (packed into Bytes) as shown below:

Byte-1	Byte-2	Byte-3	Byte-4	Byte-5	Byte-6	Byte-7	Byte-8	Byte-9
01111000	01111100	01001000	11000110	10010001	11111110	10011000	10010000	11011110

Note: Since you are packing this into bytes, depending on the length of your stream, you may have to pad the encoded stream with additional 0 bits so that the length becomes a perfect multiple of 8 (bits). For example, the encoded stream above is 71 bits long, which is 1 bit short of 72 (= 9*8 bits = 9 bytes). So, you have to pad Byte-9 with an extra 0 (shown in green above).

DECODER SPEC: [5 marks]

Program name: `decoder_lzss.py`

Arguments to your program: Output file from your encoder program.

Command line usage of your script:

`decoder_lzss.py <output_encoder_lzss.bin>`

Output file name: `output_decoder_lzss.txt`

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

  011110000111110001001000110001101001000111111110100110001001000011011111

  the output file will decode the above as:

  aacaacabcaba

----------=o0o=----------
THE END
&
BEST OF LUCK FOR YOUR EXAM
----------=o0o=----------