FIT3155 S1/2021: Assignment 3
(Due midnight 11:59pm on Friday 28 May 2021)

[Weight: $10 = 3.5 + 2.5 + 4$ 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 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 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. (3.5 marks) We say that a number p_1 is a *twin prime* if p_1 is prime and there exists another prime number p_2, such that $|p_1 - p_2| = 2$. That is, a twin prime is a prime number that is either 2 less or 2 more than another prime number. For instance, the first 3 twin prime pairs are $(3, 5)$, $(5, 7)$, and $(11, 13)$. Note that $(2, 3)$ is the only pair of primes with a prime gap of 1 but is not considered a twin prime pair, while 5 is the only prime that belongs to two twin prime pairs. Since every prime number greater than 3 is of the form $6x \pm 1$ where x is a natural number, all twin primes - excluding $(3, 5)$ - are of the form $(6x - 1, 6x + 1)$. In this task you must write a program to generate a random pairs of twin prime numbers, such that each prime in the pair is m-bits long, where $m \geq 3$ is the argument to your program.
   Your program should have the following high level structure:

```
1   # Pick a random number with m bits
2   Uniformly pick a random number n in [2^{m-1},2^m-1]
3   Test primality of n using Miller-Rabins randomised algorithm.
4   If n is prime:
5       If n is a twin prime:
6           output the decimal value of n and its twin and terminate
7       else:
8           repeat from line 2
9   else:
10      repeat from line 2
```

   To undertake this task, you will have to implement the Miller-Rabin algorithm introduced in Week 8 and to generate uniform random numbers, you are allowed to import python's `random` module and make use of its functions. However, to obtain full marks you will need to implement your own repeated squaring function.

   If the interval $[2^{m-1}, 2^m - 1]$ contains one, or more twin primes, your program should continue selecting numbers at random until it finds a twin prime pair. However, you may want to consider if it is possible to narrow the search space on line 2. Similarly, you should think carefully about how many witnesses to trial in each Miller-Rabin test in line 3.

   Before attempting this task you may want to read this (non-examinable) theoretical consideration: Slide 27 of your week 8 lecture mentions the prime number distribution function, $\pi(n)$, that gives the number of prime numbers $\leq n$. Asymptotically we have,

$$\pi(n) \sim \frac{n}{\ln(n)},$$

   and using this approximation for $\pi(n)$, you can estimate the probability of picking a prime number n between $[2^{m-1}, 2^m - 1]$.
   Similarly, we can define $\pi_2(n)$ to be the number of twin prime pairs such that each element in the pair $\leq n$ and it is conjectured that asymptotically,

$$\pi_2(n) \sim 2c_2 \frac{n}{[\ln(n)]^2},$$

   where $c_2 = 0.661618...$ is know as the twin prime constant. These results can be used to perform some 'back of the envelope' calculations to arrive at a good estimate of the *expected* number of iterations (the amount of times you pick a random number on line 2) before finding a twin prime that terminates your program.

Strictly follow the specification below to address this question:

Program name: `twin_prime.py`

Argument to your program: m

Command line usage of your script:

```
twin_prime.py <m>
```

Output file name: `output_twin_prime.txt`

Output format: The output is a text file containing both elements of the twin prime pair in decimal. The smallest of the primes in the pair should be written to the first line of the file while the larger of the two should be written to the second line.

Example: When, $m = 5$, the program will output either:

```
17
19
```

or,

```
29
31
```

Since the choices are random, the output is not deterministic.

NOTE: Questions 2 and 3 focus on the data compression algorithms we covered in week 9 and require you to work with bits. However, working with raw binary data in Python is rather tedious and so you may assume all binary data is represented as a string of '0's and '1's, i.e. we will represent 0 and 1 using the characters with ASCII codes 48 and 49 respectively. Due to this the compressed data in these questions will actually require **more** bits to represent than the uncompressed data, and in practice you should **never** do this. However, the implementation of the algorithms themselves is independent of our representation and so we will ignore this issue here.

2. (2.5 marks) It is common practice to compress data before sending it across a communications channel. To ensure that the messages can be encoded and decoded correctly, the sender and receiver simply need to agree upon a language, or a **protocol**, that dictates the compression method. However, even if the receiver knows how the data was compressed, they may need more than just the compressed data itself in order to retrieve the original message. For instance, if a stream of characters was encoded using a Huffman code, the recipient of the stream would need knowledge of the original alphabet, as well as the Huffman codes the sender assigned to each character. Without this information they would only be able to guess at the original message!

   To enable the use of various encoding schemes it is common to first send some supplementary data - often referred to as the **header** - that contains all the information required to decode the following data stream. The header itself has a standardised format (dictated by the transmission protocol) that allows all parties to interpret it without any additional information.

   Your task is to write a program to construct a header that contains all the information necessary to decode a string whose characters were encoded using Huffman codes. To construct this header, you will need to implement algorithms to construct Huffman and Elias ω codes for character and integer streams respectively. The structure of the header is detailed in the specification below.

   Strictly follow the specification below to address this question:

   Program name: `header.py`

   Argument to your program: An input text file containing a string `str`$[0 \dots n-1]$. It is safe to assume that:

   - `str` consists of ASCII characters characters in the range $[32, 127]$ and potentially newline characters (ASCII code 10).

   Command line usage of your script:
   `header.py <input_text_file>`

   Output file name: `output_header.txt`

   Output format: The output is a text file containing a header for the input string `str`. The header is a bitstring (see the note above which details how to represent the bits) made up of the following information:

   - The number of **unique** `ASCII` characters in `str` encoded using the corresponding Elias ω integer code.
   - 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 an Elias ω code.
  - To the above, append the variable-length Huffman codeword assigned to that unique `character`.

Example: Let the input string be,

$$aacaacabcaba$$

In this case we have 3 unique characters: a, which occurs 7 times, b, which occurs twice, and c, which occurs 3 times. A feasible set of Huffman codewords (any valid set of Huffman codes will be considered correct) for $\{a, b, c\}$ are $\{$`1`, `00`, `01`$\}$ respectively.

The `header` 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:

`011110000111110001001000110001101001`

3. (4 marks) Inspired by question 2, imagine that you are the recipient of a stream of characters that has been compressed/encoded using the Lempel-Ziv-Storer-Szymanski (LZSS) variation of the LZ77 algorithm. Your task is to write a LZSS **decoder - no** encoding is required - to recover the original string.

   For the purposes of this question you may assume that your decoder is given the output of an LZSS encoder that has been used to encode a string of ASCII characters.

   - The output of the encoder is a stream of bits (represented as '0' and '1') 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:

     `The information encoded in the header part is identical to the header in question 2. It contains:`

     – The number of **unique** `ASCII` characters in the input text encoded using an Elias ω integer code.

     – For each **unique** character in the text the header contains:

       ∗ The unique `character` using the fixed-length 7-bit ASCII code. (All input characters will have ASCII values < 128).

       ∗ Then the `length` of the Huffman code assigned to that unique `character`, encoded using an Elias ω code.

       ∗ Finally, append the variable-length Huffman codeword assigned to that unique `character`.

     `Information encoded in the data part:`

     – The `total number` of Format-`0`/`1` fields - encoded using Elias ω encoding - required to encode the input text.

     – This is followed by the corresponding sequence of Format-`0`/`1` fields. Each tuple in this sequence is encoded 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.

   Since it would be difficult to develop test cases for this question without writing an encoder you will be provided with a set of test cases for your decoder on Moodle. A subset of these test cases will be used to verify your solution at marking time.

   Strictly follow the specification below to address this question:

   Program name: `decoder_lzss.py`

   Arguments to your program: An input text file containing a string of '0's and '1's (without any line breaks) that represents the output of the encoder described above.

   Command line usage of your script:
   `decoder_lzss.py <encoder_output_file.txt>`

   Output file name: `output_decoder_lzss.txt`

- Output format: The output is the decoded ASCII text.

Example: Assume that we have a dictionary of length $W = 6$, a buffer of length $L = 4$, and that the string input to the **encoder** (you don't have to write any code for the encoding, it is given as input to your decoder) was:

$$aacaacabcaba$$

From question 2 we know that a feasible set of Huffman codewords for $\{a, b, c\}$ are $\{\texttt{1}, \texttt{00}, \texttt{01}\}$ respectively. Applying the LZSS approach would yield 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 $\langle \texttt{1}, a\rangle$.

Recall from question 2 that 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:

`011110000111110001001000110001101001`

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:
  - $\langle \texttt{1}, a\rangle$ encoded as `11`,
  - $\langle \texttt{1}, a\rangle$ encoded as `11`,
  - $\langle \texttt{1}, c\rangle$ encoded as `101`,
  - $\langle \texttt{0}, 3, 4\rangle$ encoded as `0011000100`,
  - $\langle \texttt{1}, b\rangle$ encoded as `100`,
  - $\langle \texttt{0}, 3, 3\rangle$ encoded as `0011011`, and finally
  - $\langle \texttt{1}, a\rangle$ encoded as `11`.

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

`0001111111101001100010010000110111 1`

Finally, concatenating the `header` and `data` parts gives the lossless encoding of the input string:

`0111100001111100010010001100011010010001111111101001100010010000110111 1`

If a file containing:

`0111100001111100010010001100011010010001111111101001100010010000110111 1`

was given as input to your decoder the output file should contain:

aacaacabcaba

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