FIT3155 S2/2018: Assignment 2
(Due midnight 11:59pm on Sunday 21 October 2018)

[Weight: $20 = 5 + 5 + 5 + 5$ 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 four questions, named as `q1/, q2/, q3/, and q4/`.
  - 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. Let $P_{100}, P_{99}, \cdots, P_k, \cdots, P_1$ be a **descending sequence** of 100 largest prime numbers less than some given N. (Assume $N \geq 542$.) Corresponding to these prime numbers let $C_{100}, C_{99}, \cdots, C_k, \cdots, C_1$ denote 100 composite numbers, where any C_k is of the form $C_k = P_k + 1$. Your task is to factorize each of these 100 composite numbers to their respective prime factors.

Strictly follow the specification below to address this question:

Program name: `factors.py`

Argument to your program: N (assume $N \geq 542$.)

Command line usage of your script:

```
factors.py <N>
```

Output file name: `output_factors.txt`

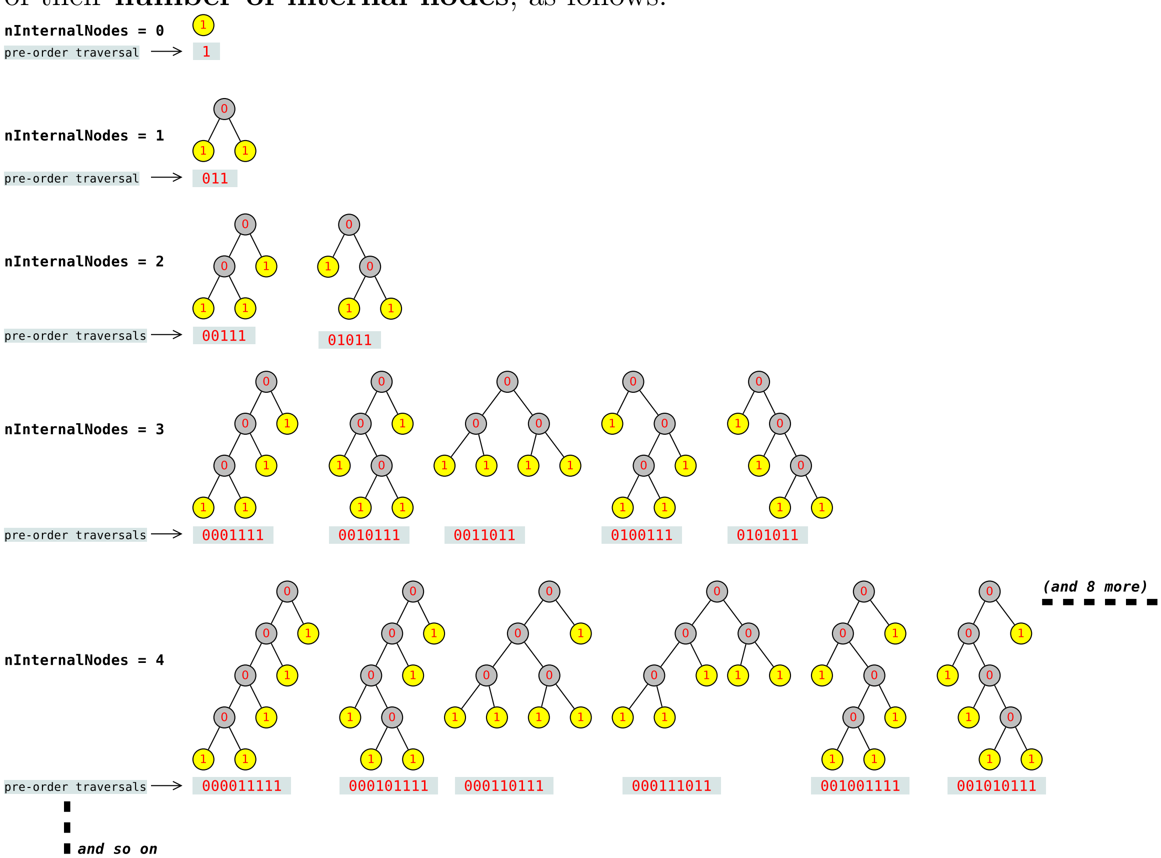
- Output format of each line of the output:

```
<C_1>    <prime factors of C_1>
<C_2>    <prime factors of C_2>
.... (and so on)
<C_100>  <prime factors of C_100>
```

- Example output for $N = 545$:

```
  3    3^1
  4    2^2
  6    2^1 x 3^1
  8    2^3
 12    2^2 x 3^1
 ...
522    2^1 x 3^2 x 29^1
524    2^2 x 131^1
542    2^1 x 271^1
```

2. **Background:** 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.

3. Building on the question above, notice from the example output for the previous question that we now have variable-length prefix-free code words associated with positive integers in the range [1,23]. In fact, as $N \to \infty$, we have variable-length code words for integers in the range $[1, \infty]$.

   In this exercise you are given a file containing a **concatenation** of variable-length code words of an arbitrary sequence of integers of $Z_1, Z_2, \cdots, Z_k$, all ≥ 1. Your program's goal is to **decode** this sequence of integers from their corresponding concatenated binary string.

   Strictly follow the specification below to address this question:

   Program name: `intseqdecode.py`

   Argument to your program: An input file containing a binary string corresponding to the **concatenation** of the variable-length code words of an arbitrary sequence of integers of $Z_1, Z_2, \cdots, Z_k$ (assume all ≥ 1).

   Command line usage of your script:
   `intseqdecode.py <inputfile>`

   Output file name: `output_intseqdecode.txt`

   - Output format: comma-separated decoded integers (in decimal): $Z_1, Z_2, Z_3, \ldots, Z_k$
   - Example: If the file contained the following concatenated bit string (where various concatenated parts are colored differently for convenience in reading):
     010011011011001010111010011101010101010101010101011
     The output file would contain the following sequence of decoded integers:
     21,2,15,8,23714

4. Write an **encoder** and **decoder** for the Lempel-Ziv-Storer-Szymanski (LZSS) variation of LZ77 algorithm, discussed in week 9 lecture.

   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 <inputfile> <W> <L>`

Output file name: `output_lzss_encoder.txt`

- Output format: The output is a binary encoded string corresponding to the input text.
  - Note: the output binary string concatenates the binary encoding of information of the form: (refer slide 38 of week 9 lecture)

    Format-0 $\langle$0-bit, `offset`, `length`$\rangle$

    Format-1 $\langle$1-bit, `character`$\rangle$.
  - `offset` and `length` are integers that are encoded using the Elias variable-length prefix-free code (refer slides 26-31 in week 9 lecture).
  - `character` is encoded using fixed-width ASCII code in binary.
- Example: If the input file contained the text:

  $$aacaacabcaba$$

  (which is a truncation of the example on slide 39 of week 9 lecture), the information that needs to be encoded is (assume $W = 6, L = 4$):
  $\langle 1, a\rangle$ encoded as `101100001`,
  $\langle 1, a\rangle$ encoded again as `101100001`,
  $\langle 1, c\rangle$ encoded as `101100011`,
  $\langle 0, 3, 4\rangle$ encoded again as `0011000100`,
  $\langle 1, b\rangle$ encoded as `101100010`,
  $\langle 0, 3, 3\rangle$ encoded again as `0011011`, and finally
  $\langle 1, a\rangle$ encoded as `101100001`.

  Therefore the output binary encoded string (which is a concatenation of all the above codes) is:

  `101100001101100001101100011001100010010110001000110111011000001`

DECODER SPEC:

Program name: `lzss_decoder.py`

Arguments to your program: (a) Output file from your encoder program above.

(b) Search window size (integer) W. (Same as the one used during encoding)

(c) Lookahead buffer size (integer) L. (Same as the one used during encoding).

Command line usage of your script:

lzss_decoder.py <output_lzss_encoder.txt> <W> <L>

Output file name: `output_lzss_decoder.txt`

- Output format: The output is plainly the decoded ASCII text.
- Example: If the input file contained:

  `101100001101100001101100011001100010010110001000110111011000001`

  the output file will decode the above as (same $W = 6, L = 4$ as during encoding):

  $$aacaacabcaba$$

-=o0o=-
END
-=o0o=-