FIT3155 S2/2025: Assignment 2
(Due date: 11:55pm Friday, 24 October 2025)

[Weight: $20 = 4 + 3 + 6 + 7$ marks.]

Assessment guidelines:

- You **must write all code yourself**. Use of Generative AI tools is not allowed.
- Any code **derived from online sources must be declared and fully cited**.
- Assessment is about your **understanding and ability to write code yourself**.
- You may be **called for an interview** to justify your understanding of your code.
- You should not use **external library routines**, except those considered standard or those that the questions specifically allow in the spec below.
  - Standard data structures (such as lists, dictionaries, tuples, and sets) are allowed if they do not conflict with the assessment objectives. Ensure their use is appropriate and efficient in terms of space and time.

Submission instructions:

- All your scripts **must** contain your name and student ID.
- Upload a .zip archive via Moodle with the filename of the form `<student_ID>.zip`.
  - Your archive should extract to a directory named as your Monash student ID.
  - This directory should contain a subdirectory for each of the questions, named as: `q1/`, `q2/`, `q3/`, and `q4/`.
  - The corresponding scripts and other question-specific reporting material (as stated in the question spec) should be placed inside these subdirectories.

Academic integrity

Monash University's academic integrity policy is accessible via this link: https://www.monash.edu/students/academic/policies/academic-integrity (click). Read it carefully to understand your responsibilities. As per FIT policy, all submissions will be scanned via MOSS (click) and/or JPlag (click).

Assignment Questions

1. **Generate a random d-digit prime number**: Your program will be given an integer input d, where $100 \leq d \leq 1000$. Your program should output a randomly chosen prime number that is exactly d digits long (when represented in base-10).

   In generating these random prime numbers, you must implement and use the Miller-Rabin primality test. Additionally, you are required to implement modular exponentiation using repeated squaring to compute x_0, the starting point of the sequence of numbers generated in the Miller-Rabin test (corresponding to Observation 2 in the algorithm) – refer lecture slides.

   Beyond modular exponentiation and primality testing, you are free to use built-in integer arithmetic operations. You can import the module `random` to assist with random number generation.

   Strictly follow the specification given below to address this question:

   Program name: `a2q1.py`

   Argument to your program: Integer d (where $100 \leq d \leq 1000$)

   Command line usage of your script:

   ```
   python a2q1.py <d>
   ```

   Do not hard-code the input filenames within your program. Include sufficient inline comments explaining each logical block of your code. Code that lacks adequate comments will require a face-to-face interview to verify your understanding before it can be assessed.

   Output file name: `output_a2q1.txt`

   - This should be a single-line file containing the d-digit prime number generated by your implementation. For example, when $d = 117$, a possible **random** output could be the following (note that, due to randomness, your code may produce a different result):

     ```
     490061338049126387557530735571315577728462381177979414454134004095857467337210092680837398434665292021197997239075453
     ```

   Written PDF Report: `report_a2q1.pdf`

   Write a brief report describing what approach you took in generating the d digit prime number.

2. **Fibonacci code for integers**: In this exercise, you will be implementing a variable-length prefix-free code for positive integers that relies on Fibonacci numbers.

   Starting from $F_1 = 1$ and $F_2 = 2$, subsequent Fibonacci numbers $F_{n\geq 3}$ are defined by the relationship $F_i = F_{i-1} + F_{i-2}$, giving the countably infinite set of Fibonacci numbers, $\{1, 2, 3, 5, 8, 13, 21, \ldots\}$.

   An interesting property we will employ in this question is that every positive integer $N \in \mathbb{Z}^+$ can be expressed **uniquely** as a sum of one or more distinct Fibonacci numbers in such a way that the sum does **not** contain two consecutive Fibonacci numbers from the infinite set. For example, $73 = 5 + 13 + 55 = F_4 + F_6 + F_9$.

   Thus, the inclusion (`1`) or exclusion (`0`) of each Fibonacci number in the sequence $\{F_1, F_2, F_3, \ldots\}$ within the unique sum representation of N can be encoded as a variable-length binary string, where the position of each bit corresponds to a specific Fibonacci number. For example, since $73 = F_4 + F_6 + F_9$, its shortest binary representation is `000101001`, where the 0s at positions $\{1, 2, 3, 5, 7, 8\}$ indicate the absence of $\{F_1, F_2, F_3, F_5, F_7, F_8\}$ in the sum, and 1s at positions $\{4, 6, 9\}$ indicate the presence of $\{F_4, F_6, F_9\}$ in the sum.

However, this encoding of a positive integer N is not prefix free. To obtain a prefix-free integer code, we append an additional `1` to the encoding, producing a codeword that always terminates with the bit-pattern . . . `11`. Below is a table of Fibonacci integer codewords for the first 10 positive integers where you can notice all codewords end in . . . `11`:

1	11
2	011
3	0011
4	1011
5	00011
6	10011
7	01011
8	000011
9	100011
10	010011

Since the Fibonacci sum representation forbids the use of two consecutive Fibonacci numbers, such a codeword can never appear as the prefix of another codeword, thereby ensuring the resulting encoding is prefix free.

Your task for this question is to write a program that reads a list of positive integers from an input file and outputs their corresponding Fibonacci codewords.

Strictly follow the specification given below to address this question:

Program name: `a2q2.py`

Argument to your program: Input filename.

- The input file will contain one positive number per line.
- You can assume the input file always has at least one number/line.

Command line usage of your script:

```
python a2q2.py <input filename>
```

Output file name: `output_a2q2.txt`

- One codeword per line corresponding to the order of numbers in the input file.

Example : If the input file contained the positive integers...

```
9
2
7
```

...the corresponding output file will contain:

```
100011
011
01011
```

Written PDF Report: `report_a2q2.pdf`

Characterize the worst-case time and space complexity of your implementation and briefly justify this.

3. Implementing Ukkonen's algorithm to construct a suffix tree:

For this task, you are required to implement Ukkonen's linear-time and linear-space algorithm (as discussed in Week 4) to **construct the suffix tree of any given string** and then **output its suffix array** derived from the suffix tree.

Your program will read the string $S[1 \ldots N]$ from an input file, whose name will be specified as a command-line argument. Every character in S falls within the ASCII range $[37, 126]$. Immediately after reading S from the file, a unique terminal character $ must be appended to S by your program, before proceeding with the construction.

Your program must also generate a **run log** that records specific information for each iteration during the construction process (see the example run log below).

Strictly follow the specification given below to address this question:

Program name: `a2q3.py`

Argument to your program: Input filename

- This file contains the input string $S[1...n]$ (without any line breaks).
- You are free to assume the input string is non-empty.

Command line usage of your script:

```
python a2q3.py <input filename>
```

Do not hard-code the input filename within your program. Ensure your program can be executed from the terminal (command line) by supplying the input filename (containing the string $S[1 \ldots n]$ as a command-line argument. Include sufficient inline comments explaining each logical block of your code. Code that lacks adequate comments will require a face-to-face interview to verify your understanding before it can be assessed.

Output file name: `output_a2q3.txt`

- Suffix array of the input string $S[1 \ldots N]$$, one number per line. For example, for $S[1 \ldots 6]$$ = `abaaba`$, the suffix array ouput will be the following:

```
7
6
3
4
1
5
2
```

Run log filename: `runlog_a2q3.txt`

- When constructing the suffix tree, your program should generate a run log text file that records key information about each step of the construction, as shown in the example below. Specifically, the log must include the phase number, the suffix extension number (of explicit extensions), the index of the active node,[1] the remainder substring range in the current state, the suffix link information of the active node, and the suffix link information of any newly created internal node (to be printed when it is resolved in the next extension). This information should be automatically produced by your program in the format shown in the example below. (Note, in the format shown below, an indent is 4 spaces. Also, by convention, root node is Node 1 and its suffix link points to itself.)

[1]Each node you create, whether internal or a leaf, should be assigned a numeric index in the order of its creation by the Ukkonen's algorithm, with the root node designated as Node 1.

```
Node 1 created: Internal node!

Phase 1 starts from Extn 1
    Extn 1 applies Rule 2 (alternate)
    Active Node = Node 1 (suffix link to Node 1); Remainder = EMPTY
        Node 2 created: Leaf node!

Phase 2 starts from Extn 2
    Extn 2 applies Rule 2 (alternate)
    Active Node = Node 1 (suffix link to Node 1); Remainder = EMPTY
        Node 3 created: Leaf node!

Phase 3 starts from Extn 3
    Extn 3 applies Rule 3
    Active Node = Node 1 (suffix link to Node 1); Remainder = EMPTY

Phase 4 starts from Extn 3
    Extn 3 applies Rule 2 (regular)
    Active Node = Node 1 (suffix link to Node 1); Remainder = S[3...3]
        Node 4 created: Internal node!
        Node 5 created: Leaf node!
    Extn 4 applies Rule 3
    Active Node = Node 1 (suffix link to Node 1); Remainder = EMPTY
        Linking Node 4 to Node 1

Phase 5 starts from Extn 4
    Extn 4 applies Rule 3
    Active Node = Node 4 (suffix link to Node 1); Remainder = EMPTY

Phase 6 starts from Extn 4
    Extn 4 applies Rule 3
    Active Node = Node 4 (suffix link to Node 1); Remainder = S[5...5]

Phase 7 starts from Extn 4
    Extn 4 applies Rule 2 (regular)
    Active Node = Node 4 (suffix link to Node 1); Remainder = S[5...6]
        Node 6 created: Internal node!
        Node 7 created: Leaf node!
    Extn 5 applies Rule 2 (regular)
    Active Node = Node 1 (suffix link to Node 1); Remainder = S[5...6]
        Node 8 created: Internal node!
        Node 9 created: Leaf node!
        Linking Node 6 to Node 8
    Extn 6 applies Rule 2 (alternate)
    Active Node = Node 4 (suffix link to Node 1); Remainder = EMPTY
        Node 10 created: Leaf node!
        Linking Node 8 to Node 4
    Extn 7 applies Rule 2 (alternate)
    Active Node = Node 1 (suffix link to Node 1); Remainder = EMPTY
        Node 11 created: Leaf node!
```

No further test cases will be provided. You will have to develop test cases yourself.

No PDF report is necessary for this question.

4. Compressed encoding of a suffix tree: The aim of this question is to encode a constructed suffix tree into a compact and self-contained ***binary*** file.[2] By *self-contained*, we mean that anyone who receives the encoded binary file should, in principle, be able to (1) decode the input string, and (2) reconstruct the suffix tree (without having to compute it from the input string again).

   Note:

   - You are **not** required to implement a decoder for this task. The focus of this question is solely on the encoding. However, implementing a decoder is an intellectually satisfying extension to pursue for motivated students (although not assessed).
   - Since the linear-time suffix tree construction is already assessed in Question 3, for this question you may use the FIT2004's quadratic-time construction method, if you are not sure that your (Q3's) implementation of the Ukkonen's algorithm is performing correctly.

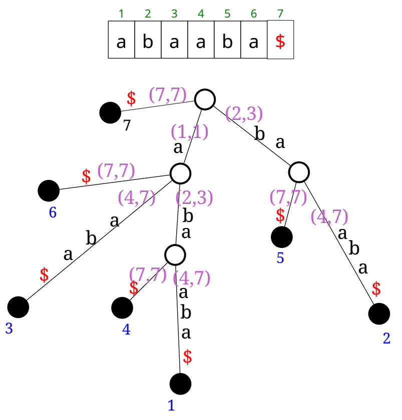
Encoding Method

The binary encoding required for this question is composed of two parts:

(I) The first part encodes the Burrows–Wheeler Transform (BWT) of the input string rather than the input string itself. This choice is because BWT is often more compressible than the original string, and the original input string can in turn be recovered by inverting the BWT.

(II) The second part encodes the suffix tree. This includes encoding the overall structure of the suffix tree as well as the labels (stored on its edges and at the leaf nodes).

To understand the method of encoding over these two parts, we will consider the concrete example of the string $S[1\dots6]\$ = \text{abaaba}\$$. The suffix tree of S is shown below:



The BWT of this string (including the terminal \$) is $\text{BWT}[1\dots7] = \text{abba\$aa}$.

[2]A binary file stores a stream of bits as a sequence of bytes (each consisting of eight bits). Encoding pieces of information generates a stream of bits, often derived by concatenating binary codewords (often of variable length) of the information you are encoding. These encoded files are not intended to be human-readable in a text editor but must instead be decoded by a separate program, called a decoder.

The goal now is to encode this BWT (which in turn is equivalent to encoding the original input string) as well as the suffix tree. To carry out this encoding, whenever we have to encode an integer we will be using the Fibonacci integer code (which you have implemented for Q2), and whenever we have to encode a character, we will mainly use the Huffman code (except once, where the 7-bit ASCII code has to be used). The details of when to use which code are specified in the sections below.

Part I: Encoding BWT

To generate a lossless encoding of the BWT string, the following pieces of information have to be encoded (in that specific order):

(a) The length of the BWT string (encoded using its Fibonacci integer codeword).

(b) The number of distinct characters in the BWT string (encoded using its Fibonacci integer codeword).

(c) For each distinct character in the BWT string (in any order):

- the distinct character (encoded using its 7-bit ASCII codeword),
- the length of its assigned Huffman codeword (using its Fibonacci integer codeword)
- the assigned Huffman codeword of that character.

(Note: Huffman codewords for a given text need not be unique and can vary depending on decisions made during Huffman tree construction.)

(d) Run-length encoding[3] of the BWT string. For each run-length tuple generated, encode:

- the run length using its Fibonacci integer codeword, and
- the character using its assigned Huffman codeword.

Using this encoding protocol, the encoding of Part 1 for the above example goes as follows:
(Note, your program does not have to produce this annotation; what you see below is merely to help you see the details of the encoding process.)

```
Information:

BWT: abba$aa

Distinct characters: ‘$’,‘a’,‘b’

Frequencies of distinct characters:
‘$’: 1
‘a’: 4
‘b’: 2

Assigned Huffman codewords for distinct characters
‘$’: 00
‘a’: 1
‘b’: 01
```

[3]Run-length encoding of any string is an encoding based on the number of repeats of individual characters that appear successively when scanning the string from left to right. For the BWT string “abba$aa”, scanning left to right, we notice one ‘a’, followed by two ‘b’s, followed by one ‘a’, followed by one ’$ followed by two ‘a’s. This yields the run-length tuples $\{\langle 1, \text{‘a’}\rangle, \langle 2, \text{‘b’}\rangle, \langle 1, \text{‘a’}\rangle, \langle 1, \text{‘\$’}\rangle, \langle 2, \text{‘a’}\rangle\}$, where each tuple is encoded using an integer code (encoding the length) and a character code (encoding the character).

```
Begin encoding Part 1:

Encode the LENGTH OF BWT: FibonacciCodeword(7) = 01011

Encode the NUMBER OF DISTINCT CHARACTERS observed in BWT: FibonacciCodeword(3) = 0011

For each DISTINCT character observed:
   Encode the DISTINCT CHARACTER '$' using its 7-bit ASCII code: ASCII('$') = 0100100
   Encode the LENGTH of its Huffman code assigned to the character '$': FibonacciCodeword(00) = 011
   Encode the HUFFMAN CODE assigned to the character '$': HuffmanCodeword('$') = 00

   Encode the DISTINCT CHARACTER 'a' using its 7-bit ASCII code: ASCII('a') = 1100001
   Encode the LENGTH of its Huffman code assigned to the character 'a': FibonacciCodeword(1) = 11
   Encode the HUFFMAN CODE assigned to the character 'a': HuffmanCodeword('a') = 1

   Encode the DISTINCT CHARACTER 'b' using its 7-bit ASCII code: ASCII('b') = 1100010
   Encode the LENGTH of its Huffman code assigned to the character 'b': FibonacciCodeword(01) = 011
   Encode the HUFFMAN CODE assigned to the character 'b': HuffmanCodeword('b') = 01


 Now, RUN LENGTH ENCODE the BWT string
Encode the run-length tuple <1,'a'>: FibonacciCodeword(1) = 11 : HuffmanCodeword('a') = 1
Encode the run-length tuple <2,'b'>: FibonacciCodeword(2) = 011 : HuffmanCodeword('b') = 01
Encode the run-length tuple <1,'a'>: FibonacciCodeword(1) = 11 : HuffmanCodeword('a') = 1
Encode the run-length tuple <1,'$'>: FibonacciCodeword(1) = 11 : HuffmanCodeword('$') = 00
Encode the run-length tuple <2,'a'>: FibonacciCodeword(2) = 011 : HuffmanCodeword('a') = 1
```

Concatenating all the codewords yields the Part I of the binary encoding that is needed:

01011001101001000110011000011111100010011011110110111111000111

Part II: Encoding the Suffix Tree

To encode any general tree's (branching) structure, a 2-bit-per-edge encoding is all one needs. Why? It is simple to observe that during any arbitrary tree traversal (starting from the root node), the traversal has to descend DOWN each edge in the tree exactly once and climbs back UP along it exactly once at some point in the traversal. Thus, by associating bit 0 to descending DOWN an edge and bit 1 to climbing back UP that edge during a tree-traversal, the resulting binary string uniquely represents the tree's branching structure.

For the suffix tree of the string $S[1\ldots6]\$ =$ abaaba$ shown on Page 6, **starting at the root node** and traversing in the lexicographic order of the edges in each node, results in the following the binary string: `0100101001011110010111`.

You will notice an extra 1 appended at the end of the string. Since you started at the root node (which technically is an internal node), that last 1 is suggesting to climb back UP from the root node, signifying the end of the encoding.

However, a suffix tree stores additional information that we want to encode. Specifically, for this question, we will additionally encode:

(a) the substring label on each edge, stored on each edge as two integers, and

(b) the suffix index stored in leaf node.

What follows now is an annotated traversal of the suffix tree shown on Page 6, with the list of codewords that are generated during the lexicographic traversal of that suffix tree starting from the root node, using the Fibonacci integer code to encode edge labels and leaf node labels at appropriate places during the traversal.

(Again, your program does not have to produce this annotation; what is provided below is for you to follow the encoding process.)

```
At INTERNAL node with untraversed edges: traverse DOWN along the untraversed edge with label (7,7):
    Codeword(DOWN)  = 0        FibonacciCodeword(7) = 01011        FibonacciCodeword(7) = 01011

Reached LEAF node with label/suffix index 7. Hence climb back UP; also encode suffix index
     Codeword(UP)  = 1        FibonacciCodeword(7) = 01011

At INTERNAL node with untraversed edges: traverse DOWN along the untraversed edge with label (1,1):
    Codeword(DOWN)  = 0        FibonacciCodeword(1) = 11        FibonacciCodeword(1) = 11

At INTERNAL node with untraversed edges: traverse DOWN along the untraversed edge with label (7,7):
    Codeword(DOWN)  = 0        FibonacciCodeword(7) = 01011        FibonacciCodeword(7) = 01011

Reached LEAF node with label/suffix index 6. Hence climb back UP; also encode suffix index
     Codeword(UP)  = 1        FibonacciCodeword(6) = 10011

At INTERNAL node with untraversed edges: traverse DOWN along the untraversed edge with label (4,7):
    Codeword(DOWN)  = 0        FibonacciCodeword(4) = 1011        FibonacciCodeword(7) = 01011

Reached LEAF node with label/suffix index 3. Hence climb back UP; also encode suffix index
     Codeword(UP)  = 1        FibonacciCodeword(3) = 0011

At INTERNAL node with untraversed edges: traverse DOWN along the untraversed edge with label (2,3):
    Codeword(DOWN)  = 0        FibonacciCodeword(2) = 011        FibonacciCodeword(3) = 0011

At INTERNAL node with untraversed edges: traverse DOWN along the untraversed edge with label (7,7):
    Codeword(DOWN)  = 0        FibonacciCodeword(7) = 01011        FibonacciCodeword(7) = 01011

Reached LEAF node with label/suffix index 4. Hence climb back UP; also encode suffix index
     Codeword(UP)  = 1        FibonacciCodeword(4) = 1011

At INTERNAL node with untraversed edges: traverse DOWN along the untraversed edge with label (4,7):
    Codeword(DOWN)  = 0        FibonacciCodeword(4) = 1011        FibonacciCodeword(7) = 01011

Reached LEAF node with label/suffix index 1. Hence climb back UP; also encode suffix index
     Codeword(UP)  = 1        FibonacciCodeword(1) = 11

At INTERNAL node but all edges already traversed. Hence climb UP
    Codeword(UP) 1

At INTERNAL node but all edges already traversed. Hence climb UP
    Codeword(UP) 1

At INTERNAL node with untraversed edges: traverse DOWN along the untraversed edge with label (2,3):
    Codeword(DOWN)  = 0        FibonacciCodeword(2) = 011        FibonacciCodeword(3) = 0011

At INTERNAL node with untraversed edges: traverse DOWN along the untraversed edge with label (7,7):
    Codeword(DOWN)  = 0        FibonacciCodeword(7) = 01011        FibonacciCodeword(7) = 01011

Reached LEAF node with label/suffix index 5. Hence climb back UP; also encode suffix index
     Codeword(UP)  = 1        FibonacciCodeword(5) = 00011

At INTERNAL node with untraversed edges: traverse DOWN along the untraversed edge with label (4,7):
    Codeword(DOWN)  = 0        FibonacciCodeword(4) = 1011        FibonacciCodeword(7) = 01011

Reached LEAF node with label/suffix index 2. Hence climb back UP; also encode suffix index
     Codeword(UP)  = 1        FibonacciCodeword(2) = 011

At INTERNAL node but all edges already traversed. Hence climb UP
    Codeword(UP) 1

At INTERNAL node but all edges already traversed. Hence climb UP
    Codeword(UP) 1
```

Now, concatenating all the codewords from the above process yields the Part II of the binary encoding:

00101101011101011011110010110101111001101011010111001100110011001011010111101101011010111111100110011001011010111
000110101101011101111

Writing out the encoding as a binary file

The binary encoding combining Parts I and II is written out into a binary file by you program as output. For the example above, the full binary stream (concatenating the encodings of Parts I and II) is as follows:

0101100110100100011001100001111110001001101111011011111100011100101101011101011011110010110101111001101011010111
0011001100110010110101111011010110101111111001100110010110101110001101011010111011 11

When you write out to the binary file, this file will contain the sequence of bytes, where each byte will hold eight successive bits from the above binary stream:

Byte-1	Byte-2	Byte-3	Byte-4	Byte-5	Byte-6	Byte-7	Byte-8	Byte-9
01011001	10100100	01100110	00011111	10001001	10111101	10111111	00011100	10110101
Byte-10	**Byte-11**	**Byte-12**	**Byte-13**	**Byte-14**	**Byte-15**	**Byte-16**	**Byte-17**	**Byte-18**
11010110	11110010	11010111	10011010	11010111	00110011	00110010	11010111	10110101
Byte-19	**Byte-20**	**Byte-21**	**Byte-22**	**Byte-23**	**Byte-24**	**Byte-25**		
10101111	11100110	01100101	10101110	00110101	10101110	11110000		

Notice, in the above example, the last Byte-25 was padded with extra 0s (shown in red) after the binary stream from the encoding ended. This was necessary because the total number of bits in the generated binary stream was not a perfect multiple of 8.

To enable wriing your bit stream into bytes, you are allowed to use the python library `bitarry`, but **exclusively** to store 8 bits as one byte and write it out. No other features that conflict with this assessment (e.g., creating Huffman codewords) can be used from this library.

Strictly follow the specification given below to address this question:

Program name: `a2q4.py`

Argument to your program: Input filename

- This file contains the input string $S[1...n]$ (without any line breaks).
- You are free to assume the input string is non-empty.

Command line usage of your script:

```
python a2q4.py <input filename>
```

Do not hard-code the input filename within your program. Ensure your program can be executed from the terminal (command line) by supplying the input filename (containing the string $S[1 \ldots n]$ as a command-line argument. Include sufficient inline comments explaining each logical block of your code. Code that lacks adequate comments will require a face-to-face interview to verify your understanding before it can be assessed.

Output file name: `output_a2q4.bin`

- This would be a binary file (containing a sequence of bytes, where each byte packs 8 bits from your encoding, as shown above).

No PDF report is needed for this question.

-=o0o=-
END
-=o0o=-