# FIT3155 S1/2024: Assignment 2
# (Due night 11:55pm on Fri 03 May 2024)

[Weight: $20 = 10 + 10$ 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.

## 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.
  - Your archive should extract to a directory which is your student ID.
  - It should contain 2 subdirectories `q1/` and `q2/`. Read each question's specification carefully and place your python scripts in their corresponding subdirectories.
- Submit your archive electronically via Moodle.
- Strictly adhere to the specifications listed in each question.
- Do NOT hard-code input filenames in your solutions. These should be passed from the command line.

## 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 or JPLAG.

### Generative AI not allowed!

This unit fully restricts you from availing/using generative AI to answer these assessed questions.

# Question 1: Compute ranks of a list of suffixes [10 Marks]

For any given string `str`$[1 \ldots n]$, the substring `str`$[i \ldots n]$ defines a suffix starting at position $i$.

Suppose a list of $m$ (distinct) positions $\{i_1, i_2, \ldots, i_m\}$ were provided to you, your task for this question is to write an *efficient* algorithm that can compute the lexicographic rank of the corresponding suffixes starting at each of those listed positons. An example is provided below.

Strictly follow the following specification to address this task:

**Program name:** `q1.py`

Give sufficient details in your inline comments for each logical block of your code. Uncommented code or code with vague/sparse/insufficient comments will automatically be flagged for a face-to-face interview with the CE before your understanding can be ascertained.

**Arguments to your program:** Your program takes two filenames as arguments, described below.

1. A `<stringFileName>` whose contents define the input string `str`$[1 \ldots n]$. It is safe to assume the following about this input file:
   - It will contain only a single line of ASCII characters defining the string.
   - All characters are in the ASCII range $[36, 126]$
   - The string contained in the input file will always terminate with the unique character `$` (ASCII 36).
2. A `<positionsFileName>` containing a list of positions $\{i_1, i_2, \ldots, i_m\}$ in the following format:
   - This file will contain $m$ lines, and each line gives a position of some suffix specified in 1-based indexing. (See example below.)
   - It is safe to assume the following about this input file:
     - The list of positions are non-redundant – meaning all positions in the input file are distinct (without repeats).
     - Each listed position is in the range $[1, n]$.

**Command line usage of your script:**

```
python q1.py <stringFileName> <positionsFileName>
```

Do not hard-code the filenames in your script. Ensure that we will be able to run your function from a terminal (command line) by supplying those arguments.

**Output file name:** `output_q1.txt`

- Output format: The output file should be in the same format as the input `<positionsFileName>`, except that each line in the output file contains the lexicographic rank of the corresponding suffix among the list of all suffixes. (See example below.)

**Example:** If `<stringFileName>` was a file that contained...

```
mississippi$
```

...and `<positionsFileName>` was a file that contained...

```
2
8
10
6
7
```

...then your program `q1.py` should output the file `output_q1.txt` containing:

```
5
3
7
11
9
```

**Explanation of the example output** : In the input list, the first listed position is 2. This position corresponds to the suffix `str`$[2 \ldots 12]$ = `ississippi$`. This suffix is lexicographically the fifth smallest (ranked) among all the suffixes in the string. The next listed position is 8. This position corresponds to the suffix `str`$[8 \ldots 12]$ = `ippi$`. This suffix is lexicographically the third smallest (ranked) among all the suffixes. Then the next listed position we see is 10. This position corresponds to the suffix `str`$[10 \ldots 12]$ = `pi$`. This suffix is lexicographically the seventh smallest (ranked), and so on.

# Question 2: Burrows-Wheeler Transform (BWT) based compression (encoder/decoder) [10 Marks]

This question deals with compressing and uncompressing text using its Burrows-Wheeler Transform.

Specifically, you will first write an encoder that will read an input file containing a string `str`$[1 \ldots n]$. (As in the previous question, you are free to assume that this string strictly contains characters in the ASCII range of $[36, 126]$.) Your task is first to compute the input string's Burrows-Wheeler Transform `bwt`$[1 \ldots n]$. After computing the BWT of the input string, the program (encoder) will then perform a **runlength binary encoding** on the BWT string using Huffman codewords for encoding characters and Elias (Omega) codewords for encoding their runlengths. (See details below. Also, refer to your week 5 lecture slides for details of these coding schemes.)

After writing your encoder, you will have to also write a decoder that reads the output generated from your encoder and is then able to decode the runlength encoded BWT string losslessly and further invert the BWT string to recover the original string `str`$[1 \ldots n]$.

**What is runlength encoding?** Runlength encoding of any string (here, a BWT string) is an encoding based on the number of repeats of individual characters that appear successively when scanning the string from left to right. The observed characters and their corresponding runlengths can be encoded as tuples of the form $\langle$`character`, `runlength`$\rangle$. For example, the runlength encoding of `bbbcbbacc` results in the tuples:

$\langle$b, 3$\rangle$, i.e. b appears 3 times, followed by
$\langle$c, 1$\rangle$, i.e. c appears 1 time, followed by
$\langle$b, 2$\rangle$, i.e. b appears 2 times, followed by
$\langle$a, 1$\rangle$, i.e. a appears 1 time, followed by
$\langle$c, 2$\rangle$, i.e. c appears 2 times.

Note: each character in the tuple is encoded using its computed variable-length Huffman codeword (in bits) whereas each positive integer is encoded using its computed variable-length Elias code (in bits). The collection of bits from this encoding process defines a continuous bit/binary stream. In other words, the goal is to write out the entire encoding in binary (i.e. as bits, and not as '0' and '1' characters). You are free to use the python library `bitarray` for this, should you choose to.

To generate a fully-decodable bit stream, the encoder needs to encode (as a continuous stream of bits) the following pieces of information in the enumerated order:

1. The length $n$ of the BWT string, and this should be encoded using Elias Omega codeword for integers.

2. The number of **distinct** characters in the BWT string, and this number again should be encoded using its corresponding Elias Omega codeword.

3. For each distinct character in the BWT string (in any order of these characters):

   - the fixed width 7-bit ASCII code word of that character,
   - the length of its constructed Huffman code word, where the length is encoded using its Elias Omega integer codeword, and
   - the Huffman codeword you computed for that character.

   (Note: The Huffman codewords for a given text string need not be unique/same all the time and can vary depending on the decisions made during the Huffman code tree construction.)

4. For each runlength encoded tuple derived on the BWT string (in left-to-right order):

   - the Huffman codeword of the character being encoded, and
   - the Elias codeword of its runlength.

Note that the collective information from items 1, 2 and 3 form the necessary 'header' of the encoding for it to be decodable, whereas the information in item 4 forms the 'data' part where compression is gained, especially when the text grows longer.

Strictly follow the following specification to address this question.

**ENCODER SPEC:**

**Program name:** `q2_encoder.py`

Give sufficient details in your inline comments for each logical block of your code. Uncommented code or code with vague/sparse/insufficient comments will automatically be flagged for a face-to-face interview with the CE before your understanding can be ascertained.

**Argument to your program:** A `<stringFileName>` containing input string `str`$[1 \ldots n]$. (Same assumptions given for Q1 apply here for this file.)

**Command line usage of your script:**

```
python q2_encoder.py <stringFileName>
```

Do not hard-code the filename in your script. Ensure that we will be able to run your function from a terminal (command line) by supplying a filename arguments.

**Output file name:** `q2_encoder_output.bin`

- Output format: The output is a **binary** stream of bits concatenating component codewords for all pieces of information enumerate in the previous page.

**Encoding example:**

```
Let <stringFileName>  contain the string str[1...7] = banana$.
Its corresponding BWT string would be   bwt[1...7] = annb$aa.

The q2_encoder_output.bin will contain the following stream of bits:
000111000100110000110110001001111011011100101001001000111110110010110111110010
(Note: different codewords are highlighted in varying colors above
for your convenience to eyeball those concatenated encoded terms.)

The above binary stream encodes the following pieces of information
--------------------------  HEADER PART  -------------------------------
Length(bwt) = 7 => EliasCode(7) = 000111
nUniqChars(bwt) = 4 (i.e., 'a','b', 'n' '$') => EliasCode(4) = 000100
Encoding unique characters in the BWT and their constructed...
 ...Huffman codewords: 'a' = 0, 'b' = 110, 'n' = 10, '$' = 111:
 ASCII('a') = 1100001 EliasCode(codelen=1) = 1   HuffmanCode('a') = 0
 ASCII('b') = 1100010 EliasCode(codelen=3) = 011 HuffmanCode('b') = 110
 ASCII('n') = 1101110 EliasCode(codelen=2) = 010 HuffmanCode('n') = 10
 ASCII('$') = 0100100 EliasCode(codelen=3) = 011 HuffmanCode('$') = 111
-----------------------------  DATA PART  --------------------------------
Runlength encoded tuples of the BTW string annb$aa
 <'a',1> =>  HuffmanCode('a') = 0    EliasCode(runlen=1) = 1
 <'n',2> =>  HuffmanCode('n') = 10   EliasCode(runlen=2) = 010
 <'b',1> =>  HuffmanCode('b') = 110  EliasCode(runlen=1) = 1
 <'$',1> =>  HuffmanCode('$') = 111  EliasCode(runlen=1) = 1
 <'a',2> =>  HuffmanCode('a') = 0    EliasCode(runlen=2) = 010
---------------------------ENCODING COMPLETE!---------------------------

Illustration of the packed (binary) representation of q2_encoder_output.bin:
|--------|--------|--------|--------|--------|--------|--------|--------|
|Byte-1  |Byte-2  |Byte-3  |Byte-4  |Byte-5  |Byte-6  |Byte-7  |Byte-8  |
|--------|--------|--------|--------|--------|--------|--------|--------|
|00011100|01001100|00110110|00100111|10110111|00101001|00100011|11101100|
|--------|--------|--------|--------|--------|--------|--------|--------|
|--------|--------| Note: This illustrates the bit stream packed into bytes.
|Byte-9  |Byte-10 | Based on the length of the stream, you may have to pad
```

```
|--------|--------| additional '0' bits so that the final length is a perfect
|10110111|11001000| multiple of 8. The example here shows a stream of 78 bits.
|--------|--------| So, the last byte (Byte-10) is padded with two extra '0's.
```

Note: You are free to use the `bitarray` library to help you output the binary stream in the packed format illustrated above.

## DECODER SPEC:

The decoder is a separate program to decode and recover the original string $\texttt{str}[1 \ldots n]$ from a binary encoded file (produced by the encoder above).

**Program name:** `q2_decoder.py`

Give sufficient details in your inline comments for each logical block of your code. Uncommented code or code with vague/sparse/insufficient comments will automatically be flagged for a face-to-face interview with the CE before your understanding can be ascertained.

**Arguments to your program:** The binary file generated by your encoder.

**Command line usage of your script:**

```
python q2_decoder.py <binary file generated by your encoder>
```

Do not hard-code the filename in your script. Ensure that we will be able to run your function from a terminal (command line) by supplying a binay file argument.

**Output file name:** `q2_decoder_output.txt`

- If the input binary file contained the following bit stream...

```
00011100010011000011011000100111101101110010100100100011111011001011011111001O
```

...then, `q2_decoder_ouput.txt` will contain:

```
banana$
```

```
-=o0o=-
  END
-=o0o=-
```