FIT3155 S1/2024: Assignment 1
(Due midnight 11:55pm on Thu 28 March 2024)

[Weight: 10 = 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. Standard data structures (e.g. list, dictionary, tuple, set etc.) that do not conflict with your assessment objectives are allowed, but ensure that their use is space/time efficient for the purpose you are using them. Also 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.
- Upload a .zip archive via Moodle with the filename of the form `<student_ID>.zip`.
  - Your archive should extract to a directory which is your Monash student ID.
  - This directory should contain a subdirectory for each of the two questions, named as: `q1/` and `q2/`.
  - Your corresponding scripts and work should be tucked within those subdirectories.

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 (click) to understand your responsibilities. As per FIT policy, all submissions will be scanned via MOSS (click) and/or JPlag (click).

Generative AI not allowed!

This unit fully prohibits you from using generative AI to answer these assessed questions.

Note: For this assignment assume that the input text and pattern strings are always drawn from the printable ASCII character set: see [Link]. Specifically, these are printable characters whose ASCII values are in the range [33, 126] (both inclusive).

Assignment Questions

1. **Boyer-Moore algorithm running the opposite direction**: In week 2 lecture, you learnt the Boyer-Moore's algorithm to find all exact occurrences of a pattern `pat`$[1 \dots m]$ in any given text `txt`$[1 \dots n]$. Recall that the algorithm you learnt involved scanning aligned characters from right to left in each iteration, while the pattern got *shifted* conceptually rightwards under the text between iterations.

   In this exercise, you will be implementing a version of Boyer-Moore that runs in the opposite direction, where the pattern is shifted **leftwards** under the text between iterations, from the rightmost end of the text towards the left. While doing so, in each iteration the scanning of characters between pattern and text proceeds from **left to right**. Specifically, in the first iteration, `pat`$[1 \dots m]$ and `txt`$[n-m+1 \dots n]$ are aligned, and the scanning is done left-to-right: `pat`$[1]$ with `txt`$[n-m+1]$, `pat`$[2]$ with `txt`$[n-m+2]$ and so on. Based on this left-to-right scanning, the pattern is then shifted leftwards under the text for the next iteration.

   Your implementation should employ all the corresponding rules you have learnt for the regular Boyer-Moore implementation, with the necessary changes required to run the algorithm in the opposite direction that we are pursuing in this exercise. (Note: to handle bad-character based shifts, implement the extended bad character shift rule.) Additionally, your implementation should include the optimization that avoids any unnecessary character comparisons between pattern and text (i.e. those you know from the previous iteration are already matched/identical).

   Strictly follow the following specification to address this question:

   Program/Function name: `q1.py`

   Arguments to your program/function: Two filenames, containing:

   (a) the string corresponding to `txt`$[1...n]$ (without any line breaks).
   (b) the string corresponding to `pat`$[1..m]$ (without any line breaks).

   Command line usage of your script:

   ```
   python q1.py <text filename> <pattern filename>
   ```

   Do not hard-code the filenames/input in your function. Ensure that we will be able to run your function from a terminal (command line) supplying any pair of text and pattern filenames as arguments. 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.

   Output file name: `output_q1.txt`

   - Each position (**in 1-based indexing**) where `pat` matches the `txt` (**in the order you compute them running Boyer-Moore the opposite way**) should appear on a separate line. For example, when text = `aaaabaaacaaaaa`, and pattern = `aaa`, the output should be:

     ```
     12
     11
     10
     6
     2
     1
     ```

2. **Yet another string pattern matching algorithm (that can also be implemented in hardware)**: In this exercise, you will attempt to perform a new exact pattern matching using a simple bitwise operations.

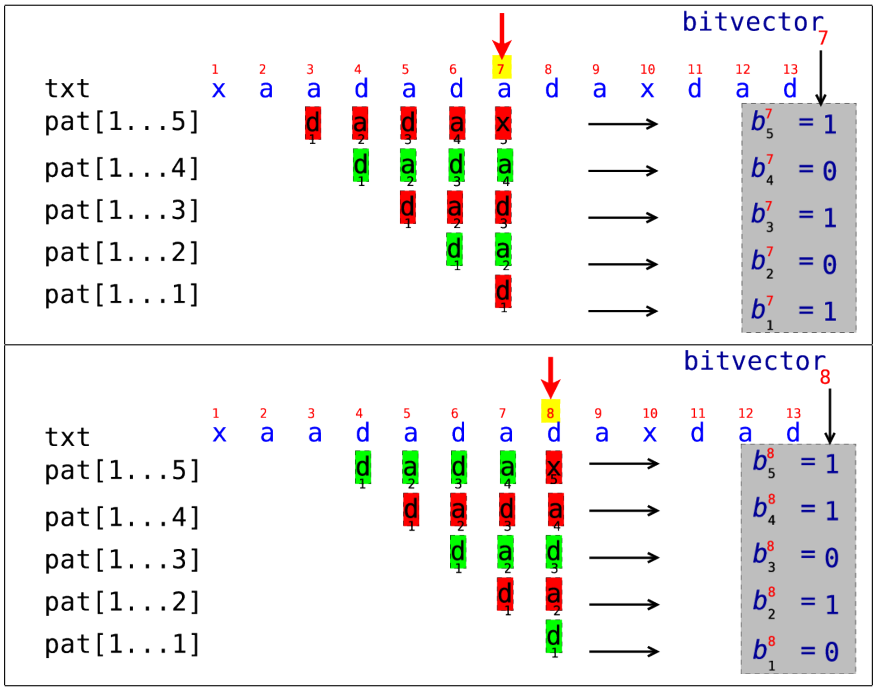
As before, if $\texttt{pat}[1 \dots m]$ be a pattern of length m and $\texttt{txt}[1 \dots n]$ some reference text of length n, we want to find all occurrences of `pat` that appear in `txt`. Now imagine that, for each position j in `txt`, there is a $\texttt{bitvector}_j$ containing m *bits* of the form:

$$\texttt{bitvector}_j = (b^j_m, b^j_{m-1}, b^j_{m-2}, \dots, b^j_1)$$

where, for each i decreasing from m to 1, we have:

$$b^j_i = \begin{cases} 1 & \text{if } \texttt{pat}[1\dots i] \text{ does } \textbf{not} \text{ match the region } \texttt{txt}[j-i+1\dots j]. \\ 0 & \text{if } \texttt{pat}[1\dots i] \textbf{ matches} \text{ the region } \texttt{txt}[j-i+1\dots j]. \end{cases}$$

For example, for $\texttt{txt}[1\dots 13] = \texttt{xaadadadaxdad}$ and $\texttt{pat}[1\dots 5] = \texttt{dadax}$, we get $\texttt{bitvector}_7 = (1,0,1,0,1)$ and $\texttt{bitvector}_8 = (1,1,0,1,0)$, as shown in the illustration below.



Key insight to drive exact pattern matching: While visually sliding the pattern left to right by one position at a time under the text, there is an interesting relationship that can be noticed between the `bitvectors` of any two successive positions. Specifically, any $\texttt{bitvector}_j$ can be derived from $\texttt{bitvector}_{j-1}$ by accounting for the relationship between the new character $\texttt{txt}[j]$ compared with each of the characters in the pattern, $\texttt{pat}[m \dots 1]$.

More formally, define another vector $\texttt{Delta}_j = (\delta^j_m, \delta^j_{m-1}, \delta^j_{m-2}, \dots, \delta^j_1)$ such that, for i decreasing from m to 1, these bits are computed as follows:

$$\delta^j_i = \begin{cases} 1 & \text{if } \texttt{txt}[j] \neq \texttt{pat}[i], \\ 0 & \text{otherwise.} \end{cases}$$

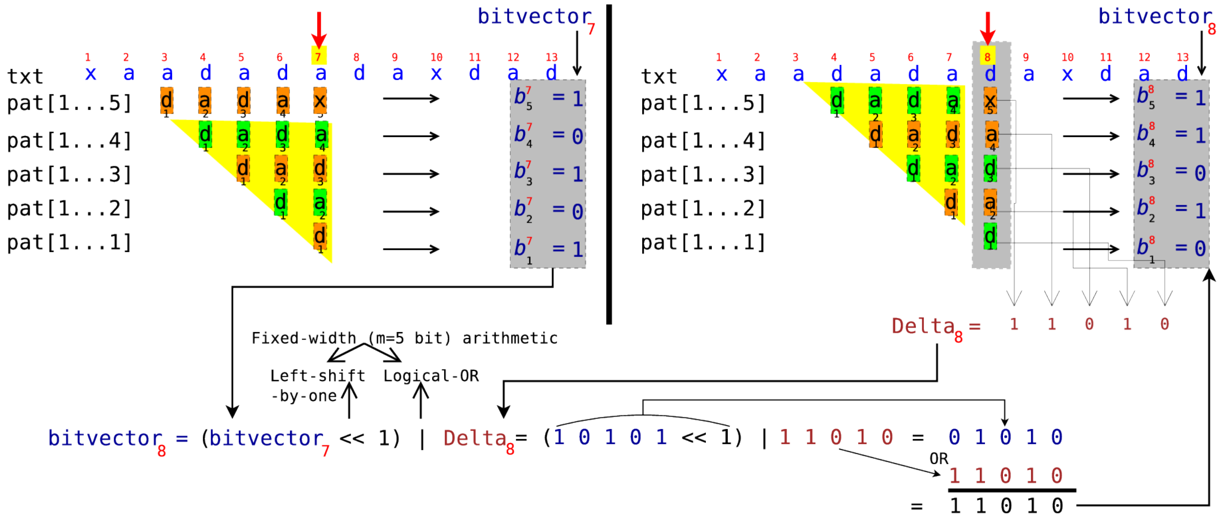
Then, the relationship between $\texttt{bitvector}_j$ and $\texttt{bitvector}_{j-1}$ in terms of $\texttt{Delta}_j$ is simply:

$$\texttt{bitvector}_j = (\texttt{bitvector}_{j-1} \ll 1) \mid \texttt{Delta}_j$$

where:

- $\ll 1$ is the m-bit logical LEFT-SHIFT-BY-ONE operator (i.e., the result after the shift operation remains a m-bit vector), and
- $\mid$ is the m-bit operator performing the logical inclusive OR operation between two m-bit vectors.

The illustration below provides a concrete example of this relationship:



Your task: Use the above insight to write a python program to compute the positions of all exact occurrences of any given `pat` in `txt`. In your implementation, amongst others, you will have to carefully consider how the m-bit vectors are represented and operated on. You are free to use the python bitarray library to store bit vectors and perform bitwise operations.

Strictly follow the following specification to address this question:

Program name: `q2.py`

Arguments to your program: Same as Q1, two plain text filenames:

(a) the first containing $\texttt{txt}[1...n]$ (without any line breaks).

(b) the second containing $\texttt{pat}[1..m]$ (without any line breaks).

Command line usage of your script:

```
python q2.py <text filename> <pattern filename>
```

Do not hard-code the filenames/input in your function. Ensure that we will be able to run your function from a terminal (command line) supplying any pair of text and pattern filenames as arguments. Give detailed 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.

Output filename: `output_q2.txt` (Format same as for Q1, except the indexes reported here are in the natural increasing order.)

-=o0o=-
END
-=o0o=-