FIT3155 S1/2021: Assignment 1
(Due midnight 11:59pm on Fri 26 March 2021)

[Weight: $10 = 4 + 6$ 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 two questions, named as: `q1/ and q2/`.
  - 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. **Approximate pattern matching:** The edit distance between two strings is the minimum number of edit operations (insertions, deletions, and substitutions) required to transform one string into the other. With this in mind, consider a variant of the exact pattern matching problem where the pattern is said to match the text at a given position if the *edit distance* between the pattern and the text is less than or equal to 1. This problem is an instance of an *approximate* pattern matching problem, which have a wide range of applications including in spell checkers and natural language processing.

   Given some text `txt`$[0...n-1]$ and a pattern `pat`$[0...m-1]$, your task is to write a program to find all positions within `txt`$[0...n-1]$ that match `pat`$[0...m-1]$ with an edit distance ≤ 1.
   Hint: You should use the Z-algorithm to address this question.

   Strictly follow the following specification to address this question:

   Program name: `editdist.py`

   Arguments to your program: Two plain text files:

   (a) an input file containing `txt`$[0...n-1]$ (without any line breaks).
   (b) another input file containing `pat`$[0..m-1]$ (without any line breaks).

   Command line usage of your script:

   ```
   editdist.py <text file> <pattern file>
   ```

   Do not hard-code the file names/input in your program. The pattern and text should be specified as arguments. Penalties apply if you do.

   Output file name: `output_editdist.txt`

   - Each position where `pat` matches the `txt` (with edit distance ≤ 1) should appear on a separate line. in the format
     `<position_in_txt> <edit_distance_with_pat>`
   - For example, when:
     `pat`$[0\ldots3]$ = `abcd` and `txt`$[0\ldots14]$ = `abdyabxdcyabcdz`,
     the output should be:

     ```
      0  1
      4  1
     10  0
     ```

     Note that positions 9 and 11 are missing from the above output even though the pattern matches the text at these positions with an edit distance of 1. This is because these matches can be considered *redundant* in the following sense: when there is an exact (edit distance = 0) match at position i in the text, there are ALWAYS edit distance = 1 matches at positions $i-1$ and $i+1$ (you should reason why). Thus, these matches are redundant and need not be reported in your output.
     Finally, it is possible for a region in the text to match the pattern in multiple ways under the edit distance ≤ 1 threshold. When this is the case you are only required to report any one of the possibilities.

2. **Binary Boyer-Moore**: Boyer-Moore performs extremely well on natural languages, but its effectiveness is somewhat diminished when searching binary files. Tables 1 and 2 show that when compared to naive pattern matching Boyer-Moore's advantage is much more pronounced for patterns and texts that consist of random lower case English characters than when the text and pattern are random binary strings. This should not be overly surprising, but we can improve Boyer-Moore's performance somewhat if we know in advance that both the pattern and the text will be defined over a binary alphabet.

   Your task is to implement an altered version of the Boyer-Moore algorithm that is optimised to deal with this particular application. You should assume that the text is very large compared to the pattern and to avoid the complications that arise when working with binary data[1] we'll take our alphabet to be $\aleph = \{0, 1\}$, i.e. the characters '0' and '1' with ASCII codes 48 and 49 respectively.

   When optimising your algorithm your goal should be to minimise the number comparisons it makes, **without** unreasonably sacrificing the space or time that the algorithm requires. In particular, you should be able to improve on the number of comparisons made by the general Boyer-Moore algorithm taught in lectures.

Algorithm	Number of comparisons	number of Shifts
Boyer-Moore	122614	117791
Naive algorithm	1040264	999991

Table 1: The performance of the Boyer-Moore and the naive pattern matching algorithms discussed in lectures on a random text and pattern of lower case English characters. The text and the pattern contained 1000000 characters and 10 characters respectively.

Algorithm	Number of comparisons	number of Shifts
Boyer-Moore	642096	298816
Naive algorithm	1998276	999991

Table 2: The performance of the Boyer-Moore and the naive pattern matching algorithms discussed in lectures on random bitstrings. The text and the pattern contained 1000000 characters and 10 characters respectively.

If you are struggling with this question consider taking the following approach:

- If you haven't already, consider quickly coding up the naive pattern matching algorithm to enable you to check the correctness of your implementation.
- Implement the full version of Boyer-Moore (including Galil's and the other standard optimisations) taught in lectures. This will serve as both a good template and reference point for your optimised version.
- Consider some example bitstrings as input and trace - using pencil and paper - the execution of the regular Boyer-Moore algorithm. Pay careful attention to the size of the shifts suggested by both the bad character and the good suffix rule and which shift is chosen by the algorithm each iteration. What do you notice?

[1] While avoiding raw binary data will save us some headaches it also somewhat limits the optimisations we can make, and you should ponder what further improvements we could make if we were working with raw binary data.

- Consider how you could leverage your discoveries from the previous step to optimise your implementation of Boyer-Moore. To help you test the effectiveness of your own optimisations one of the texts and its corresponding pattern used to gather the data for table 2 is available on Moodle.

Strictly follow the following specification to address this question:

Program name: `binary_boyermoore.py`

Arguments to your program: Two plain text files:

(a) an input file containing `txt`$[0...n-1]$ (without any line breaks), a string of characters from $\aleph$.

(b) another input file containing `pat`$[0..m-1]$ (without any line breaks), a string of characters from $\aleph$.

Command line usage of your script:

```
binary_boyermoore.py <text file> <pattern file>
```

Do not hard-code the file names/input in your program. The pattern and text should be specified as arguments. Penalties apply if you do.

Output: The number of comparisons made by your algorithm should be output to the terminal **and** you should write the result of your pattern matching to a file: `output_binary_boyermoore.txt`

- Each position where `pat` matches the `txt` should appear in a separate line. For example, when:
  `pat`$[0 \ldots 2]$ = `010` and `txt`$[0 \ldots 24]$ = `00110101011110010011011100`,
  the output should be:

```
4
6
14
```

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