FIT3155 S1/2025: Assignment 1 (updated 20 March 3pm) (Updated due date: 11:55pm Sunday, 6 April 2025)

The spec for Q1 as been updated to now remove the filtering of redundant matches. This is really a very minor update as the entire logic of Q1 remains unaffected. Nevertheless, the deadline for submission has been now been extended to midnight (11:55pm) on Sunday 6 April 2025.

[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.

Assignment Questions

1. **Near-exact pattern matching under DL-distance ≤ 1.** A string S can be transformed into a string T through a sequence of edit operations. Each edit operation can be any one of these **four** types:
(1) Replace an existing character in S with a new character (substitution).
(2) Swap two consecutive characters in S (transposition).
(3) Insert a new character in the string S (insertion).
(4) Delete an existing character in the string S (deletion).

   The **Damerau–Levenshtein (DL) distance** between two strings S and T is defined as the **minimum number of edit operations** (i.e., insertions, deletions, substitutions, and transpositions) required to transform S into T.[1] Worth also noting, that if $S = T$, the DL-distance between S and T is 0. Further, when the DL-distance is 1, only one edit operation (any one of the four types above) is sufficient to change S into T.

   In this exercise, your goal is to identify all (exact and near-exact) occurrences of a given pattern within the text under a DL-distance of ≤ 1. When DL-distance is 0 you have found an exact match. When DL-distance is 1, you have found a near-exact match.

   Your program will be given two 7-bit ASCII files as input, one containing the characters of the text and another containing the characters of the pattern. Your task is to write a program to find all positions in the text where the pattern matches the corresponding region of the text with a DL-distance ≤ 1.
Hint: You should use the Z-algorithm learnt in Week 1 to address this question.

   Strictly follow the following specification to address this question:

   Program/Function name: `a1q1.py`

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

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

   Command line usage of your script:

   ```
   python a1q1.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 Chief Examiner before your understanding can be ascertained.

   Output file name: `output_a1q1.txt`

   - The start position (reported in 1-based indexing, although your program will be computing in 0-based indexing) of each region in the text where the pattern matches with DL-distance ≤ 1, in the following format in each line of the output
`<position_in_txt> <DL-distance_with_pat>`

[1] In general, finding DL-distance between two strings is a fascinating Computer Science problem in itself, with a clever and concise Dynamic Programming solution. However, this specific assignment will NOT require you to implement the dynamic program, as it deals with identifying regions in text that match the pattern under a strict ≤ 1 DL-distance constraint. (See **hint** provided above.)

- For example, when:
  `pat`$[1 \ldots 4]$ = `abcd` and `txt`$[1 \ldots 20]$ = `abdcabxdcyabcdzabxcd`
  the output should be:

```
 1  1
 5  1
10  1
11  0
12  1
16  1
```

  Updated (20 March 2025): You are now asked to report all DL-distance ≤ 1 matches without filtering for redundencies (as stated originally before the update). As before, it is possible for a region in the text to match the pattern in multiple ways under the DL-distance ≤ 1 threshold, and you are only required to report any one of the possibilities.

Written PDF Report: `report_a1q1.pdf`

Write a brief report addressing these criteria:

- What the main idea behind your approach? (This should be in words; do not reproduce code here.)
- What is the worst-case space and worst-case time complexity of your approach? (Provide a brief justification.)

2. **Optimizing Boyer-Moore for binary strings**: In week 2, you have learnt the Boyer-Moore algorithm. In understanding this algorithm, it is instructive to assess how the overall number of character-comparisons (between opposing characters of pattern and text during right-to-left scanning) and the overall number of shifts (of pattern under the text) vary with the size of the alphabet from which text and pattern strings are drawn.

Table 1: The performance of the Boyer-Moore and the näive pattern matching algorithms implemented to handle text and pattern strings from any 7-bit ASCII alphabet. The comparison in the table below shows the number of character comparisons and number of shifts when handling a text and the pattern containing 1,000,000 characters and 10 characters respectively over two distinct cases. In the first case, the text and pattern were random strings drawn from the ASCII range [97, 122] (lower case English letters). In the second case, the text and pattern were random strings drawn from the ASCII range [48, 49] ('0' and '1' characters).

	Text and pattern composed of lower case English characters		Text and pattern composed of '0', '1' characters	
Algorithm	**Number of comparisons**	**Number of shifts**	**Number of comparisons**	**Number of shifts**
Boyer-Moore (general)	122,614	117,791	642,096	299,758
Näive algorithm	1,040,264	999,990	1,998,276	999,990

Tables 1 show that when compared to näive 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.

In this assignment exercise, we want optimize Boyer-Moore's performance if we knew 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 the binary pattern matching problem. 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 (in Python)[2] we will take our alphabet to be the ASCII characters '0' and '1' (i.e. characters whose ASCII values are 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.

Tips to guide your approach:

- If you have not already, you should code up the näive pattern matching algorithm to enable you to check the correctness of your implementation.
- Attend Week 3 Lab and implement the full version of Boyer-Moore (including the optimisations) taught in lectures.This will serve as both a good template and reference point for your optimised version you will implement in this exercise.
- Consider some example binary strings as input and work out – on 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.
- Consider how you could leverage your discoveries (if any) from the previous step to optimise your implementation of Boyer-Moore.

Strictly follow the following specification to address this question:

Program/Function name: `a1q2.py`

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

(a) the string corresponding to `txt`$[1 \ldots n]$ (without any line breaks).

(b) the string corresponding to `pat`$[1 \ldots m]$ (without any line breaks).

Command line usage of your script:

```
python a1q2.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 Chief Examiner before your understanding can be ascertained.

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_a1q2.txt`

- Each position where `pat` matches the `txt` should appear in a separate line. For example, when:
  `pat`$[0 \ldots 2]$ = `010` and `txt`$[0 \ldots 24]$ = `0011010101111001001101100`,
  the output file will contain:

[2]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.

```
 5
 7
15
```

Written PDF Report: `report_a1q2.pdf`

Write a brief report addressing this criteria:

- How does your binary version of Boyer-Moore differ from the version discussed in Week 2? (This should be in words; do not reproduce code.)

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