DS2030 Data Structures and Algorithms for Data Science
Week-8 Practice Set

October 14th, 2025

1 Brute Force Pattern Matching

Problem Statement

You are tasked with developing a simple substring search tool for text documents. The tool should check whether a pattern string occurs inside a larger text string. To achieve this, you will implement the **Brute Force Pattern Matching Algorithm**.
In this algorithm, the pattern is aligned with the text at every possible starting position. Characters are compared either left-to-right or right-to-left, and if all characters match at some alignment, the starting position is reported. Otherwise, the pattern is shifted by one position.

Tasks

Implement the Brute Force Pattern Matching with the following functionalities:

1. **Search for the first occurrence of a pattern (left-to-right)** Function: `brute_force_match(text, pattern)` Return the index of the first match, or -1 if not found. Print the character comparisons made.
2. **Search for the first occurrence of a pattern (right-to-left)** Function: `brute_force_match_r2l(text, pattern)` Return the index of the first match, or -1 if not found. Print the character comparisons made.
3. **Efficiency observation** For both approaches, print the total number of character comparisons. Compare results from left-to-right and right-to-left implementations.

Input

```
Text    : aaabaadaabaaa
Pattern : aabaaa
```

Expected Output (Left-to-Right)

```
Text    : aaabaadaabaaa
Pattern : aabaaa

Alignment 0:
Comparing text[0] vs pattern[0]     match
Comparing text[1] vs pattern[1]     match
Comparing text[2] vs pattern[2]     mismatch
Shift by 1

Alignment 1:
Comparing text[1] vs pattern[0]     match
Comparing text[2] vs pattern[1]     match
Comparing text[3] vs pattern[2]     match
Comparing text[4] vs pattern[3]     match
Comparing text[5] vs pattern[4]     match
Comparing text[6] vs pattern[5]     mismatch
Shift by 1

Alignment 2:
```

```
Comparing text[2] vs pattern[0]     match
Comparing text[3] vs pattern[1]     mismatch
Shift by 1

Alignment 3:
Comparing text[3] vs pattern[0]     mismatch
Shift by 1

Alignment 4:
Comparing text[4] vs pattern[0]     match
Comparing text[5] vs pattern[1]     match
Comparing text[6] vs pattern[2]     mismatch
Shift by 1

Alignment 5:
Comparing text[5] vs pattern[0]     match
Comparing text[6] vs pattern[1]     mismatch
Shift by 1

Alignment 6:
Comparing text[6] vs pattern[0]     mismatch
Shift by 1

Alignment 7:
Comparing text[7] vs pattern[0]     match
Comparing text[8] vs pattern[1]     match
Comparing text[9] vs pattern[2]     match
Comparing text[10] vs pattern[3]     match
Comparing text[11] vs pattern[4]     match
Comparing text[12] vs pattern[5]     match
Pattern found at index 7

Total comparisons = 24
```

Expected Output (Right-to-Left)

```
Text    : aaabaadaabaaa
Pattern : aabaaa

Alignment 0:
Comparing text[5] vs pattern[5]     match
Comparing text[4] vs pattern[4]     match
Comparing text[3] vs pattern[3]     mismatch
Shift by 1

Alignment 1:
Comparing text[6] vs pattern[5]     mismatch
Shift by 1

Alignment 2:
Comparing text[7] vs pattern[5]     match
Comparing text[6] vs pattern[4]     mismatch
Shift by 1

Alignment 3:
Comparing text[8] vs pattern[5]     match
Comparing text[7] vs pattern[4]     match
Comparing text[6] vs pattern[3]     mismatch
Shift by 1

Alignment 4:
Comparing text[9] vs pattern[5]     mismatch
Shift by 1

Alignment 5:
```

```
Comparing text[10] vs pattern[5]     match
Comparing text[9] vs pattern[4]     mismatch
Shift by 1

Alignment 6:
Comparing text[11] vs pattern[5]     match
Comparing text[10] vs pattern[4]     match
Comparing text[9] vs pattern[3]     mismatch
Shift by 1

Alignment 7:
Comparing text[12] vs pattern[5]     match
Comparing text[11] vs pattern[4]     match
Comparing text[10] vs pattern[3]     match
Comparing text[9] vs pattern[2]     match
Comparing text[8] vs pattern[1]     match
Comparing text[7] vs pattern[0]     match
Pattern found at index 7

Total comparisons = 21
```

2 Pattern Matching using the Bad Character Rule

Problem Statement

To reduce the number of unnecessary comparisons in pattern matching, the **Bad Character Rule (BCR)** is used. This algorithm compares the pattern with the text from right-to-left, and on a mismatch, shifts the pattern based on the last occurrence of the mismatched character in the pattern. If the mismatched character does not exist in the pattern, the pattern is shifted past that character entirely.

Tasks

Implement the Bad Character Rule with the following functionalities:

1. **Preprocessing** Function: `last_occurrence_table(pattern)` For each character in the pattern, store its last index of occurrence (or -1 if absent). Print this table.
2. **Pattern Search using BCR** Function: `bad_character_match(text, pattern)` Start by aligning the pattern with the text. Compare characters right-to-left. On mismatch, decide shift using the last occurrence table. Print each alignment, the comparisons made, and the shift applied. Return the index of the first match, or -1 if not found.

Input

```
Text    : aaabaadaabaaa
Pattern : aabaaa
```

Expected Output (Bad Character Rule)

```
Text    : aaabaadaabaaa
Pattern : aabaaa
last occurrence table: {'a': 5, 'b': 2}

Alignment at index 0:
  Compare text[5] vs pattern[5]     'a' vs 'a'     match
  Compare text[4] vs pattern[4]     'a' vs 'a'     match
  Compare text[3] vs pattern[3]     'b' vs 'a'     mismatch
  Shift = max(1, 3 - last('b')) = max(1, 3 - 2) = 1
```

```
Alignment at index 1:
  Compare text[6] vs pattern[5]     'd' vs 'a'     mismatch
  Shift = max(1, 5 - last('d')) = max(1, 5 - (-1)) = 6

Alignment at index 7:
  Compare text[12] vs pattern[5]     'a' vs 'a'     match
  Compare text[11] vs pattern[4]     'a' vs 'a'     match
  Compare text[10] vs pattern[3]     'a' vs 'a'     match
  Compare text[9]  vs pattern[2]     'b' vs 'b'     match
  Compare text[8]  vs pattern[1]     'a' vs 'a'     match
  Compare text[7]  vs pattern[0]     'a' vs 'a'     match
Pattern found at index 7

Total comparisons = 10
```