DS2030 Data Structures and Algorithms for Data Science
Lab 8

October 21st, 2025

Lab Instructions

- Create a folder named **"DS2030_<RollNo.>"** (all letters in capital) in **"home"** directory. Eg- **DS2030_142402022**
- Name the script files in the given format **"<your_roll_no>_<Name>_Lab8.py"**
- Make sure the **folder, files, classes, functions and attributes** are named as instructed in the lab sheet.
- We will not be able to evaluate your work if the folder is not properly named or is not located in the home directory.
- Make sure to save your progress before leaving the lab.
- Do not shut down the system after completing the lab.
- You are not allowed to share code with your classmates nor allowed to use code from other sources.

Pattern Matching: Brute Force vs Boyer-Moore

Problem Statement

You are to build a small pattern matching tool for string search in text documents. Given a text string and a pattern string, you will implement the Brute Force Pattern Matching algorithm (both left-to-right and right-to-left variants) and the Boyer-Moore algorithm (with Bad Character Rule and Good Suffix Rule). Compare their outputs and count the number of character comparisons made by each method.

Functionalities

Your program must support the following operations:

1. **Brute Force Pattern Matching (Left-to-Right)**
   - Function: `brute_force_match(text, pattern)`
   - Align the pattern with the text at every position from 0 to len(text) - len(pattern).
   - Compare characters left-to-right starting from the first character.
   - Print each alignment, character comparisons (with match/mismatch), and shifts.
   - Return the index of the first match or -1 if not found.
   - Print total character comparisons.
2. **Brute Force Pattern Matching (Right-to-Left)**
   - Function: `brute_force_match_r2l(text, pattern)`
   - Similar to above, but compare characters right-to-left (end of pattern first).
   - Print each alignment, character comparisons (with match/mismatch), and shifts.

- Return the index of the first match or -1 if not found.
- Print total character comparisons.

3. **Boyer-Moore Preprocessing (Bad Character Rule)**
   - Function: `last_occurrence_table(pattern)`
   - Create a dictionary where keys are characters in the pattern, and values are their last occurrence indices. Indices start from 1.
   - Print the table.
4. **Boyer-Moore Preprocessing (Good Suffix Rule)**
   - Function: `good_suffix_table(pattern)`
   - Create the shift table for good suffix shifts. This involves two cases:
   - Case 1: The matched suffix occurs again as a substring in the pattern. The shift aligns this other occurrence with the text.
   - Case 2: A prefix of the pattern matches a suffix of the matched suffix. The shift aligns this prefix.
   - Print the good suffix shift table.
   - Return the shift table.
5. **Boyer-Moore Pattern Matching**
   - Function: `boyer_moore_match(text, pattern)`
   - Use the last occurrence table and good suffix table to preprocess.
   - Compare characters right-to-left.
   - On mismatch, calculate the bad character shift and the good suffix shift, then shift by the maximum of the two.
   - Print each alignment, comparisons, and shift calculations.
   - Return the index of the first match or -1 if not found.
   - Print total character comparisons, and counts of bad and good shifts.

Data Structure Requirements

- Use Python lists for strings (treat as arrays).
- Use dictionaries for the last occurrence table.
- Do not use Pythons built-in string methods like `find()`, `in`, or regex.
- Implement all comparisons manually by indexing characters.

Instructions

- Do not add extra functions, attributes, or parameters.
- Strictly follow the function signatures provided.
- Include appropriate comments to document the code you have implemented.
- **Make sure to save the file before final submission to avoid discrepancies in the file used for evaluation.**

Sample Input 1

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

Sample Output 1 (Brute Force Left-to-Right)

```
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
```

Sample Output 1 (Brute Force Right-to-Left)

```
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
```

Sample Output 1 (Boyer-Moore Algorithm)

```
last occurrence table: {'a': 6, 'b': 3}
good suffix shift table: [4, 4, 4, 4, 1, 1, 1]
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
  Bad shift = 1
  Good shift = 1
  Shift = 1 using bad
Alignment at index 1:
  Compare text[6] vs pattern[5] -> 'd' vs 'a' -> mismatch
  Bad shift = 6
  Good shift = 1
  Shift = 6 using bad
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
Bad Character Heuristic used for 2 shifts.
Good Suffix Heuristic used for 0 shifts.

Summary: L2R found at 7, R2L at 7, BM at 7
```

Sample Input 2

```
Text : THIS IS A TEST TEXT
Pattern : TEST
```

Sample Output 2 (Brute Force Left-to-Right)

```
Alignment 0:
Comparing text[0] vs pattern[0] -> match
Comparing text[1] vs pattern[1] -> mismatch
Shift by 1
Alignment 1:
Comparing text[1] vs pattern[0] -> mismatch
Shift by 1
... (continues with mismatches until alignment 10)
Alignment 10:
Comparing text[10] vs pattern[0] -> match
Comparing text[11] vs pattern[1] -> match
Comparing text[12] vs pattern[2] -> match
Comparing text[13] vs pattern[3] -> match
Pattern found at index 10
Total comparisons = 15
```

Sample Output 2 (Brute Force Right-to-Left)

```
Alignment 0:
Comparing text[3] vs pattern[3] -> mismatch
Shift by 1
... (similar mismatches, fewer in some alignments due to RTL)
Alignment 10:
Comparing text[13] vs pattern[3] -> match
Comparing text[12] vs pattern[2] -> match
Comparing text[11] vs pattern[1] -> match
Comparing text[10] vs pattern[0] -> match
Pattern found at index 10
Total comparisons = 15
```

Sample Output 2 (Boyer-Moore Algorithm)

```
last occurrence table: {'T': 4, 'E': 2, 'S': 3}
good suffix shift table: [3, 3, 3, 3, 1]
Alignment at index 0:
  Compare text[3] vs pattern[3] -> 'S' vs 'T' -> mismatch
  Bad shift = 1
  Good shift = 1
  Shift = 1 using bad
Alignment at index 1:
  Compare text[4] vs pattern[3] -> ' ' vs 'T' -> mismatch
  Bad shift = 4
  Good shift = 1
  Shift = 4 using bad
Alignment at index 5:
  Compare text[8] vs pattern[3] -> ' ' vs 'T' -> mismatch
  Bad shift = 4
  Good shift = 1
  Shift = 4 using bad
Alignment at index 9:
  Compare text[12] vs pattern[3] -> 'S' vs 'T' -> mismatch
  Bad shift = 1
  Good shift = 1
  Shift = 1 using bad
Alignment at index 10:
  Compare text[13] vs pattern[3] -> 'T' vs 'T' -> match
  Compare text[12] vs pattern[2] -> 'S' vs 'S' -> match
  Compare text[11] vs pattern[1] -> 'E' vs 'E' -> match
  Compare text[10] vs pattern[0] -> 'T' vs 'T' -> match
Pattern found at index 10
```

```
Total comparisons = 8
Bad Character Heuristic used for 4 shifts.
Good Suffix Heuristic used for 0 shifts.
Summary: L2R found at 10, R2L at 10, BM at 10
```

Sample Input 3 (Pattern Not Found)

```
Text : AABAACAADAABAABA
Pattern : XYZ
```

Sample Output 3 (Brute Force Left-to-Right)

```
Alignment 0:
Comparing text[0] vs pattern[0] -> mismatch
Shift by 1
Alignment 1:
Comparing text[1] vs pattern[0] -> mismatch
Shift by 1
... (all alignments mismatch on first char)
Total comparisons = 14
Pattern not found
```

Sample Output 3 (Brute Force Right-to-Left)

```
Alignment 0:
Comparing text[2] vs pattern[2] -> mismatch
Shift by 1
... (similar, often 1 comparison per alignment)
Total comparisons = 14
Pattern not found
```

Sample Output 3 (Boyer-Moore Algorithm)

```
last occurrence table: {'X': 1, 'Y': 2, 'Z': 3}
good suffix shift table: [3, 3, 3, 1]
Alignment at index 0:
  Compare text[2] vs pattern[2] -> 'B' vs 'Z' -> mismatch
  Bad shift = 3
  Good shift = 1
  Shift = 3 using bad
Alignment at index 3:
  Compare text[5] vs pattern[2] -> 'A' vs 'Z' -> mismatch
  Bad shift = 3
  Good shift = 1
  Shift = 3 using bad
Alignment at index 6:
  Compare text[8] vs pattern[2] -> 'A' vs 'Z' -> mismatch
  Bad shift = 3
  Good shift = 1
  Shift = 3 using bad
Alignment at index 9:
  Compare text[11] vs pattern[2] -> 'A' vs 'Z' -> mismatch
  Bad shift = 3
  Good shift = 1
  Shift = 3 using bad
Alignment at index 12:
  Compare text[14] vs pattern[2] -> 'B' vs 'Z' -> mismatch
```

```
  Bad shift = 3
  Good shift = 1
  Shift = 3 using bad
Total comparisons = 5
Pattern not found
Bad Character Heuristic used for 5 shifts.
Good Suffix Heuristic used for 0 shifts.
Summary: L2R found at -1, R2L at -1, BM at -1
```

Starter Code

```
def brute_force_match(text, pattern):
    """
    Brute Force Pattern Matching (Left-to-Right).
    Hints:
    - Compute the length of text (n = len(text)) and the length of pattern (m = len(
    pattern)).
    - If the pattern is empty, return 0 immediately.
    - Initialize a counter for total comparisons and a variable for the found_index
    (-1 initially).
    - Loop over possible starting positions.
    - For each position, print the alignment.
    - Check for a match by comparing characters.
    - For each comparison, print whether it's a match or mismatch and increment the
    counter.
    - On mismatch, print "Shift by 1" and stop comparing for this alignment.
    - If a full match is found, print the found message, set the found_index, and stop
     searching.
    - Print the total comparisons.
    - If the pattern is not found after all alignments, print the 'Pattern not found'
    message.
    - Return the found_index.
    """
    pass

def brute_force_match_r2l(text, pattern):
    """
    Brute Force Pattern Matching (Right-to-Left).
    Hints:
    - Compute the length of text (n = len(text)) and the length of pattern (m = len(
    pattern)).
    - If the pattern is empty, return 0 immediately.
    - Initialize a counter for total comparisons and a variable for the found_index
    (-1 initially).
    - Loop over possible starting positions.
    - For each position, print the alignment.
    - Check for a match by comparing characters.
    - For each comparison, print whether it's a match or mismatch and increment the
    counter.
    - On mismatch, print "Shift by 1" and stop comparing for this alignment.
    - If a full match is found, print the found message, set the found_index, and stop
     searching.
    - Print the total comparisons.
    - If the pattern is not found after all alignments, print the 'Pattern not found'
    message.
    - Return the found_index.
    """
    pass

def last_occurrence_table(pattern):
    """
    Preprocessing: Last Occurrence Table for Bad Character Rule.
    Hints:
    - Create a dictionary for last occurrences.
    - For each character in the pattern, update its value in the dictionary (values
    are 1-based index).
    - Print the table in the format shown in samples (only characters present in the
    pattern).
    - Return the table.
    """
    pass

def good_suffix_table(pattern):

    m = len(pattern)
```

```
    # This is the "weak" good suffix rule.
    shift = [m] * (m + 1) # Default shift
    k_border = 0
    for i in range(1, m):
        if pattern[:i] == pattern[m-i:]:
            k_border = i
    for i in range(m + 1):
        shift[i] = m - k_border

    for k in range(1, m):
        suffix = pattern[m-k:]
        for j in range(m - k - 1, -1, -1):
            if pattern[j : j+k] == suffix:
                shift[m-k] = m - k - j # Note: index m-k
                break
    shift[m] = 1 # shift[m] corresponds to k=0

    # shift table tells how far to move the pattern to align the next possible
    matching suffix

    print(f"good suffix shift table: {shift}")
    return shift




def boyer_moore_match(text, pattern):
    """
    Pattern Matching using Boyer-Moore with Bad Character and Good Suffix Rules.
    Hints:
    - Compute the lengths of text (n) and pattern (m).
    - If the pattern is empty, return 0 immediately.
    - Build the last occurrence table and the good suffix table.
    - Loop over possible alignments.
    - For each alignment, compare characters from right to left.
    - On mismatch:
      - Calculate bad shift from the last occurrence table.
      - Get the good shift value  from the good suffix table.
      k = length of the matched suffix at the end of the pattern
      shift_table[m - k] gives the number of positions to shift the pattern
      - shift = max(bad_shift, good_shift)
      - Print the bad and good shift calculations, and which one was chosen.
      - Increment bad_shifts or good_shifts counter based on which value was larger.
      - Stop this alignment and move to the next based on the calculated shift.
    - If a full match is found, print the found message, set the found_index, and stop
     searching.
    - Print the total comparisons and shift counts.
    - If the pattern is not found after all alignments, print the 'Pattern not found'
    message.
    - Return the found_index.
    """
    pass

def test_pattern_matching():
    """
    Test function to demonstrate all implementations.
    """
    # Test Sample 1
    text1 = "aaabaadaabaaa"
    pattern1 = "aabaaa"
    print(f"--- Testing Sample 1 ---\nText: {text1}\nPattern: {pattern1}\n")
    idx_l2r = brute_force_match(text1, pattern1)
```

```
    idx_r2l = brute_force_match_r2l(text1, pattern1)
    idx_bm = boyer_moore_match(text1, pattern1)
    print(f"Summary: L2R found at {idx_l2r}, R2L at {idx_r2l}, BM at {idx_bm}\n")

    # Test Sample 2
    text2 = "THIS IS A TEST TEXT"
    pattern2 = "TEST"
    print(f"\n--- Testing Sample 2 ---\nText: {text2}\nPattern: {pattern2}\n")
    idx_l2r = brute_force_match(text2, pattern2)
    idx_r2l = brute_force_match_r2l(text2, pattern2)
    idx_bm = boyer_moore_match(text2, pattern2)
    print(f"Summary: L2R found at {idx_l2r}, R2L at {idx_r2l}, BM at {idx_bm}\n")

    # Test Sample 3
    text3 = "AABAACAADAABAABA"
    pattern3 = "XYZ"
    print(f"\n--- Testing Sample 3 ---\nText: {text3}\nPattern: {pattern3}\n")
    idx_l2r = brute_force_match(text3, pattern3)
    idx_r2l = brute_force_match_r2l(text3, pattern3)
    idx_bm = boyer_moore_match(text3, pattern3)
    print(f"Summary: L2R found at {idx_l2r}, R2L at {idx_r2l}, BM at {idx_bm}\n")

if __name__ == "__main__":
    test_pattern_matching()
```