**Report on Brute-Force and Sunday Pattern Matching Algorithms with Wildcard Support Using Python**

**Brute-Force Pattern Matching Algorithm with Wildcards**

**Introduction**

This report provides an overview of the implementation of a brute-force pattern matching algorithm with support for wildcards. The algorithm determines whether a given pattern matches a given text considering wildcards `'\*'` (matches zero or more characters) and `'?'` (matches any single character). The implementation also handles the escaping of wildcards and backslashes.

**Implementation Overview**

The implementation consists of two main functions: `preprocess(pattern)` and `is\_match(text, pattern)`.

**Preprocessing Function (`preprocess(pattern)`)**

**Description:**

This function preprocesses the pattern to handle the escaping of wildcards and backslashes. It iterates through the pattern character by character to ensure special characters are treated correctly if escaped.

**Implementation:**

- **Initialization:**

- `new\_pattern`: Stores the modified pattern after preprocessing.

- `escaped`: List to indicate whether each character in the pattern is escaped.

- `i`: Index for iterating through the pattern.

- **Processing:**

- Loop through each character in the pattern.

- If a backslash (`\`) is encountered:

- Check the next character:

- If it is another backslash, `\*`, or `?`, add the next character to `new\_pattern` and mark it as escaped.

- Otherwise, add the backslash to `new\_pattern` as a literal.

- Append other characters to `new\_pattern` and update the `escaped` list accordingly.

- **Return:**

- The function returns the preprocessed pattern and the list of escaped flags.

def preprocess(pattern):

new\_pattern = ""

escaped = []

i = 0

while i < len(pattern):

if pattern[i] == "\\":

if i + 1 < len(pattern) and pattern[i + 1] in ["\*", "?", "\\"]:

new\_pattern += pattern[i + 1]

escaped.append(True)

i += 2

else:

new\_pattern += "\\"

escaped.append(False)

i += 1

else:

new\_pattern += pattern[i]

escaped.append(False)

i += 1

return new\_pattern, escaped

**Brute-Force Pattern Matching Function (`is\_match(text, pattern)`)**

**Description:**

This function serves as the core of the pattern matching algorithm, implementing a brute-force approach with wildcard characters.

**Implementation:**

1. **Preprocessing:**

- The pattern is preprocessed to handle escaped characters.

2. **Initialization:**

- Variables initialized: `text\_index`, `pattern\_index`, `last\_wildcard\_index`, `text\_backtrack\_index`, and `next\_to\_wildcard\_index`.

3. **Matching Loop:**

- The function iterates over each character in the text string:

- If the current character in the pattern is `'?'` (and not escaped) or matches the text character, advance both indices.

- If `'\*'` is encountered, update indices to handle backtracking.

- If there is no match and no previous `'\*'`, return `False`.

- If there was a previous `'\*'`, backtrack to the next character after that `'\*'`.

4. **Post-loop Processing:**

- Skip any trailing `'\*'` characters in the pattern.

while pattern\_index < len(pattern) and pattern[pattern\_index] == '\*' and not escaped[pattern\_index]:

pattern\_index += 1

5. **Final Comparison:**

- Return `True` if the pattern index reaches the end of the pattern, indicating a successful match, otherwise `False`.

def is\_match(text, pattern):

pattern, escaped = preprocess(pattern)

text\_index = 0

pattern\_index = 0

last\_wildcard\_index = -1

text\_backtrack\_index = -1

next\_to\_wildcard\_index = -1

while text\_index < len(text):

if pattern\_index < len(pattern) and (pattern[pattern\_index] == '?' and not escaped[pattern\_index] or pattern[pattern\_index] == text[text\_index]):

text\_index += 1

pattern\_index += 1

elif pattern\_index < len(pattern) and pattern[pattern\_index] == '\*' and not escaped[pattern\_index]:

last\_wildcard\_index = pattern\_index

next\_to\_wildcard\_index = pattern\_index + 1

text\_backtrack\_index = text\_index

pattern\_index += 1

elif last\_wildcard\_index == -1:

return False

else:

pattern\_index = next\_to\_wildcard\_index

text\_backtrack\_index += 1

text\_index = text\_backtrack\_index

while pattern\_index < len(pattern) and pattern[pattern\_index] == '\*' and not escaped[pattern\_index]:

pattern\_index += 1

return pattern\_index == len(pattern)

**Testing**

The implementation includes a set of test cases to ensure correctness. The test cases cover various scenarios:

- Single character matches.

- Exact matches.

- Wildcard matches.

- Escaped characters.

- Combinations of wildcards and escaped characters.

- Empty patterns and texts.

Each test case asserts the expected output of the `is\_match` function for different input combinations. The test cases ensure that the algorithm behaves as expected across a wide range of scenarios.

**Conclusion**

The implementation of the brute-force pattern matching algorithm with wildcard support effectively handles wildcards, escaped characters, and various edge cases. The preprocessing step ensures proper handling of escaped characters, including the backslash escaping itself. Overall, the algorithm provides a reliable solution for pattern matching with wildcard support.

**Sunday Pattern Matching Algorithm with Wildcards using Python**

**Introduction**

This report provides an overview of the implementation of a Sunday pattern matching algorithm with support for wildcards. The algorithm determines whether a given pattern matches a given text considering wildcards `'\*'` (matches zero or more characters) and `'?'` (matches any single character). The implementation also handles the escaping of wildcards and backslashes.

**Implementation Overview**

The implementation consists of three main functions: `preprocess(pattern)`, `sunday\_algorithm\_with\_wildcard(text, pattern)`, and `find\_subpattern(text, subpattern, start, escaped)`.

**Preprocessing Function (`preprocess(pattern)`)**

**Description:**

This function preprocesses the pattern to handle the escaping of wildcards and backslashes. It iterates through the pattern character by character, identifying escaped characters. The function returns the preprocessed pattern and a list indicating which characters are escaped.

def preprocess(pattern):

new\_pattern = ""

escaped = []

i = 0

while i < len(pattern):

if pattern[i] == "\\":

if i + 1 < len(pattern) and pattern[i + 1] in ["\*", "?", "\\"]:

new\_pattern += pattern[i + 1]

escaped.append(True)

i += 2

else:

new\_pattern += "\\"

escaped.append(False)

i += 1

else:

new\_pattern += pattern[i]

escaped.append(False)

i += 1

return new\_pattern, escaped

**Split Pattern Function (`split\_pattern(pattern, escaped)`)**

**Description:**

This function splits the preprocessed pattern into parts based on the presence of the '\*' wildcard, treating each part as a subpattern. It also keeps track of whether characters are escaped.

def split\_pattern(pattern, escaped):

parts = []

current\_part = []

current\_escaped = []

for i in range(len(pattern)):

if pattern[i] == "\*" and not escaped[i]:

if current\_part:

parts.append(("".join(current\_part), current\_escaped))

current\_part = []

current\_escaped = []

else:

current\_part.append(pattern[i])

current\_escaped.append(escaped[i])

if current\_part:

parts.append(("".join(current\_part), current\_escaped))

return parts

**Skip Table Function (`build\_skip\_table(pattern)`)**

**Description:**

This function builds a skip table used in the Sunday algorithm. The skip table helps to determine the number of characters to skip in the text when a mismatch occurs.

def build\_skip\_table(pattern):

m = len(pattern)

skip\_table = {}

for i in range(m - 1, -1, -1):

if pattern[i] not in skip\_table:

skip\_table[pattern[i]] = m - i

return skip\_table

**Subpattern Matching Function (`find\_subpattern(text, subpattern, start, escaped)`)**

**Description:**

This function searches for a subpattern in the text starting from a given position. It uses the skip table to efficiently find the next possible match.

def find\_subpattern(text, subpattern, start, escaped):

m = len(subpattern)

n = len(text)

if m == 0:

return start

skip\_table = build\_skip\_table(subpattern)

i = start

while i <= n - m:

j = 0

while j < m and (text[i + j] == subpattern[j] or (subpattern[j] == '?' and not escaped[j])):

j += 1

if j == m:

return i + m

if i + m < n and text[i + m] in skip\_table:

i += skip\_table[text[i + m]]

else:

i += m + 1

return -1

**Sunday Algorithm with Wildcard Function (`sunday\_algorithm\_with\_wildcard(text, pattern)`)**

**Description:**

This function implements the Sunday pattern matching algorithm with support for wildcards. It preprocesses the pattern, splits it into subpatterns, and uses the `find\_subpattern` function to search for each subpattern in the text sequentially.

def sunday\_algorithm\_with\_wildcard(text, pattern):

if len(pattern) == 0 and len(text) == 0:

return True

elif len(pattern) == 0:

return False

pattern, escaped = preprocess(pattern)

parts = split\_pattern(pattern, escaped)

start = 0

for part, part\_escaped in parts:

if part == '':

continue

start = find\_subpattern(text, part, start, part\_escaped)

if start == -1:

return False

return True

**Testing**

The implementation includes a comprehensive set of test cases to validate the correctness of the algorithm. The test cases cover a wide range of scenarios including single character matches, exact matches, wildcard matches, escaped characters, and combinations of wildcards and escaped characters. The test cases ensure that the algorithm behaves correctly in various situations.

**Conclusion**

The implementation of the Sunday pattern matching algorithm with wildcard support efficiently handles wildcards, escaped characters, and various edge cases. The preprocessing step ensures proper handling of escaped characters, including the backslash escaping itself. The use of the skip table in the Sunday algorithm provides a more efficient search compared to a simple brute-force approach. Overall, the algorithm provides a reliable and efficient solution for pattern matching with wildcard support.

The provided implementations and their corresponding tests demonstrate the effectiveness of both brute-force and Sunday pattern matching algorithms with wildcard support. Each algorithm has been thoroughly tested to ensure correctness across a wide range of scenarios, making them suitable for various applications requiring pattern matching with wildcards.