Lesson 7: String Matching Algorithms

String matching is a fundamental concept in computer science and information retrieval that involves finding occurrences of one or more strings (patterns) within a larger text (source string). This process is used in a wide range of applications, including text search engines, data validation, DNA sequence analysis, compilers, and more. The primary goal of string matching is to identify the positions or indices where a particular pattern appears within the source string.

There are various methods and algorithms for performing string matching, each with its own characteristics, advantages, and disadvantages. Here are some of the key approaches to string matching:

Naive String Matching: The simplest method involves comparing the pattern against each substring of the source string in a brute-force manner. While easy to implement, this approach can be inefficient for large texts and patterns since it has a worst-case time complexity of O((n-m+1)m), where n is the length of the source string and m is the length of the pattern.

Knuth-Morris-Pratt (KMP) Algorithm: The KMP algorithm improves upon the naive approach by utilizing information about the pattern to avoid unnecessary comparisons. It precomputes a partial match table (also known as the failure function) based on the pattern, allowing the algorithm to skip ahead in the source string whenever a mismatch occurs. This results in a linear time complexity of O(n+m) in most cases.

Boyer-Moore Algorithm: This algorithm takes advantage of the mismatch information to skip more characters in the source string, leading to faster search times. It involves two main heuristics: the bad character rule and the good suffix rule. The Boyer-Moore algorithm typically performs well on large texts and patterns, and it can achieve a best-case time complexity of O(n/m) for certain patterns.

Rabin-Karp Algorithm: This algorithm employs a hash function to efficiently compare the pattern with potential substrings of the source string. By comparing hash values instead of characters, it can quickly eliminate many substrings that do not match the pattern. The Rabin-Karp algorithm has an average-case time complexity of O(n+m), although its performance can degrade if hash collisions occur frequently.

Aho-Corasick Algorithm: This algorithm is particularly useful for searching a large set of patterns within a single source string. It constructs a finite automaton (trie) that

efficiently matches multiple patterns in a single pass through the source string. The Aho-Corasick algorithm is often used in applications like keyword searching in text editors and network security.

Regular Expressions: Regular expressions are a powerful and flexible way to define patterns for string matching. They allow you to specify complex search patterns using a concise syntax. Many programming languages and text processing tools provide built-in support for regular expressions.

The choice of which string matching algorithm to use depends on factors such as the size of the source string and pattern, the frequency of matching, and the computational resources available. Different algorithms have different trade-offs in terms of time complexity, memory usage, and ease of implementation. Therefore, understanding the characteristics of each algorithm is crucial for selecting the most suitable approach for a given problem.

Brute force pattern matching

Brute force pattern matching is the simplest and most straightforward approach to finding occurrences of a pattern within a text. In this method, the algorithm compares the pattern to every possible substring of the text, one by one, to determine if there is a match. While simple to implement, it can be inefficient for large texts and patterns due to its quadratic time complexity.

Here's how the brute force pattern matching algorithm works:

1. Start with the first character of the pattern and the first character of the text.
2. Compare the characters of the pattern and the text one by one, moving the pattern forward if the characters match.
3. If a mismatch is encountered at any position, shift the pattern one position to the right and start the comparison again with the new pattern position.
4. Repeat steps 2 and 3 until either a full match is found or the end of the text is reached.

The worst-case time complexity of the brute force pattern matching algorithm is O((n - m + 1) * m), where:

- n is the length of the text.
- m is the length of the pattern.

The worst case occurs when the algorithm needs to compare the pattern against every possible substring of the text. This quadratic complexity makes the brute force approach inefficient for large texts and patterns.

Here's a basic implementation of the brute force pattern matching algorithm in Python:

```python
def brute_force_pattern_matching(text, pattern):
    n = len(text)
    m = len(pattern)

    for i in range(n - m + 1):
        j = 0
        while j < m and text[i + j] == pattern[j]:
            j += 1
        if j == m:
            print("Pattern found at index:", i)

# Example usage
text = "ababcababcabc"
pattern = "abc"
brute_force_pattern_matching(text, pattern)
```

Despite its inefficiency, the brute force algorithm serves as a baseline for understanding more advanced string matching algorithms. Algorithms like the Knuth-Morris-Pratt, Boyer-Moore, and Rabin-Karp improve upon the brute force approach by taking advantage of specific patterns in the data to reduce the number of unnecessary comparisons and improve search efficiency.

Knuth-Morris-Pratt (KMP) algorithm

The Knuth-Morris-Pratt (KMP) algorithm is a highly efficient string matching algorithm that improves upon the naive brute force approach by utilizing the information about the pattern to avoid unnecessary character comparisons. It achieves this by precomputing a partial match table (also known as the failure function) based on the pattern. This table

helps the algorithm determine how far it can skip ahead in the text when a mismatch occurs.

Here's how the Knuth-Morris-Pratt algorithm works:

1. Preprocessing: Build the partial match table (failure function) for the pattern. This table provides information about how much to shift the pattern when a mismatch occurs at a certain position.

2. String Matching: Start comparing the pattern with the text from left to right. If a mismatch occurs at position j of the pattern while comparing with position i of the text, use the information from the failure function to determine how far to shift the pattern. Instead of going back to the beginning of the pattern, the algorithm uses the value from the failure function to find the next potential starting position of the pattern.

3. Updating Failure Function: While building the failure function, for each position j in the pattern, calculate the length of the longest proper suffix (substring ending at j) that is also a proper prefix of the pattern. This value is stored in the failure function and is used to decide how far to shift the pattern in case of a mismatch.

The key advantage of the KMP algorithm is that it avoids redundant character comparisons. It skips ahead by using the failure function, which is precomputed based on the pattern's internal structure. This results in a linear time complexity of O(n + m), where:

- n is the length of the text.
- m is the length of the pattern.

Here's a basic implementation of the Knuth-Morris-Pratt algorithm in Python:

```
def build_failure_function(pattern):
    m = len(pattern)
    failure = [0] * m
    j = 0
    for i in range(1, m):
        while j > 0 and pattern[i] != pattern[j]:
            j = failure[j - 1]
        if pattern[i] == pattern[j]:
            j += 1
```

```
            failure[i] = j
    return failure

def kmp_search(text, pattern):
    n = len(text)
    m = len(pattern)
    failure = build_failure_function(pattern)
    j = 0
    for i in range(n):
        while j > 0 and text[i] != pattern[j]:
            j = failure[j - 1]
        if text[i] == pattern[j]:
            j += 1
        if j == m:
            print("Pattern found at index:", i - m + 1)
            j = failure[j - 1]

# Example usage
text = "ABABDABACDABABCABAB"
pattern = "ABABCABAB"
kmp_search(text, pattern)
```

The Knuth-Morris-Pratt algorithm is particularly useful when searching for a relatively small pattern within a large text, as it significantly reduces the number of character comparisons needed compared to the brute force approach.

Boyer-Moore algorithm

The Boyer-Moore algorithm is a powerful and efficient string searching algorithm that uses two main heuristics to skip ahead in the text and reduce the number of character comparisons required. These heuristics are the "bad character rule" and the "good suffix rule." The algorithm preprocesses the pattern to build tables that provide information on how to shift the pattern when a mismatch occurs.

Here's how the Boyer-Moore algorithm works:

Preprocessing:

- Build the "bad character" table: For each character in the pattern, calculate the distance between the rightmost occurrence of that character and the current position. If the character does not appear in the pattern, use the pattern length as the default distance.
- Build the "good suffix" table: For each position in the pattern, calculate the length of the longest suffix that matches a suffix of the pattern. This information helps to determine how far the pattern can be shifted to align with a matching suffix.

String Matching:

- Start comparing the pattern with the text from right to left. Compare characters starting from the end of the pattern and moving towards the beginning.
- When a mismatch occurs at position j of the pattern while comparing with position i of the text, use the bad character rule to calculate the shift amount based on the rightmost occurrence of the mismatched character in the pattern and the good suffix rule to determine additional shift amounts.

Updating Shift:

- Calculate the shift as the maximum of the shifts suggested by the bad character rule and the good suffix rule.
- Move the pattern so that its end lines up with the mismatched character in the text and repeat the comparison.

The Boyer-Moore algorithm is particularly efficient when there are many mismatches, as it can skip larger portions of the text with its shift rules. It achieves an average-case time complexity of O(n/m) for searching a pattern of length m in a text of length n.

Here's a basic implementation of the Boyer-Moore algorithm in Python:

```
def build_bad_character_table(pattern):
    m = len(pattern)
    bad_char = {}
    for i in range(m):
        bad_char[pattern[i]] = i
    return bad_char

def build_good_suffix_table(pattern):
    m = len(pattern)
    suffix = [0] * (m + 1)
```

```
    border = 0
    for i in range(m, 0, -1):
        while border > 0 and pattern[i - 1] != pattern[m - border -
1]:
            border = suffix[border]
        if pattern[i - 1] == pattern[m - border - 1]:
            border += 1
        suffix[i] = border
    return suffix

def boyer_moore_search(text, pattern):
    n = len(text)
    m = len(pattern)
    bad_char = build_bad_character_table(pattern)
    good_suffix = build_good_suffix_table(pattern)
    j = 0
    while j <= n - m:
        i = m - 1
        while i >= 0 and pattern[i] == text[j + i]:
            i -= 1
        if i < 0:
            print("Pattern found at index:", j)
            if j + m < n:
                shift = m - good_suffix[1]
            else:
                shift = 1
        else:
            bad_char_shift = max(1, i - bad_char.get(text[j + i],
-1))
            good_suffix_shift = m - good_suffix[i + 1]
            shift = max(bad_char_shift, good_suffix_shift)
        j += shift

# Example usage
text = "ABABDABACDABABCABAB"
pattern = "ABABCABAB"
```

```
boyer_moore_search(text, pattern)
```

The Boyer-Moore algorithm's ability to skip ahead efficiently makes it especially effective for searching large texts or patterns with many mismatches.

Rabin-Karp algorithm for substring search

The Rabin-Karp algorithm is a widely used string searching algorithm that efficiently finds occurrences of a shorter "pattern" string within a longer "text" string. It's based on using a rolling hash function to quickly compare the hash values of the pattern and overlapping substrings of the text.

Here's a basic overview of how the Rabin-Karp algorithm works:

Hashing Function: Choose a hash function that converts a string of characters into a numerical value. This function should have the property that if two strings are equal, their hash values will also be equal. A common choice is polynomial rolling hash function.

Calculate Hashes: Compute the hash value of the pattern and the initial hash value of the first window (substring) of the text that has the same length as the pattern.

Comparison and Rolling: Compare the hash value of the pattern with the hash value of the current window in the text. If they match, do a character-by-character comparison to confirm if it's a spurious hit or an actual match. If they don't match, move the window one character ahead in the text and update the hash value using the rolling hash technique.

Repeat: Keep sliding the window and recalculating the hash value using the rolling hash technique until you've checked all possible windows in the text.

Collision Handling: In the case of a hash collision (different substrings have the same hash value), perform a full character-by-character comparison to confirm if it's a true match.

The rolling hash technique involves updating the hash value for the next window by subtracting the contribution of the first character in the current window and adding the

contribution of the next character in the text. This allows you to avoid recomputing the entire hash value from scratch for each window.

The Rabin-Karp algorithm is efficient when the hash function is well-designed and the pattern's hash value can be compared to the hash value of the text substrings in constant time. On average, the algorithm has a time complexity of O(n + m), where n is the length of the text and m is the length of the pattern. However, in worst cases, it can take O(n * m) time if hash collisions are frequent.

Keep in mind that while the Rabin-Karp algorithm has the advantage of handling multiple pattern searches with a single preprocessing of the text, other algorithms like the Knuth-Morris-Pratt (KMP) algorithm or Boyer-Moore algorithm can provide even better performance in many practical scenarios.

Applications in text processing and search engines

The string matching algorithms like Knuth-Morris-Pratt (KMP) and Boyer-Moore, along with other techniques, play a crucial role in various applications within text processing and search engines. Here are some of the key applications where these algorithms are utilized:

1. Text Search Engines: Search engines like Google, Bing, and others rely heavily on efficient string matching algorithms to quickly retrieve relevant results from vast amounts of textual data. These algorithms allow users to input search queries and find matching documents, web pages, or other content quickly.
2. Keyword Searching: In databases, file systems, and text editors, string matching algorithms help perform keyword searches. Users can search for specific words or phrases within documents, emails, source code files, and more.
3. Spelling Correction and Autocompletion: String matching algorithms are used in spell checkers and autocomplete features. They help suggest correct spellings or complete words based on partial input, enhancing the user experience while typing.
4. Data Validation: String matching algorithms can be used to validate user inputs, such as email addresses, phone numbers, and credit card numbers, against predefined patterns. This ensures that the entered data follows the expected format.

5. Programming Language Compilers: Compilers use string matching algorithms to efficiently process source code files. For instance, they might search for variable or function names to resolve references or identify errors.
6. DNA Sequence Matching: In bioinformatics, DNA sequence matching is critical for analyzing genetic data. String matching algorithms are used to find similarities in DNA sequences, aiding in gene identification, evolutionary studies, and medical research.
7. Network Intrusion Detection: Intrusion detection systems use string matching algorithms to identify patterns or signatures of malicious activities in network traffic, helping to detect potential threats or attacks.
8. Text Extraction and Information Retrieval: String matching algorithms can be employed to extract specific information from unstructured text, such as extracting dates, addresses, or product names from documents.
9. Natural Language Processing (NLP): NLP tasks, like sentiment analysis or named entity recognition, may involve string matching to identify specific words or phrases of interest within text.
10. Regular Expressions: Regular expressions, which are used to define complex search patterns, are implemented using string matching algorithms. They find applications in data extraction, text processing, and search operations.
11. Web Crawlers: String matching is used in web crawling to identify URLs or keywords that indicate links to other web pages, helping to build comprehensive indexes of the web.
12. Data Mining and Text Analytics: String matching algorithms can assist in data mining tasks, such as identifying patterns in customer reviews, social media posts, or news articles.

These applications highlight the importance of efficient string matching algorithms in various domains, enabling the processing, analysis, and retrieval of textual information in a wide range of contexts.