Lecture Notes
Data Structures and Algorithms

String Algorithms

Overview

1. Introduction to string pattern matching
    - Definition of the problem
    - Applications of string pattern matching
    - Examples of string pattern matching algorithms
2. Brute force algorithm
    - Idea of the brute force algorithm
    - Time complexity analysis
    - Implementation of the brute force algorithm in pseudocode
3. Knuth-Morris-Pratt (KMP) algorithm
    - Idea of the KMP algorithm
    - Time complexity analysis
    - Implementation of the KMP algorithm in pseudocode
4. Boyer-Moore algorithm
    - Idea of the Boyer-Moore algorithm
    - Time complexity analysis
    - Implementation of the Boyer-Moore algorithm in pseudocode
5. Comparison of string pattern matching algorithms
    - Comparison of time complexities
    - Trade-offs between the different algorithms
    - When to use each algorithm
6. Conclusion
    - Summary of the different string pattern matching algorithms
    - Future directions in string pattern matching research

Introduction

- Definition of the problem: String pattern matching is the problem of finding the occurrences of a pattern string (also called the needle) within a larger text string (also called the haystack). The goal is to find all the starting indices of the pattern within the text.

  I.e. given a text string **T** of length **n** and a pattern string **P** of length **m**, find all the occurrences of **P** within **T**. More formally, we want to find all the indices **i** such that **P** is a substring of **T** starting at position **i**.

- In Java, we could represent the input strings T and P as **String** objects and the output as a list of integers **List<Integer>**. Here is a possible Java function signature for a string pattern matching function: `List<Integer> findPattern(String T, String P)`

- Applications of string pattern matching: String pattern matching has a wide range of applications, including text editing, text search engines, and bioinformatics.
- Examples of string pattern matching algorithms: Some common string pattern matching algorithms include the brute force algorithm, the Knuth-Morris-Pratt (KMP) algorithm, the Boyer-Moore algorithm, and the Rabin-Karp algorithm (which we will not discuss).

Brute Force

- The brute force algorithm is a simple but inefficient string pattern-matching algorithm. It works by checking all possible substrings of the text **T** to see if they match the pattern **P**.
- The time complexity of the brute force algorithm is ***O(n * m),*** where ***n*** is the length of the text and ***m*** is the length of the pattern. This is because the algorithm needs to check each character of the text against the characters of the pattern, and there are ***n - m + 1*** substrings of the text that need to be checked.
- Implementation of the brute force algorithm in pseudocode: Here is pseudocode for the brute force algorithm:

```
import java.util.ArrayList;
import java.util.List;

public class BruteForce {
  public static List<Integer> findPattern(String T, String P) {
    int n = T.length();
    int m = P.length();
    List<Integer> result = new ArrayList<>();
    for (int i = 0; i <= n - m; i++) {
      boolean found = true;
      for (int j = 0; j < m; j++) {
        if (T.charAt(i + j) != P.charAt(j)) {
          found = false;
          break;
        }
      }
      if (found) {
        result.add(i);
      }
    }
    return result;
  }
}
```

The Knuth-Morris-Pratt (KMP) Algorithm

- The KMP algorithm is a string pattern matching algorithm that uses information from the pattern itself to avoid checking substrings that have already been checked. It does this by preprocessing the pattern to create a table that stores the maximum number of characters that can be skipped when a mismatch occurs.

- This is achieved by precomputing a prefix table that tells the algorithm how much of the pattern can be skipped when a mismatch occurs.
  To compute the prefix table, we look for the longest proper prefix of the pattern that is also a suffix of the pattern for each index in the pattern. This prefix table tells us how much of the pattern we can skip over when we encounter a mismatch, because we know that the skipped portion already matches a portion of the text.

  For example, consider the pattern "ababc". The prefix table for this pattern would be:

  ```
  Index:      0 1 2 3 4
  Pattern:    a b a b c
  Prefix:     0 0 1 2 0
  ```

- The first two values of the prefix table **are always 0**, because a prefix or suffix of length 1 is always the same character, and so cannot be a proper prefix.

  To compute the remaining values of the prefix table, we start by comparing the characters at the current index and the index indicated by the previous value of the prefix table. If they match, we increment the prefix value for the current index by 1. If they don't match, we move to the index indicated by the previous value of the prefix table and try again, until we either find a matching character or reach the beginning of the pattern.

```java
private static int[] computePrefix(char[] pattern) {
    int m = pattern.length;
    int[] prefix = new int[m];
    prefix[0] = 0;

    int j = 0;
    int i = 1;
    while (i < m) {
        if (pattern[i] == pattern[j]) {
            j++;
            prefix[i] = j;
            i++;
        } else if (j > 0) {
```

```
            j = prefix[j - 1];
        } else {
            prefix[i] = 0;
            i++;
        }
    }
    return prefix;
}
```

- Once we have computed the prefix table, we can use it to search for the pattern in the text. We start by initializing variables **i** and **j** to 0, representing the current indices in the text and pattern, respectively. We then loop through the text until we reach the end or find a match. If the characters at the current indices match, we increment both **i** and **j** and continue. If they don't match, we look up the value of the prefix table for the current index in the pattern, and use it to update **j**. Specifically, we set **j** to the value in the prefix table for the current index, and then try again with the same **i** and the new **j**.
- Here is java code for the KMP algorithm:

```
public static List<Integer> kmp(String text, String pattern) {
    List<Integer> matches = new ArrayList<>();
    int n = text.length();
    int m = pattern.length();
    // Step 1: Compute prefix table for pattern
    int[] prefix = computePrefix(pattern.toCharArray());
    // Step 2: Initialize variables
    int i = 0;
    int j = 0;
    // Step 3: Search for pattern in text
    while (i < n) {
        if (text.charAt(i) == pattern.charAt(j)) {
            if (j == m - 1) {  // Pattern found
                matches.add(i - j);
                j = prefix[j];  // Continue search
            } else {  // Continue matching
```

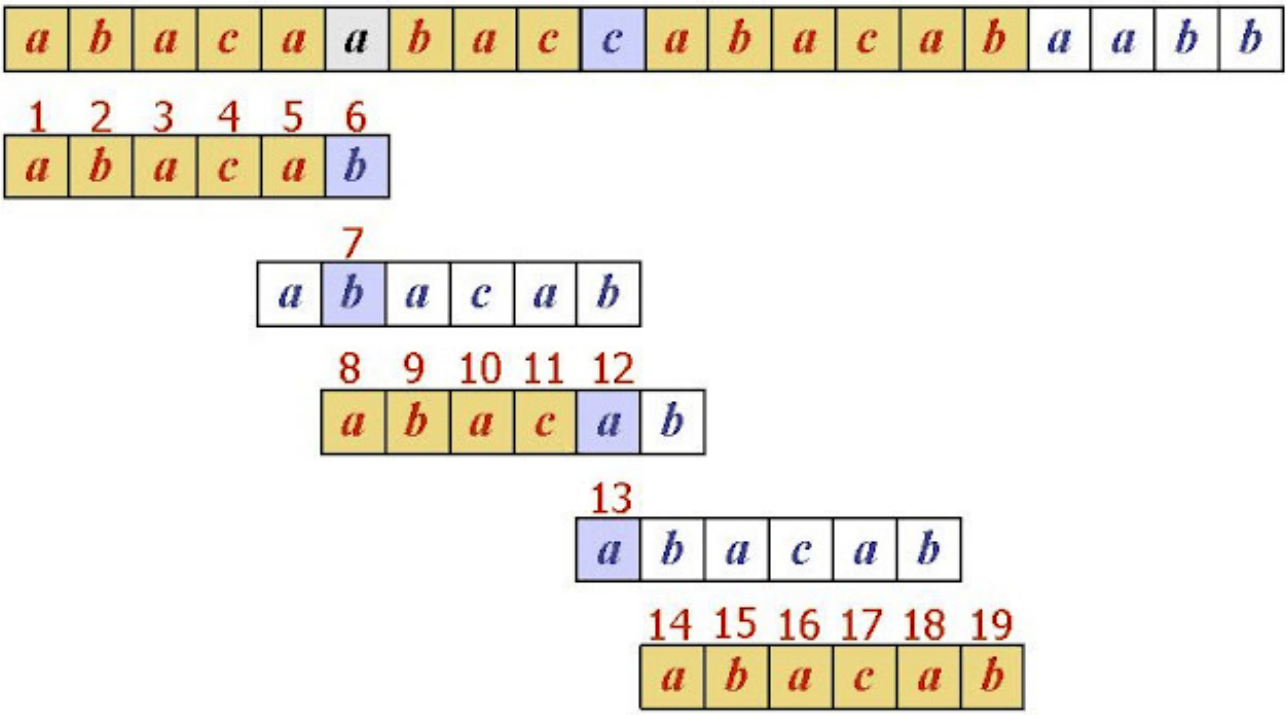
```
                i++;
                j++;
            }
        } else if (j > 0) {  // Mismatch
            j = prefix[j - 1];  // Skip over matched portion
        } else {  // No match found
            i++;
        }
    }

    return matches;
}
```

- Time complexity analysis: The time complexity of the KMP algorithm is ***O(n + m)***, where ***n*** is the length of the text and ***m*** is the length of the pattern. This is because the algorithm only needs to scan the text and the pattern once, and the time to create the table is ***O(m)***.
  The time complexity of the KMP algorithm is O(n + m), where n is the length of the text and m is the length of the pattern. This is because the algorithm only needs to scan the text and the pattern once, and the time to create the table is O(m).

  In contrast, the time complexity of the brute force algorithm is ***O(n * m)***, where n is the length of the text and m is the length of the pattern. This is because the brute force algorithm needs to check each character of the text against the characters of the pattern, and there are ***n - m + 1*** substrings of the text that need to be checked.

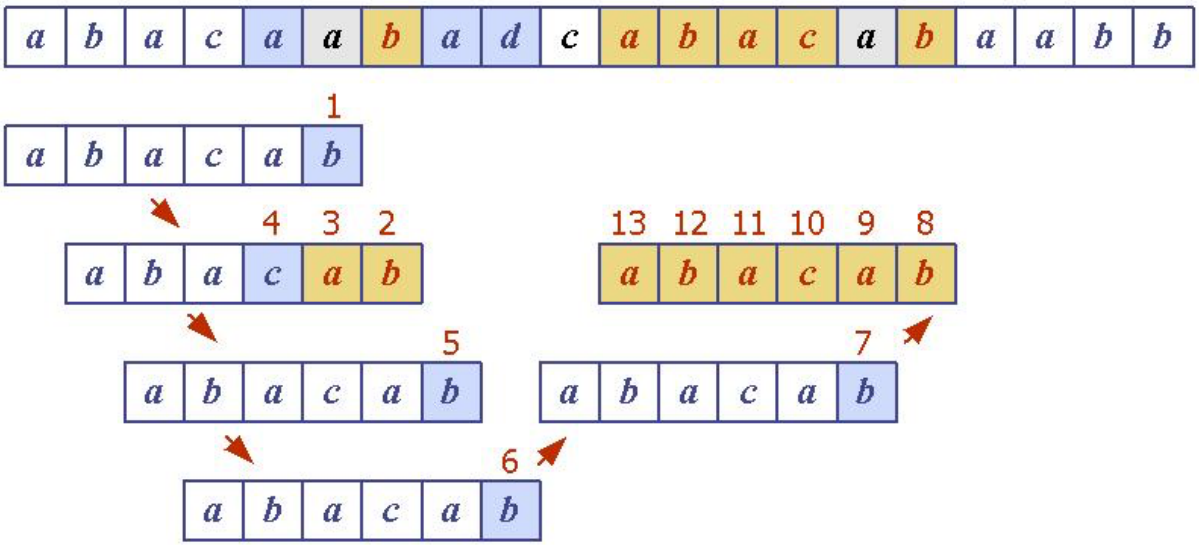
KMP Example



x	0	1	2	3	4	5
P[x]	*a*	*b*	*a*	*c*	*a*	*b*
f(x)	0	0	1	0	1	2

The Boyer-More Algorithm

- The Boyer-Moore algorithm is a string pattern matching algorithm that uses information from the pattern itself to avoid checking substrings that are unlikely to match the pattern. It does this by preprocessing the pattern to create a "skip table".



- The “last occurrence” table stores the last occurrence of each character in the pattern:

	pattern: foxtrot				
Character	f	o	x	t	r
Last Index	0	5	2	6	4

The pattern-matching algorithm algorithm can then use this table to determine how many characters it can skip when a mismatch occurs.

```
public static int[] lastOccurrence(char[] pattern) {
    int[] last = new int[Character.MAX_VALUE + 1];
    Arrays.fill(last, -1);
    for (int i = 0; i < pattern.length; i++) {
        last[pattern[i]] = i;
    }
    return last;
}
```

This function takes in a character array **pattern** representing the pattern we want to search for, and returns an integer array **last** containing the last occurrence of each character in the pattern.

The function first initializes the **last** array with **-1** for all character values using **Arrays.fill()**. Then,

it loops through the characters in the pattern and sets the corresponding index in the **last** array to the current index of the character. This means that if the same character appears multiple times in the pattern, the index in the **last** array will be updated to the last occurrence.

- During the matching process, the algorithm starts at the end of the pattern and compares it to the characters in the text. If there is a mismatch, it uses the last occurrence table to determine how many characters to skip in the text. If there is no mismatch, it moves to the next character in the text.
- This process is repeated until a match is found or the end of the text is reached.

```java
public static List<Integer> boyerMoore(String text, String pattern) {
    List<Integer> matches = new ArrayList<>();
    int n = text.length(); int m = pattern.length();
    // Step 1: Precompute last occurrence of each character in pattern
    int[] last = lastOccurrence(pattern.toCharArray());
    // Step 2: Initialize variables
    int i = m - 1; int j = m - 1;
    // Step 3: Search for pattern in text
    while (i < n) {
        if (text.charAt(i) == pattern.charAt(j)) {
            if (j == 0) {  // Pattern found
                matches.add(i);
                i += m;  // Skip over pattern
                j = m - 1;
            } else {  // Continue matching
                i--;
                j--;
            }
        } else {  // Mismatch
            int lastOccur = last[text.charAt(i)];
            i += m - Math.min(j, 1 + lastOccur);
            j = m - 1;
        }
    }
    return matches;
}
```

This function takes in a **text** string and a **pattern** string, and returns a list of integer indices where the pattern is found in the text.

The function first computes the **last** array using the **lastOccurrence** function from my previous answer. It then initializes variables **i** and **j** to the end of the pattern.

The function then searches for the pattern in the text by looping through the text from the end of the pattern to the end of the text. If a character in the text matches the corresponding character in the pattern, the function continues to match subsequent characters until the pattern is found or a mismatch is detected. If a mismatch is detected, the function uses the **last** array to skip over as much of the text as possible before continuing the search.

When the pattern is found, the function adds the index of the pattern in the text to the **matches** list, and then skips over the pattern in the text to continue the search.

Finally, the function returns the list of indices where the pattern was found in the text.

The best way to do a step by step visualization of the algorithm is using animation. The algorithm makes many decisions at each iteration, making it difficult to go through it step by step in writing. Here are some resources I recommend you try to get a feel of how the algorithm works:

- https://dwnusbaum.github.io/boyer-moore-demo/
- https://cmps-people.ok.ubc.ca/ylucet/DS/BoyerMoore.html
- http://whocouldthat.be/visualizing-string-matching/

Comparison of Algorithms

- Brute force algorithm: The brute force algorithm is the simplest string pattern matching algorithm, but it has a time complexity of O(n * m), where n is the length of the text and m is the length of the pattern. This means that it is inefficient for long texts or patterns.
- Knuth-Morris-Pratt (KMP) algorithm: The KMP 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. It is more efficient than the brute force algorithm, especially for long patterns.
- Boyer-Moore algorithm: The Boyer-Moore 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. It is generally considered to be the most efficient string pattern matching algorithm, as it can skip over large substrings quickly using the "bad character" and "good suffix" tables.

Other String Pattern Matching Algorithms

- There are many other string pattern-matching algorithms in addition to the ones discussed in this lecture. Some examples include:
    - Rabin-Karp algorithm: The Rabin-Karp algorithm is a string pattern-matching algorithm that uses hashing to quickly search for patterns in a text. It 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, it is not as efficient as the KMP or Boyer-Moore algorithms in practice due to the overhead of hashing.
    - Aho-Corasick algorithm: The Aho-Corasick algorithm is a string pattern-matching algorithm that can find multiple patterns in a text simultaneously. It has a time complexity of ***O(n + m + z)***, where n is the length of the text, m is the length of the pattern, and z is the number of occurrences of the pattern in the text. It is efficient for finding multiple patterns in a text, but may not be as efficient as the KMP or Boyer-Moore algorithms for finding a single pattern.
    - Suffix array: A suffix array is a data structure that can be used for efficient string pattern matching. It has a time complexity of ***O(m * log(n))***, where n is the length of the text and m is the length of the pattern. It is efficient for finding patterns in long texts, but may not be as efficient as the KMP or Boyer-Moore algorithms for shorter texts.