CHAPTER 10
STRING MATCHING

10.1 Learning Objectives

1. Students are able to understand the concept of string matching.
2. Students are able to understand the naïve algorithm for string matching.
3. Students are able to understand the KMP algorithm for string matching.
4. Students are able to solve problems related to string matching.

10.2 Material

10.2.1 String Matching

Let's start with the following problem: Given a text, find all occurrences of a given pattern in it. For example:

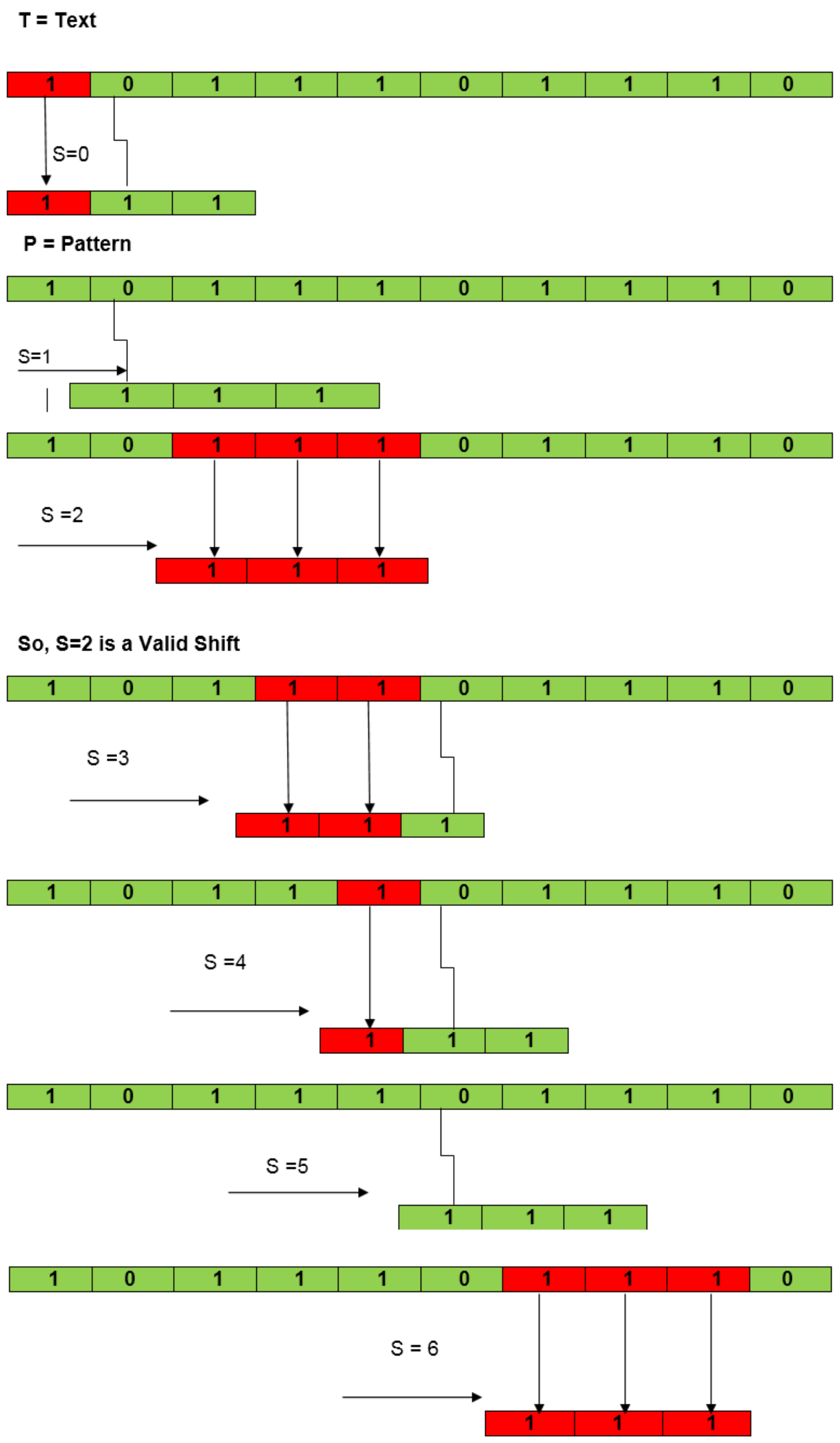
1. For text T = "ABCABAABCABAC" and pattern P = "ABAA", the pattern occurs only once in the text, starting from index 3 ("ABC**ABAA**BCABAC").
2. For text T = "ABCABAABCABAC" and pattern P = "CAB", the pattern occurs twice in the text, starting from index 2 and 8 ("AB**CAB**AABCABAC" and "ABCABAAB**CAB**AC").

The string matching is a widespread real-life problem that frequently arises in text-editing programs such as MS Word, notepad, notepad++, etc. String matching algorithms are also used to search for particular patterns in DNA sequences. The goal is to find all occurrences of pattern **P[0…(m-1)]** of length **m** in the given text **T[0…(n-1)]** of length **n**. There are several algorithms in string matching, such as naïve algorithm, Knuth-Morris-Pratt (KMP) algorithm, Z algorithm, Rabin-Karp algorithm, etc. In this chapter, we will only discuss about the naïve algorithm and the KMP algorithm.

10.2.2 The Naïve Algorithm

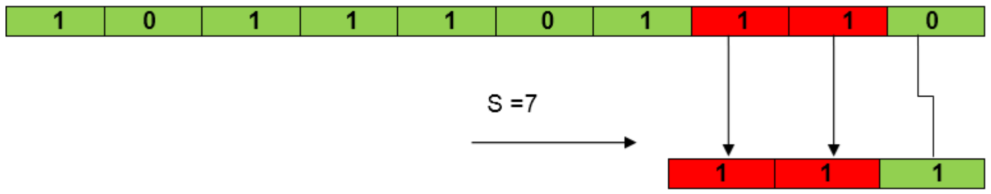
The naive algorithm finds all occurrences of a pattern by using a loop that checks the condition **P[0…(m-1)] = T[s…(s+m-1)]** for each of the **n-m+1** possible values of s.

For example, we have T = "1011101110" and P = "111". At first, s has a value of 0, and therefore we try to check whether P[0...2] = T[0..2]. Notice that P[0] = T[0], but P[1] and T[1] are different, so update s into 1. After checking for s=1, continue with s=2, and so on until s=n-m=7. Below is the illustration from the given example.



So, S=2 is a Valid Shift

So, S=6 is a Valid Shift



10.2.3 The KMP Algorithm

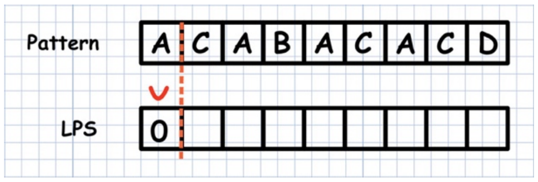
The naïve algorithm doesn't remember any information about the past matched characters. It basically matches a character with a different pattern character over and over again. It can be optimized by utilizing past matched characters.

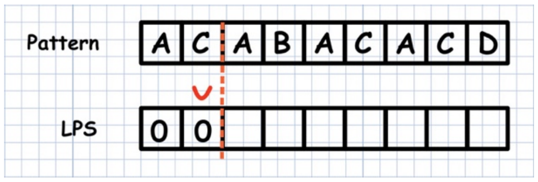
The KMP Algorithm (or Knuth, Morris, and Pratt string searching algorithm) cleverly uses the previous comparison data. It can search for a pattern faster that the naïve algorithm as it never re-compares a text symbol that has matched a pattern symbol. However, it uses a comparison table to analyze the pattern structure.

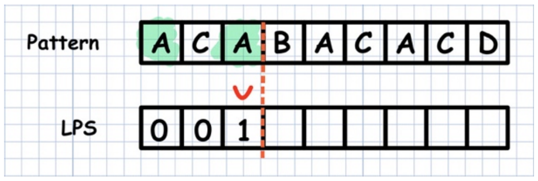
Before continuing to how to create a comparison table, we need to know about **proper prefixes** and **proper suffixes**:

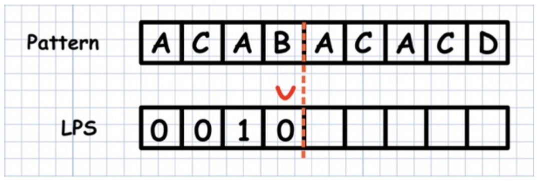
- **Proper prefix**: A string generated by removing one or more characters from the end. "S", "Sn", "Sna", and "Snap" are all the proper prefixes of "Snape".
- **Proper suffix**: A string generated by removing one or more characters from the beginning. "agrid", "grid", "rid", "id", and "d" are all proper suffixes of "Hagrid".

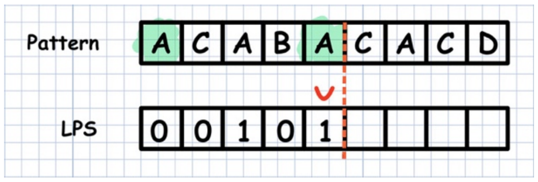
The comparison table, or widely known as LPS table, is an array that stores **the length of the longest proper prefix in the (sub)pattern that matches a proper suffix in the same (sub)pattern** for each index i, from i=0 to m-1. For example, given a pattern "ACABACACD", here is the manual step by step construction of the LPS array for the pattern. Red arrow indicates the current element of LPS. Dash bar shows the sub-string of a pattern considered for the current LPS index. Equal proper prefixes and proper suffixes are highlighted with green.

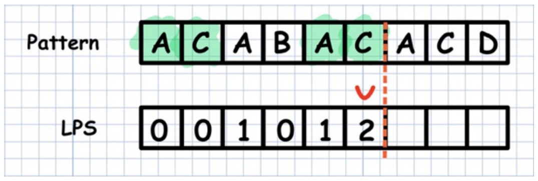

Pattern
A C A B A C A C D
LPS
0

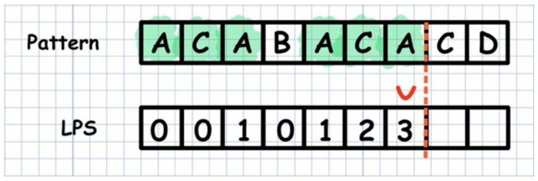

Pattern
A C A B A C A C D
LPS
0 0

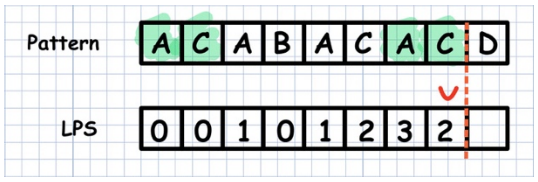

Pattern
A C A B A C A C D
LPS
0 0 1

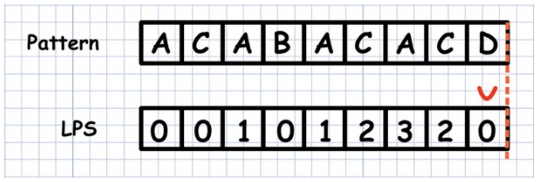

Pattern
A C A B A C A C D
LPS
0 0 1 0


Pattern
A C A B A C A C D
LPS
0 0 1 0 1


Pattern
A C A B A C A C D
LPS
0 0 1 0 1 2


Pattern
A C A B A C A C D
LPS
0 0 1 0 1 2 3


Pattern
A C A B A C A C D
LPS
0 0 1 0 1 2 3 2


Pattern
A C A B A C A C D
LPS
0 0 1 0 1 2 3 2 0

These are other examples:

- For the pattern “AAAA”, LPS[] is [0, 1, 2, 3]
- For the pattern “ABCDE”, LPS[] is [0, 0, 0, 0, 0]
- For the pattern “AABAACAABAA”, LPS[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]
- For the pattern “AAACAAAAAC”, LPS[] is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]
- For the pattern “AAABAAA”, LPS[] is [0, 1, 2, 0, 1, 2, 3]

This is the algorithm to construct the LPS table:

1. Initialize LPS[0] to 0 and assume that our pattern is stored in a variable named P.
2. Iterate from i=1 to i = P.length - 1, basically the iteration starts from the second character of P until the last character of P.
3. To calculate the current value of LPS[i], set the variable j denoting the length of the best suffix for S[0..(i-1)]. Initially j = LPS[i-1].
4. Test if the suffix of length j+1 is also prefix, by comparing P[j] and P[i].
    a. If P[i] = P[j], then assign LPS[i] = j+1, because we find a prefix of length j+1 that is also a suffix of P[0..i].
    b. Otherwise, update j to LPS[j-1] and repeat the step 4a.
    c. If we have reached j=0 after repeating the step 4b and still don't have a match, then assign LPS[i]=0.

Now we are moving on to the KMP algorithm. Suppose that we are given a text T and a pattern P. The following are the steps of the KMP algorithm:

1. Create a new string S that is formed by combining P and T separated by a single '#' character. For example, if T = “ABCABAABCABAC” and P = “ABAA”, then S = “ABAA#ABCABAABCABAC”. We may assume that the character '#' appear neither in T nor in P.
2. Create an LPS table of S.
3. Iterate from i=P.length+1 to i=S.length-1.
4. If we have LPS[i] = P.length for some i, it means that we found a suffix of S[0..i] of length P.length that is also prefix. Hence, we find the occurrence of P that starts from index i - (P.length + 1) - P.length + 1 = i - 2*(P.length).

10.2.4 Java Implementation

We will create the **StringMatcher** class, and it contains two public static methods, naive and kmp, each represents the algorithm covered in this chapter. The class also contains one private static method, computeLPSArray, as helper method in the KMP algorithm.

```java
public class StringMatcher {
    public static void naive(String text, String pattern){
        int textLen = text.length();
        int patternLen = pattern.length();

        boolean found = false;

        for (int i=0 ; i+patternLen <= textLen ; i++){
            boolean current_found = true;
            for (int j=0 ; j<patternLen ; j++){
                if (text.charAt(i+j) != pattern.charAt(j)){
                    current_found = false;
                    break;
                }
            }
            if (current_found){
                found = true;
                System.out.println("Found pattern at index " + i + " using naive.");
            }
        }
        if (!found){
            System.out.println("Pattern not found using naive.");
        }
    }
```

```java
    private static int[] computeLPSArray(String str){
        int len = str.length();

        int[] lps = new int[len];
        lps[0] = 0;

        for (int i=1 ; i<len ; i++){
            int j = lps[i-1];

            while ((j > 0) && (str.charAt(i) != str.charAt(j))){
                j = lps[j-1];
            }
            if (str.charAt(i) == str.charAt(j)){
                j++;
            }
            lps[i] = j;
        }

        return lps;
    }
```

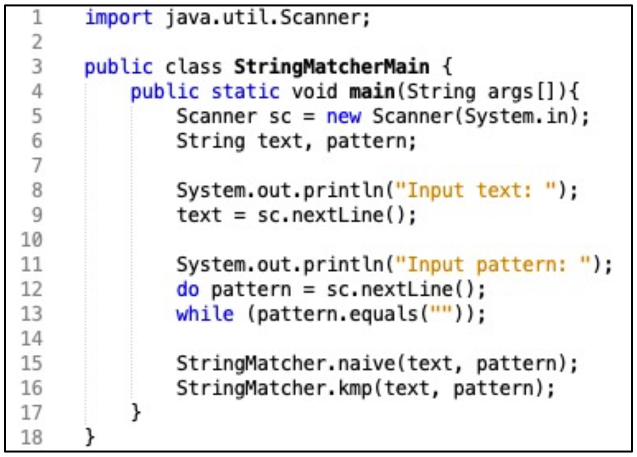
```java
    public static void kmp(String text, String pattern){
        String combined = pattern + "#" + text;
        int combinedLen = combined.length();
        int patternLen = pattern.length();

        int lps[] = computeLPSArray(combined);

        boolean found = false;
        for (int i=patternLen+1 ; i<combinedLen ; i++){
            if (lps[i] == patternLen){
                found = true;
                System.out.println("Found pattern at index " + (i - 2*patternLen) + " using KMP.");
            }
        }

        if (!found){
            System.out.println("Pattern not found using KMP.");
        }
    }
}
```

Finally, we create the **StringMatcherMain** class that contains main method for the string-matching program.



```
import java.util.Scanner;

public class StringMatcherMain {
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        String text, pattern;

        System.out.println("Input text: ");
        text = sc.nextLine();

        System.out.println("Input pattern: ");
        do pattern = sc.nextLine();
        while (pattern.equals(""));

        StringMatcher.naive(text, pattern);
        StringMatcher.kmp(text, pattern);
    }
}
```

10.3 Assignments

1. Given a text = "aaaa...aa" (100.000 times of letter a) and a pattern = "aaa...aab" (10.000 times of letter a and 1 letter b). Compare the running time of naïve algorithm and KMP algorithm to perform string matching! Perform the string matching 10 times, and then compute the average running time of each algorithm.
2. Given a text = "aaaa...aa" (100.000 times of letter a) and a pattern = "aaa...aa" (10.000 times of letter a). Compare the running time of naïve algorithm and KMP algorithm (**without any output**) to perform string matching! Perform the string matching 10 times, and then compute the average running time of each algorithm.
3. Given two words S and T, both has equal length (the maximum length is 100.000). Your task is to determine whether T can be created by performing multiple cycle shifts to S (a cycle shift is a transfer of the first character of the string to the end of this string). For example, if S = "erwineko" and T = "ekoerwin", the answer must be "YES", because "erwineko" -> "rwinekoe" -> "winekoer" -> "inekoerw" -> "nekoerwi" -> "ekoerwin". In this problem, you have to use the KMP algorithm to solve this task (the approach may not be obvious, but you have to figure the usage of the KMP algorithm in this problem).