# String matching: KMP Approach

**Session No.: 12**
**Course Name: Design and analysis of algorithm**
**Course Code: R1UC407B**
**Instructor Name: Mili Dhar**
**Duration: 50 Min.**
**Date of Conduction of Class:**

## Review of the key concepts

1. Review of previous session

Q: What was the reason that generated worst-case complexity by the naïve method or by Rabin Karp approach of string matching?

## Learning Outcome

Apply the algorithm to solve string matching problems

Analyse the time complexity of the algorithm.

## Session Outline

1 Introduction to KMP approach

2 Step by Step demonstration of KMP approach

3 Apply KMP approach to find a pattern in a given text

4 Analyse its complexity

# Drawback of O(mn) approach

String T

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | a | b | a | a | b | c | a | b | a | c |

String P

| | | | |
|---|---|---|---|
| a | b | a | a |

- If ‘m’ is the length of pattern ‘P’ and ‘n’ the length of string ‘T’, the matching time is of the order O(mn). This is certainly a very slow-running algorithm.
- What makes this approach so slow is the fact that elements of ‘T’ with which comparisons had been performed earlier are involved again and again in comparisons in some future iterations. For example, when a mismatch is detected for the first time in comparison of P[3] with T[3], pattern ‘P’ would be moved one position to the right, and the matching procedure would resume from here. Here, the first comparison that would take place would be between P[0]=‘a’ and T[1]=‘b’. It should be noted here that T[1]=‘b’ had been previously involved in a comparison. This is a repetitive use of T[1] in another comparison.
- It is these repetitive comparisons that lead to the runtime of O(mn).

## KMP Approach

Knuth, Morris and Pratt (KMP) proposed a linear time algorithm for the string matching problem.

A matching time of O(n) is achieved by avoiding comparisons with elements of ‘S’ that have previously been involved in comparison with some element of the pattern ‘p’ to be matched. i.e., backtracking on the string ‘S’ never occurs

## Components and Terminology of KMP Algorithm

For understanding this section, let’s take an example pattern :
Pattern : a b c d a b c
**Prefix :**

In the KMP algorithm, we have two terms, proper prefix and suffix. A proper prefix of the pattern will be a subset of the pattern using only the beginning portion (the first index), or the first few indices of the pattern. How many characters should be in the substring prefix?
You can take as many as you want, except the last element. Why? If you take the last element of the string as well, it is the complete string, and will not be counted as a proper prefix.
Some examples of prefixes of the above pattern are :

- a
- ab
- abc
- abcd

## Components and Terminology of KMP Algorithm

**Suffix :**

A proper suffix of any pattern would be a subset of the pattern with elements taken only from the right end of the pattern as in, any number of elements, starting from the last character.
Taking the first character of the string is not allowed, as it would be the complete string and hence will not be considered as a proper suffix of the string. Some examples for the same are :

- c
- bc
- abc
- Dabc

Note that there could be more than the above-mentioned prefixes and suffixes.

## Components and Terminology of KMP Algorithm

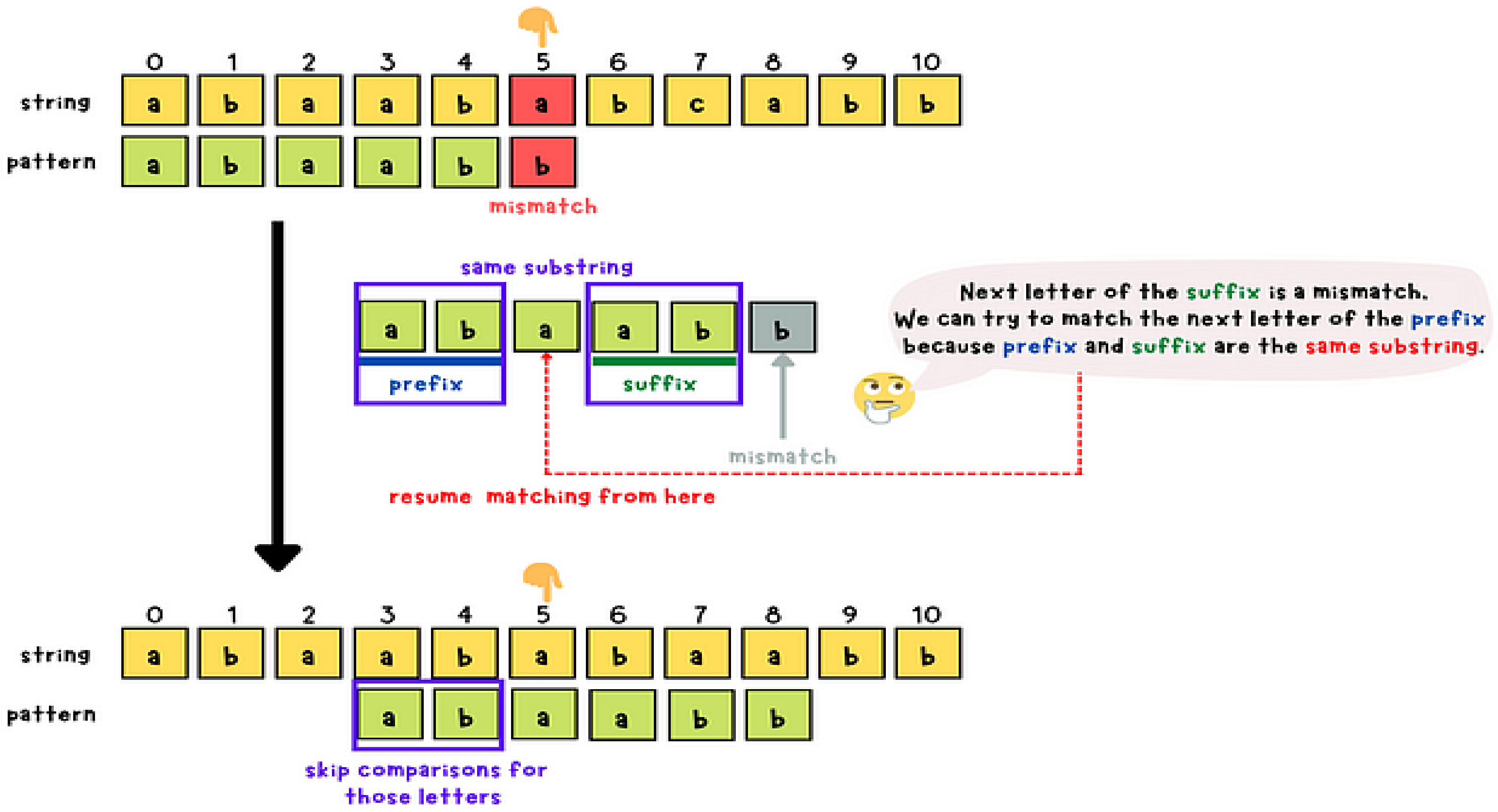
**LPS Array**

**LPS** stands for **Longest Prefix Suffix. It is used to find pattern over the given pattern for matching to overcome the limitation of O(mn) complexity approaches.**

While implementing string matching, having LPS information is quite handy. After a mismatch, we go back to the letter right before the current mismatch letter and check if LPS is present in the substring ending at this letter. The prefix and suffix are the same substring that appears at different positions in a string. If the suffix has been matched correctly, the prefix must be matched too. Therefore, we just jump back to the next letter of the early occurring prefix to resume the process of string matching.

Knuth-Morris-Pratt(KMP) Algorithm utilizes LPS array computed from the Pattern to determine how many **letters can be skipped for comparison after a mismatch.**

## How is LPS related to String Matching



0 1 2 3 4 5 6 7 8 9 10
string a b a a b a b c a b b
pattern a b a a b b
mismatch
same substring
a b a a b b
prefix
suffix
mismatch
resume matching from here
Next letter of the suffix is a mismatch.
We can try to match the next letter of the prefix
because prefix and suffix are the same substring.
0 1 2 3 4 5 6 7 8 9 10
string a b a a b a b a a b b
pattern a b a a b b
skip comparisons for those letters

# Compute LPS Array

Let us prepare the Longest Prefix Suffix (LPS) table using a few pattern examples.

**Example 1: Pattern string P → o n i o n s**

**LPS Array:**

| o | n | i | o | n | s |
|---|---|---|---|---|---|
| 0 | 0 | | | | |

Step 1: pick 1st character from the P string i.e., **o**

find prefix of o:- null
find suffix of o:- null
prefix != suffix; Thus, LPS value = 0

Step 2: Consider 1st and 2nd characters from the P string i.e., **o n**

find prefix of on:- o
find suffix of on:- n
prefix != suffix; Thus, LPS value = 0

# Compute LPS Array

**LPS Array:**

| o | n | i | o | n | s |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | | |

Step 3: Consider next character from the P string i.e., **o n i**

find prefix of **o n i**:- o , on

find suffix of **o n i** :- i, ni

prefix != suffix;    Thus, LPS value = 0

Step 4: Consider next character from the P string i.e., **o n i o**

find prefix of **o n i o**:- o, on, oni

find suffix of **o n i o**:- o, io, nio

**prefix == suffix; Thus, LPS value = 1 [ prefix 'o' matches with suffix 'o'. Length of 'o' is 1]**

## Compute LPS Array

**LPS Array:**

| o | n | i | o | n | s |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 2 | 0 |

Step 5: Consider next character from the P string i.e., **o n i o n**

find prefix of **o n i o n** :- o , on, oni, onio

find suffix of **o n i o n** :- n, on, ion, nion

**prefix == suffix; Thus, LPS value = 2 [ prefix 'on' matches with suffix 'on'. Length of 'on' is 2]**

Step 6: Consider next character from the P string i.e., **o n i o n s**

find prefix of **o n i o n s**:- o, on, oni, onio, onion

find suffix of **o n i o n s**:- s, ns, ons, ions, nions

prefix != suffix; Thus, LPS value = 0

## Alternate way to Compute LPS Array: Shortcut method

| **index** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **pattern P** | o | n | i | o | n | s |
| **LPS Array** | 0 | 0 | 0 | 1 | 2 | 0 |

```
1.  Set j at index 0
2.  Set i at index 1
3.  Initially set LPS[0] = 0
4.  If P[j] == P[i]
      i.   LPS[i] = j+1
      ii.  Increment both i and j by 1
      iii. Go to step 4
5.  Else if (j!= 0)
      i.   j=LPS [j-1]
      ii.  Go to step 4
6.  Else
      i.   LPS[i] =0
      ii.  Increment i by 1
      iii. Go to step 4
```

## Activity-1
## Computer LPS Array

**Example 2:**
**Pattern P = a b c d a b c a**

**Answer:**

| a | b | c | d | a | b | c | a |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 2 | 3 | 1 |

## Activity-2
## Computer LPS Array

**Example 3:**
**Pattern P = a a b a a b a a a**

**Answer:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **a** | **a** | **b** | **a** | **a** | **b** | **a** | **a** | **a** |
| **0** | **1** | **0** | **1** | **2** | **3** | **4** | **5** | **2** |

# KMP String Matching Process

**Text (T): ABABDABACDABABCABAB**

**Pattern (P): ABABCABAB**

**Step 1: Compute LPS Array for the Pattern P**

LPS array = [0, 0, 1, 2, 0, 1, 2, 3, 4]

**Step 2: Matching Using LPS**

We use two pointers:

- i → Text pointer; starts from 0
- j → Pattern pointer; starts from 0

**Matching Steps**:

1. T[0] = P[0] → A = A ✅ i++, j++
2. T[1] = P[1] → B = B ✅ i++, j++
3. T[2] = P[2] → A = A ✅ i++, j++
4. T[3] = P[3] → B = B ✅ i++, j++
5. T[4] = P[4] → D ≠ C ❌ → Use LPS: j = LPS[j-1] = LPS[3] = 2
6. T[4] = P[2] → D ≠ A ❌ → j = LPS[j-1] = LPS[1] = 0
7. T[4] = P[0] → D ≠ A ❌ → i++; as j already at index 0
8. T[5] = P[0] → A = A ✅ i++, j++

7.Move i and j forward and repeat...

Eventually, the pattern is found at index **10** in the text.

## Pseudocode: KMP String Matching

**LPS Array Construction**

```
function computeLPSArray (P, lps [] ):
    j ← 0
    lps[0] ← 0
    i ← 1
    while i < length of pattern:
        if P[i] == P[j]:
            lps[i] ← j+1
            i ← i + 1
            j ← j + 1
        else:
            if j != 0:
                j ← lps[j - 1]
            else:
                lps[i] ← 0
                i ← i + 1
    return lps
```

**KMP Search**

```
function KMPSearch(text, pattern):
    n ← length of text
    m ← length of pattern
    lps ← computeLPSArray(pattern)
    i ← 0  // index for text
    j ← 0  // index for pattern
    while i <= n-m:
        if P[j] == T[i]:
            i ← i + 1
            j ← j + 1
        else
            if  j != 0:
                j ← lps[j - 1]
            else:
                i ← i + 1
        if j == m:
            print "Pattern found at index", (i - j)
            j ← lps[j - 1]  // Look for next possible match
```

## Complexity Analysis

- **LPS computation:** O(m)
- **Matching:** O(n)
- **Total: O(m + n)**

*In the next class we will going through Strassen's Matrix Multiplication*

Thank You