# ITDO 5014 ADSA
# Module 5

# String Matching

# ITDO 5014 ADSA

## Module 5 : String Matching (7 Hrs )

1. Introduction
2. The naïve string matching algorithm
3. Rabin Karp algorithm
4. Boyer Moore algorithm
5. Knuth-Morris-Pratt algorithm (KMP)
6. Longest common subsequence(LCS)

Analysis of All algorithms and problem solving.

Self-learning Topics: Implementation of Robin Karp algorithm, KMP algorithm and LCS.

# 1.Naive algorithm for Pattern Searching :

**Given a text : txt[0..n-1] and a pattern : pat[0..m-1],**

**write a function search ( char pat[], char txt[] )**
**that prints all occurrences of pat[] in txt[] where n > m.**

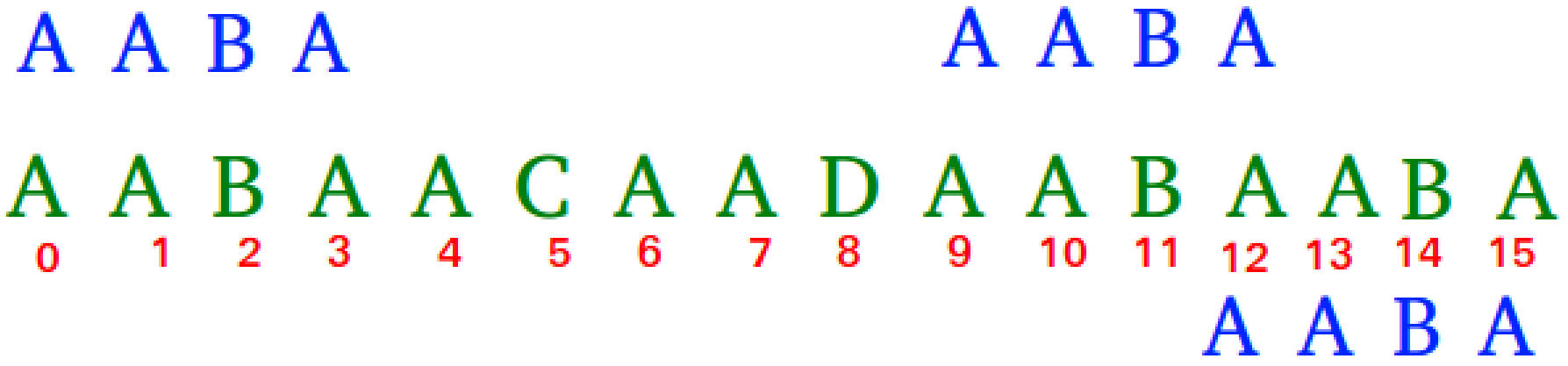
**Examples:**

**Input: txt[] = "THIS IS A TEA POT"**
**pat[] = "TEA"**
**Output: Pattern found at index 10**

**Input: txt[] = "AABAACAADAABAABA"**
**pat[] = "AABA"**
**Output: Pattern found at index 0**
**Pattern found at index 9**
**Pattern found at index 12**

Text : A A B A A C A A D A A B A A B A

Pattern : A A B A




Pattern Found at 0, 9 and 12

**Pattern searching is an important problem in computer science.**

**When we do search for a string in notepad/word file or browser or database,**

**pattern searching algorithms are used to show the search results.**

## Naive Pattern Searching:

**Slide the pattern over text one by one and check for a match.**

**If a match is found, then slides by 1 again to check for subsequent matches.**

```
// C program for Naive Pattern Searching algorithm
 void search(char* pat, char* txt)
{           int M = strlen(pat);
            int N = strlen(txt);

   /* A loop to slide pat[] one by one */
   for (int i = 0; i <= N - M; i++)          O(n–m +1)
   {          int j;
 // For current index i, check for pattern match
        for (j = 0; j < M; j++)                  O (m)
            if (txt[i + j] != pat[j])
                break;
        if (j == M)       // if pat[0...M-1] = txt[i, i+1,
                          //  ...i+M-1]
         printf("Pattern found at index %d \n", i);
   }
}
```

```
/* Driver program to test above function */

int main()
{                    012345678910
   char txt[] = "AABAACAADAABAAABAA";
   char pat[]= "AABA";

   search(pat, txt);

   return 0;
}
```

**Output:**

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

**Best case :**

**The best case occurs when the first character of the pattern is not present in text at all.**

**txt[] = "AABCCAADDEE";**
**pat[] = "FAA";**

**The number of comparisons in best case is O(n).**

**Worst case :**

**The worst case of Naive Pattern Searching occurs in following scenarios.**

**1) When all characters of the text and pattern are same.**

**txt[] = "AAAAAAAAAAAAAAAAAA";**
**pat[] = "AAAAA";**

**2) Worst case also occurs when only the last character is different.**

**txt[] = "AAAAAAAAAAAAAAAAAB";**
**pat[] = "AAAAB";**

**The number of comparisons in the worst case is O(m*(n-m+1)).**

**Although strings which have repeated characters are not likely to appear in English text, they may well occur in other applications (for example, in binary texts).**

**The KMP matching algorithm improves the worst case to O(n).**

# ITDO 5014 ADSA

## Module 5 : String Matching (7 Hrs )

1. Introduction
2. The naïve string matching algorithm
3. Rabin Karp algorithm
4. Boyer Moore algorithm
5. Knuth-Morris-Pratt algorithm (KMP)
6. Longest common subsequence(LCS)

Analysis of All algorithms and problem solving.

Self-learning Topics: Implementation of Robin Karp algorithm, KMP algorithm and LCS.

# 2. Rabin-Karp Algorithm

## String Matching

# Introduction

- **Rabin-Karp algorithm** is a **string-matching algorithm** created by Richard M. Karp and Michael O. Rabin that uses **hashing** to find any one of a **set of pattern strings** in a text.
- It moves from one ‘window’ to another to check hash value without checking all characters of all cases. When the **hash value** of the **window** is **matched**, then only it **checks each character**.
- This makes algorithm more efficient.

- NAÏVE ALGO : ➔
  - **txt[] = "AAAAAAAAAAAAAAAAAA";**
  - **pat[] = "AAAAA";**

- **2) Worst case also occurs when only the last character is different.**
  - **txt[] = "AAAAAAAAAAAAAAAAAB";**
  - **pat[] = "AAAAB";**
-

# Pseudo Code

Step 1: Take input text ‘T’.
Let ‘n’ be the length of T.

Step 2: Let ‘P’ be a pattern (substring) of T.
Let ‘m’ be the length of P.

Step 3: Compute hashing using a suitable hash function **(mod hash function preferred)**
Choose a **random prime number** ‘q’ and find **P mod q**

Step 4: for i=0 to n-m

if hash value matches do:
if all characters of window **match**,
then **valid hit**

if all characters of window **don’t match**,
then **spurious hit**

# Example

**Q. Apply Rabin-Karp algorithm to Find whether the input text contains the given pattern ( 234 ) or not.**

Step 1: T =

| 3 | 1 | 2 | 3 | 4 | 8 | 6 | 2 |
|---|---|---|---|---|---|---|---|

Length of input text (n) = 8

for i=0 to n-m
  if hash value matches do:
    if all characters of window **match**,
      then **valid hit**
    if all characters of window **don't match**,
      then **spurious hit**

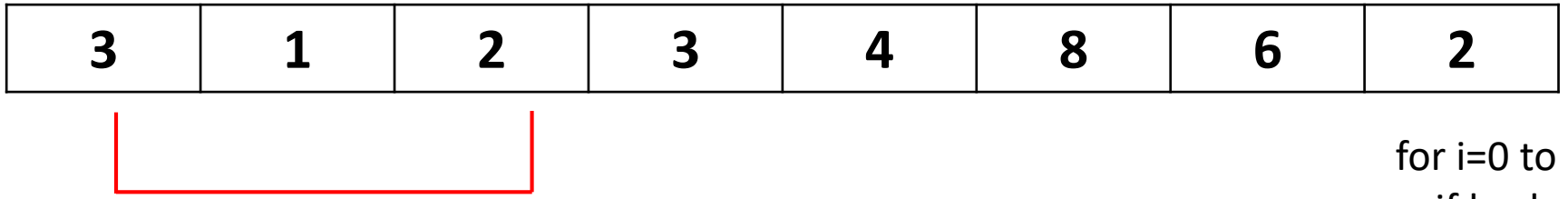
Step 2: Suppose we want to match a pattern **P=234**
Length of pattern (m) = 3
Hence, **each window is of size = 3**

Step 3: Compute hashing. Choose any random prime number q.
Let q=13

∴ P mod q = 234 mod 13 = 0
∴ H = 0

# Example

**Window 1:**

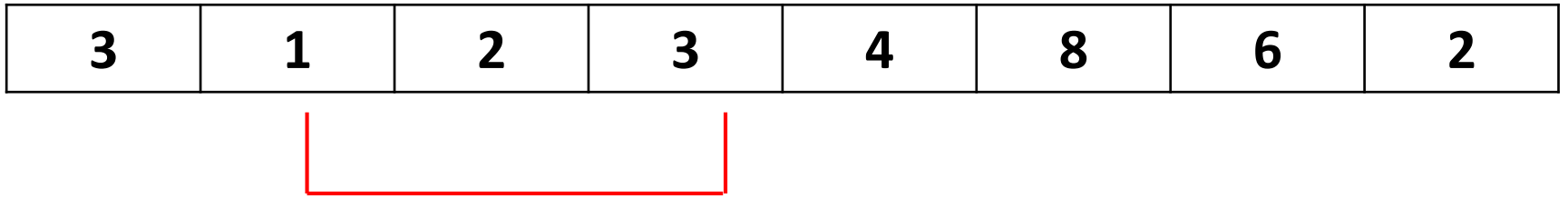



312 mod 13 = 0  Hash value matches.

Now check for each character of the window.

312 ≠ 234. **Spurious Hit.**

**Window 2:**




123 mod 13 = 6

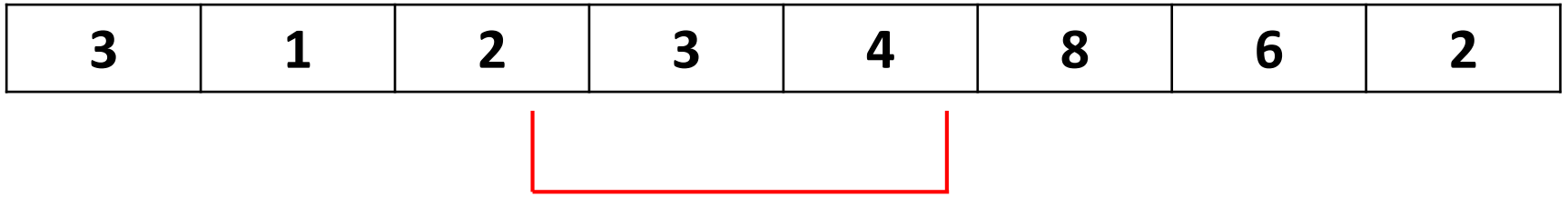
Hash value doesn't matches.

for i=0 to n-m
  if hash value matches do:
    if all characters of window **match**,
      then **valid hit**

    if all characters of window **don't match**,
      then **spurious hit**

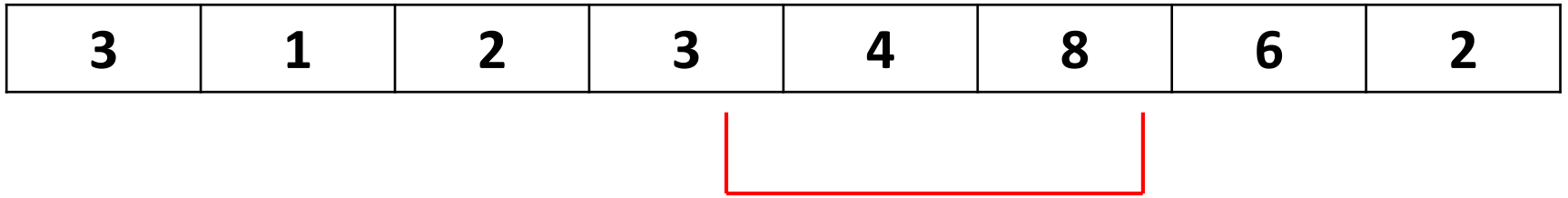
# Example

**Window 3:**




234 mod 13 = 0
Hash value matches. Now check for each character of the window.
234 = 234. **Valid Hit**. Pattern found at position 3

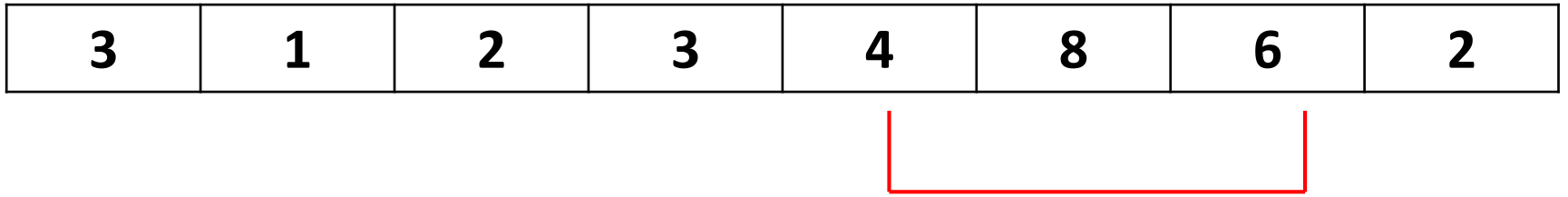
**Window 4:**




348 mod 13 = 10
Hash value doesn’t matches.

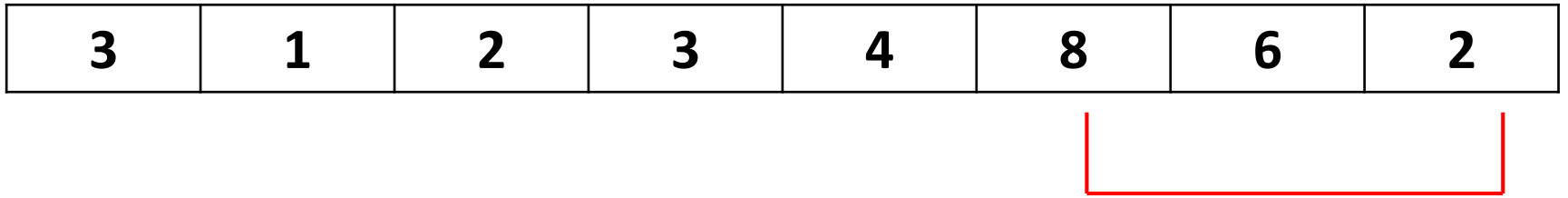
# Example

**Window 5:**




486 mod 13 = 5
Hash value doesn’t matches.

**Window 6:**




862 mod 13 = 4
Hash value doesn’t matches.

**Q. Apply Rabin-Karp algorithm to Find whether the input text contains the given pattern ( NST ) or not.**

| A | N | S | T | Z | X | N | S | T | U |
|---|---|---|---|---|---|---|---|---|---|

Step 1: T =

Length of input text (n) = 10

Step 2: Suppose we want to match a pattern **P=NST**

Length of pattern (m) = 3

Hence, **each window is of size = 3**

Step 3: Now since the input text is an alphabetical string in this case,

we can use either **ASCII codes** or **(A:1 – Z:26)** system.

Compute hashing using a suitable hash function.

N(14) + S(19) + T(20) = 53

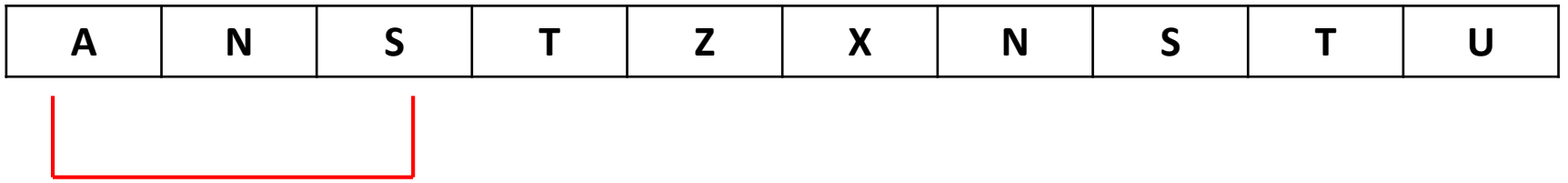
Choose any random prime number ‘q’.

Let q=17

∴ 53 mod 17 = 2

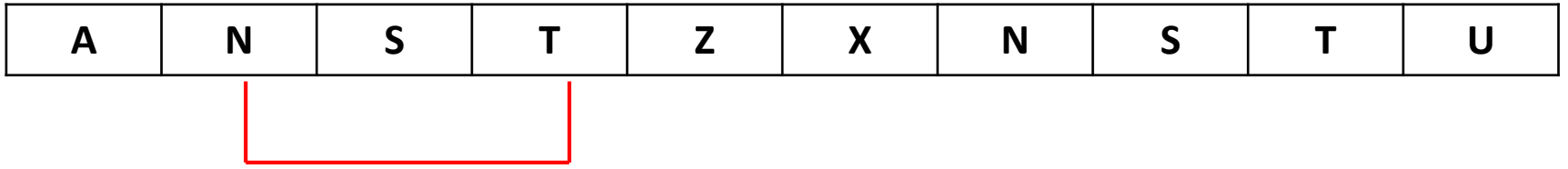
# Example

**Window 1:**




A(1) + N(14) + S(19) mod 17 = 34 mod 17 = 0
Hash value doesn't matches.

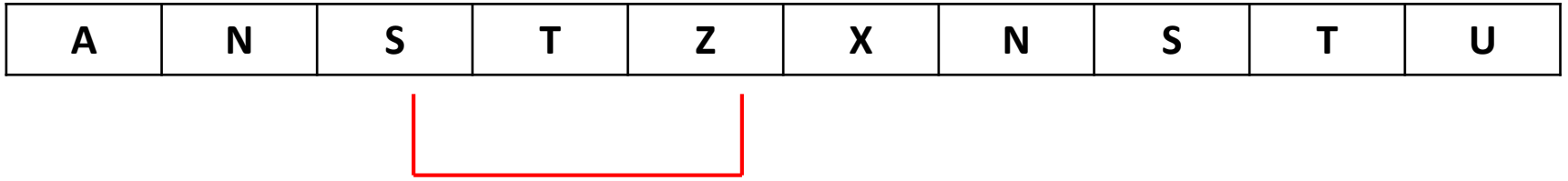
**Window 2:**




N(14) + S(19) + T(20) = 53 mod 17 = 2
Hash value matches. Now check for each character of the window.
53 = 53. **Valid Hit**. Pattern found at position 2

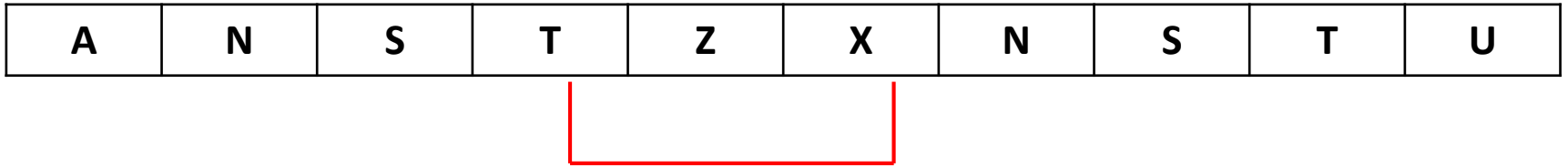
# Example

**Window 3:**




S(19) + T(20) + Z(26) mod 17 = 65 mod 17 = 14
Hash value doesn’t matches.

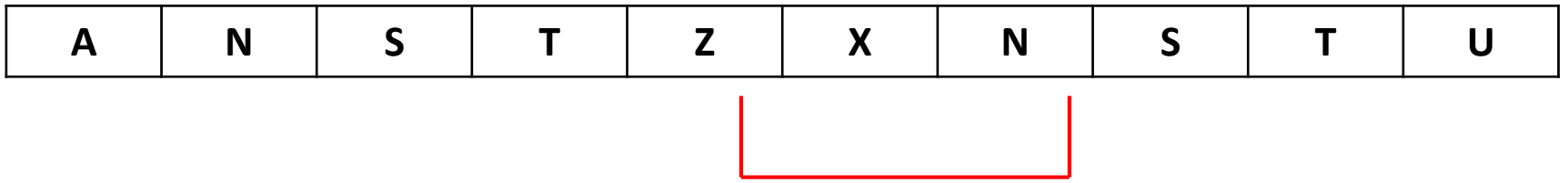
**Window 4:**




T(20) + Z(26) + X(24) = 70 mod 17 = 2
Hash value matches. Now check for each character of the window.
70 ≠ 53. **Spurious Hit**.

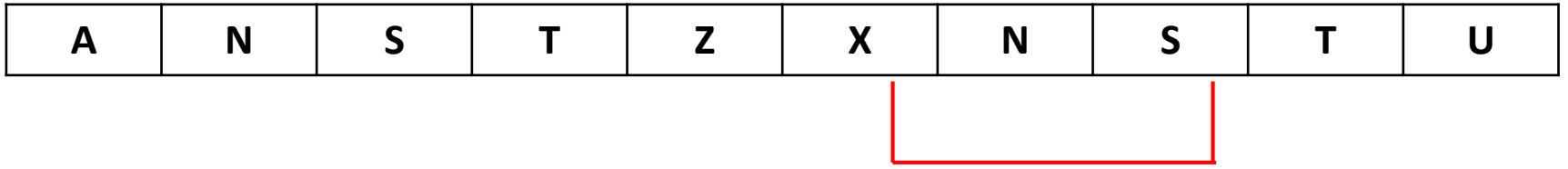
# Example

**Window 5:**




Z(26) + X(24) + N(14) mod 17 = 64 mod 17 = 13
Hash value doesn’t matches.

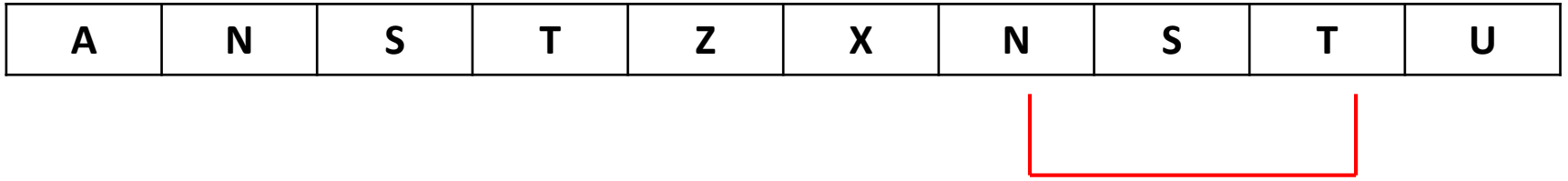
**Window 6:**




X(24) + N(14) + S(19) = 57 mod 17 = 6
Hash value doesn’t matches.

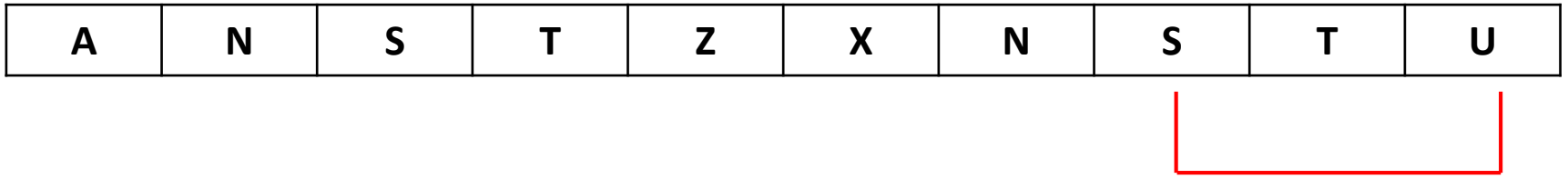
# Example

**Window 7:**




N(14) + S(19) + T(20) mod 17 = 53 mod 17 = 2
Hash value matches. Now check for each character of the window.
53 = 53. **Valid Hit**. Pattern found at position 7

**Window 8:**




S(19) + T(20) + U(21) = 60 mod 17 = 9
Hash value doesn’t matches.

# Algorithm

```
Rabin_Karp (T, P)
{
n = T.length                          // length of input text
m = P.length                          // length of pattern
h^p = Hash ( P[…] )                   // P mod q
h^T = Hash ( T[…] )                   // T mod q

for i=0 to n-m                O(n+m)
{
   if (h^p == h^T)                    // if hash values are equal
   {
       if (P [0 … m-1] == T [i+0… i+m-1])   // if pattern and window characters are equal
           {
               print “Pattern found at position i+1”
           }
   }
}
}
```

# Time Complexity & Applications

**Time Complexity:**

The time complexity of Rabin-Karp algorithm is **O(n+m)** where,
n = length of input text
m = length of pattern

**Applications:**

(i) Plagiarism detection
(ii) Text processing
(iii) Bioinformatics
(iv) Compression

# ITDO 5014 ADSA

## Module 5 : String Matching (7 Hrs )

1. Introduction
2. The naïve string matching algorithm
3. Rabin Karp algorithm
4. Boyer Moore algorithm
5. Knuth-Morris-Pratt algorithm (KMP)
6. Longest common subsequence(LCS)

Analysis of All algorithms and problem solving.

Self-learning Topics: Implementation of Robin Karp algorithm, KMP algorithm and LCS.

# 3.Knuth-Morris-Pratt (KMP) Algorithm

# The problem of String Matching

Given a string ‘S’, the problem of string matching deals with finding whether a pattern ‘p’ occurs in ‘S’ and if ‘p’ does occur then returning position in ‘S’ where ‘p’ occurs.

# …. a O(mn) approach

One of the most obvious approach towards the string matching problem would be to compare the first element of the pattern to be searched ‘p’, with the first element of the string ‘S’ in which to locate ‘p’.

If the first element of ‘p’ matches the first element of ‘S’, compare the second element of ‘p’ with second element of ‘S’.

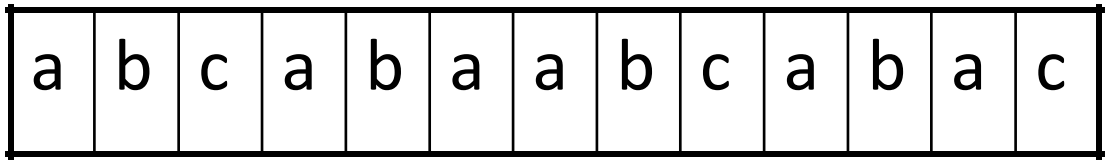
If match found proceed likewise until entire ‘p’ is found.

If a mismatch is found at any position, shift ‘p’ one position to the right and repeat comparison beginning from first element of ‘p’.

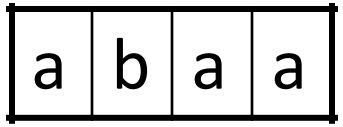
## How does the O(mn) approach work :

**Below is an illustration of how the previously described O(mn) approach works.**

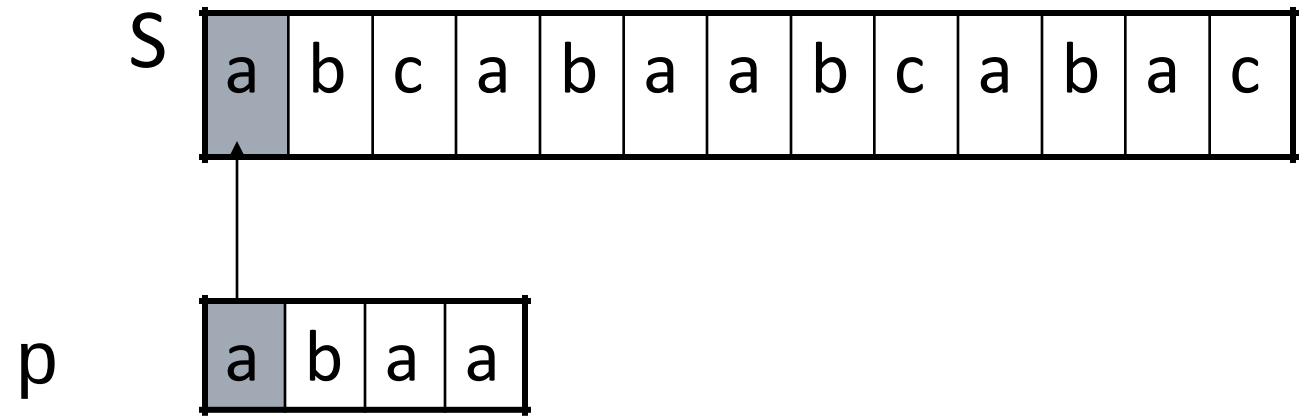
**String S**



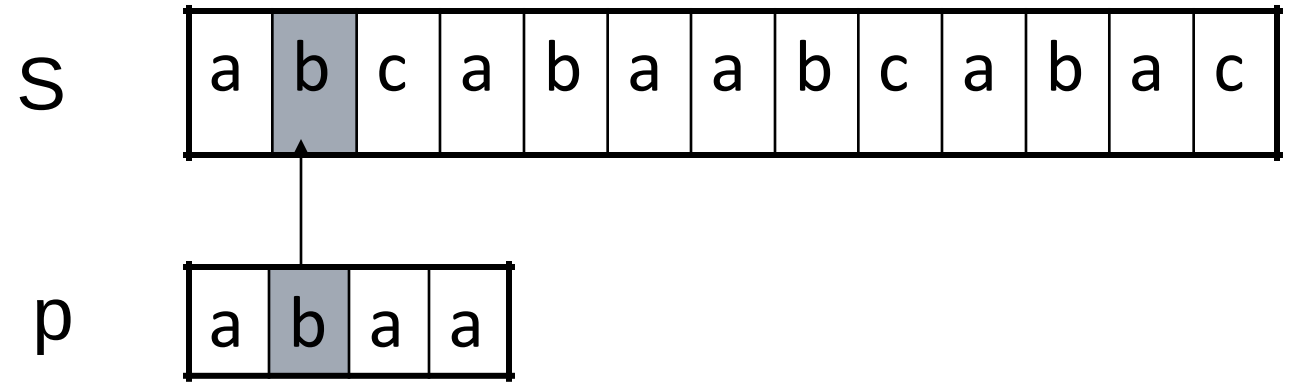

**Pattern p**



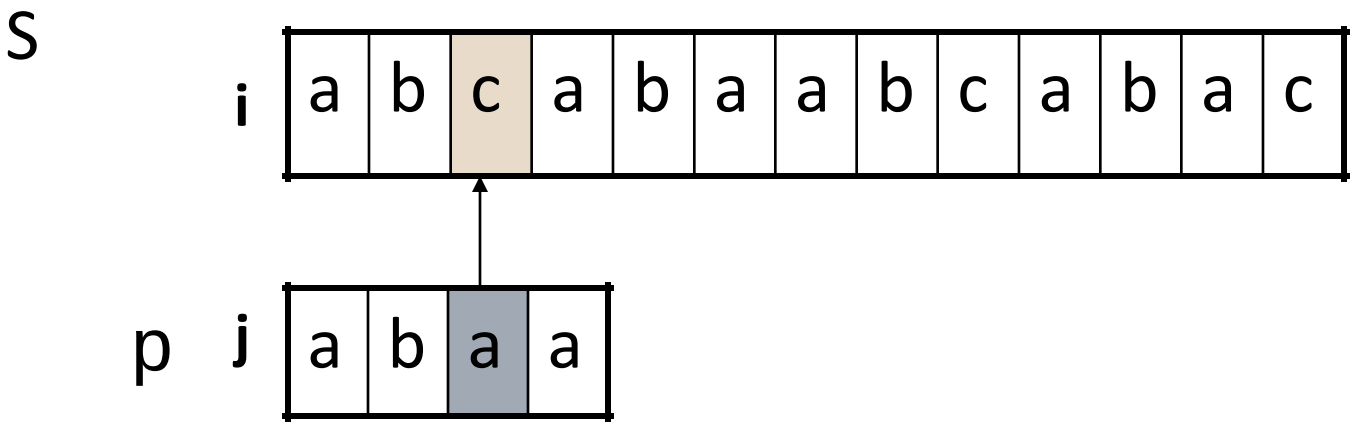
Step 1:compare p[1] with S[1]




Step 2: compare p[2] with S[2]

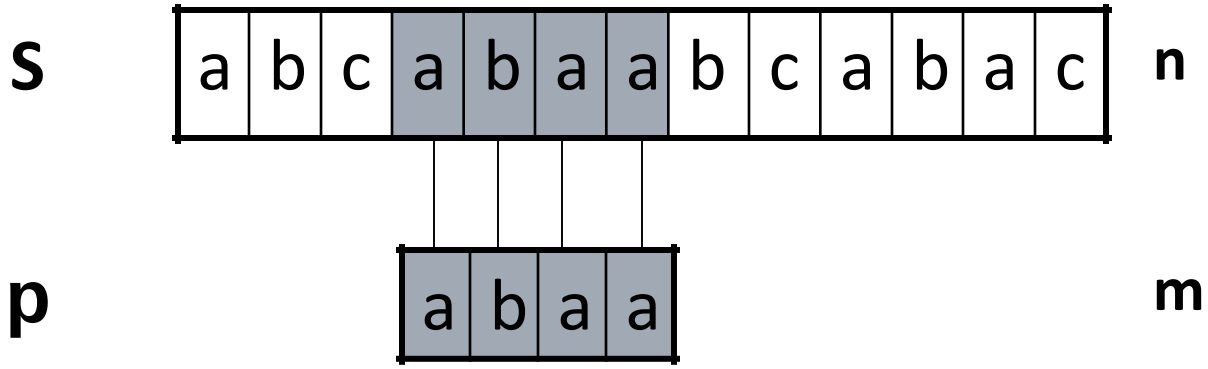


# Step 3: compare p[3] with S[3]




*Mismatch occurs here..*

Since mismatch is detected, shift ‘p’ one position to the left and perform steps analogous to those from step 1 to step 3. At position where mismatch is detected, shift ‘p’ one position to the right and repeat matching procedure.




**Finally, a match would be found after shifting 'p' three times to the right side.**

**Drawbacks of this approach: if 'm' is the length of pattern 'p' and 'n' the length of string 'S', the matching time is of the order O(mn). This is a certainly a very slow running algorithm.**

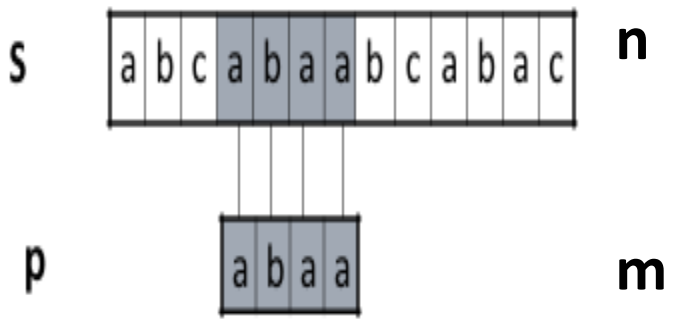
**What makes this approach so slow is the fact that elements of 'S' with which comparisons had been performed earlier are involved again and again in comparisons in some future iterations.**

**For example: when mismatch is detected for the first time in comparison of p[3] with S[3], pattern 'p' would be moved one position to the right and matching procedure would resume from here.**

**Here the first comparison that would take place would be between p[0]='a' and S[1]='b'. It should be noted here that S[1]='b' had been previously involved in a comparison in step 2. this is a repetitive use of S[1] in another comparison.**

**It is these repetitive comparisons that lead to the runtime of O(mn).**

# The Knuth-Morris-Pratt Algorithm :




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

A matching time of O(n) is achieved by avoiding comparison 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 of KMP algorithm

- The prefix function, Π

The prefix function,Π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself.

This information can be used to avoid useless shifts of the pattern ‘p’.
In other words, this enables avoiding backtracking on the string ‘S’.

- The KMP Matcher

With string ‘S’, pattern ‘p’ and prefix function ‘Π’ as inputs,

finds the occurrence of ‘p’ in ‘S’ and returns the number of shifts of ‘p’ after which occurrence is found.

# The prefix function, Π

Following pseudocode computes the prefix fucnction, Π:

Compute-Prefix-Function (p)

```
1  m ← length[p]              //'p' pattern to be matched
2  Π[1] ← 0
3  k ← 0
4       for q ← 2 to m
5            do while k > 0 and p[k+1] != p[q]
6                 do k ← Π[k]
7               If p[k+1] = p[q]
8                 then k ← k +1
9               Π[q] ← k
10     return Π
```

# Example: compute Π for the pattern ‘p’ below:

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

Initially: m = length[p] = 7
Π[1] = 0
k = 0

Step 1: q = 2, k=0
Π[2] = 0

| q | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| p | a | b | a | b | a | c | a |
| Π | 0 | 0 | | | | | |

Step 2: q = 3, k = 0,
Π[3] = 1

| q | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| p | a | b | a | b | a | c | a |
| Π | 0 | 0 | 1 | | | | |

Step 3: q = 4, k = 1
Π[4] = 2

| q | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| p | a | b | a | b | a | c | A |
| Π | 0 | 0 | 1 | 2 | | | |

Step 4: q = 5, k =2

Π[5] = 3

| q | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| p | a | b | a | b | a | c | a |
| Π | 0 | 0 | 1 | 2 | 3 | | |

Step 5: q = 6, k = 3

Π[6] = 1

| q | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| p | a | b | a | b | a | c | a |
| Π | 0 | 0 | 1 | 2 | 3 | 1 | |

Step 6: q = 7, k = 1

Π[7] = 1

| q | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| p | a | b | a | b | a | c | a |
| Π | 0 | 0 | 1 | 2 | 3 | 1 | 1 |

After iterating 6 times, the prefix function computation is complete:

→

| q | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| p | a | b | A | b | a | c | a |
| Π | 0 | 0 | 1 | 2 | 3 | 1 | 1 |

# The KMP Matcher

The KMP Matcher, with pattern ‘p’, string ‘S’ and prefix function ‘Π’ as input, finds a match of p in S. Following pseudocode computes the matching component of KMP algorithm:

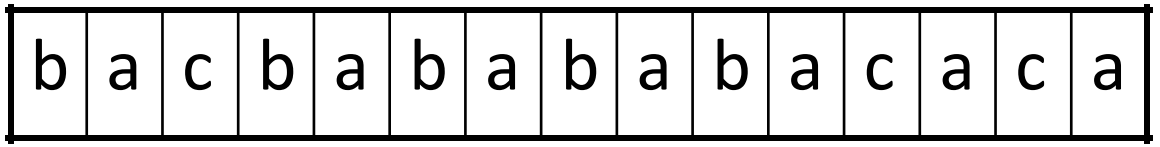
KMP-Matcher(S,p)

```
1 n ← length[S]
2 m ← length[p]
3 Π ← Compute-Prefix-Function(p)
4 q ← 0                                   //number of characters matched
5 for i ← 1 to n                          //scan S from left to right
6     do while  q > 0 and p[q+1] != S[i]
7             do  q ← Π[q]                //next character does not match
8         if p[q+1] = S[i]
9           then q ← q + 1                //next character matches
10        if q = m                        //is all of p matched?
11          then print “Pattern occurs with shift” i – m
12               q ← Π[ q]                // look for the next match
```

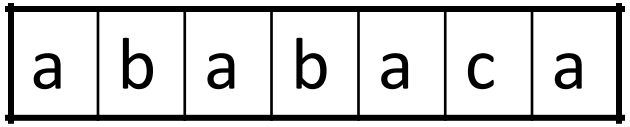
13 *Note: KMP finds every occurrence of a ‘p’ in ‘S’. That is why KMP does not terminate in step 12, rather it searches remainder of ‘S’ for any more occurrences of ‘p’.*

Illustration: given a String ‘S’ and pattern ‘p’ as follows:

S




p



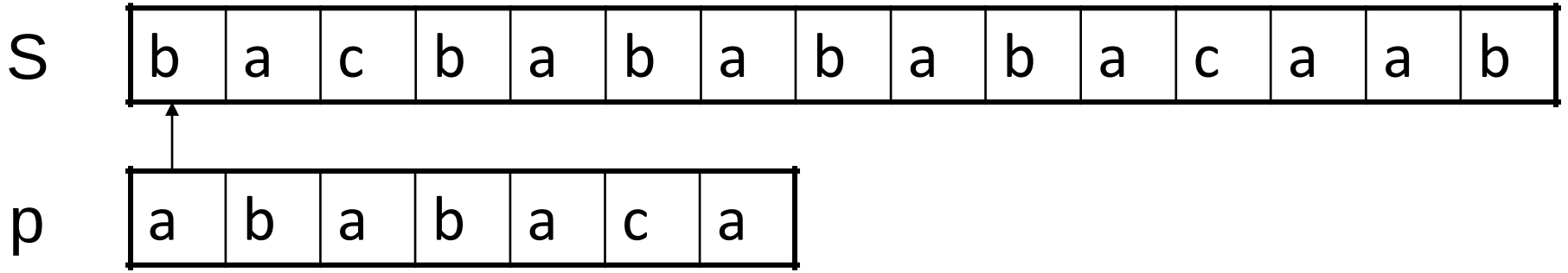

Let us execute the KMP algorithm to find whe ther ‘p’ occurs in ‘S’.

*For ‘p’ the prefix function, Π was computed previously and is as follows:*

| q | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| p | a | b | A | b | a | c | a |
| Π | 0 | 0 | 1 | 2 | 3 | 1 | 1 |

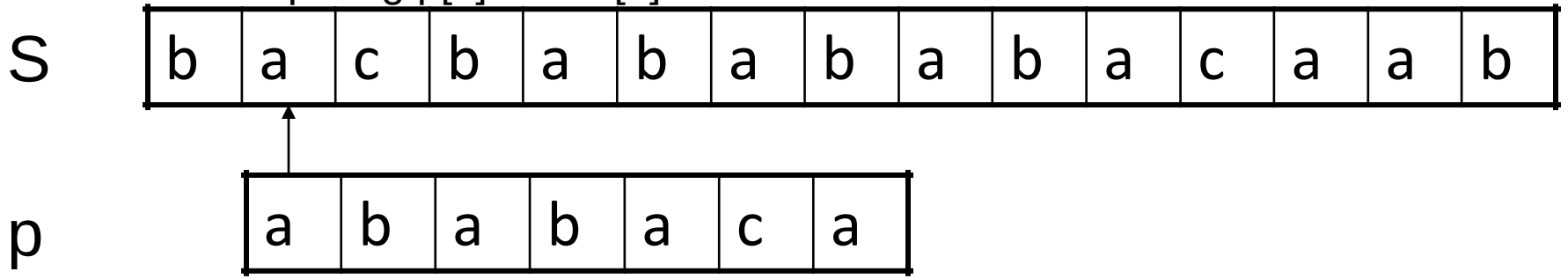
Initially: n = size of S = 15;
m = size of p = 7

Step 1: i = 1, q = 0
comparing p[1] with S[1]




P[1] does not match with S[1]. ‘p’ will be shifted one position to the right.

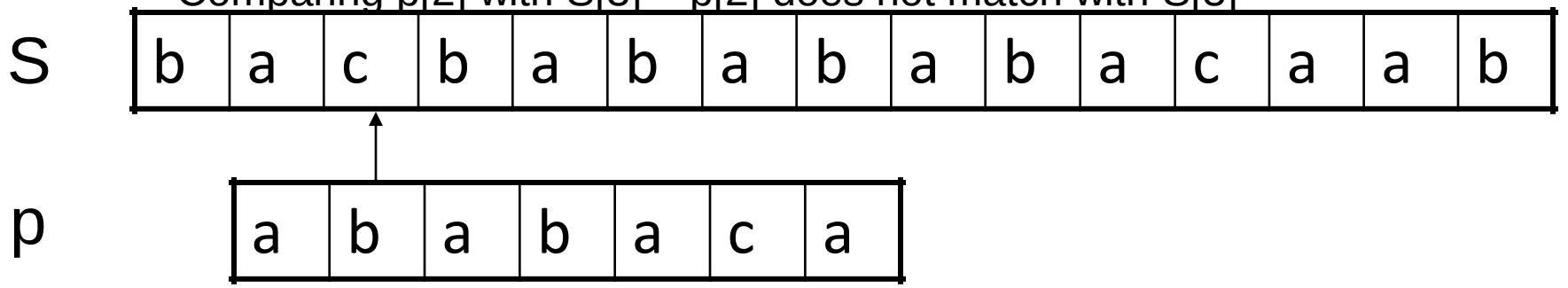
Step 2: i = 2, q = 0
comparing p[1] with S[2]




P[1] matches S[2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

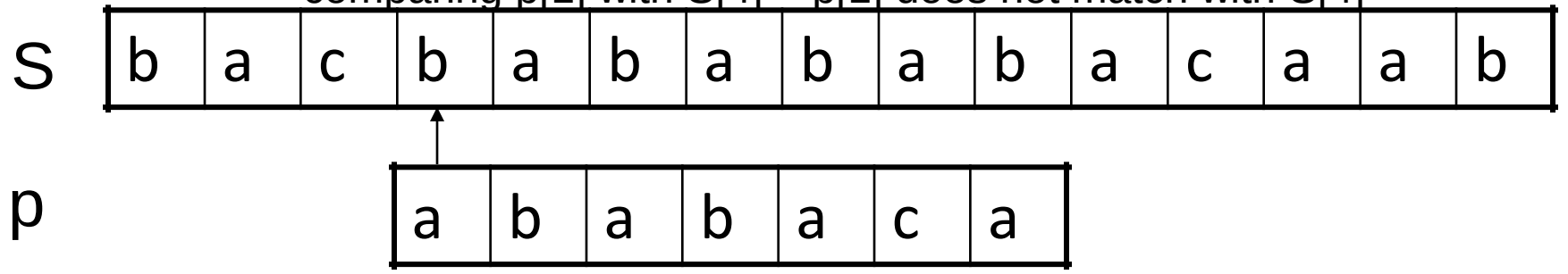
Comparing p[2] with S[3]  p[2] does not match with S[3]




Backtracking on p, comparing p[1] and S[3]

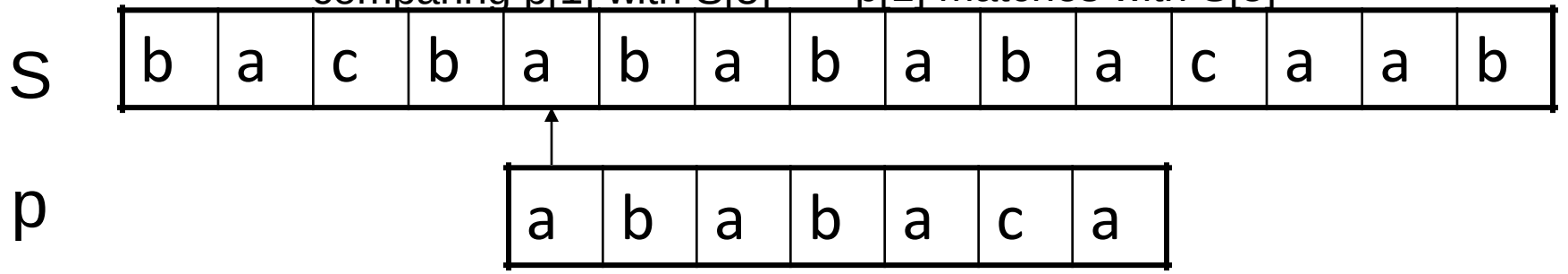
Step 4: i = 4, q = 0

comparing p[1] with S[4]  p[1] does not match with S[4]



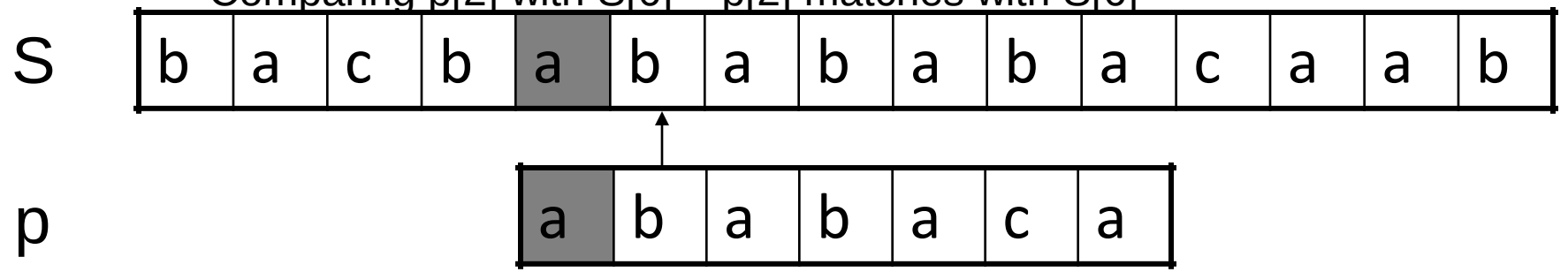

Step 5: i = 5, q = 0

comparing p[1] with S[5]  p[1] matches with S[5]



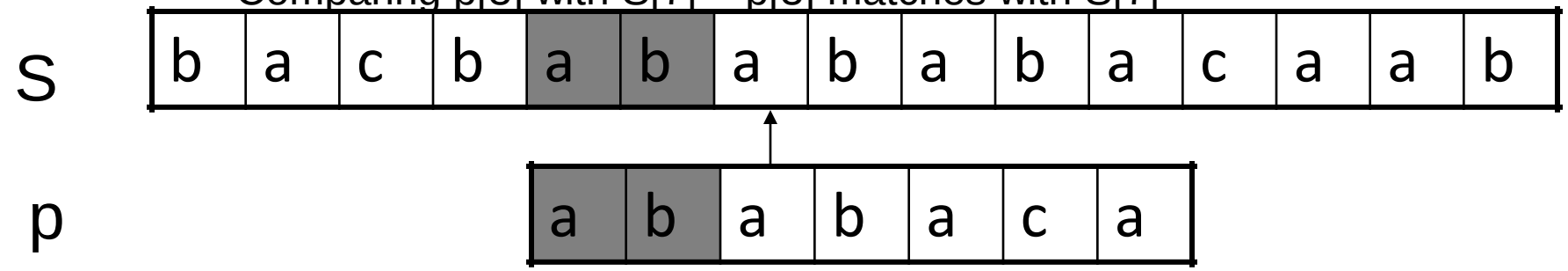
Step 6: i = 6, q = 1

Comparing p[2] with S[6]  p[2] matches with S[6]



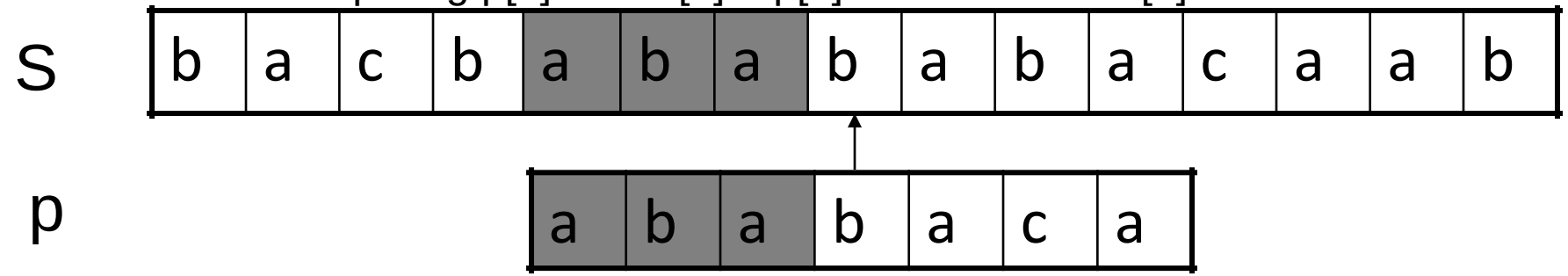

Step 7: i = 7, q = 2

Comparing p[3] with S[7]  p[3] matches with S[7]

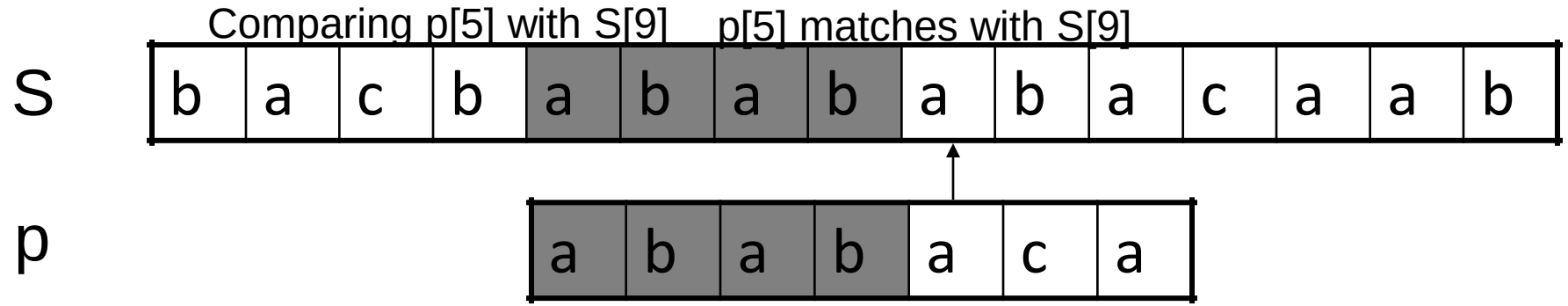



Step 8: i = 8, q = 3

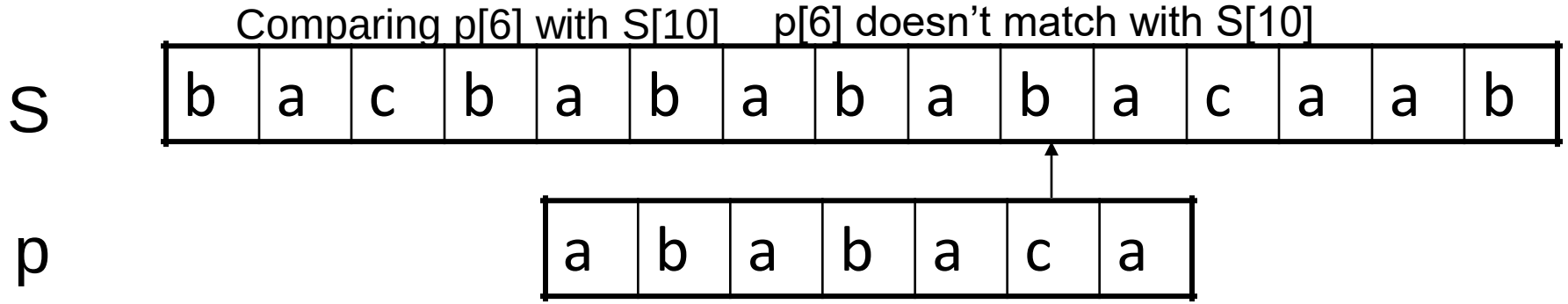
Comparing p[4] with S[8]  p[4] matches with S[8]



Step 9: i = 9, q = 4

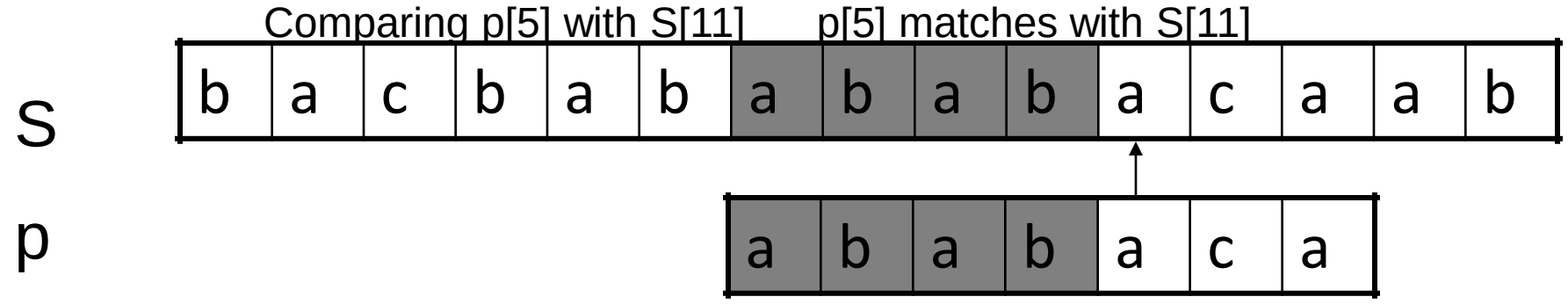



Step 10: i = 10, q = 5

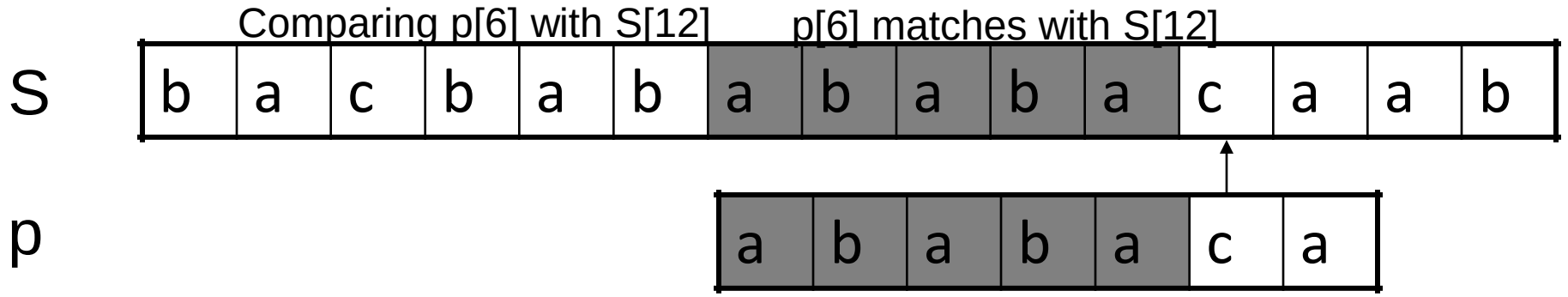



Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

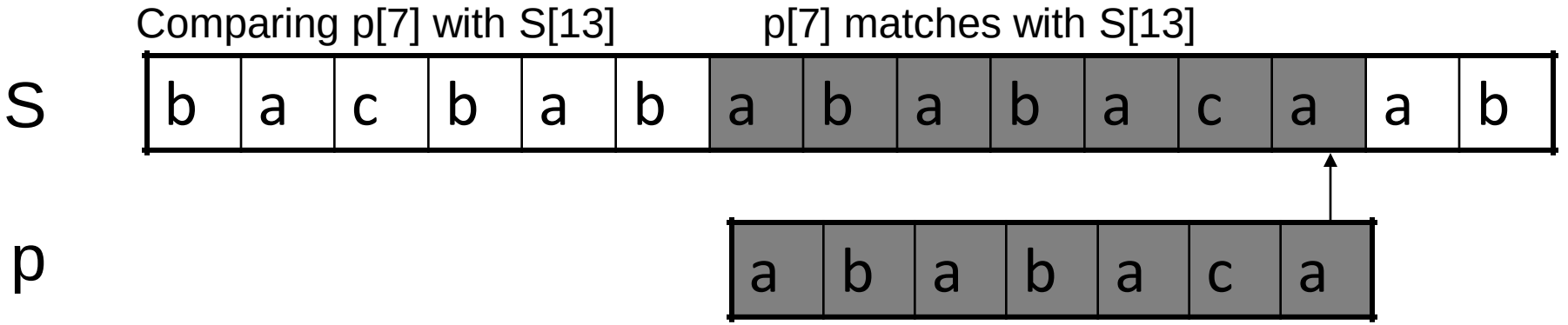
Step 11: i = 11, q = 4



Step 12: i = 12, q = 5




Step 13: i = 13, q = 6




Pattern ‘p’ has been found to completely occur in string ‘S’. The total number of shifts that took place for the match to be found are: i – m = 13 – 7 = 6 shifts.

# Running - time analysis

➢ Compute-Prefix-Function (Π)

```
7    1  m ← length[p]          //'p' pattern to be
     matched
8    2  Π[1] ← 0
9    3  k ← 0
9     for q ← 2 to m
10         do while k > 0 and p[k+1] != p[q]
12   6                do k ← Π[k]
•              If p[k+1] = p[q]
•                then k ← k +1
•              Π[q] ← k
•    return Π
```

In the above pseudocode for computing the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step 1 to step 3 take constant time. Hence the running time of compute prefix function is Θ(m).

➢ KMP Matcher

```
1 n ← length[S]
2 m ← length[p]
3 Π ← Compute-Prefix-Function(p)
4 q ← 0
5 for i ← 1 to n
6    do while  q > 0 and p[q+1] != S[i]
7          do  q ← Π[q]
8      if p[q+1] = S[i]
9        then q ← q + 1
10     if q = m
11      then print "Pattern occurs with shift" i – m
12             q ← Π[ q]
```

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S'. Since step 1 to step 4 take constant time, the running time is dominated by this for loop. Thus running time of matching function is Θ(n).

# ITDO 5014 ADSA

## Module 5 : String Matching (7 Hrs )

1. Introduction
2. The naïve string matching algorithm
3. Rabin Karp algorithm
4. Boyer Moore algorithm
5. Knuth-Morris-Pratt algorithm (KMP)
6. Longest common subsequence(LCS)

Analysis of All algorithms and problem solving.

Self-learning Topics: Implementation of Robin Karp algorithm, KMP algorithm and LCS.

## 4. Boyer Moore algorithm :

**Like KMP and Finite Automata algorithms, Boyer Moore algorithm also preprocesses the pattern.**

**Boyer Moore is a combination of the following two approaches.**
**1) Bad Character Heuristic**
**2) Good Suffix Heuristic**

**Both of the above heuristics can also be used independently to search a pattern in a text.**

**Let us first understand how two independent approaches work together in the Boyer Moore algorithm.**

**If we take a look at the Naive algorithm, it slides the pattern over the text one by one.**
**KMP algorithm does preprocessing over the pattern so that the pattern can be shifted by more than one.**

**The Boyer Moore algorithm does preprocessing for the same reason.**

**It processes the pattern and creates different arrays for each of the two heuristics.**

**At every step, it slides the pattern by the max of the slides suggested by each of the two heuristics.**
**So it uses greatest offset suggested by the two heuristics at every step.**
**Unlike the previous pattern searching algorithms, the Boyer Moore algorithm starts matching from the last character of the pattern.**

## 1. Bad Character Heuristic :

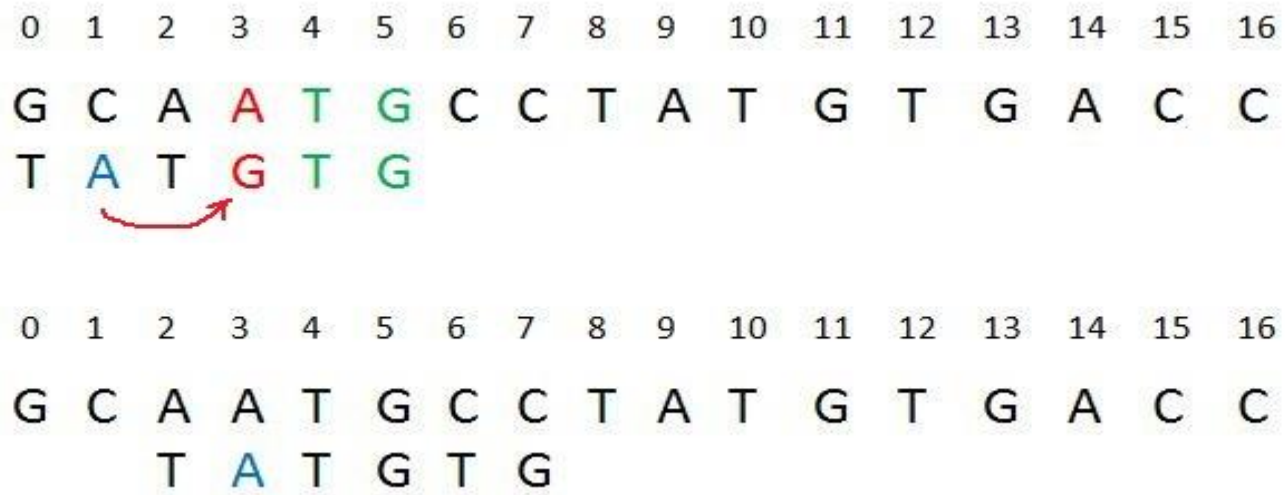
**The character of the text which doesn't match with the current character of the pattern is called the Bad Character.**

**Upon mismatch, we shift the pattern until –**

**1) The mismatch becomes a match**
**2) Pattern P moves past the mismatched character.**

### Case 1 – Mismatch become match

**We lookup the position of the last occurrence of the mismatched character in the pattern, and if the mismatched character exists in the pattern, then we'll shift the pattern such that it becomes aligned to the mismatched character in the text T.**



**In the above example, we got a mismatch at position 3.**
**Here our mismatching character is "A".**
**Now we will search for last occurrence of "A" in pattern.**

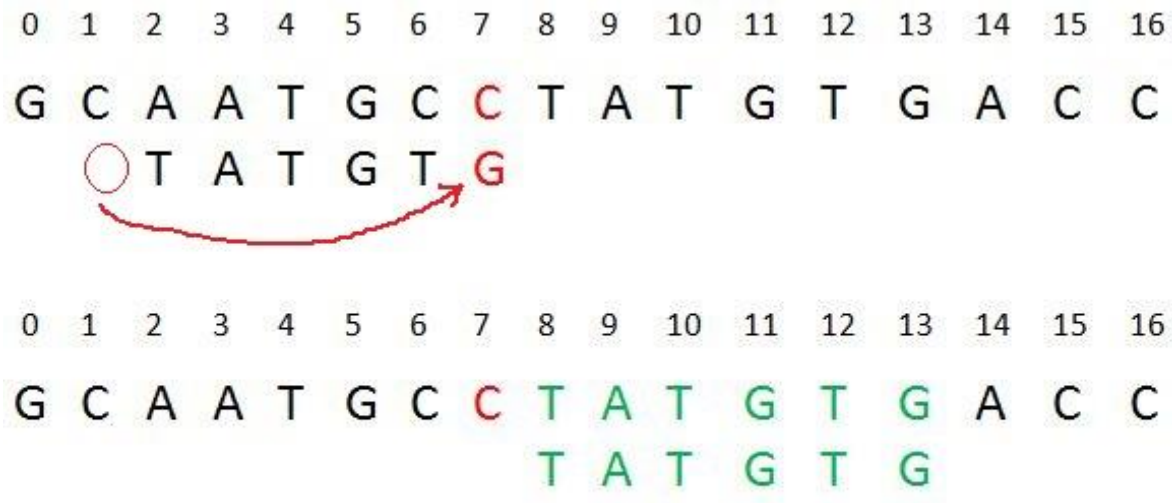
**We got "A" at position 1 in pattern (displayed in Blue) and this is the last occurrence of it.**

**Now we will shift pattern 2 times so that "A" in pattern get aligned with "A" in text.**

## Case 2 – Pattern move past the mismatch character :

**We'll lookup the position of last occurrence of mismatching character in pattern and if character does not exist we will shift pattern past the mismatching character.**

**Here we have a mismatch at position 7.**
**The mismatching character "C" does not exist in pattern before position 7 so we'll shift pattern past to the position 7 and eventually in above example, we have got a perfect match of pattern (displayed in Green).**
**We are doing this because "C" does not exist in the pattern so at every shift before position 7 we will get mismatch and our search will be fruitless.**



**The Bad Character Heuristic may take O(mn) time in worst case.**

**The worst case occurs when all characters of the text and pattern are same.**

**For example, txt[] = “AAAAAAAAAAAAAAAAAA” and pat[] = “AAAAA”.**

**The Bad Character Heuristic may take O(n/m) in the best case.**

**The best case occurs when all the characters of the text and pattern are different.**

# Good Suffix Heuristic :

**Just like bad character heuristic, a preprocessing table is generated for good suffix heuristic.**

**Let t be substring of text T which is matched with substring of pattern P.**

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| T | A | B | A | A | B | A | B | A | C | B | A |
| P | C | A | B | A | B | | | | | | |

**Now we shift pattern until :**
**1) Another occurrence of t in P matched with t in T.**
**2) A prefix of P, which matches with suffix of t**
**3) P moves past t**

## Case 1: Another occurrence of t in P matched with t in T

**Pattern P might contain few more occurrences of t.**
**In such case, we will try to shift the pattern to align that occurrence with t in text T.**
**For example-**

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| T | A | B | A | A | B | A | B | A | C | B | A |
| P | | | C | A | B | A | B | | | | |

Figure – Case 1

**Explanation:**
**In the above example, we have got a substring t of text T matched with pattern P (in green) before mismatch at index 2.**

**Now we will search for occurrence of t (“AB”) in P.**

**We have found an occurrence starting at position 1 (in yellow background) so we will right shift the pattern 2 times to align t in P with t in T.**

**This is weak rule of original Boyer Moore and not much effective, we will discuss a Strong Good Suffix rule shortly.**

## Case 2: A prefix of P, which matches with suffix of t in T

**It is not always likely that we will find the occurrence of t in P.**

**Sometimes there is no occurrence at all, in such cases sometimes we can search for some suffix of t matching with some prefix of P and try to align them by shifting P.**
**For example –**

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| T | A | A | B | A | B | A | B | A | C | B | A |
| P | A | B | B | A | B | | | | | | |

**Explanation:**
**In above example,**
**we have got t (“BAB”) matched with P (in green) at index 2-4 before mismatch .**

**But because there exists no occurrence of t in P we will search for some prefix of P which matches with some suffix of t.**

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| T | A | A | B | A | B | A | B | A | C | B | A |
| P | | | | A | B | B | A | B | | | |

**Figure** – Case 2

**We have found prefix “AB” (in the yellow background) starting at index 0 which matches not with whole t but the suffix of t “AB” starting at index 3. So now we will shift pattern 3 times to align prefix with the suffix.**

**Case 3: P moves past t**

**If the above two cases are not satisfied, we will shift the pattern past the t. For example –**

**Explanation:**

**If above example, there exist no occurrence of t (“AB”) in P and also there is no prefix in P which matches with the suffix of t.**

**So, in that case, we can never find any perfect match before index 4, so we will shift the P past the t ie. to index 5.**

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| T | A | A | C | A | B | A | B | A | C | B | A |
| P | C | B | A | A | B | | | | | | |

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| T | A | B | A | A | B | A | B | A | C | B | A |
| P | | | | | | C | B | A | A | B | |

**Figure** – Case 3

## Strong Good suffix Heuristic :

Suppose substring **q = P[i to n]** got matched with **t** in T and **c = P[i-1]** is the mismatching character. Now unlike case 1 we will search for t in P which is not preceded by character **c**. The closest such occurrence is then aligned with t in T by shifting pattern P. For example –

**Explanation:** In above example, **q = P[7 to 8]** got matched with t in T. The mismatching character **c** is "C" at position P[6]. Now if we start searching t in P we will get the first occurrence of t starting at position 4. But this occurrence is preceded by "C" which is equal to c, so we will skip this and carry on searching. At position 1 we got another occurrence of t (in the yellow background). This occurrence is preceded by "A" (in blue) which is not equivalent to c. So we will shift pattern P 6 times to align this occurrence with t in T.We are doing this because we already know that character **c = "C"** causes the mismatch. So any occurrence of t preceded by c will again cause mismatch when aligned with t, so that's why it is better to skip this.

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T | A | A | B | A | B | A | B | A | C | B | A | C | A | B | B | C | A | B |
| P | A | A | C | C | A | C | C | A | C | | | | | | | | | |

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T | A | A | B | A | B | A | B | A | C | B | A | C | A | B | B | C | A | B |
| P | | | | | | | A | A | C | C | A | C | C | A | C | | | |

**Figure** – strong suffix rule

**Explanation:** In above example, **q = P[7 to 8]** got matched with t in T. The mismatching character **c** is “C” at position P[6]. Now if we start searching t in P we will get the first occurrence of t starting at position 4. But this occurrence is preceded by “C” which is equal to c, so we will skip this and carry on searching. At position 1 we got another occurrence of t (in the yellow background). This occurrence is preceded by “A” (in blue) which is not equivalent to c. So we will shift pattern P 6 times to align this occurrence with t in T.We are doing this because we already know that character **c = “C”** causes the mismatch. So any occurrence of t preceded by c will again cause mismatch when aligned with t, so that’s why it is better to skip this.

**Preprocessing for Good suffix heuristic**

As a part of preprocessing, an array **shift** is created. Each entry **shift[i]** contain the distance pattern will shift if mismatch occur at position **i-1**. That is, the suffix of pattern starting at position **i** is matched and a mismatch occur at position **i-1**. Preprocessing is done separately for strong good suffix and case 2 discussed above.

**1) Preprocessing for Strong Good Suffix**

Before discussing preprocessing, let us first discuss the idea of border. A **border** is a substring which is both proper suffix and proper prefix. For example, in string **“ccacc”**, **“c”** is a border, **“cc”** is a border because it appears in both end of string but **“cca”** is not a border.

As a part of preprocessing an array **bpos** (border position) is calculated. Each entry **bpos[i]** contains the starting index of border for suffix starting at index i in given pattern P.

The suffix **ϕ** beginning at position m has no border, so **bpos[m]** is set to **m+1** where **m** is the length of the pattern.

The shift position is obtained by the borders which cannot be extended to the left.

---

# ITDO 5014 ADSA

## Module 5 : String Matching (7 Hrs )

1. Introduction
2. The naïve string matching algorithm
3. Rabin Karp algorithm
4. Boyer Moore algorithm
5. Knuth-Morris-Pratt algorithm (KMP)
6. Longest common subsequence(LCS)

Analysis of All algorithms and problem solving.

**Self-learning Topics: Implementation of Robin Karp algorithm, KMP algorithm and LCS.**

# 4.Longest Common Subsequence (LCS)

- **LCS: The longest common subsequence (LCS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences).**
- **Subsequences are not required to occupy consecutive positions within the original sequences.**

**Given two sequences X and Y, a sequence Z is said to be a common subsequence of X and Y, if Z is a subsequence of both X and Y. For example, if**

**X={ A,C,B,D,E,G,C,E,D,B,G} and**

**Y={B,E,G,C,F,E,U,B,K }**

**then a common subsequences of X and Y could be**

**Z={ B,E,E,B}**

**Z = { B,E,G,C,E,B }**

---

**The longest common subsequence of X and Y is Z = {B,E,G,C,E,B } .**

- **Consider two strings:**

**str1 = “abcdaf”**

**str2 =“acbcf”**

**Now here we can say that :**

**longest common subsequence will be : abcf**

**and its length = 4**

- **If we use divide and conquer approach to solve LCS problem then:**


- **So to solve this problem programmatically we use dynamic programming technique.**

- By using dynamic programming technique for LCS, arrange both the strings in following manner:

| str1 → | | A | B | C | D | A | F |
|---|---|---|---|---|---|---|---|
| str2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | | | | | | |
| c | 0 | | | | | | |
| b | 0 | | | | | | |
| c | 0 | | | | | | |
| f | 0 | | | | | | |

# Dynamic Programming

Approach:

- If the last characters match:
  - LCS[i][j] = LCS[i-1][j-1] + 1
- If the last characters don't match:
  - LCS[i][j] = max( LCS[i-1][j] , LCS[i][j-1] )

**Dynamic programming approach for LCS:**

**if X[i] = Y[j]  ( If last characters match i.e. )**

**LCS[i][j] := LCS[i-1][j-1] + 1**

**else  (If last characters don’t match i.e. )**

**LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])**

- Step 1: Consider only 1st character from both the strings, and find its LCS. Str1 = **A B C D A F  Str2 =  A C B C F**

| j=Str1 | | A | | | | | |
|---|---|---|---|---|---|---|---|
| i=Str2 | 0 | 0 | | | | | |
| A | 0 | 0+1 1 | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

**if X[i] = Y[j]  ( If last characters match i.e. )**

**LCS[i][j] := LCS[i-1][j-1] + 1**

**else  (If last characters don't match i.e. )**

**LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])**

- Now consider 2 characters from str1 and single character from str2.

| Str1 | j → | A | B | | | | |
|---|---|---|---|---|---|---|---|
| i str2 | 0 | 0 | 0 | | | | |
| A | 0 | 1 | 1 (max) | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

| Str1 | | A | B | C | | | |
|---|---|---|---|---|---|---|---|
| str2 | 0 | 0 | 0 | 0 | | | |
| A | 0 | 1 | 1 (max) | 1 | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

| Str1 | | A | B | C | D | A | F |
|---|---|---|---|---|---|---|---|
| str2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

**if X[i] = Y[j]  ( If last characters match i.e. )**

**LCS[i][j] := LCS[i-1][j-1] + 1**

**else  (If last characters don't match i.e. )**

**LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])**

**if X[i] = Y[j]  LCS[i][j] := LCS[i-1][j-1] + 1  else  LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])**

| Str1 | | A | B | C | D | A | F |
|---|---|---|---|---|---|---|---|
| str2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 1 | 2 | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

0+1

| Str1 | | A | B | C | D | A | F |
|---|---|---|---|---|---|---|---|
| str2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 1 | 2 | 2 | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

max

| Str1 | | A | B | C | D | A | F |
|---|---|---|---|---|---|---|---|
| str2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| B | 0 | 1 | | | | | |
| | | | | | | | |
| | | | | | | | |

max

| Str1 | | A | B | C | D | A | F |
|---|---|---|---|---|---|---|---|
| str2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| B | 0 | 1 | 2 | | | | |
| | | | | | | | |
| | | | | | | | |

1+1

| Str1 | | A | B | C | D | A | F |
|---|---|---|---|---|---|---|---|
| str2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| B | 0 | 1 | 2 | 0+1 2 | 2 | 2 | 2 |
| C | 0 | 1 | 2 | 3 | | | |
| | | | | | | | |

| Str1 | | A | B | C | D | A | F |
|---|---|---|---|---|---|---|---|
| str2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| B | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| C | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| | | | | | | | |

| Str1 | | A | B | C | D | A | F |
|---|---|---|---|---|---|---|---|
| str2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| B | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| C | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| F | 0 | 1 | 2 | 3 | 3 | 3 | 0+1 4 |

if X[i] = Y[j] ( If last characters match i.e. )

LCS[i][j] := LCS[i-1][j-1] + 1

else (If last characters don't match i.e. )

LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])

**if X[i] = Y[j]  LCS[i][j] := LCS[i-1][j-1] + 1  else  LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])**

# Final Matrix:

| Str1 | | A | B | C | D | A | F |
|---|---|---|---|---|---|---|---|
| str2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| B | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| C | 0 | 1 | 2 | 3 | 3 | 3 | 3 |
| F | 0 | 1 | 2 | 3 | 3 | 3 | 4 |

Total length = 4

F, C, B, A

←

Longest Common sequence:
A,B,C,F

# Algorithm

```
function LCSLength(X[1..m], Y[1..n])
    C = array(0..m, 0..n)
    for i := 0..m
       C[i][0] = 0
    for j := 0..n
       C[0][j] = 0
    for i := 1..m
        for j := 1..n
            if X[i] = Y[j]
                C[i][j] := C[i-1][j-1] + 1
            else
                C[i][j] := max(C[i][j-1], C[i-1][j])
    return C[m][n]
```

# Complexity

- Time Complexity for LCS using dynamic programming is:

*T(n) = m(loop1) + n(loop2) + (m*n)(loop3)*

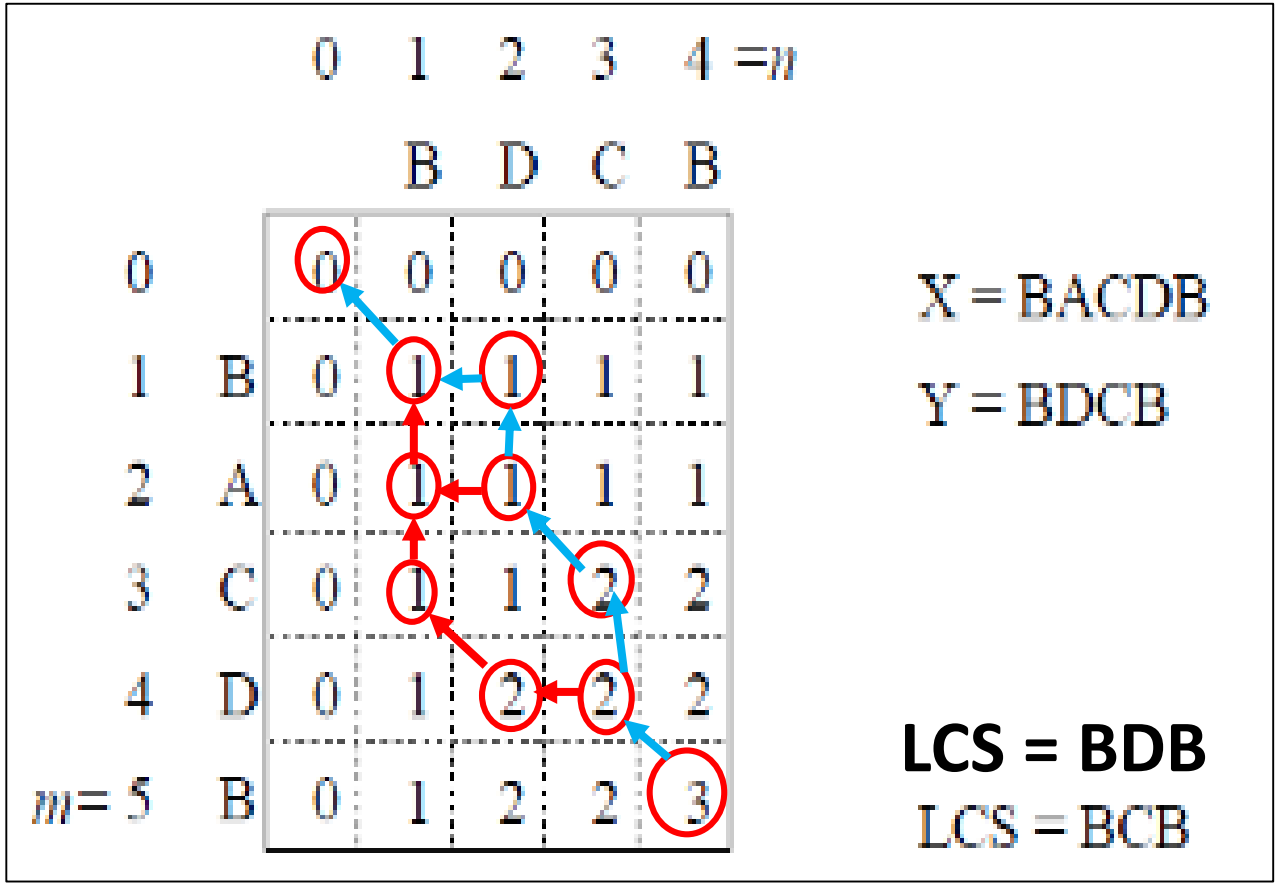
*T(n) = m + n + (mn)*

*T(n) = O(mn)*

**if X[i] = Y[j]  LCS[i][j] := LCS[i-1][j-1] + 1  else  LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])**

Solve Following Example:

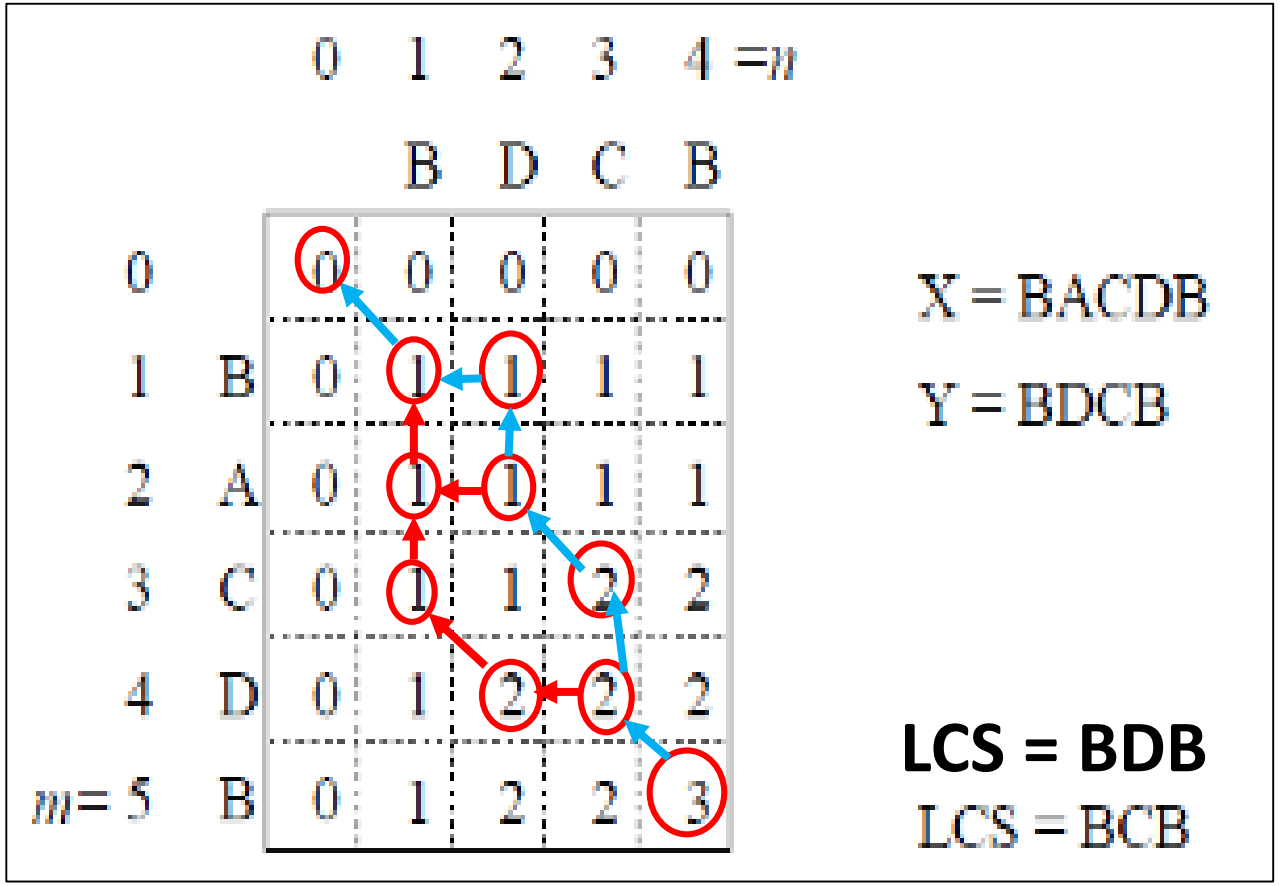
1. X=BACDB, Y=BDCB



**if X[i] = Y[j]  LCS[i][j] := LCS[i-1][j-1] + 1  else  LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])**

Solve Following Example:

1. X=BACDB, Y=BDCB

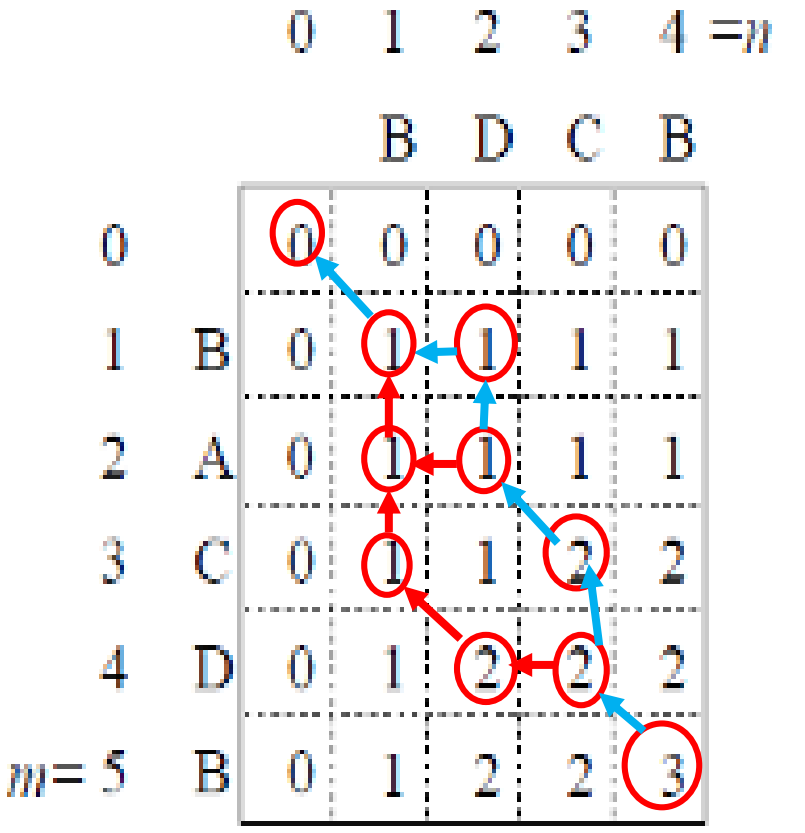


**if X[i] = Y[j]  LCS[i][j] := LCS[i-1][j-1] + 1  else  LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])**

Solve Following Example:

1. X=BACDB, Y=BDCB

2. Str1=bcacbcab, Str2= bccabcc




X = BACDB

Y = BDCB

**LCS = BDB**

LCS = BCB

String 1 = bcacbcab; String 2 = bccabcc

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

Here Maximum Common Subsequence between String1 and String2 is of length 5.

# ITDO 5014 ADSA

## Module 5 : String Matching (7 Hrs )

1. Introduction
2. The naïve string matching algorithm
3. Rabin Karp algorithm
4. Boyer Moore algorithm
5. Knuth-Morris-Pratt algorithm (KMP)
6. Longest common subsequence(LCS)

Analysis of All algorithms and problem solving.

**Self-learning Topics: Implementation of Robin Karp algorithm, KMP algorithm and LCS.**

# Thank You