Analysis of Algorithms

Semester IV EXCP

Course Code
116U40C403
January – April 2024

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

String Matching Algorithms

Arati Phadke
SY EXCP Div A
2023-24

Table 18.1 String-matching algorithms

Name of the algorithm	Complexity
Naïve string matching	$O(m(n-m+1))$
Rabin–Karp	$O(m(n-m))$ in the average case and $O(n+m)$ in the best case
Finite automata	$O(n)$
Knuth–Morris–Pratt	$O(n)$

String Matching Algorithms

- Naïve or Brute Force Search
- Knuth-Morris-Pratt Algorithm
- Longest Common subsequence
- Finite Automata Search

Introduction

- String matching or searching algorithms try to find places where one or several strings (also called patterns) are found within a larger string (searched text):
- Text:

  ... try to find places where one or several strings

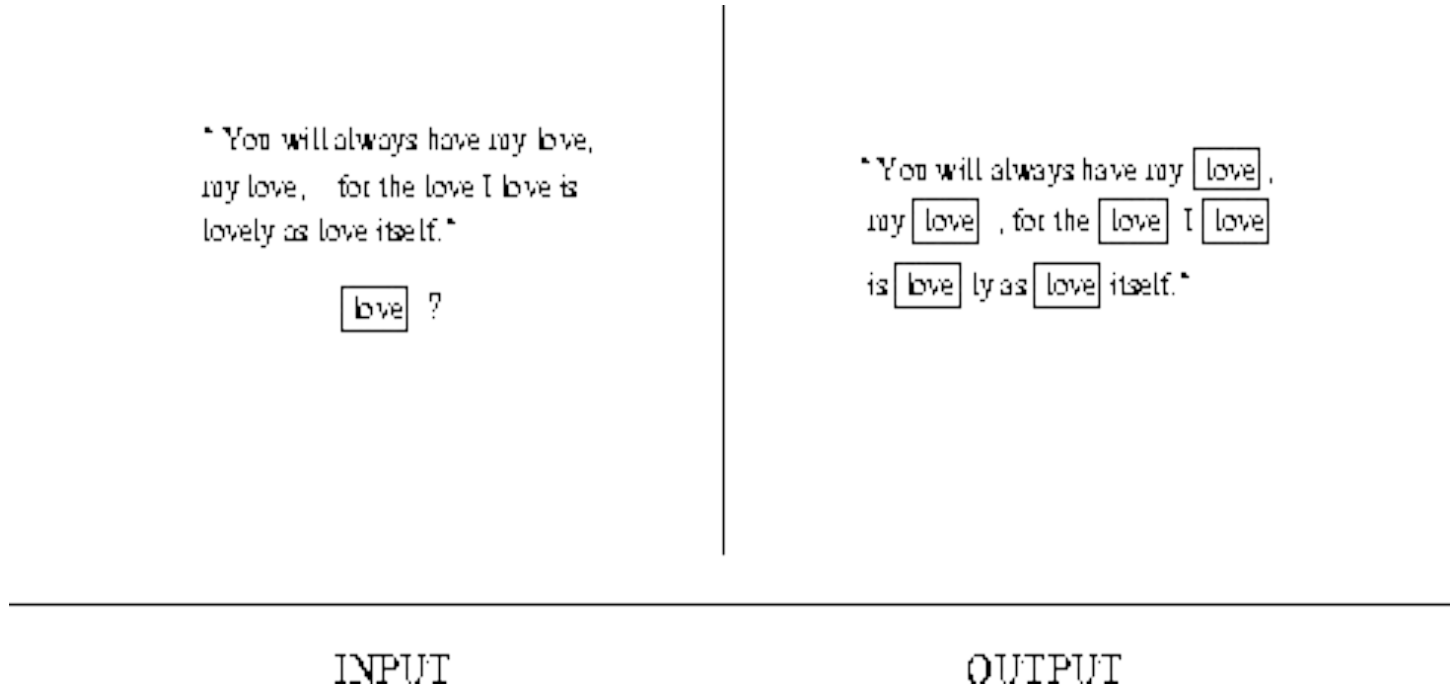
  Pattern: ace

  ... try to find places where one or several strings

APPLICATION

- Finding all occurrences of a pattern in a text is a problem that arises frequently in text-editing programs.
- To search for particular patterns in DNA sequences.

- Input description: A text string t of length n. A pattern string p of length m.
- Problem description: Find the first (or all) instances of the pattern in the text.



Naïve string matching Algorithm

Naïve string matching Algorithm

- String matching problem: Given a text string *t* and a pattern string *p*, find all occurrences of *p* in *t*
- A naive algorithm for this problem simply considers all possible starting positions *i* of a matching string within *t*, and compares *p* to the substring of *t* beginning at each such position *I*

Naive algorithm for Pattern Searching (Brute-Force Algorithm)

- The brute force algorithm consists in checking, at all positions in the text between 0 and *n*-*m*, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right.
- **E xample** (1)

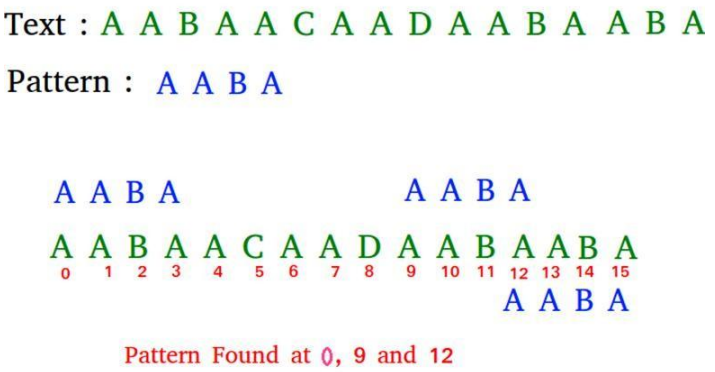
Input: txt[] = "AABAACAADAABAABA"

pat[] = "AABA"

Output: Pattern found at index 0

Pattern found at index 9

Pattern found at index 12



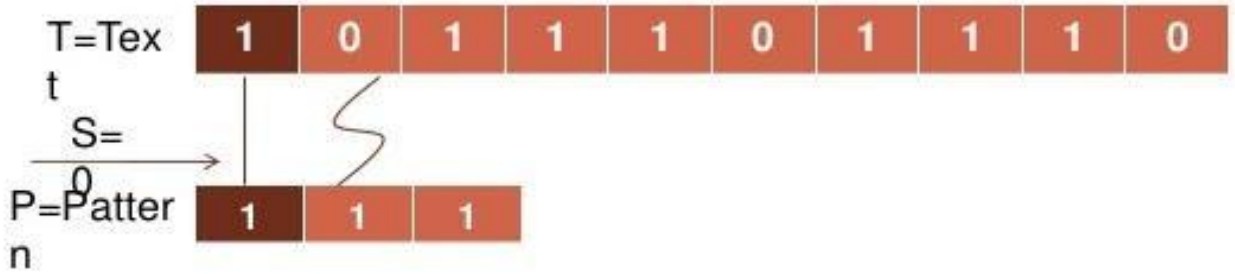
Example (2)

- SUPPOSE,

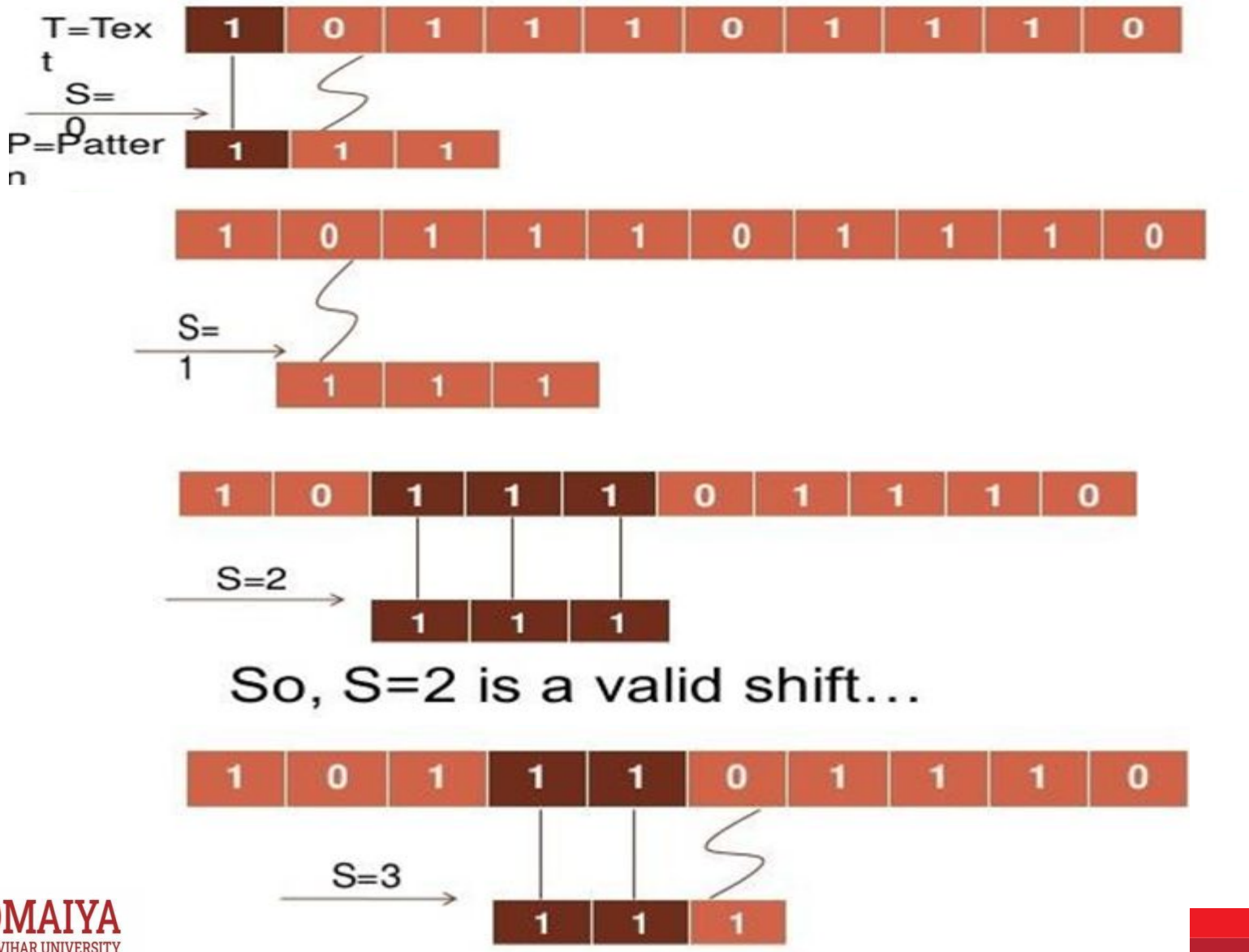
  T=1011101110

  P=111

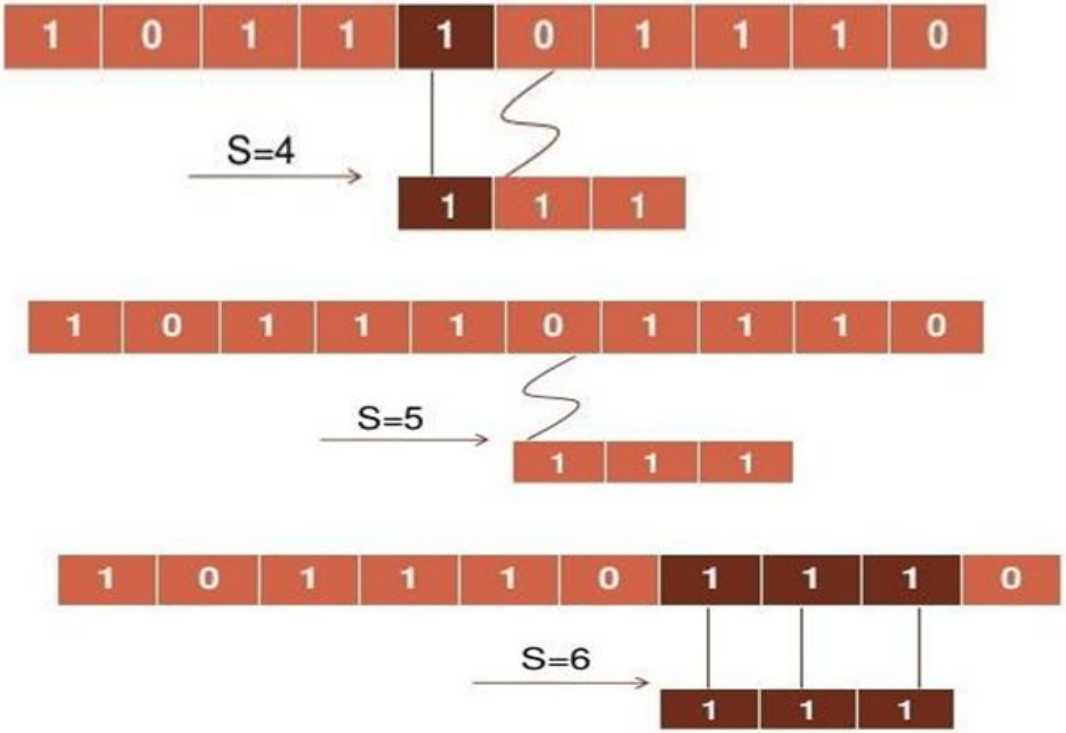
  FIND ALL VALID SHIFT……



Example (2)


T=Text
S=0
P=Pattern
1 0 1 1 1 0 1 1 1 0
1 1 1
1 0 1 1 1 0 1 1 1 0
S=1
1 1 1
1 0 1 1 1 0 1 1 1 0
S=2
1 1 1
So, S=2 is a valid shift…
1 0 1 1 1 0 1 1 1 0
S=3
1 1 1

Example (2)



So, S=6 is a valid shift...

Main Features

- No preprocessing phase;
- Constant extra space needed;
- Always shifts the window by exactly 1 position to the right;
- Comparisons can be done in any order;
- Searching phase in O(mn) time complexity;
- 2n expected text characters comparisons.

Algorithm

NAIVE-STRING-MATCHER (T, P)

1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n -m
4. do if P [1.....m] = T [s + 1....s + m]
5. then print "Pattern occurs with shift" s

The naive algorithm finds all valid shifts using a loop that checks the condition P[1....m]=T[s+1.... s+m] for each of the n- m+1 possible values of s.

(P=pattern , T=text/string , s=shift)

What is the 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).

What is the 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[] = "AAAAAAAAAAAAAAAAA"; pat[] = "AAAAA";

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

txt[] = "AAAAAAAAAAAAAAAAB"; 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).

String Matching with Finite Automata

Finite automata

- A finite state machine (FSM, also known as a deterministic finite automaton or DFA) is a way of representing a *language*
(meaning a set of strings; we're interested in representing the
set strings matching some pattern).
  - A finite automaton is a quintuple (Q, Σ, δ, q0 , F):
  - **Q**: the finite **set of states**
  - **Σ**: the finite **input alphabet**
  - **δ**: the "transition function" from QxΣ to Q
  - **q0** ∈ Q: the start state
  - **F** ⊂ Q: the set of final (accepting) states

- The finite automaton begins in state q0 and read the characters of its input string one at a time. If the automaton is in state q and reads input character a, it moves from state q to state d(q,a).

- As long as M is in a state belonging to A, M is said to have accepted the string read so far, an input that is not accepted is said to be rejected.

FINITE AUTOMATA

- Finite Automata(FA) is the simplest machine to recognize patterns. FA is characterized into two types:

1) **Deterministic Finite Automata (DFA)**
2) **Nondeterministic Finite Automata(NFA)**

- A finite automaton accepts strings in a specific language. It begins in state q0 and reads characters one at a time from the input string. It makes transitions (φ) based on these characters, and if when it reaches the end of the tape it is in one of the accept states, that string is accepted by the language.
- It has **finite number of states**
- Digraphs used to represent DFA is known as **state diagram**.
- It is useful for doing **lexical analysis** and **pattern matching.**
- ***There can be many possible DFAs for a pattern***. A DFA with minimum number of states is generally preferred.

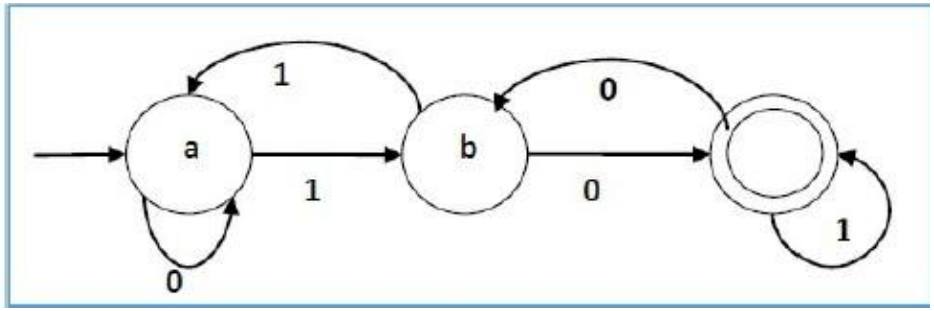
EXAMPLE

- Let a deterministic finite automata be
- Q = {a, b, c}, Σ = {0, 1}, q_0 = {a}, F = {c}, and

Transition function δ

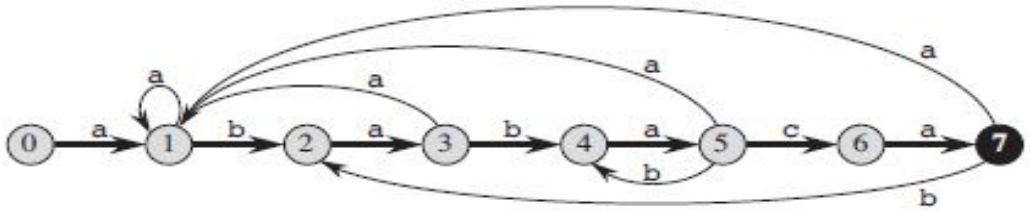
Present State	Next State for Input 0	Next State for Input 1
a	a	b
b	c	a
c	b	c

State Diagram



Example (2)

- The string-matching automaton is very efficient: it examines each character in the text exactly once and reports all the valid shifts in O(n) time.



(a)

state	input a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

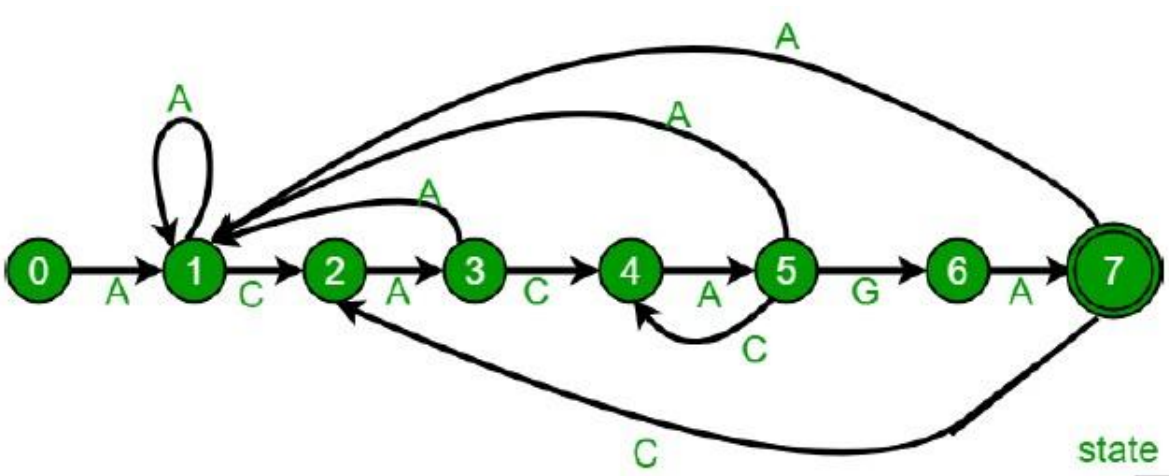
i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

- A state-transition diagram (previous slide) for the string-matching automaton that accepts all strings ending in the string **ababaca.**
- State 0 is the start state, and state 7 (shown blackened) is the only accepting state. A directed edge from state i to state j labeled a represents (i,a) = j.
- The right-going edges forming the "spine" of the automaton, shown heavy in the figure, correspond to successful matches between pattern and input characters.
- The left-going edges correspond to failing matches. Some edges corresponding to failing matches are not shown; by convention, if a state i has no outgoing edge labeled a for some a , then (i,a) = 0. (b) The corresponding transition function , and the pattern string P = ababaca.

- The entries corresponding to successful matches between pattern and input characters are shown shaded. (c) The operation of the automaton on the text T = abababacaba.
- Under each text character T[i] is given the state ø(Ti) the automaton is in after processing the prefix Ti. One occurrence of the pattern is found, ending in position 9.

Example (3)



Pattern P=acacaga

	character			
state	A	C	G	T
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

BASIC IDEA

The basic idea is to build a automaton in which

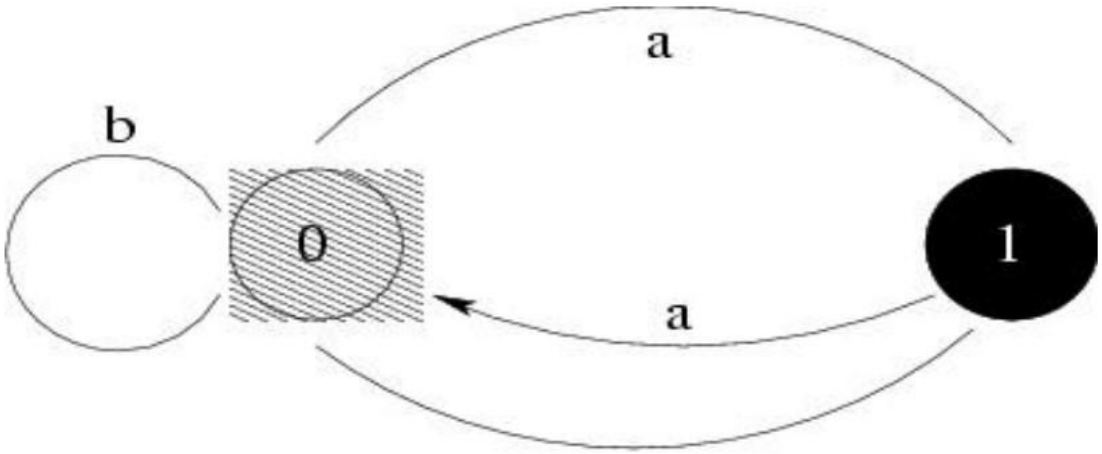
- Each character in the pattern has a state.
- Each match sends the automaton into a new state.
- If all the characters in the pattern has been matched, the automaton enters the accepting state.
- Otherwise, the automaton will return to a suitable state according to the current state and the input character such that this returned state reflects the maximum advantage we can take from the previous matching.
- The matching takes O(n) time since each character is examined once.

BASIC IDEA:

- To search a pattern P = p1p2...pm in a text T =t1t2...tn. Number of states in FA will be M+1where M is length of the pattern.
- State 0 will be the starting state, and state m will be the only accepting state.
- Get the next state from the current state for every possible character.
- We can get the next state by getting the length of the longest prefix of the given pattern such that the prefix is also suffix . The transition function chooses the next state to maintain the invariant: **φ(Ti) = σ(Ti)** After scanning in i characters, the state number is the longest prefix of P that is also a suffix of Ti.

The Suffix Function

- In order to properly search for the string, the program must define a **suffix function (σ)** which checks to see how much of what it is reading matches the search string at any given moment.
- A suffix function w.r.t. pattern P[1..m], s , is a mapping from S * to {0,1,...,m} **such that s(x) is the length of the longest prefix of P that is a suffix of x**:
- Let x = abcab.
- The prefixes of x are a, ab, abc, abca.
- The suffixes of x are b, ab, cab, bcab.
- The **σ** of x is ab ie 2



State	input a	b
0	1	0
1	0	0

Given pattern: a^{2k+1}
Input string = abaaa
Start state: 0
Terminate state: 1

Figure 1: An automaton.

The construction of the string matching automaton is based on the given pattern. The time of this construction may be O(m3 |S|)

Computing the transition function.

COMPUTE-TRANSITION-FUNCTION (P, ∑)

m ← length [P]

2. for q ← 0 to m // cycles through all the states ie pattern
3. do for each character a ∈ ∑* // cycles through the input alphabets

4.do k ← min (m+1, q+2) //set δ(q,a) to be the largest k

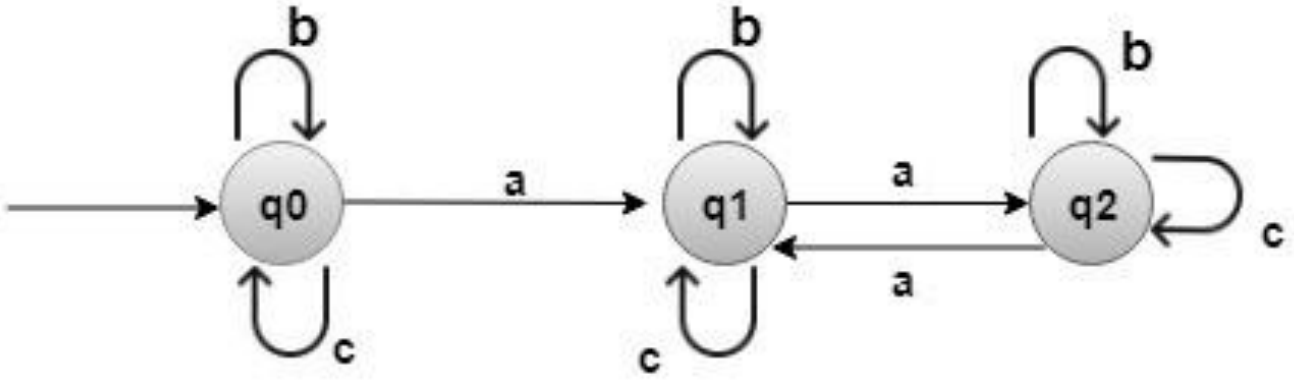
Pqa.

5. repeat k←k-1
6. Until
7. Pk ⊃ Pqa
8. δ(q,a)←k
9. Return δ

Running Time Complexity

- The time complexity of the computing Transition
- Function is O(m^3*NO_OF_CHARS) where m is length of the pattern and NO_OF_CHARS is size of alphabet (total number of possible characters in pattern and text). The implementation tries all possible prefixes starting from the longest possible that can be a suffix of “pat[0..k-1]x”. O(m3 |Σ|)
- Outer loop: m |Σ|
- Inner loop: runs at most m+1
- Pk ⊃ Pqa: requires up to m comparisons

Example: Suppose a finite automaton which accepts even number of a's where $\sum = \{a, b, c\}$



Solution:

q0 is the initial state.

q_2 is the accepting or final state, and transition function δ is defined as

$\delta(q_0, a) = q_1$ $\quad\quad$ $\delta(q_1, c) = q_1$

$\delta(q_0, b) = q_0$ $\quad\quad$ $\delta(q_2, a) = q_1$

$\delta(q_0, c) = q_0$ $\quad\quad$ $\delta(q_2, b) = q_2$

$\delta(q_1, a) = q_2$ $\quad\quad$ $\delta(q_2, c) = q_2$

$\delta(q_1, b) = q_1$ $\quad\quad$ $\delta(q_1, c) = q_1$

Suppose w is a string such as

w = b c a a b c a a a b a c, then $\delta(q_0,$ b c a a a b c a a a b a c)

$= \delta(q_0,$ c a a b c a a a b a c)

$= \delta\ (q_0,$ a a b c a a a b a c) $= \delta(q_1,$ a b c a a a b a c)

$= \delta(q_2,$ b c a a a b a c) $= \delta(q_2,$ c a a a b a c)

$= \delta(q_2,$ a a a b a c) $= \delta(q_1,$ a a b a c)

$= \delta(q_2, a, b, a, c) = \delta(q_1,$ b a c)

$= \delta(q_1, b, c) = \delta(q_2, c) = q_2$(Accepting State)

Thus, given finite automation accepts w.

The Knuth-Morris-Pratt (KMP) Algorithm

The Knuth-Morris-Pratt (KMP)Algorithm

- Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem.
- A matching time of O (n) is achieved by avoiding comparison with an element 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:

1. **The Prefix Function (Π):** The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'

2. **The KMP Matcher:** With string 'S,' pattern 'p' and prefix function 'Π' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

PREFIX- FUNCTION (P)

- Following pseudo code compute the prefix function, Π:

COMPUTE- PREFIX- FUNCTION (P)

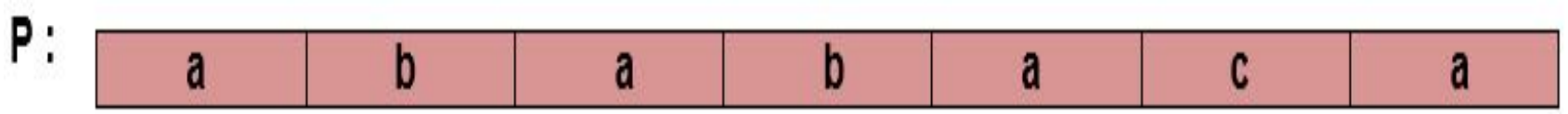
1. m ←length [P] //'p' pattern to be matched
2. Π [1] ← 0
3. k ← 0
4. for q ← 2 to m
5. do while k > 0 and P [k + 1] ≠ P [q]
6. do k ← Π [k]
7. If P [k + 1] = P [q]
8. then k← k + 1
9. Π [q] ← k
10. Return Π

```
/ Function to compute the Prefix Function using an O(N^3) approach
vector<int> prefix_function(string s) {
int n = s.size();
vector<int> pi(n);

// Iterate through each position in the string
for (int i = 0; i < n; i++) {
   pi[i] = 0; // Initialize the value at the current position

   // Try all possible lengths for the prefix/suffix
   for (int j = 0; j < i; j++) {

   // Check if the substrings are equal
   if (s.substr(0, j + 1) == s.substr(i - j, j + 1)) {
      pi[i] = j + 1; // If equal, update the value at the current position
   }
   }
}

// Return the computed Prefix Function
return pi;
  }
```

Running Time Analysis:

- In the above pseudo code for calculating the prefix function, the for loop from step 4 to step 10 runs 'm' times.
- Step1 to Step3 take constant time. Hence the running time of computing prefix function is O (m).

- **Example:** Compute Π for the pattern 'p' below:



- Solution
- 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				

Step3: 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			

Step4: 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		

Step5: q = 6, k = 3

Π [6] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	

Step6: 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	0	1

- After iteration 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	0	1

The KMP Matcher

1. The KMP Matcher with the pattern 'p,' the string 'S' and prefix function 'Π' as input, finds a match of p in S. Following pseudo code compute the matching component of KMP algorithm:

KMP-MATCHER (T, P)

```
1.  n ← length [T]
2.  m ← length [P]
3.  Π← COMPUTE-PREFIX-FUNCTION (P)
4.  q ← 0                          // numbers of characters matched
5.  for i ← 1 to n                 // scan S from left to right
6.     do while q > 0 and P [q + 1] ≠ T [i]
7.     do q ← Π [q]
8.  If P [q + 1] = T [i]           // next character does not match
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
```

Running Time Analysis

- 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 times, the running time is dominated by this for the loop. Thus running time of the matching function is **O (n).**
- Example: Given a string 'T' and pattern 'P' as follows:

- Example (2)
- Let us execute the KMP Algorithm to find whether 'P' occurs in 'T.'

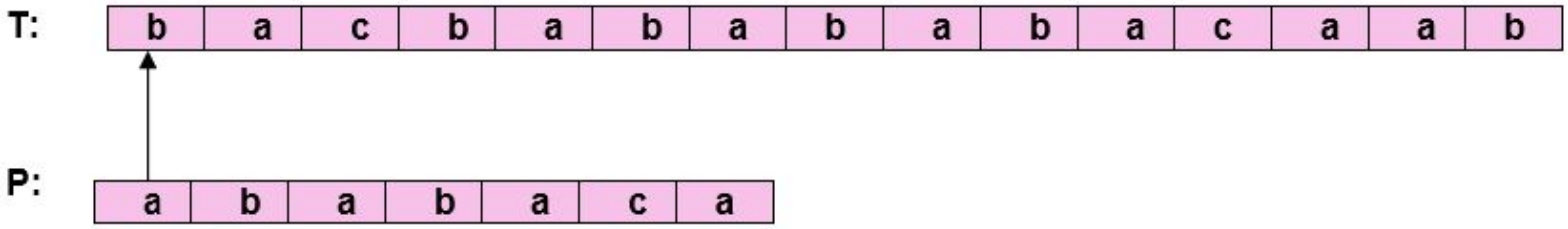
q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

- For 'p' the prefix function, ? was computed previously and is as follows:

Solution:
Initially: n = size of T = 15 m = size of P = 7

Step1: i=1, q=0

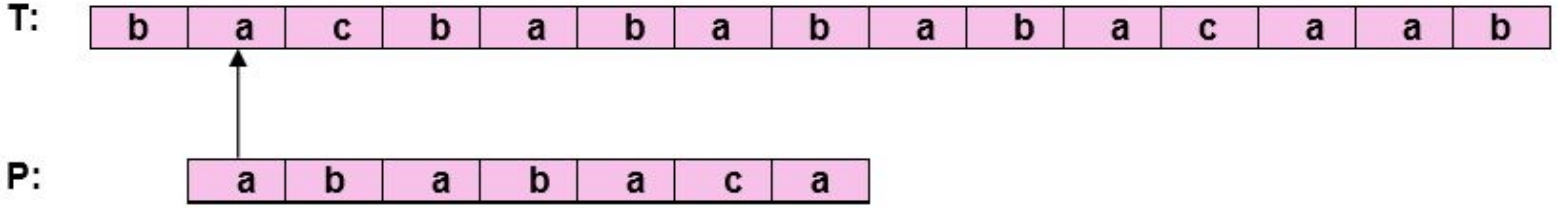
Comparing P [1] with T [1]



P [1] does not match with T [1]. 'p' will be shifted one position to the right.

Step2: i = 2, q = 0

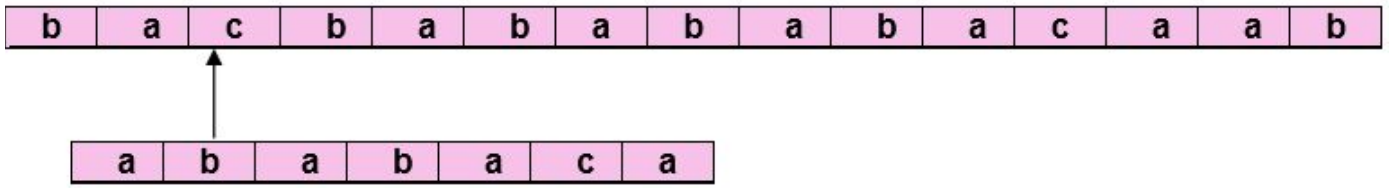
Comparing P [1] with T [2]



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

3: i = 3, q = 1

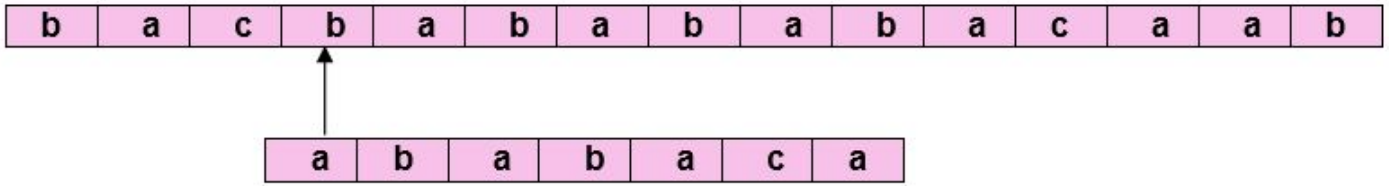
Comparing P [2] with T [3] P [2] doesn't match with T [3]



tracking on p, Comparing P [1] and T [3]

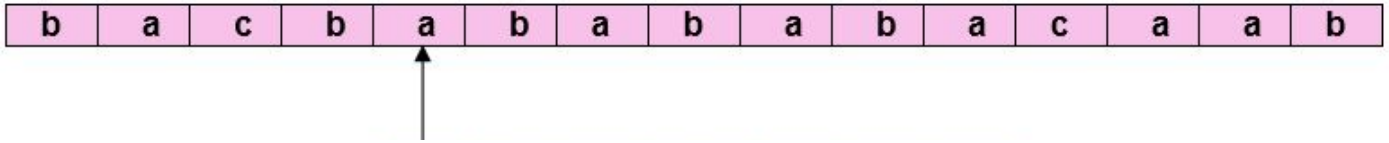
4: i = 4, q = 0

Comparing P [1] with T [4] P [1] doesn't match with T [4]



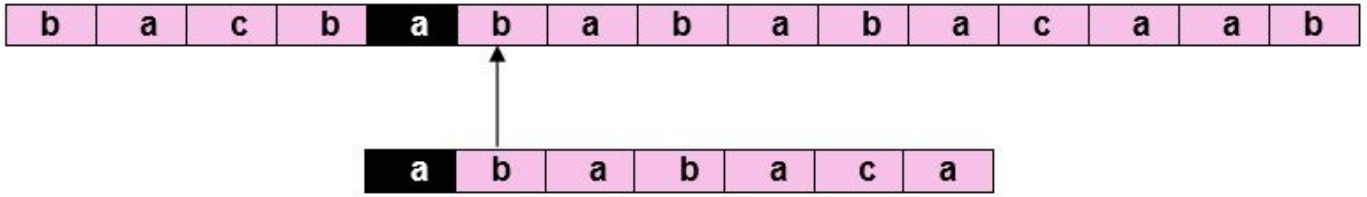
5: i = 5, q = 0

Comparing P [1] with T [5] P [1] match with T [5]



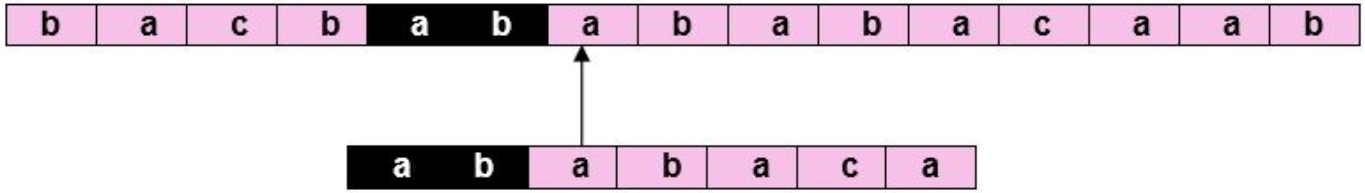
6: i = 6, q = 1

Comparing P [2] with T [6] P [2] matches with T [6]



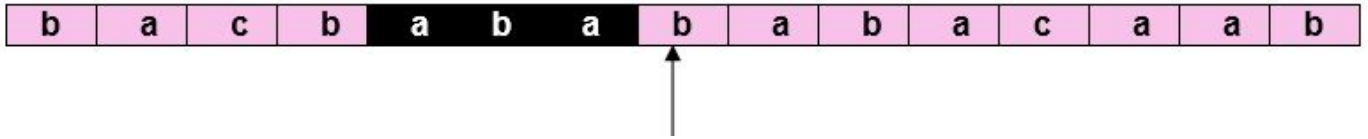
7: i = 7, q = 2

Comparing P [3] with T [7] P [3] matches with T [7]



8: i = 8, q =3

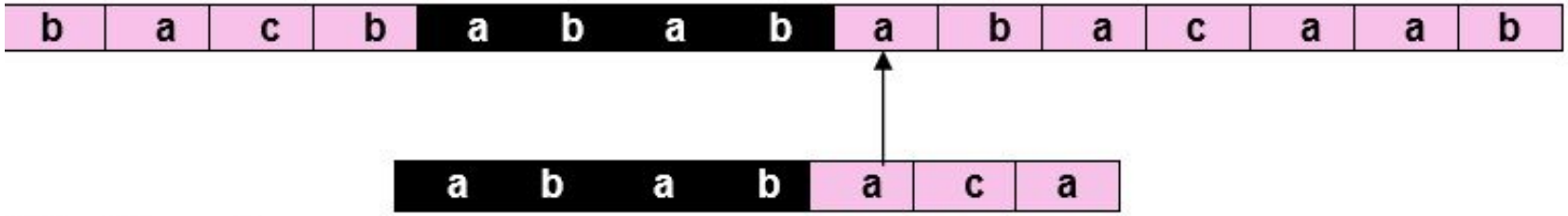
Comparing P [4] with T [8] P [4] matches with T [8]



: i = 9, q = 4

Comparing P [5] with T [9]

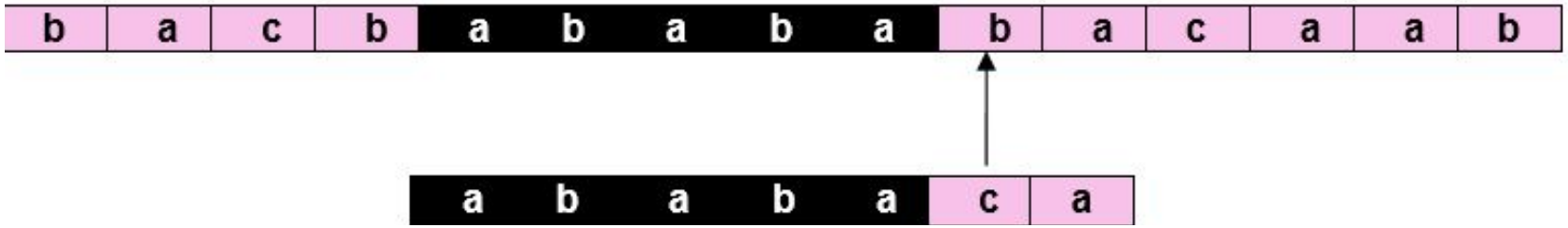
P [5] matches with T [9]



0: i = 10, q = 5

Comparing P [6] with T [10]

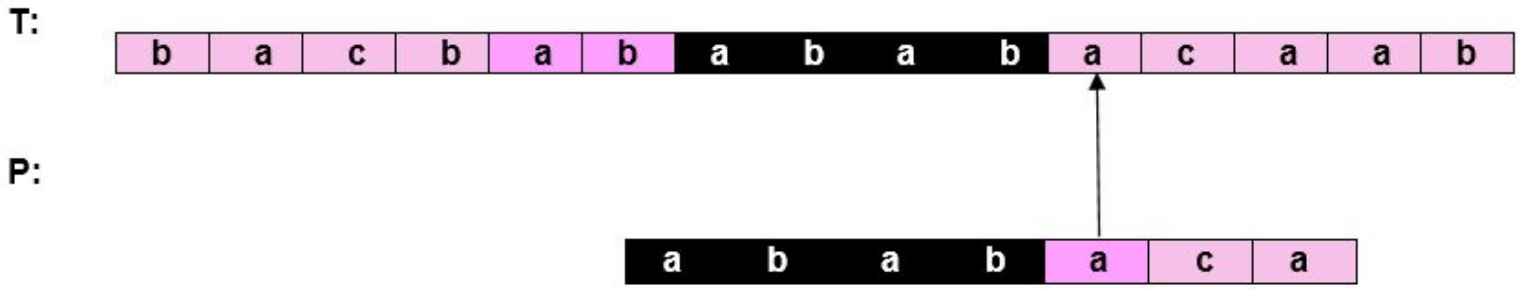
P [6] doesn't match with T [10]



Step11: i = 11, q =4

Comparing P [5] with T [11]

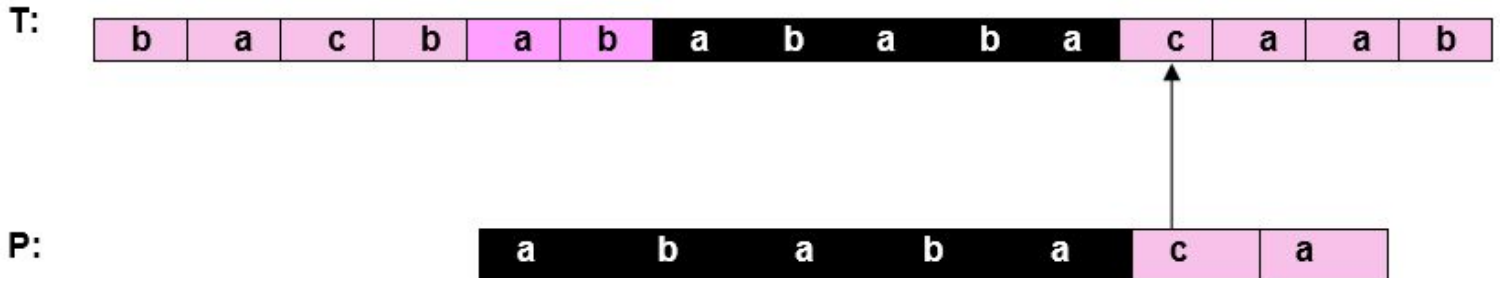
P [5] match with T [11]



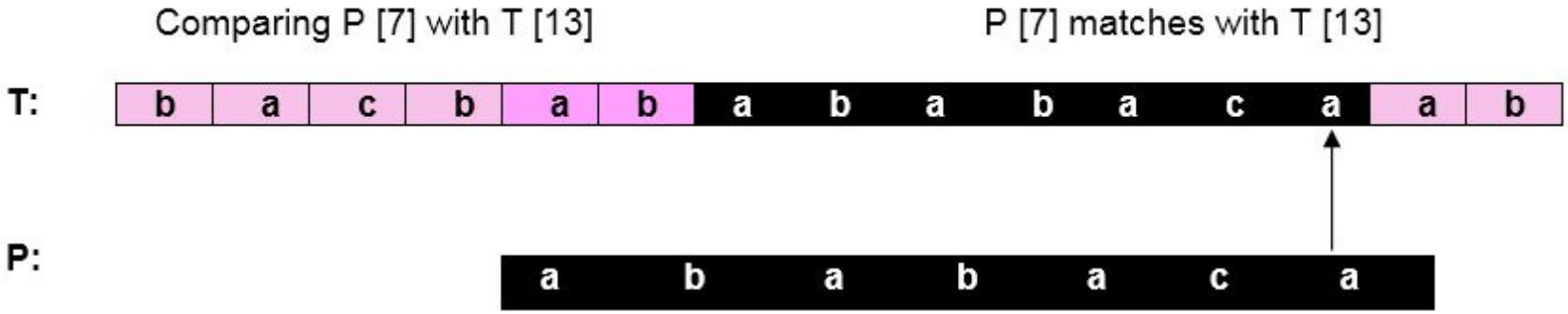
Step12: i = 12, q = 5

Comparing P [6] with T [12]

P [6] matches with T [12]



Step13: i = 3, q = 6



Pattern 'P' has been found to complexity occur in a string 'T.' The total number of shifts that took place for the match to be found is i-m = 13 - 7 = 6 shifts.

Longest Common Subsequence Algorithm

Longest Common Subsequence Algorithm

- **PROBLEM DEFINITION:**A subsequence of a string S, is a set of characters that appear in left –to right order, but not necessarily consecutively.
- Example
- ACTTGCG
- • ACT , ATTC , T , ACTTGC are all subsequences.
- • TTA is not a subequence
- • There are 2n subsequences of string S of length n.

What is LCS?

- A common subsequence of two strings is a subsequence that appears in both strings. A longest common subsequence is a common subsequence of maximal length.

Example

S1 = AAACCGTGAGTTATTCGTTCTAGAA S2 = CACCCCTAAGGTACCTTTGGTTC

- LCS is ACCTAGTACTTTG
- • Enumerate all sub-sequences of S1, and check if they are sub-sequences of S2. So the complexity will be $O(2^{nm})$

OPTIMAL SUBSTRUCTURE

If last characters of both sequences match (or X[m-1] ==Y[n-1]) then L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

- If last characters of both sequences do not match (or X[m-1] != Y[n-1]) then L(X[0..m-1], Y[0..n-1]) = MAX (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2])

Examples:

1)Consider the input strings "AGGTAB" and "GXTXAYB". Last characters match for the strings. So length of LCS can be written as: L("AGGTAB", "GXTXAYB") = 1 + L("AGGTA", "GXTXAY")

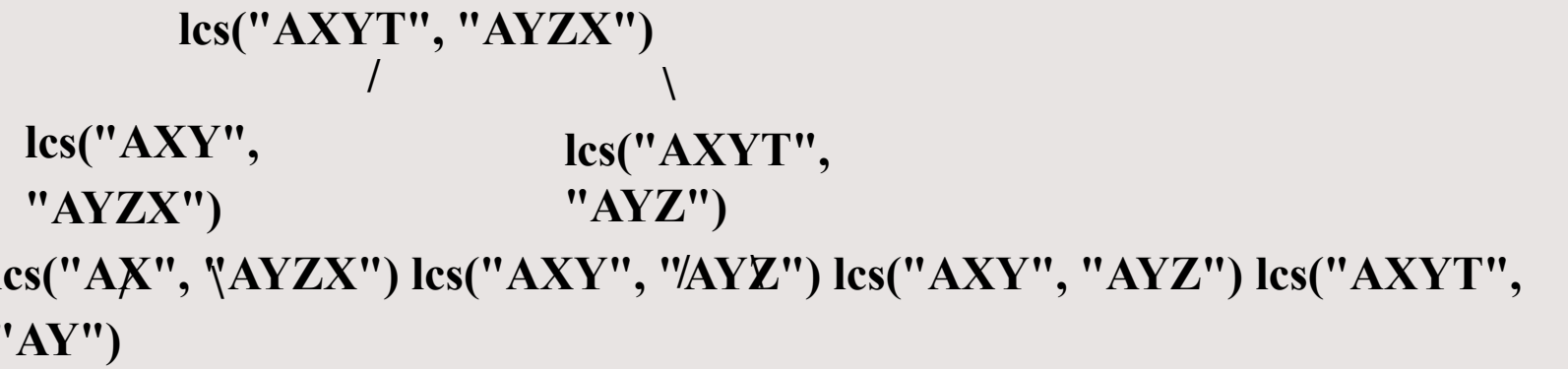
2)Consider the input strings "ABCDGH" and "AEDFHR". Last characters do not matchfor the strings. So length of LCS can be written as:

L("ABCDGH", "AEDFHR") = MAX (L("ABCDG", "AEDFHR"), L("ABCDGH", "AEDFH"))

Overlapping Subproblem

- This is a correct solution but it's very time consuming. For example, if the two strings have no matching characters, so the last line always gets executed, the the time bounds are binomial coefficients, which (if m=n)

are close to 2^n.



- In the above partial recursion tree, lcs("AXY", "AYZ") is being solved twice.

So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation by using "top down" approach of dynamic programming . The concept is to cache the result of a function given its parameter so that the calculation will not be repeated; it is simply retrieved, or memo-ed. Most of the time a simple array is used for the cache table, but a hash table or map could also be employed.

LCS Recursive equation

Theorem:

Let X =< x1, x2, . . . , xm > and Y =< y1, y2, . . . , yn > be sequences,and let Z =< z1, z2, . . . , zk > be any LCS of X and Y .

1.If xm = yn, then zk = xm = yn and Zk−1 is an LCS of Xm−1 and Yn−1.

2. If xm ≠ yn, then zk ≠ xm implies that Z is an LCS of Xm−1 and Yn .

3. If xm ≠ yn, then zk ≠ yn implies that Z is an LCS of X and Yn−1.

•So recursive solution for LCS IS

0 if i = 0 or j = 0

c[i, j] = c[i − 1, j − 1] + 1 if i, j > 0 and xi = yj ,

max(c[i, j − 1], c[i − 1, j]) if i, j > 0 and xi ≠yj

Algorithm

LCS − Length(X, Y)

m= length[X]
n= length[Y]

for i =1 to m

c[i, 0] = 0

for j =0 to n

c[0, j] = 0

for i = 1 to m

for j = 1 to n

if xi = yj

c[i, j] = c[i − 1, j − 1] + 1

B[i, j] := 'D' or ↖

else if c[i − 1, j] c[i, j − 1]

c[i, j] =c[i − 1, j]

B[i, j] := 'U' or ↑

else

c[i, j] =c[i, j − 1]
B[i, j] := 'L' or ←

return c and B

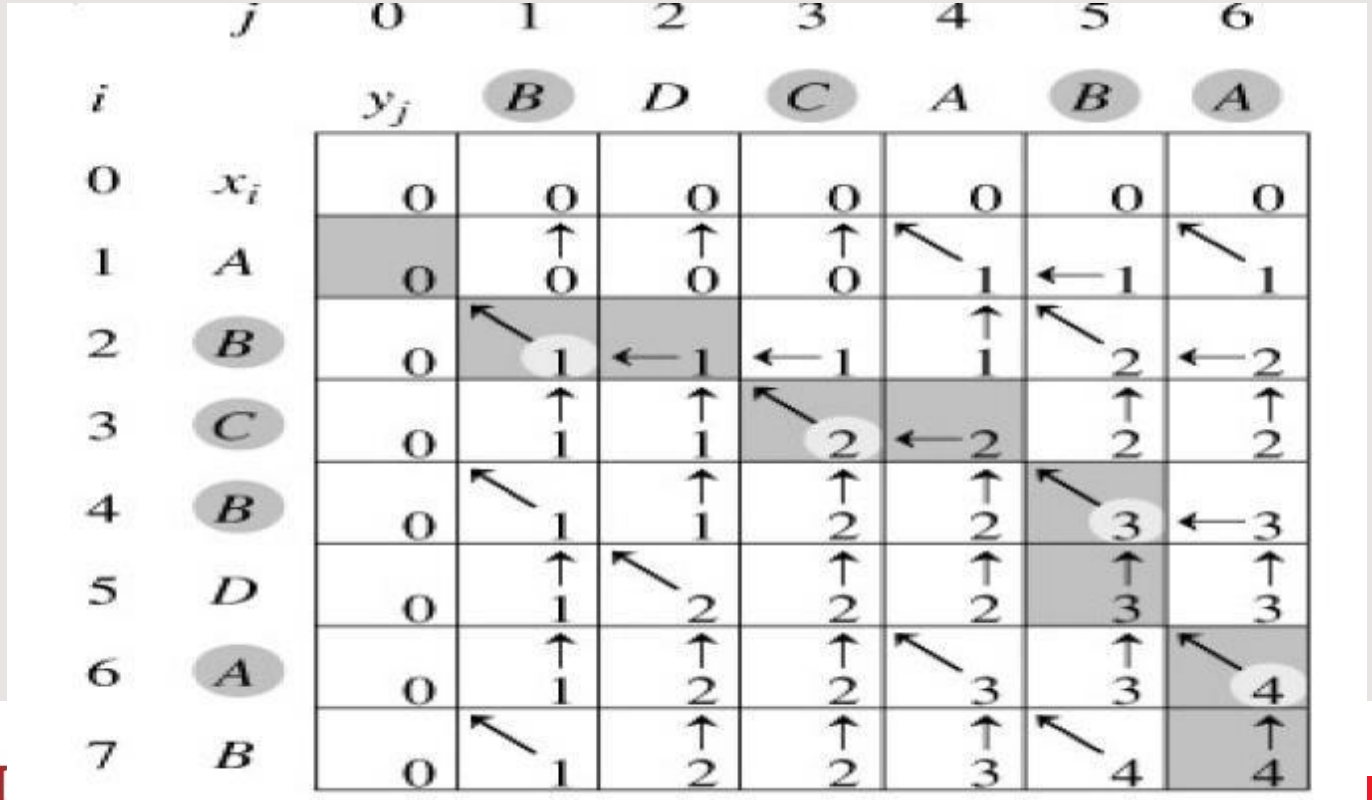
SEQUENCE RETRIEVAL

Algorithm: Print-LCS (B, X, i, j)

```
if i = 0 and j = 0
return
if B[i, j] = 'D'
        Print-LCS(B, X, i-1, j-1)
                Print(xi)
else if B[i, j] = 'U'
        Print-LCS(B, X, i-1, j)
else
        Print-LCS(B, X, i, j-1)
```

Example:

- We have two strings X = ABCBDAB and Y = BDCABA to find the longest common subsequence. Following the algorithm LCS Length-
-



	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Time complexity analysis

- Each call to subproblem takes constant time. We call it once from the main routine, and at most twice every time we fill in an entry of array L. There are (m)(n) entries, so the total number of calls is at most 2(m)(n) and the time is O (mn).

- As usual, this is a worst case analysis. The time might sometimes better, if not all array entries get filled out. For instance if the two strings match exactly, we'll only fill in diagonal entries and the algorithm will be fast.