Unit 6: String Matching Algorithms

String Matching means to find the occurrence of pattern within another string or text or body. There are many algorithms for performing efficient string matching. String Matching is used in various fields like plagiarism, information security, text mining, etc.

Naïve pattern searching is the simplest method among other pattern searching algorithms. It checks for all character of the main string to the pattern. This algorithm is helpful for smaller texts. It does not need any pre-processing phases. We can find substring by checking once for the string. It also does not occupy extra space to perform the operation.

String Matching Algorithm is an algorithm needed to find a place where particular or multiple strings are found within the larger string. All the alphabets of patterns(searched string) should be matched to the corresponding sequence.

Example: **S:** a b c d a b g | **P:** b c d

Now if you look for the pattern "bcd" in the string then it exists.

There are three major string matching algorithms:

- **Naive Algorithm (Brute Force)**
- **KMP Algorithm**
- **Rabin-Karp Algorithm**
  - **Applications of String Matching Algorithms:**

    Plagiarism Detection: The documents to be compared are decomposed into string tokens and compared using string matching algorithms. Thus, these algorithms are used to detect similarities between them and declare if the work is plagiarized or original.

    Bioinformatics and DNA Sequencing: Bioinformatics involves applying information technology and computer science to problems involving genetic sequences to find DNA patterns. String matching algorithms and DNA analysis are both collectively used for finding the occurrence of the pattern set.

    Digital Forensics: String matching algorithms are used to locate specific text strings of interest in the digital forensic text, which are useful for the investigation.

Spelling Checker: Trie is built based on a predefined set of patterns. Then, this trie is used for string matching. The text is taken as input, and if any such pattern occurs, it is shown by reaching the acceptance state.

Spam filters: Spam filters use string matching to discard the spam. For example, to categorize an email as spam or not, suspected spam keywords are searched in the content of the email by string matching algorithms. Hence, the content is classified as spam or not.

Search engines or content search in large databases: To categorize and organize data efficiently, string matching algorithms are used. Categorization is done based on the search keywords. Thus, string matching algorithms make it easier for one to find the information they are searching for.

Intrusion Detection System: The data packets containing intrusion-related keywords are found by applying string matching algorithms. All the malicious code is stored in the database, and every incoming data is compared with stored data. If a match is found, then the alarm is generated. It is based on exact string matching algorithms where each intruded packet must be detected.

A) Naive Pattern Searching Approach

Naive Pattern Searching Approach is one of the easy pattern-searching algorithms. All the characters of the pattern are matched to the characters of the corresponding/main string or text. In this approach, we will check all the possible placements of our input pattern with the input text.

This is simple and efficient brute force approach. It compares the first character of pattern with searchable text. If a match is found, pointers in both strings are advanced. If a match is not found, the pointer to text is incremented and pointer of the pattern is reset. This process is repeated till the end of the text.

The naïve approach does not require any pre-processing. Given text T and pattern P, it directly starts comparing both strings character by character. After each comparison, it shifts pattern string one position to the right.

Advantages of Naive String Matching Algorithm

- The comparison of the pattern with the given string can be done in any order.
- There is no extra space required.
- The most important thing that it doesn't require the pre-processing phase, as the running time is equal to matching time

Disadvantages of Naive String Matching Algorithm

- The one thing that makes this algorithm inefficient (time-consuming) is when it founds the pattern at an index then it does not use it again to find the other index. It again starts from the beginning and starts matching the pattern all over again.
- It doesn't use information from the previous iteration.

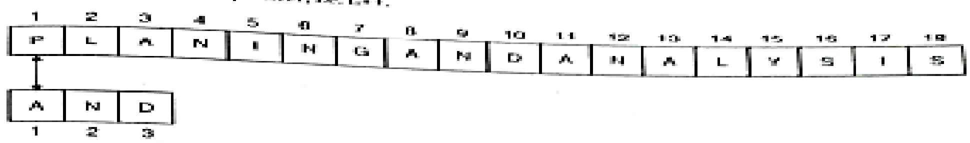
Algorithm

```
NAIVE_STRING_MATCHING(T, P)
n ← length [T]
m ← length [P]
for i ← 0 to n – m
do
if P[1... m] == T[i+1...i+m]
Then
print "Pattern occurs with shift" s
end
end
```

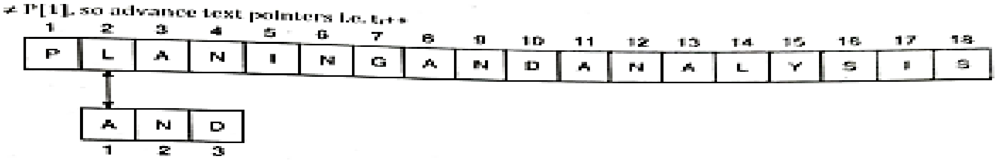
Examples:

Suppose given text T = [P,L,A,N,I,N,G,A,N,D,A,N,A,L,Y,S,I,S]] and Pattern to be found is P= [A, N, D]

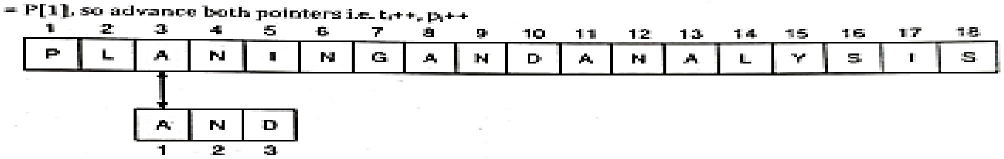
Step 1 : T[1] ≠ P[1], so advance text pointer, i.e. t_i++.



Step 2 : T[2] ≠ P[1], so advance text pointers i.e. t_i++

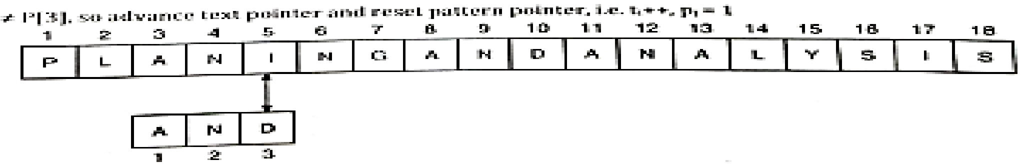


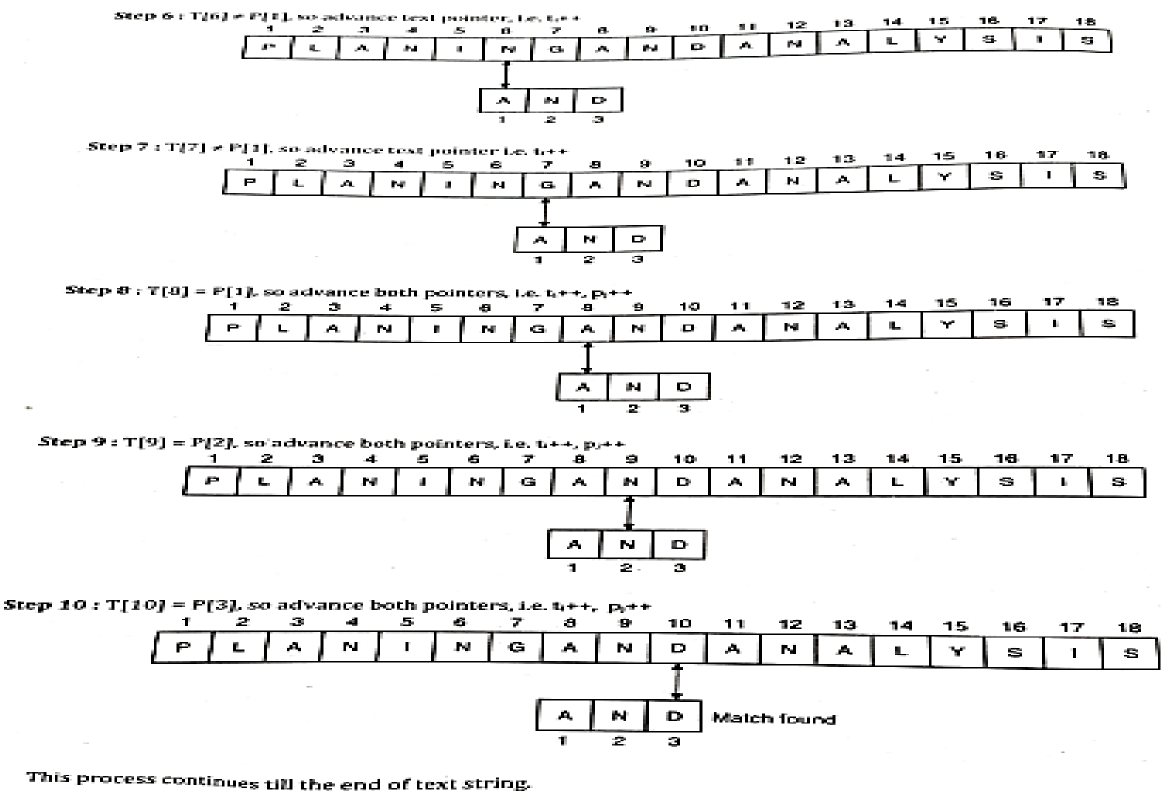
Step 3 : T[3] = P[1], so advance both pointers i.e. t_i++, p_i++



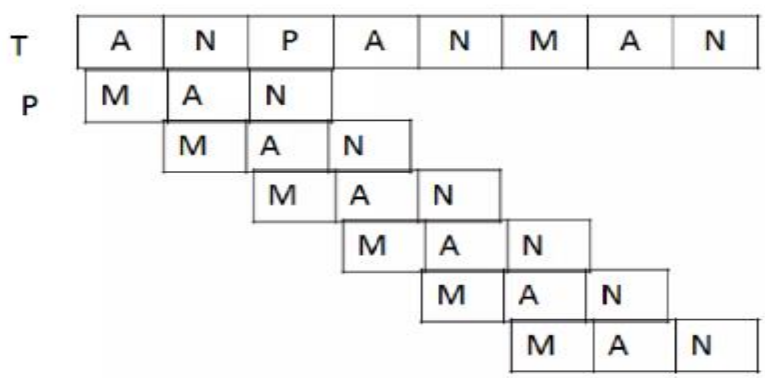
Step 4 : T[4] = P[2], so advance both pointers, i.e. t_i++, p_i++

Step 5 : T[5] ≠ P[3], so advance text pointer and reset pattern pointer, i.e. t_i++, p_i = 1





Suppose given text is T= A N P A N M A N and Pattern is P= M A N.



So pattern found at position 6

Refer class notes for more examples

❖ **Analysis of Algorithm:**

a) Time complexity

- This for loop from 3 to 5 executes for n-m + 1(we need at least m characters at the end) times and in iteration we are doing m comparisons. So the total complexity is O (n-m+1).In the naive string matching algorithm, the time complexity of the algorithm comes out to be **O(n-m+1)**, where n is the size of the input string and m is the size of the input pattern string.

b) Space complexity

- In the naive string matching algorithm, the space complexity of the algorithm comes out to be **O(1)**.

B) Rabin Karp Algorithm

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

ALGORITHM:

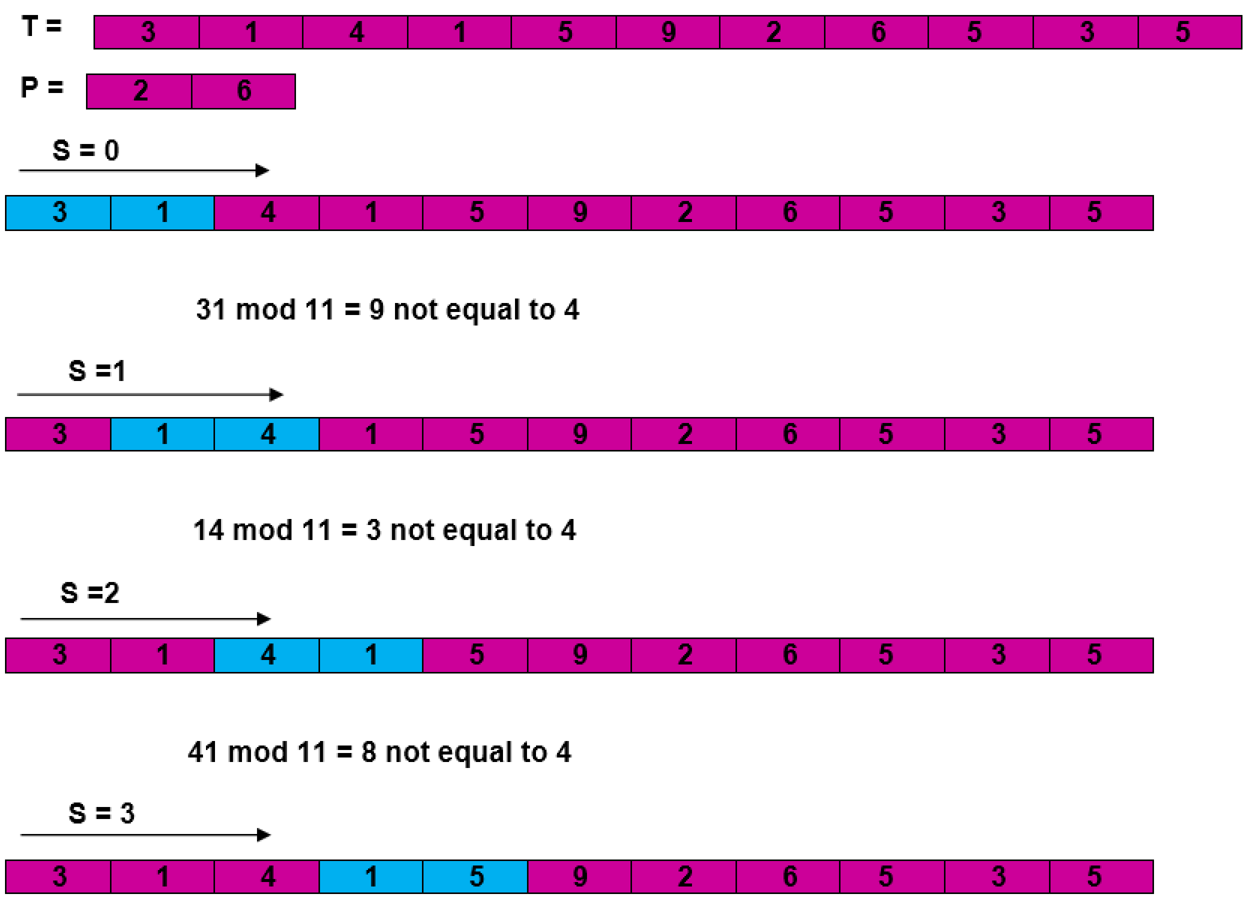
RABIN-KARP-MATCHER (T, P, d, q)

```
n ← length [T]
m  ← length [P]
h  ←  d^(m-1) mod q
p ←  0
t0 ←  0
for i ← 1 to m
do p ←  (dp + P[i]) mod q
t0 ← (dt0+T [i]) mod q
for s  ←  0 to n-m
do if p = ts
then if P [1.....m] = T [s+1.....s + m]
then "Pattern occurs with shift" s
If s < n-m
then ts+1 ←  (d (ts-T [s+1]h)+T [s+m+1])mod q
```

Example:

For string matching, working module q = 11, Suppose the given text is Text T = 31415926535 and Pattern to be found is P= 26. Let T = 31415926535....... P = 26 Here T.Length =11 so Q is prime no. assumed for computing hash value is = 11. For pattern h(p)= P mod Q = 26 mod 11 = 4 . Now find the exact match of P mod Q...

Solution:



15 mod 11 = 4 equal to 4 SPURIOUS HIT

S = 4

3	1	4	1	5	9	2	6	5	3	5

59 mod 11 = 4 equal to 4 SPURIOUS HIT

S = 5

3	1	4	1	5	9	2	6	5	3	5

92 mod 11 = 4 equal to 4 SPURIOUS HIT

S = 6

3	1	4	1	5	9	2	6	5	3	5

26 mod 11 = 4 EXACT MATCH

S = 7

3	1	4	1	5	9	2	6	5	3	5

S = 7

3	1	4	1	5	9	2	6	5	3	5

65 mod 11 = 10 not equal to 4

S = 8

3	1	4	1	5	9	2	6	5	3	5

53 mod 11 = 9 not equal to 4

S = 9

3	1	4	1	5	9	2	6	5	3	5

35 mod 11 = 2 not equal to 4

The Pattern occurs with shift 6.

For more examples, refer class notes and refer https://www.programiz.com/dsa/rabin-karp-algorithm for Advanced Rabin Karp.

❖

❖ **Limitation of Rabin Karp**

Spurious Hit

- When the hash value of the pattern matches with the hash value of a window of the text but the window is not the actual pattern then it is called a spurious hit.
- Spurious hit increases the time complexity of the algorithm. In order to minimize spurious hit, we use modulus. It greatly reduces the spurious hit.

❖ **Rabin-Karp Algorithm Complexity**

Time Complexity:

- The average case and best case complexity of Rabin-Karp algorithm is O(m + n) and the worst case complexity is O(mn).
- The worst-case complexity occurs when spurious hits occur a number for all the windows.

Space Complexity: O(1)

- It uses constant space. So, the space complexity is O(1).

C) Knuth-Morris-Pratt 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.

Example:

Suppose given string (T) and pattern are (P)

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

P:

a	b	a	b	a	c	a

Findout whether pattern is found in the given string or not?

Solution:

A) Generating Pi table for Pattern

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

Step II: Run KMP Matching Algorithm

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: Let us execute the KMP Algorithm to find whether 'P' occurs in 'T.'

T:	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

P:	a	b	a	b	a	c	a

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

Initially: n = size of T = 15

m = size of P = 7

Step1: i=1, q=0

Comparing P [1] with T [1]

T:	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

P:	a	b	a	b	a	c	a

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]

T:	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

P:		a	b	a	b	a	c	a

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

Step 3: i = 3, q = 1

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

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

P:

a	b	a	b	a	c	a

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

Step4: i = 4, q = 0

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

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

P:

a	b	a	b	a	c	a

Step5: i = 5, q = 0

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

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

P:

a	b	a	b	a	c	a

Step6: i = 6, q = 1

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

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

P:

a	b	a	b	a	c	a

Step7: i = 7, q = 2

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

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

P:

a	b	a	b	a	c	a

Step8: i = 8, q =3

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

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

P:

a	b	a	b	a	c	a

Step9: i = 9, q = 4

Comparing P [5] with T [9]   P [5] matches with T [9]

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

P:

a	b	a	b	a	c	a

Step10: i = 10, q = 5

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

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

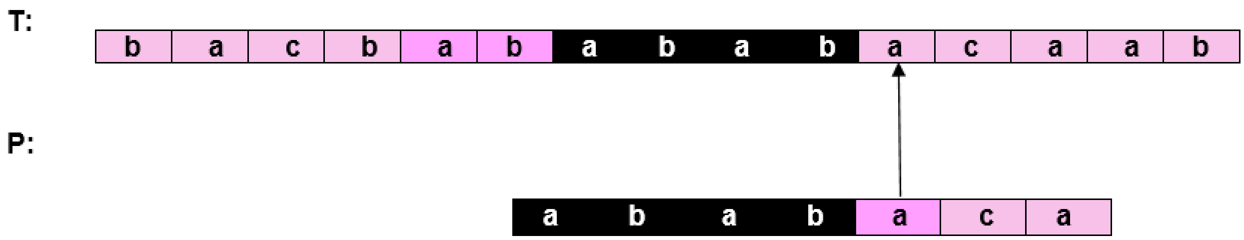
P:

a	b	a	b	a	c	a

Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

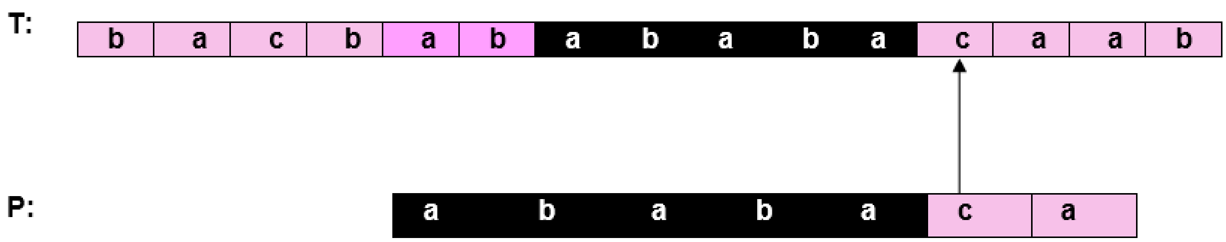
Step11: i = 11, q =4

Comparing P [5] with T [11] P [5] match with T [11]



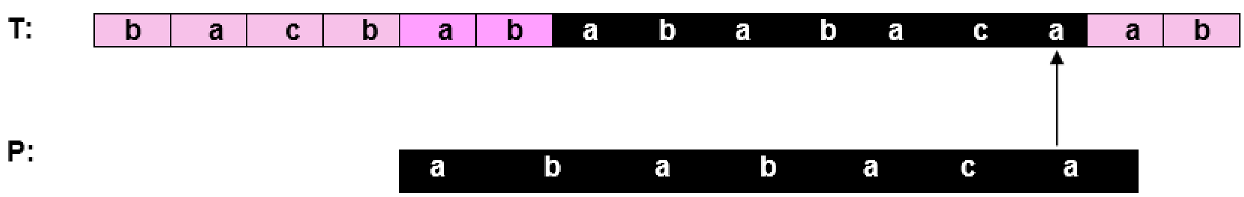
Step12: i = 12, q = 5

Comparing P [6] with T [12] P [6] matches with T [12]



Step13: i = 3, q = 6

Comparing P [7] with T [13] P [7] matches with T [13]



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.

Algorithm:

COMPUTE- PREFIX- FUNCTION (P)

```
m ←length [P]                //'p' pattern to be matched
Π [1] ← 0
k ← 0
for q ← 2 to m
do while k > 0 and P [k + 1] ≠ P [q]
do k ← Π [k]
If P [k + 1] = P [q]
then k← k + 1
Π [q] ← k
Return Π
```

KMP-MATCHER (T, P)

```
n ← length [T]
m ← length [P]
Π← COMPUTE-PREFIX-FUNCTION (P)
q ← 0                  // numbers of characters matched
for i ← 1 to n         // scan S from left to right
do while q > 0 and P [q + 1] ≠ T [i]
do q ← Π [q]                  // next character does not match
If P [q + 1] = T [i]
then q ← q + 1                // next character matches
If q = m                                // is all of p matched?
then print "Pattern occurs with shift" i - m
q ← Π [q]                               // look for the next match
```

Complexity Analysis:

- **Time Complexity:** Time complexity of the search algorithm is $O(n)$. These complexities are the same, no matter how many repetitive patterns are in.

- **Space Complexity:** It has a space complexity of O (m) because there's some pre-processing involved.
