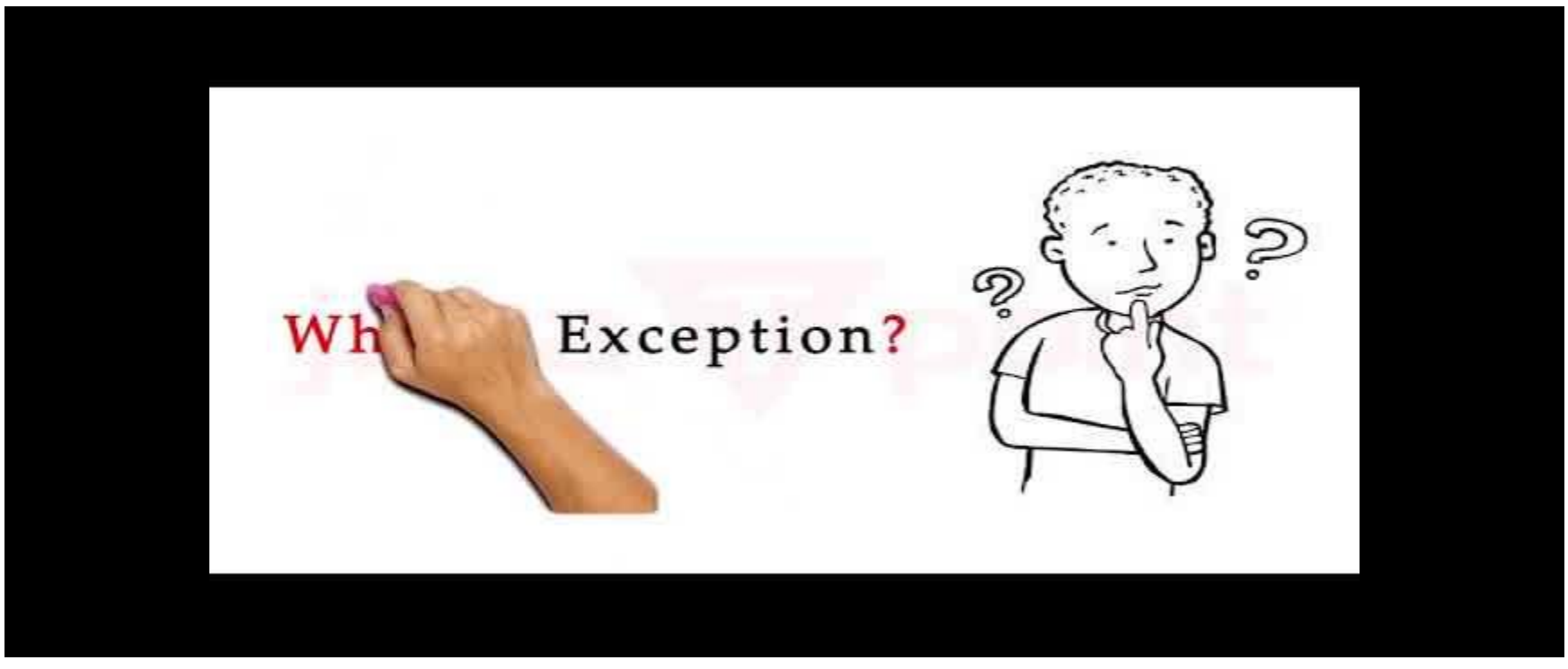
String Matching Introduction

String Matching Algorithm is also called "String Searching Algorithm." This is a vital class of string algorithm is declared as "this is the method to find a place where one is several strings are found within the larger string."

Given a text array, T [1.....n], of n character and a pattern array, P [1.....m], of m characters. The problems are to find an integer s, called **valid shift** where 0 ≤ s < n-m and T [s+1.....s+m] = P [1.....m]. In other words, to find even if P in T, i.e., where P is a substring of T. The item of P and T are character drawn from some finite alphabet such as {0, 1} or {A, B .....Z, a, b..... z}.Given a string T [1.....n], the **substrings** are represented as T [i.....j] for some 0≤i ≤ j≤n-1, the string formed by the characters in T from index i to index j, inclusive. This process that a string is a substring of itself (take i = 0 and j =m).

The **proper substring** of string T [1.....n] is T [1.....j] for some 0<i ≤ j≤n-1. That is, we must have either i>0 or j < m-1.

Using these descriptions, we can say given any string T [1.....n], the substrings are



1. T [i.....j] = T [i] T [i +1] T [i+2].....T [j] **for** some 0≤i ≤ j≤n-1.

And proper substrings are

1. T [i.....j] = T [i] T [i +1] T [i+2].....T [j] **for** some 0≤i ≤ j≤n-1.

Note: If i>j, then T [i.....j] is equal to the empty string or null, which has length zero.

Algorithms used for String Matching:

There are different types of method is used to finding the string

1. The Naive String Matching Algorithm
2. The Rabin-Karp-Algorithm
3. Finite Automata
4. The Knuth-Morris-Pratt Algorithm
5. The Boyer-Moore Algorithm

The Naive String Matching Algorithm

The naïve approach tests all the possible placement of Pattern P [1.....m] relative to text T [1.....n]. We try shift s = 0, 1.....n-m, successively and for each shift s. Compare T [s+1.....s+m] to P [1.....m].

The naïve 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 value of s.

NAIVE-STRING-MATCHER (T, P)

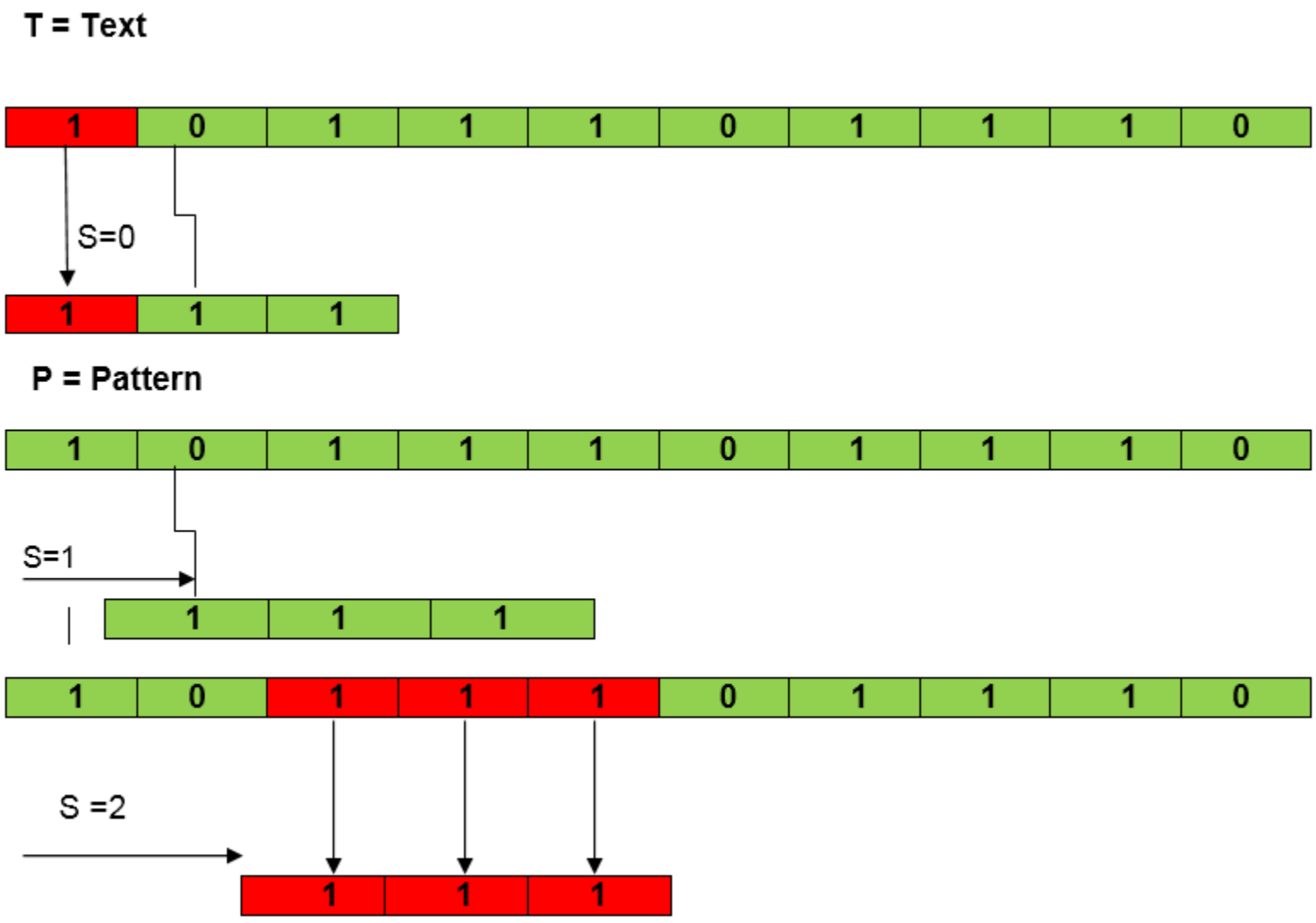
```
1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n -m
4. do if P [1.....m] = T [s + 1....s + m]
5. then print "Pattern occurs with shift" s
```

Analysis: 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).

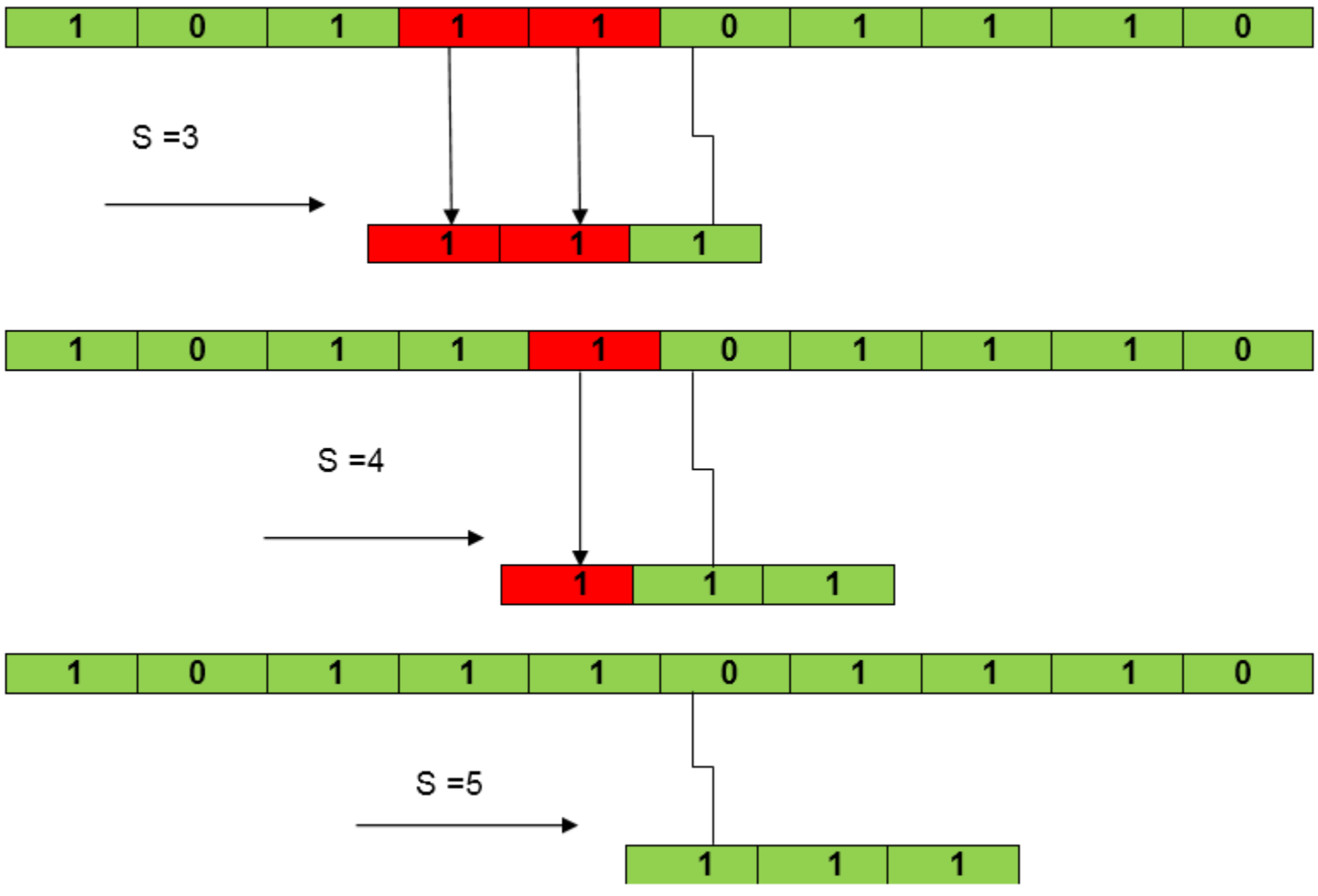
Example:

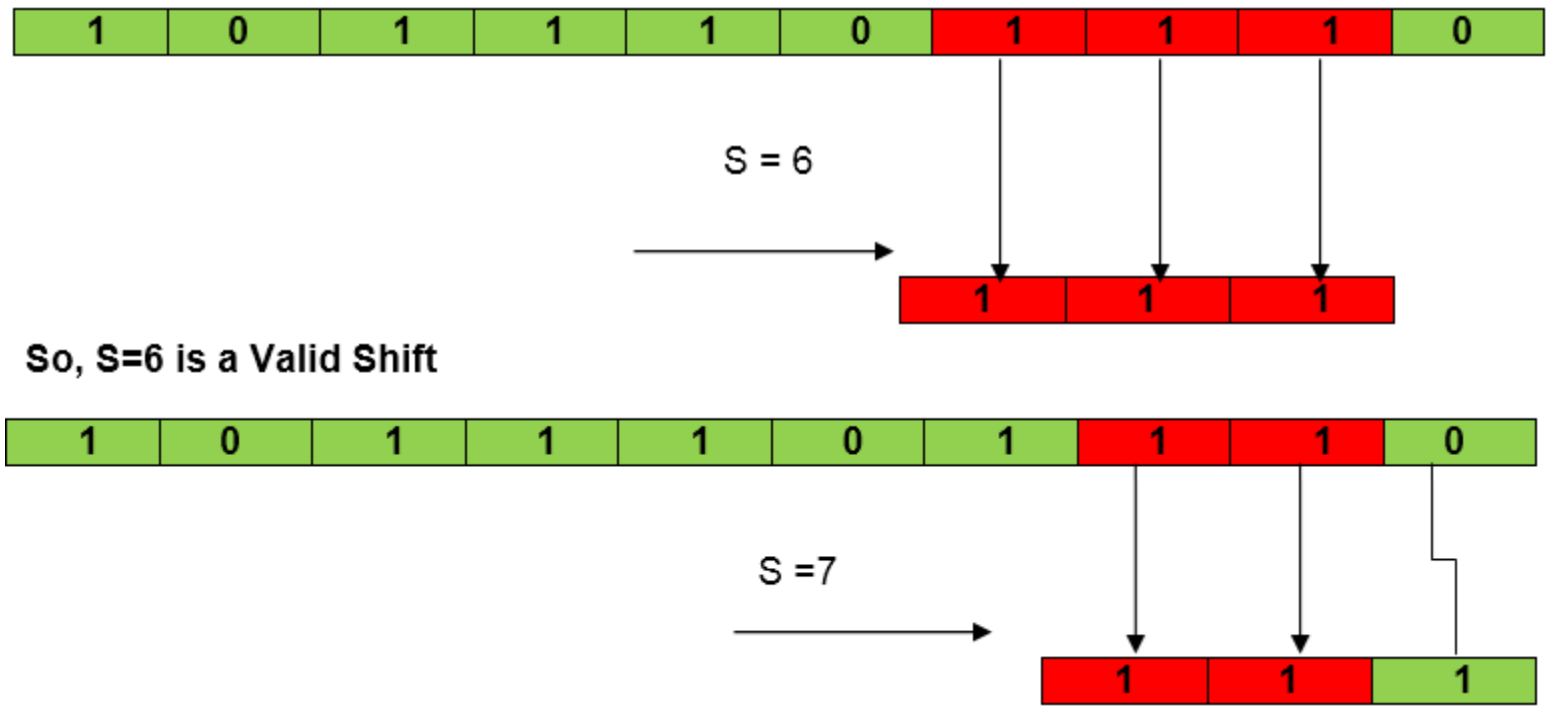
1. Suppose T = 1011101110
2. P = 111
3. Find all the Valid Shift

Solution:



So, S=2 is a Valid Shift





The 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.

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

```
1. n ← length [T]
2. m  ← length [P]
3. h  ←  d^(m-1) mod q
4. p ←  0
5. t0 ←  0
6. for i ← 1 to m
7. do p ←  (dp + P[i]) mod q
8. t0 ← (dt0+T [i]) mod q
9. for s  ←  0 to n-m
10. do if p = ts
11. then if P [1.....m] = T [s+1.....s + m]
12. then "Pattern occurs with shift" s
13. If s < n-m
14. then ts+1 ←  (d (ts-T [s+1]h)+T [s+m+1])mod q
```

Example: For string matching, working module q = 11, how many spurious hits does the Rabin-Karp matcher encounters in Text T = 31415926535.....

1. T = 31415926535.....
2. P = 26
3. Here T.Length =11 so Q = 11
4. And P mod Q = 26 mod 11 = 4
5. Now find the exact match of P mod Q...

Solution:

T = 3 1 4 1 5 9 2 6 5 3 5

P = 2 6

S = 0

3 1 4 1 5 9 2 6 5 3 5

31 mod 11 = 9 not equal to 4

S =1

3 1 4 1 5 9 2 6 5 3 5

14 mod 11 = 3 not equal to 4

S =2

3 1 4 1 5 9 2 6 5 3 5

41 mod 11 = 8 not equal to 4

S = 3

3 1 4 1 5 9 2 6 5 3 5

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

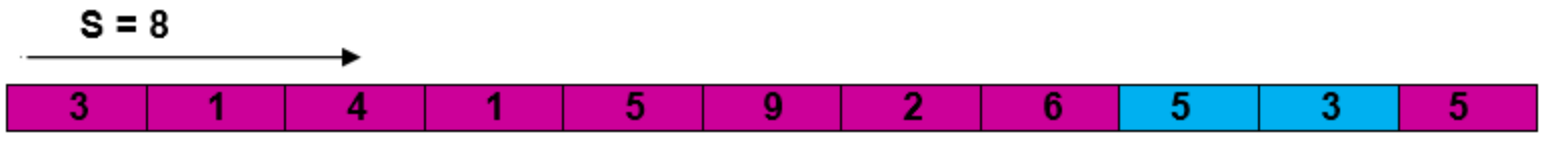
3 1 4 1 5 9 2 6 5 3 5

26 mod 11 = 4 EXACT MATCH

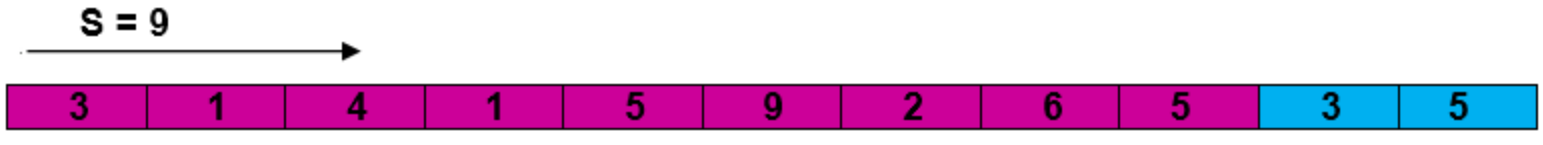
S = 7

3 1 4 1 5 9 2 6 5 3 5

65 mod 11 = 10 not equal to 4



53 mod 11 = 9 not equal to 4



35 mod 11 = 2 not equal to 4

The Pattern occurs with shift 6.

Complexity:

The running time of **RABIN-KARP-MATCHER** in the worst case scenario **O ((n-m+1) m** but it has a good average case running time. If the expected number of strong shifts is small **O (1)** and prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time **O (n+m)** plus the time to require to process spurious hits.

String Matching with Finite Automata

The string-matching automaton is a very useful tool which is used in string matching algorithm. It examines every character in the text exactly once and reports all the valid shifts in O (n) time. The goal of string matching is to find the location of specific text pattern within the larger body of text (a sentence, a paragraph, a book, etc.)

Finite Automata:

A finite automaton **M** is a 5-tuple **$(Q, q_0, A, \sum \delta)$**, where

- Q is a finite set of **states**,
- $q_0 \in Q$ is the **start state**,
- $A \subseteq Q$ is a notable set of **accepting states**,
- $\sum$ is a **finite input alphabet**,
- δ is a function from **Q x $\sum$** into **Q** called the **transition function** of **M**.

The finite automaton starts in state **q_0** and reads 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 δ (q, a). Whenever its current state q is a member of A, the machine M has accepted the string read so far. An input that is not allowed is **rejected**.

A finite automaton M induces a function $\emptyset$ called the called the **final-state function**, from $\sum^*$ to Q such that $\emptyset$(w) is the state M ends up in after scanning the string w. Thus, M accepts a string w if and only if $\emptyset(w) \in A$.

The function f is defined as

$\emptyset(\in) = q_0$

$\emptyset(wa) = \delta((\emptyset(w), a)$ for $w \in \sum^*, a \in \sum$)

FINITE- AUTOMATON-MATCHER (T,δ, m),

```
1. n ← length [T]
2. q ← 0
3. for i ← 1 to n
4. do q ← δ (q, T[i])
5. If q =m
6. then s←i-m
7. print "Pattern occurs with shift s" s
```

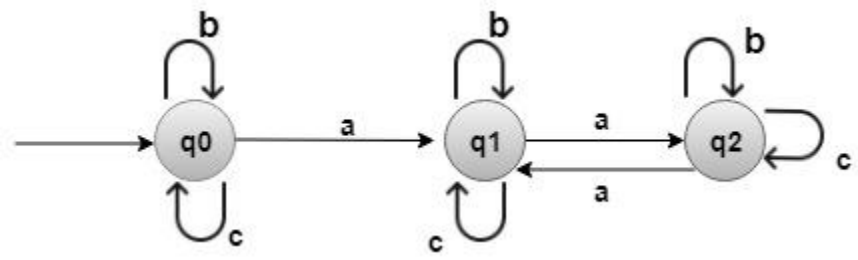
The primary loop structure of FINITE- AUTOMATON-MATCHER implies that its running time on a text string of length n is O (n).

Computing the Transition Function: The following procedure computes the transition function δ from given pattern P [1.....m]

COMPUTE-TRANSITION-FUNCTION (P, ∑)

```
1. m ← length [P]
2. for q ← 0 to m
3. do for each character a ∈ ∑*
4. do k ← min (m+1, q+2)
5. repeat k←k-1
6. Until
7. δ(q,a)←k
8. Return δ
```

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



Solution:

q_0 is the initial state.

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

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

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

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

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

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

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

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.

The Prefix Function (Π)

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
```

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:

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]              // next character does not match
8. If P [q + 1] = T [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
```

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:

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

P:	a	b	a	b	a	c	a

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

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

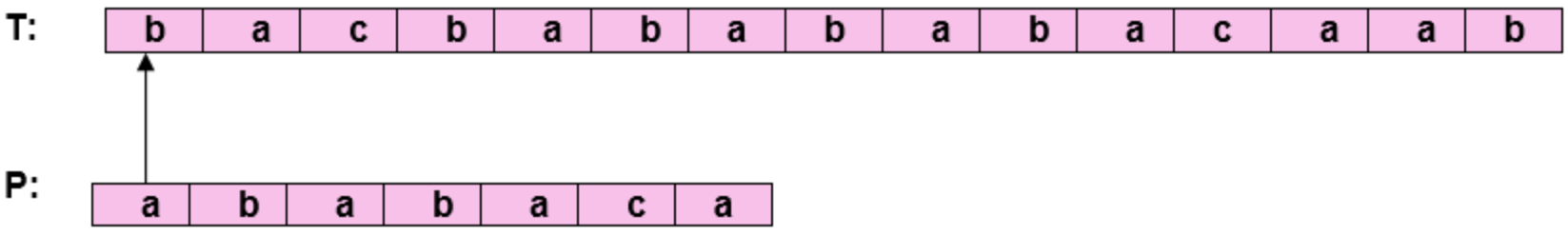
q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

Solution:

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

Step1: i=1, q=0

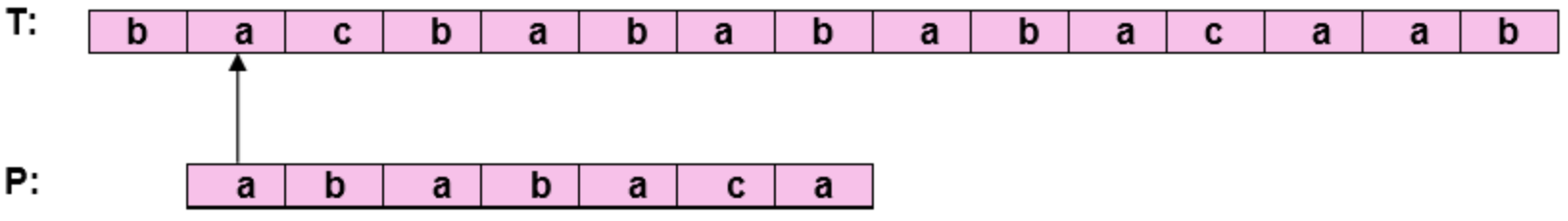
Comparing P [1] with T [1]



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

Step2: i = 2, q = 0

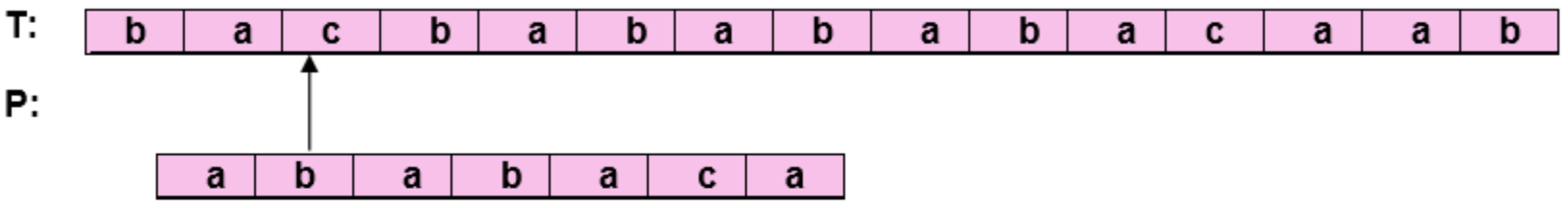
Comparing P [1] with T [2]



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

Step 3: i = 3, q = 1

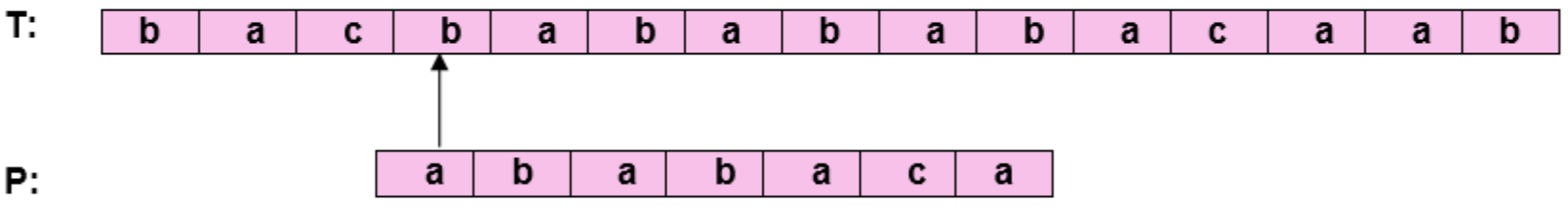
Comparing P [2] with T [3] P [2] doesn't match with T [3]



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

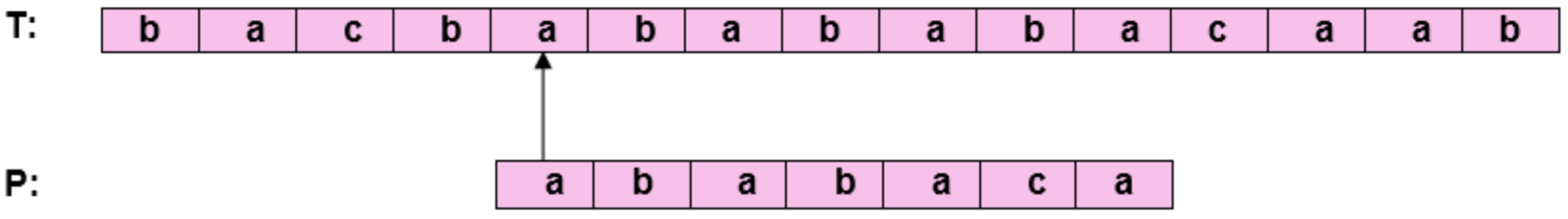
Step4: i = 4, q = 0

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



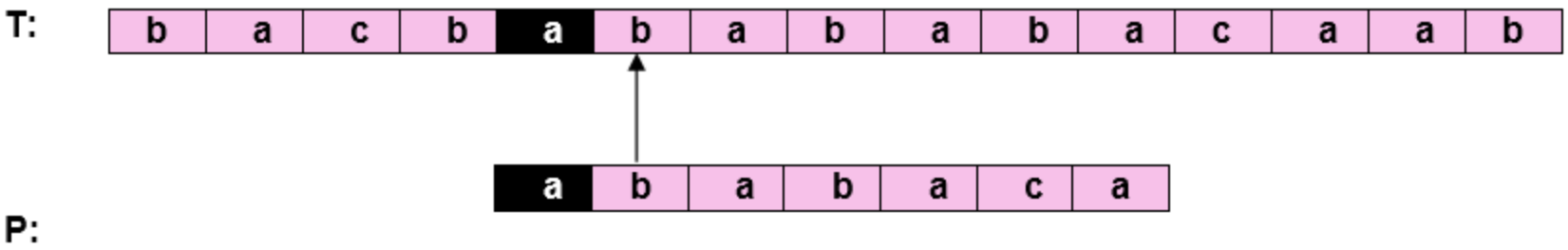
Step5: i = 5, q = 0

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



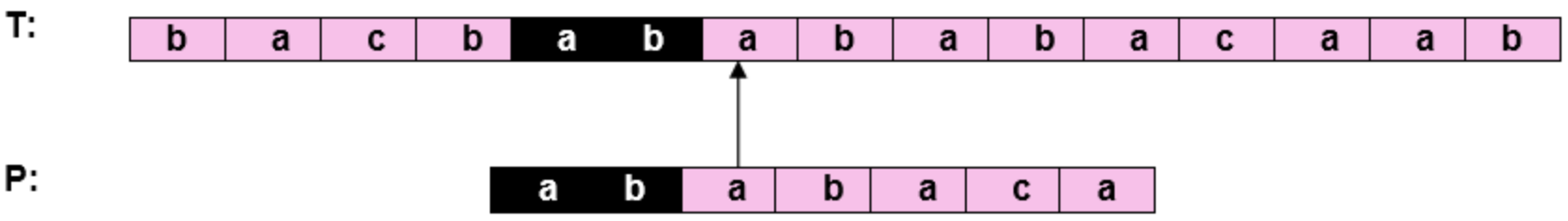
Step6: i = 6, q = 1

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



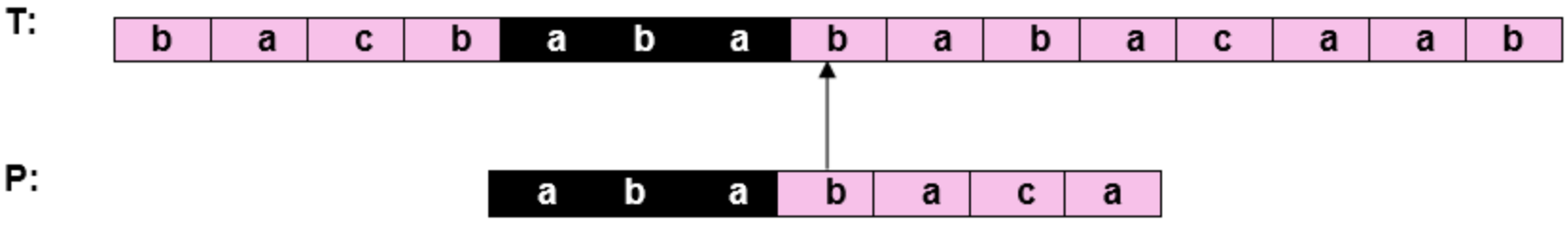
Step7: i = 7, q = 2

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



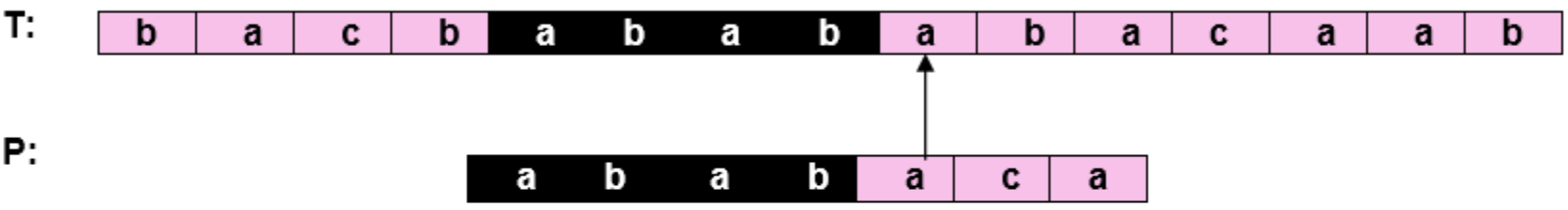
Step8: i = 8, q =3

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



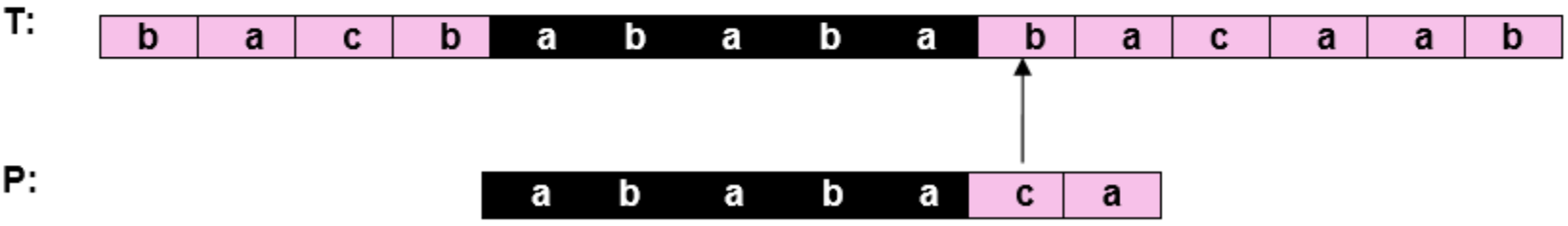
Step9: i = 9, q = 4

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



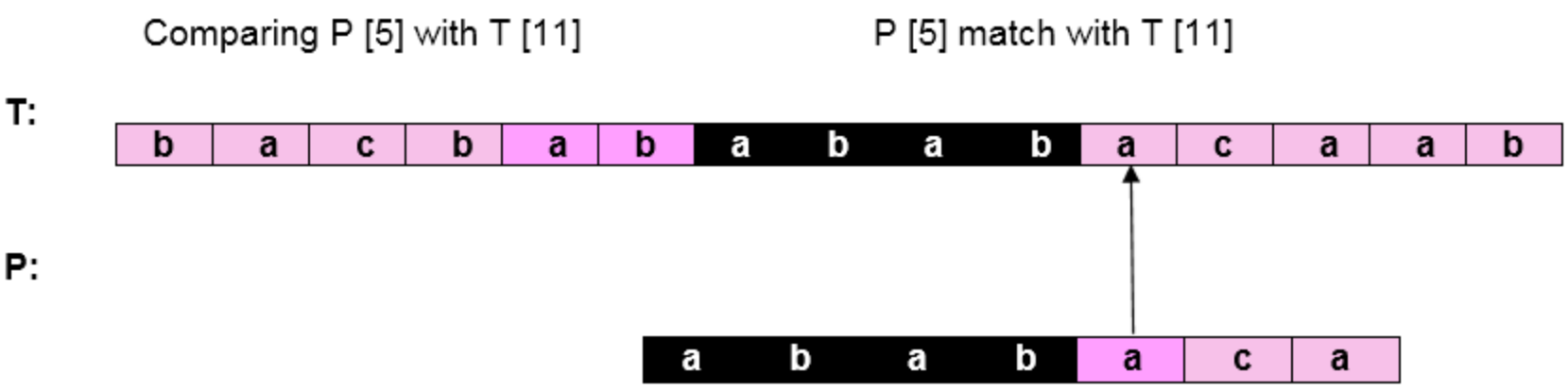
Step10: i = 10, q = 5

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

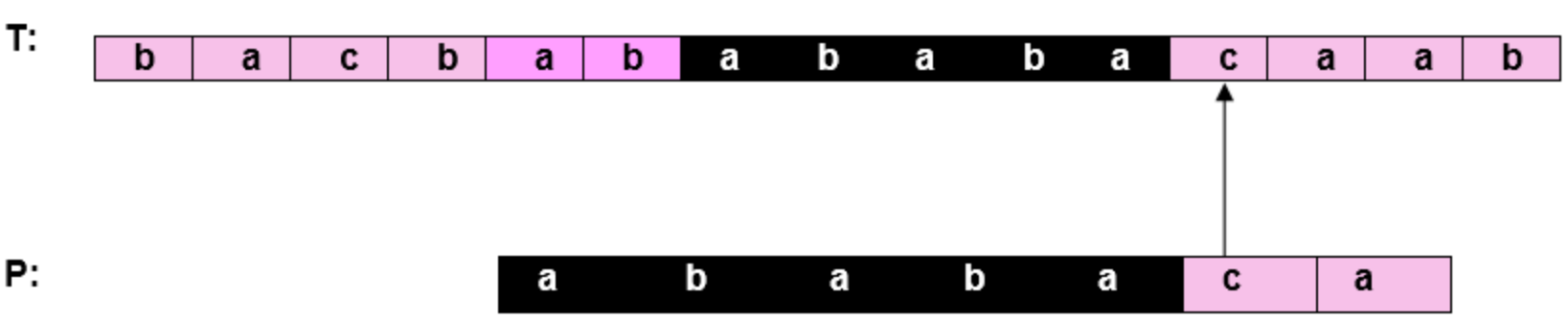


Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

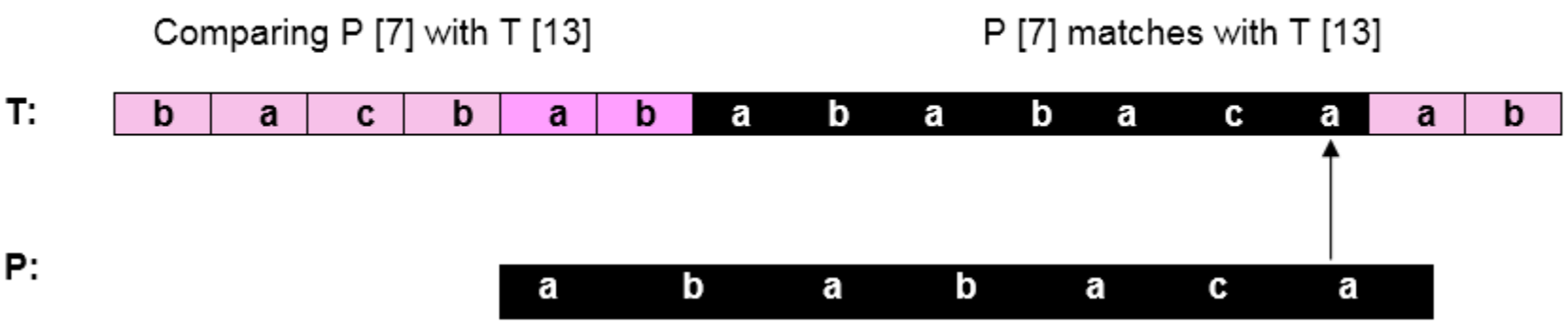
Step11: i = 11, q =4



Step12: i = 12, q = 5



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.

The Boyer-Moore Algorithm

Robert Boyer and J Strother Moore established it in 1977. The B-M String search algorithm is a particularly efficient algorithm and has served as a standard benchmark for string search algorithm ever since.

The B-M algorithm takes a 'backward' approach: the pattern string (P) is aligned with the start of the text string (T), and then compares the characters of a pattern from right to left, beginning with rightmost character.

If a character is compared that is not within the pattern, no match can be found by analyzing any further aspects at this position so the pattern can be changed entirely past the mismatching character.

For deciding the possible shifts, B-M algorithm uses two preprocessing strategies simultaneously. Whenever a mismatch occurs, the algorithm calculates a variation using both approaches and selects the more significant shift thus, if make use of the most effective strategy for each case.

The two strategies are called heuristics of B - M as they are used to reduce the search. They are:

1. Bad Character Heuristics
2. Good Suffix Heuristics

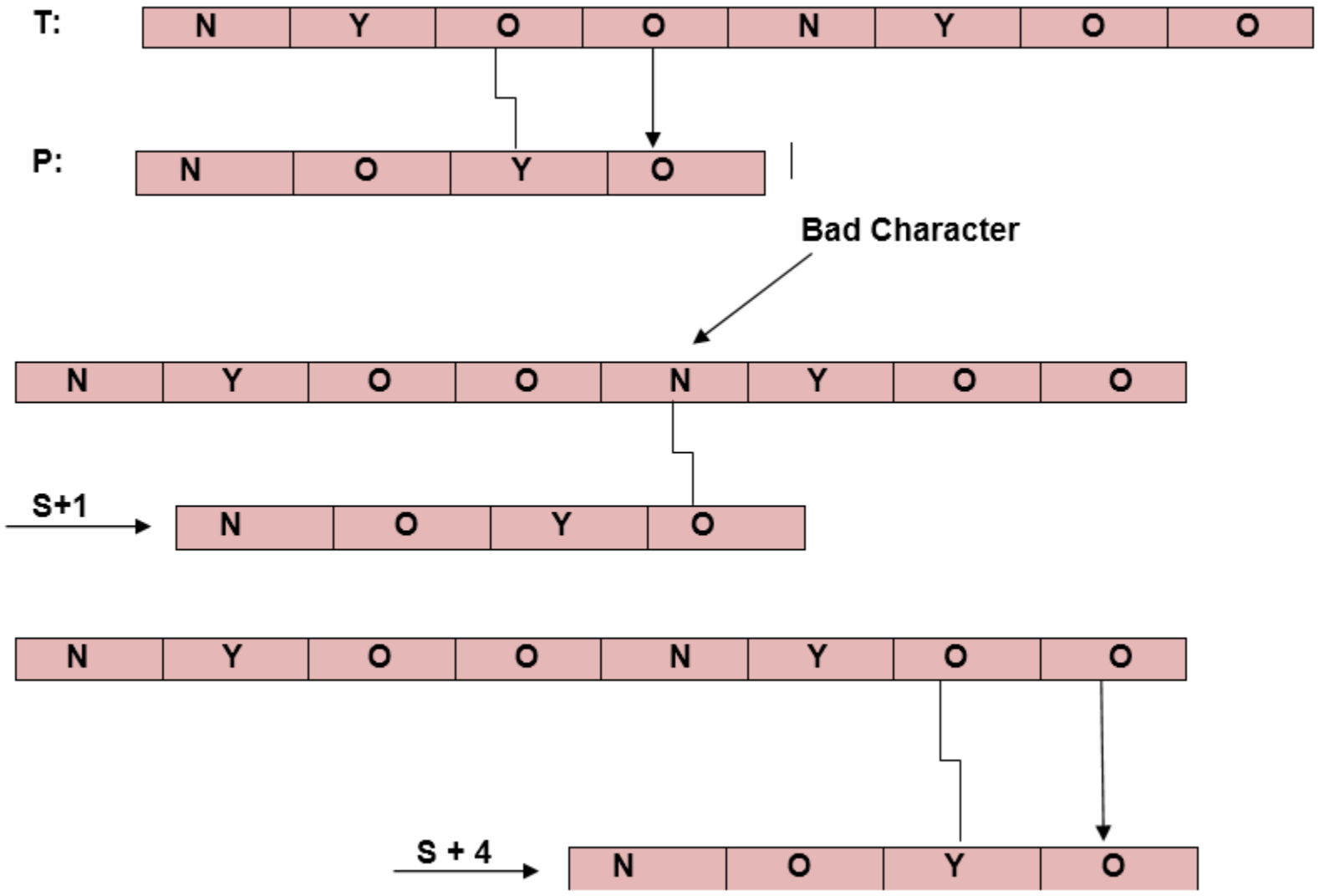
1. Bad Character Heuristics

This Heuristics has two implications:

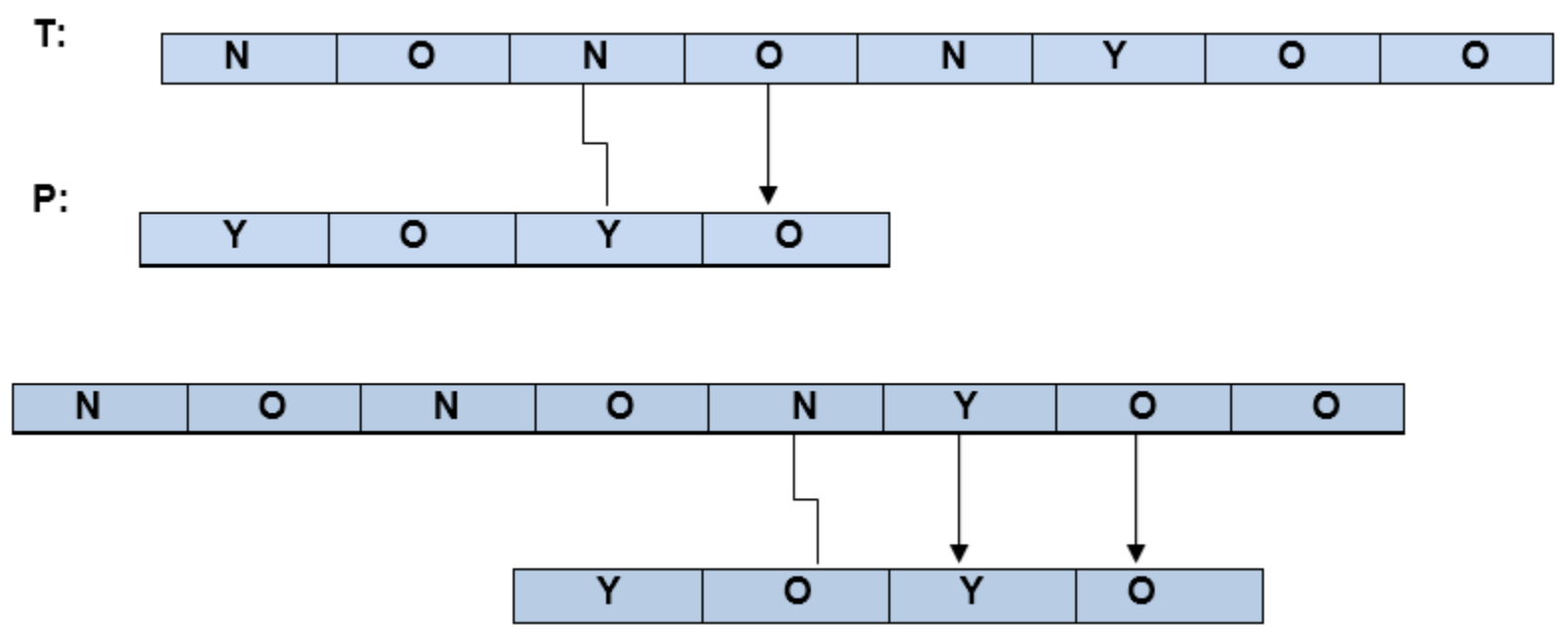
- Suppose there is a character in a text in which does not occur in a pattern at all. When a mismatch happens at this character (called as bad character), the whole pattern can be changed, begin matching form substring next to this 'bad character.'
- On the other hand, it might be that a bad character is present in the pattern, in this case, align the nature of the pattern with a bad character in the text.

Thus in any case shift may be higher than one.

Example1: Let Text T = <nyoo nyoo> and pattern P = <noyo>



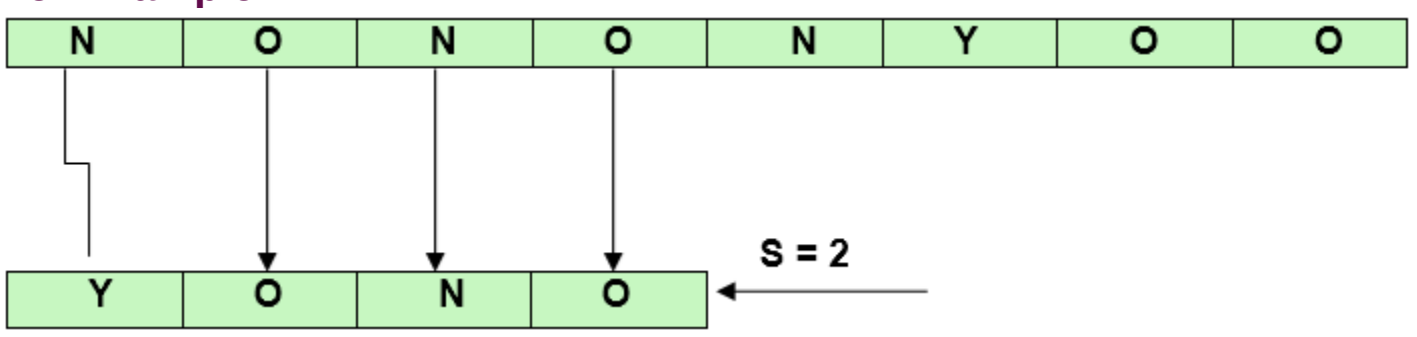
Example2: If a bad character doesn't exist the pattern then.



Problem in Bad-Character Heuristics:

In some cases, Bad-Character Heuristics produces some negative shifts.

For Example:



This means that we need some extra information to produce a shift on encountering a bad character. This information is about the last position of every aspect in the pattern and also the set of characters used in a pattern (often called the alphabet ∑of a pattern).

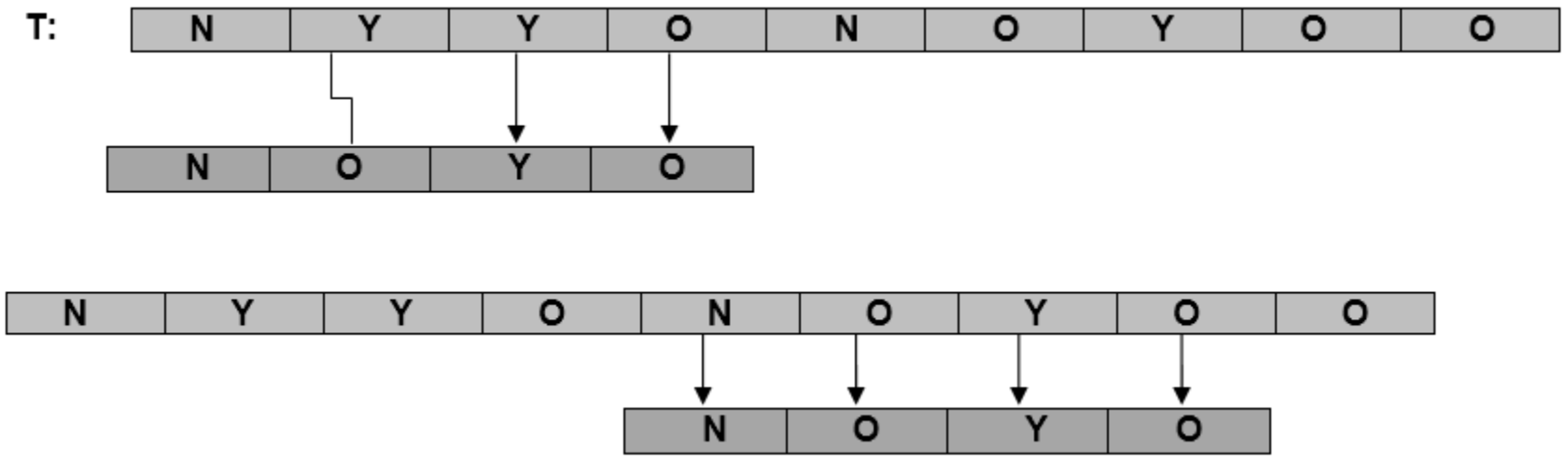
COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, ∑)

```
1. for each character a ∈ ∑
2. do λ [a] = 0
3. for j ← 1 to m
4. do λ [P [j]] ← j
5. Return λ
```

2. Good Suffix Heuristics:

A good suffix is a suffix that has matched successfully. After a mismatch which has a negative shift in bad character heuristics, look if a substring of pattern matched till bad character has a good suffix in it, if it is so then we have an onward jump equal to the length of suffix found.

Example:



COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)

```
1. Π ← COMPUTE-PREFIX-FUNCTION (P)
2. P'← reverse (P)
3. Π'← COMPUTE-PREFIX-FUNCTION (P')
4. for j ← 0 to m
5. do ɣ [j] ← m - Π [m]
6. for l ← 1 to m
7. do j ← m - Π' [L]
8. If ɣ [j] > l - Π' [L]
9. then ɣ [j] ← 1 - Π'[L]
10. Return ɣ
```

BOYER-MOORE-MATCHER (T, P, ∑)

```
1. n ←length [T]
2. m ←length [P]
3. λ← COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, ∑ )
4. ɣ← COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)
5. s ←0
6. While s ≤ n - m
7. do j ← m
8. While j > 0 and P [j] = T [s + j]
9. do j ←j-1
10. If j = 0
11. then print "Pattern occurs at shift" s
12. s ← s + ɣ[0]
13. else s ← s + max (ɣ [j], j -  λ[T[s+j]])
```

Complexity Comparison of String Matching Algorithm:

Algorithm	Preprocessing Time	Matching Time
Naive	O	(O (n - m + 1)m)
Rabin-Karp	O(m)	(O (n - m + 1)m)
Finite Automata	O(m|∑|)	O (n)
Knuth-Morris-Pratt	O(m)	O (n)
Boyer-Moore	O(|∑|)	(O ((n - m + 1) + |∑|))