Knuth-Morris-Pratt 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.

How does the O(mn) approach work

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

String S a b c a b a a b c a b a c

Pattern p a b a a

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

S

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

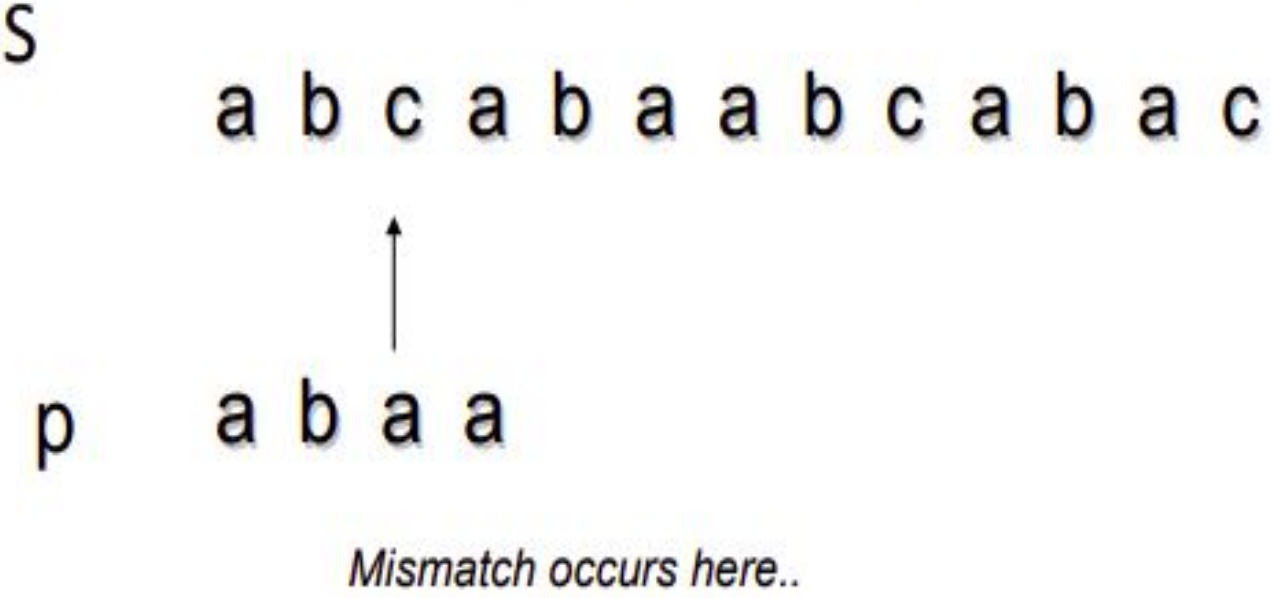
p a b a a

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

S a b c a b a a b c a b a c

p a b a a

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



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.

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 comparisons with elements of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs

Components of KMP

1. LPS Array
2. KMP matching function

LPS (Longest Proper prefix which is also a suffix) denoted by Π later on in slides.

LPS array is created for pattern string. For each sub-pattern p[0..i] where i = 0 to m-1, lps[i] stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

For eg.

For the pattern “AABAACAABAA”,
lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

The KMP Matcher

With string ‘S’, pattern ‘p’ and LPS array ‘Π’ as
inputs, finds the occurrence of ‘p’ in ‘S’ and returns
the number of shifts of ‘p’ after which occurrence is found.

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

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

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	2			

Step 3: q = 4, k = 1
Π[4] = 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] = 0

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

Step 6: q = 7, k = 0

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 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	0	1

Try to generate LPS for this pattern

pattern - > b a b a b a a b a

Try to generate LPS for this pattern

pattern - > b a b a b a a b a

LPS -> 0 0 1 2 3 4 0 1 2

```
// Fills lps[] for given patttern pat[0..M-1]
void computeLPSArray(char* pat, int M, int* lps)
{
    // length of the previous longest prefix suffix
    int len = 0;

    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {
            // This is tricky. Consider the example.
            // AAACAAAA and i = 7. The idea is similar
            // to search step.
            if (len != 0) {
                len = lps[len - 1];

                // Also, note that we do not increment
                // i here
            }
            else // if (len == 0)
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}
```

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

S

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

p a b a b a c a

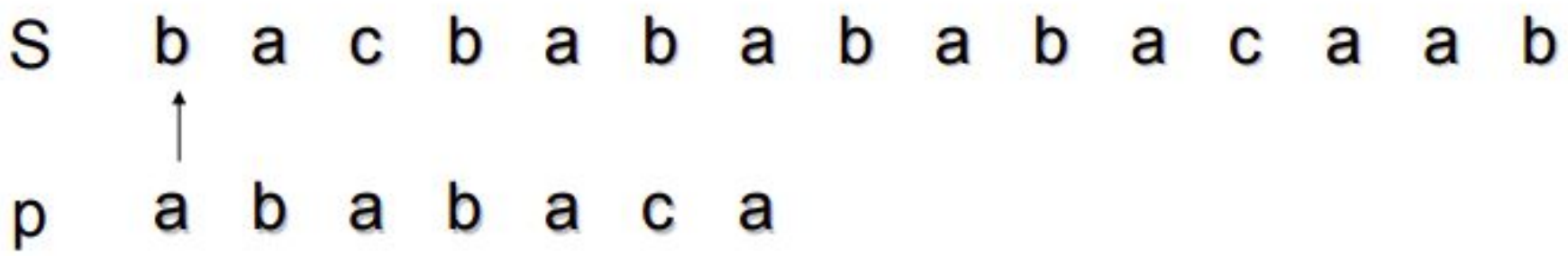
Let us execute the KMP algorithm to find whether ‘p’ occurs in ‘S’.

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

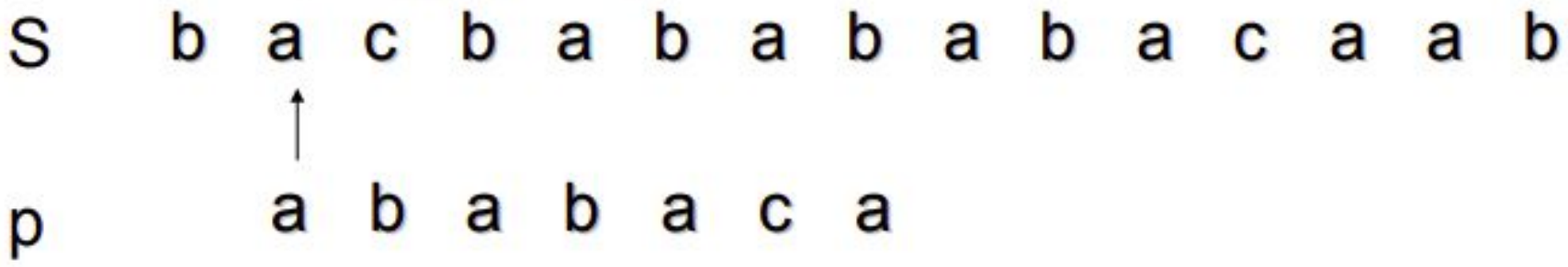
Initially: n = size of S = 15;
m = size of p = 7

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.

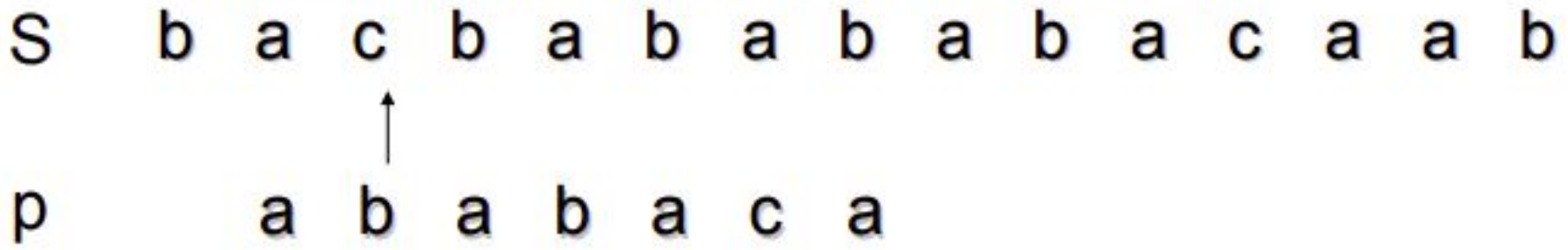
Step 2: i = 2, q = 0
comparing p[1] with S[2]



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

Step 3: i = 3, q = 1

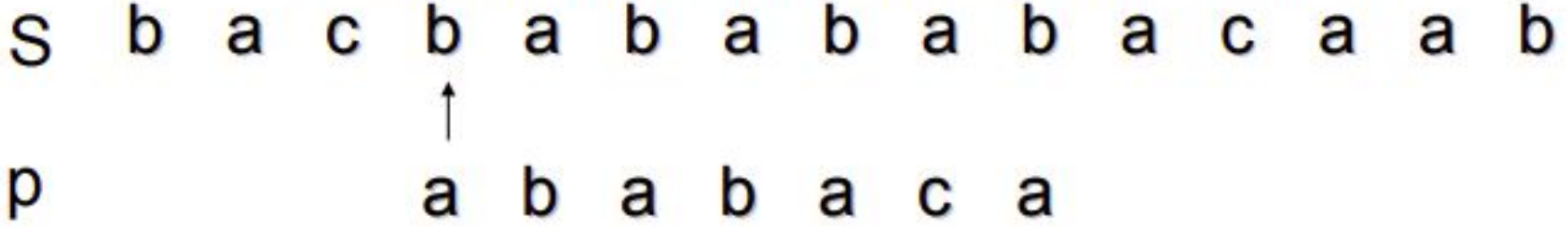
Comparing p[2] with S[3] p[2] does not match with S[3]



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

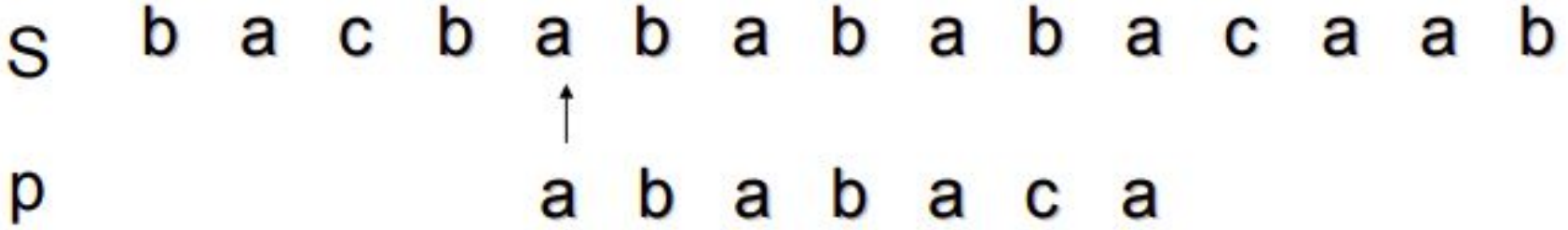
Step 4: i = 4, q = 0

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



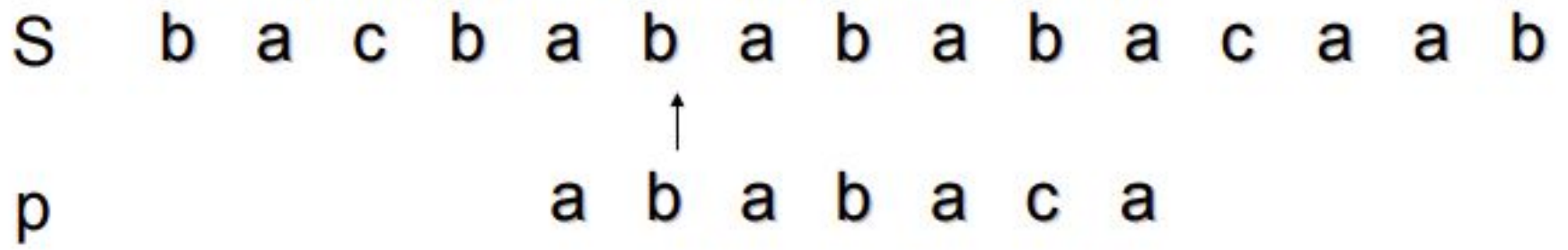
Step 5: i = 5, q = 0

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



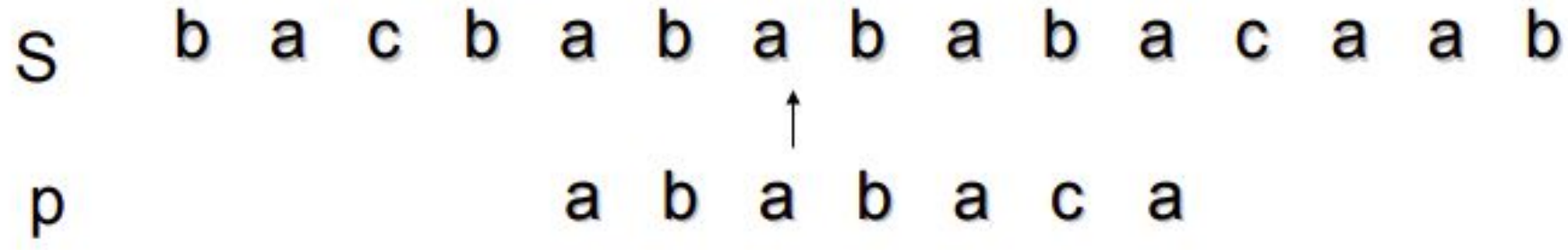
Step 6: i = 6, q = 1

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



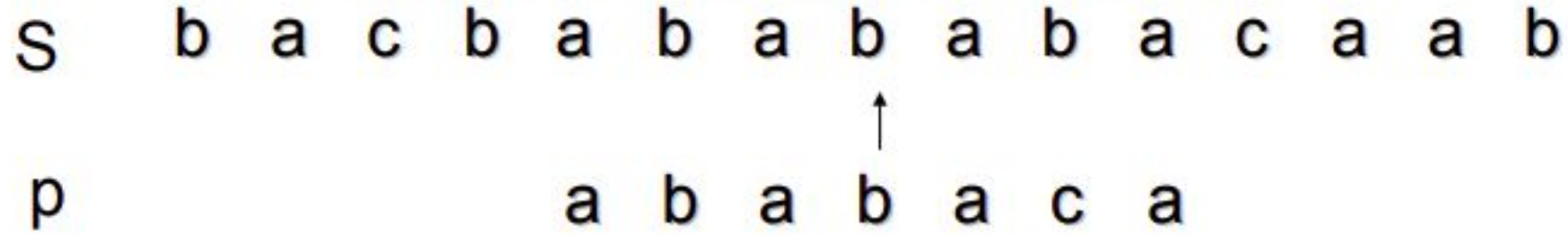
Step 7: i = 7, q = 2

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



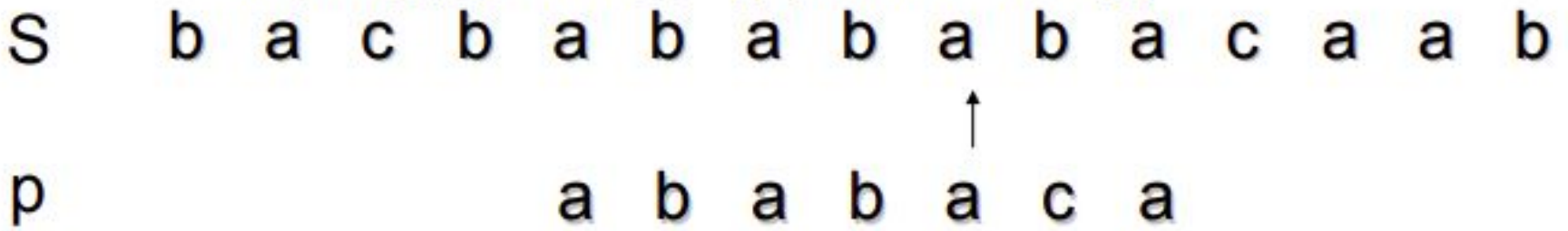
Step 8: i = 8, q = 3

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



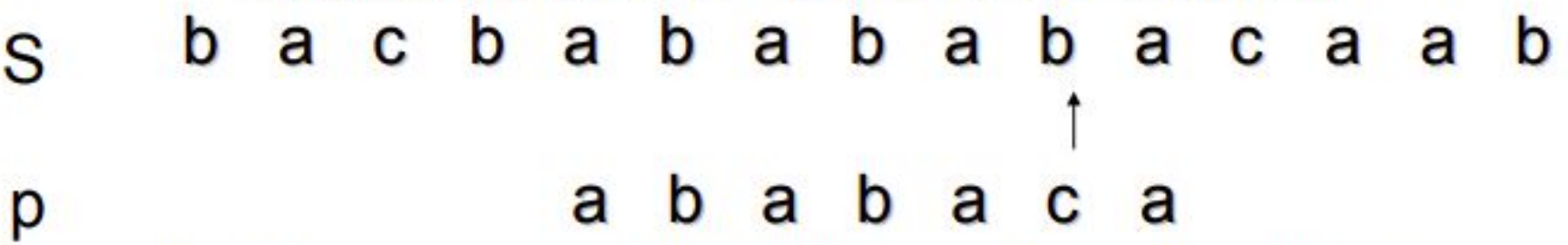
Step 9: i = 9, q = 4

Comparing p[5] with S[9] p[5] matches with S[9]



Step 10: i = 10, q = 5

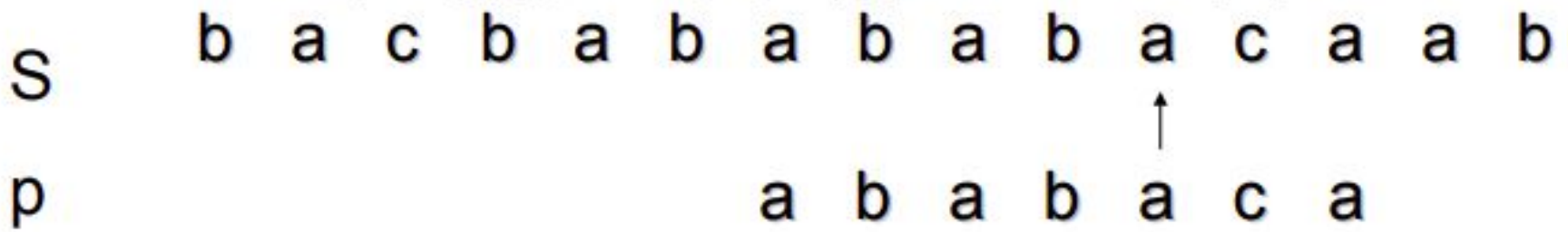
Comparing p[6] with S[10] p[6] doesn't match with S[10]



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

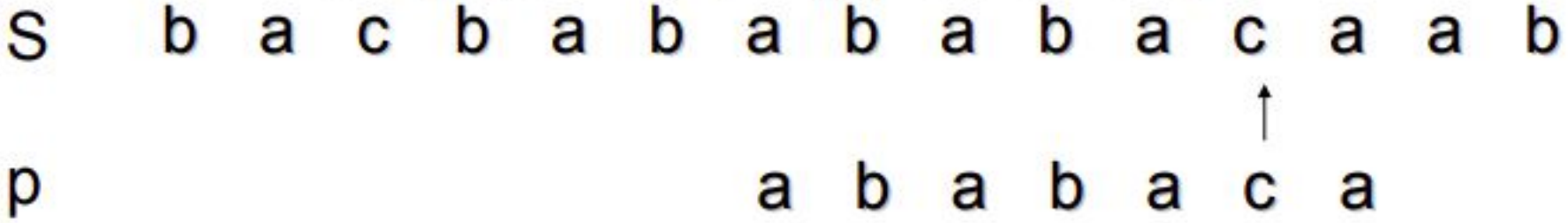
Step 11: i = 11, q = 4

Comparing p[5] with S[11] p[5] matches with S[11]



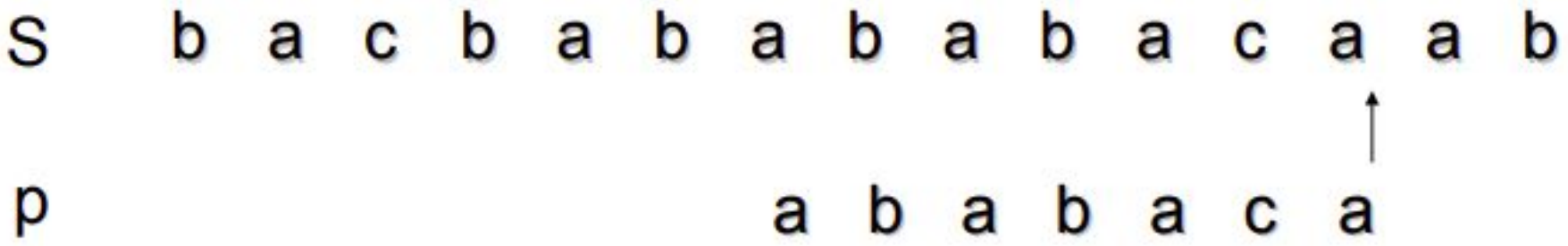
Step 12: i = 12, q = 5

Comparing p[6] with S[12] p[6] matches with S[12]



Step 13: i = 13, q = 6

Comparing p[7] with S[13] p[7] matches with S[13]



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.

```c
// Prints occurrences of txt[] in pat[]
void KMPSearch(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // create lps[] that will hold the longest prefix suffix
    // values for pattern
    int lps[M];

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);

    int i = 0; // index for txt[]
    int j = 0; // index for pat[]
    while (i < N) {
        if (pat[j] == txt[i]) {
            j++;
            i++;
        }

        if (j == M) {
            printf("Found pattern at index %d ", i - j);
            j = lps[j - 1];
        }

        // mismatch after j matches
        else if (i < N && pat[j] != txt[i]) {
            // Do not match lps[0..lps[j-1]] characters,
            // they will match anyway
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
}
```

Time Complexity of KMP -> O(M+N)
M=length of pattern
N=Length of text