Learn DAA: From B K Sharma

Knuth-Morris-Pratt Algorithm

Th String Matching Problem

Given a string 'T', the problem of string matching deals with finding whether a pattern 'P' occurs in 'T' and if 'P' does occur, then returning position in 'T' where 'P' occurs.

The naïve String Matching Algorithms (Brute- Force Algorithm): $O(mn)$

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 'T' in which to locate 'P'.

If the first element of 'P' matches the first element of 'T', compare the second element of 'P' with second element of 'T'.

If match found proceed likewise until entire 'P' is found.

The naïve String Matching Algorithms (Brute- Force Algorithm): $O(mn)$

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

The naïve String Matching Algorithms (Brute- Force Algorithm): $O(mn)$

Alg. NAÏVE-STRING-MATCHER(T,P)

1. n ← length[T]
2. m ← length[P]
3. For s ← 0 to n-m do
4. If P[1…..m]=T[s+1,s=2,……,s+m] then

The naïve algorithm finds all valid shifts using a loop that checks the condition P[1…..m]=T[s+1,s=2,……,s+m] for each of the n-m+1 possible values of s.

The naïve String Matching Algorithms (Brute- Force Algorithm): $O(mn)$

How does the O(mn) approach work?

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

Text T

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

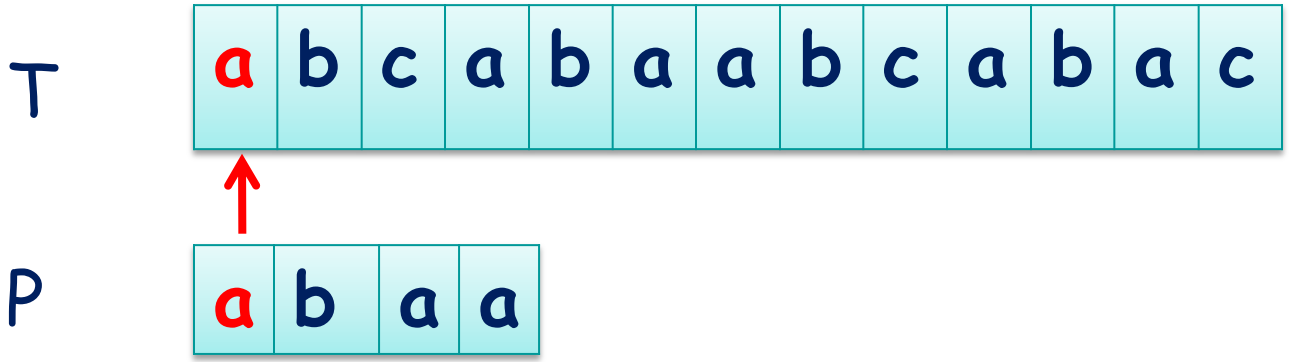
Pattern p

a	b	a	a

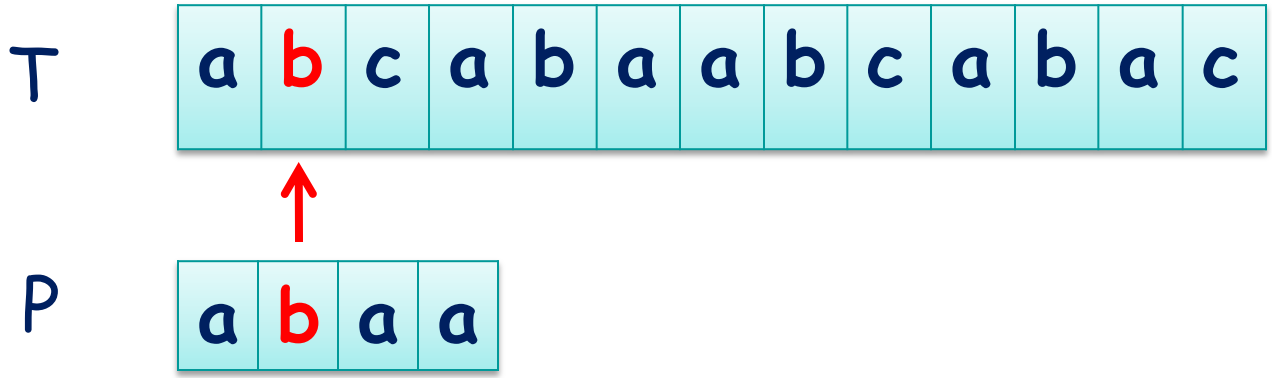
The naïve String Matching Algorithms (Brute- Force Algorithm): $O(mn)$

How does the O(mn) approach work?

Step 1:compare P[1] with T[1]



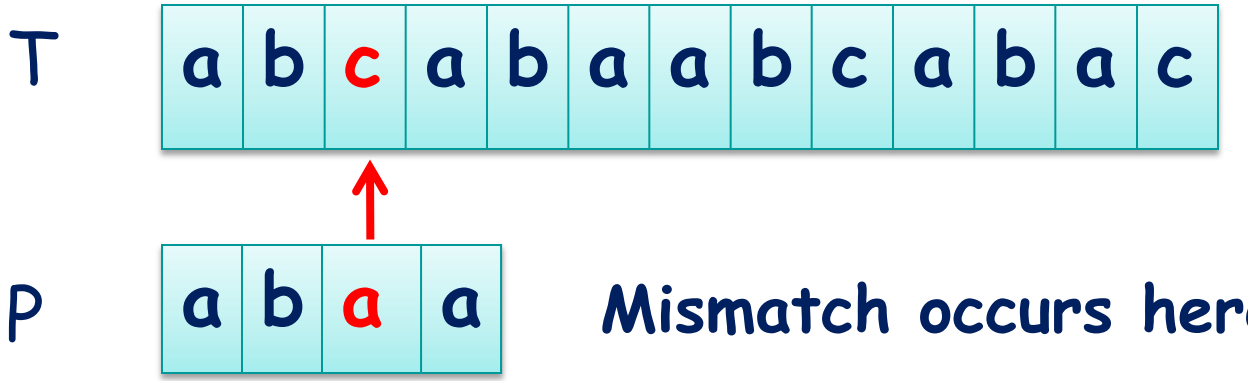
Step 2: compare P[2] with T[2]



The naïve String Matching Algorithms (Brute- Force Algorithm): $O(mn)$

How does the O(mn) approach work?

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



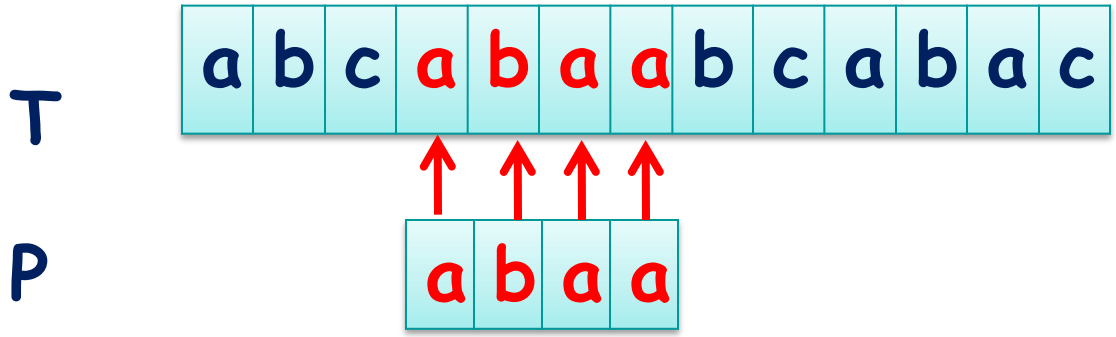
Mismatch occurs here..

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.

The naïve String Matching Algorithms (Brute- Force Algorithm): $O(mn)$

How does the O(mn) approach work?



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

The naïve String Matching Algorithms (Brute- Force Algorithm): $O(mn)$

Drawbacks of this approach:

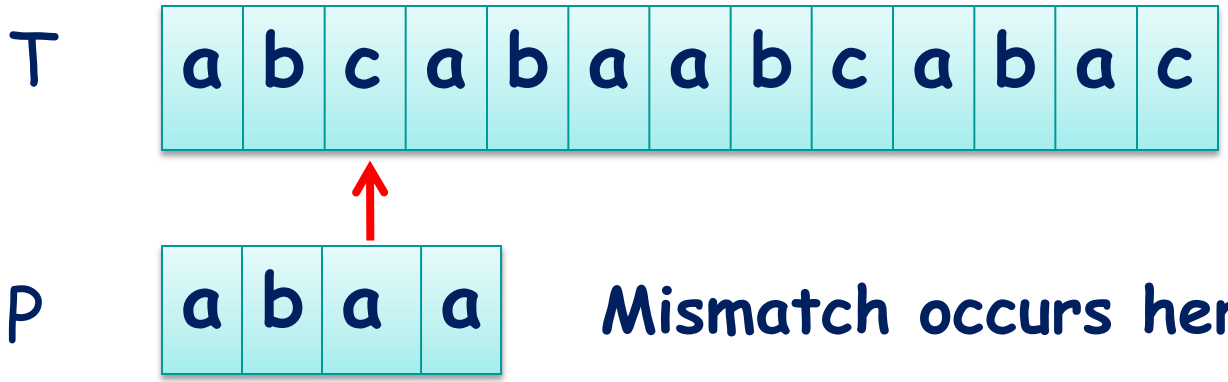
If 'm' is the length of pattern 'P' and 'n' the length of string 'T', the matching time is of the order O(mn). This is a certainly a very slow running algorithm.

What makes this approach so slow is the fact that elements of 'T' with which comparisons had been performed earlier are involved again and again in comparisons in some future iterations.

The naïve String Matching Algorithms (Brute- Force Algorithm): $O(mn)$

Drawbacks of this approach:

For example:



Mismatch occurs here..

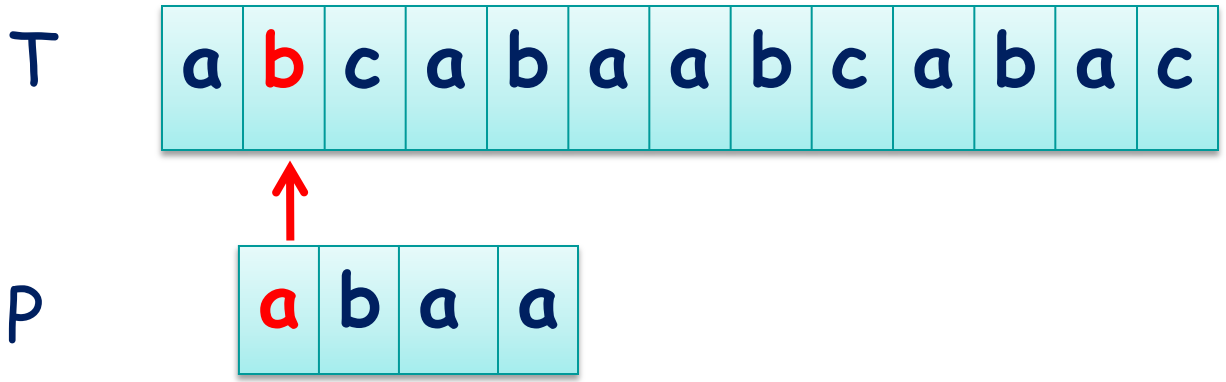
When mismatch is detected for the first time in comparison of P[3] with T[3], pattern 'P' would be moved one position to the right and matching procedure would resume from here.

Here the first comparison that would take place would be between P[0]='a' and T[1]='b'.

The naïve String Matching Algorithms (Brute- Force Algorithm): $O(mn)$

Drawbacks of this approach:

For example:



It should be noted here that T[1]='b' had been previously involved in a comparison in step 2. This is a repetitive use of T[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 comparisons with elements of 'T' that have previously been involved in comparison with some element of the pattern 'P' to be matched. i.e., backtracking on the string 'T' 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 'T'.

Components of KMP Algorithm

The KMP Matcher

With string 'T', pattern 'P' and prefix function 'Π' as inputs, finds the occurrence of 'P' in 'T' and returns the number of shifts of 'P' after which occurrence is found.

The prefix function, Π

Alg. **Compute-Prefix-Function (P)**

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

Learn DAA : From B K Sharma

The prefix function, Π

Example: compute Π for the pattern 'p' below:

Initially:

m = length[p] = 7

Π[1] = 0

k = 0

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					

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

The prefix function, Π

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

Alg. KMP-Matcher(T,P)

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

Note: KMP finds every occurrence of a 'P' in 'T'. That is why KMP does not terminate in step 12, rather it searches remainder of 'T' for any more occurrences of 'P'.

The KMP Matcher

Illustration: 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

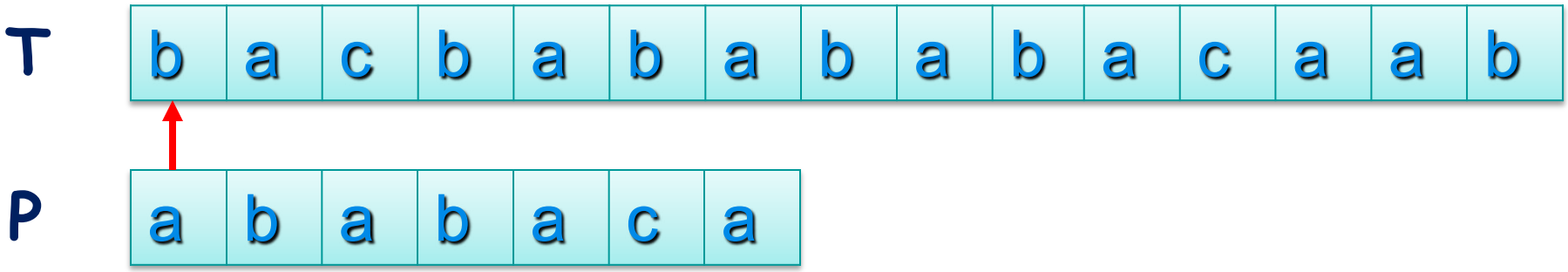
Let us execute the KMP algorithm to find whether 'p' occurs in 'S'.

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

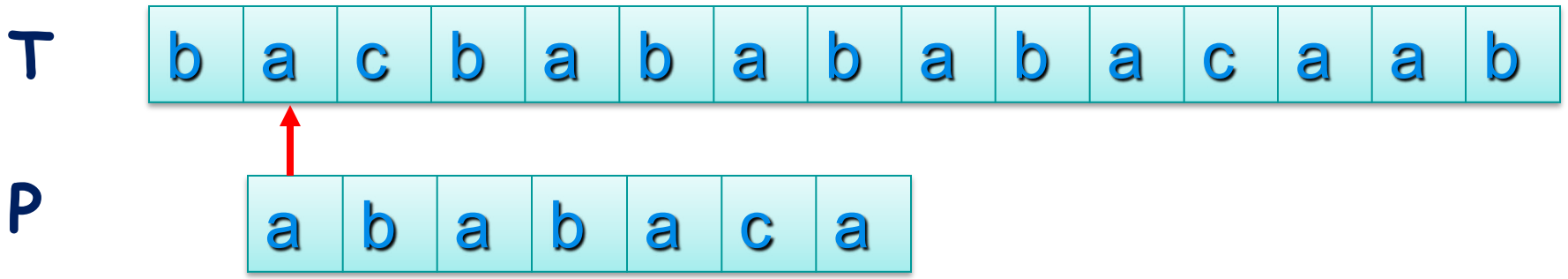
Initially: n = size of S = 15;
m = size of p = 7

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



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

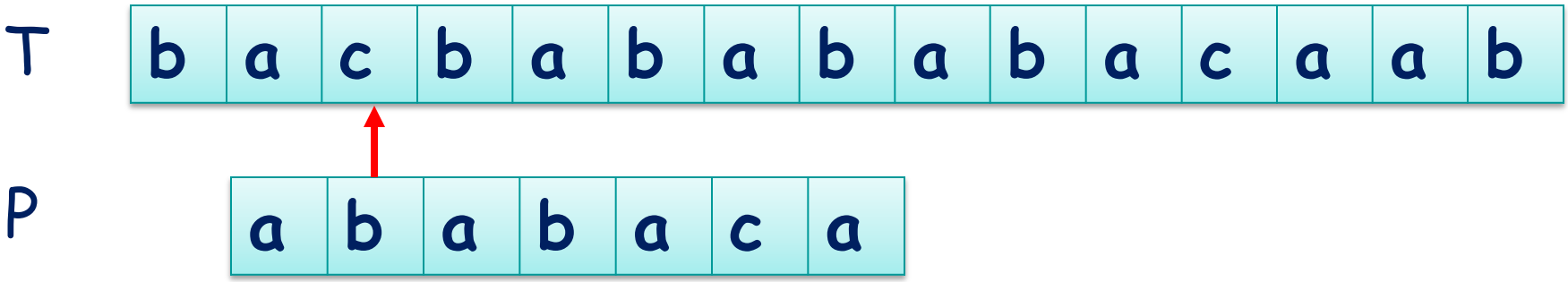
Step 2: i = 2, q = 0
comparing P[1] with T[2]



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

Step 3: i = 3, q = 1

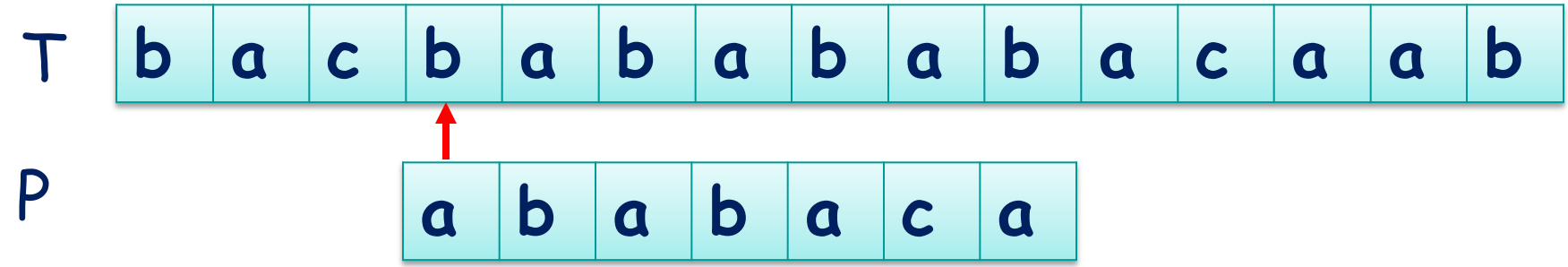
Comparing P[2] with T[3]P[2] does not match with T[3]



Backtracking on P, comparing P[1] and T[3]

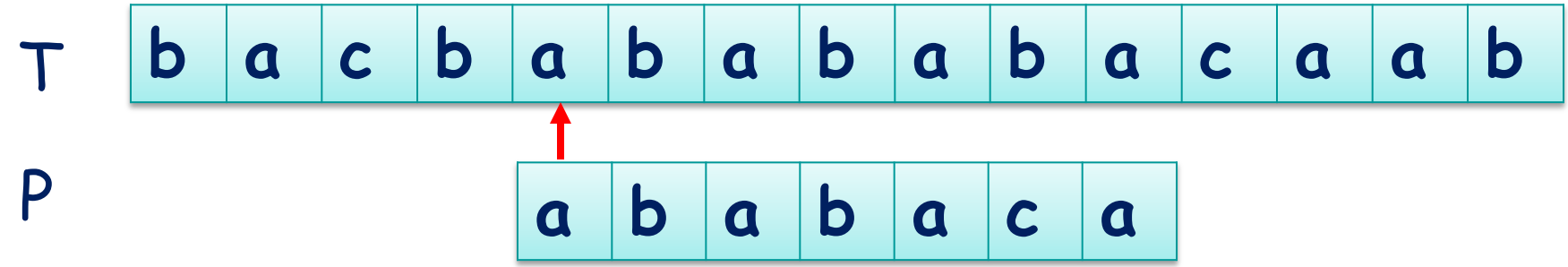
Step 4: i = 4, q = 0

comparing P[1] with T[4] P[1] does not match with T[4]



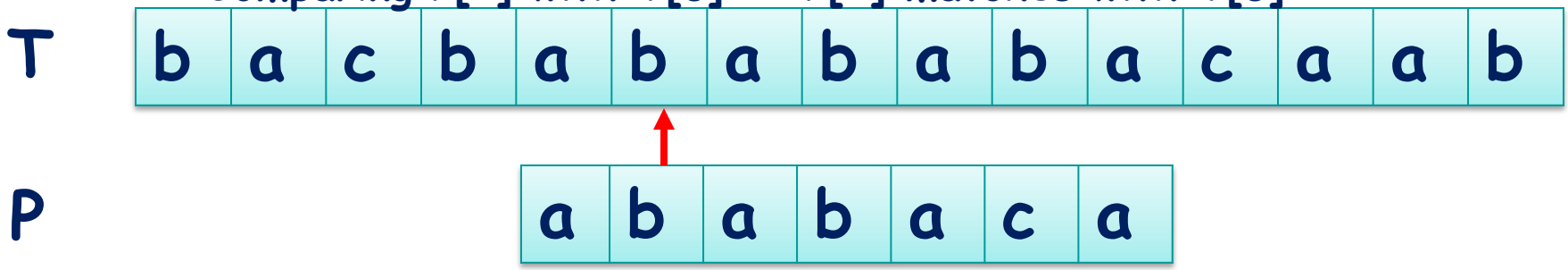
Step 5: i = 5, q = 0

comparing P[1] with T[5] P[1] matches with T[5]



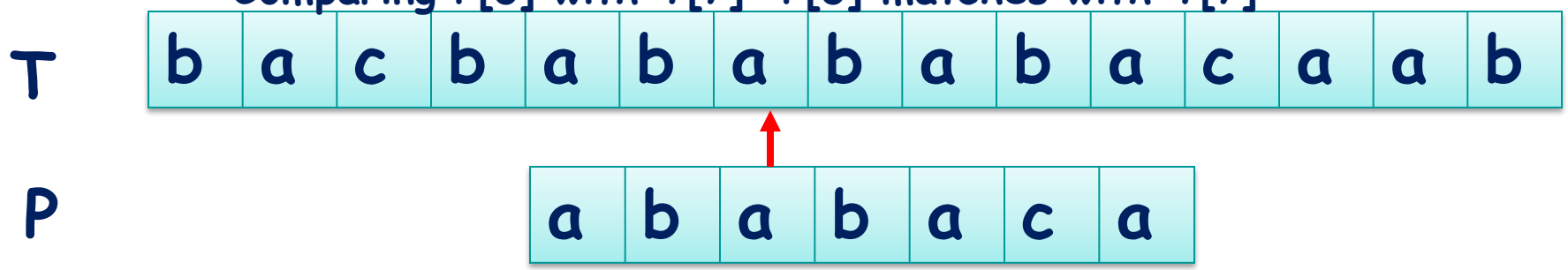
Step 6: i = 6, q = 1

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



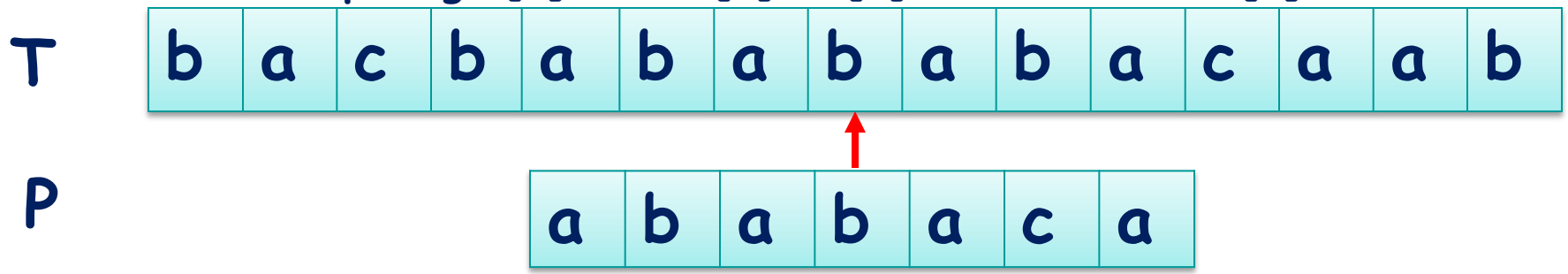
Step 7: i = 7, q = 2

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



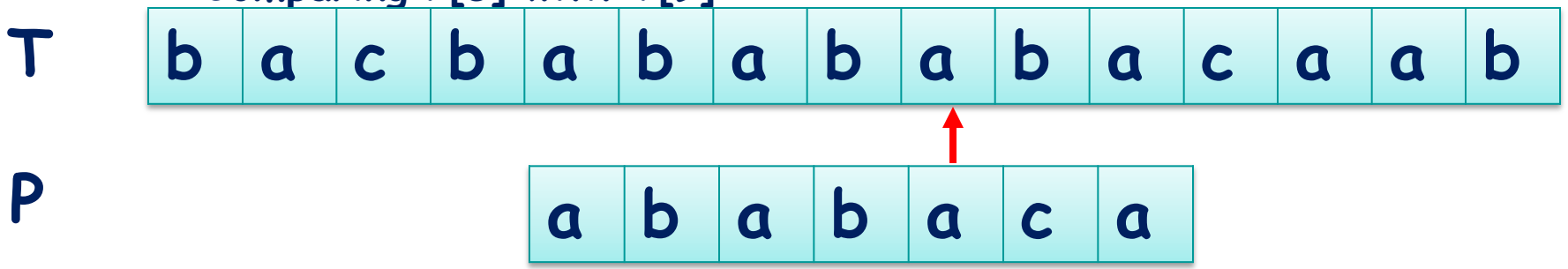
Step 8: i = 8, q = 3

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



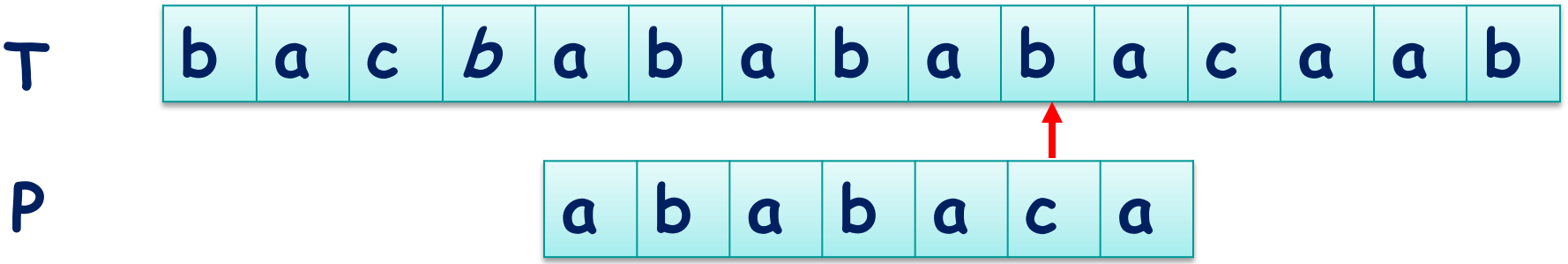
Step 9: i = 9, q = 4

Comparing P[5] with T[9] P5] matches with T[9]



Step 10: i = 10, q = 5

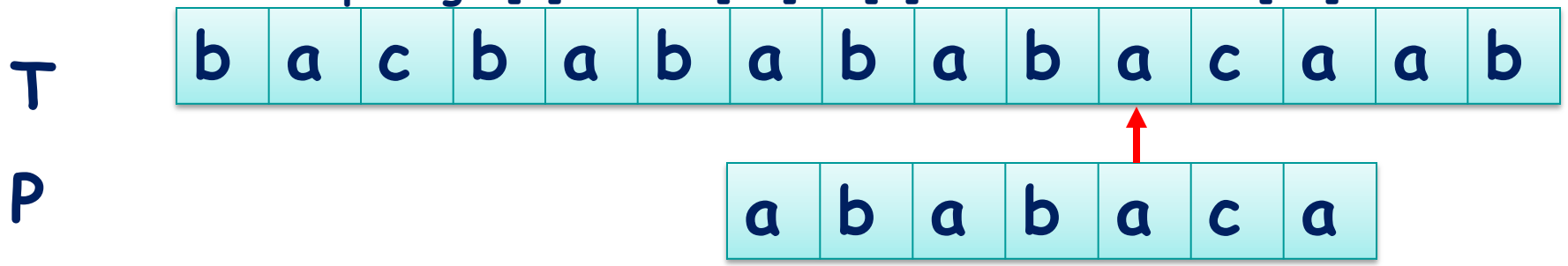
Comparing P[6] with T[10] P[6] doesn't match with T[10]



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

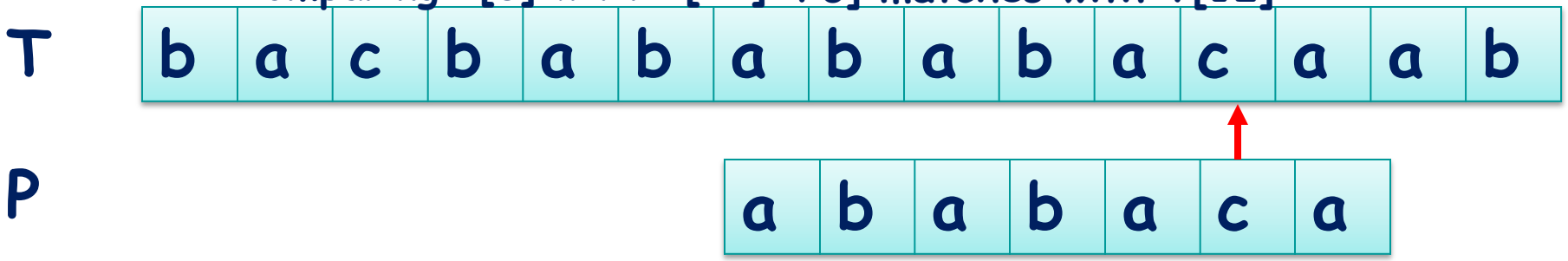
Step 11: i = 11, q = 4

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



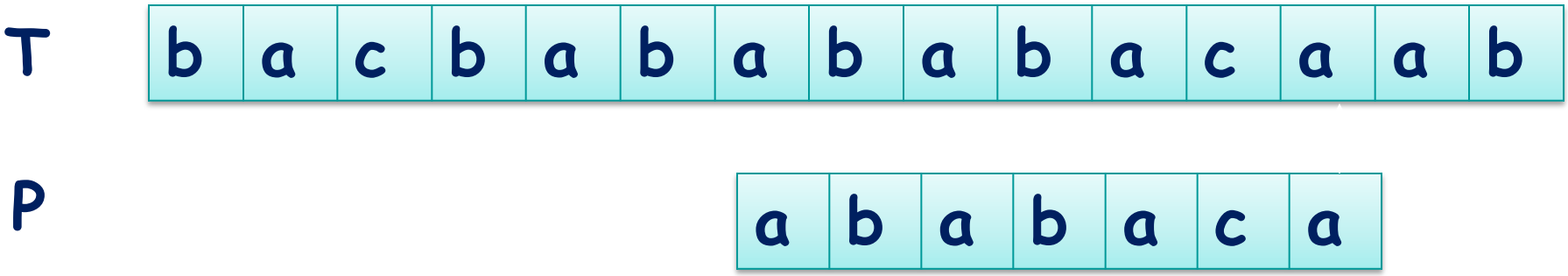
Step 12: i = 12, q = 5

Comparing P[6] with T[12] P6] matches with T[12]



Step 13: i = 13, q = 6

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



Pattern 'P' has been found to completely occur in string 'T'. 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)

```
1  m ← length[]      //'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
```

In the above pseudocode for computing the prefix function, the for loop from step 4 to step 10 runs 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[T]
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] != T[i]
7           do  q ← Π[q]
8      if P[q+1] = T[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).