T&R Team of Algorithm Design
College of Computer Science and Engineering, CQU

Algorithm Analysis & Design

Introduction to Algorithm

research.nii.ac.jp

Home Problem Rules Ranking Submit Events Q&A About

Graph Golf

The Order/degree Problem Competition

Problem statement

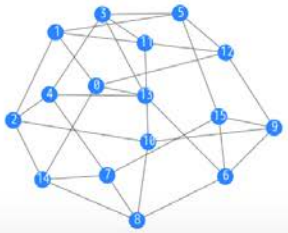
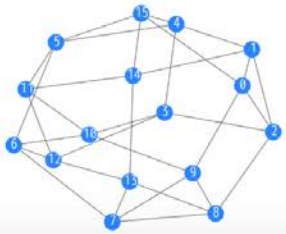
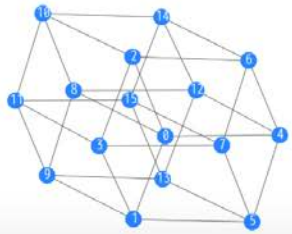
Update 2016-05-25

Definition

The order/degree problem with parameters n and d: Find a graph with minimum diameter over all undirected graphs with the number of vertices = n and degree ≤ d. If two or more graphs take the minimum diameter, a graph with minimum average shortest path length (ASPL) over all graphs with minimum diameter must be found.

Example

Let us show examples for the order/degree problem with parameters n = 16 and d = 4. The figure illustrates three examples of a 4-regular graph with 16 nodes. The diameter of the first and the second graphs is 3, while that of the third one is 4. Thus, we should select the first or the second one. Next, we compare the average shortest path length (ASPL) of these two graphs. Since that of the first one is smaller, this is the best solution of the problem among the three.

	1	2	3
Graph			

STRING MATCHING

- Introduction
- Naïve Algorithm
- Rabin-Karp Algorithm
- String Matching using Finite Automata
- Knuth-Morris-Pratt (KMP) Algorithm

- Literature
  - Cormen, Leiserson, Rivest, “Introduction to Algorithms”, chapter 32, string matching, The MIT Press, 2001, 906-932.

Introduction

- What is *string matching*?
  - Finding all occurrences of a *pattern* in a given *text* (or *body of text*)
- Many applications
  - While using editor/word processor/browser
  - Login name & password checking
  - Virus detection
  - Header analysis in data communications
  - DNA sequence analysis

History of String Search

- The brute force algorithm:
  - invented in the dawn of computer history
  - re-invented many times, *still* common
- Knuth & Pratt invented a better one in 1970
  - published 1976 as “Knuth-Morris-Pratt”
- Boyer & Moore found a better one before 1976
  - Published 1977
- Karp & Rabin found a “better” one in 1980
  - Published 1987

History of String Search

- The brute force algorithm:
  - invented in the dawn of computer history
  - re-invented many times, *still* common
- Knuth & Pratt invented a better one in 1970
  - published 1976 as “Knuth-Morris-Pratt”
- Boyer & Moore found a better one before 1976
  - Published 1977
- Karp & Rabin found a “better” one in 1980
  - Published 1987

Algorithm	Preprocessing Time	Matching Time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite Automaton	$O(m\|\Sigma\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$
Boyer-Moore	$\Theta(m)$	$\Theta(n)$

String-Matching Problem

- The *text* is in an array $T[1..n]$ of length n
- The *pattern* is in an array $P[1..m]$ of length m
- Elements of T and P are characters from a *finite alphabet* Σ
  - E.g., $\Sigma = \{0,1\}$ or $\Sigma = \{a, b, \ldots, z\}$
- Usually T and P are called *strings* of characters

String-Matching Problem …contd

- We say that pattern *P* *occurs with shift s* in text *T* if:

  a) $0 \le s \le n-m$ and

  b) $T[(s+1)..(s+m)] = P[1..m]$

- If *P* occurs with shift *s* in *T*, then *s* is a *valid shift*, otherwise *s* is an *invalid shift*

String-Matching Problem …contd

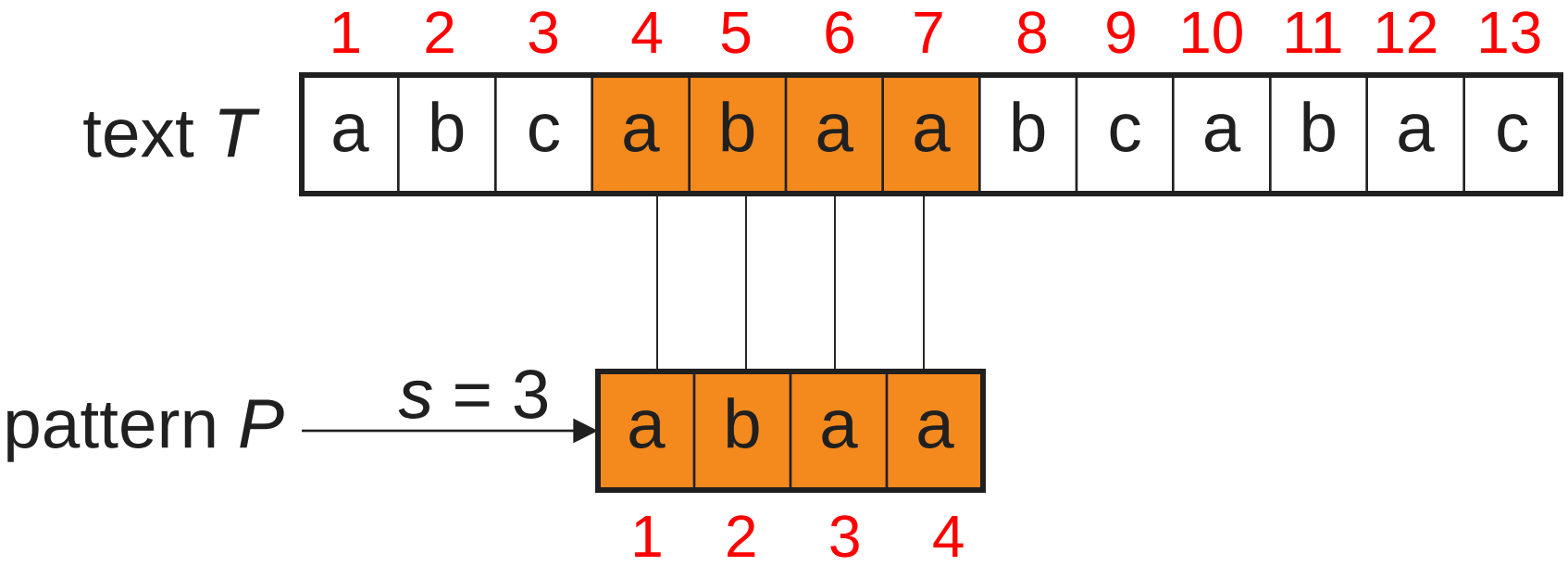
- We say that pattern P *occurs with shift* s in text T if:

  a) $0 \leq s \leq n-m$ and

  b) $T[(s+1)..(s+m)] = P[1..m]$

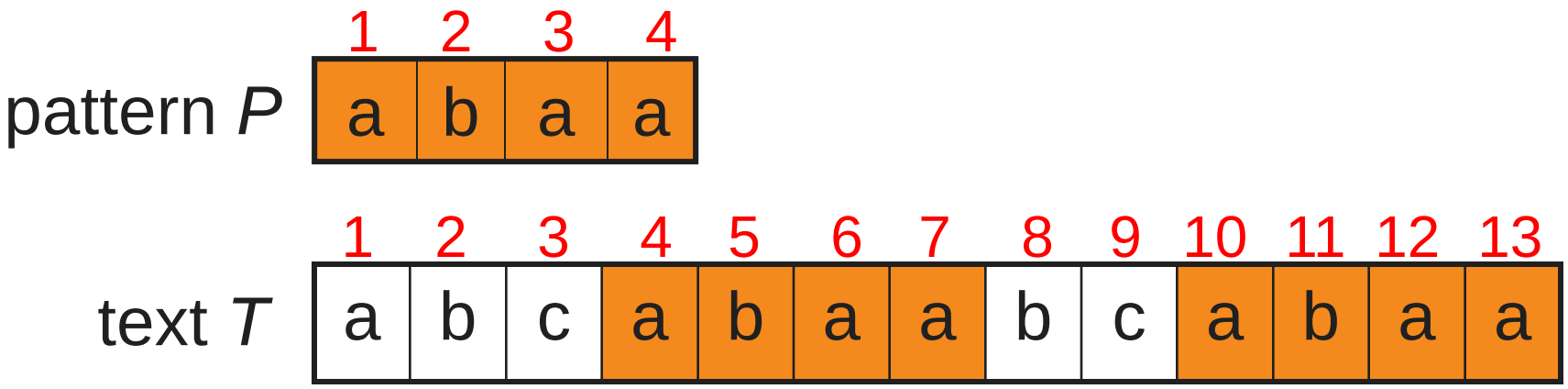
- If P occurs with shift s in T, then s is a *valid shift*, otherwise s is an *invalid shift*
- String-matching problem: finding all valid shifts for a given T and P

Example 1

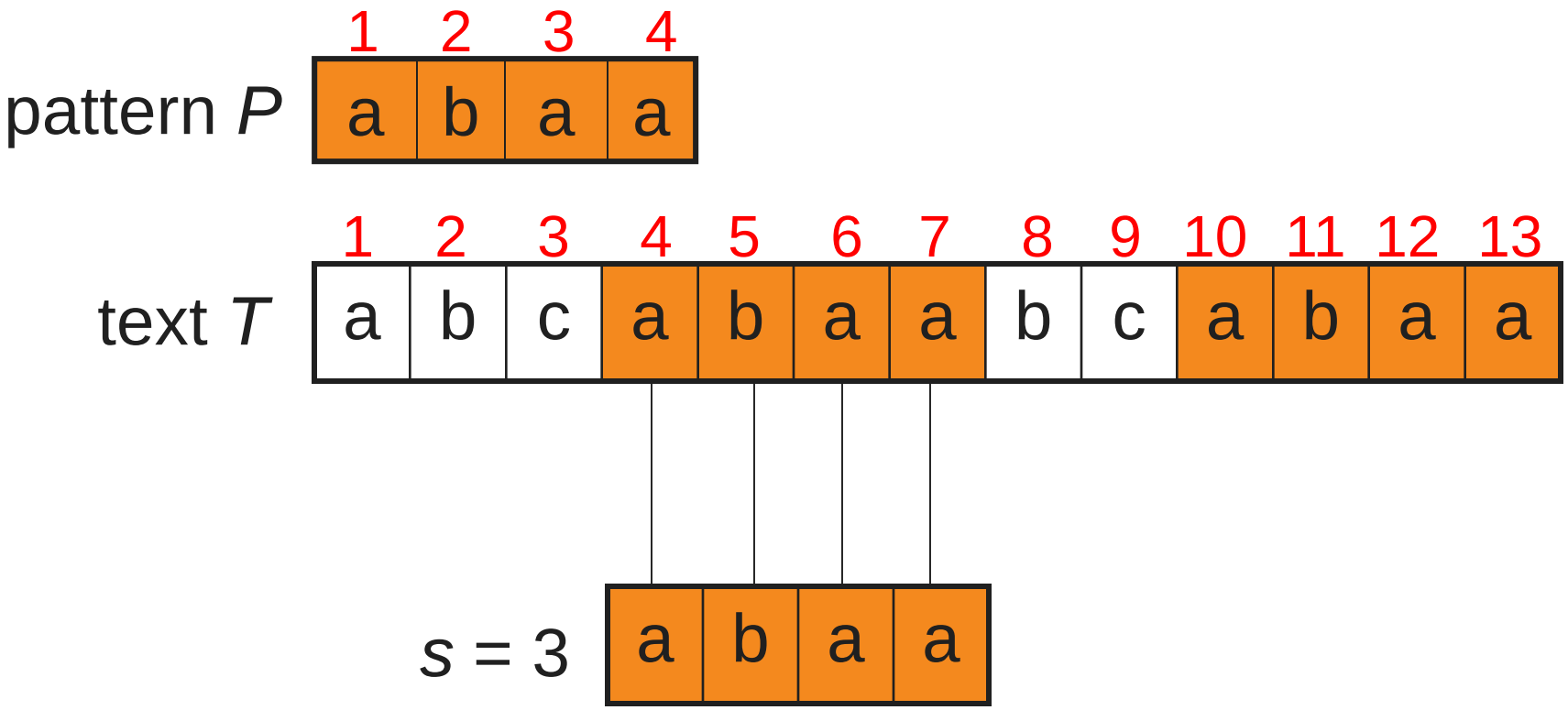


shift $s = 3$ is a valid shift
(n=13, m=4 and $0 \leq s \leq n\text{-}m$ holds)

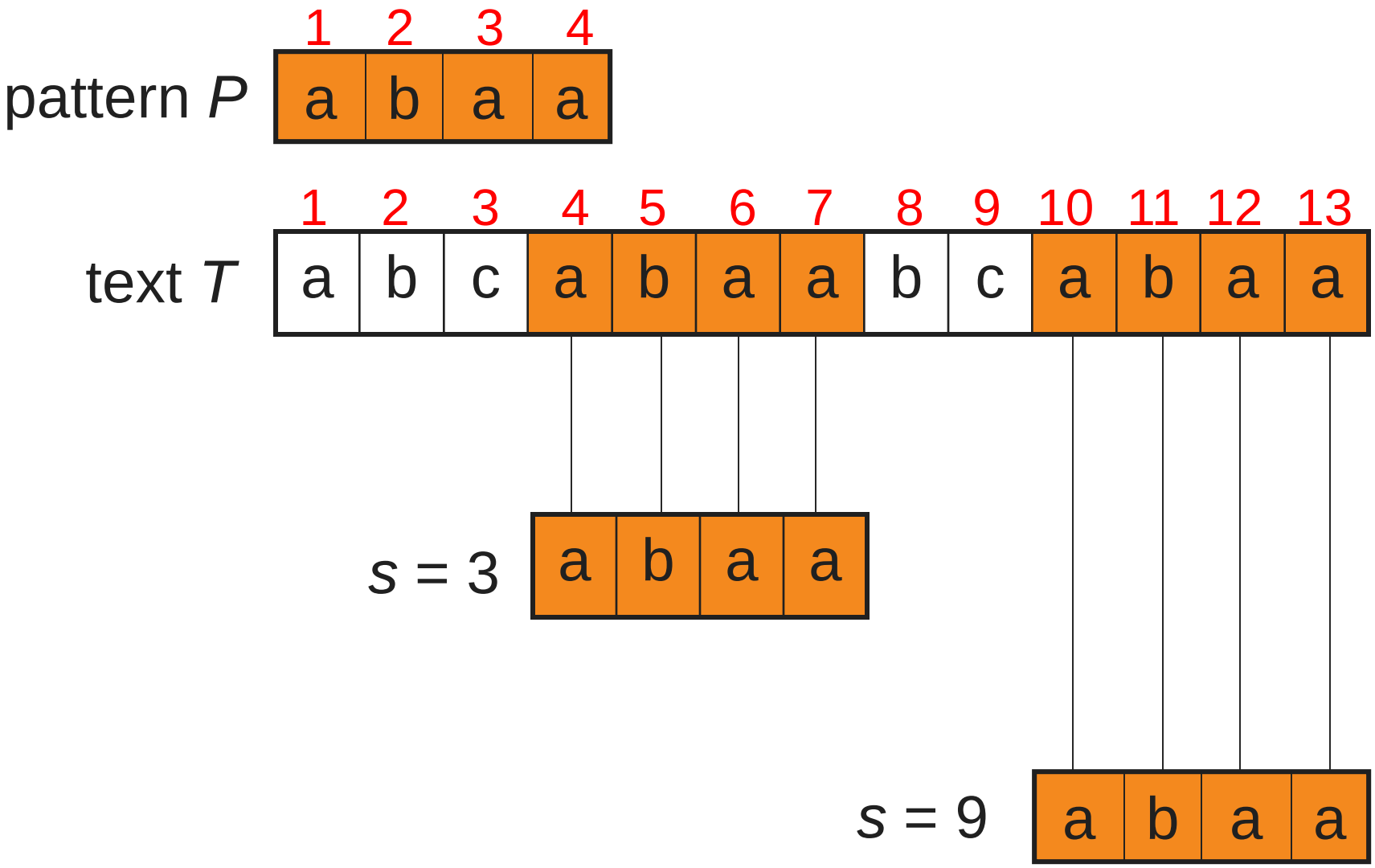
Example 2


1 2 3 4
pattern P
a b a a
1 2 3 4 5 6 7 8 9 10 11 12 13
text T
a b c a b a a b c a b a a

Example 2


1 2 3 4
pattern P
a b a a
1 2 3 4 5 6 7 8 9 10 11 12 13
text T
a b c a b a a b c a b a a
s = 3
a b a a

Example 2


1 2 3 4
pattern P
a b a a
1 2 3 4 5 6 7 8 9 10 11 12 13
text T
a b c a b a a b c a b a a
s = 3
a b a a
s = 9
a b a a

Naïve String-Matching Algorithm

Input: Text strings $T[1..n]$ and $P[1..m]$

Result: All valid shifts displayed

NAÏVE-STRING-MATCHER (T, P)

```
n ← length[T]
m ← length[P]
for s ← 0 to n-m
  if P[1..m] = T[(s+1)..(s+m)]
        print "pattern occurs with shift" s
```

Example

P=“abxyabxz” and T=“xabxyabxyabxz”

Example

P="abxyabxz" and T="xabxyabxyabxz"

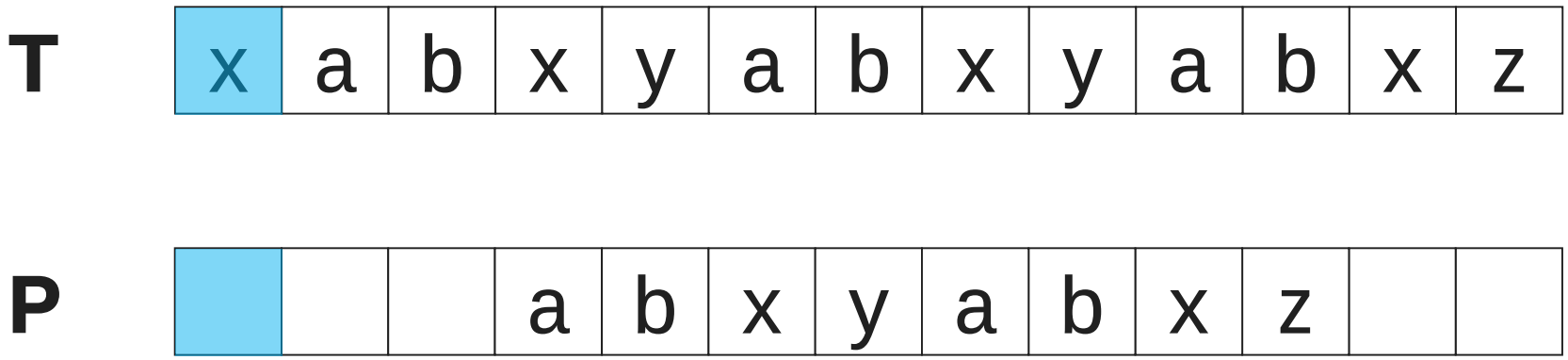
T

x	a	b	x	y	a	b	x	y	a	b	x	z

P

Example

P="abxyabxz" and T="xabxyabxyabxz"



Example

P="abxyabxz" and T="xabxyabxyabxz"

T

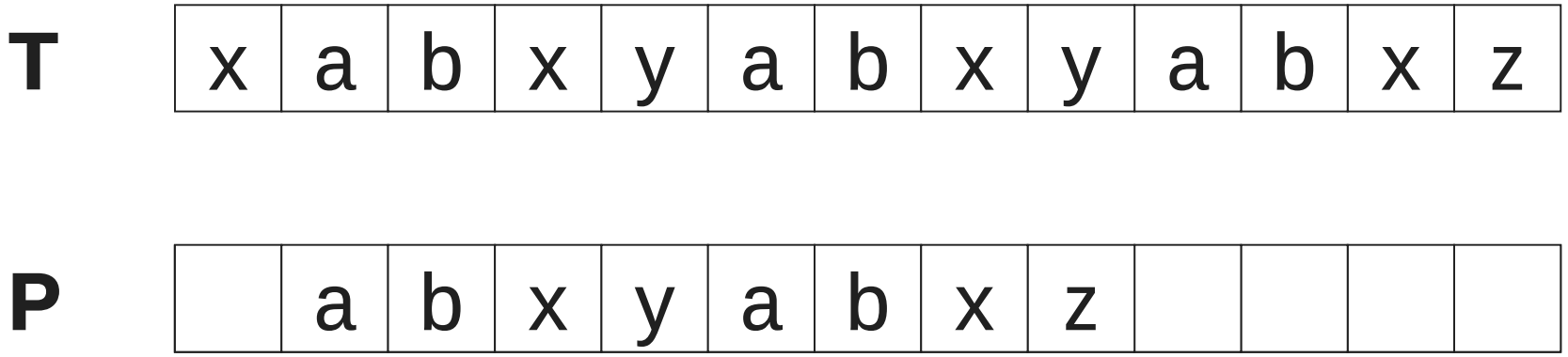
x	a	b	x	y	a	b	x	y	a	b	x	z

P

a	b	x	y	a	b	x	z					

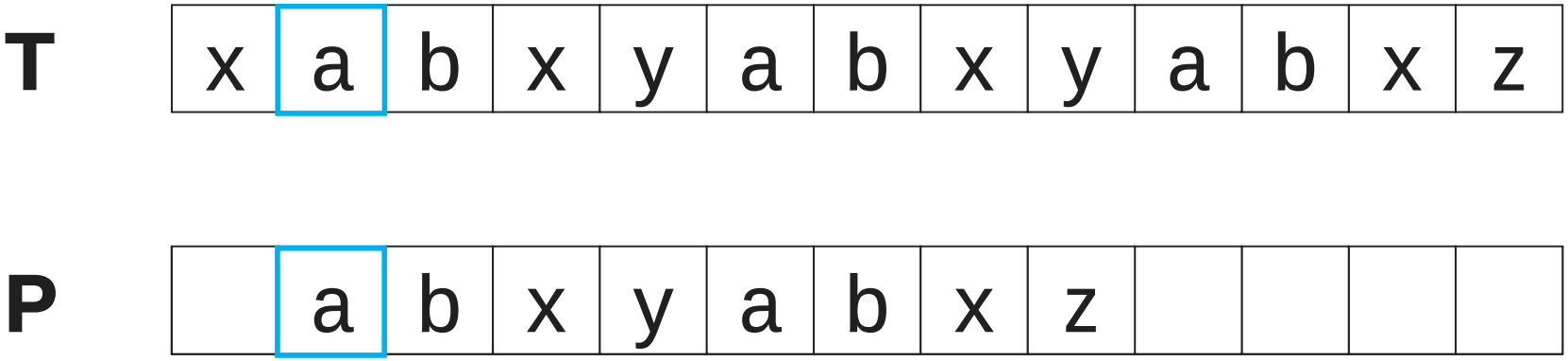
Example

P="abxyabxz" and T="xabxyabxyabxz"



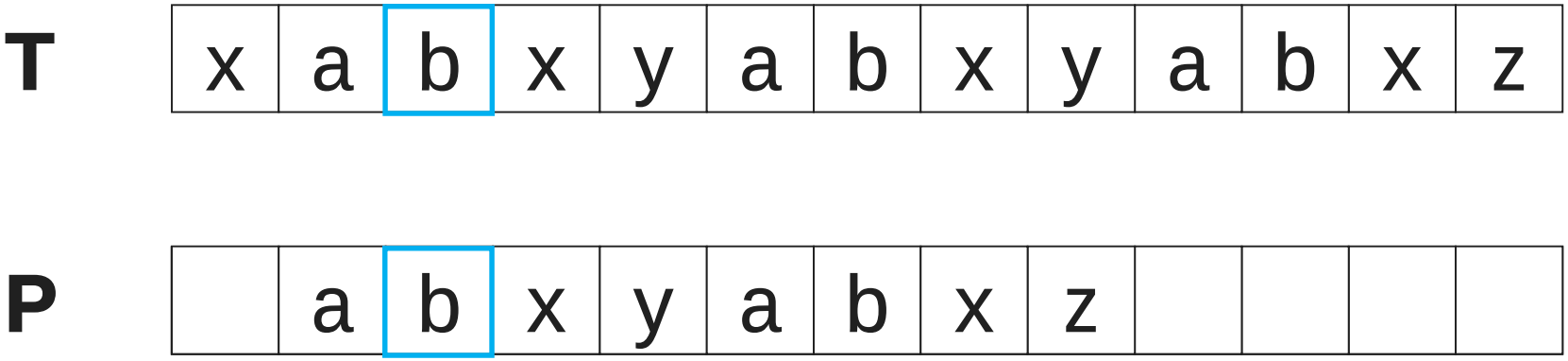
Example

P="abxyabxz" and T="xabxyabxyabxz"



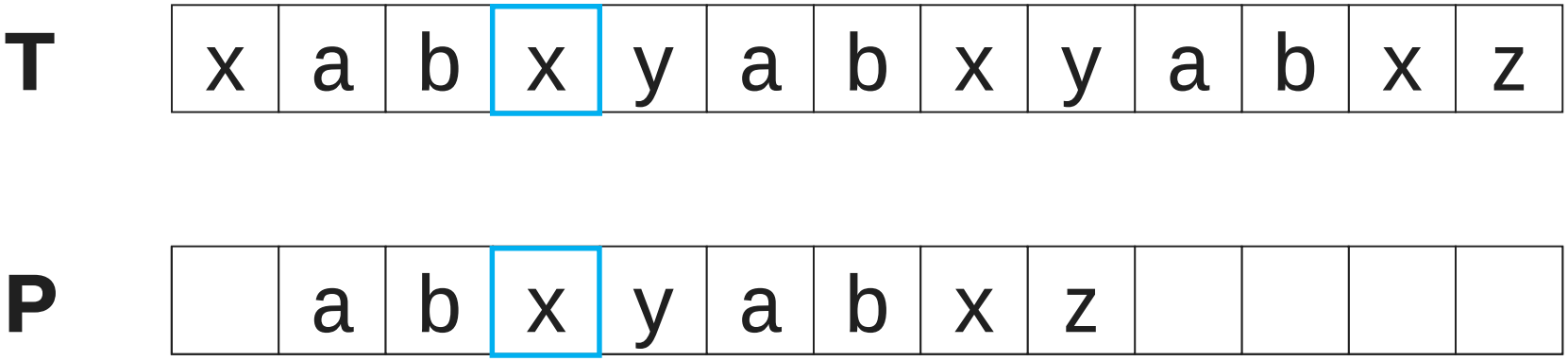
Example

P="abxyabxz" and T="xabxyabxyabxz"



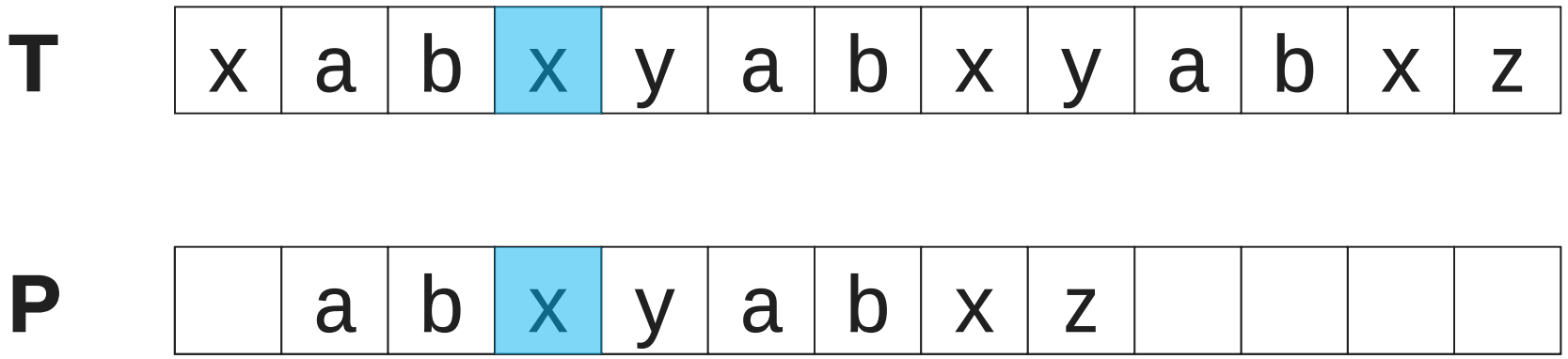
Example

P="abxyabxz" and T="xabxyabxyabxz"



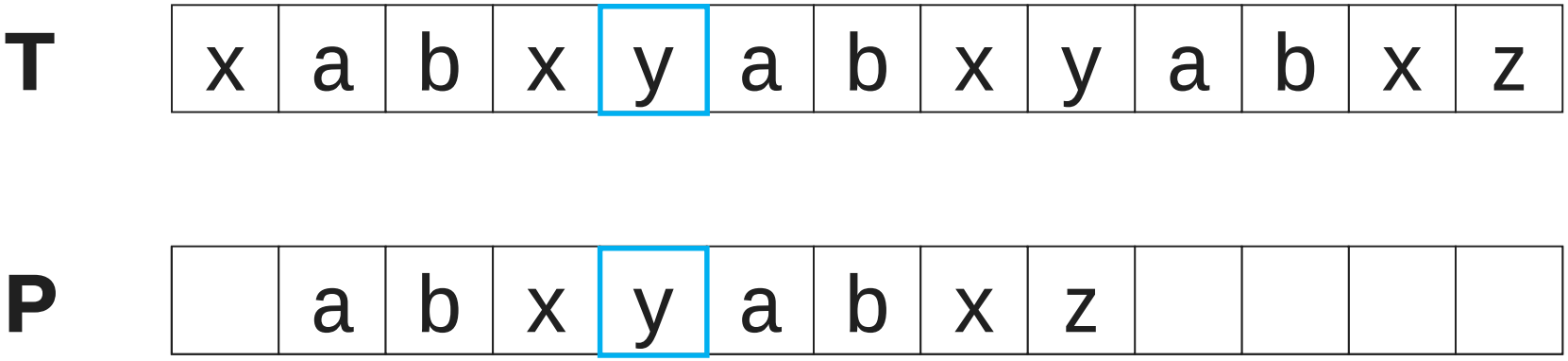
Example

P="abxyabxz" and T="xabxyabxyabxz"



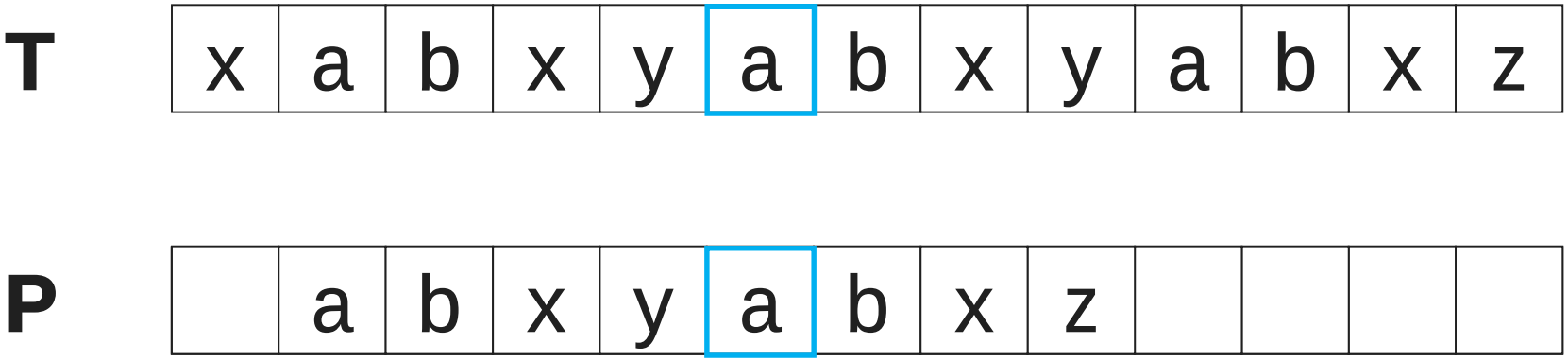
Example

P="abxyabxz" and T="xabxyabxyabxz"



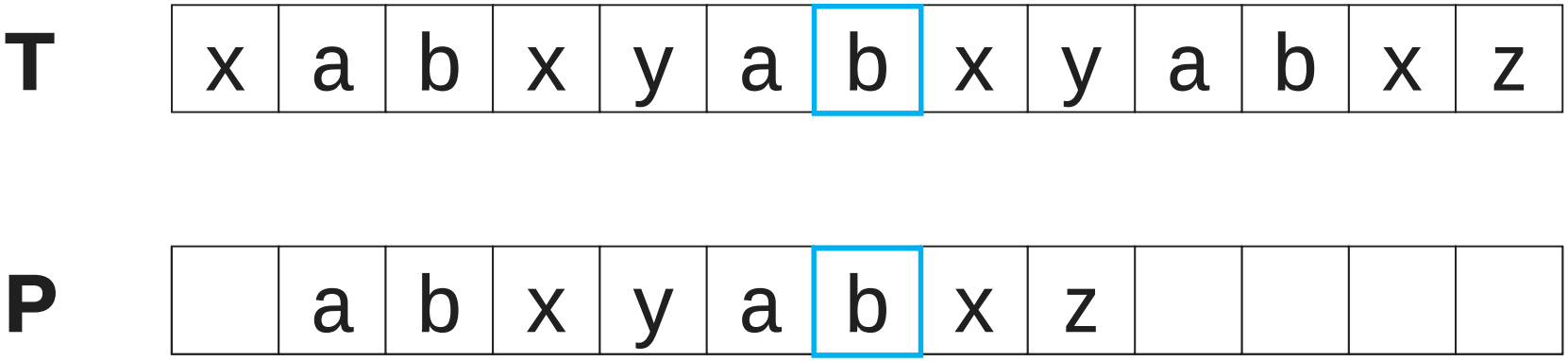
Example

P="abxyabxz" and T="xabxyabxyabxz"



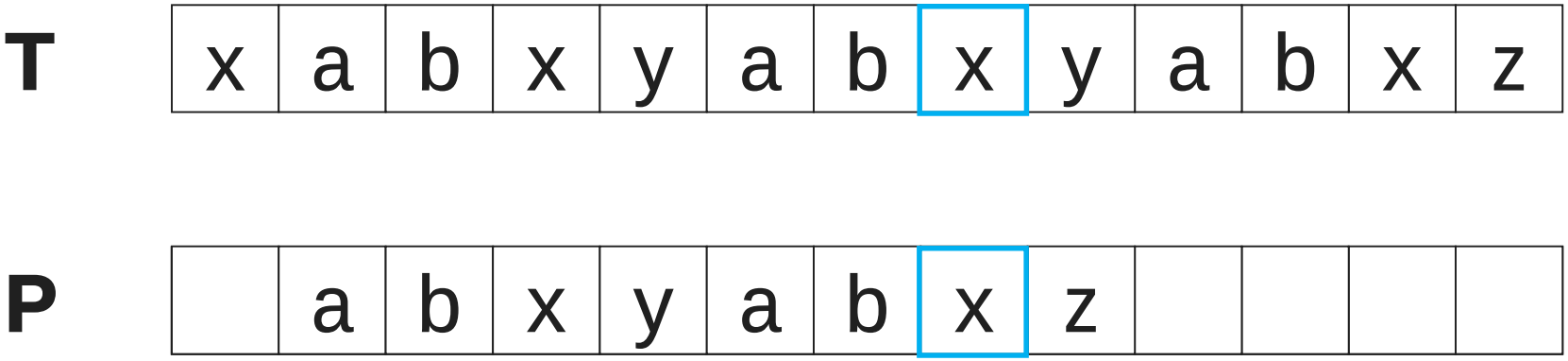
Example

P="abxyabxz" and T="xabxyabxyabxz"



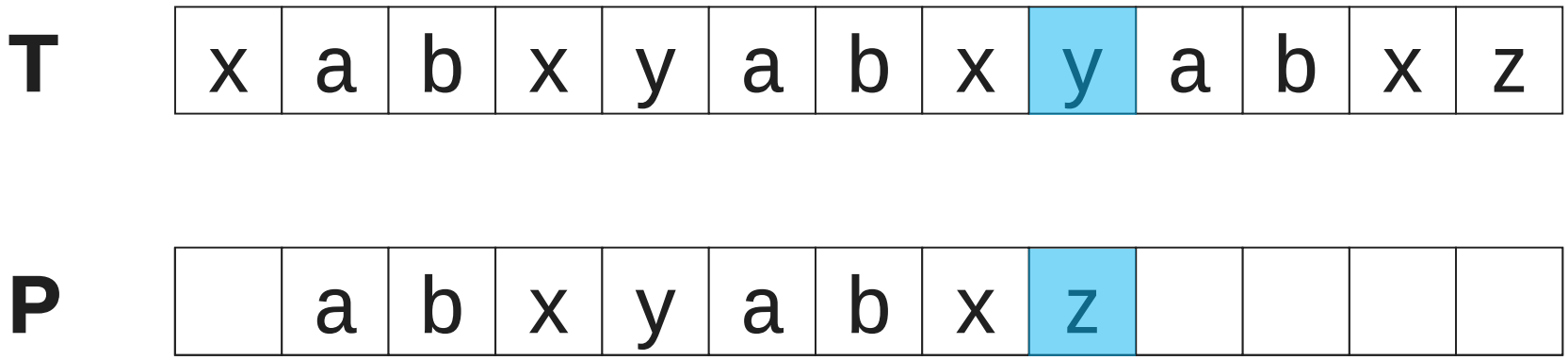
Example

P="abxyabxz" and T="xabxyabxyabxz"



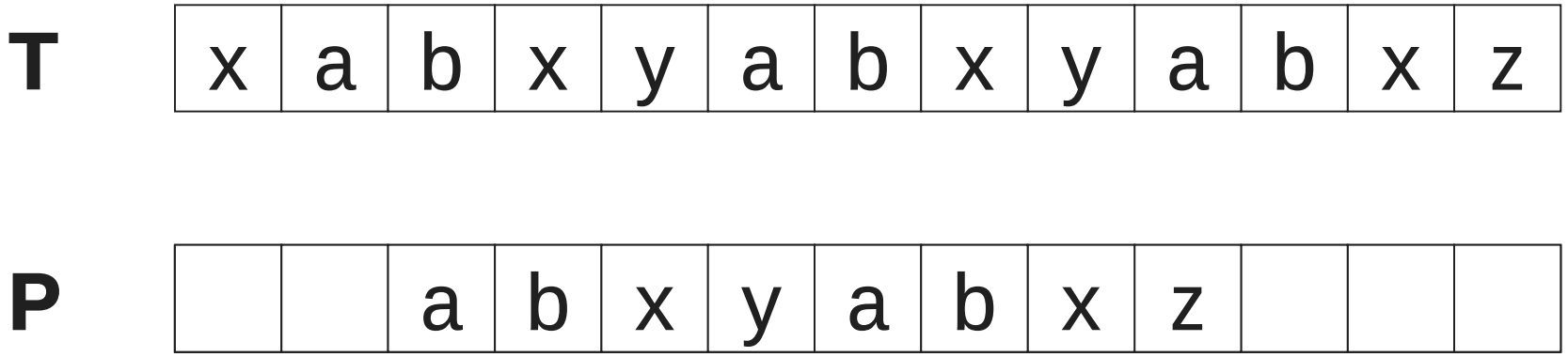
Example

P="abxyabxz" and T="xabxyabxyabxz"



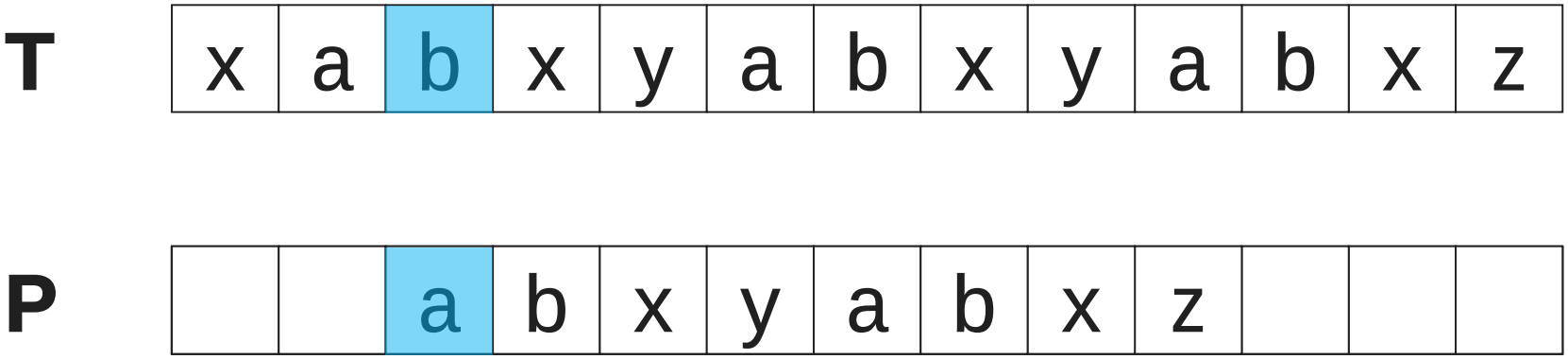
Example

P="abxyabxz" and T="xabxyabxyabxz"



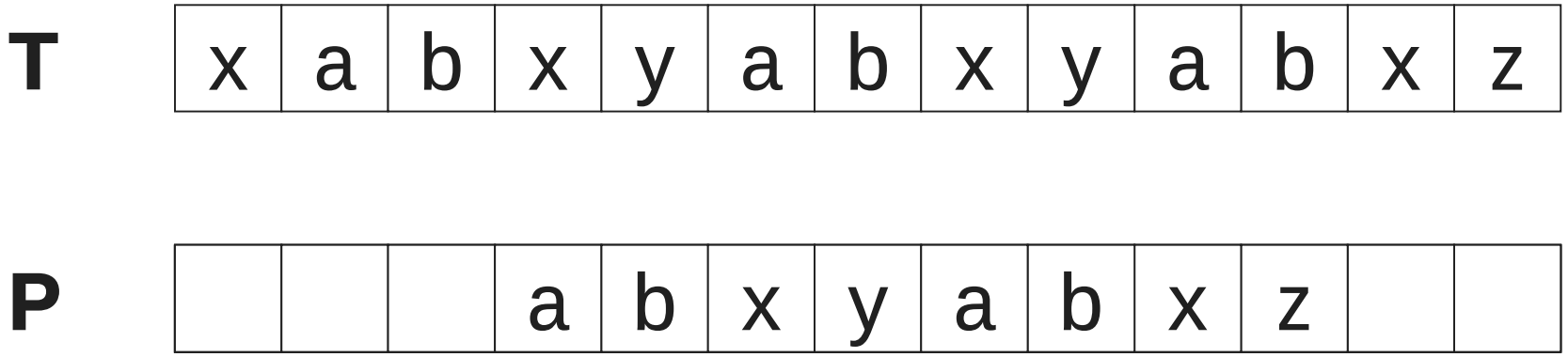
Example

P="abxyabxz" and T="xabxyabxyabxz"



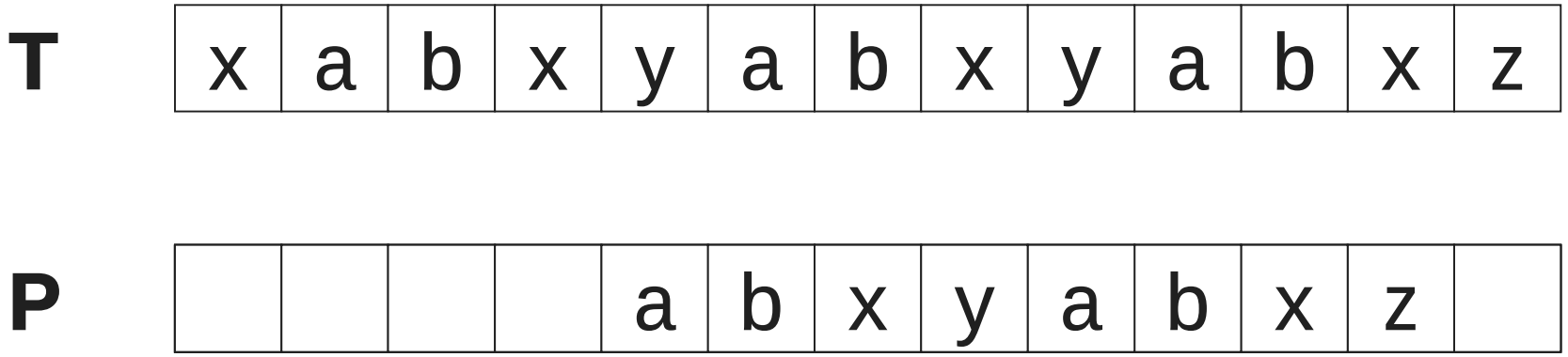
Example

P="abxyabxz" and T="xabxyabxyabxz"



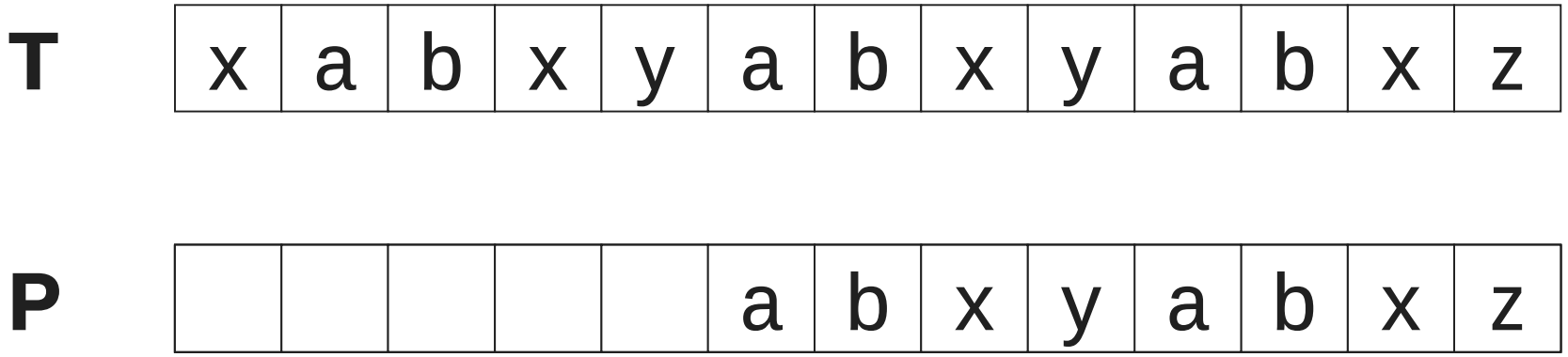
Example

P="abxyabxz" and T="xabxyabxyabxz"



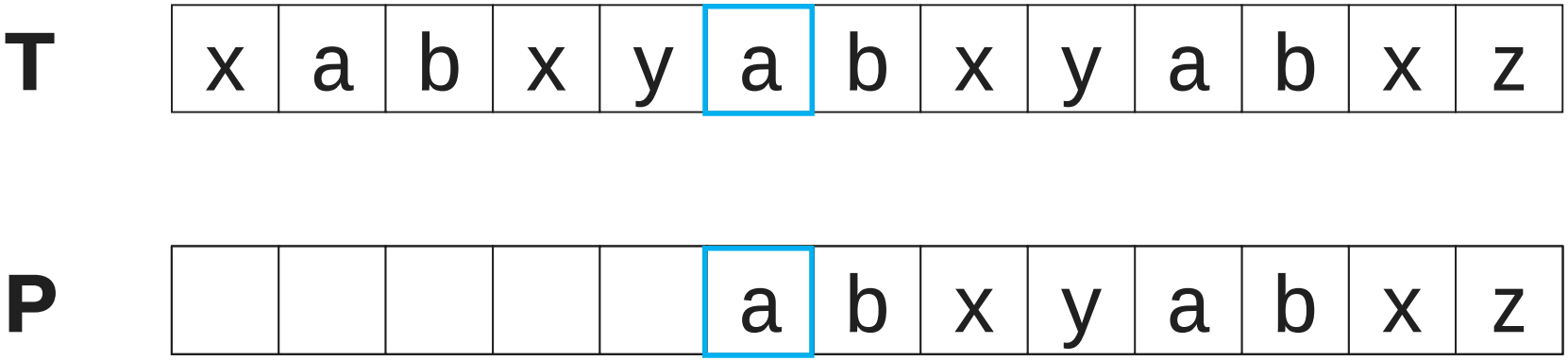
Example

P="abxyabxz" and T="xabxyabxyabxz"



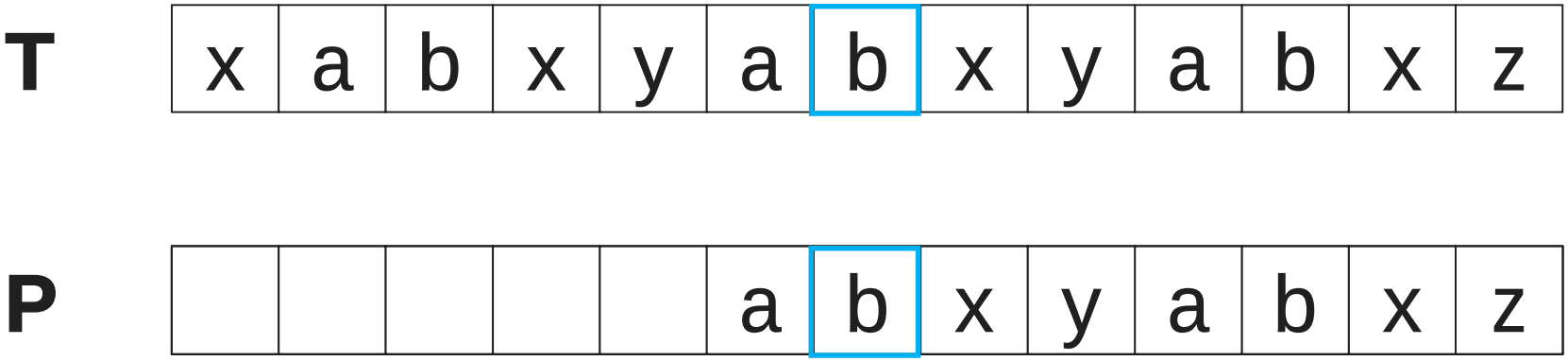
Example

P="abxyabxz" and T="xabxyabxyabxz"



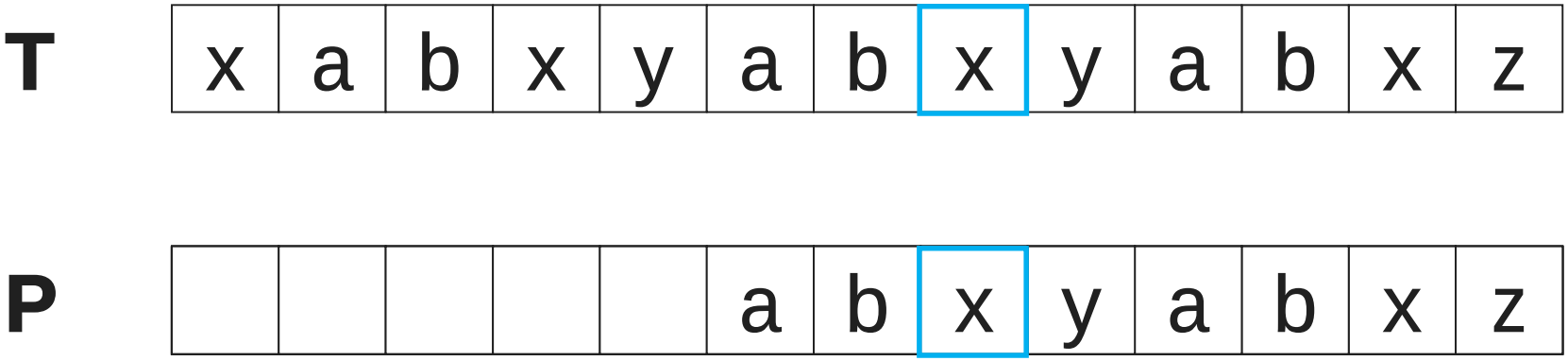
Example

P="abxyabxz" and T="xabxyabxyabxz"



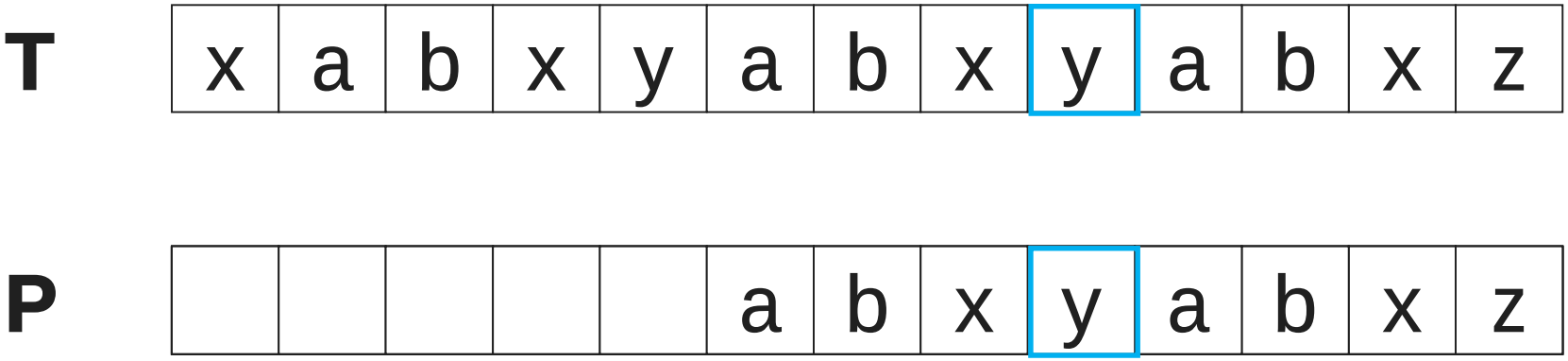
Example

P="abxyabxz" and T="xabxyabxyabxz"



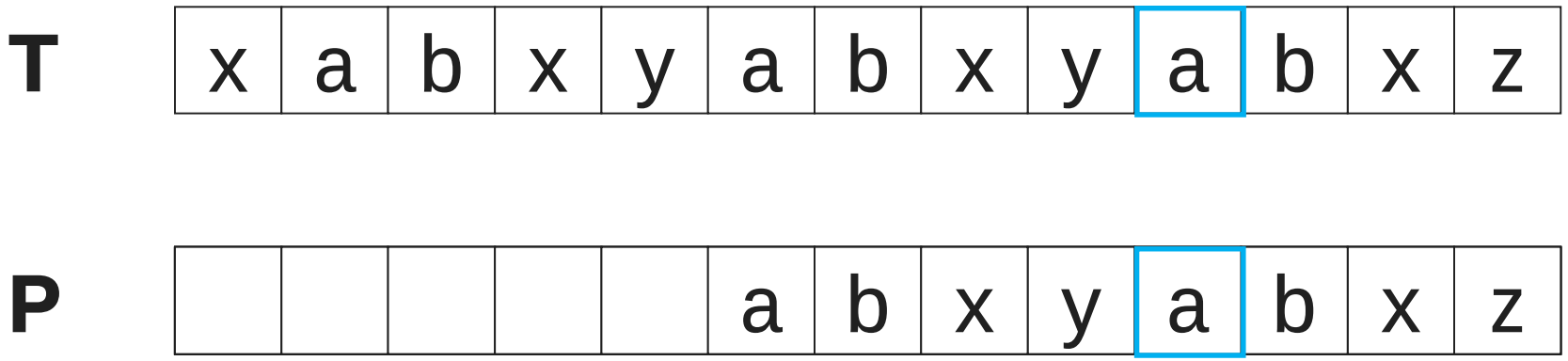
Example

P="abxyabxz" and T="xabxyabxyabxz"



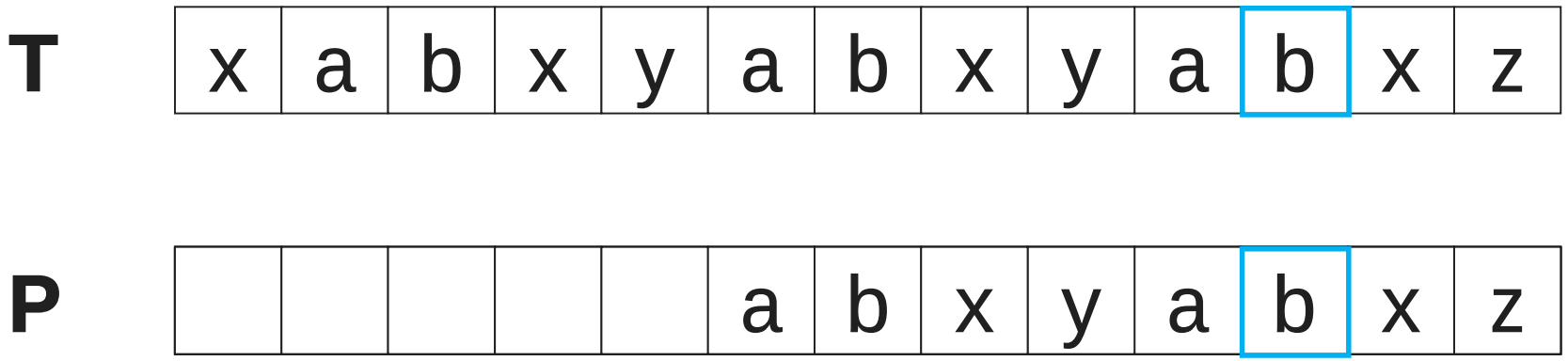
Example

P="abxyabxz" and T="xabxyabxyabxz"



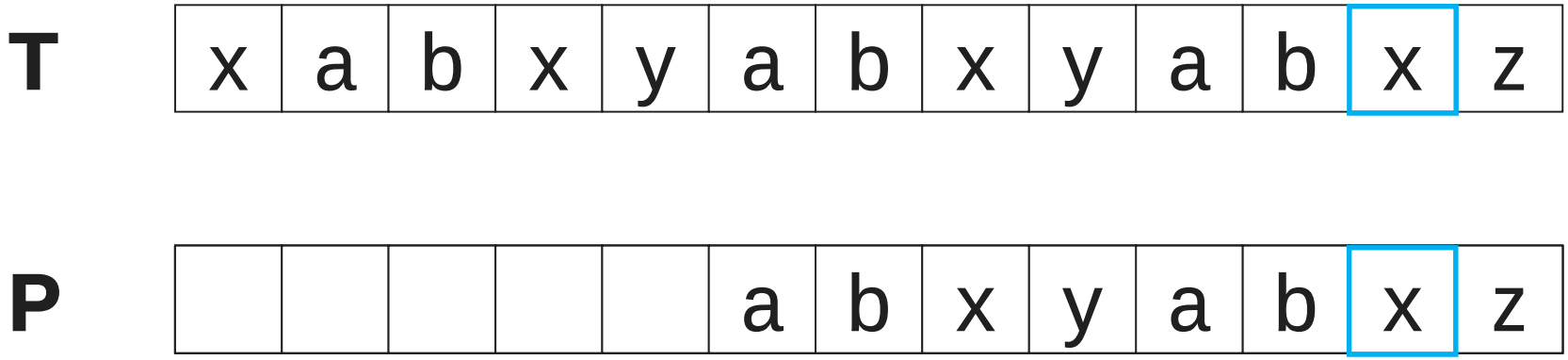
Example

P="abxyabxz" and T="xabxyabxyabxz"



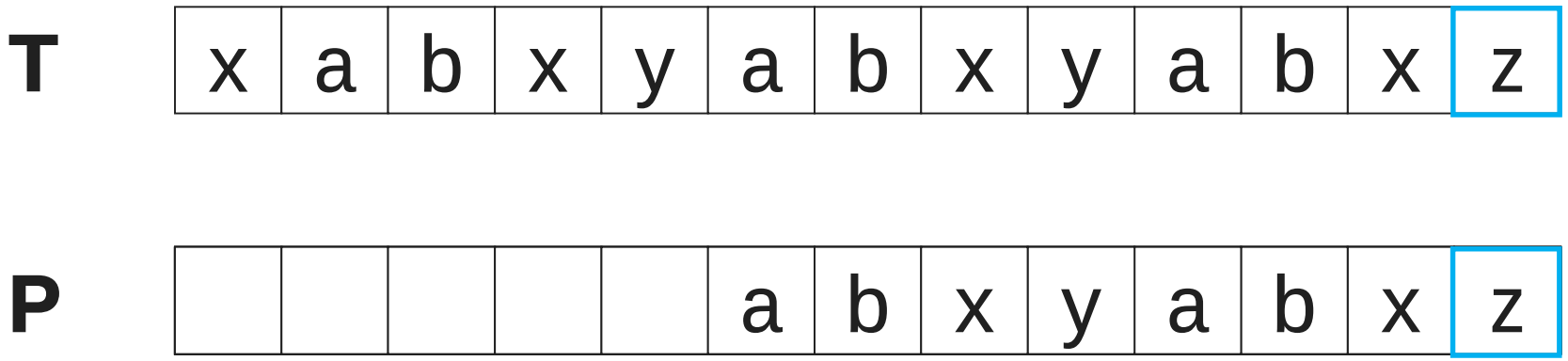
Example

P="abxyabxz" and T="xabxyabxyabxz"



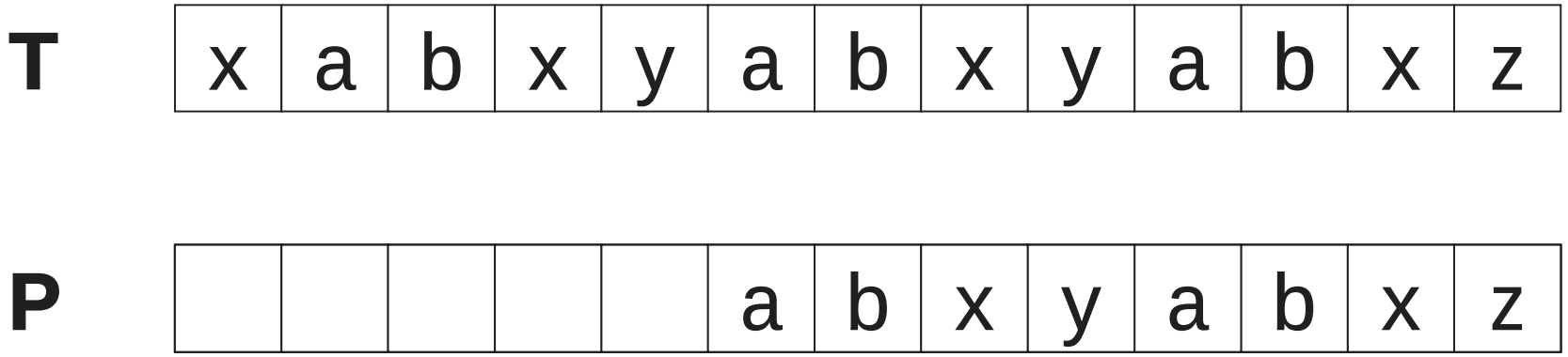
Example

P="abxyabxz" and T="xabxyabxyabxz"



Example

P="abxyabxz" and T="xabxyabxyabxz"



Worst-case Analysis

- There are m comparisons for each shift in the worst case
- There are n-m+1 shifts
- So, the worst-case running time is $\Theta((n-m+1)m)$
- Naïve method is inefficient because information from a shift is not used again

Reflection Question

Suppose that all characters in the pattern P are different. Show how to accelerate NAIVE-STRING-MATCHER to run in time O(n) on an n-character text T .

Excercise

Ch. 32.1-4

Rabin-Karp Algorithm

- Has a worst-case running time of $O((n-m+1)m)$ but average-case is $O(n+m)$
  - Also works well in practice
- Based on number-theoretic notion of *modulo equivalence*
- We assume that $\sum = \{0,1, 2, \ldots, 9\}$, i.e., each character is a decimal digit
  - In general, use radix-d where $d = |\sum|$

Modulo Equivalence

- If *(a mod n) = (b mod n),* then we say
  "*a* is equivalent to *b*, modulo *n*"
- Denoted by $a \equiv b \pmod{n}$

- That is, $a \equiv b \pmod{n}$ if *a* and *b* have the same remainder when divided by *n*
  - E.g., $23 \equiv 37 \equiv -19 \pmod{7}$

Rabin-Karp Approach

- We can view a string of k characters (digits) as a length-k decimal number
  - E.g., the string "31425" corresponds to the decimal number 31,425
- Given a pattern $P[1..m]$, let p denote the corresponding decimal value
- Given a text $T[1..n]$, let t_s denote the decimal value of the length-m substring $T[(s+1)..(s+m)]$ for $s=0,1,\ldots,(n-m)$

Rabin-Karp Approach …contd

- $t_s = p$ iff $T[(s+1)..(s+m)] = P[1..m]$
- s is a valid shift iff $t_s = p$
- p can be computed in $O(m)$ time
  - $p = P[m] + 10\,(P[m-1] + 10\,(P[m-2]+\ldots))$
- t_0 can similarly be computed in $O(m)$ time
- Other $t_1, t_2, \ldots, t_{n-m}$ can be computed in O(n-m) time since t_{s+1} can be computed from t_s in constant time

Rabin-Karp Approach …contd

- $t_{s+1} = 10(t_s - 10^{m-1} \cdot T[s+1]) + T[s+m+1]$
  - E.g., if T={…,3,1,4,1,5,2,…}, m=5 and t_s= 31,415, then $t_{s+1} = 10(31415 - 10000 \cdot 3) + 2$
- We can compute p, t_0, t_1, t_2,…, t_{n-m} in $O(n+m)$ time

Rabin-Karp Approach …contd

- $t_{s+1} = 10(t_s - 10^{m-1} \cdot T[s+1]) + T[s+m+1]$
  - E.g., if T={…,3,1,4,1,5,2,…}, m=5 and t_s= 31,415, then $t_{s+1} = 10(31415 - 10000 \cdot 3) + 2$
- We can compute p, t_0, t_1, t_2,…, t_{n-m} in $O(n+m)$ time

- But…a problem: this is assuming p and t_s are small numbers
  - They may be too large to work with easily

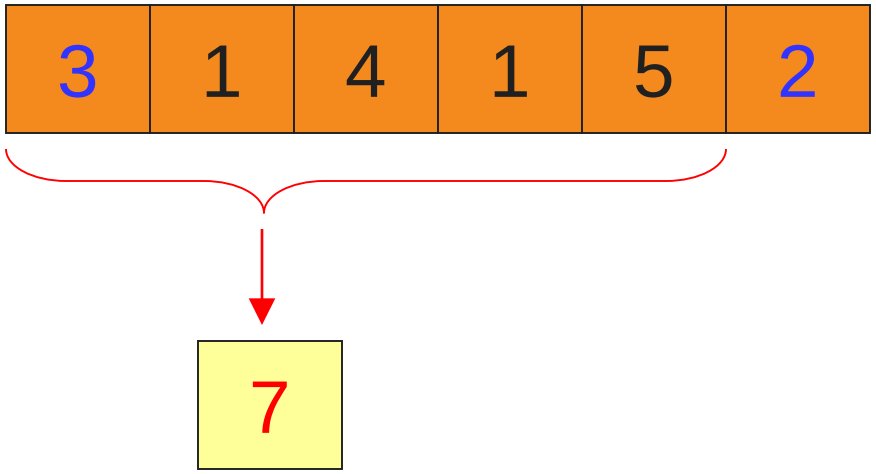
Rabin-Karp Approach …contd

- Solution: we can use modulus arithmetic with a suitable modulus, q
  - E.g., $t_{s+1} \equiv 10(t_s - \ldots) + T[s+m+1] \pmod{q}$

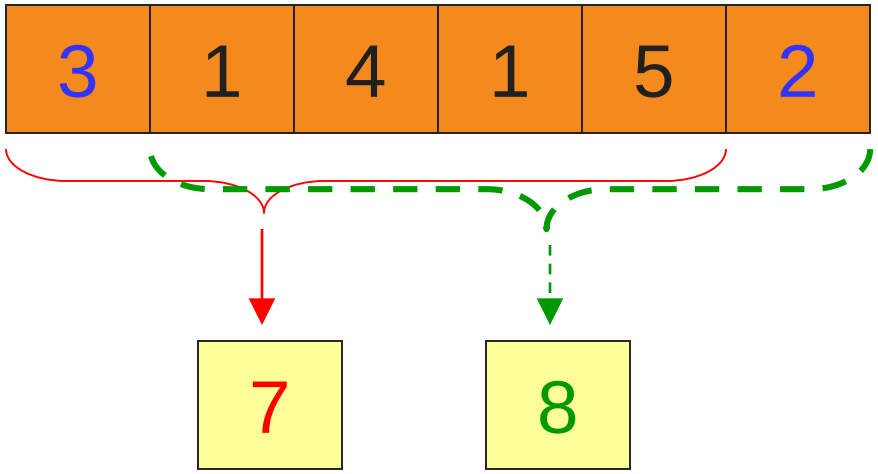
- q is chosen as a small *prime number* ; e.g., 13 for radix 10
  - Generally, if the radix is d, then dq should fit within one computer word

How values modulo 13 are computed

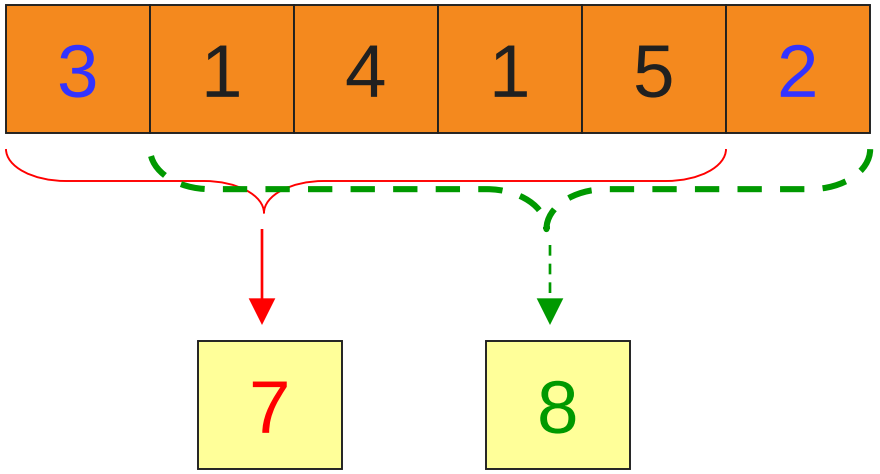
How values modulo 13 are computed



How values modulo 13 are computed



How values modulo 13 are computed

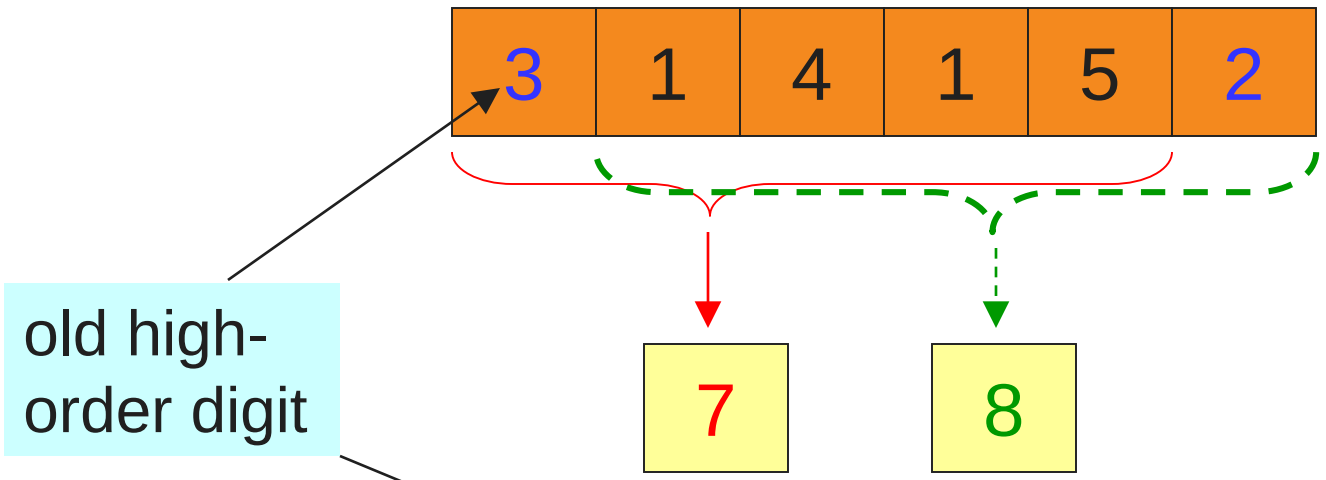


$$14152 \equiv (31415 - 3 \cdot 10000) \cdot 10 + 2 \pmod{13}$$

$$\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13}$$

$$\equiv 8 \pmod{13}$$

How values modulo 13 are computed

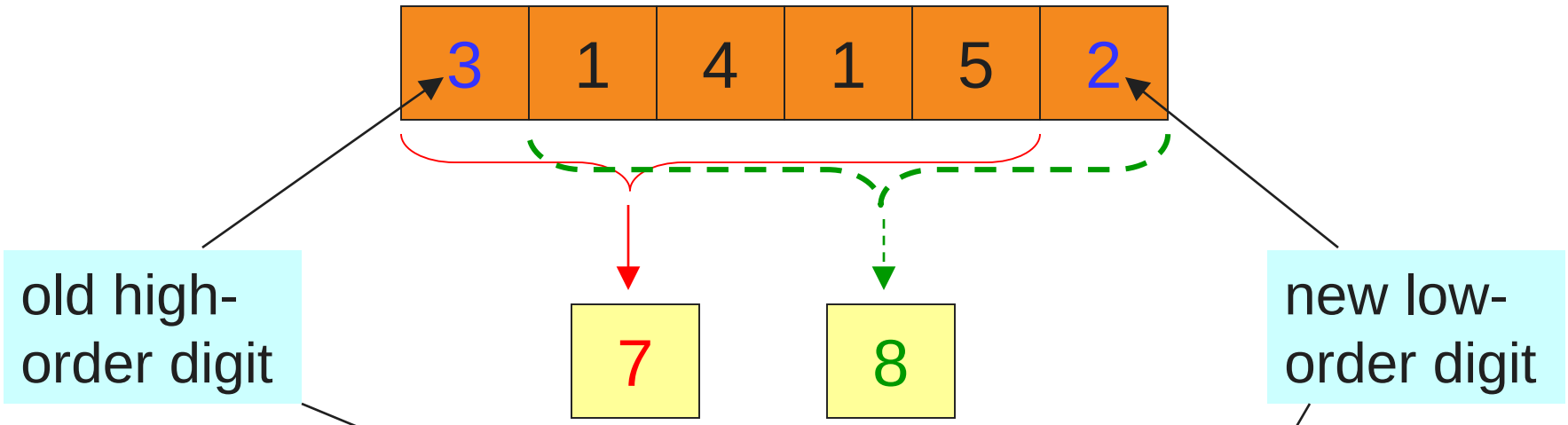


$$14152 \equiv (31415 - 3 \cdot 10000) \cdot 10 + 2 \pmod{13}$$

$$\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13}$$

$$\equiv 8 \pmod{13}$$

How values modulo 13 are computed



$$14152 \equiv (31415 - 3 \cdot 10000) \cdot 10 + 2 \pmod{13}$$

$$\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13}$$

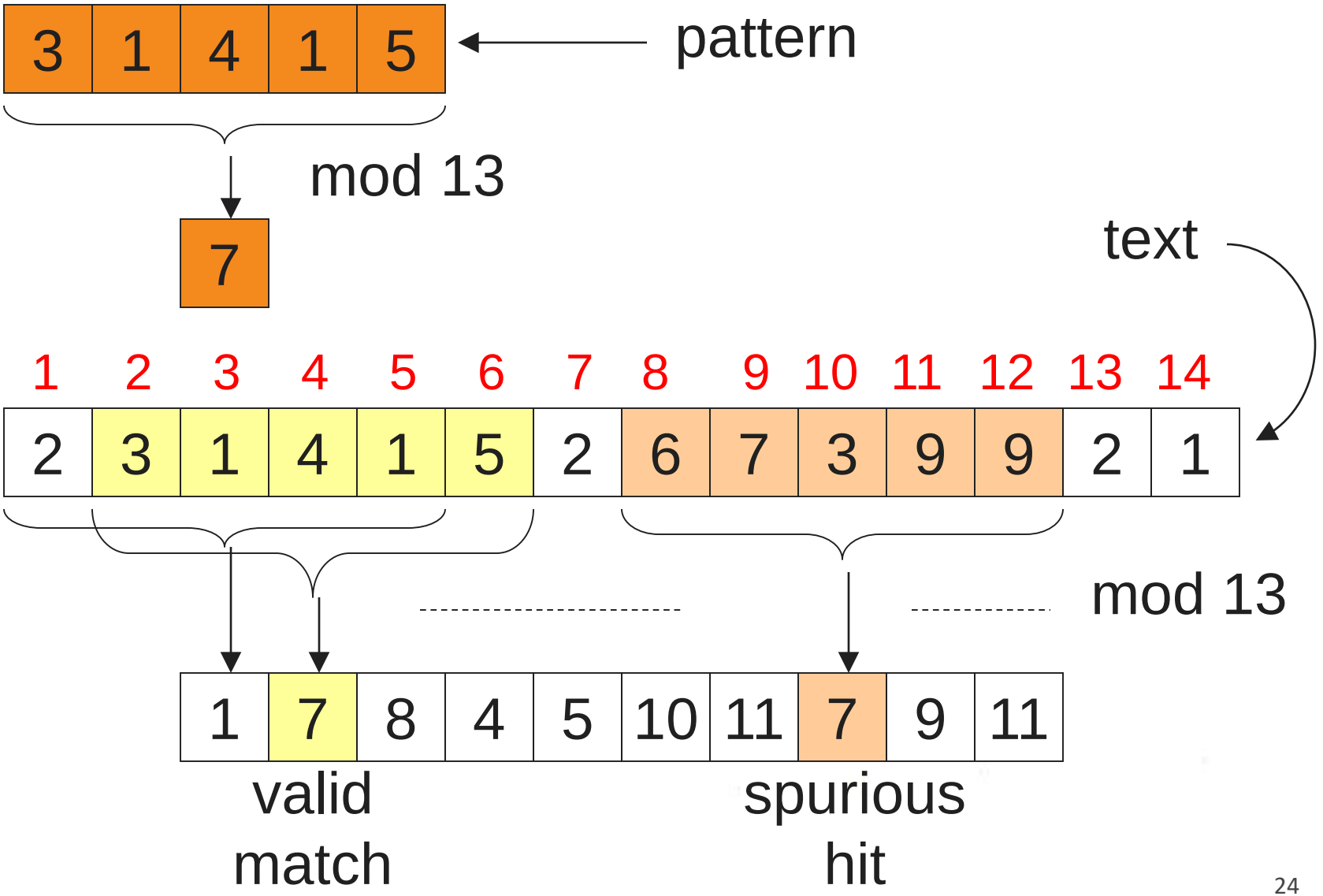
$$\equiv 8 \pmod{13}$$

Problem of Spurious Hits

- $t_s \equiv p \pmod{q}$ does not imply that $t_s = p$
  - Modular equivalence does not necessarily mean that two integers are equal
- A case in which $t_s \equiv p \pmod{q}$ when $t_s \neq p$ is called a *spurious hit*

- On the other hand, if two integers are not modulo equivalent, then they cannot be equal

Example


3 1 4 1 5
pattern
mod 13
7
text
1 2 3 4 5 6 7 8 9 10 11 12 13 14
2 3 1 4 1 5 2 6 7 3 9 9 2 1
mod 13
1 7 8 4 5 10 11 7 9 11
valid match
spurious hit

Rabin-Karp Algorithm

- Basic structure like the naïve algorithm, but uses modular arithmetic as described
- For each *hit*, i.e., for each s where $t_s \equiv p \pmod{q}$, verify character by character whether s is a valid shift or a spurious hit
- In the worst case, every shift is verified
  - Running time can be shown as $O((n-m+1)m)$
- Average-case running time is $O(n+m)$

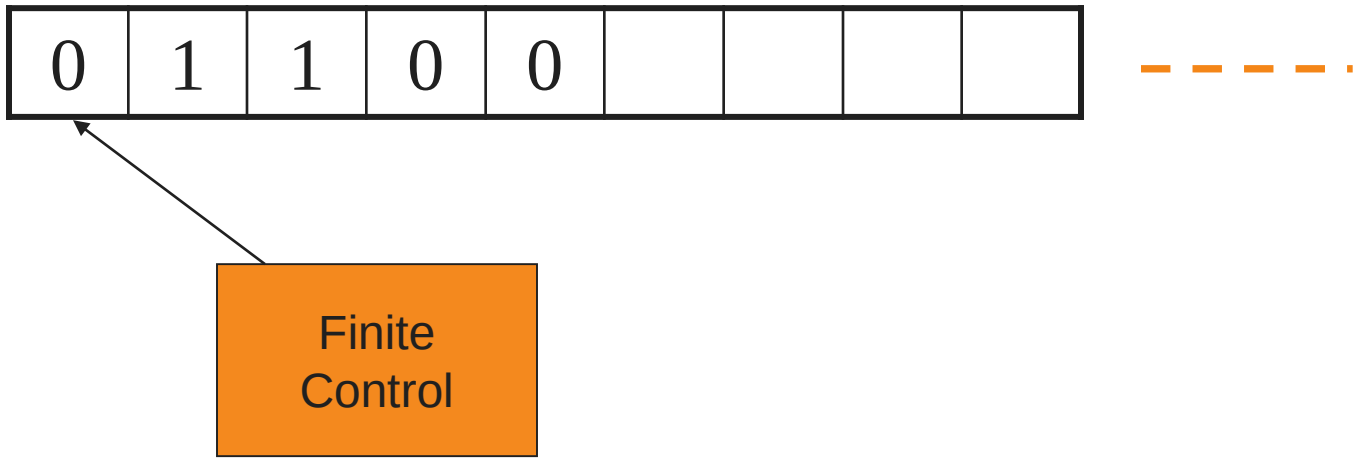
Excercise

- Ch. 32.2-3

  Show how to extend the Rabin-Karp method to handle the problem of looking for a given mxm pattern in an nxn array of characters.

  (The pattern may be shifted vertically and horizontally, but it may not be rotated.)

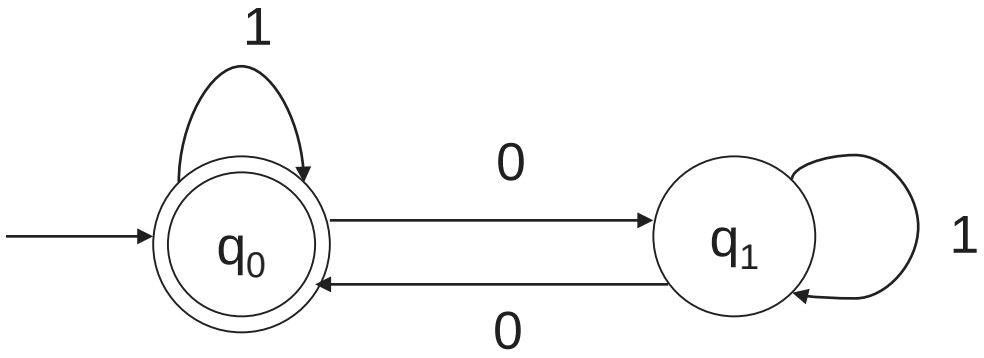
Deterministic Finite State Automata (DFA)



- One-way, infinite tape, broken into cells
- One-way, read-only tape head.
- Finite control, I.e., a program, containing the position of the read head, current symbol being scanned, and the current “state.”
- A string is placed on the tape, read head is positioned at the left end, and the DFA will read the string one symbol at a time until all symbols have been read. The DFA will then either accept or reject.

- The finite control can be described by a transition diagram:

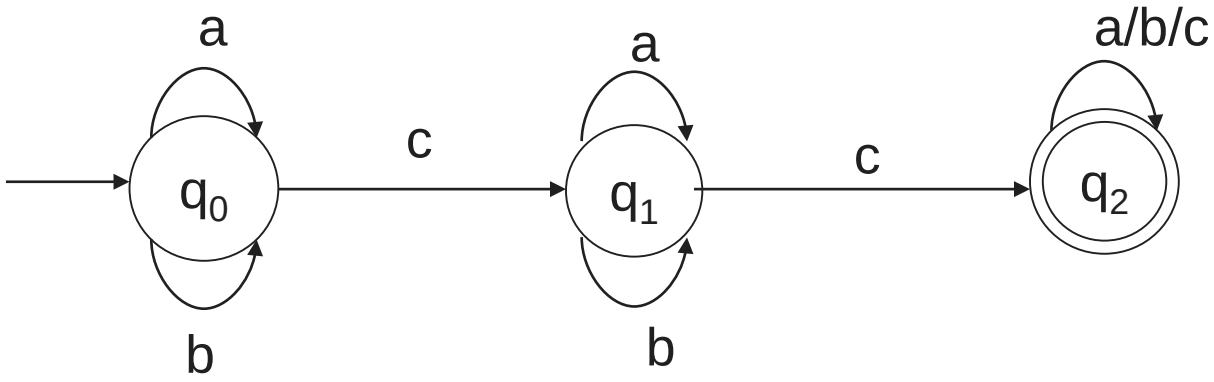
- Example #1:



1	0	0	1	1	
q_0	q_0	q_1	q_0	q_0	q_0

- One state is final/accepting, all others are rejecting.
- The above DFA accepts those strings that contain an even number of 0's

- Example #2:



a	c	c	c	b		accepted
q_0	q_0	q_1	q_2	q_2	q_2	
a	a	c				rejected
q_0	q_0	q_0	q_1			

- Accepts those strings that contain <u>at least</u> two c's

Finite Automata

- A *finite automaton M* is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where
  - Q is a finite set of *states*
  - $q_0 \in Q$ is the *start state*
  - $A \subseteq Q$ is a set of *accepting states*
  - Σ is a finite *input alphabet*
  - δ is the *transition function* that gives the next state for a given current state and input

How a Finite Automaton Works

- The finite automaton M begins in state q_0
- Reads characters from $\sum$ one at a time
- If M is in state q and reads input character a, M moves to state $\delta(q,a)$
- If its current state q is in A, M is said to have *accepted* the string read so far
- An input string that is not accepted is said to be *rejected*

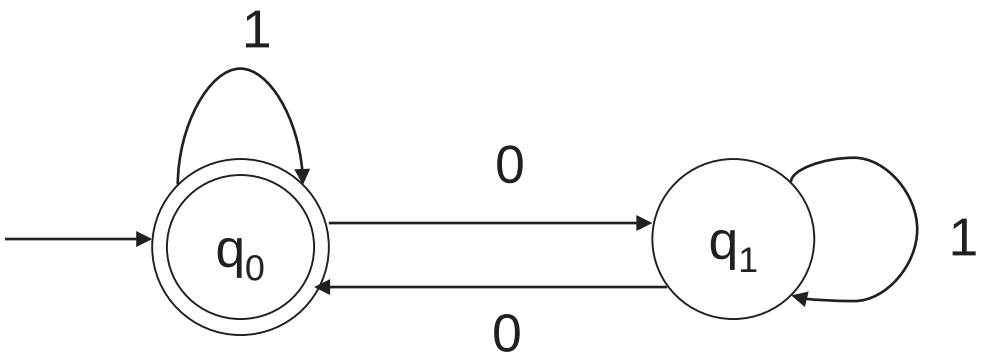
- For example #1:

$Q = \{q_0, q_1\}$

$\Sigma = \{0, 1\}$

Start state is q_0

$F = \{q_0\}$



δ:

	0	1
q_0	q_1	q_0
q_1	q_0	q_1

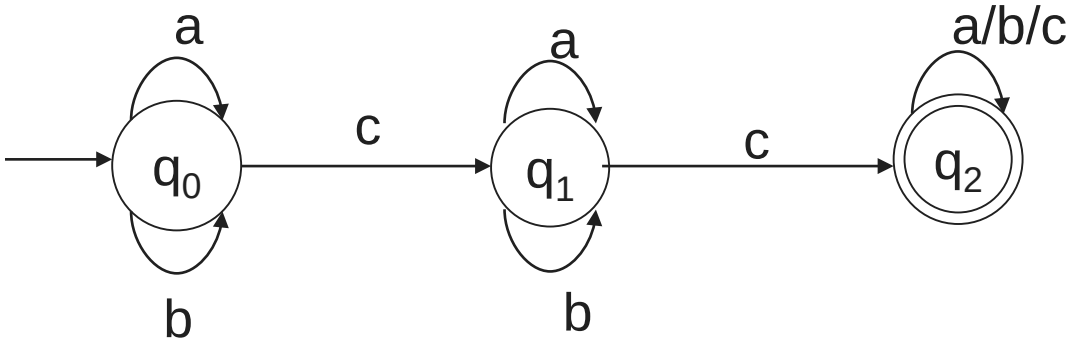
- For example #2:

$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{a, b, c\}$

Start state is q_0

$F = \{q_2\}$



δ:

	a	b	c
q_0	q_0	q_0	q_1
q_1	q_1	q_1	q_2
q_2	q_2	q_2	q_2

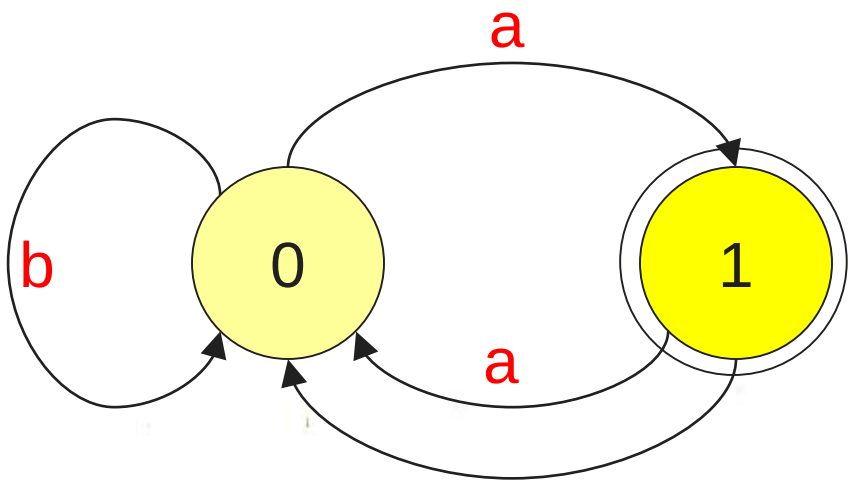
- Since δ is a function, at each step M has exactly one option.

Example 3

- $Q = \{0,1\}$, $q_0 = 0$, A={1}, $\sum = \{a, b\}$
- $\delta(q,a)$ shown in the transition table/diagram
- This accepts strings that end in an odd number of a's; e.g., abbaaa is accepted, aa is rejected

state	input a	input b
0	1	0
1	0	0

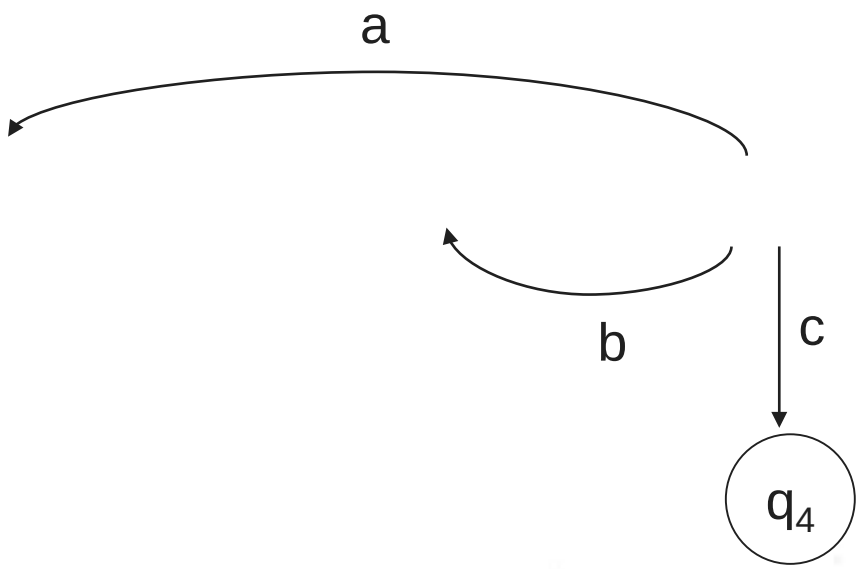
transition table



transition diagram

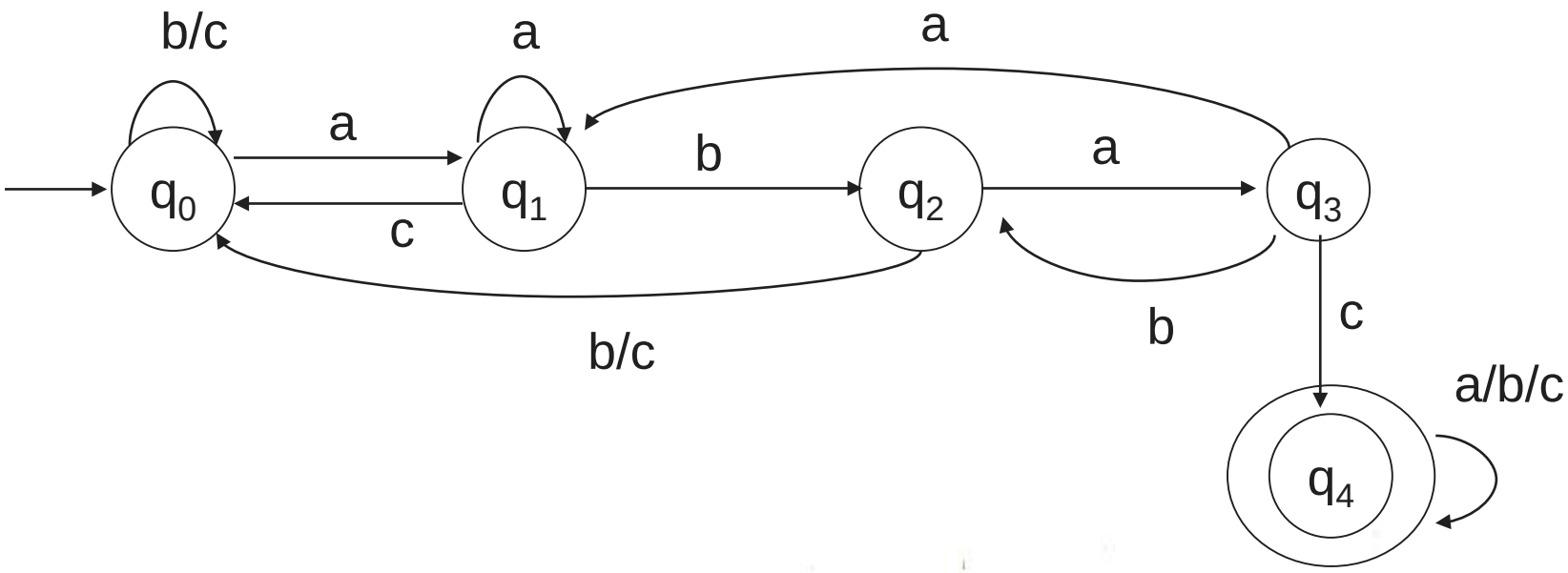
Example 4

- Give a DFA M to accept all strings of a's, b's and c's which contain the substring *abac*



Example 4

- Give a DFA M to accept all strings of a's, b's and c's which contain the <u>substring *abac*</u>



String-Matching Automata

- Given the pattern $P[1..m]$, build a finite automaton M
  - The state set is $Q=\{0, 1, 2, \ldots, m\}$
  - The start state is 0
  - The only accepting state is m

- Time to build M can be large if $\sum$ is large

String-Matching Automata …contd

- Scan the text string T [1..n] to find all occurrences of the pattern P [1..m]

- String matching is efficient: $\Theta(n)$
  - Each character is examined exactly once
  - Constant time for each character
- But …time to compute δ is $O(m\ |\sum|)$
  - δ Has $O(m\ |\sum|\ )$ entries

Algorithm

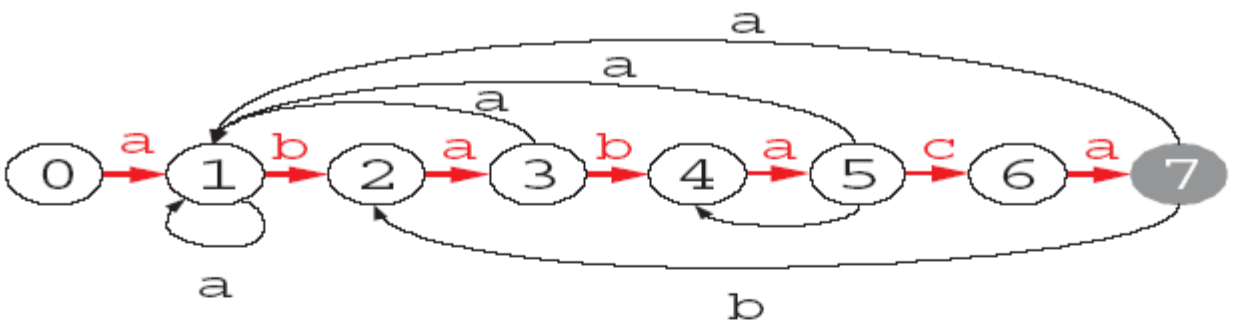
Input: Text string T [1..n], δ and m

Result: All valid shifts displayed

FINITE-AUTOMATON-MATCHER (T, m, δ)

```
    n ← length[T]
    q ← 0
    for i ← 1 to n
        q ← δ (q, T [i])
        if q = m
                print "pattern occurs with shift" i-m
```

Example



state	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	

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
$\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

Knuth-Morris-Pratt (KMP) Method

- Avoids computing δ (transition function)
- Instead computes a *prefix function* π in $O(m)$ time
  - π has only m entries

- Prefix function stores info about how the pattern matches against shifts of itself
  - Can avoid testing useless shifts

Terminology/Notations

- String *w* is a *prefix* of string *x*, if *x=wy* for some string *y* (e.g., “srilan” of “srilanka”)
- String *w* is a *suffix* of string *x*, if *x=yw* for some string *y* (e.g., “anka” of “srilanka”)
- The *k*-character prefix of the pattern *P* [1..*m*] denoted by P_k
  - E.g., $P_0 = \varepsilon$, $P_m = P = P[1..m]$

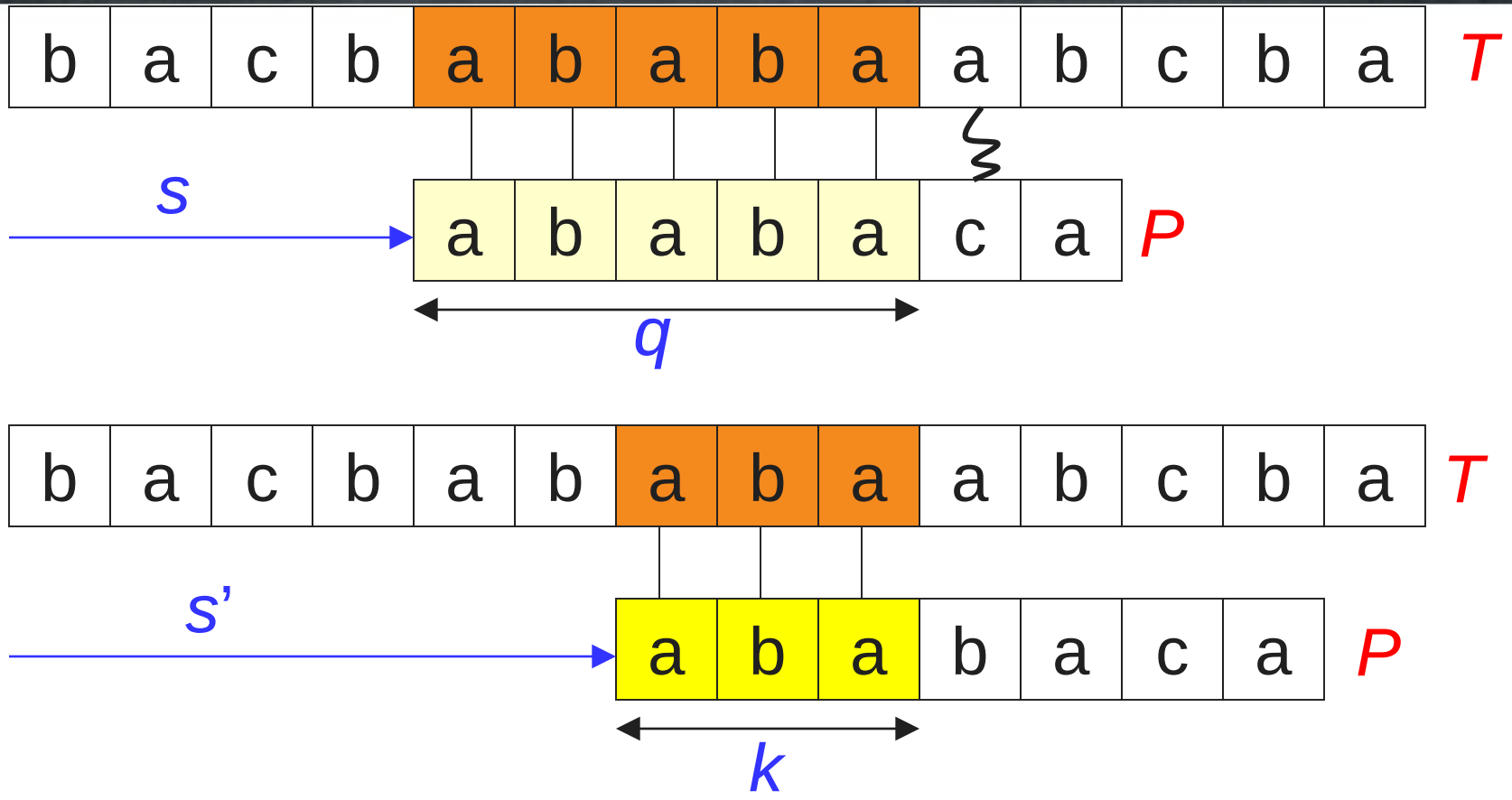
Prefix Function for a Pattern

- Given that pattern prefix $P[1..q]$ matches text characters $T[(s+1)..(s+q)]$, what is the least shift $s' > s$ such that

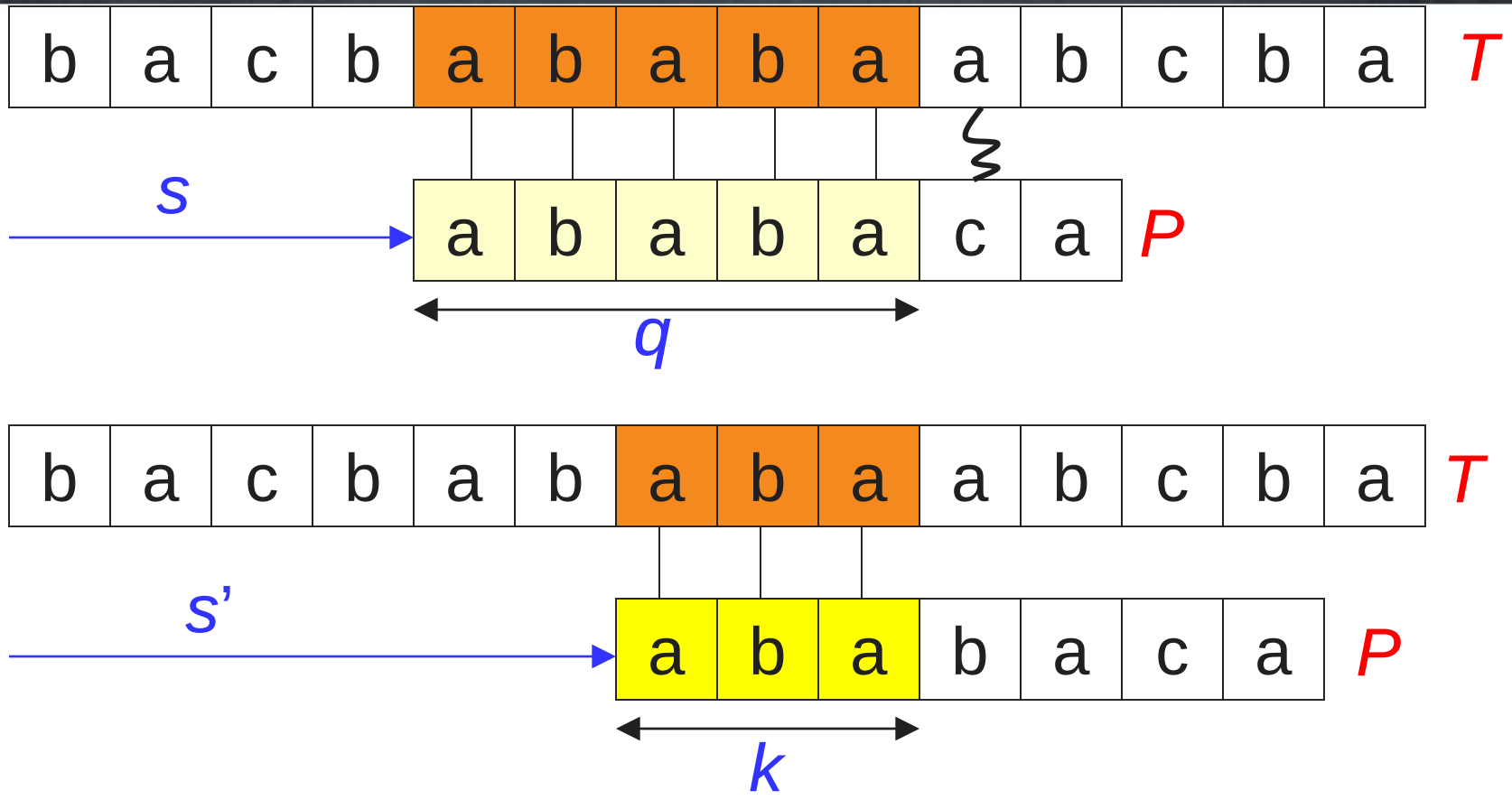
  $P[1..k] = T[(s'+1)..(s'+k)]$ where $s'+k=s+q$?

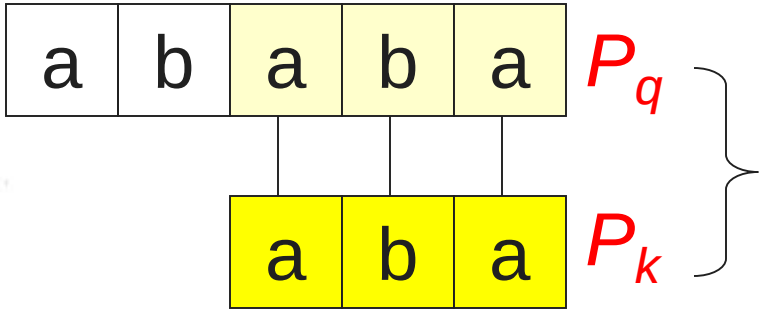
- At the new shift s', no need to compare the first k characters of P with corresponding characters of T
  - Since we know that they match

Prefix Function: Example 1



Prefix Function: Example 1





Compare pattern against itself; longest prefix of P that is also a suffix of P_5 is P_3; so $\pi[5]= 3$

Knuth-Morris-Pratt (KMP) Algorithm

```
COMPUTE-PREFIX-FUNCTION(P)
 1  m = P.length
 2  let π[1..m] be a new array
 3  π[1] = 0
 4  k = 0
 5  for q = 2 to m
 6       while k > 0 and P[k + 1] ≠ P[q]
 7            k = π[k]
 8       if P[k + 1] == P[q]
 9            k = k + 1
10       π[q] = k
11  return π
```

Prefix Function: Example 2

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	**a**	**b**	**a**	**b**	**a**	**b**	**a**	**b**	**c**	**a**
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

$\pi[q] = \max \{ k \mid k<q \text{ and } P_k \text{ is a suffix of } P_q \}$

Knuth-Morris-Pratt (KMP) Algorithm

```
KMP-MATCHER(T, P)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0                                  // number of characters matched
 5  for i = 1 to n                         // scan the text from left to right
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]                       // next character does not match
 8      if P[q + 1] == T[i]
 9          q = q + 1                      // next character matches
10      if q == m                          // is all of P matched?
11          print "Pattern occurs with shift" i − m
12          q = π[q]                       // look for the next match
```

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

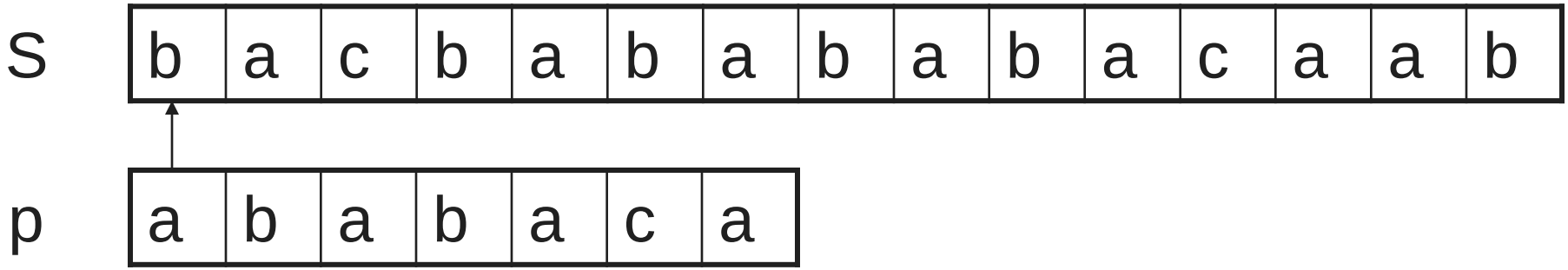
Let us execute the KMP algorithm to find whether ‘p’ occurs in ‘S’.

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	0	1

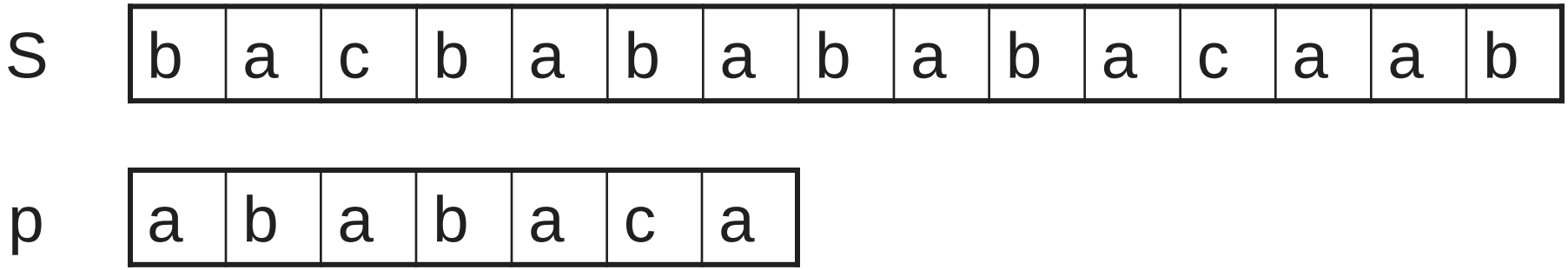
Initially: n = size of S = 15;
m = size of p = 7

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



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

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



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

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

S

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 S[1]. ‘p’ will be shifted one position to the right.

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

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

S

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

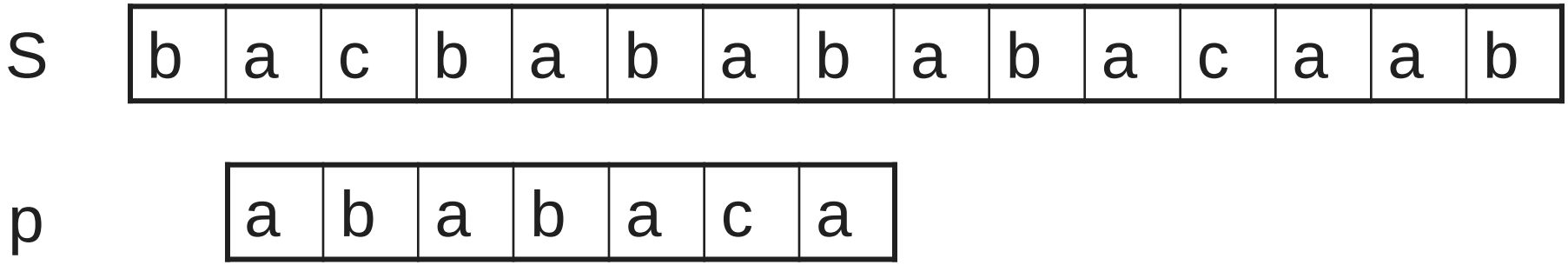
p

a	b	a	b	a	c	a

P[1] does not match with S[1]. ‘p’ will be shifted one position to the right.

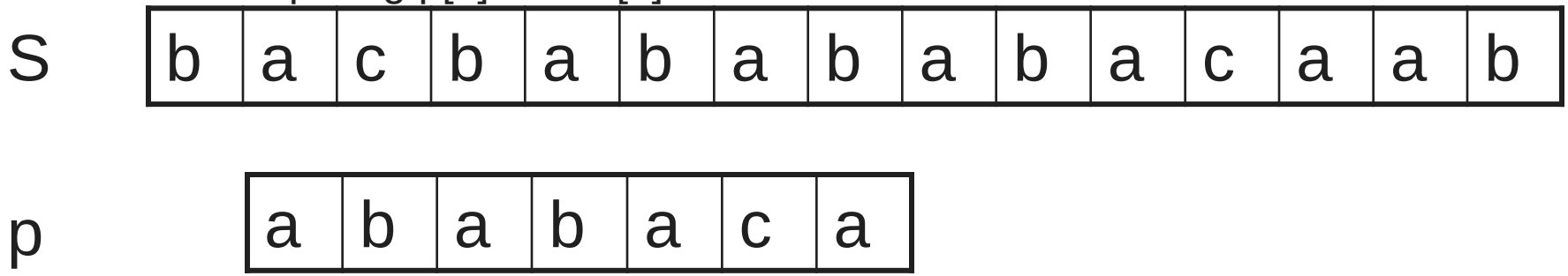
Initially: n = size of S = 15;
m = size of p = 7

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



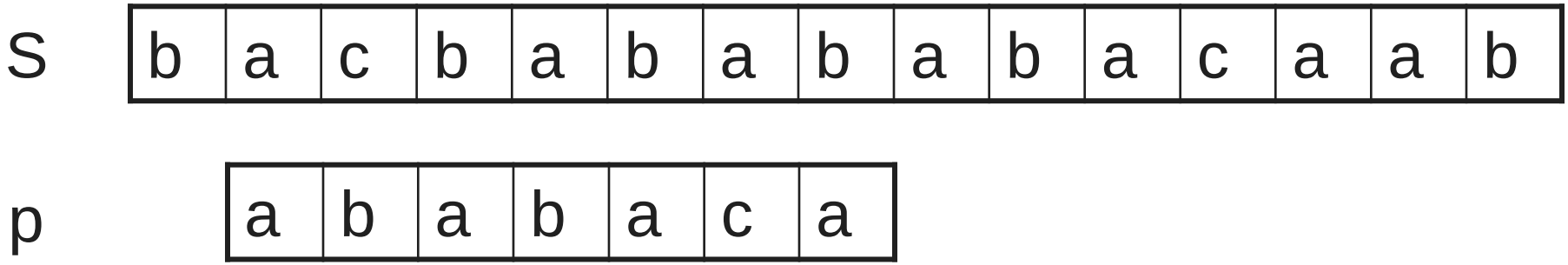
P[1] does not match with S[1]. ‘p’ will be shifted one position to the right.

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



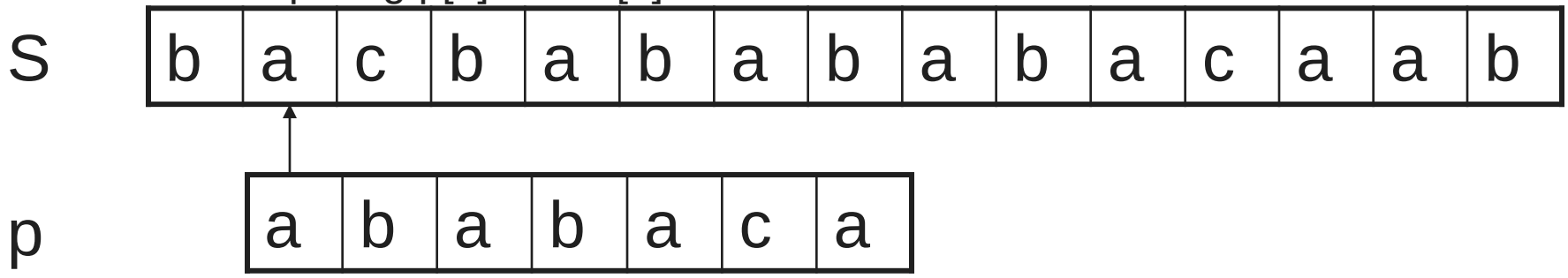
Initially: n = size of S = 15;
m = size of p = 7

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



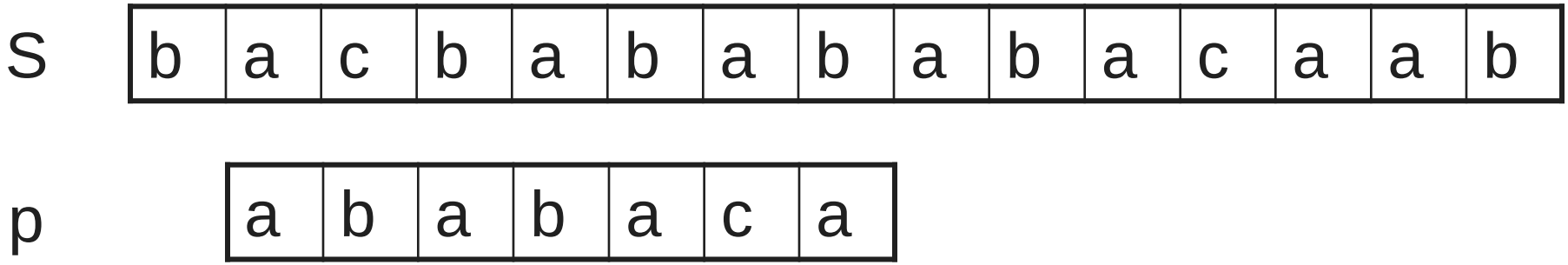
P[1] does not match with S[1]. ‘p’ will be shifted one position to the right.

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



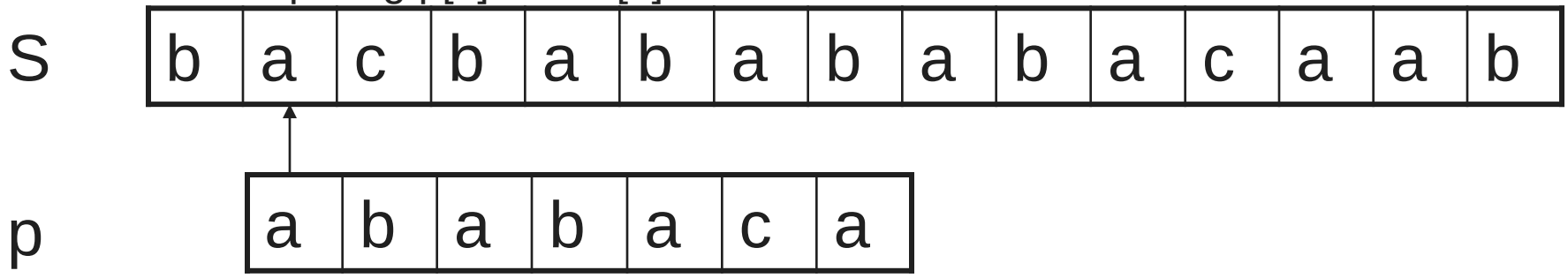
Initially: n = size of S = 15;
m = size of p = 7

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



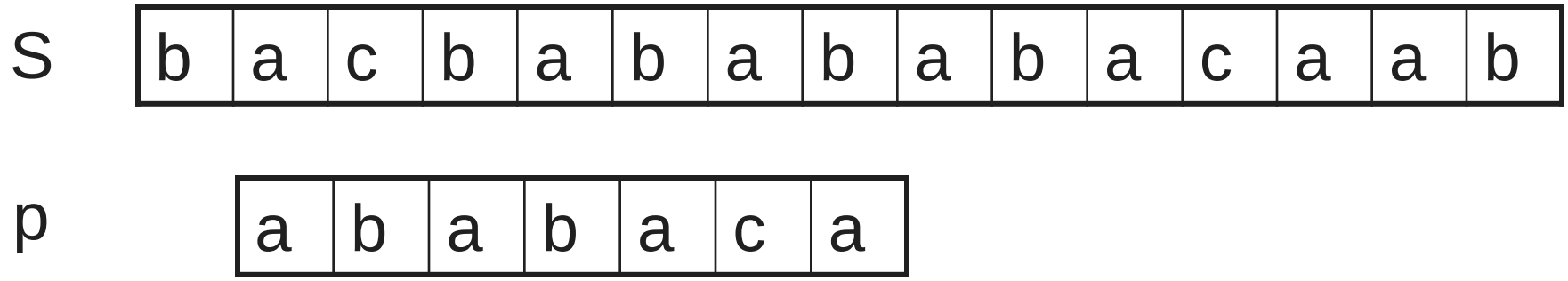
P[1] does not match with S[1]. ‘p’ will be shifted one position to the right.

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



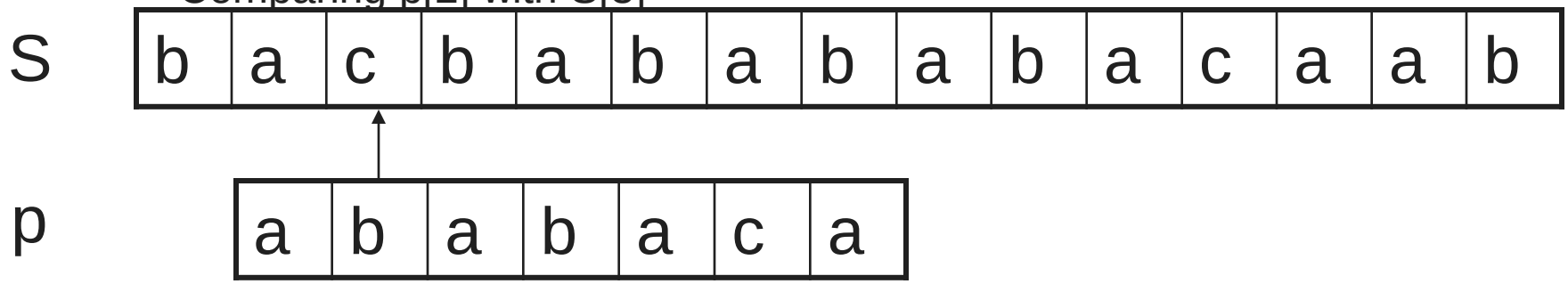
P[1] matches S[2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1



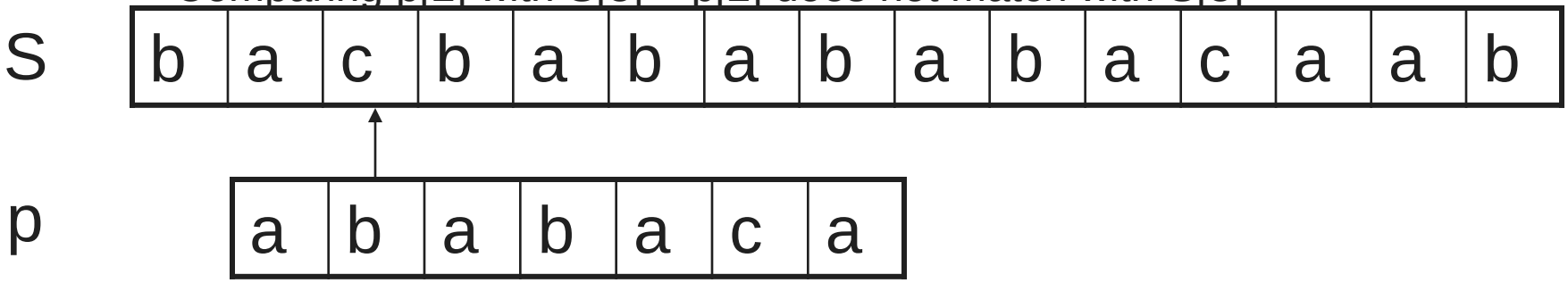
Step 3: i = 3, q = 1

Comparing p[2] with S[3]



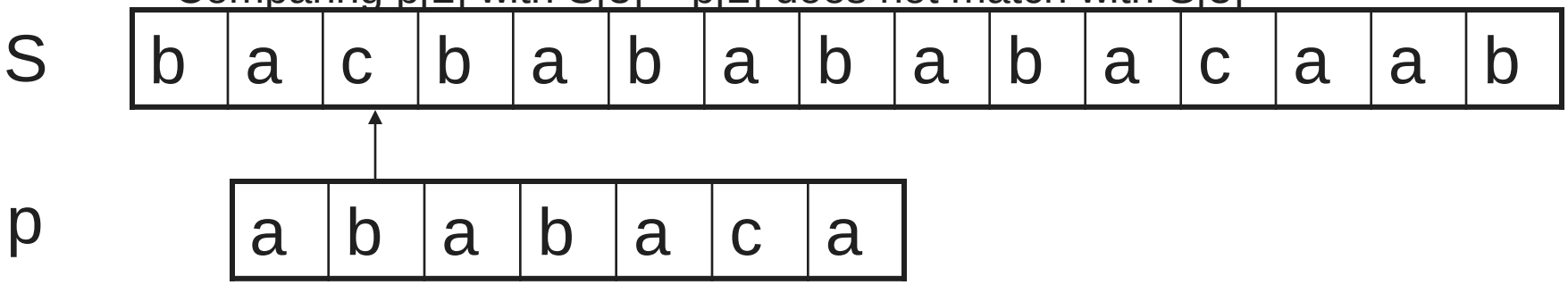
Step 3: i = 3, q = 1

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



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 3: i = 3, q = 1

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

S

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 S[3]

Step 3: i = 3, q = 1

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

S

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

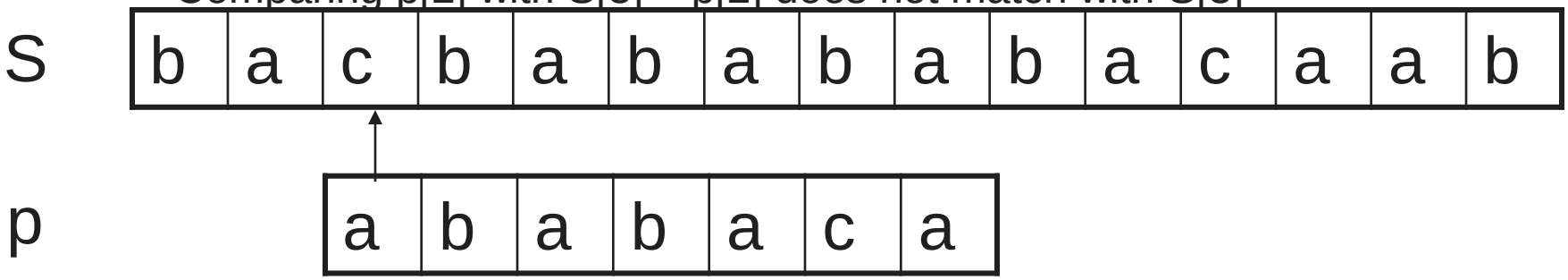
p

a	b	a	b	a	c	a

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

Step 3: i = 3, q = 1

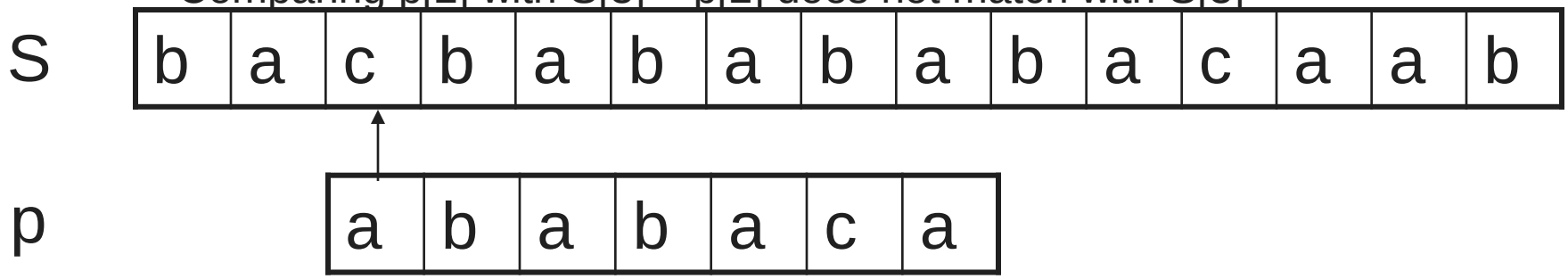
Comparing p[2] with S[3] p[2] does not match with S[3]



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

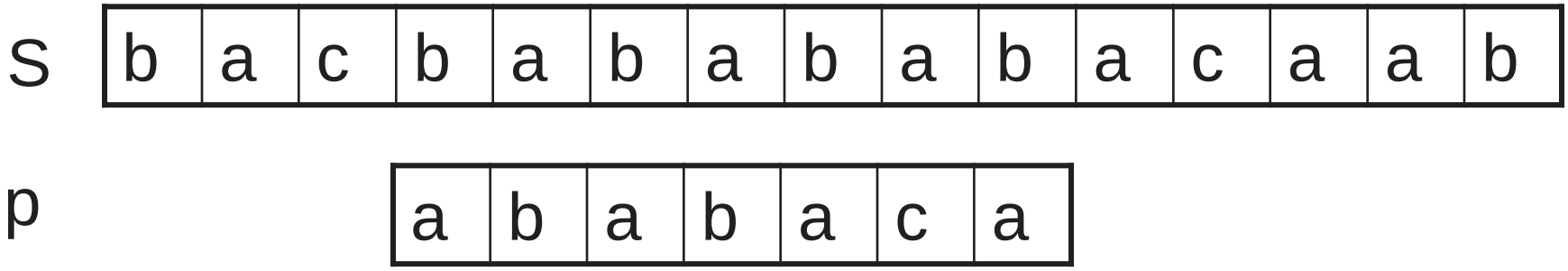
Step 3: i = 3, q = 1

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



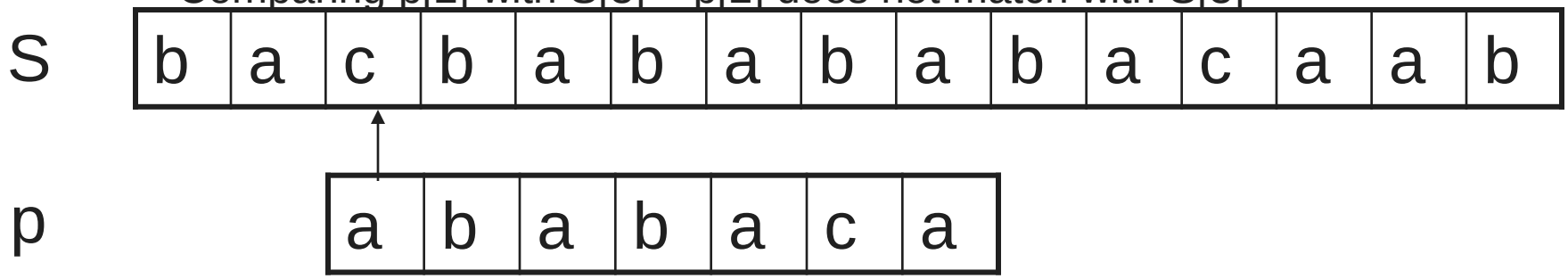
Backtracking on p, comparing p[1] and S[3]

Step 4: i = 4, q = 0



Step 3: i = 3, q = 1

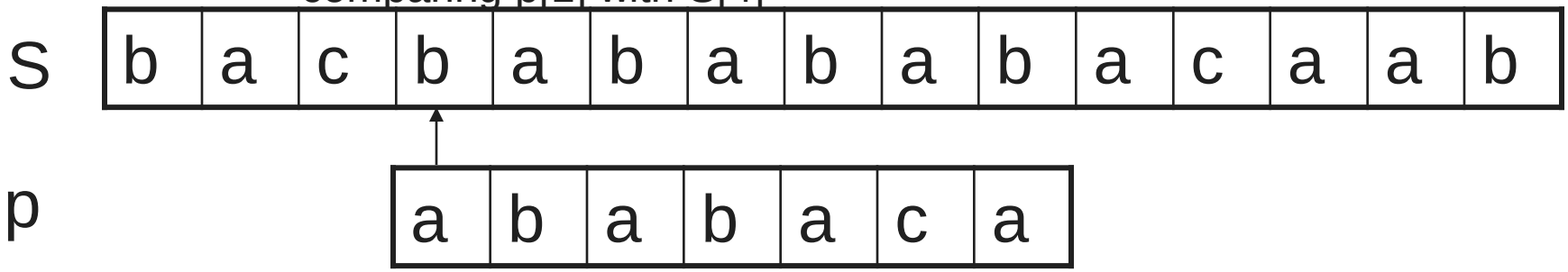
Comparing p[2] with S[3] p[2] does not match with S[3]



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

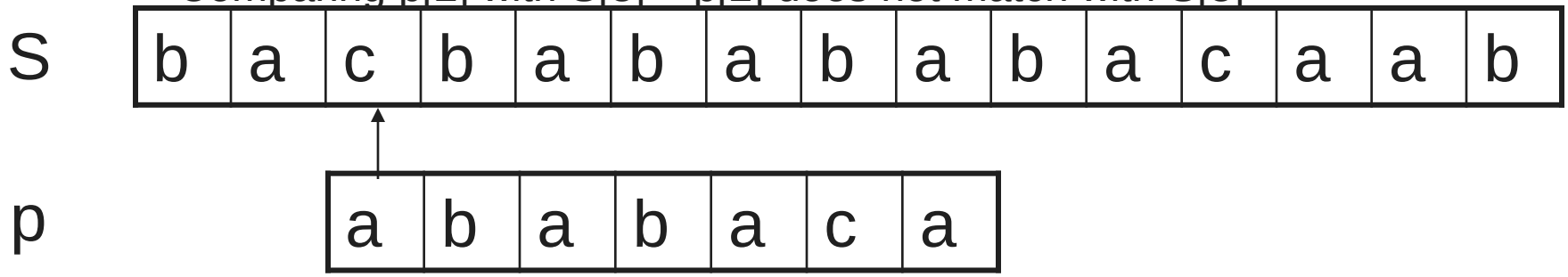
Step 4: i = 4, q = 0

comparing p[1] with S[4]



Step 3: i = 3, q = 1

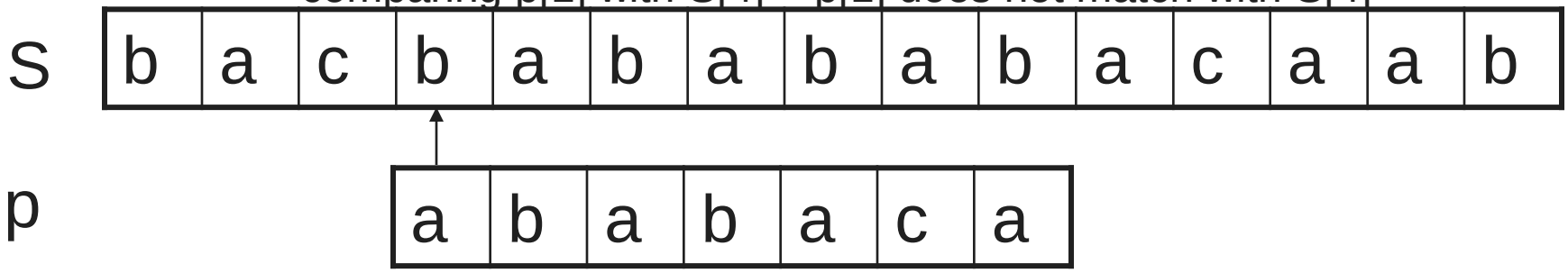
Comparing p[2] with S[3] p[2] does not match with S[3]



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

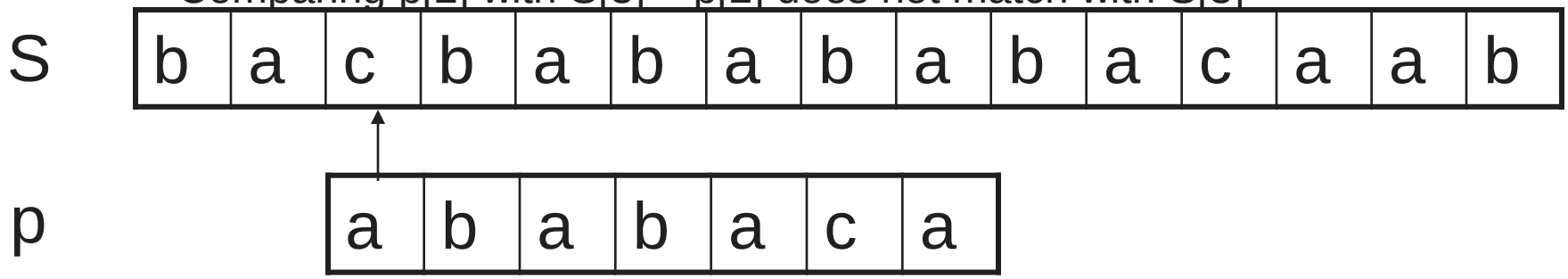
Step 4: i = 4, q = 0

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



Step 3: i = 3, q = 1

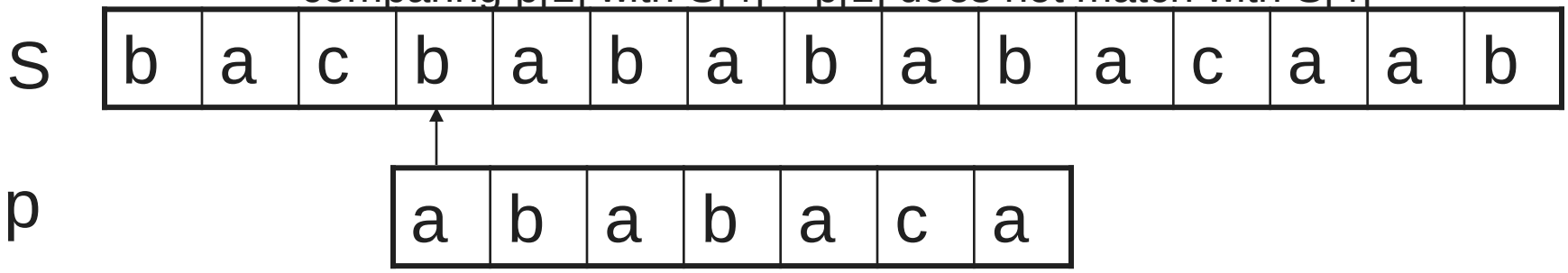
Comparing p[2] with S[3] p[2] does not match with S[3]



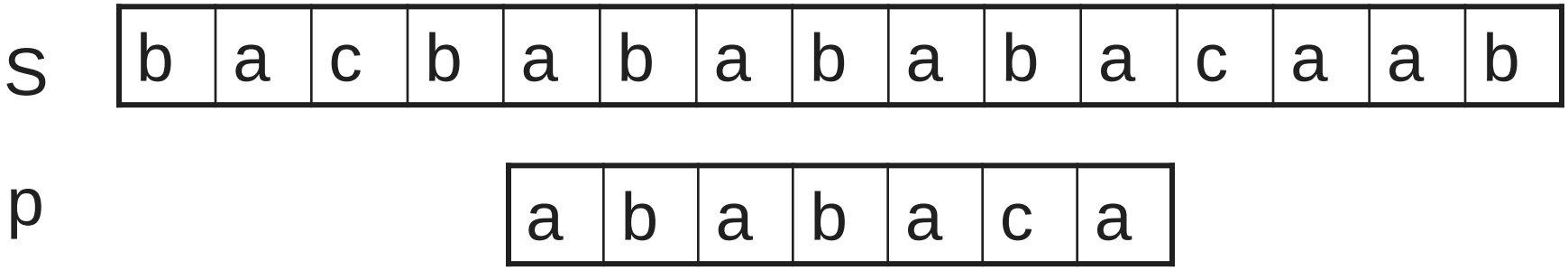
Backtracking on p, comparing p[1] and S[3]

Step 4: i = 4, q = 0

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

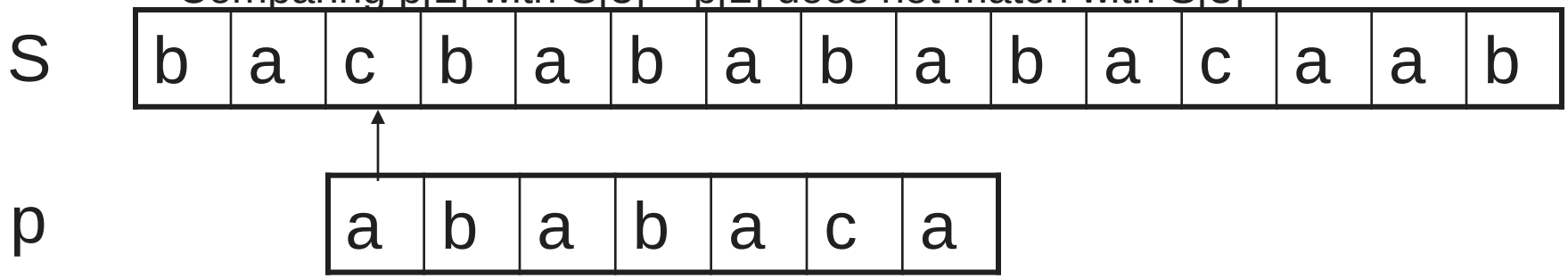


Step 5: i = 5, q = 0



Step 3: i = 3, q = 1

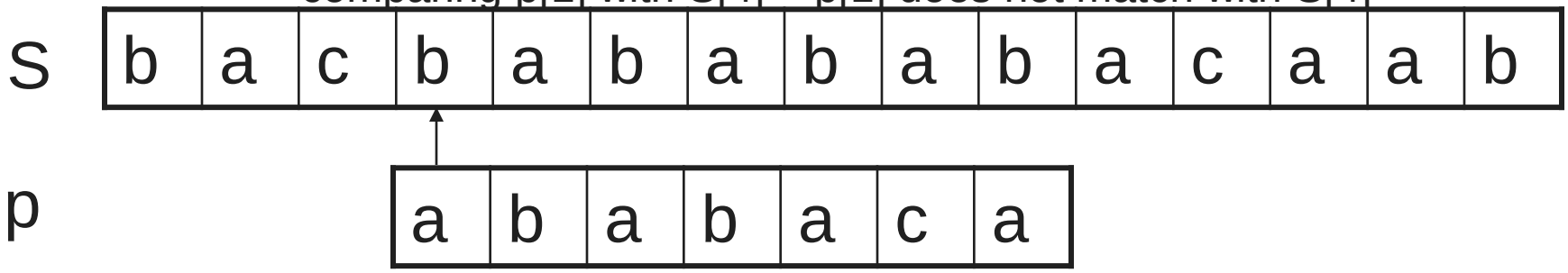
Comparing p[2] with S[3] p[2] does not match with S[3]



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

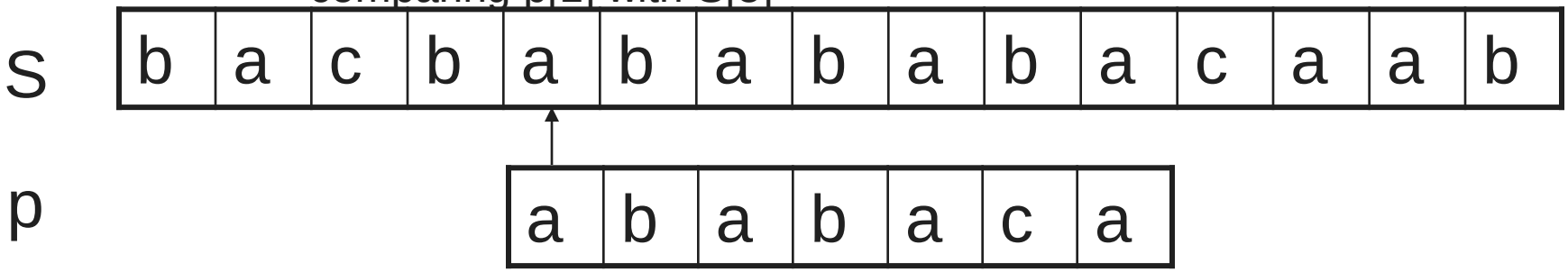
Step 4: i = 4, q = 0

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



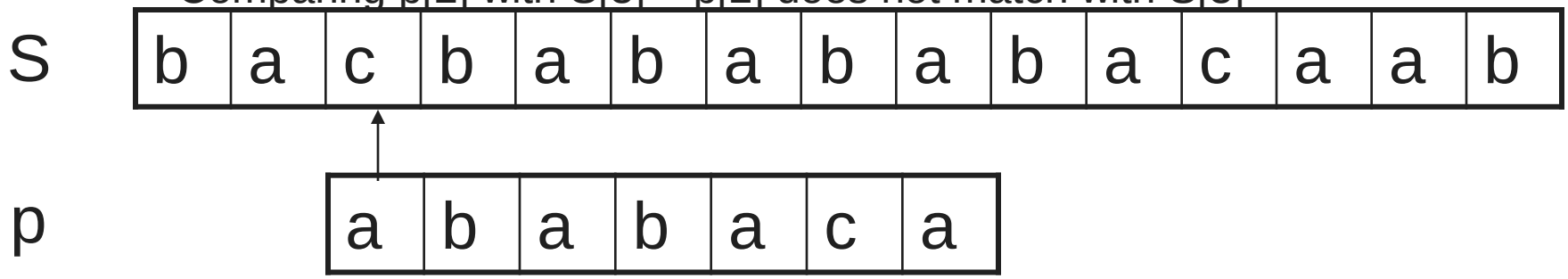
Step 5: i = 5, q = 0

comparing p[1] with S[5]



Step 3: i = 3, q = 1

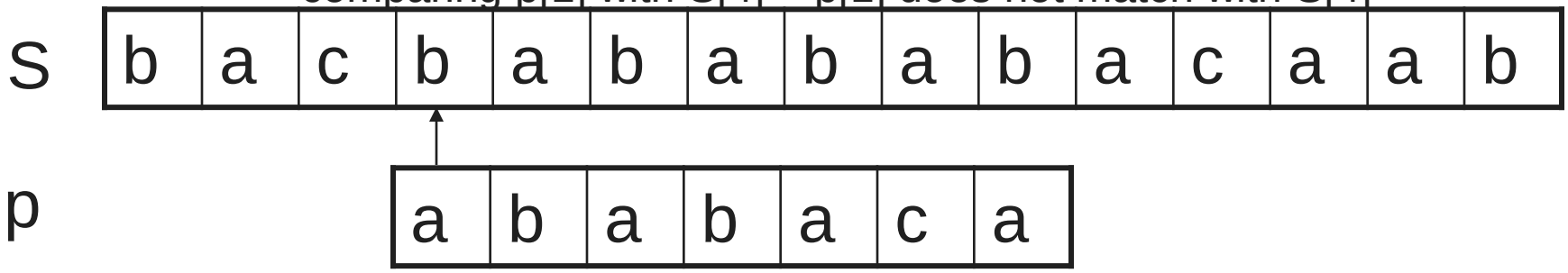
Comparing p[2] with S[3] p[2] does not match with S[3]



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

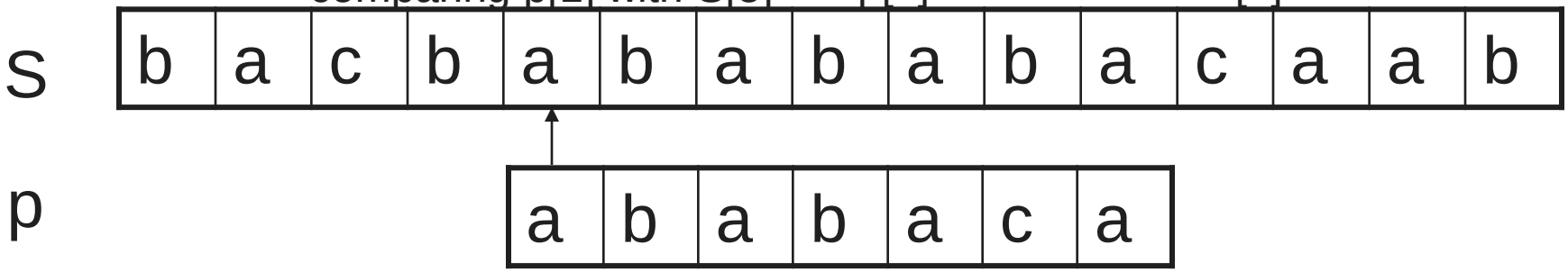
Step 4: i = 4, q = 0

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

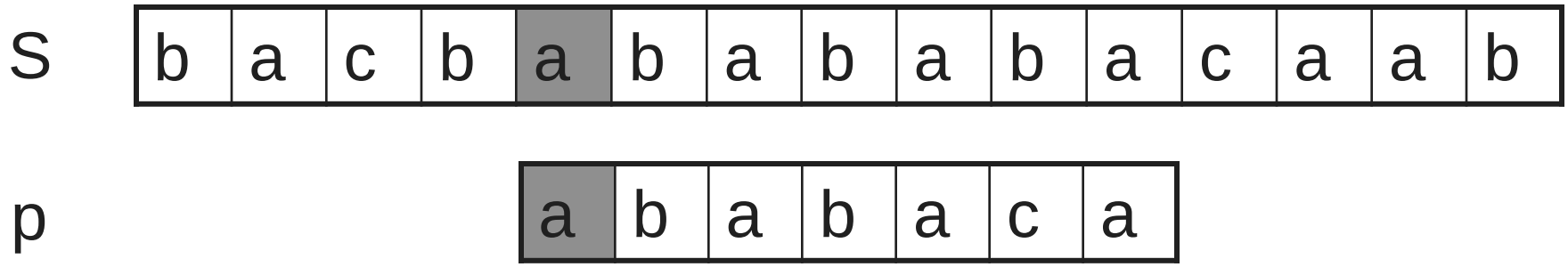


Step 5: i = 5, q = 0

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

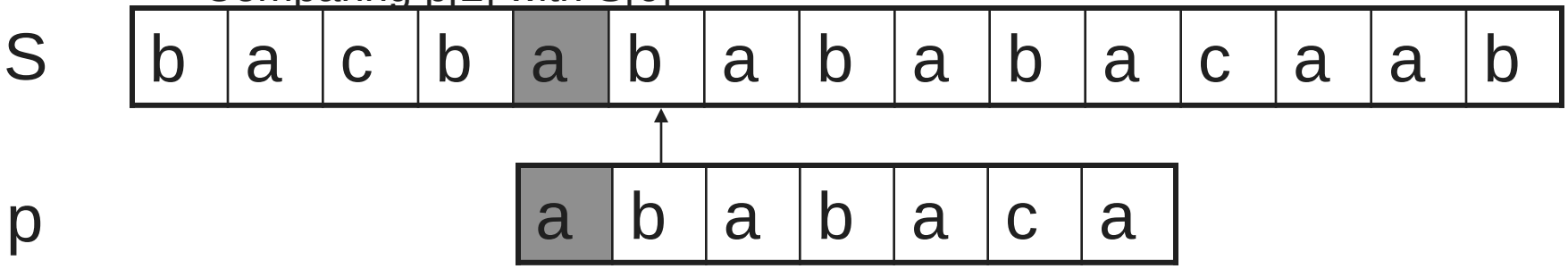


Step 6: i = 6, q = 1



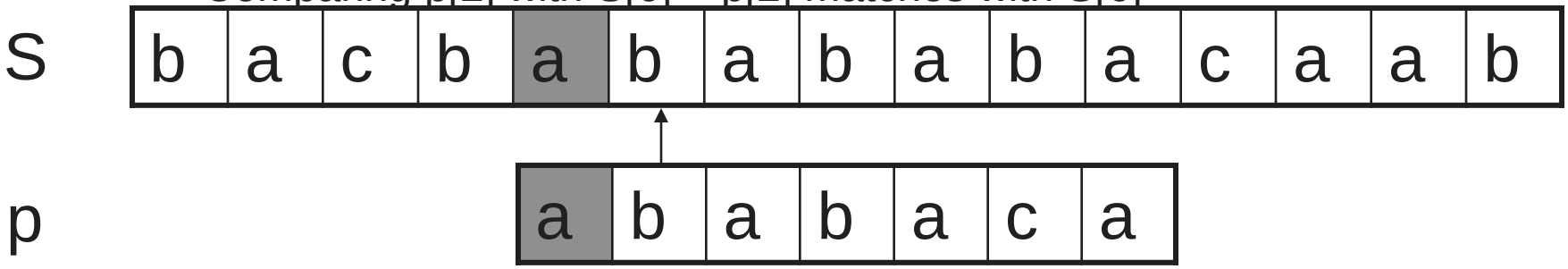
Step 6: i = 6, q = 1

Comparing p[2] with S[6]



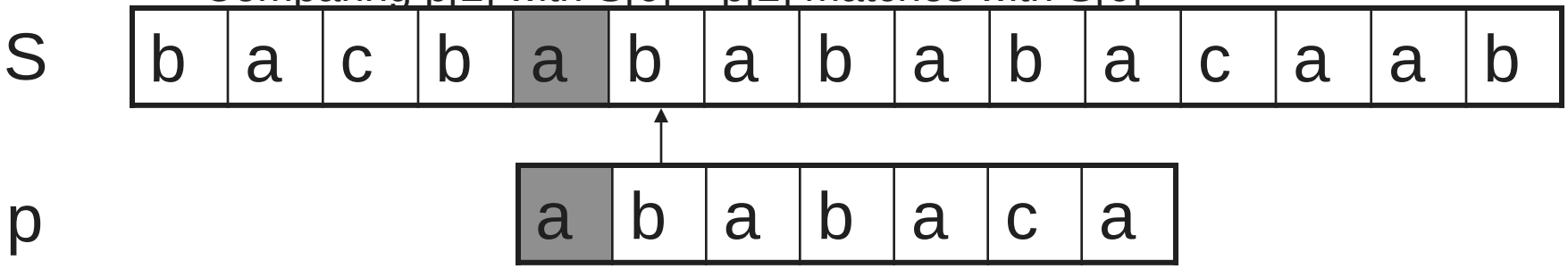
Step 6: i = 6, q = 1

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

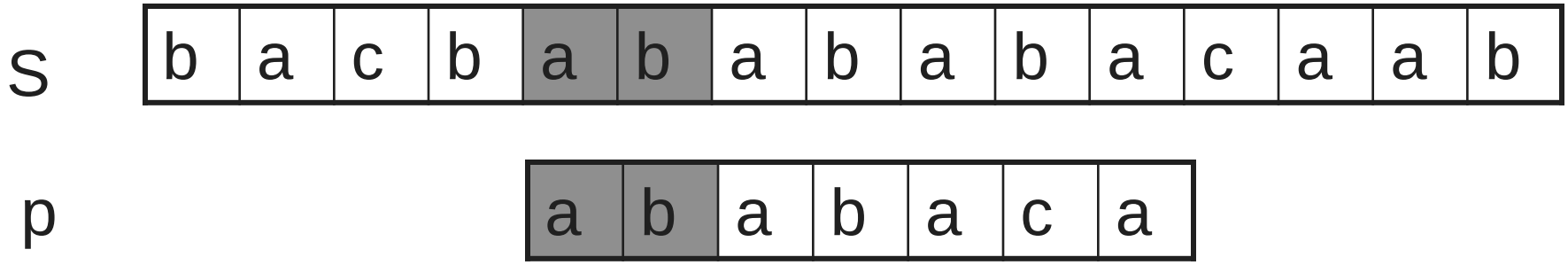


Step 6: i = 6, q = 1

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

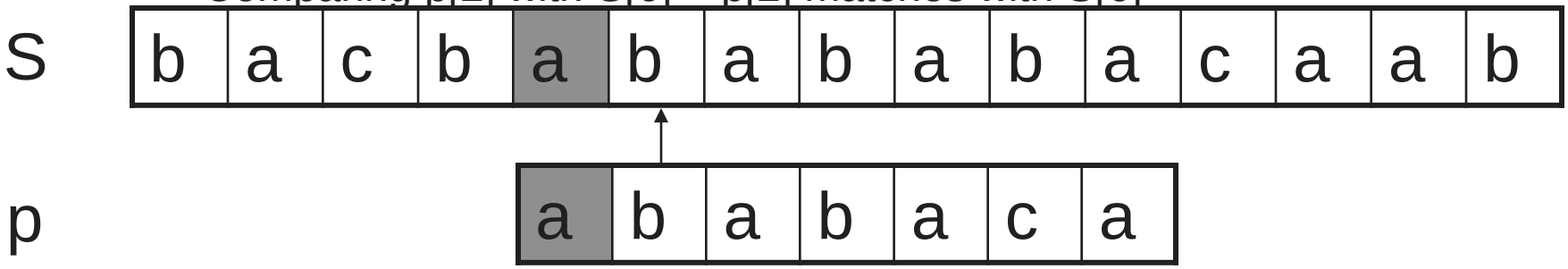


Step 7: i = 7, q = 2



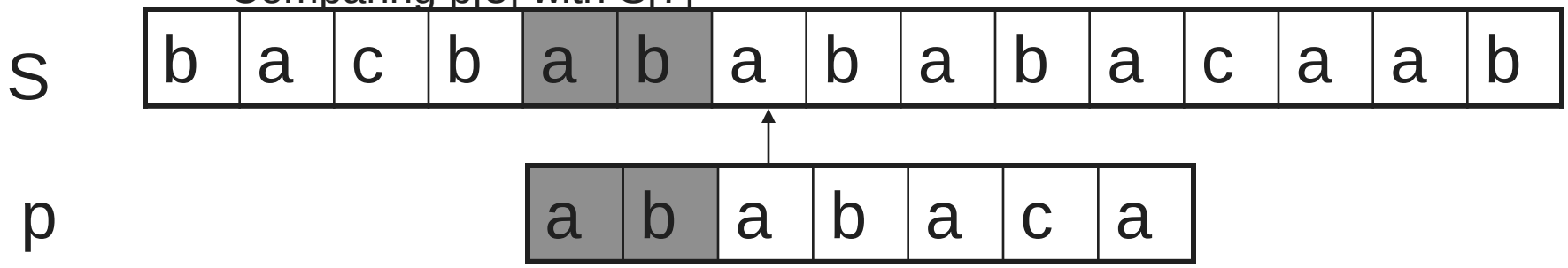
Step 6: i = 6, q = 1

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



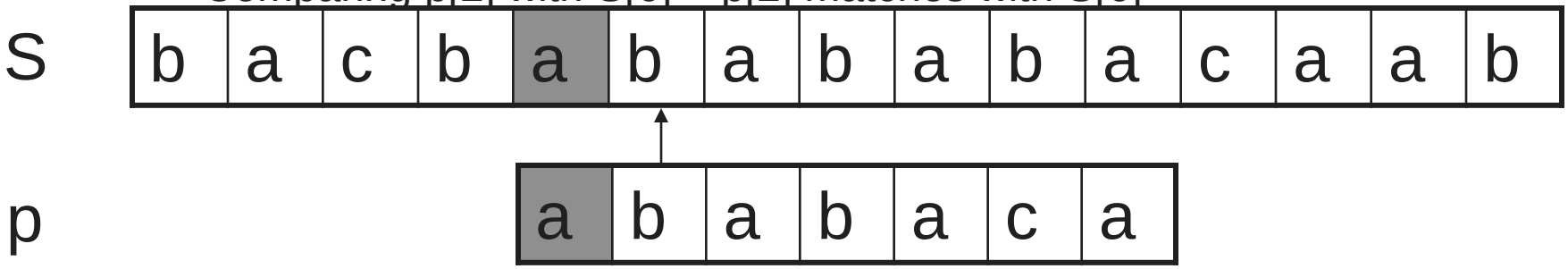
Step 7: i = 7, q = 2

Comparing p[3] with S[7]



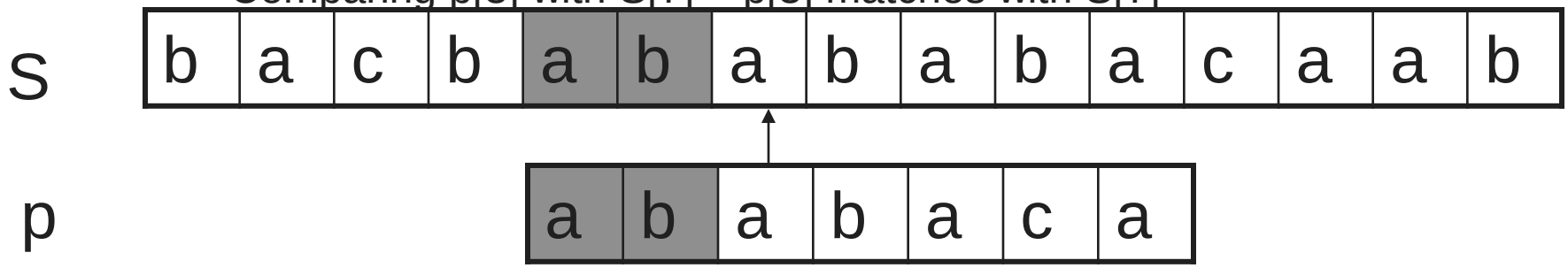
Step 6: i = 6, q = 1

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



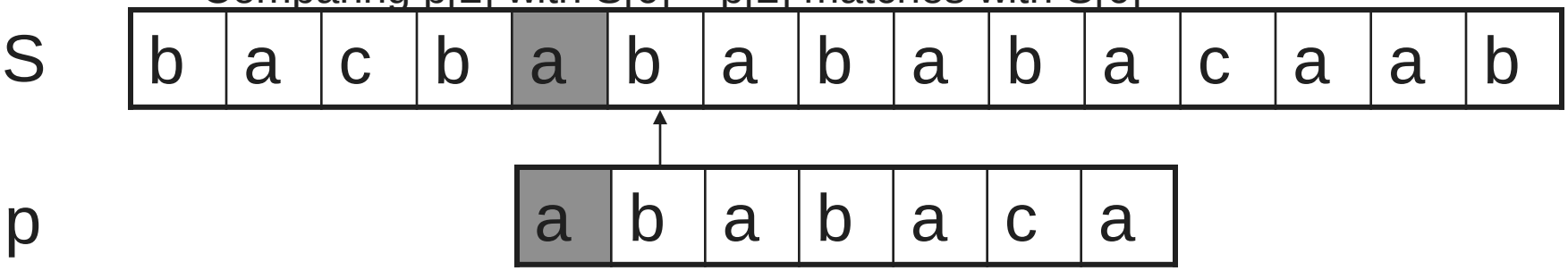
Step 7: i = 7, q = 2

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



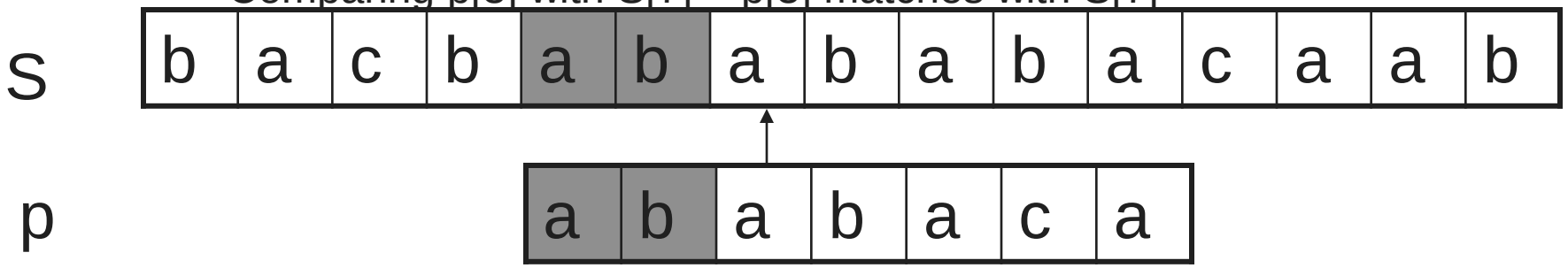
Step 6: i = 6, q = 1

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

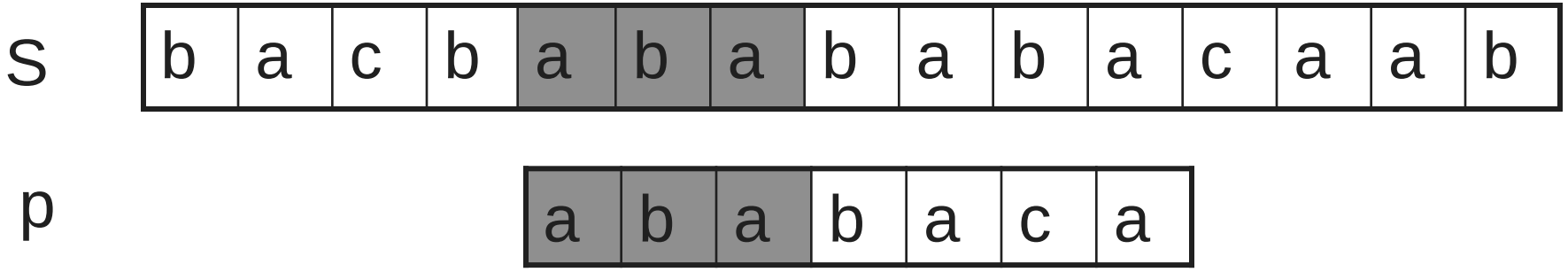


Step 7: i = 7, q = 2

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

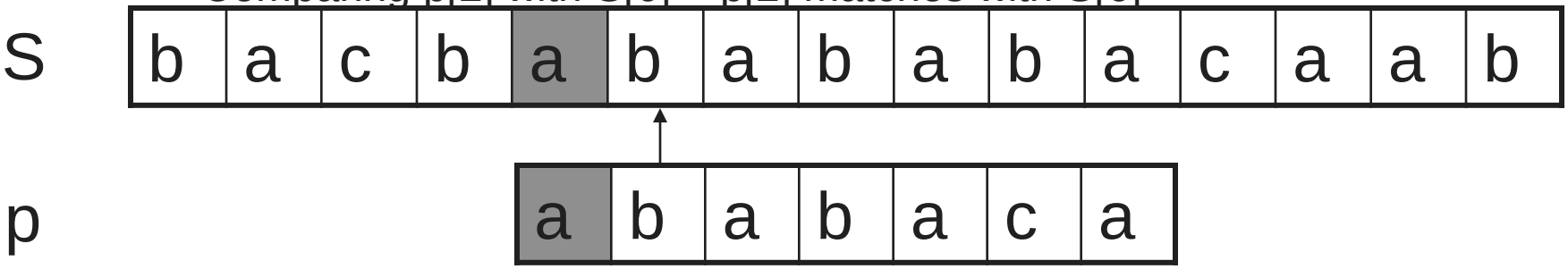


Step 8: i = 8, q = 3



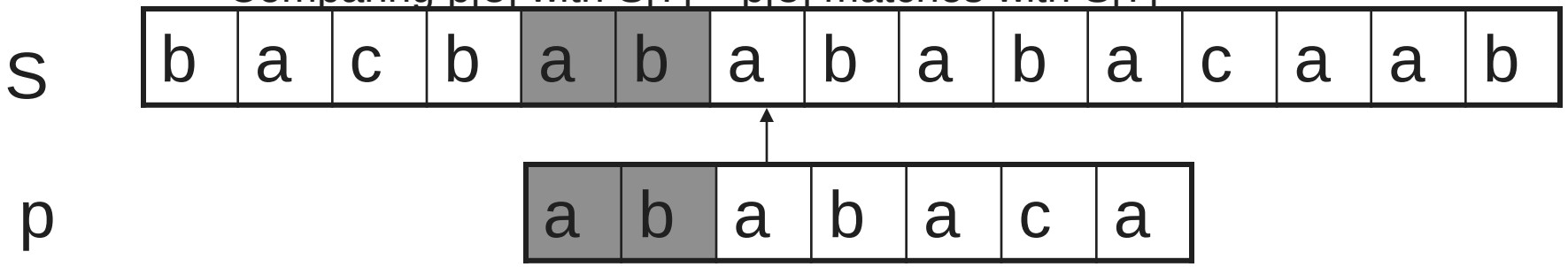
Step 6: i = 6, q = 1

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



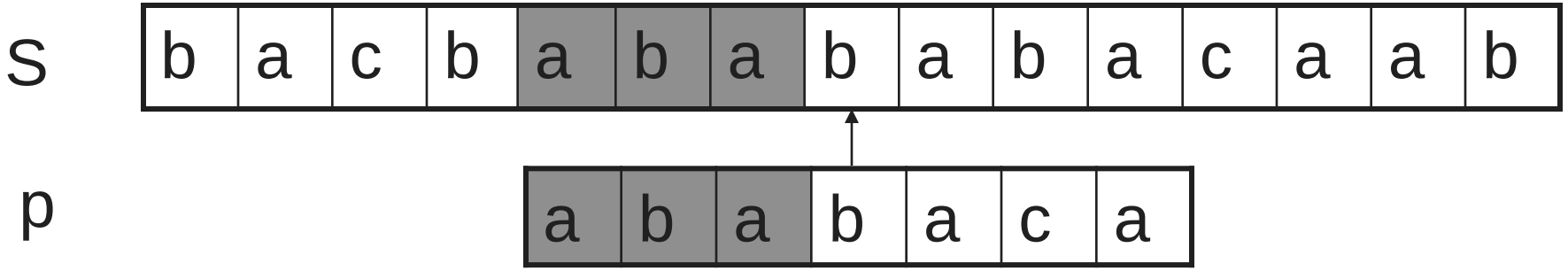
Step 7: i = 7, q = 2

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



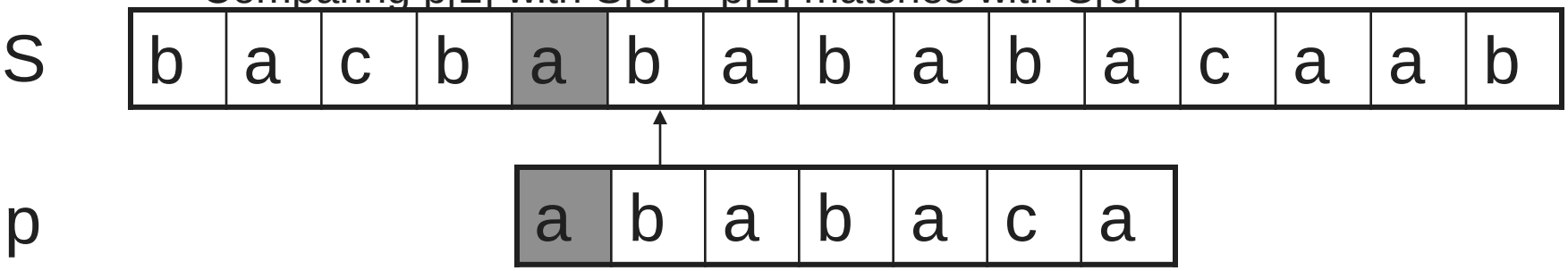
Step 8: i = 8, q = 3

Comparing p[4] with S[8]



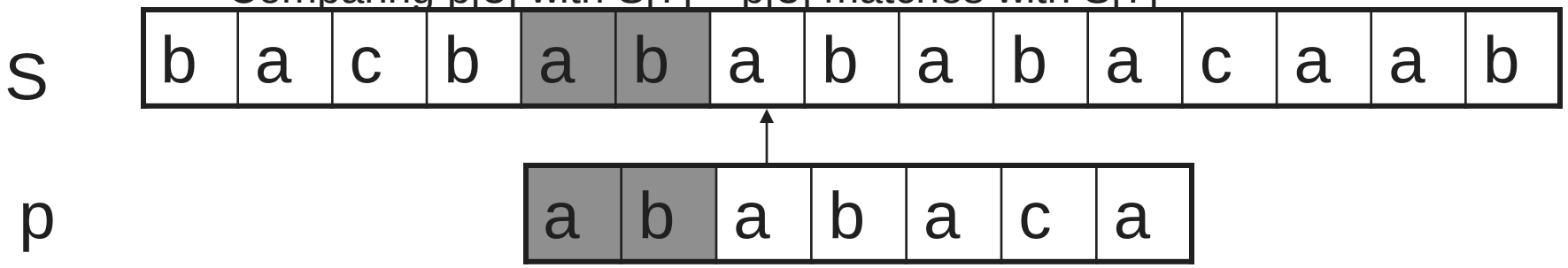
Step 6: i = 6, q = 1

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



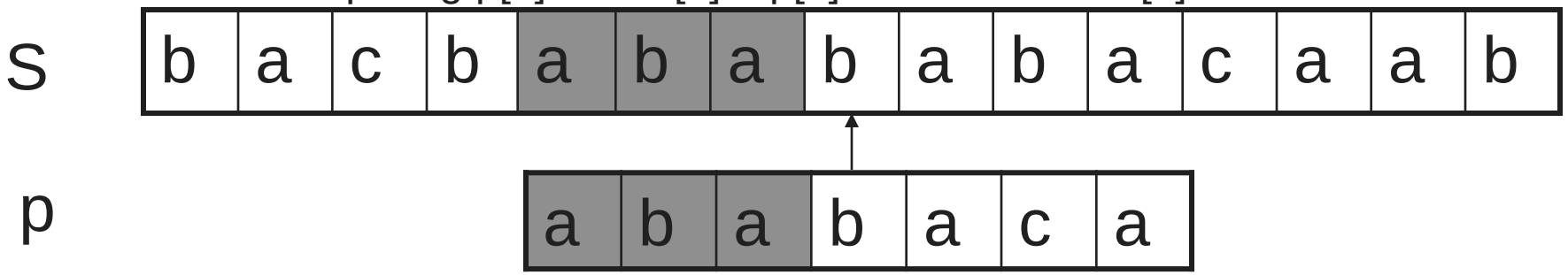
Step 7: i = 7, q = 2

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

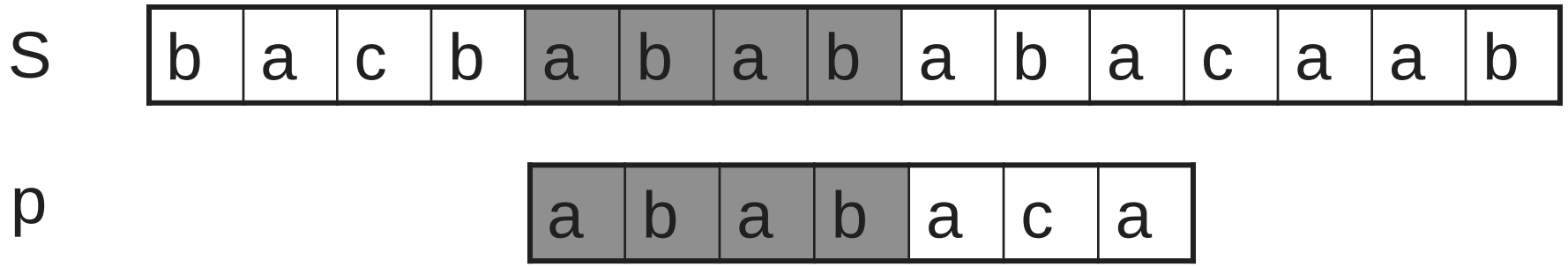


Step 8: i = 8, q = 3

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

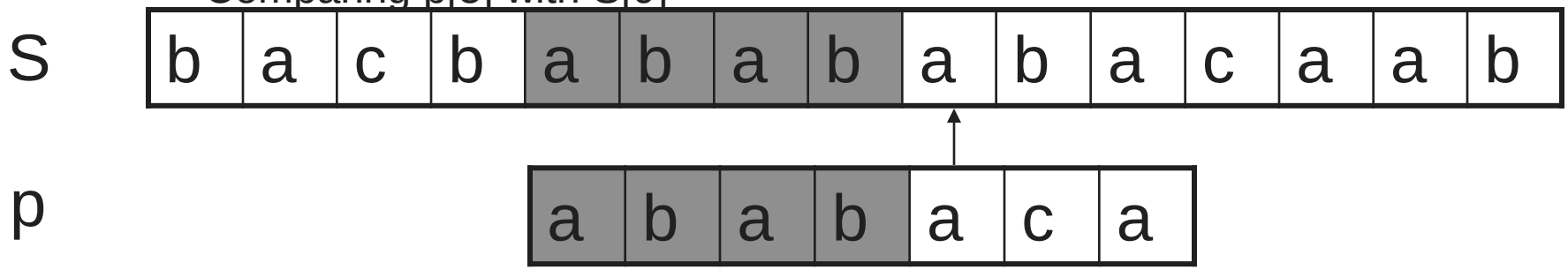


Step 9: i = 9, q = 4

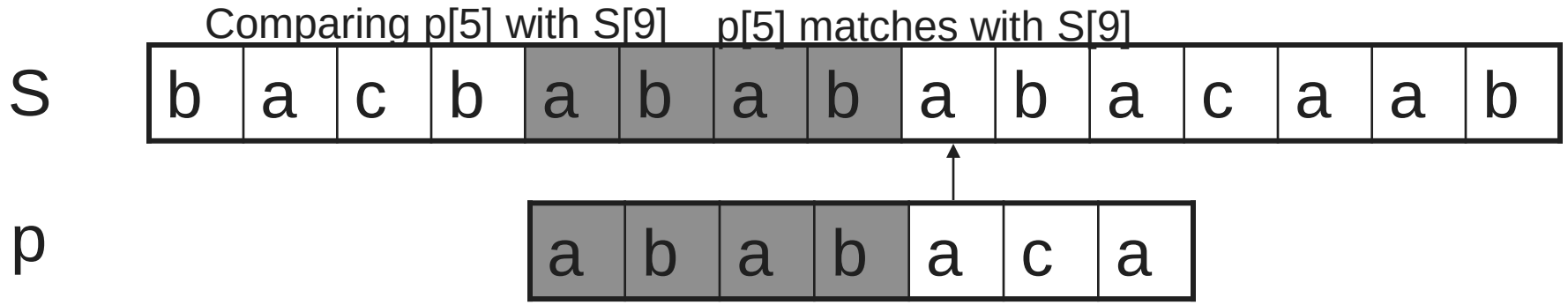


Step 9: i = 9, q = 4

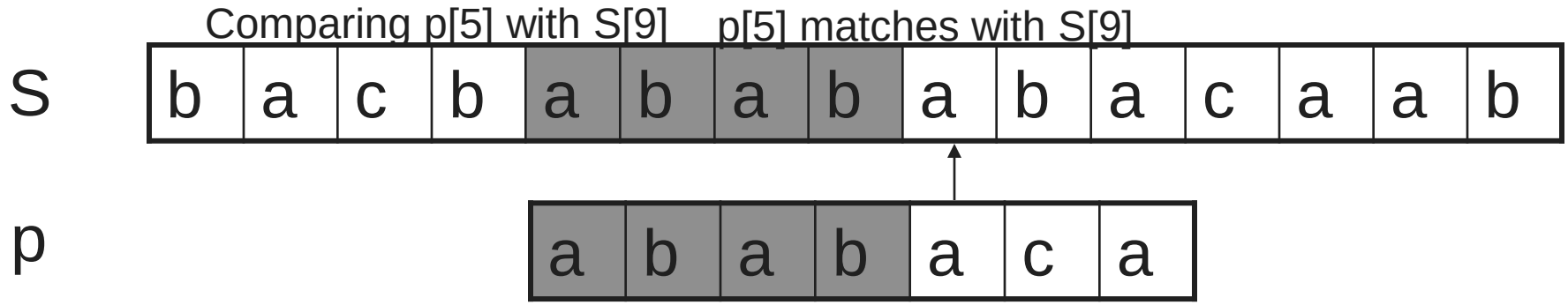
Comparing p[5] with S[9]



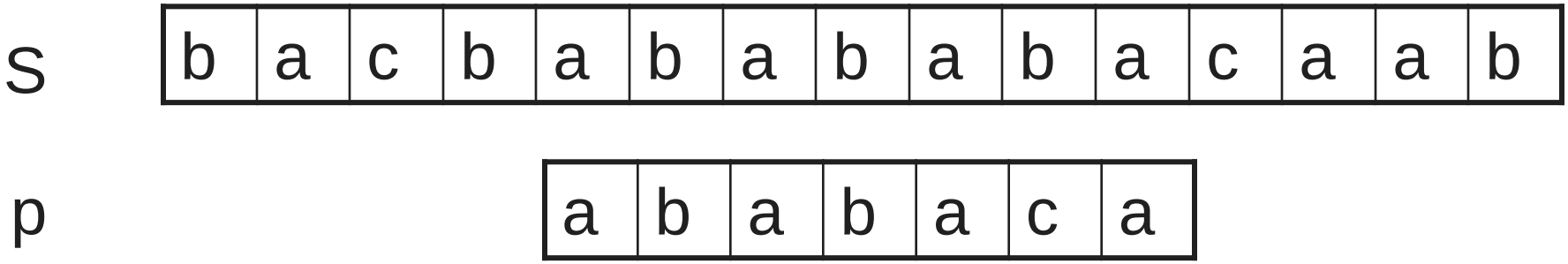
Step 9: i = 9, q = 4


Comparing p[5] with S[9]
p[5] matches with S[9]
S
b a c b a b a b a b a c a a b
p
a b a b a c a

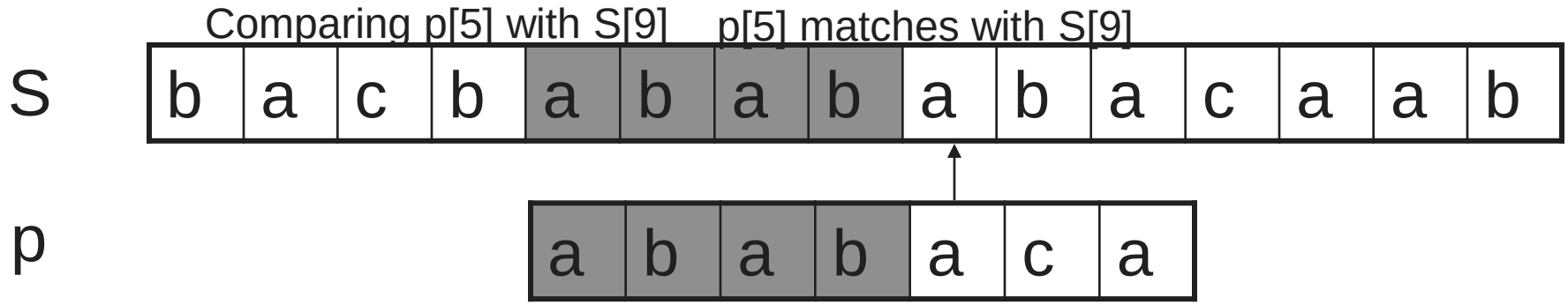
Step 9: i = 9, q = 4



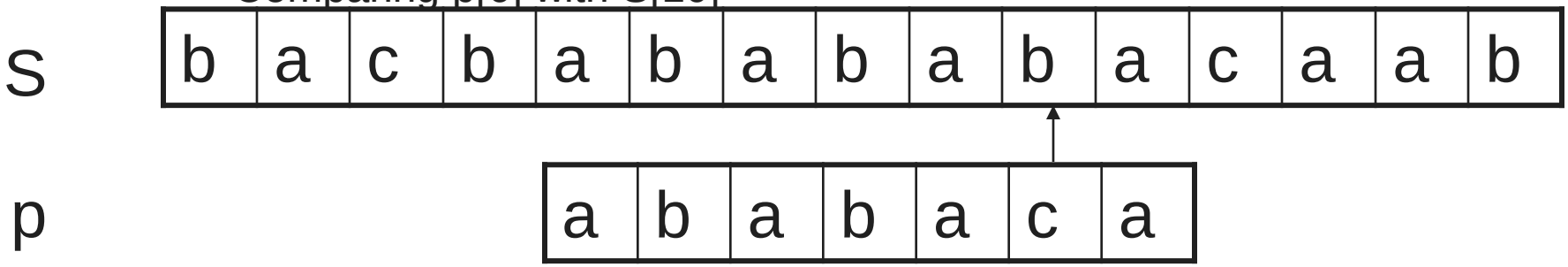
Step 10: i = 10, q = 5



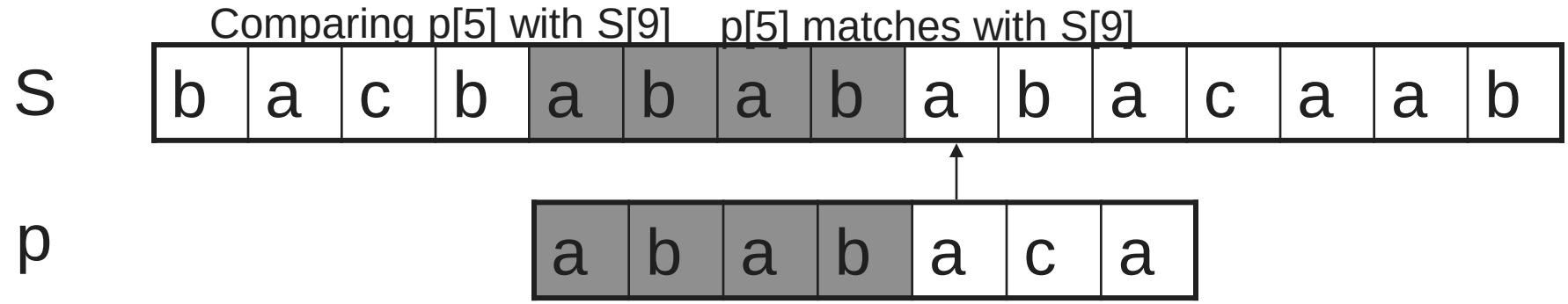
Step 9: i = 9, q = 4



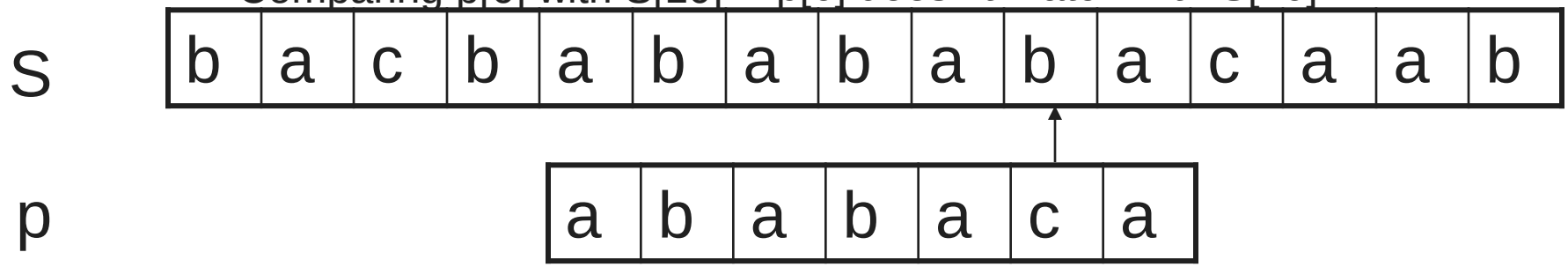
Step 10: i = 10, q = 5



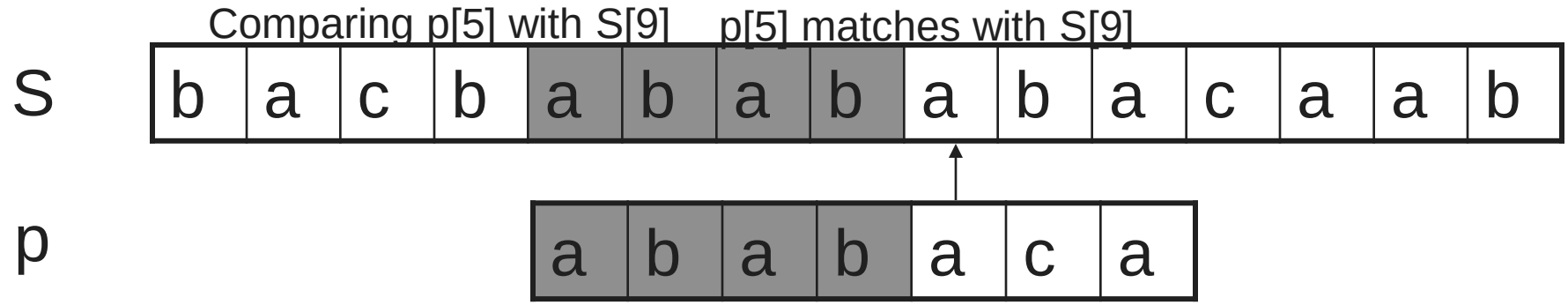
Step 9: i = 9, q = 4



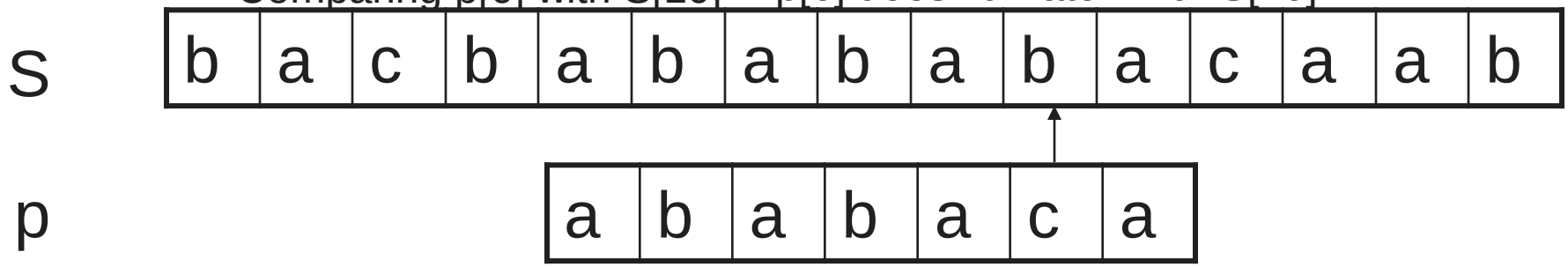
Step 10: i = 10, q = 5



Step 9: i = 9, q = 4



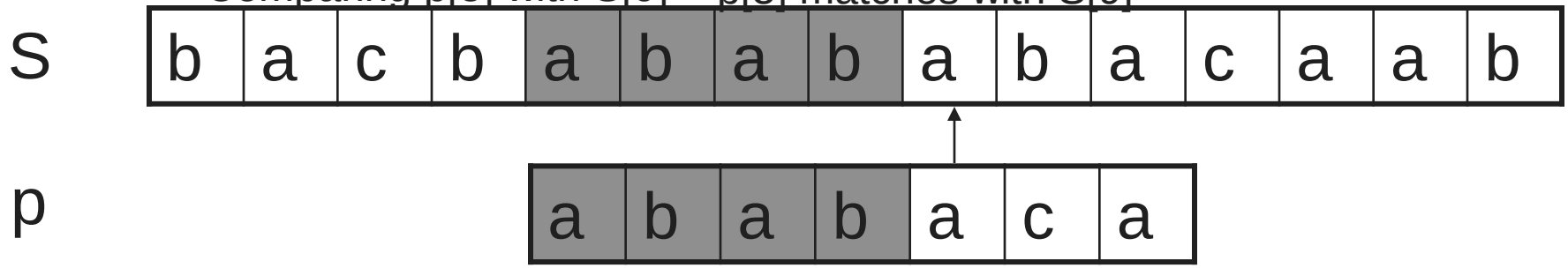
Step 10: i = 10, q = 5



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

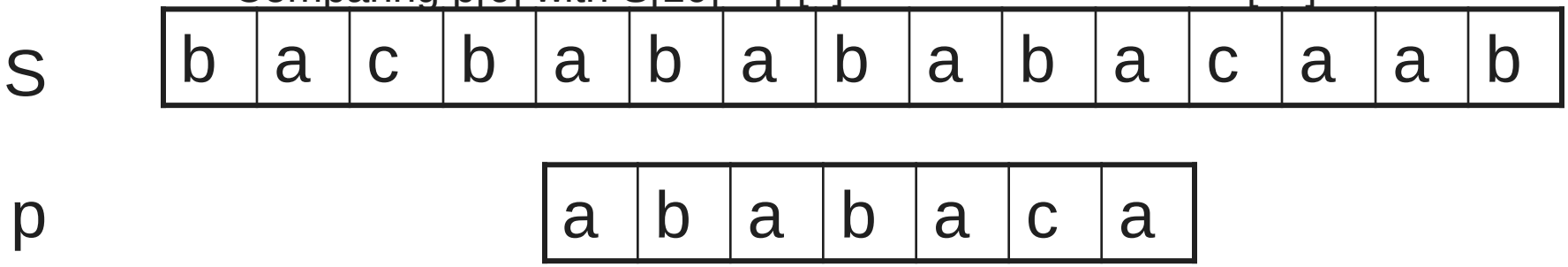
Step 9: i = 9, q = 4

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



Step 10: i = 10, q = 5

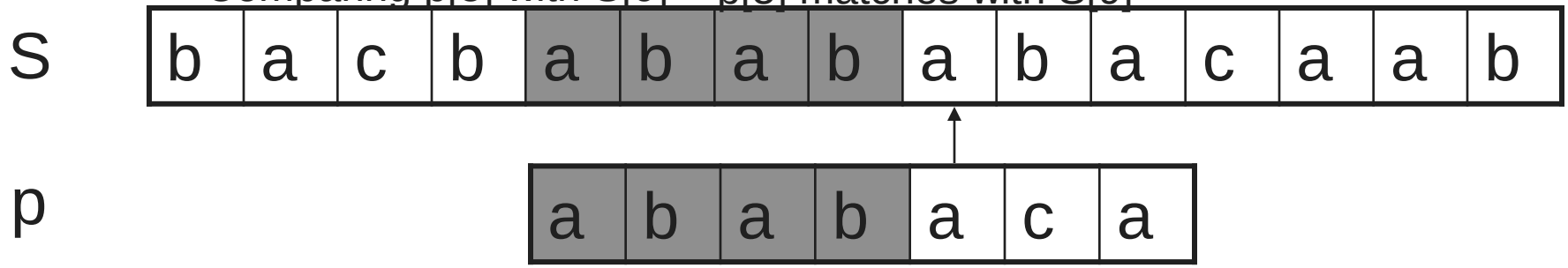
Comparing p[6] with S[10] p[6] doesn't match with S[10]



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

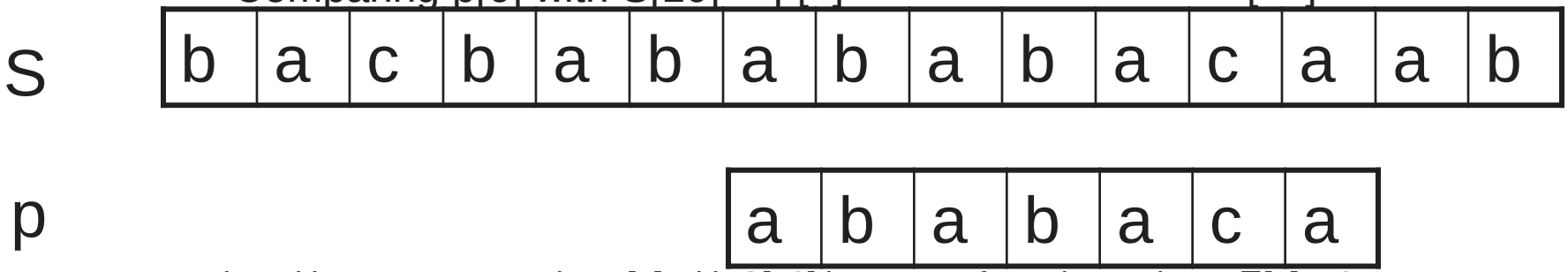
Step 9: i = 9, q = 4

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



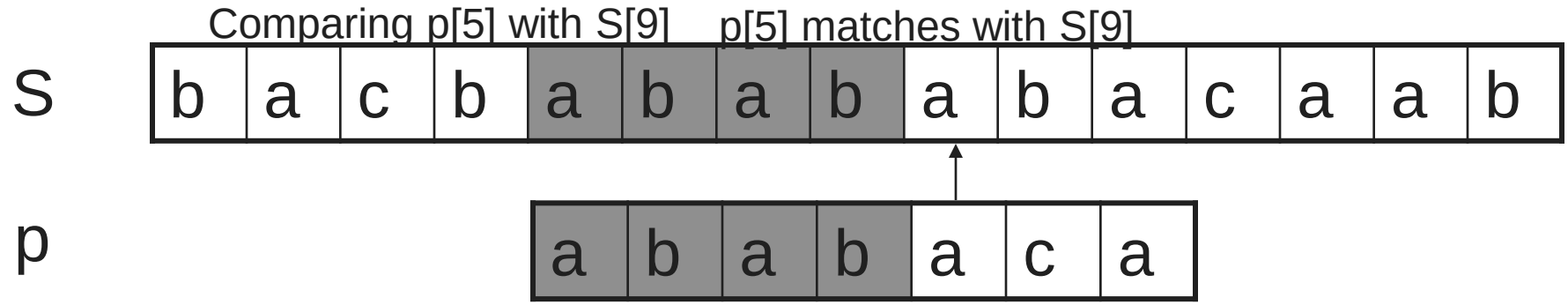
Step 10: i = 10, q = 5

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

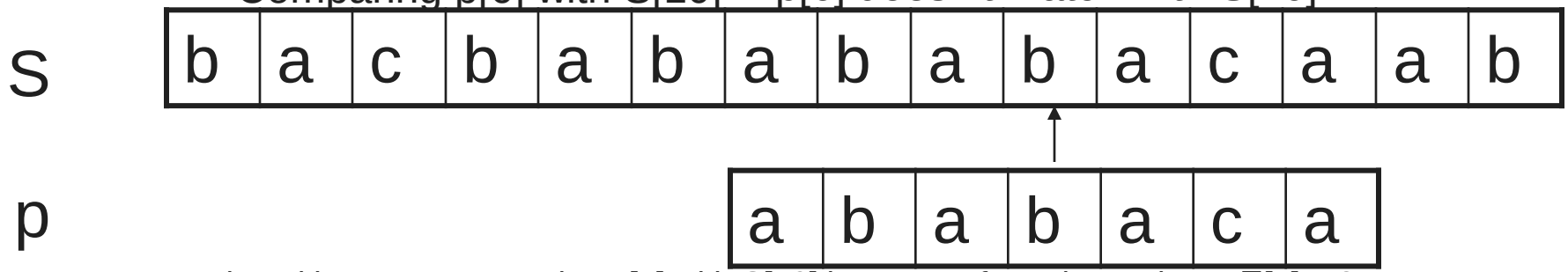


Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

Step 9: i = 9, q = 4

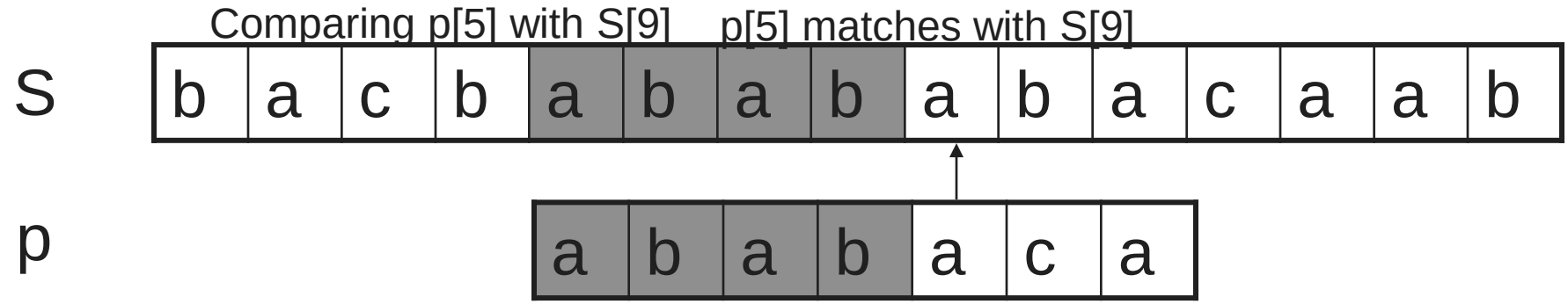


Step 10: i = 10, q = 5

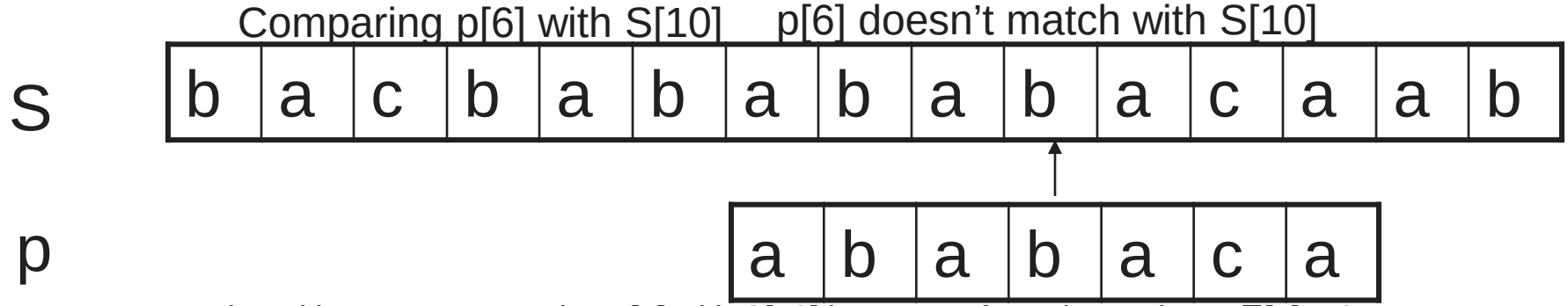


Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

Step 9: i = 9, q = 4

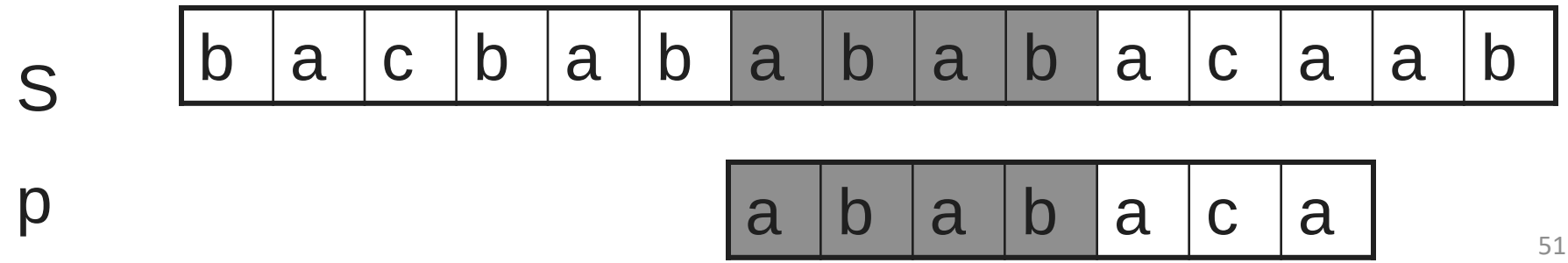


Step 10: i = 10, q = 5

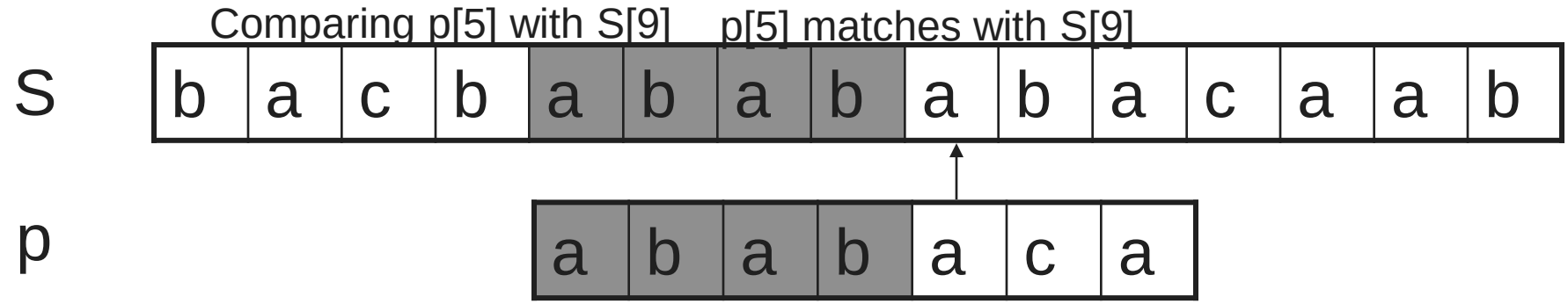


Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

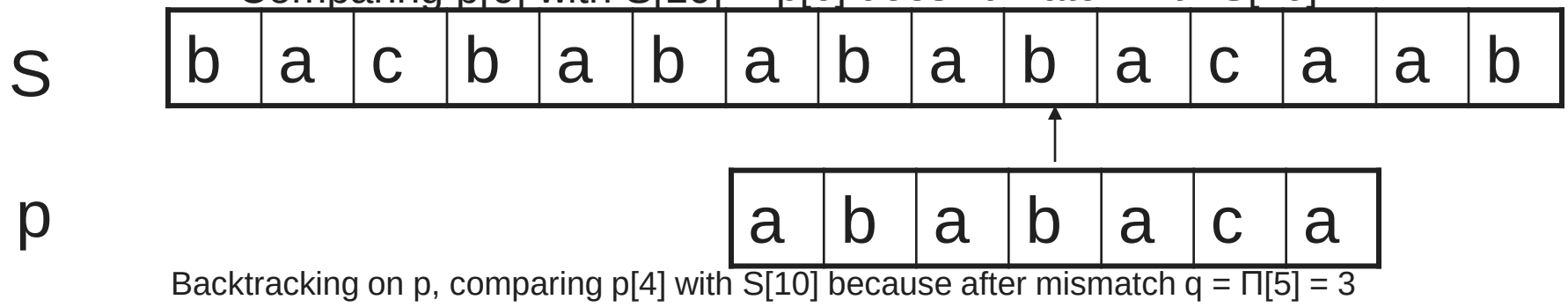
Step 11: i = 11, q = 4



Step 9: i = 9, q = 4

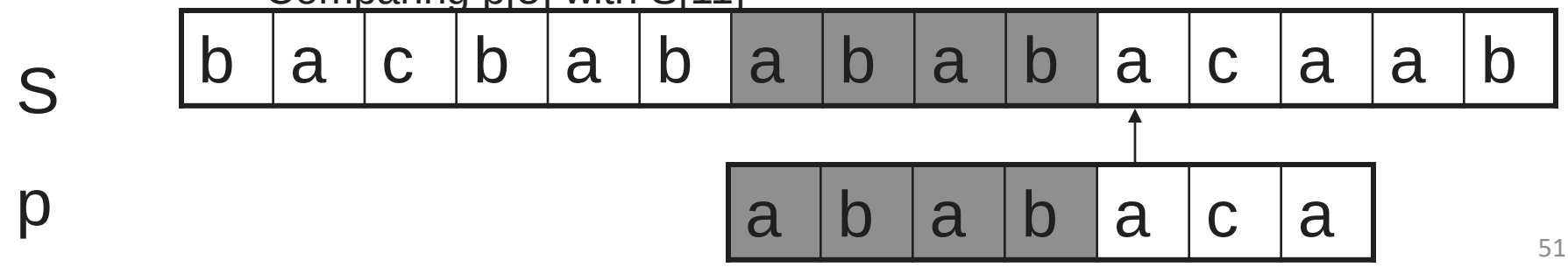


Step 10: i = 10, q = 5

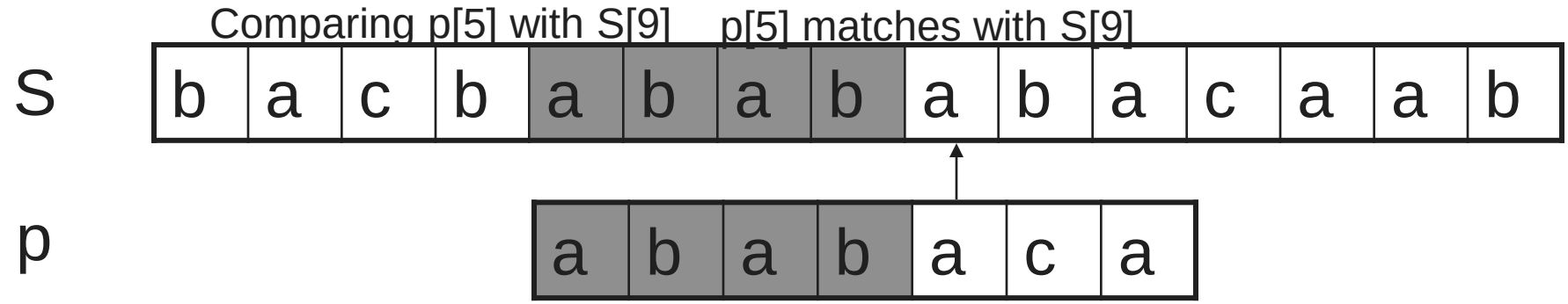


Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

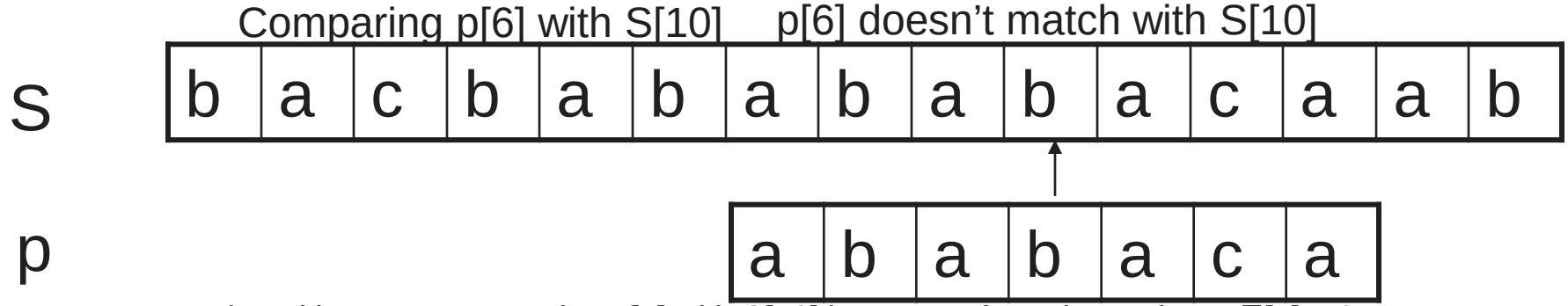
Step 11: i = 11, q = 4



Step 9: i = 9, q = 4

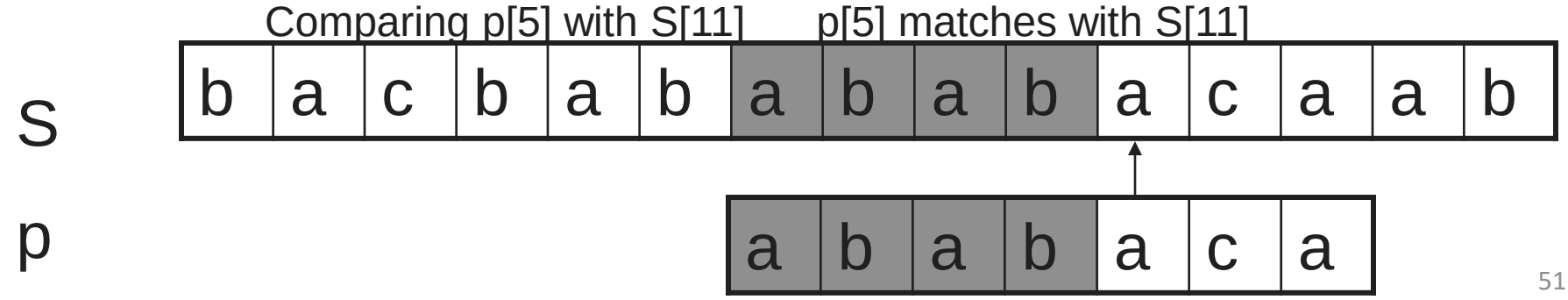


Step 10: i = 10, q = 5

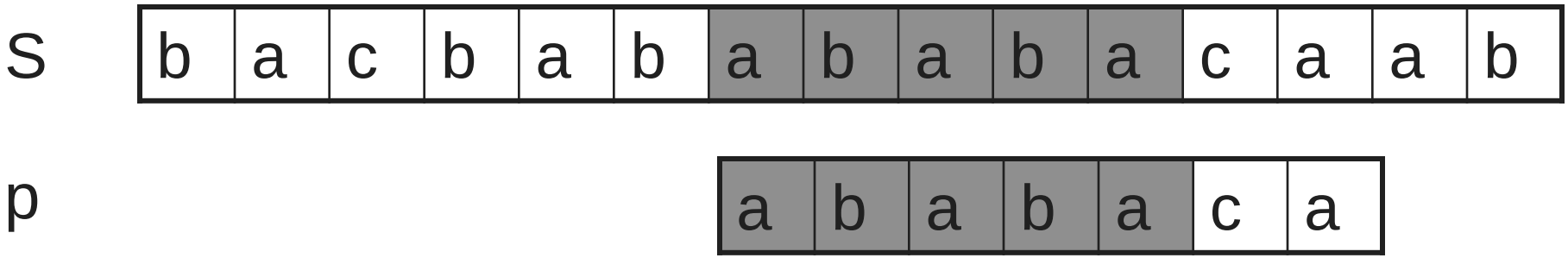


Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

Step 11: i = 11, q = 4

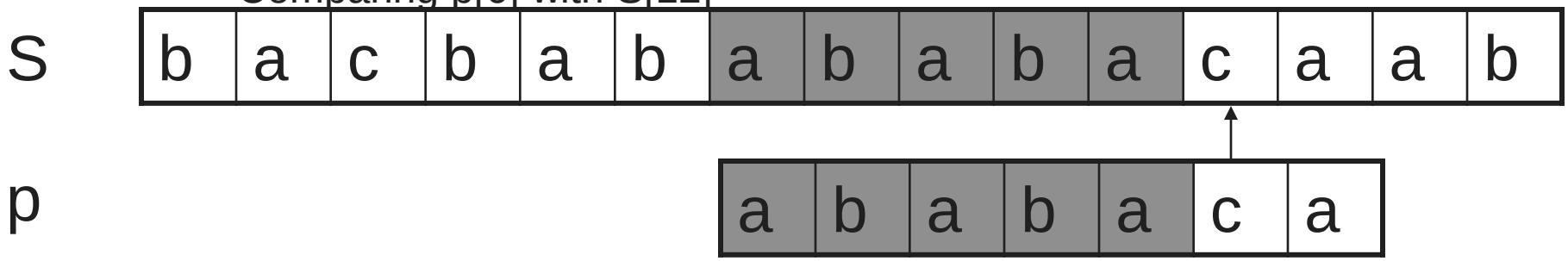


Step 12: i = 12, q = 5

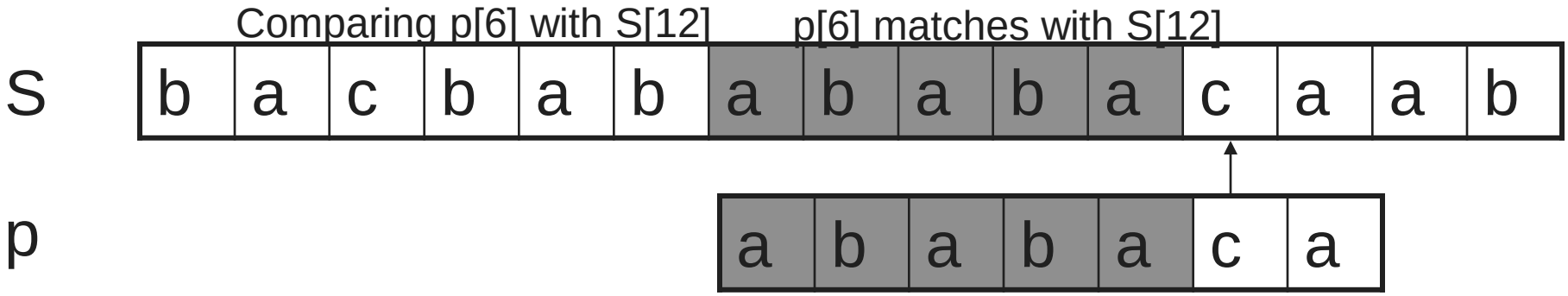


Step 12: i = 12, q = 5

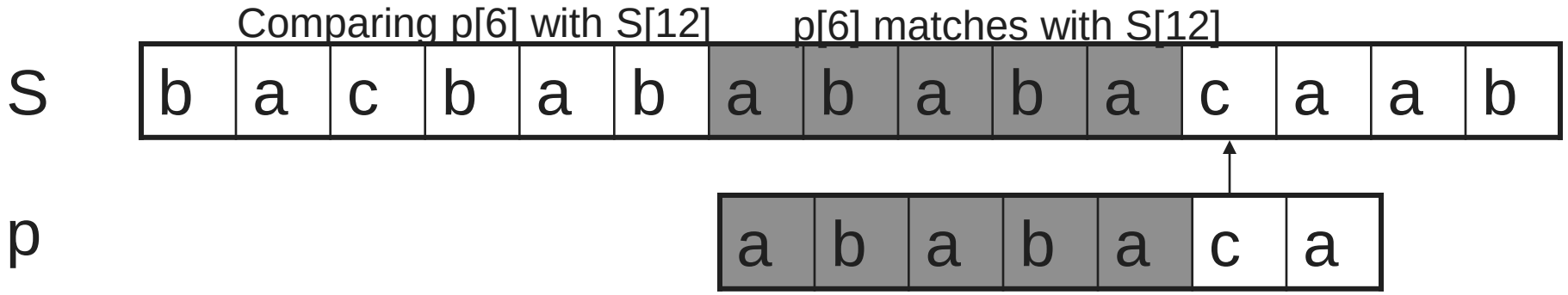
Comparing p[6] with S[12]



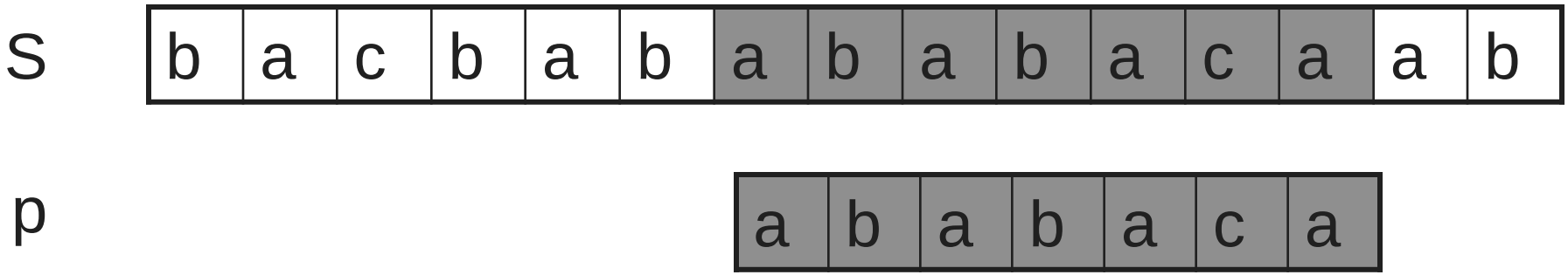
Step 12: i = 12, q = 5



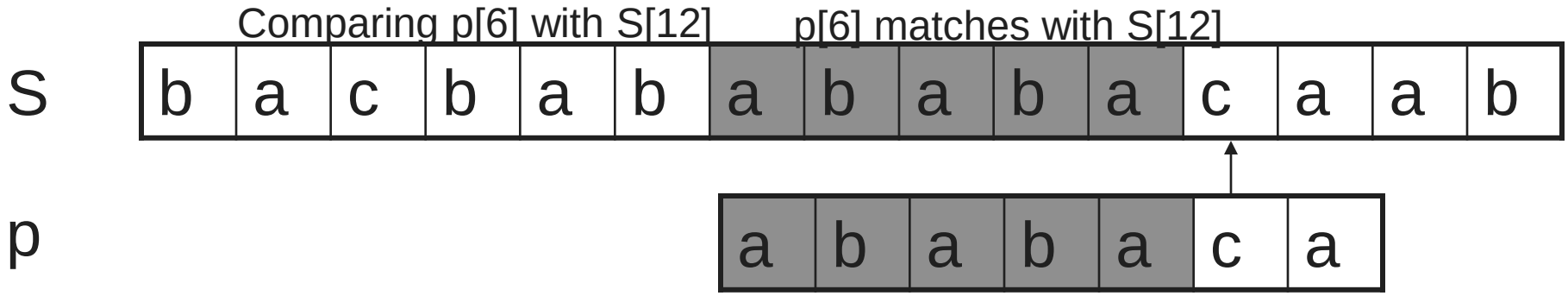
Step 12: i = 12, q = 5



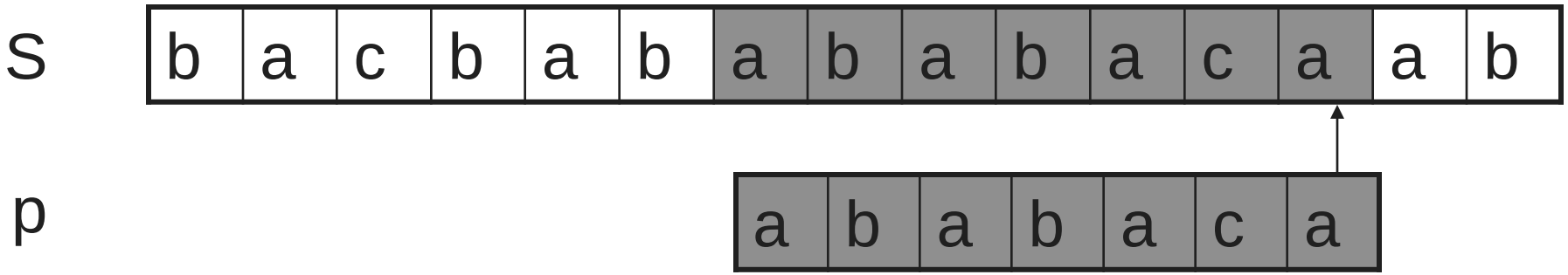
Step 13: i = 13, q = 6



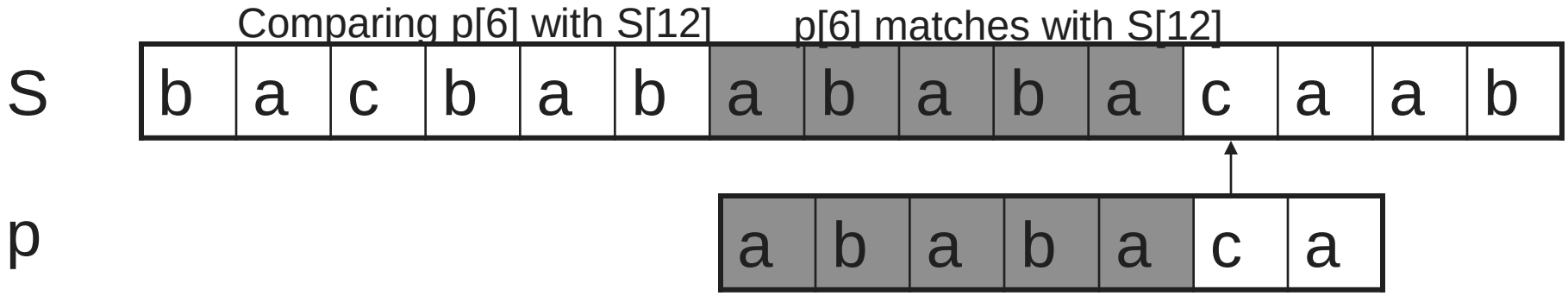
Step 12: i = 12, q = 5



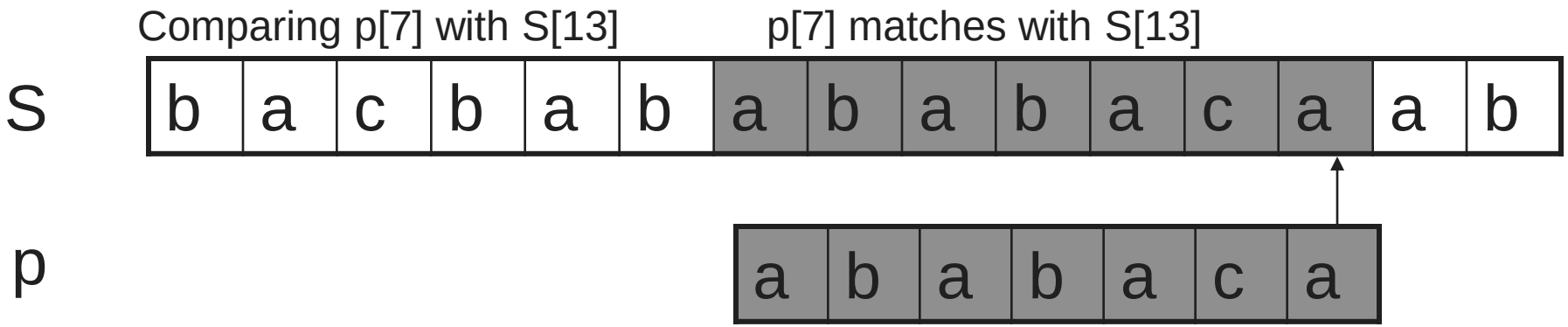
Step 13: i = 13, q = 6



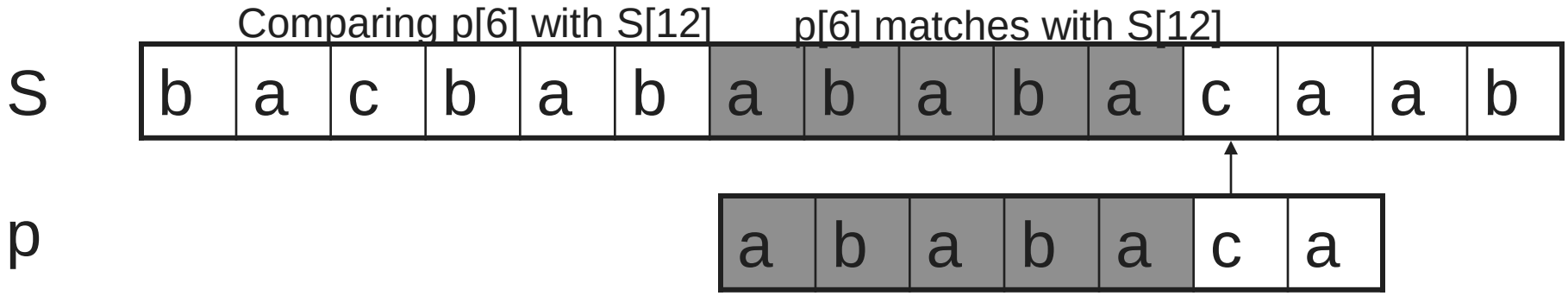
Step 12: i = 12, q = 5



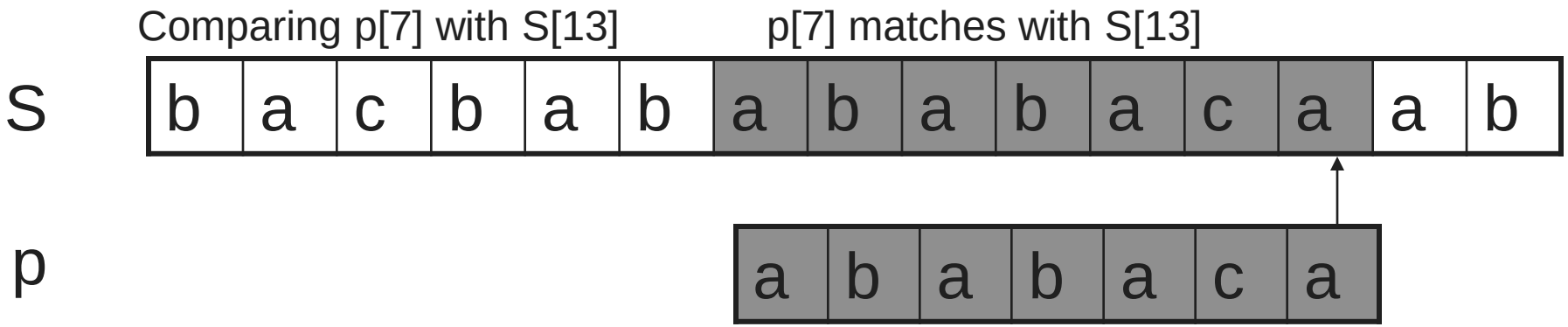
Step 13: i = 13, q = 6



Step 12: i = 12, q = 5



Step 13: i = 13, q = 6



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.

Knuth-Morris-Pratt (KMP) Algorithm

- Information stored in prefix function
  - Can speed up both the naïve algorithm and the finite-automaton matcher
- KMP Algorithm
  - 2 parts: KMP-MATCHER, PREFIX
- Running time
  - PREFIX takes $O(m)$
  - KMP-MATCHER takes $O(m+n)$

Knuth-Morris-Pratt (KMP) Algorithm

```
KMP-MATCHER(T, P)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0                                  // number of characters matched
 5  for i = 1 to n                         // scan the text from left to right
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]                       // next character does not match
 8      if P[q + 1] == T[i]
 9          q = q + 1                      // next character matches
10      if q == m                          // is all of P matched?
11          print "Pattern occurs with shift" i − m
12          q = π[q]                       // look for the next match
```

Knuth-Morris-Pratt (KMP) Algorithm

```
COMPUTE-PREFIX-FUNCTION(P)
 1  m = P.length
 2  let π[1..m] be a new array
 3  π[1] = 0
 4  k = 0
 5  for q = 2 to m
 6       while k > 0 and P[k + 1] ≠ P[q]
 7            k = π[k]
 8       if P[k + 1] == P[q]
 9            k = k + 1
10       π[q] = k
11  return π
```

Exercises

Ch. 32 3-1
Ch. 32 4-1

Boyer-Moore Algorithm

- Published in 1977
- The longer the pattern is, the faster it works
- Starts from the end of pattern, while KMP starts from the beginning
- Works best for character string, while KMP works best for binary string
- KMP and Boyer-Moore
  - Preprocessing existing patterns
  - Searching patterns in input strings