Imperial College
London

CO202 – Software Engineering – Algorithms

String Matching

Ben Glocker
Huxley Building, Room 377
b.glocker@imperial.ac.uk

The material is partly based on previous lectures by Prof Alex Wolf

Motivation

In text-editing programs, a frequent problem is **finding** occurrences of a particular **pattern** in a text document

Efficient **string matching** algorithms are required for responsive programs and applications

There are two types of string matching

- approximate (e.g. Levenshtein distance)
- **exact** (this lecture)

Some Applications

- Text-editing / word processors
- Virus scanning
- Text-based information retrieval
- Digital libraries
- Web search
- Computational biology (DNA/gene sequencing)

Example Problem

Given the text of 430 characters

"New York, New York, a helluva town.
The Bronx is up, but the Battery's down.
The people ride in a hole in the groun'.
New York, New York, it's a helluva town!"
— On The Town (Comden and Green)

"These little town blues are melting away
I'll make a brand new start of it in old New York
If I can make it there, I'll make it anywhere
It's up to you, New York, New York!"
— New York, New York (Kander and Epp)

find the string "own"

Example Problem

Given the text of 430 characters

"New York, New York, a helluva town.
The Bronx is up, but the Battery's down.
The people ride in a hole in the groun'.
New York, New York, it's a helluva town!"
— On The Town (Comden and Green)

"These little town blues are melting away
I'll make a brand new start of it in old New York
If I can make it there, I'll make it anywhere
It's up to you, New York, New York!"
— New York, New York (Kander and Epp)

find the string "own"

Example Problem

Given the text of 250 characters

0011110101011010011000110101111011010111
0110111001001010101011111011110110000101
1011000001011111101111001100001111000100
1001010010111011101011011110101001100101
0010111001000011111110010011011101011010
0110011011101001010010101000010100111110

find the string “110011”

Example Problem

Given the text of 250 characters

00111101010110100110001101011110110101110110011001010101011111011110110000101011111101111001100000111110001001001011101110101101111010100110010101100100001111111001001101110101101001101100111010010100101000010100111110

find the string “110011”

Formalisation of String Matching

We assume that the **text** is an array $T[1..n]$ with n characters drawn from a finite alphabet Σ.

A **pattern** is an array $P[1..m]$ of length $m \leq n$ where characters are drawn from the same alphabet Σ.

Example alphabets

- $\Sigma = \{0,1\}$
- $\Sigma = \{a, b, \ldots, z\}$

P and T are called **strings** of characters

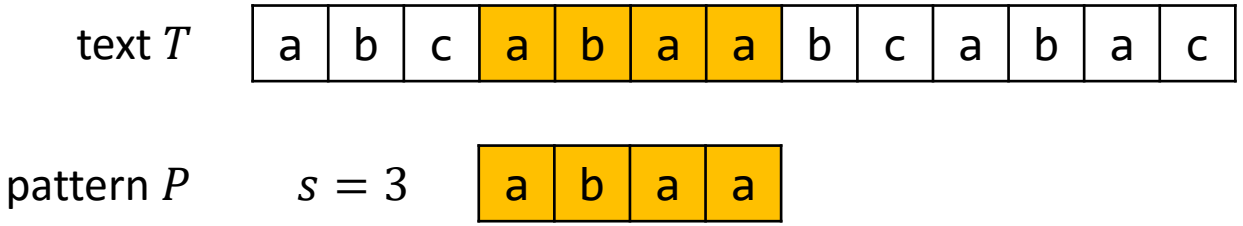
Formalisation of String Matching

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

$$0 \leq s \leq n - m \text{ and } T[s+1..s+m] = P[1..m]$$

that is, if $T[s+j] = P[j]$ for $1 \leq j \leq m$

If P occurs with shift s in T, we call s a **valid shift**; otherwise, we call s an **invalid shift**.



String matching is about finding all valid shifts for given P and T.

Notation and Terminology

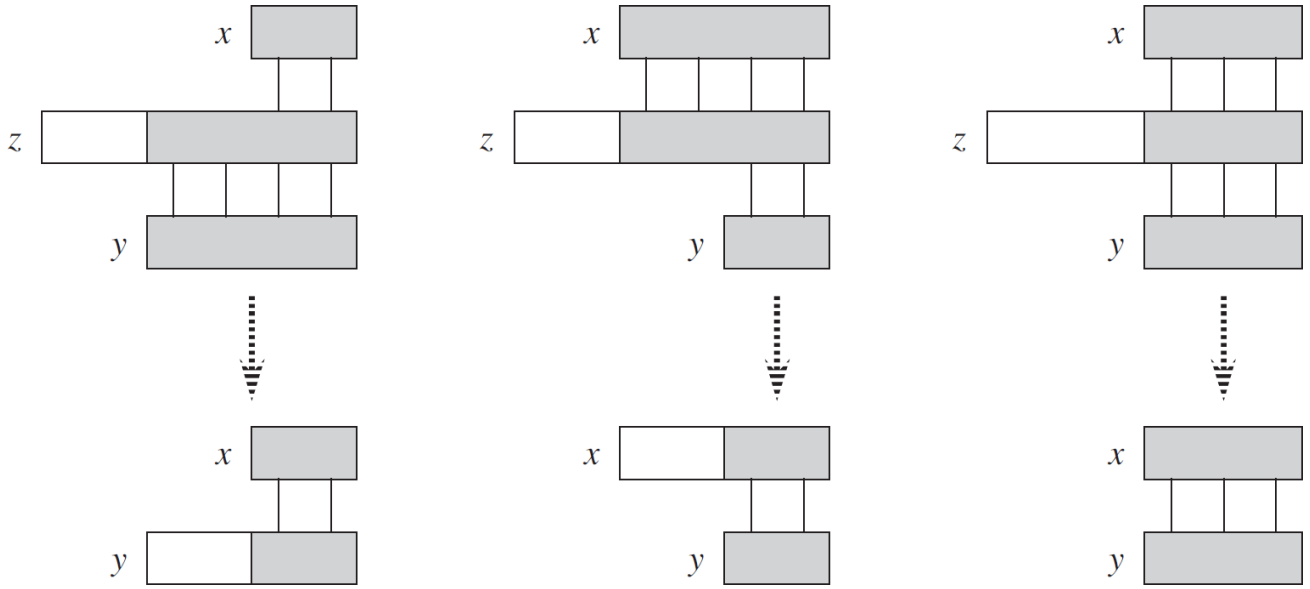
- We denote by Σ^* the set of all finite-length strings formed using characters from alphabet Σ.
- The zero-length **empty string** ϵ belongs to Σ^*
- The **concatenation** of two string x and y, denoted by xy, has length $|x| + |y|$.
- A string w is **prefix** of x, denoted by $w \sqsubset x$, if $x = wy$
- A string w is **suffix** of x, denoted by $w \sqsupset x$, if $x = yw$
- We denote the prefix $P[1..k]$ of pattern $P[1..m]$ by P_k and thus, $P_0 = \epsilon$, $P_m = P = P[1..m]$. Similar for T and prefix T_k
- The string-matching problem is then finding all shifts s in the range $0 \leq \mathrm{s} \leq \mathrm{n} - m$ such that $P \sqsupset T_{s+m}$.

Overlapping-Suffix Lemma

Suppose that x, y, and z are strings such that $x \sqsupset z$ and $y \sqsupset z$.

- If $|x| \leq |y|$, then $x \sqsupset y$.
- If $|x| \geq |y|$, then $y \sqsupset x$.
- If $|x| = |y|$, then $x = y$.

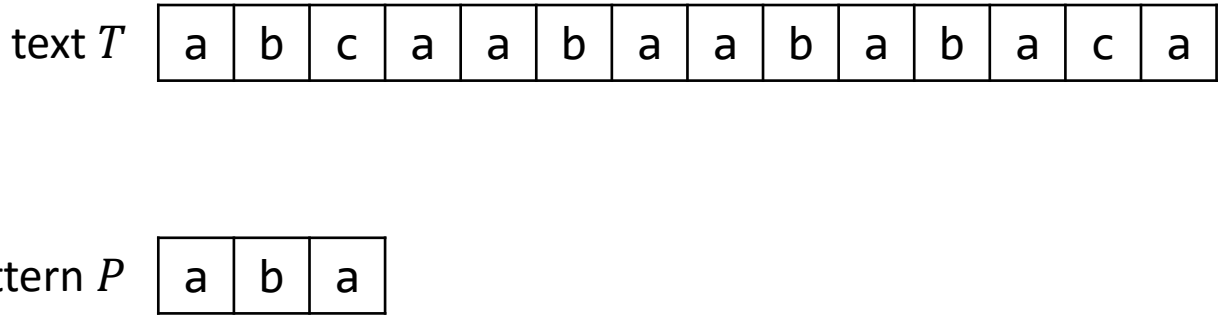
Proof



[Cormen] p.987

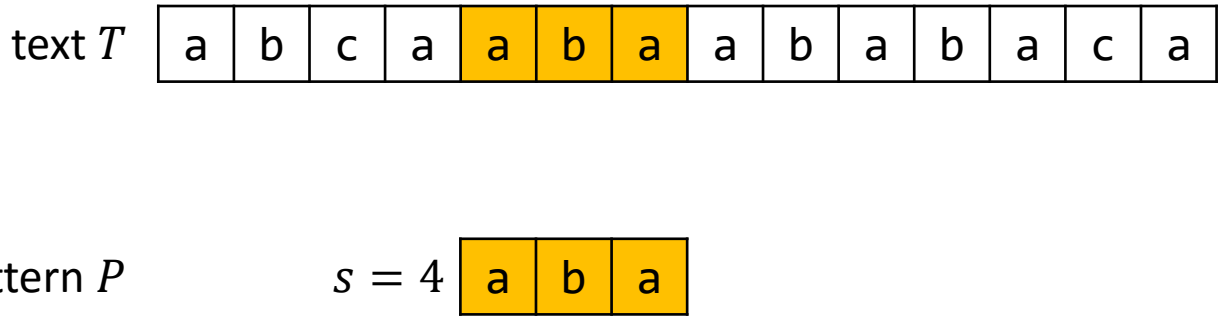
Another Example

Find all valid shifts



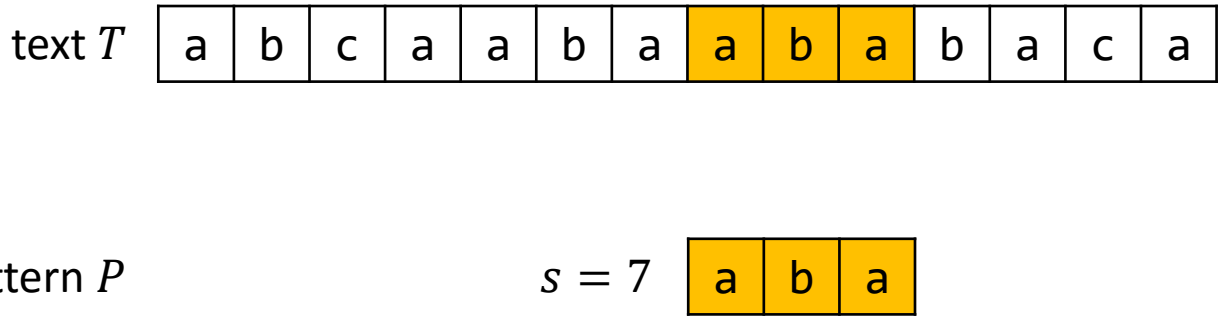
Another Example

Find all valid shifts



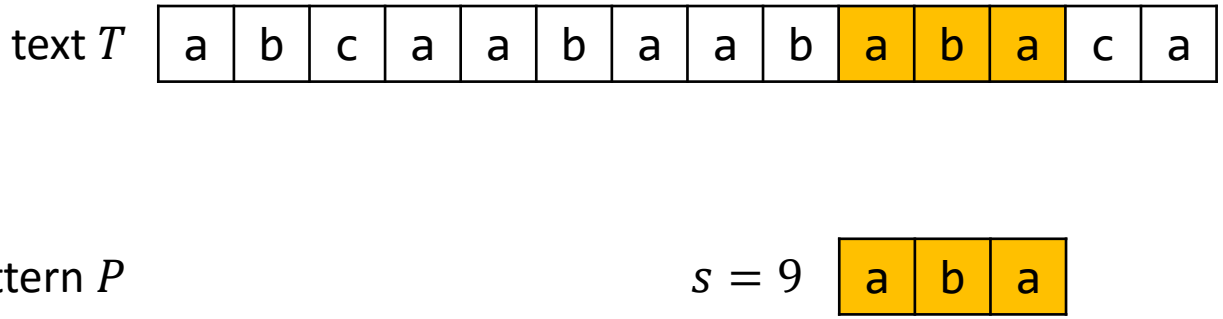
Another Example

Find all valid shifts



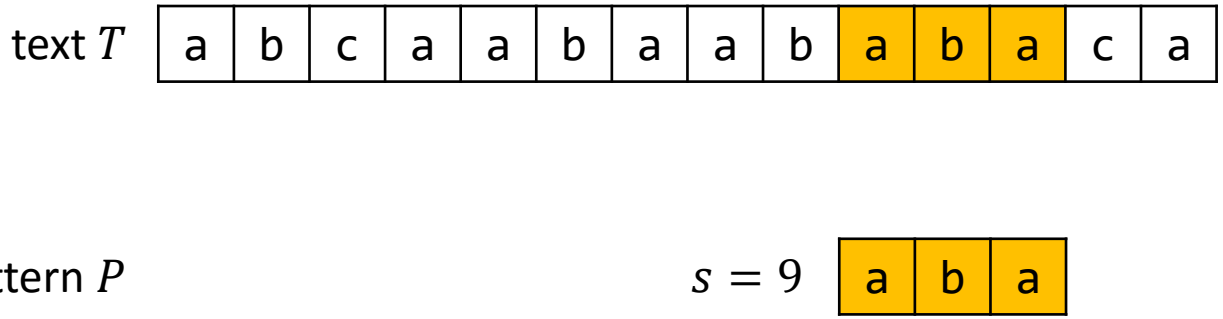
Another Example

Find all valid shifts



Another Example

Find all valid shifts



valid shifts $\{4,7,9\}$

Brute-Force Algorithm

```
NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: for s = 0 to n-m
 4:     if P[1..m] == T[s+1..s+m]
 5:         PRINT(s)
```

Why is this not an optimal procedure?

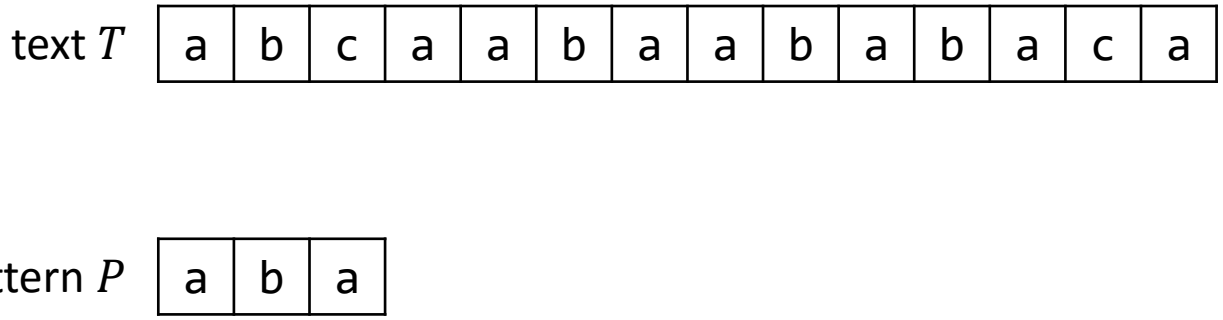
Consider the following example:

$P = aaab$ and $s = 0$ is valid

this means $s = 1,2,3$ cannot be valid

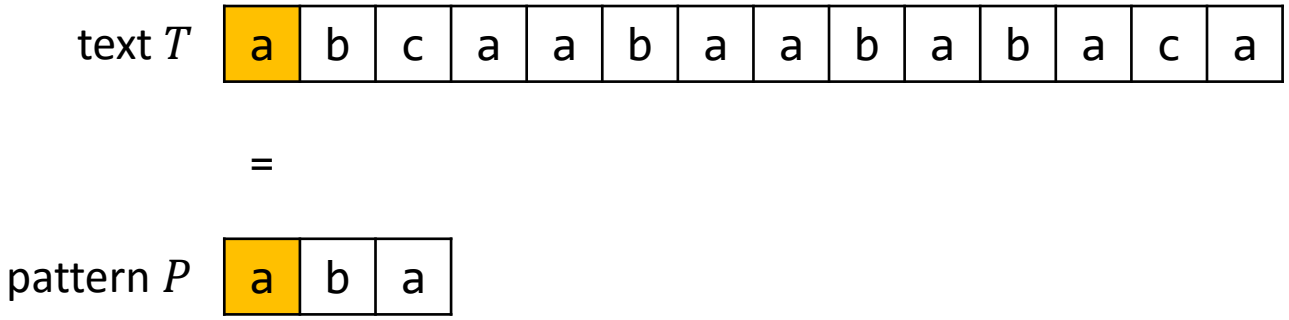
Devising a New Strategy

Let’s observe the behaviour of the brute-force algorithm



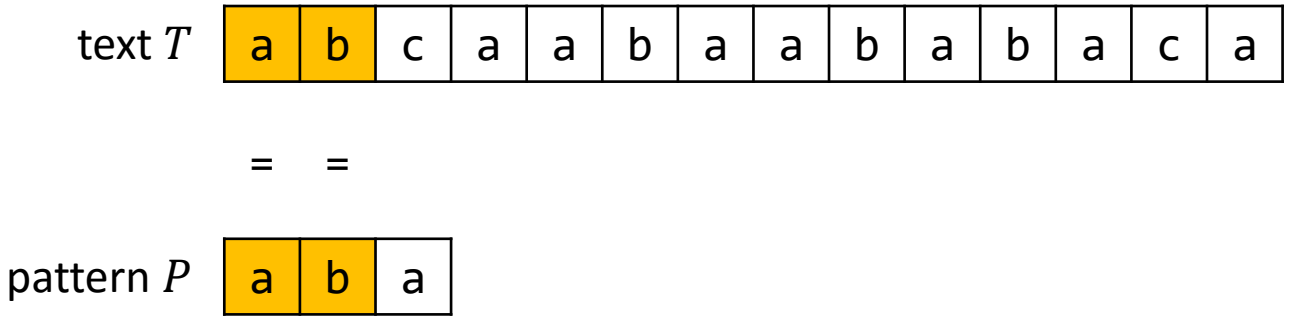
Devising a New Strategy

Let’s observe the behaviour of the brute-force algorithm



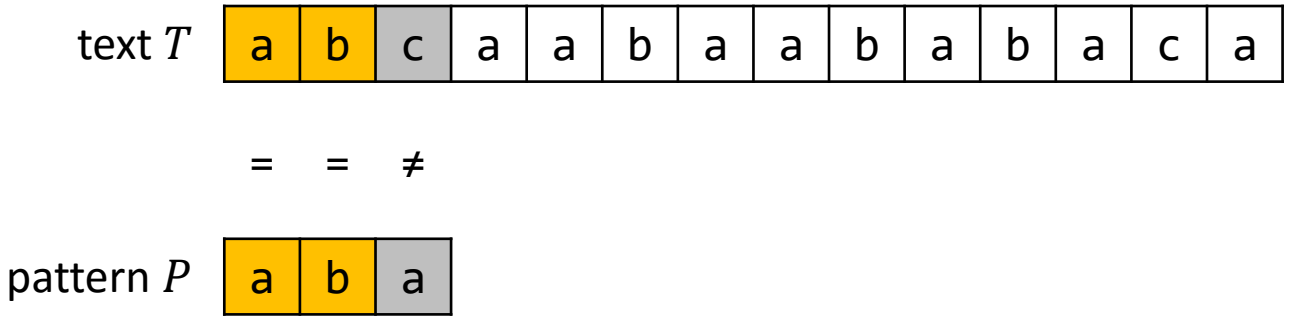
Devising a New Strategy

Let’s observe the behaviour of the brute-force algorithm



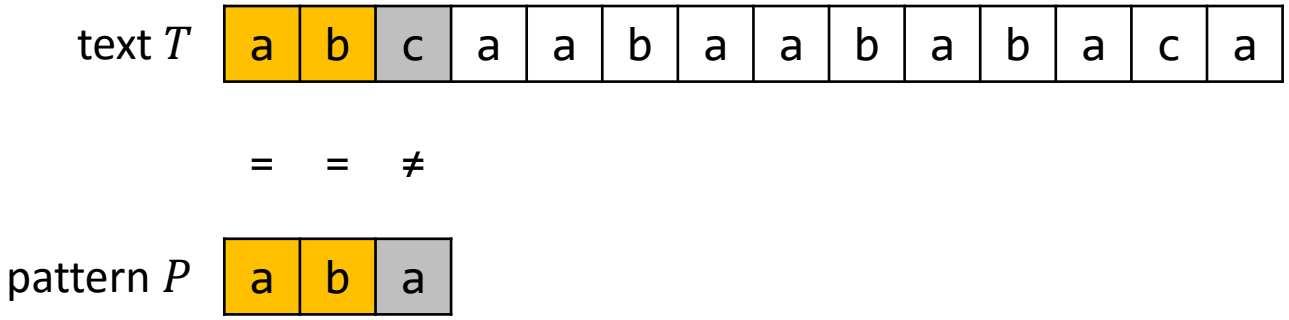
Devising a New Strategy

Let’s observe the behaviour of the brute-force algorithm



Devising a New Strategy

Let's observe the behaviour of the brute-force algorithm



What now?

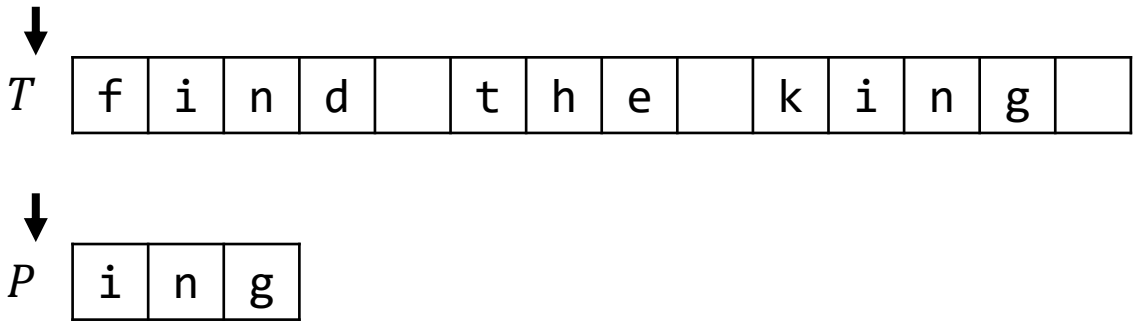
- brute-force algorithms says: go back to the second position in T and start again from the beginning of P
- but why not simply continue moving through T?

Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```

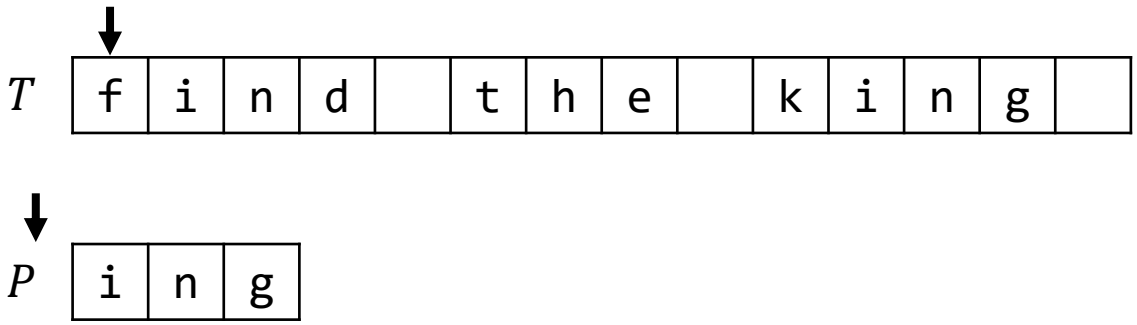
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



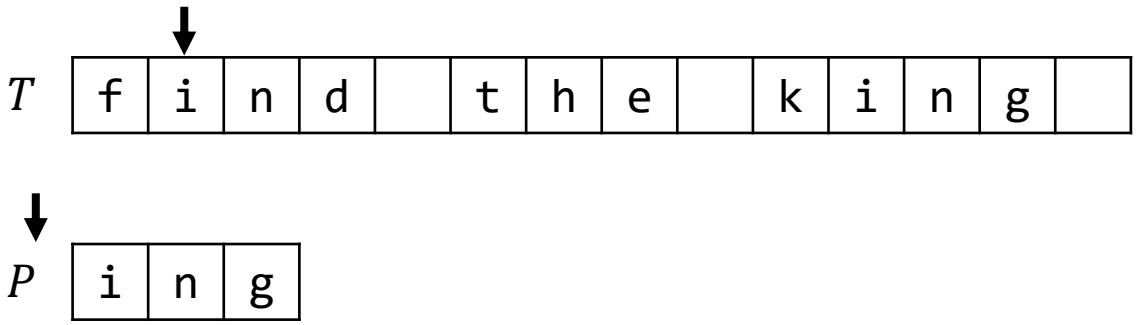
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



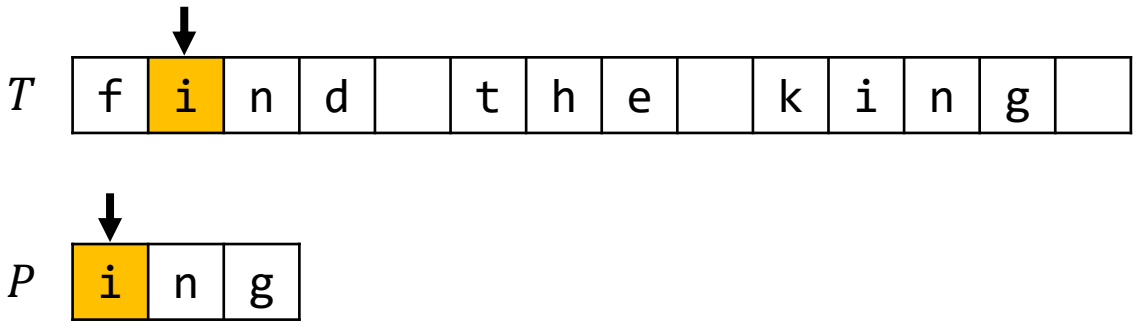
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



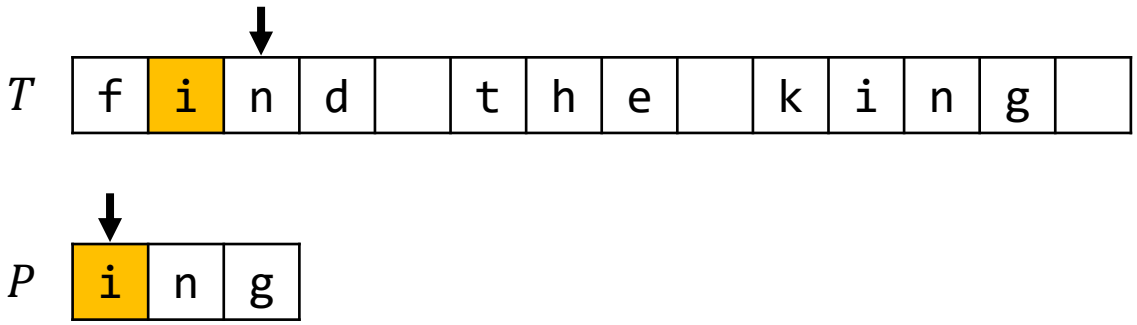
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



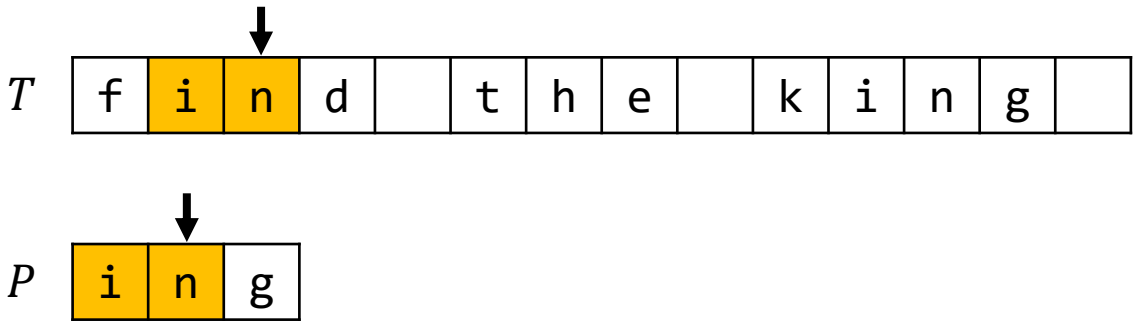
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



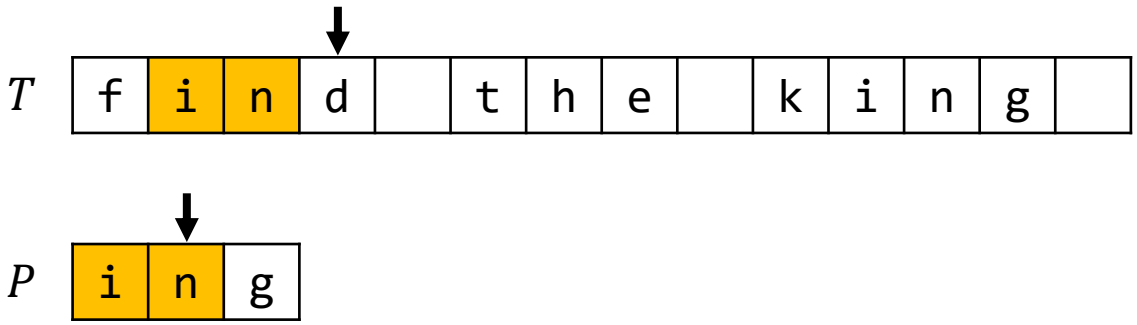
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



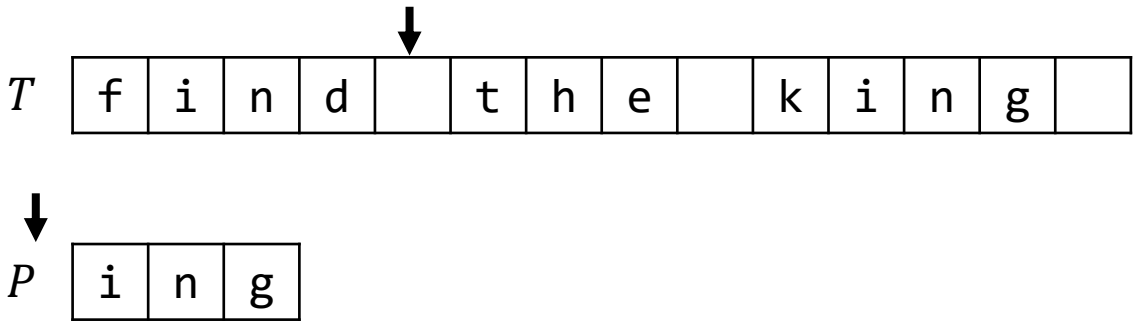
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



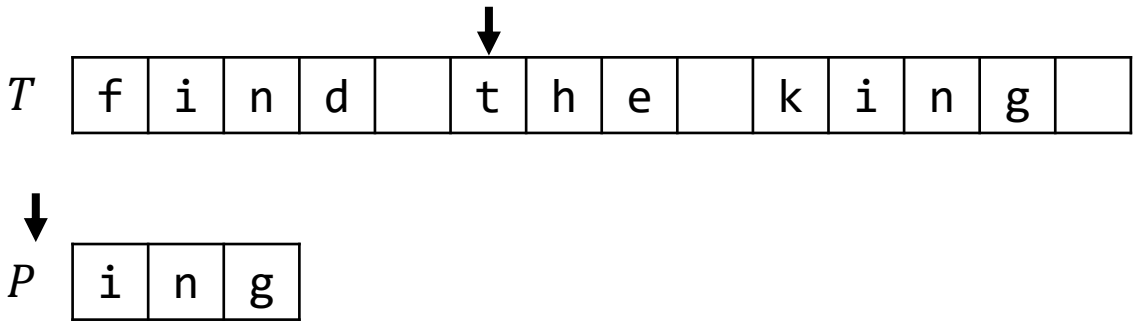
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



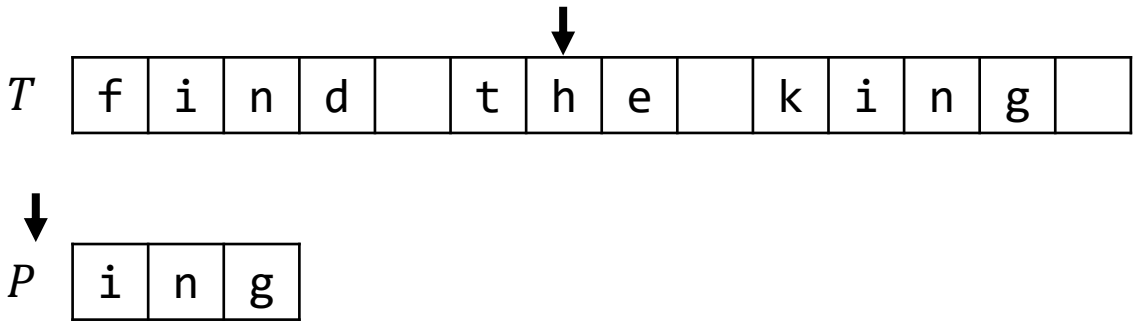
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



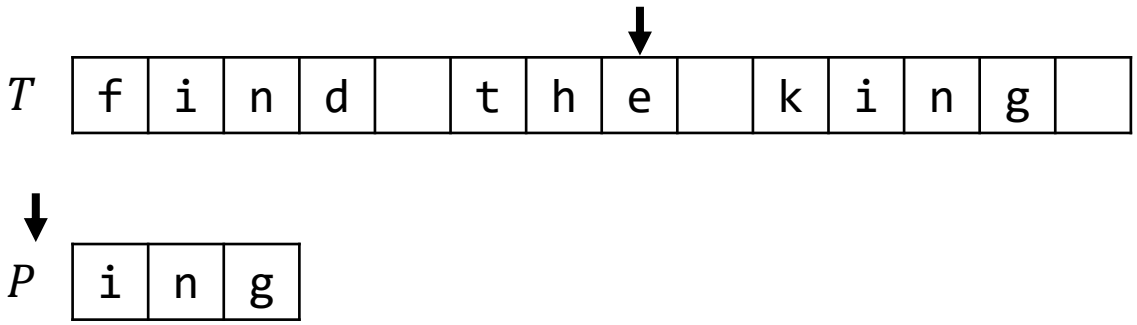
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



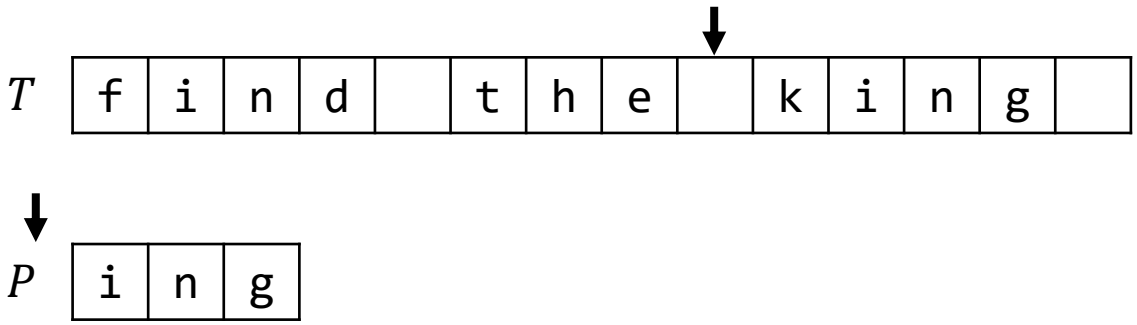
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



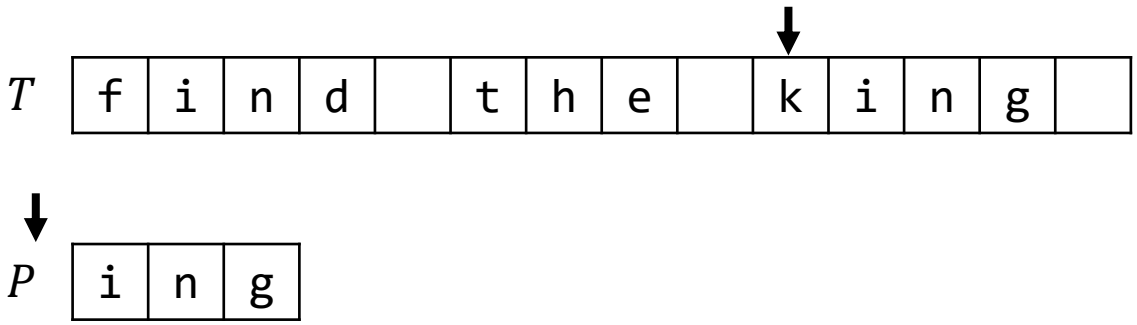
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



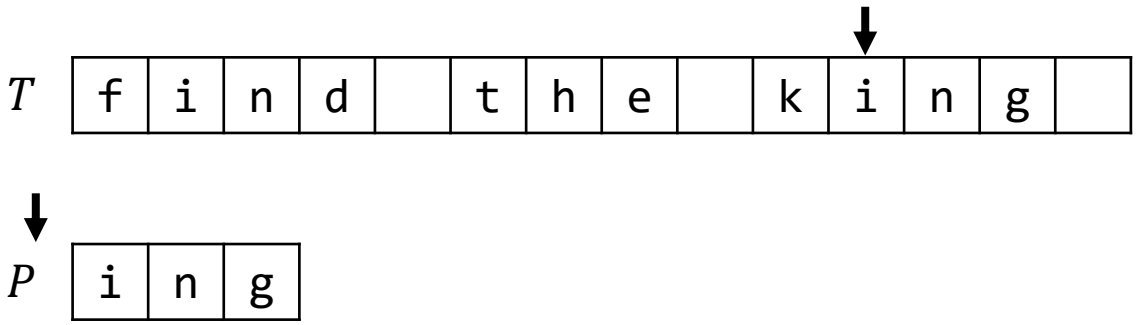
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



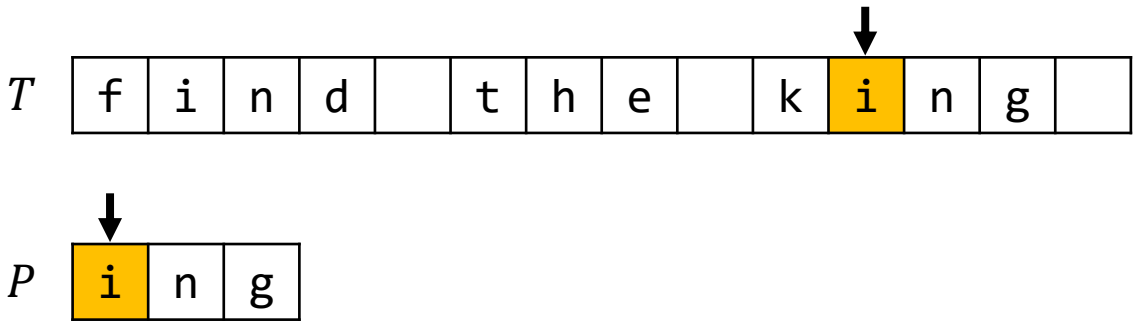
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



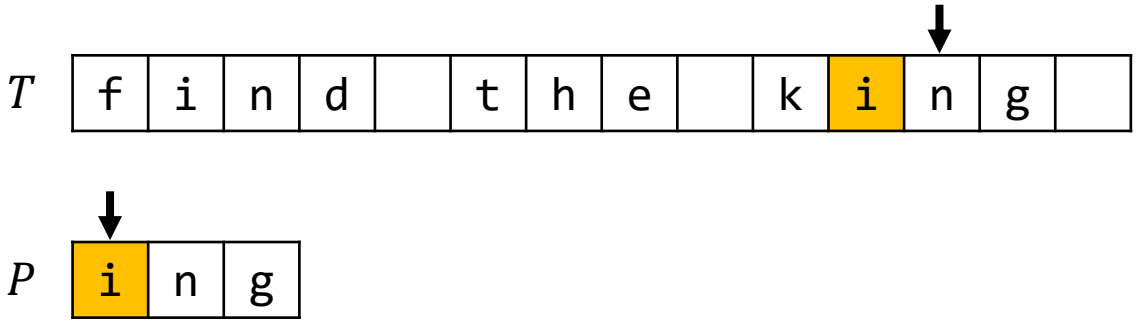
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



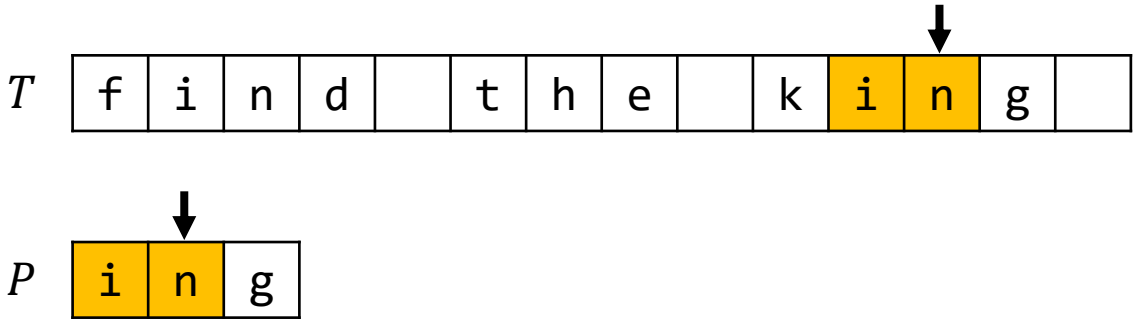
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



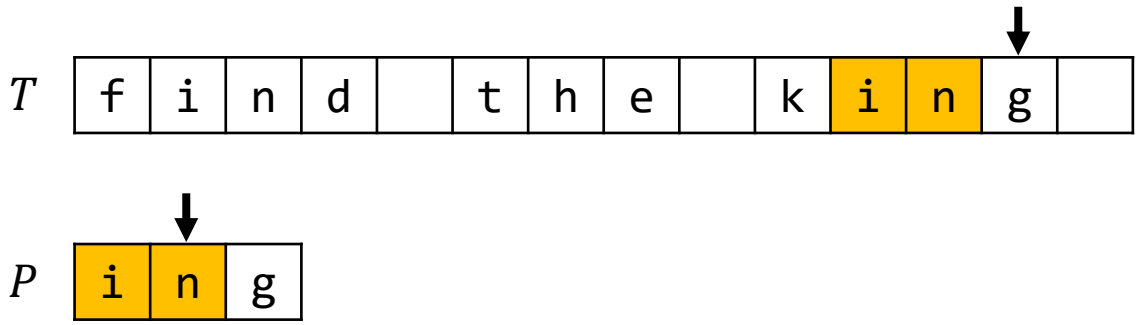
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



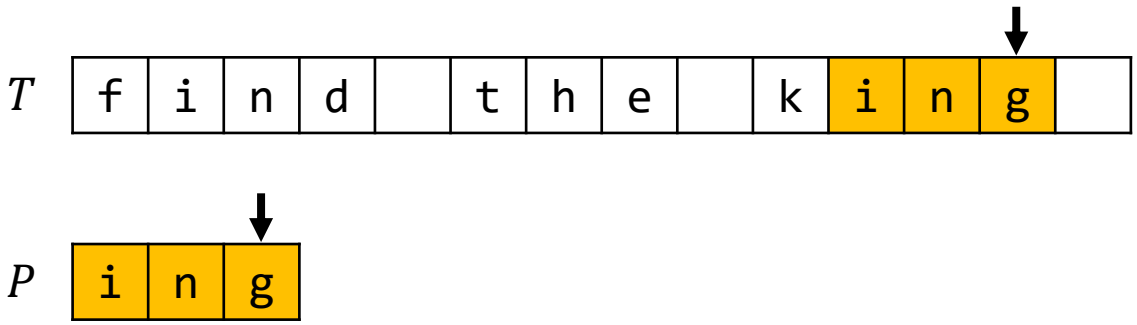
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



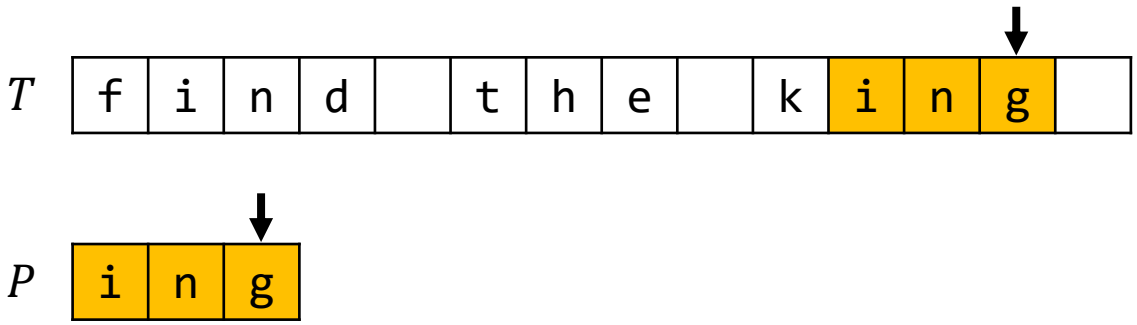
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



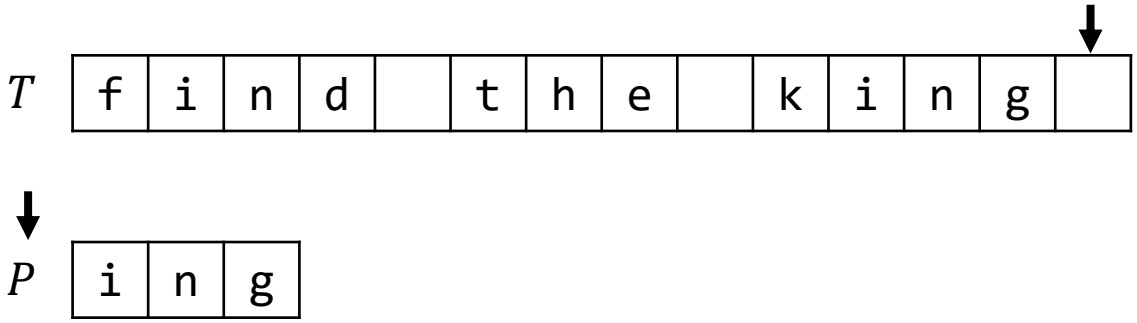
Example: Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



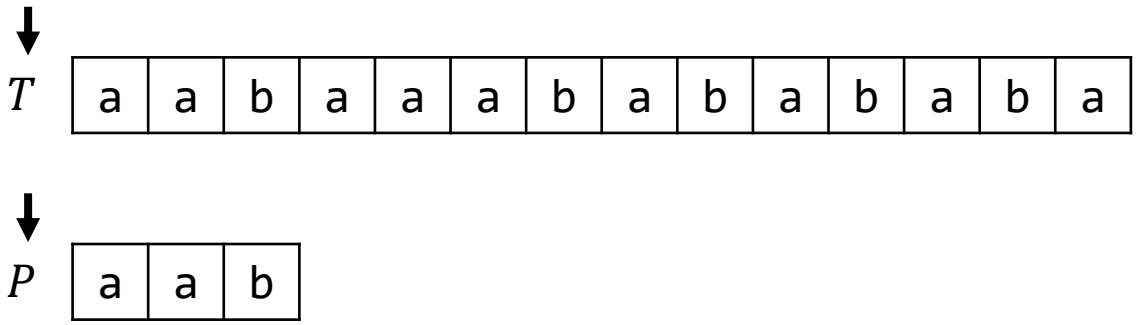
What is the running time?

$O(n)$

Problem: it's broken!

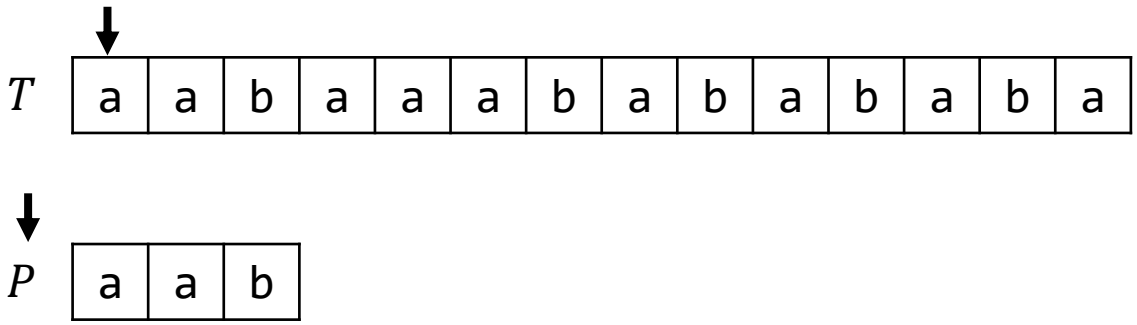
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



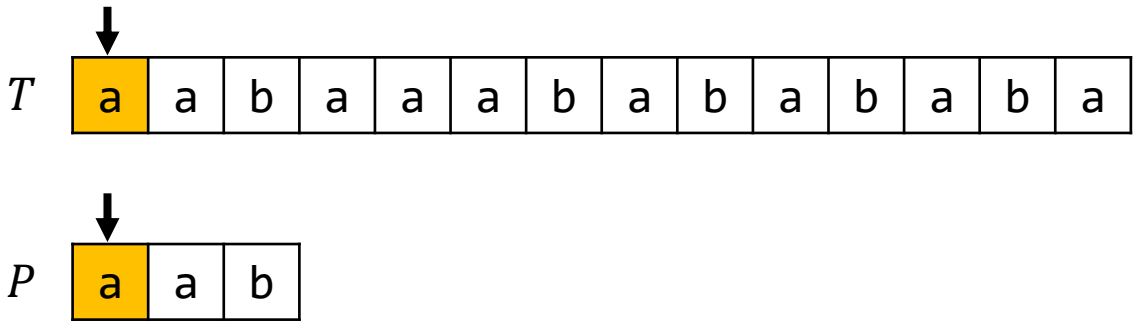
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



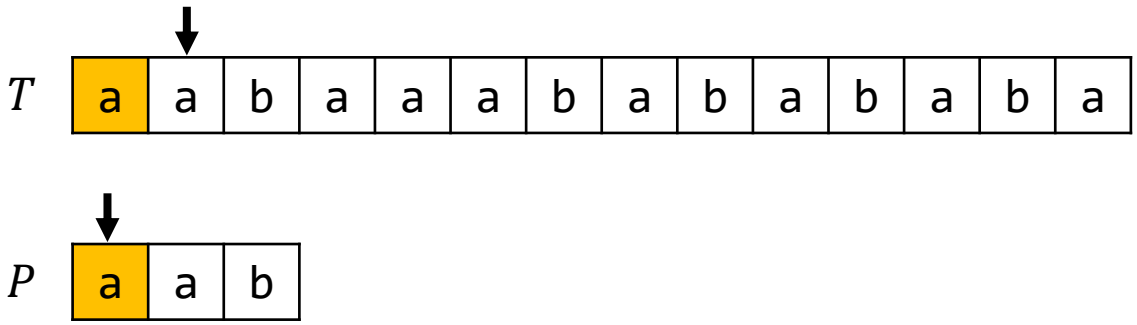
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



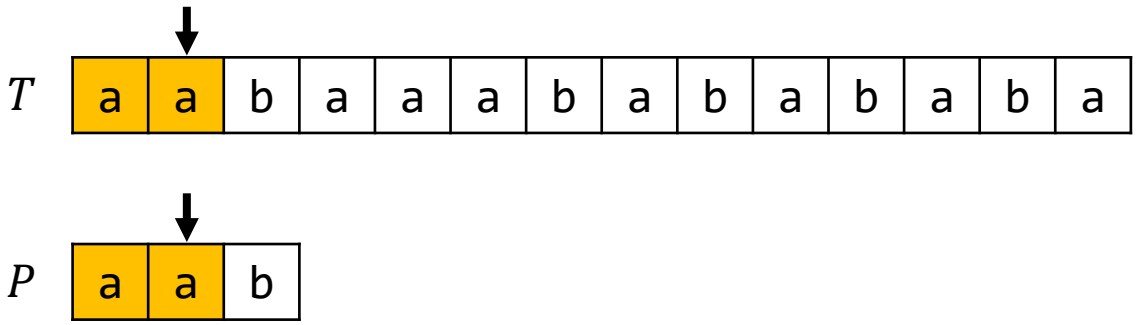
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



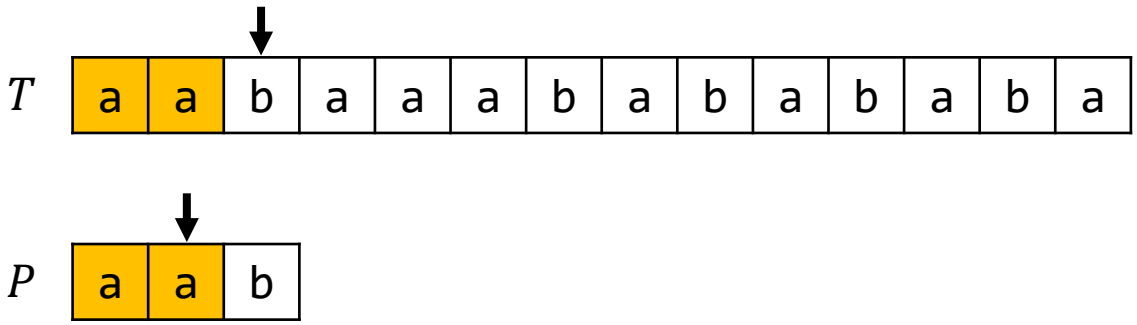
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



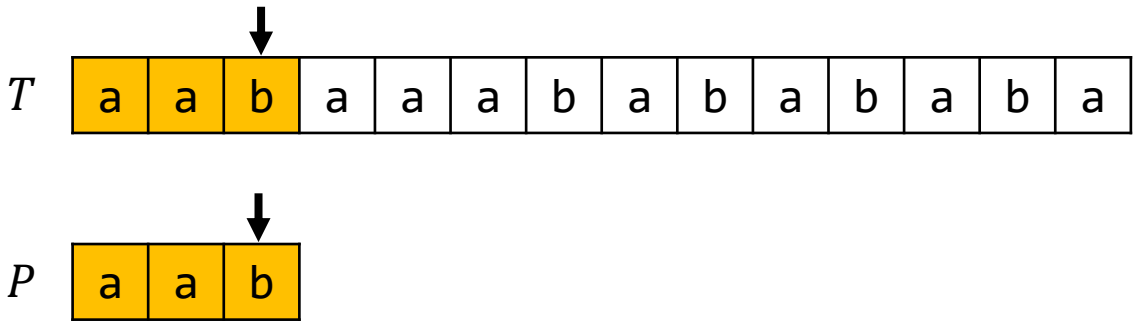
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



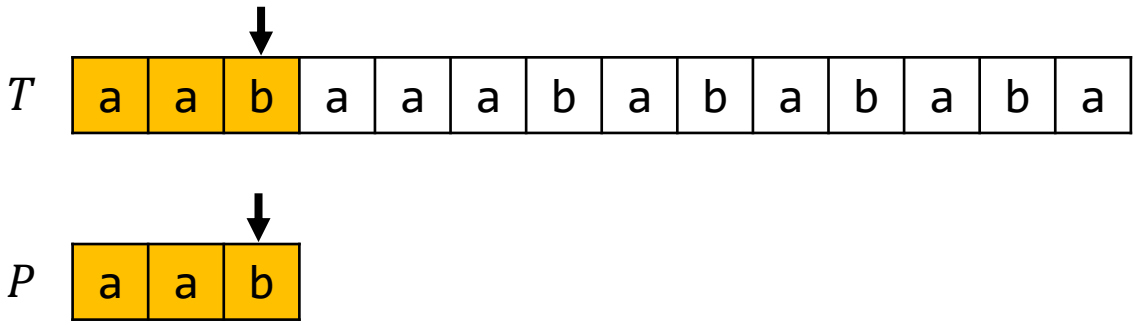
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



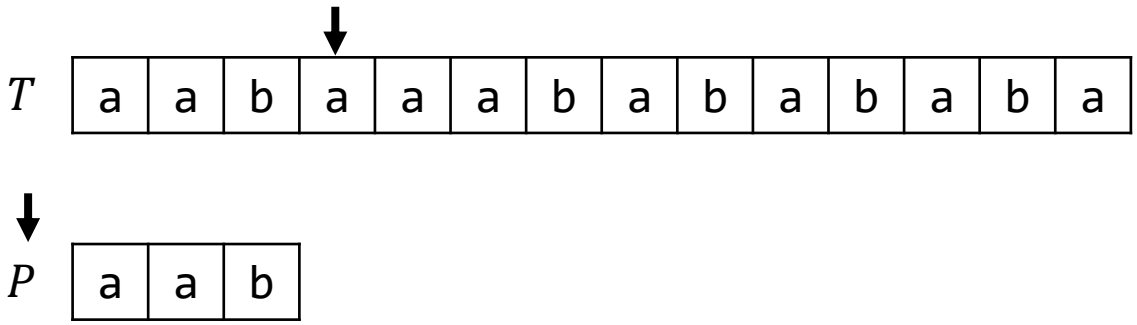
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



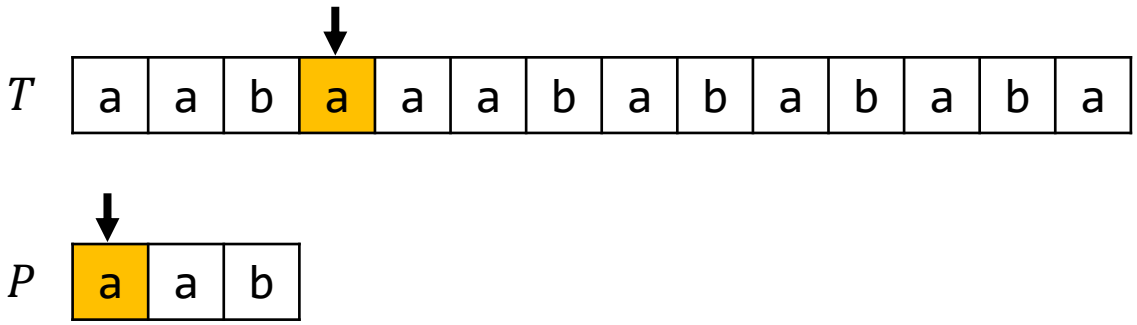
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



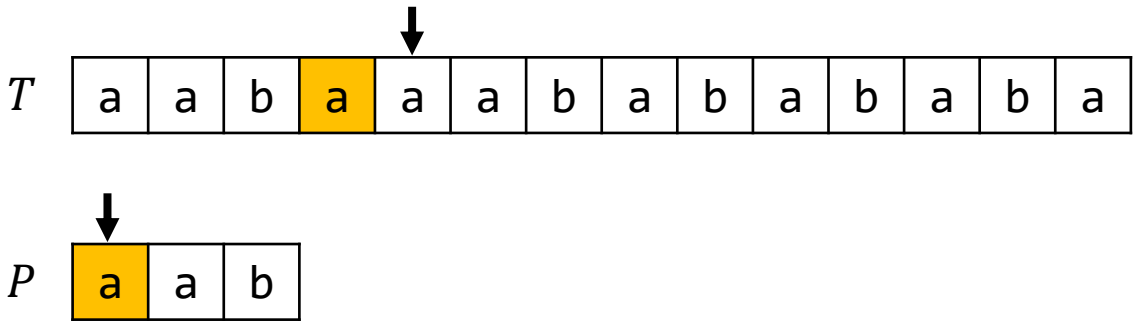
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



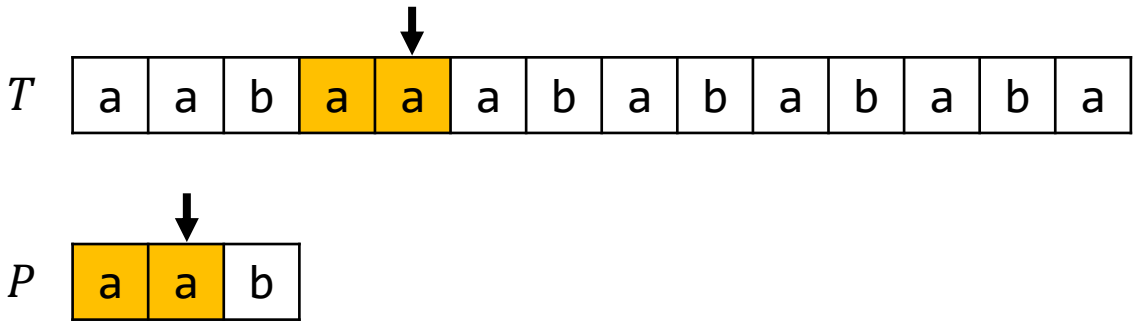
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



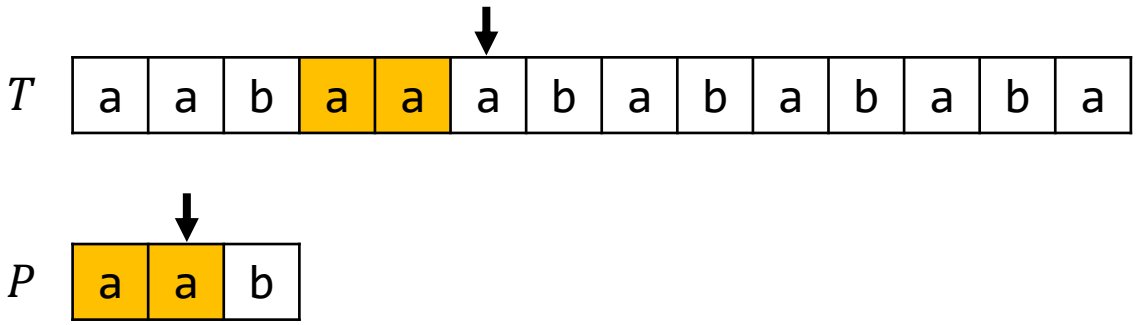
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



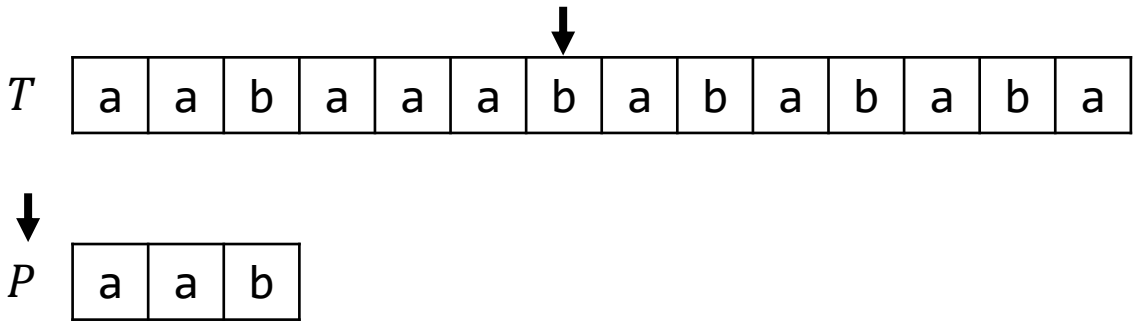
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



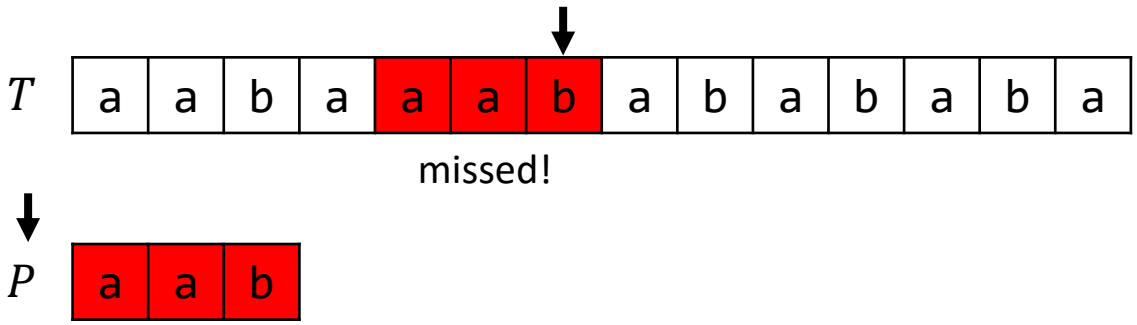
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



Faster Algorithm

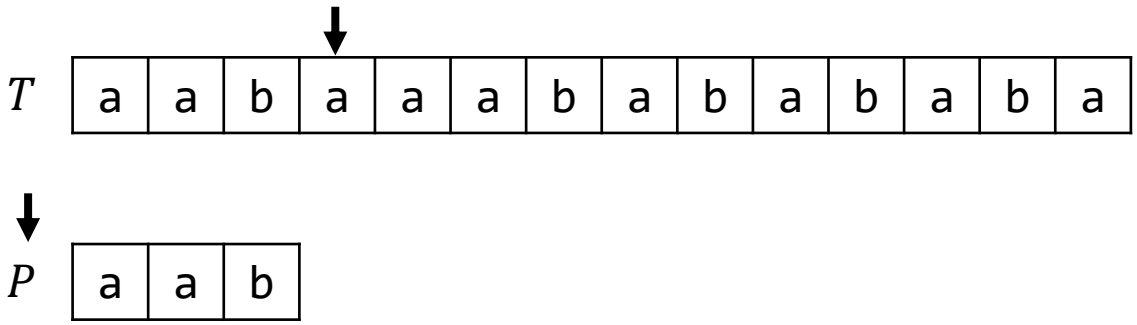
```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



Where does it go wrong?

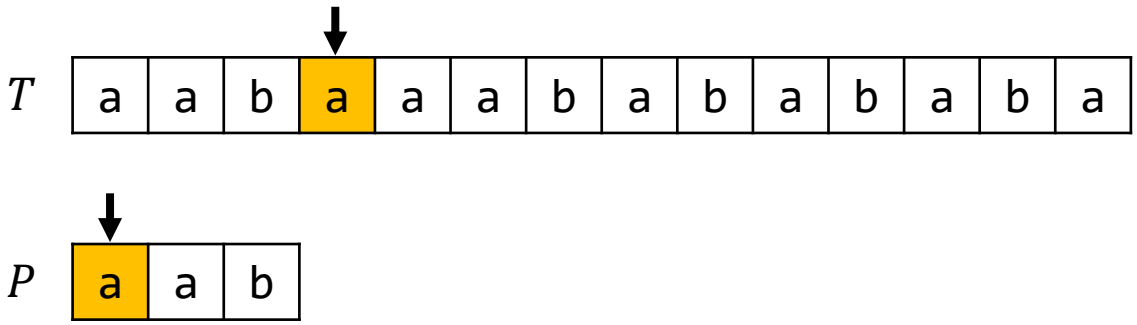
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



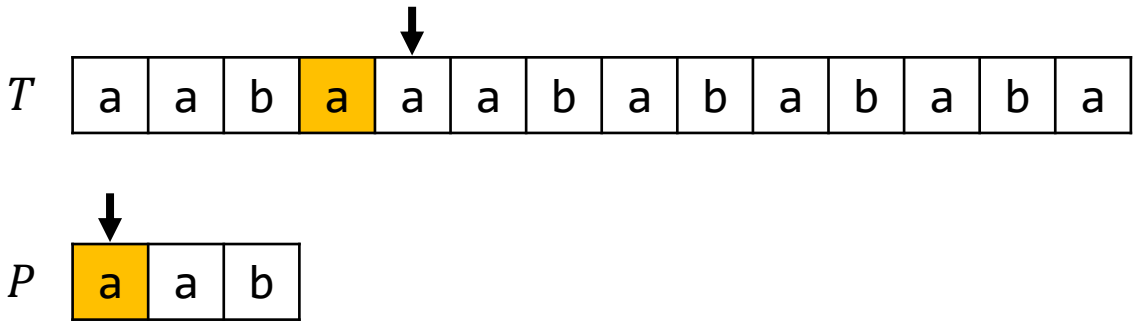
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



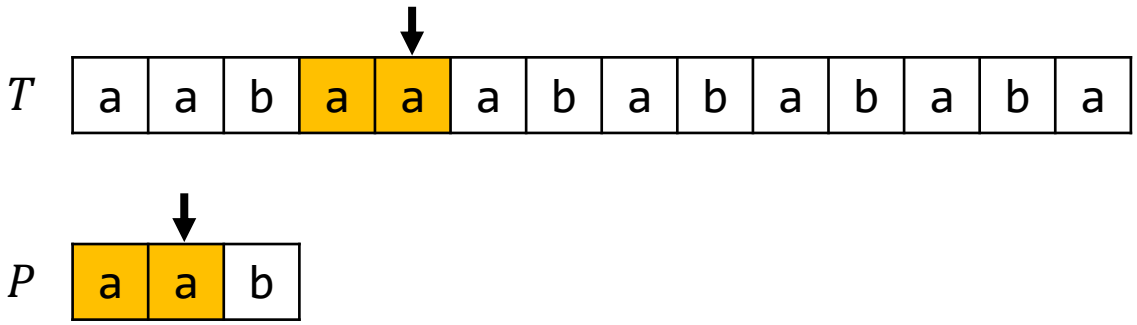
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



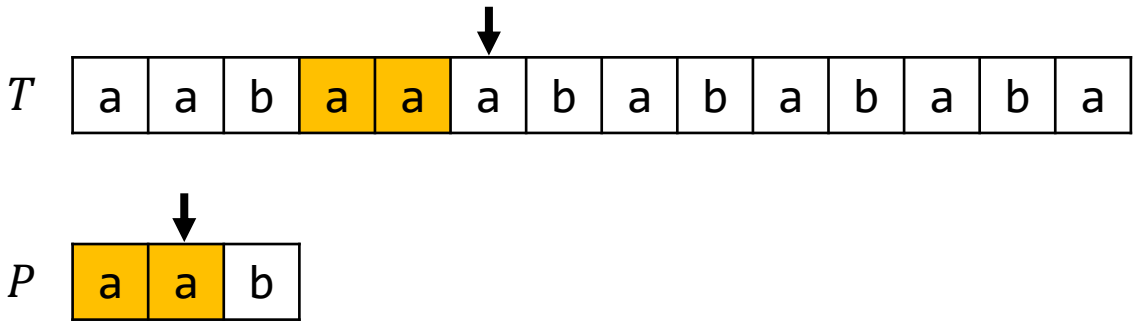
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



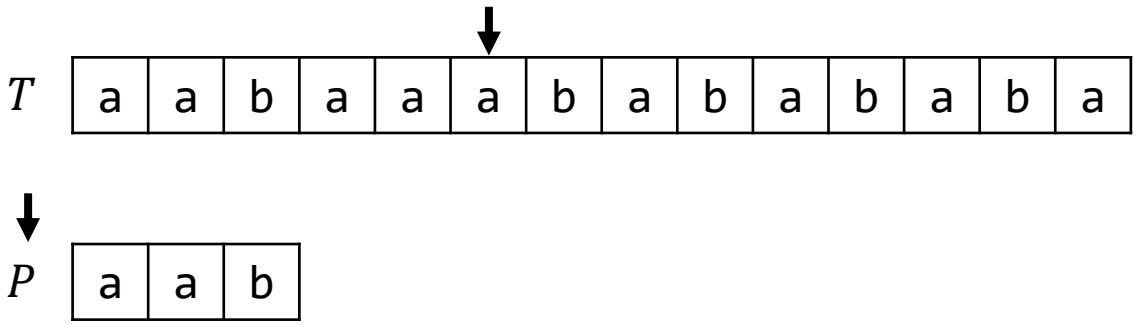
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



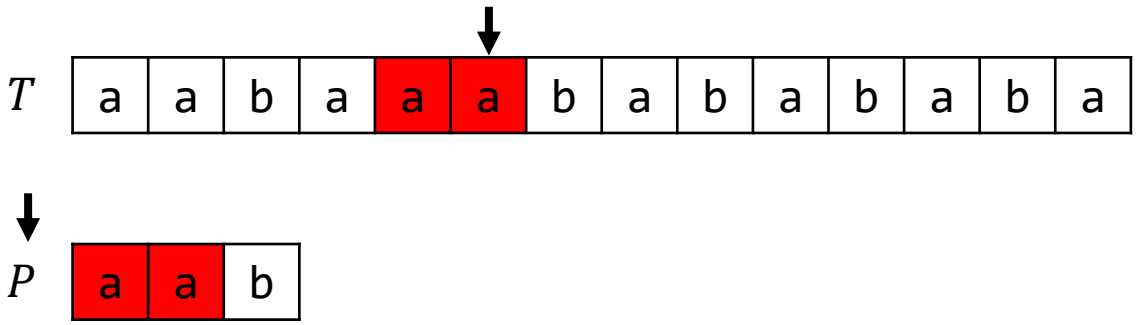
Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```



Faster Algorithm

```
FASTER-NAIVE-STRING-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: q = 0
 4: s = 0
 5: while s < n
 6:     s = s+1
 7:     if T[s] == P[q+1]
 8:         q = q+1
 9:         if q == m
10:             PRINT(s-m)
11:             q = 0
12:     else q = 0
```

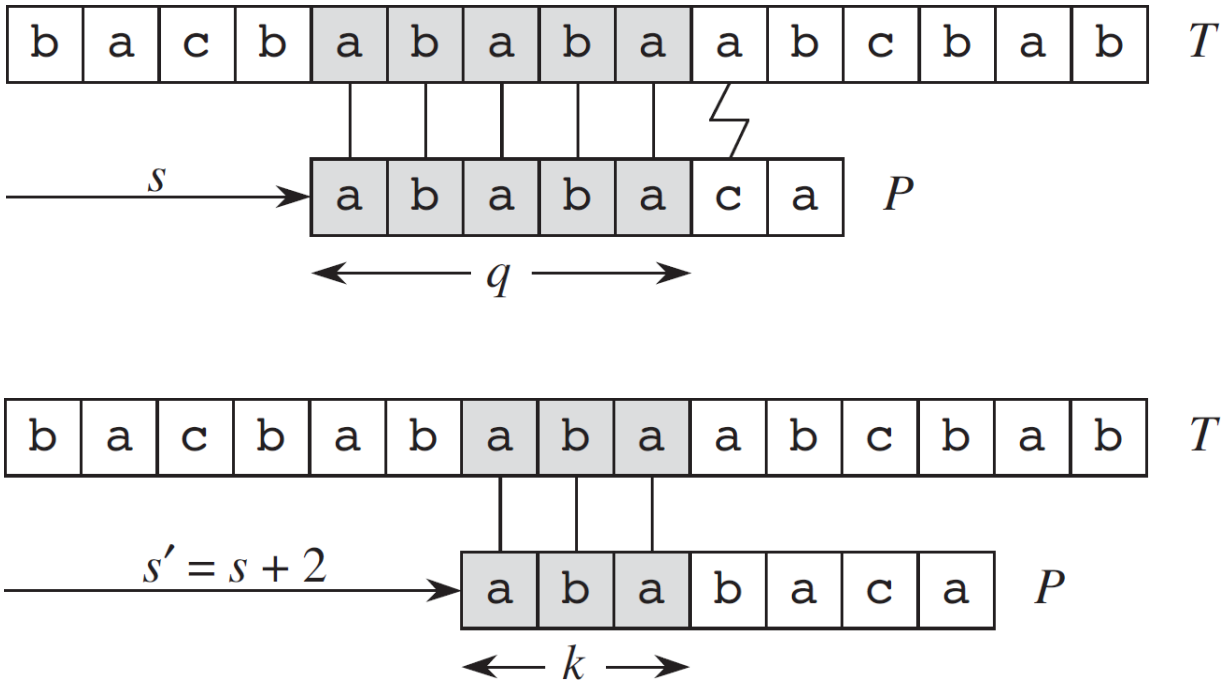


By returning to the beginning, we miss a legitimate **prefix** of P

Prefix Function: Intuition

The prefix function π encapsulates knowledge about how a pattern matches against shifts of itself

This information can be used to avoid testing useless shifts

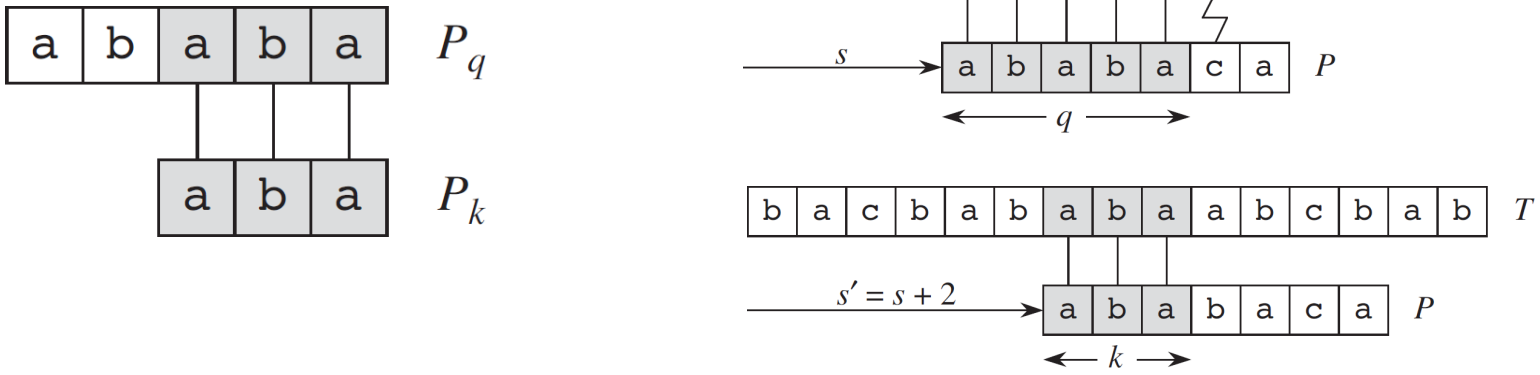


[Cormen] p.1004

Prefix Function: Intuition

The prefix function π encapsulates knowledge about how a pattern matches against shifts of itself

This information can be used to avoid testing useless shifts, and depends only on the pattern P, thus, can be **precomputed**



The **longest prefix** of P that is also a **suffix** of P_5 is P_3. This can be stored in an array $\pi[5]=3$.

Knowing that q-characters have been matched at shift s, the **next potentially valid shift** is $s' = s + (q - \pi[q])$.

[Cormen] p.1004

Prefix Function: Formalisation

Given a pattern $P[1..m]$, the prefix function for P is the function $\pi : \{1,2,\ldots,m\} \to \{0,1,\ldots,m-1\}$ such that

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

So, $\pi[q]$ is the length of the longest prefix of P that is a suffix of P_q.

Prefix Function - Example 1

Complete the prefix function table

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$							

Prefix Function - Example 1

Complete the prefix function table

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

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

Prefix Function - Example 1

Complete the prefix function table

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

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

Prefix Function - Example 1

Complete the prefix function table

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

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

Prefix Function - Example 1

Complete the prefix function table

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

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

Prefix Function - Example 1

Complete the prefix function table

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

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

Prefix Function - Example 1

Complete the prefix function table

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

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

Prefix Function - Example 1

Complete the prefix function table

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

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

Prefix Function - Example 2

Complete the prefix function table

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

i	1	2	3	4	5	6	7
$P[i]$	a	a	a	a	a	a	a
$\pi[i]$							

Prefix Function - Example 2

Complete the prefix function table

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

i	1	2	3	4	5	6	7
$P[i]$	a	a	a	a	a	a	a
$\pi[i]$	0	1	2	3	4	5	6

Prefix Function Algorithm

```
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 Algorithm

```
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
```

$P[i]$	a	b	a	b	a	c	a
$\pi[i]$							

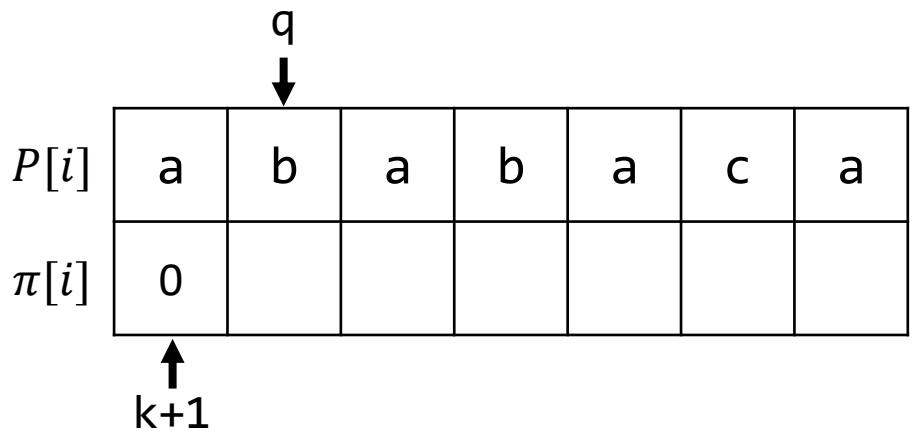
Prefix Function Algorithm

```
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
```

$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0						

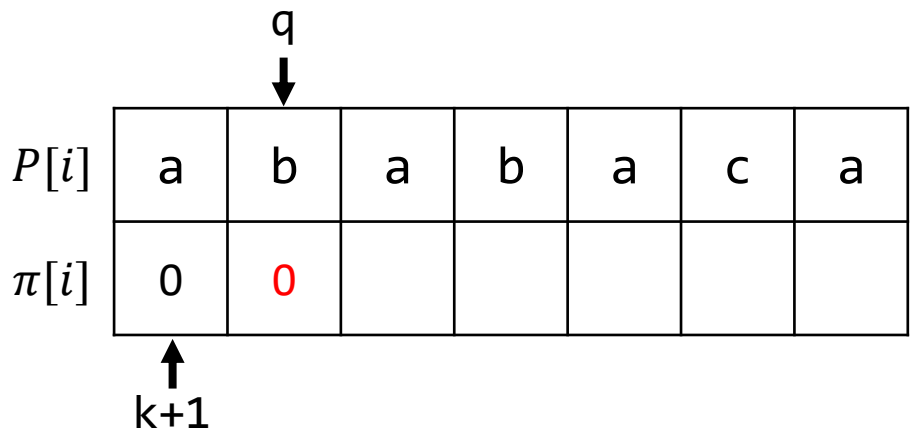
Prefix Function Algorithm

```
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
```



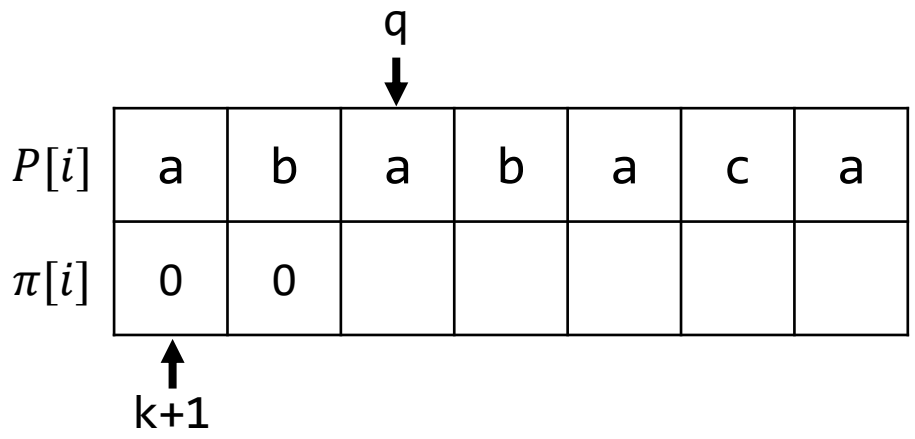
Prefix Function Algorithm

```
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
```



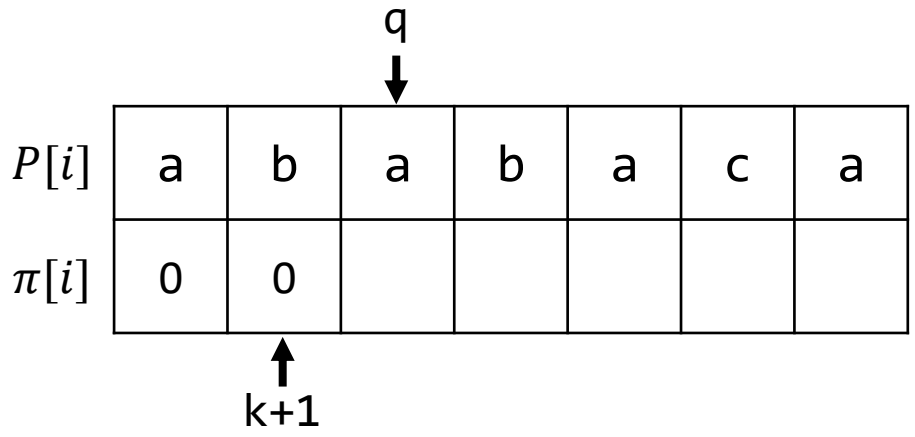
Prefix Function Algorithm

```
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
```



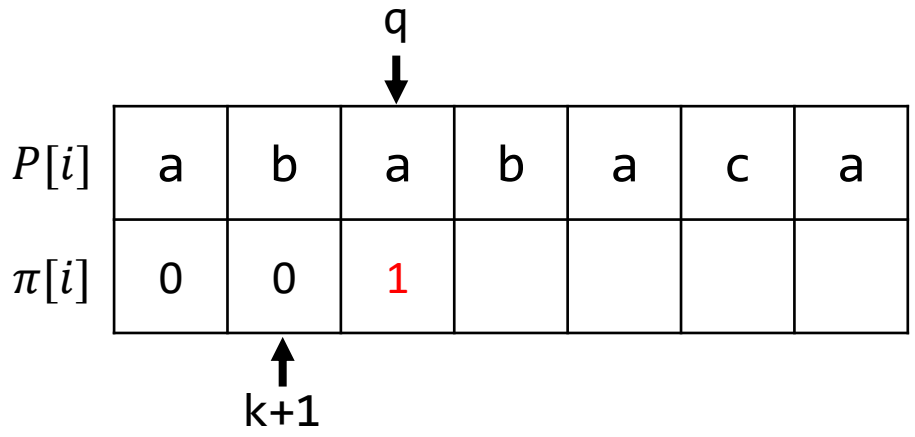
Prefix Function Algorithm

```
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
```



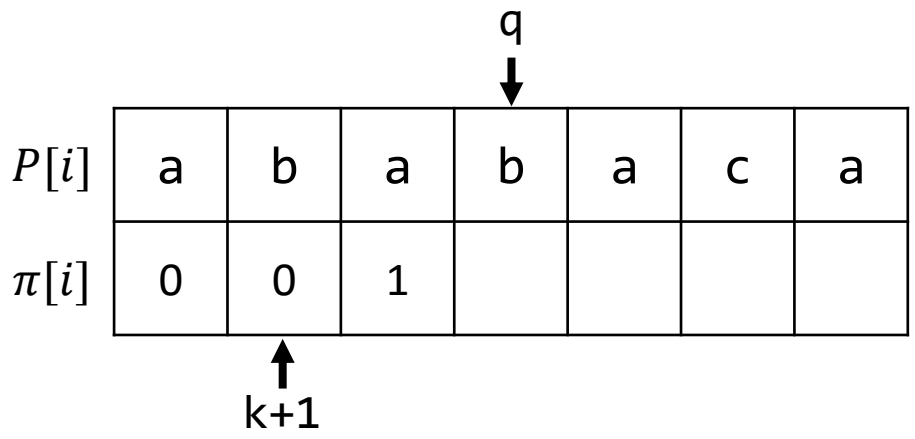
Prefix Function Algorithm

```
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
```



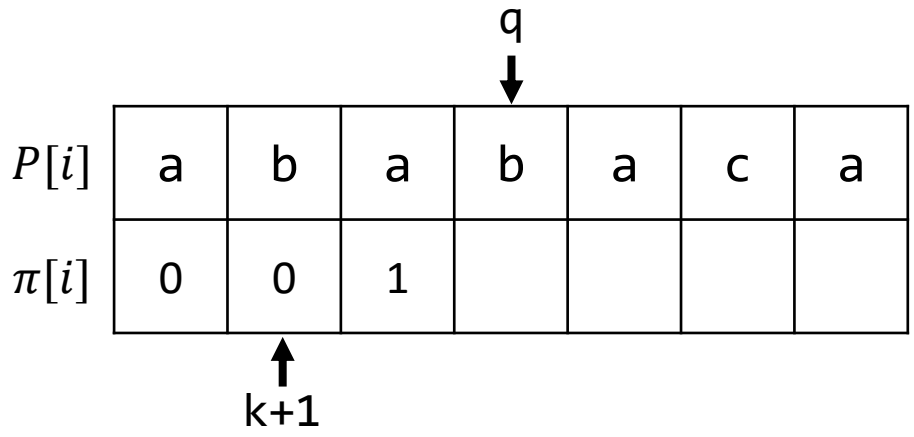
Prefix Function Algorithm

```
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
```



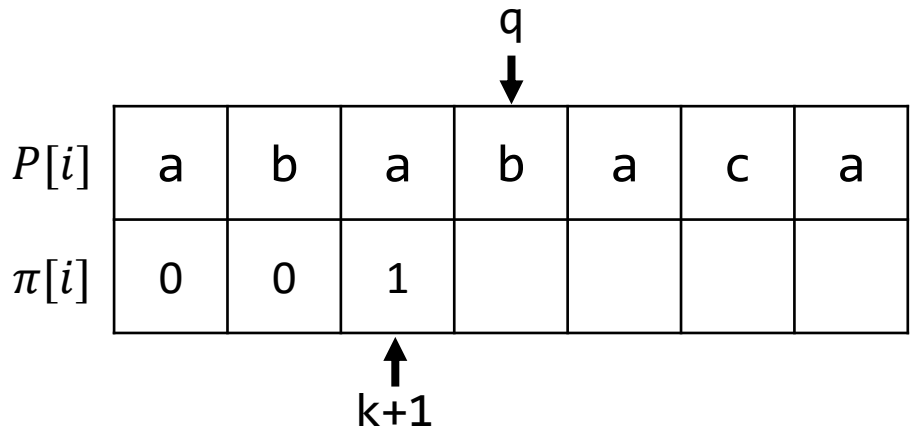
Prefix Function Algorithm

```
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
```



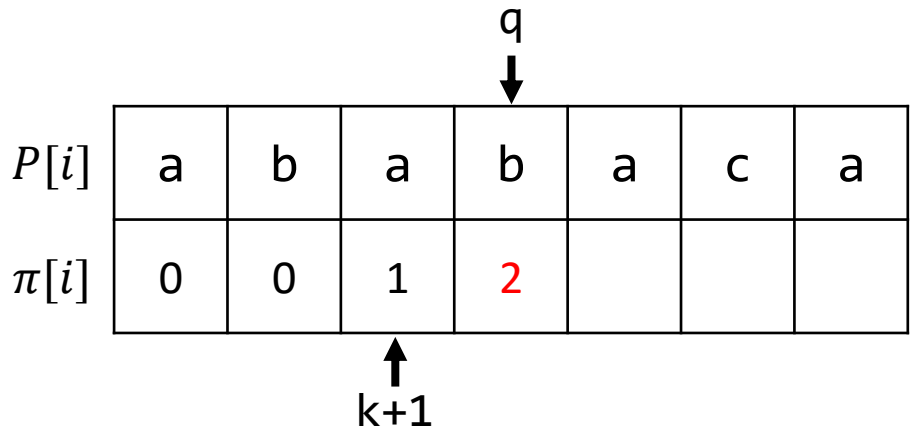
Prefix Function Algorithm

```
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
```



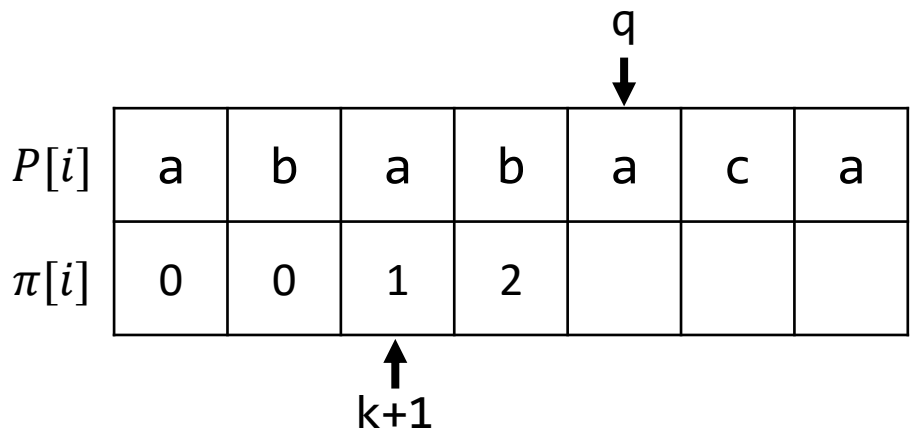
Prefix Function Algorithm

```
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
```



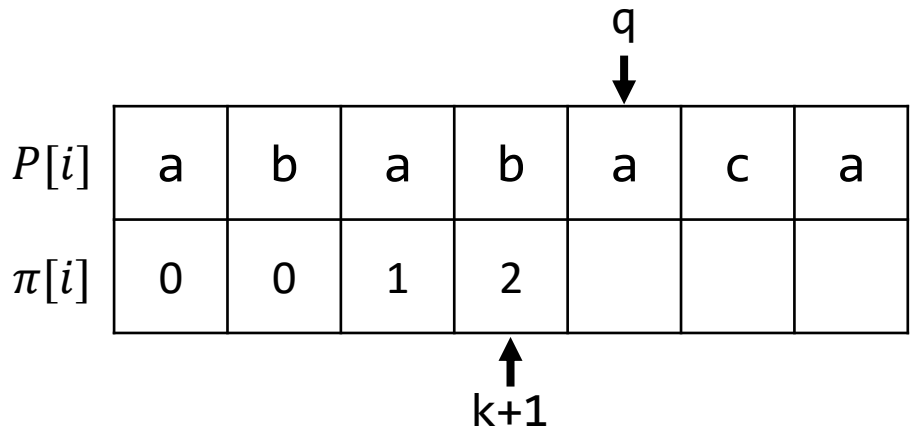
Prefix Function Algorithm

```
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
```



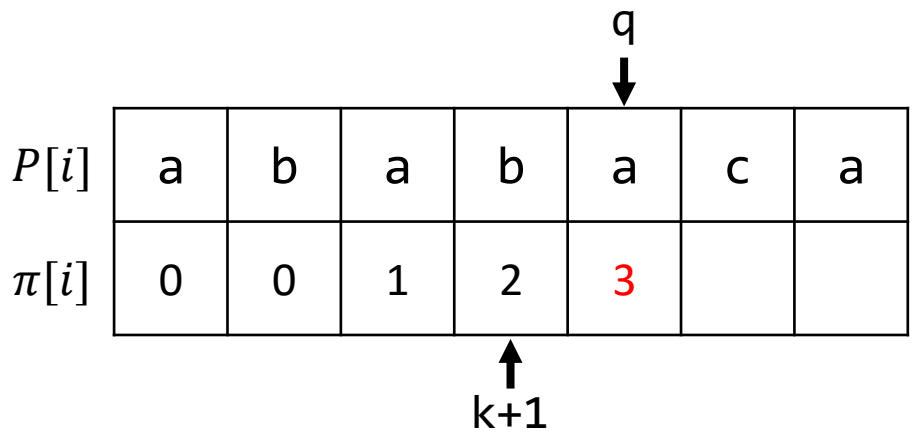
Prefix Function Algorithm

```
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
```



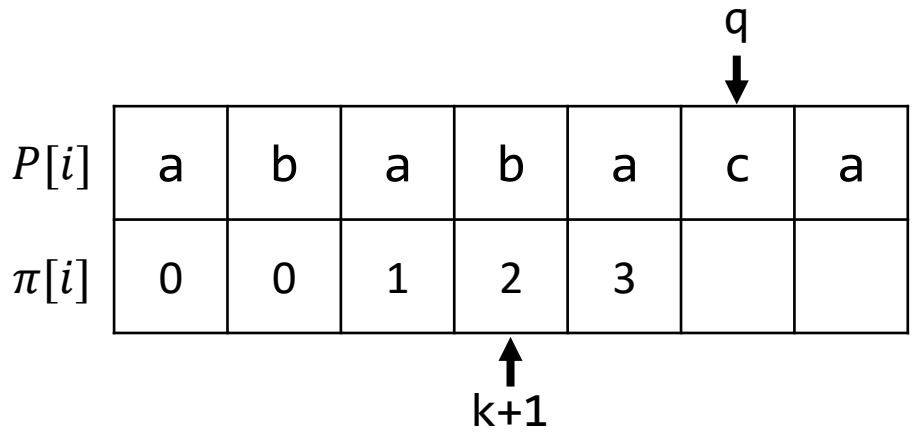
Prefix Function Algorithm

```
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
```



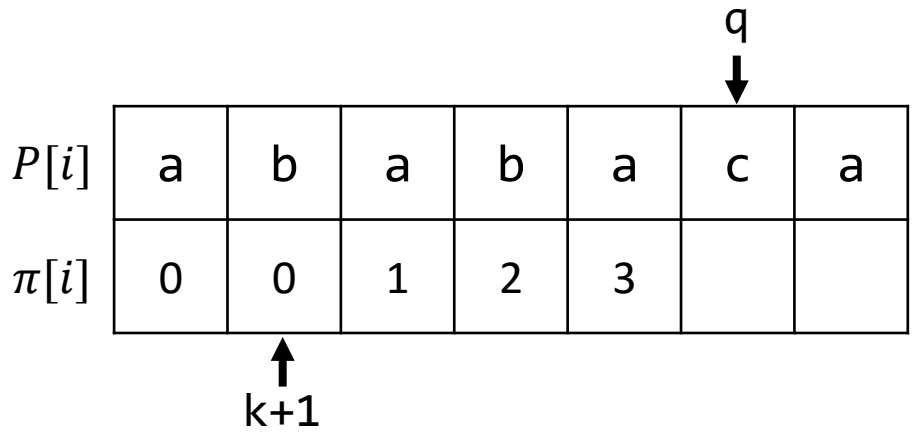
Prefix Function Algorithm

```
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
```



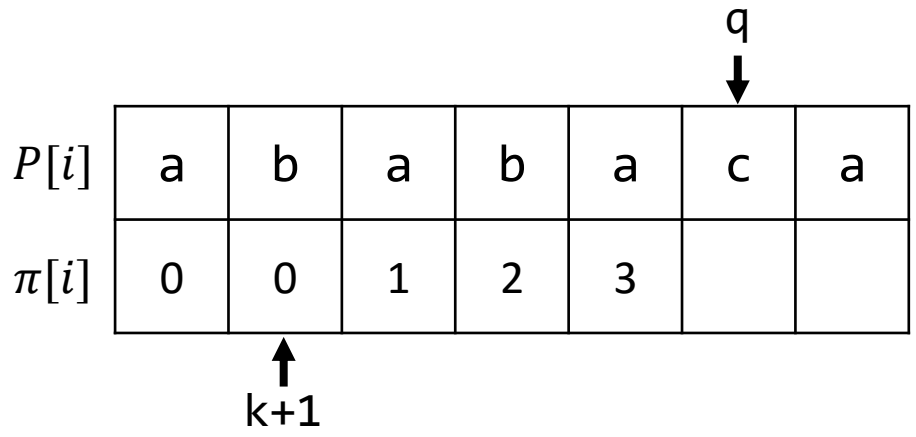
Prefix Function Algorithm

```
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
```



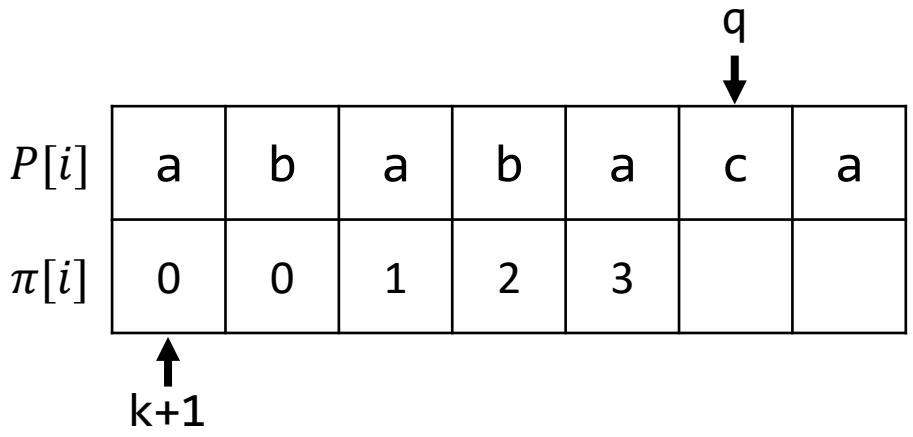
Prefix Function Algorithm

```
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
```



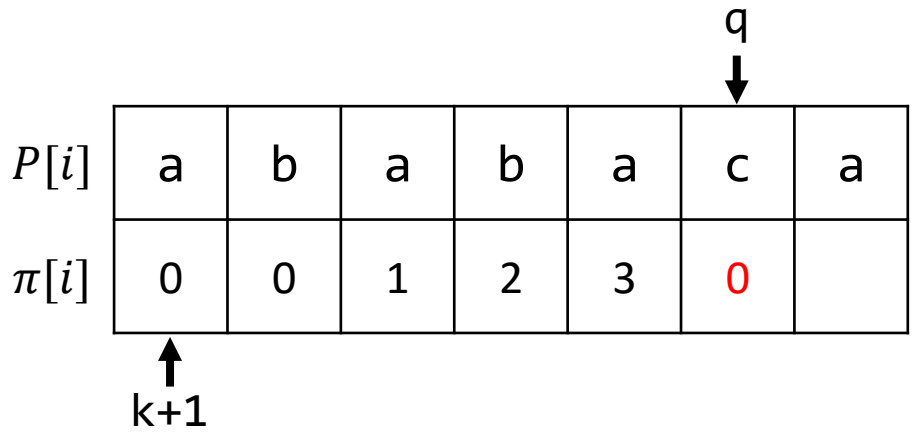
Prefix Function Algorithm

```
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
```



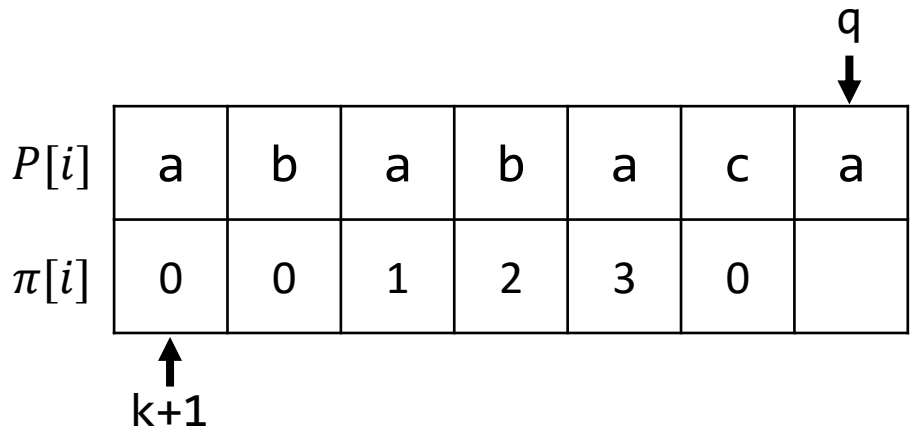
Prefix Function Algorithm

```
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
```



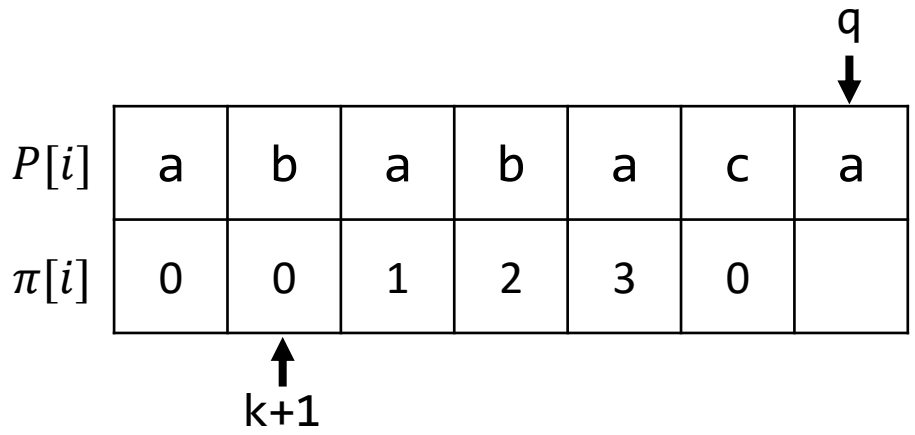
Prefix Function Algorithm

```
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
```



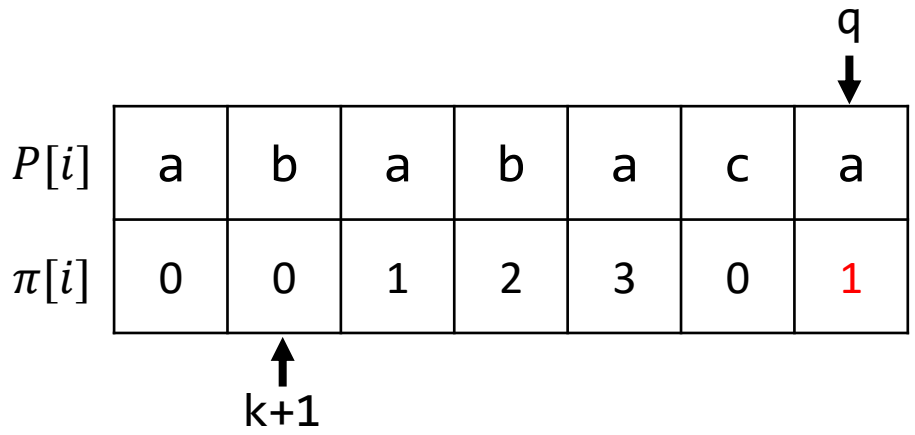
Prefix Function Algorithm

```
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
```



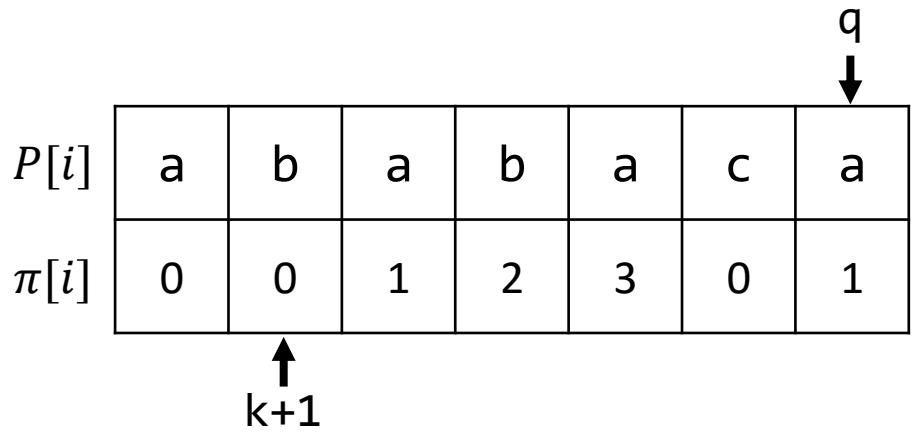
Prefix Function Algorithm

```
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 Algorithm

```
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 Algorithm – Another go

```
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
```

$P[i]$	a	a	b	a	a	a	b
$\pi[i]$							

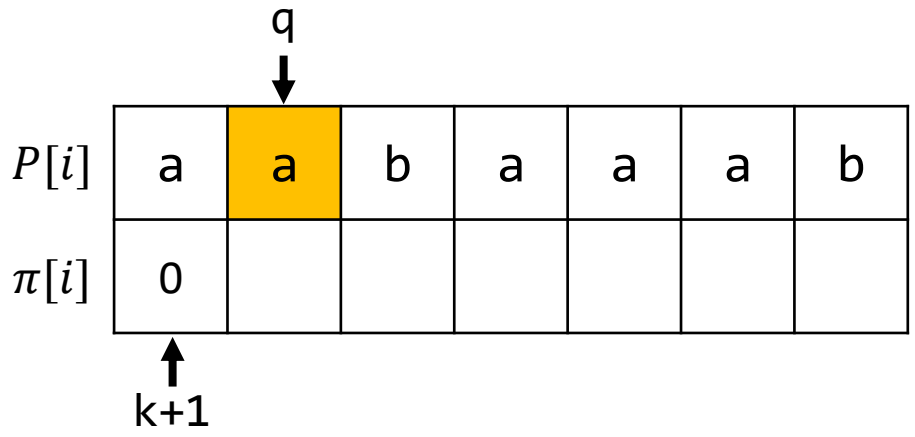
Prefix Function Algorithm – Another go

```
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
```

$P[i]$	a	a	b	a	a	a	b
$\pi[i]$	0						

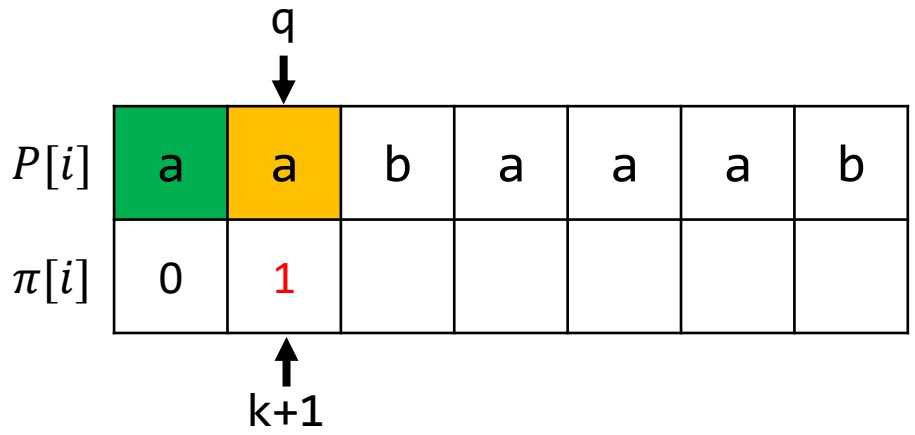
Prefix Function Algorithm – Another go

```
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
```



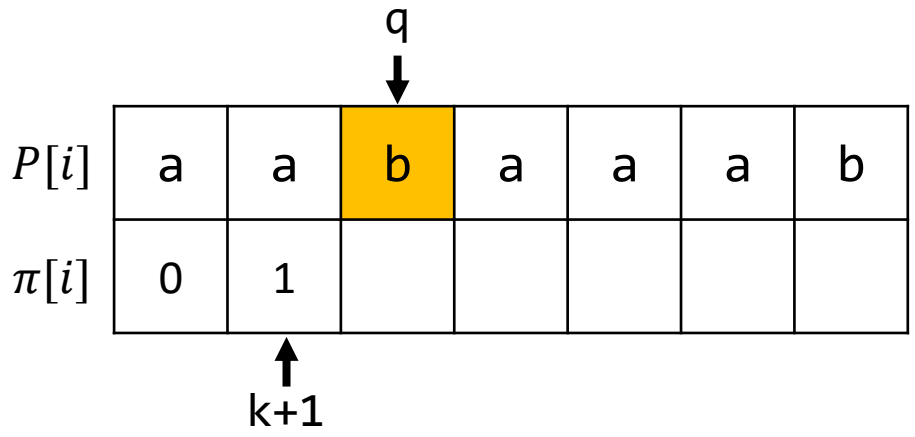
Prefix Function Algorithm – Another go

```
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
```



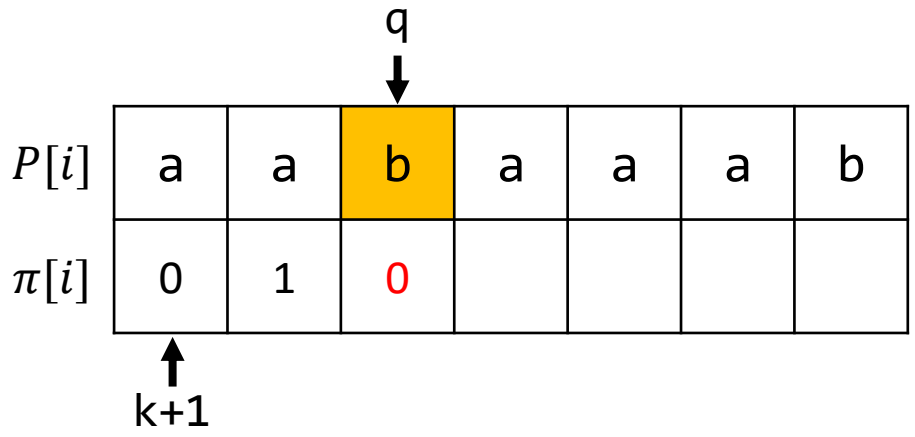
Prefix Function Algorithm – Another go

```
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
```



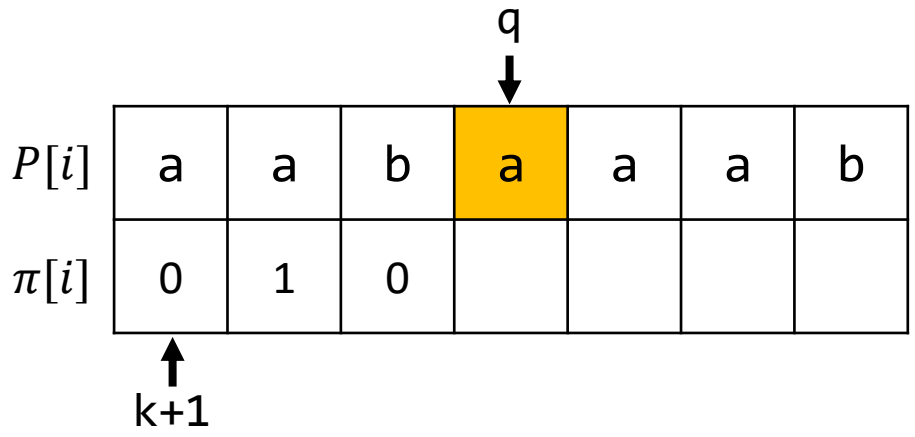
Prefix Function Algorithm – Another go

```
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
```



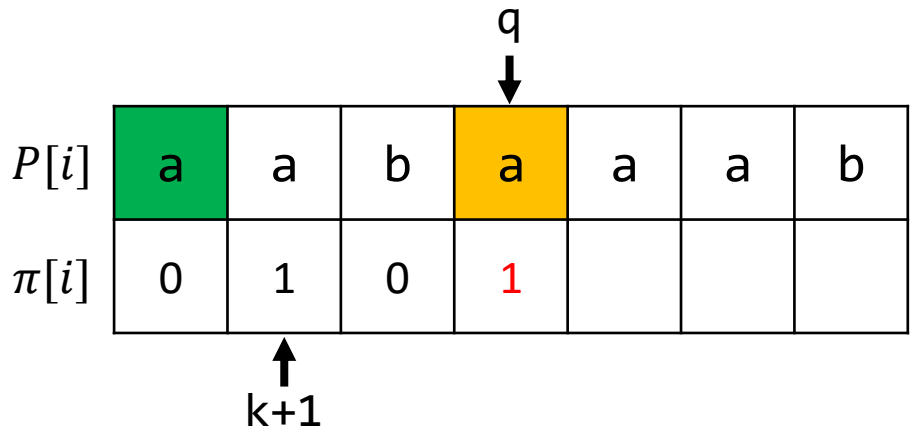
Prefix Function Algorithm – Another go

```
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
```



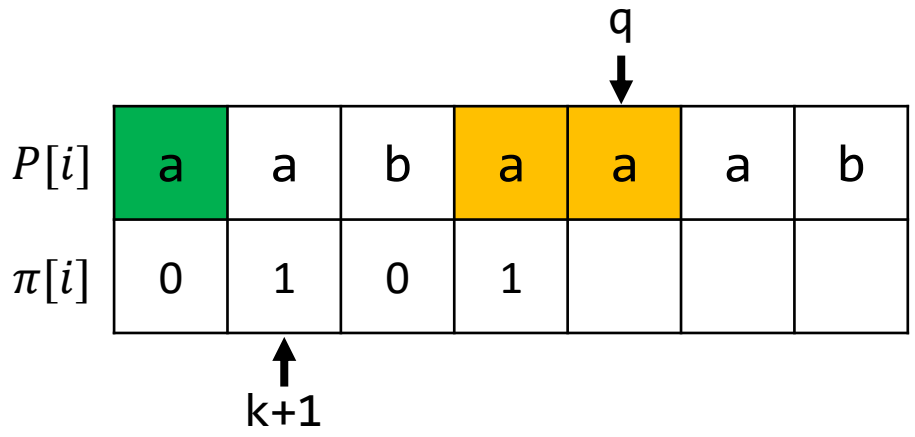
Prefix Function Algorithm – Another go

```
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
```



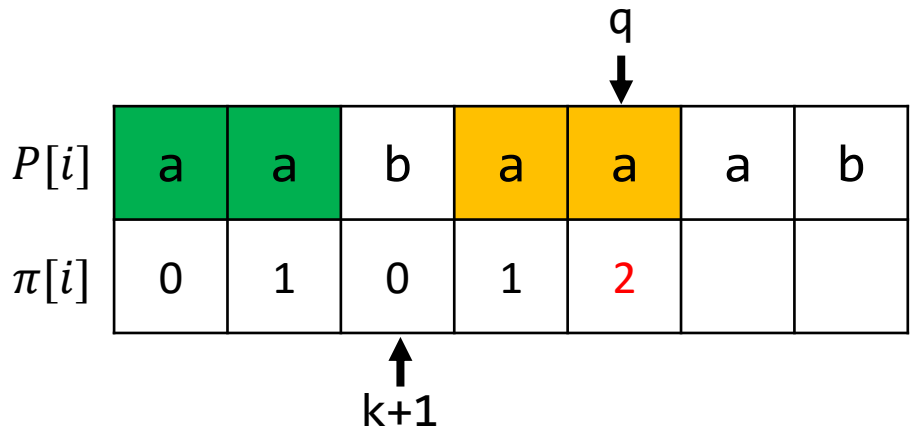
Prefix Function Algorithm – Another go

```
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
```



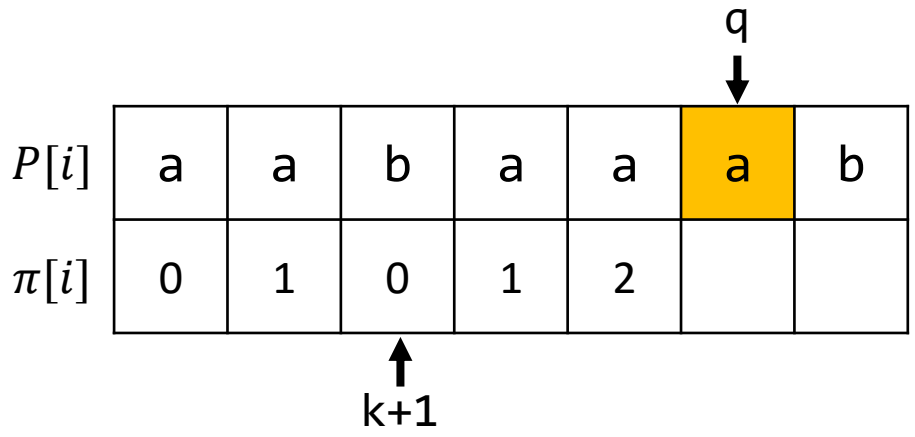
Prefix Function Algorithm – Another go

```
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
```



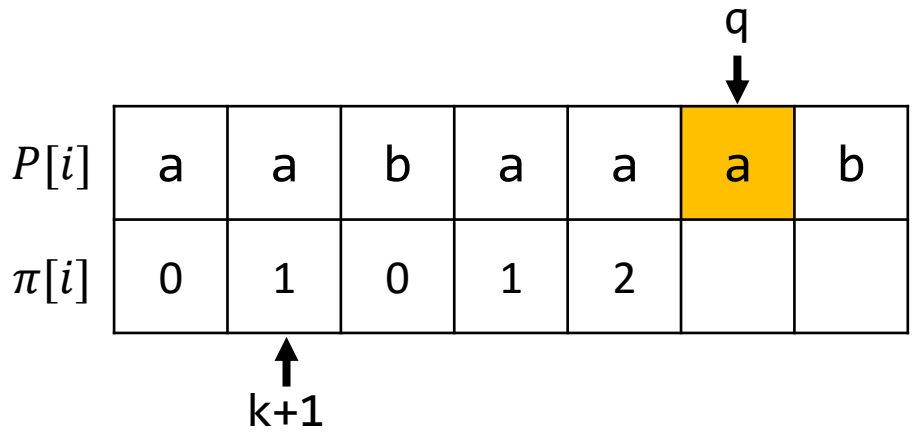
Prefix Function Algorithm – Another go

```
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
```



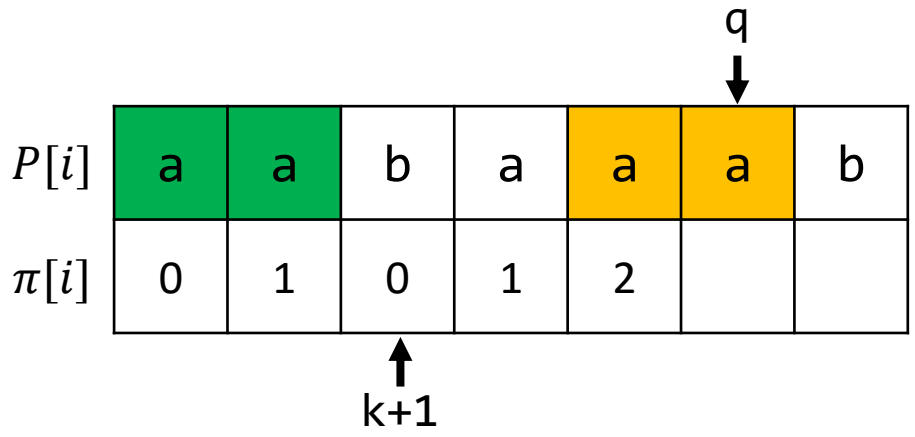
Prefix Function Algorithm – Another go

```
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
```



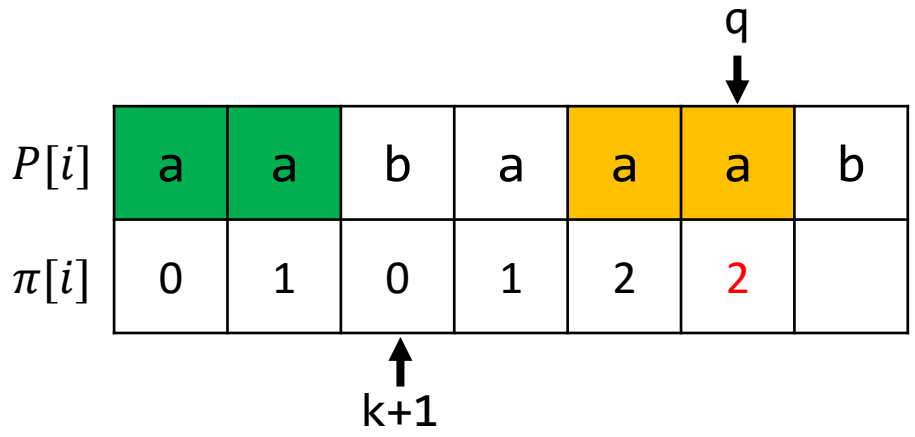
Prefix Function Algorithm – Another go

```
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
```



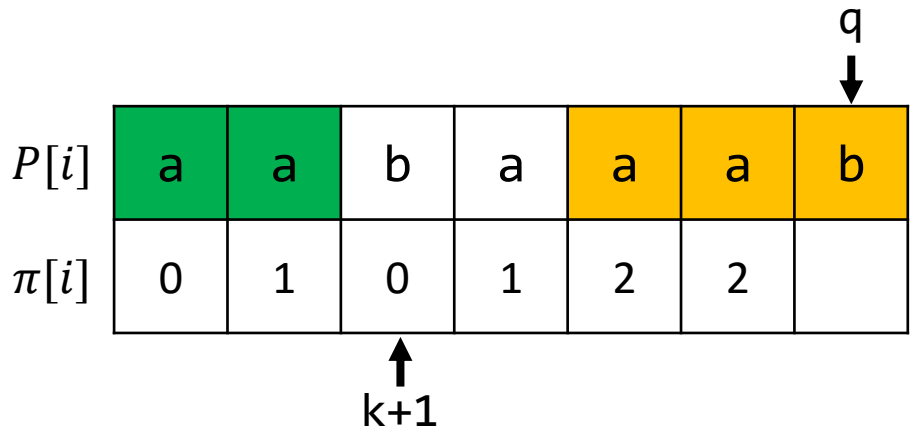
Prefix Function Algorithm – Another go

```
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
```



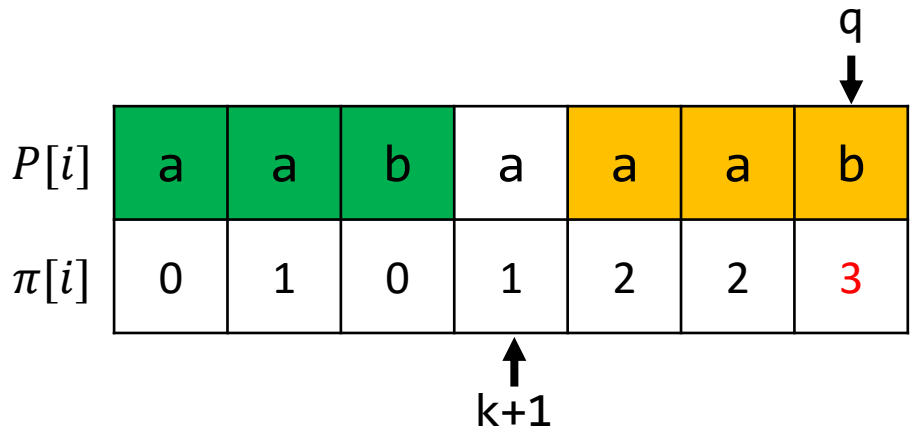
Prefix Function Algorithm – Another go

```
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 Algorithm – Another go

```
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
```



Recap: String Matching

Given a text T and a pattern P

- The string-matching problem is about finding all shifts s in the range $0 \leq s \leq n - m$ such that $P \sqsupset T_{s+m}$.
- `NAIVE-STRING-MATCHER` is slow, because it ignores valuable information that can be extracted from P
  - P=$aaab$ and s=0 is valid, means s=1,2,3 cannot be valid
- `FASTER-NAIVE-STRING-MATCHER` is broken, because it misses legitimate prefixes
- Both problems can potentially be fixed via **preprocessing** on the pattern, using the `PREFIX-FUNCTION`

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

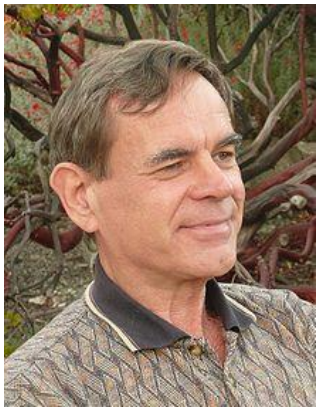
i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

Knuth-Morris-Pratt Matching

Donald Knuth (*1938)

- Author of The Art of Computer Programming
- “Father of the analysis of algorithms”
- Knuth is strongly opposed to the policy of granting software patents.
- Awards (among others)
  - ACM Grace Murray Hopper Award
  - ACM Turing Award
  - John von Neumann Medal
  - Kyoto Prize

Donald Knuth



Vaughan Pratt

James Morris

Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = 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(i-m)
12:         q = π[q]         # look for the next match
```

Knuth-Morris-Pratt Matching

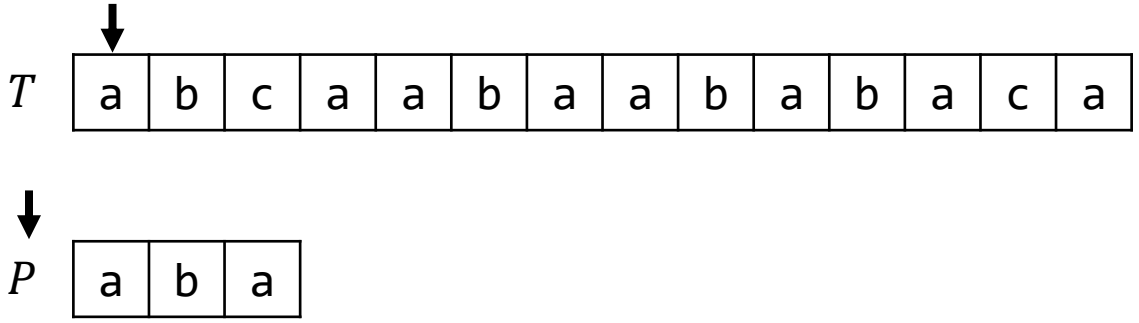
```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

```
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
```

Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

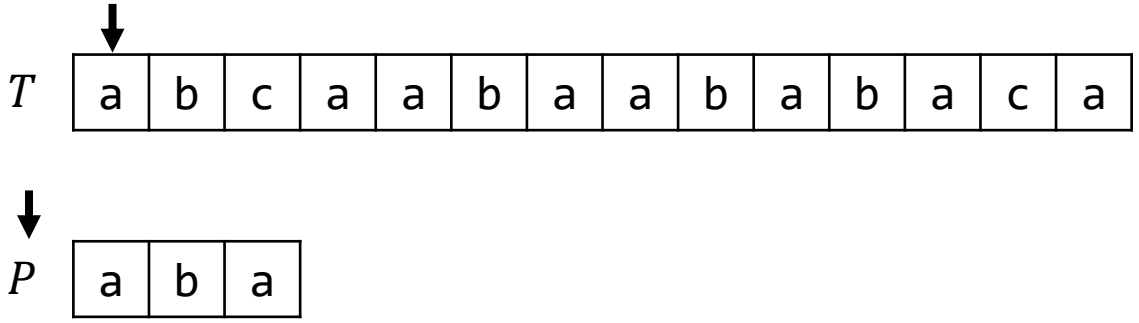
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$			



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

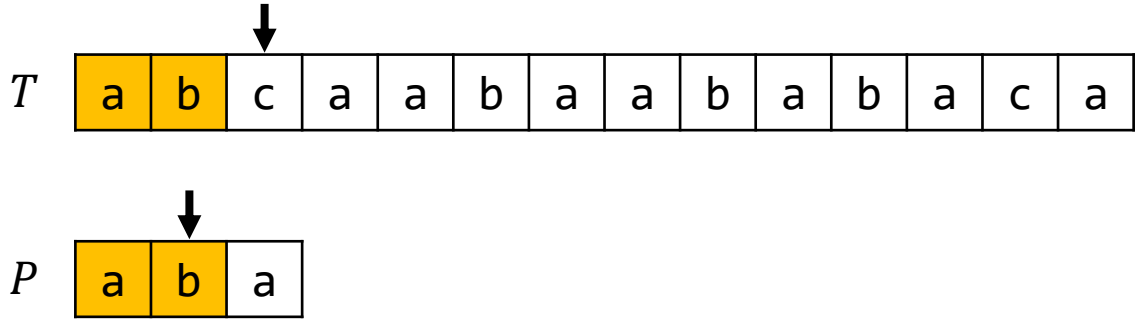
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

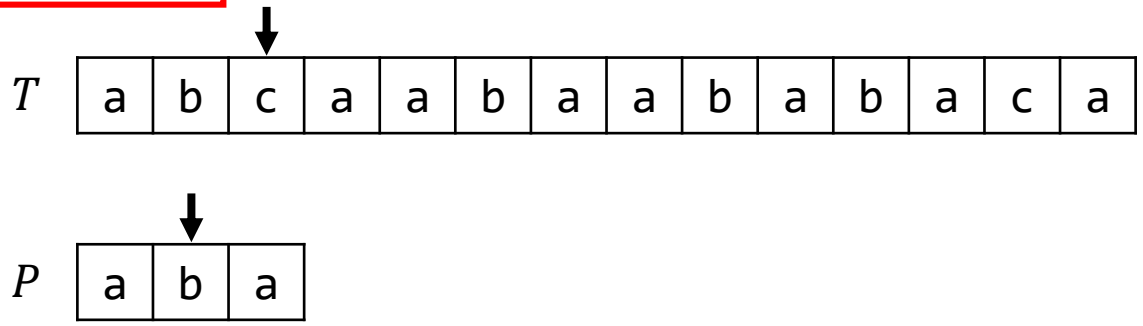
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

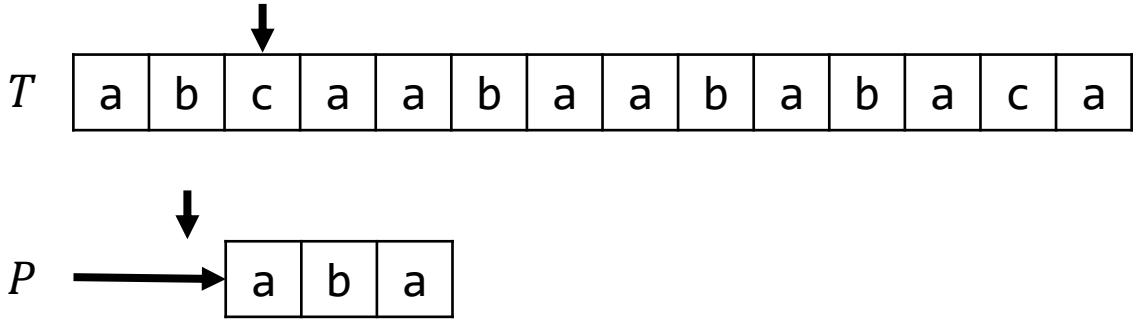
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

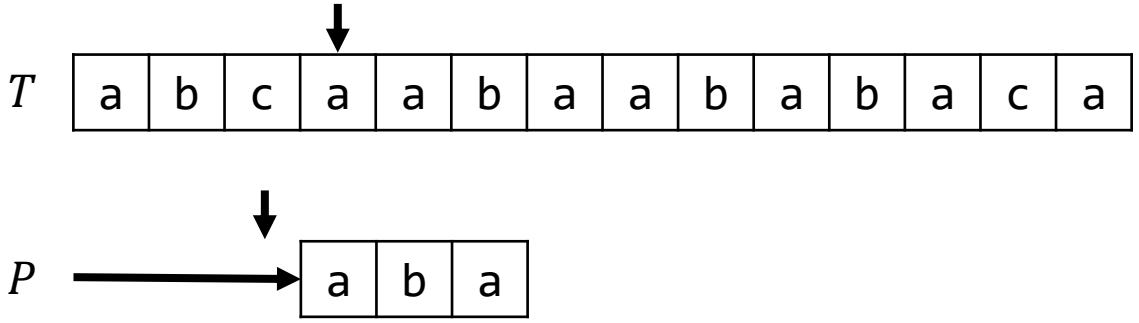
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

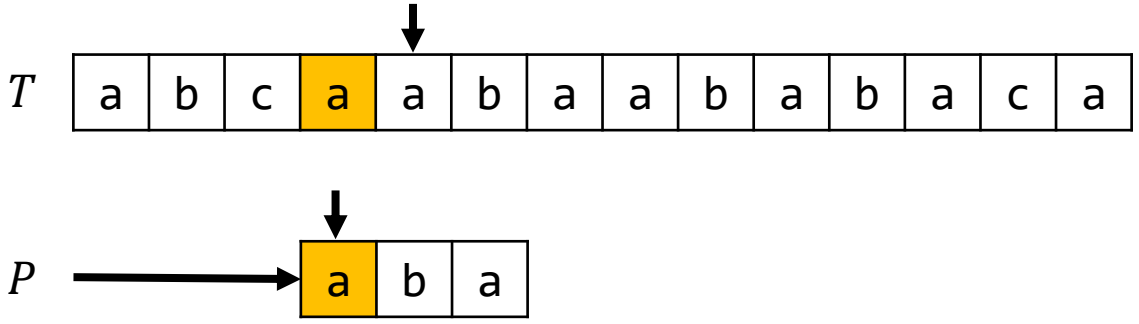
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

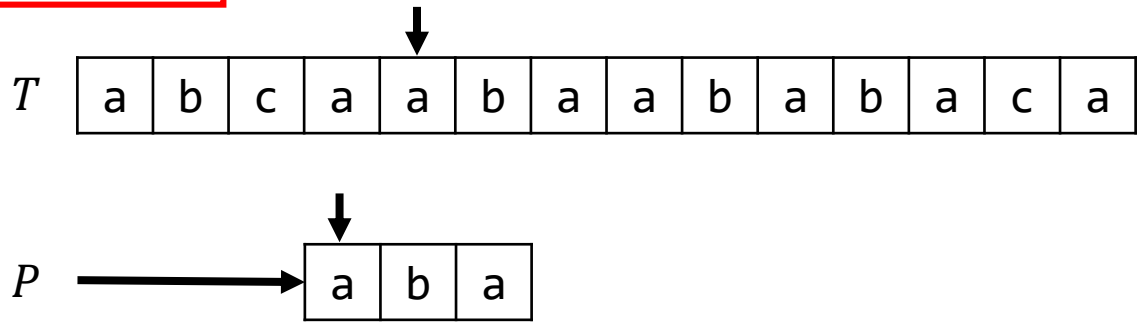
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

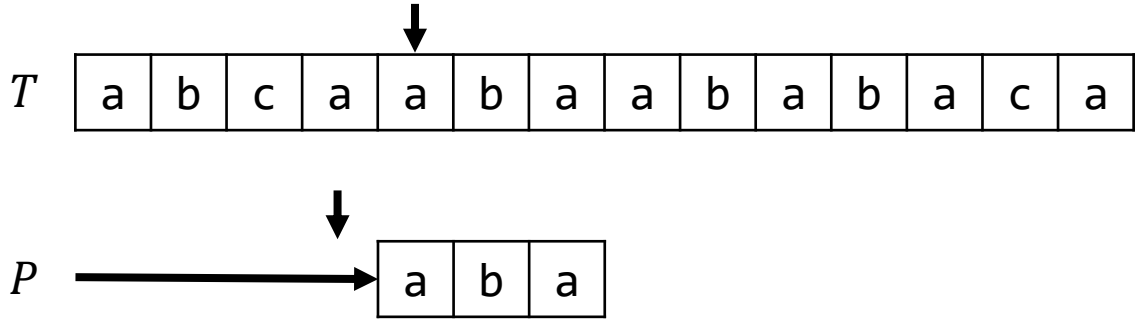
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

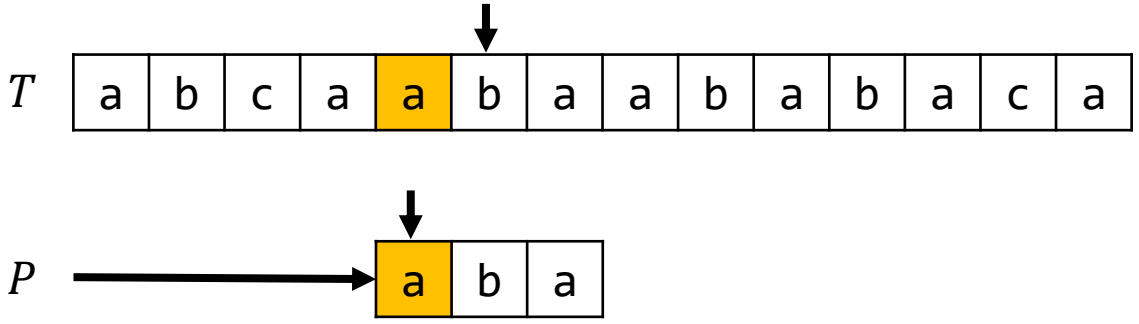
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

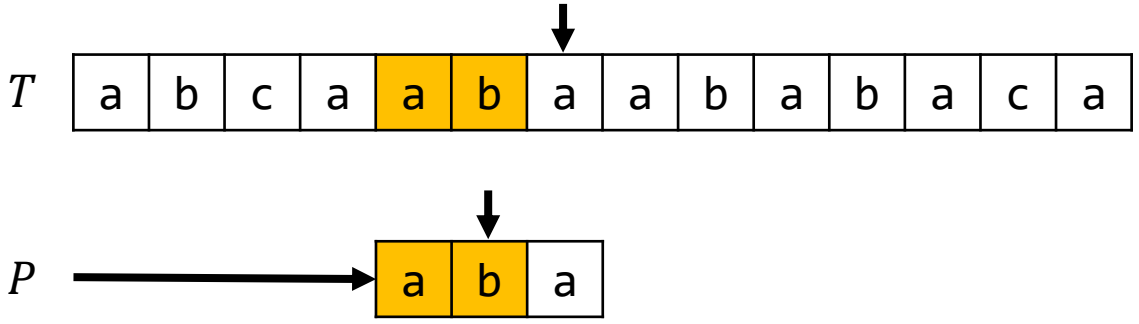
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

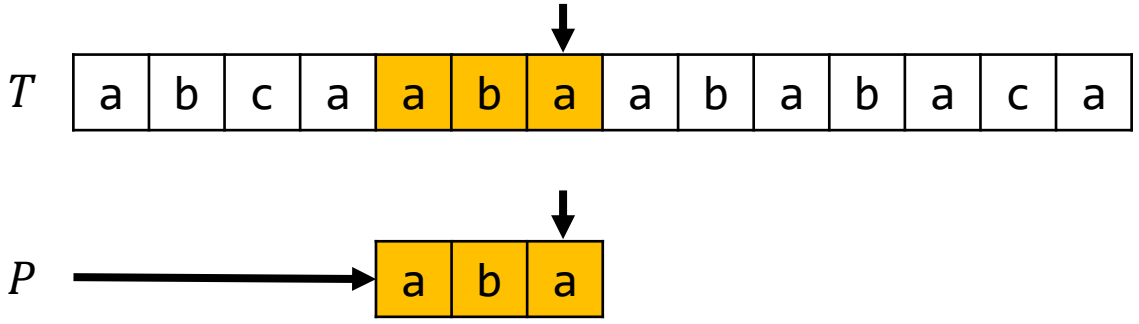
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

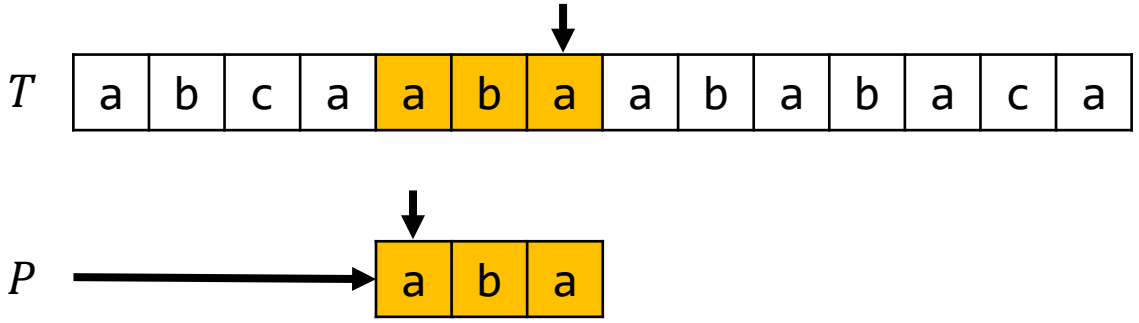
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

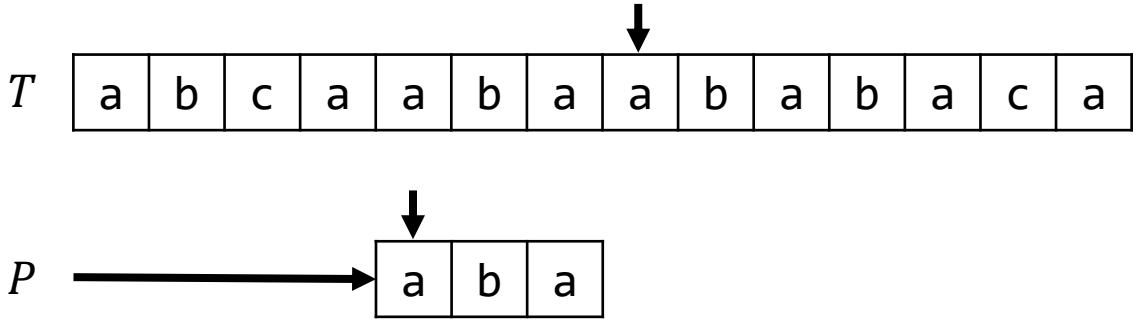
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

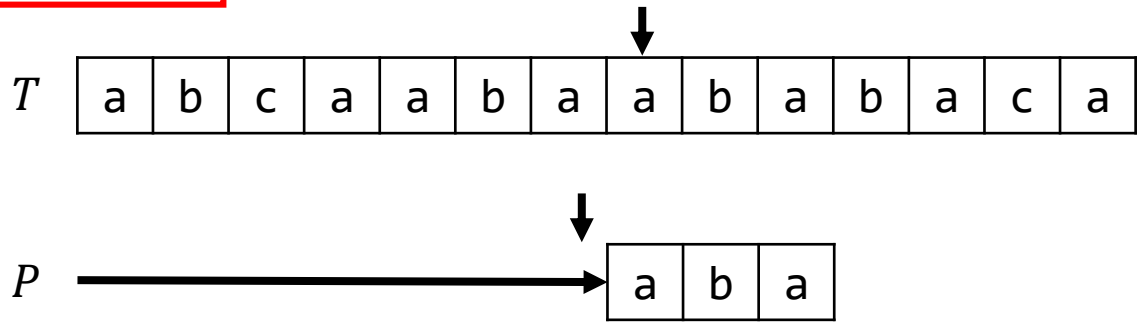
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

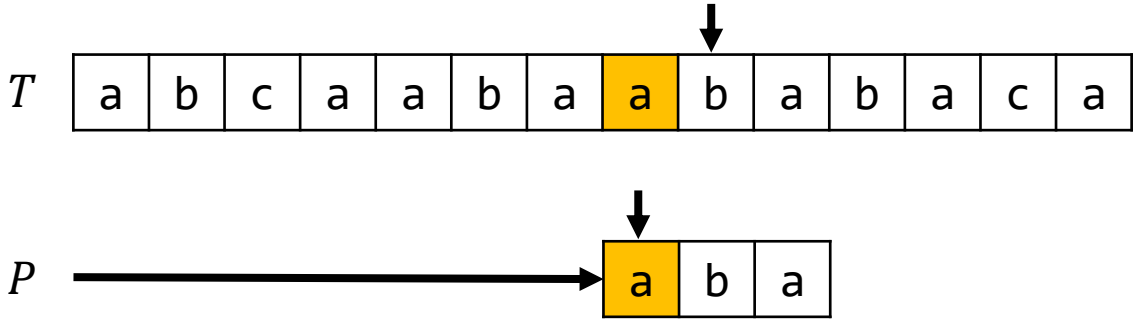
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

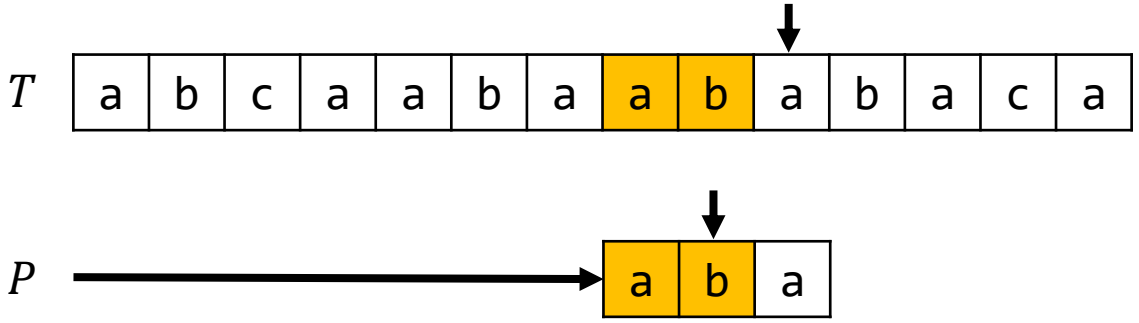
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

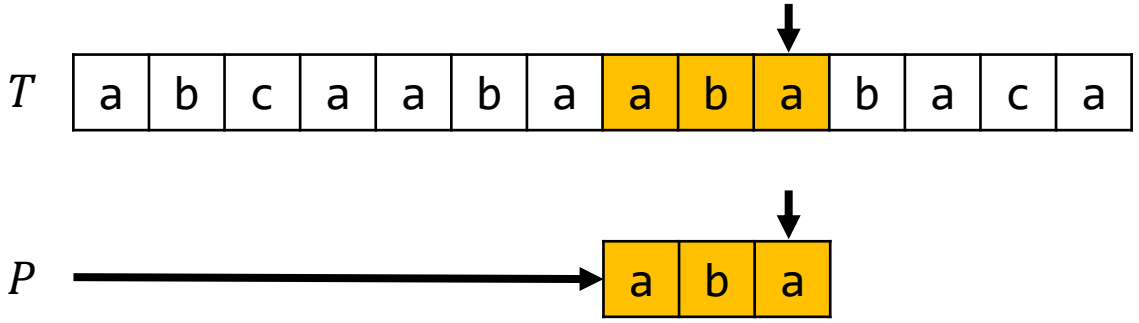
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

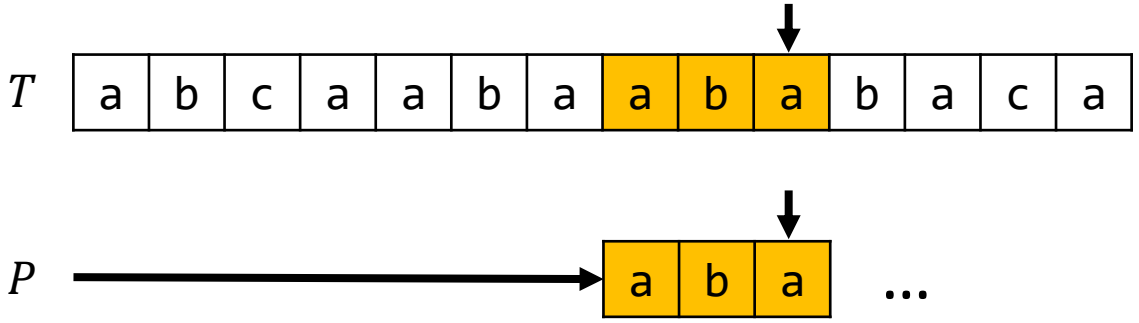
i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Knuth-Morris-Pratt Matching

```
KMP-MATCHER(T,P)
 1: n = T.length
 2: m = P.length
 3: π = PREFIX-FUNCTION(P)
 4: q = 0
 5: for i = 1 to n
 6:     while q > 0 and P[q+1] ≠ T[i]
 7:         q = π[q]
 8:     if P[q+1] == T[i]
 9:         q = q+1
10:     if q == m
11:         PRINT(i-m)
12:         q = π[q]
```

i	1	2	3
$P[i]$	a	b	a
$\pi[i]$	0	0	1



Running Time of Knuth-Morris-Pratt

First, let us consider the running time of PREFIX-FUNCTION

Observations

- Line 4 starts k at 0, and k increases in line 9, which executes at most once per for-loop iteration and thus, the total increase is at most $m - 1$.
- Since $k < q$ upon entering the for-loop and each iteration increments q, we always have $k < q$.
- Therefore, lines 3 and 10 ensure that $\pi[q] < q$.
- This means that each iteration of the while-loop decreases k, and k never becomes negative.
- Putting all together, the total decrease in k from the while-loop is bounded from above by the total increase in k over all iterations of the for-loop which is $m - 1$.
- Thus, the while-loop iterates at most $m - 1$ times in total, and PREFIX-FUNCTION runs in time $O(m)$.

```
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
```

Aggregate method of **amortized analysis** (see Section 17.1 in [Cormen])

Running Time of Knuth-Morris-Pratt

The running time of KMP-MATCHER can be analysed in a very similar way, using aggregate analysis, and yields $O(n)$.

Algorithm	Preprocessing	Matching
Naive	0	$O(nm)$
Faster-Naive	0	$O(n)$
Knuth-Morris-Pratt		

Running Time of Knuth-Morris-Pratt

The running time of KMP-MATCHER can be analysed in a very similar way, using aggregate analysis, and yields $O(n)$.

Algorithm	Preprocessing	Matching
Naive	0	$O(nm)$
Faster-Naive	0	$O(n)$
Knuth-Morris-Pratt	$O(m)$	$O(n)$

Running Time of Knuth-Morris-Pratt

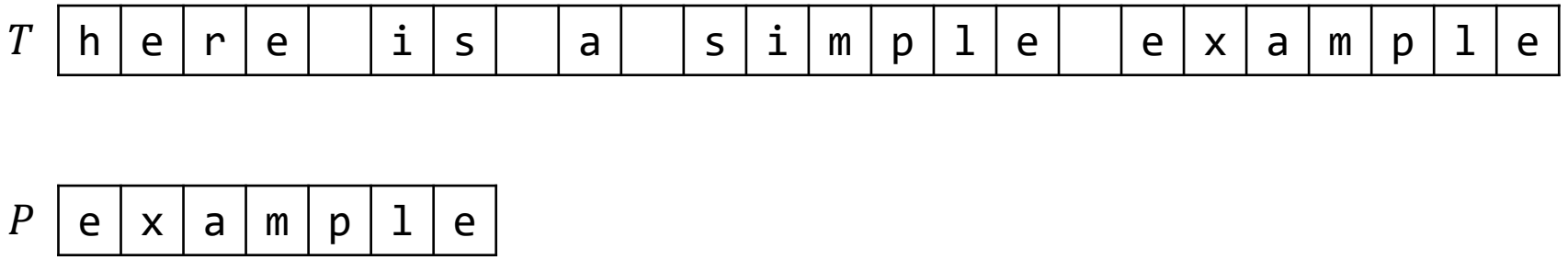
The running time of KMP-MATCHER can be analysed in a very similar way, using aggregate analysis, and yields $O(n)$.

Algorithm	Preprocessing	Matching
Naive	0	$O(nm)$
~~Faster-Naive~~ (broken)	~~0~~	~~$O(n)$~~
Knuth-Morris-Pratt	$O(m)$	$O(n)$

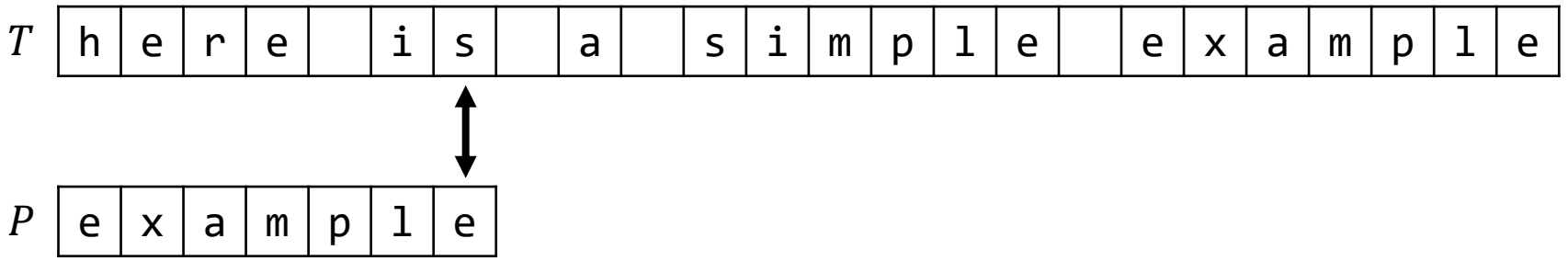
Can we do better?

This means we would need a sublinear algorithm. Skip some text?

Yet Another New Strategy



Yet Another New Strategy

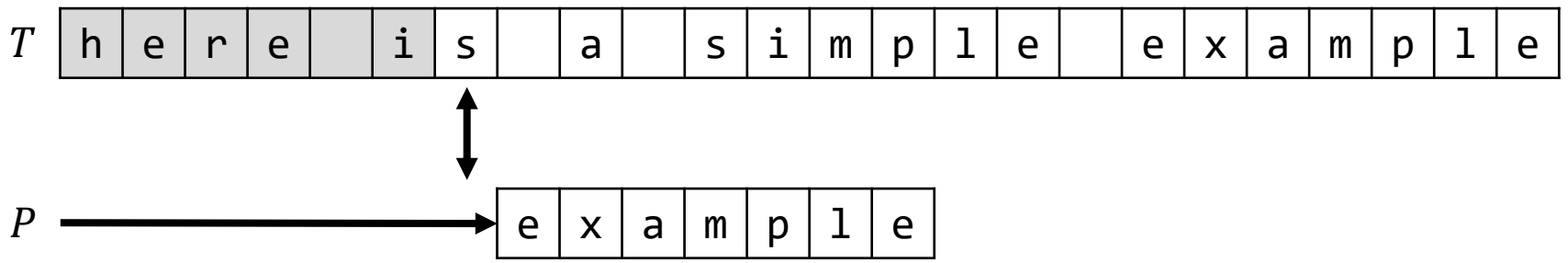


Match P from **right-to-left**

- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

Yet Another New Strategy

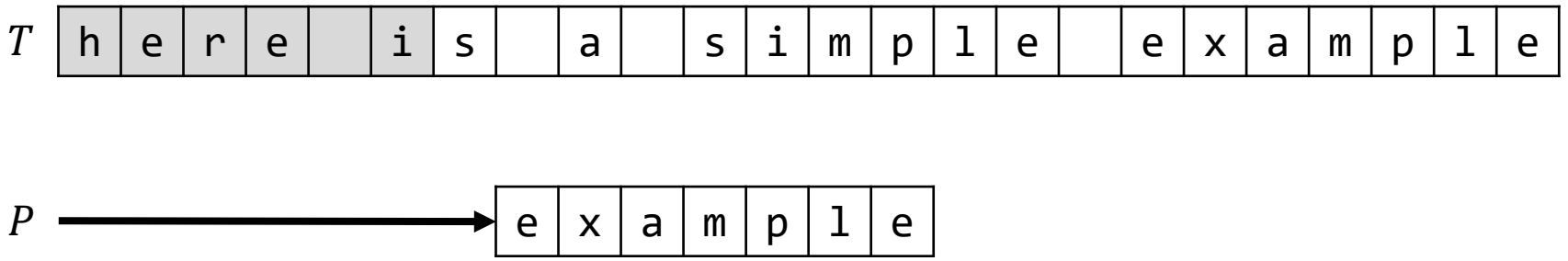


Match P from **right-to-left**

- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

Yet Another New Strategy

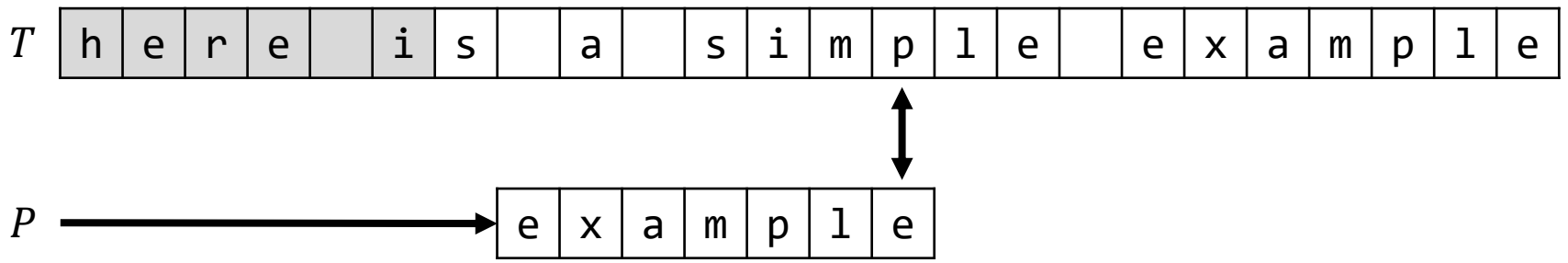


Match P from **right-to-left**

- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

Yet Another New Strategy

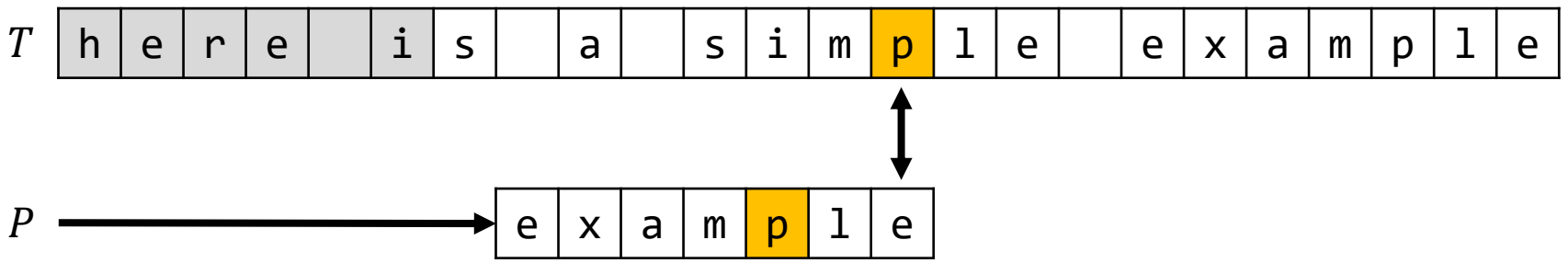


Match P from **right-to-left**

- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

Yet Another New Strategy



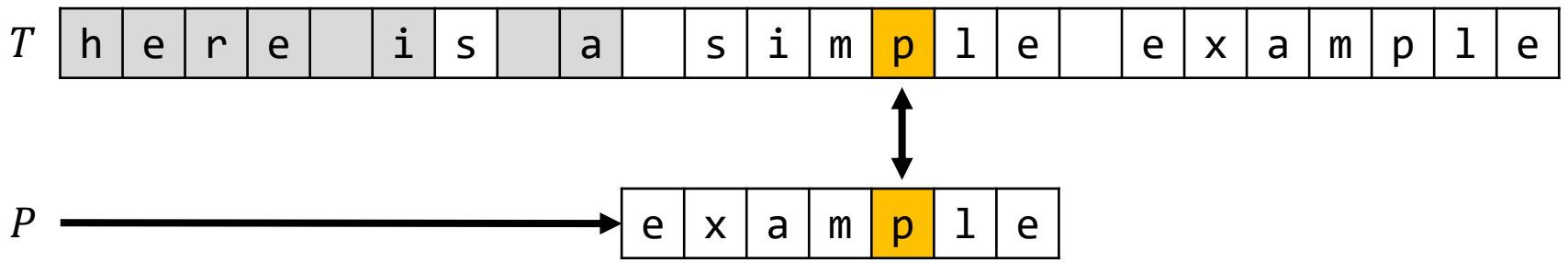
Match P from **right-to-left**

- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

Yet Another New Strategy



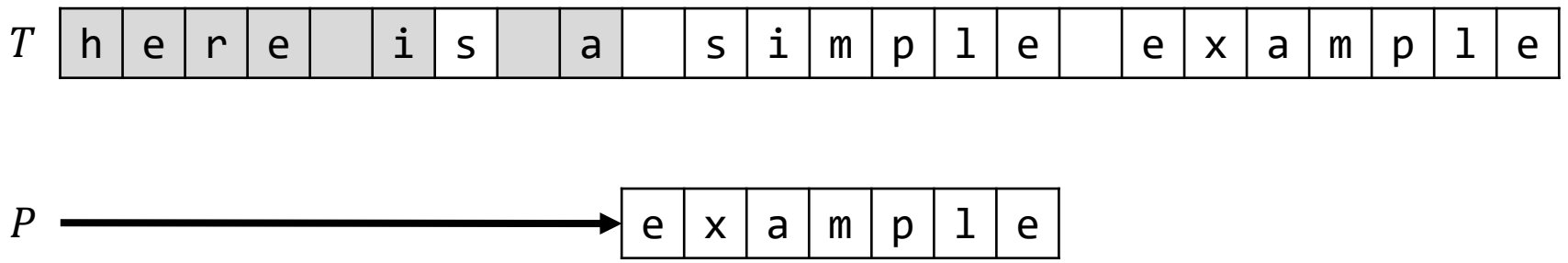
Match P from **right-to-left**

- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

Yet Another New Strategy



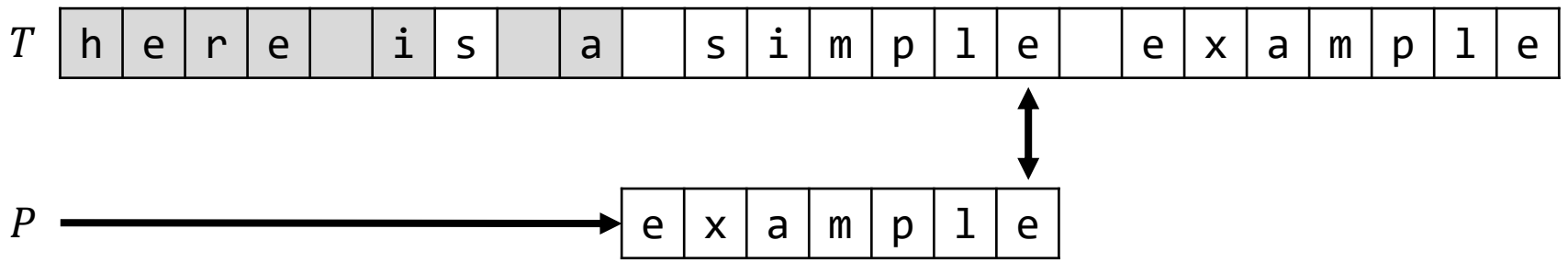
Match P from **right-to-left**

- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

Yet Another New Strategy



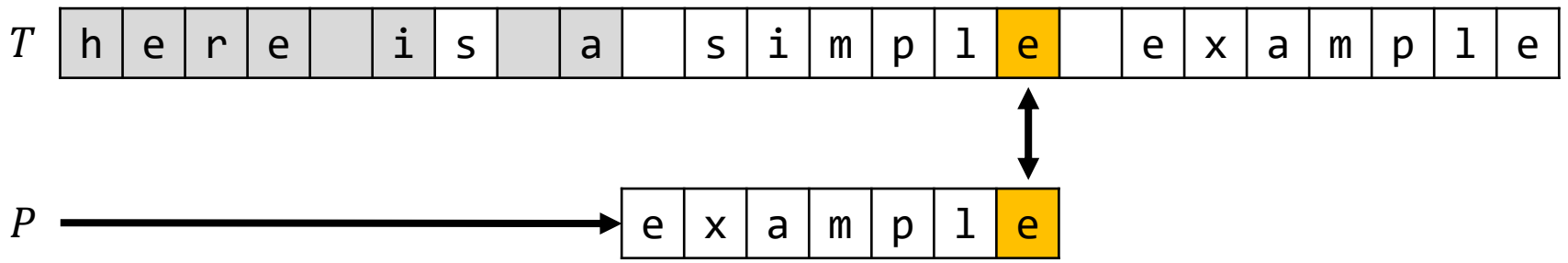
Match P from **right-to-left**

- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

Yet Another New Strategy



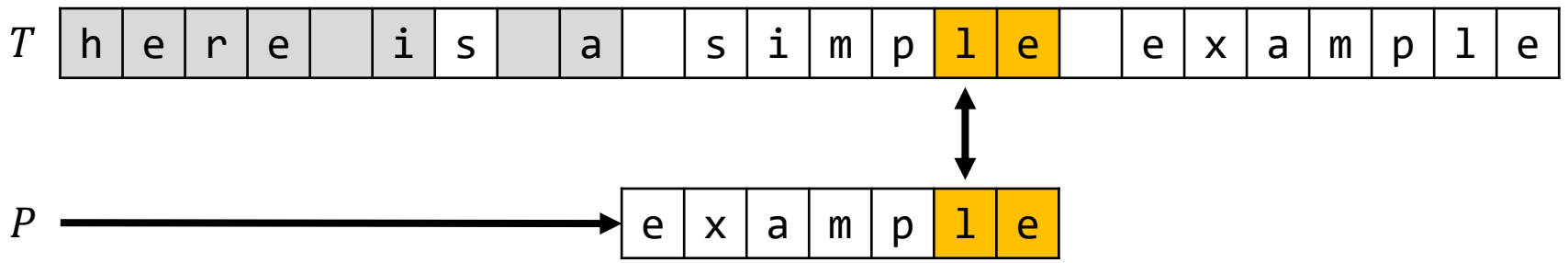
Match P from **right-to-left**

- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

Yet Another New Strategy



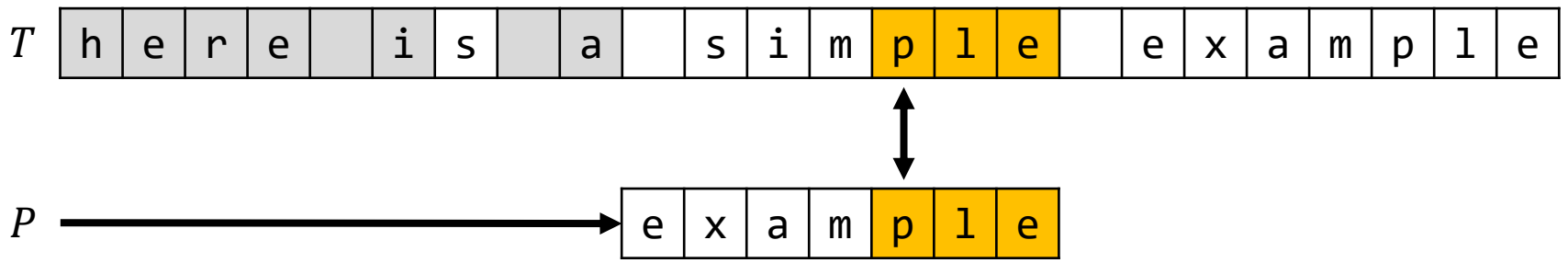
Match P from **right-to-left**

- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

Yet Another New Strategy



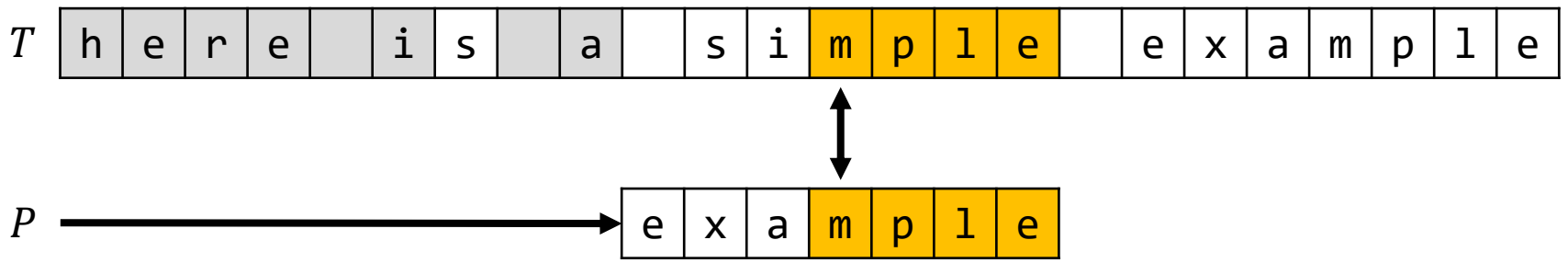
Match P from **right-to-left**

- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

Yet Another New Strategy



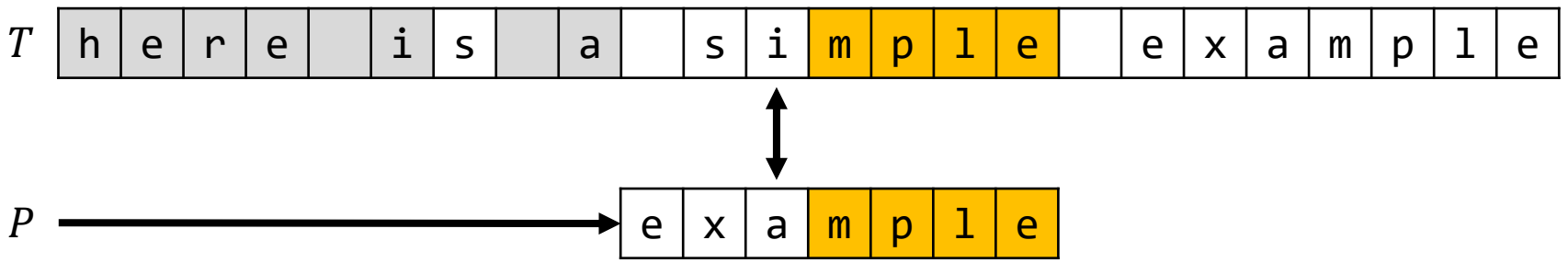
Match P from **right-to-left**

- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

Yet Another New Strategy



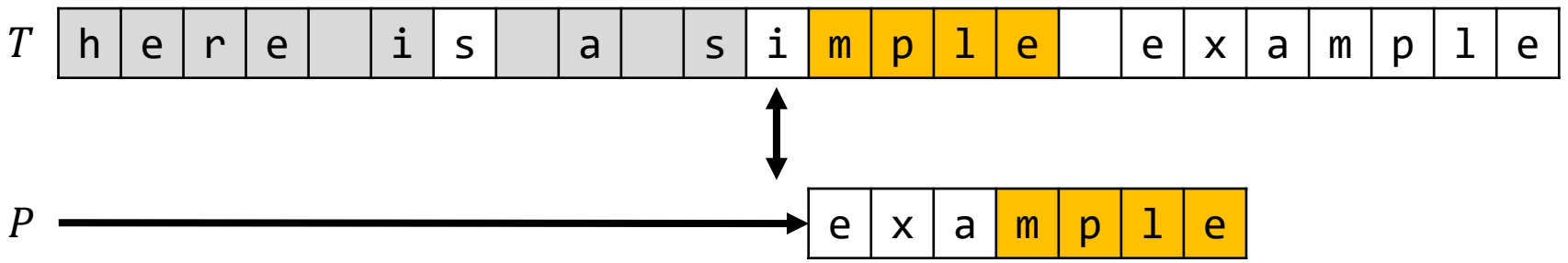
Match P from **right-to-left**

- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

Yet Another New Strategy



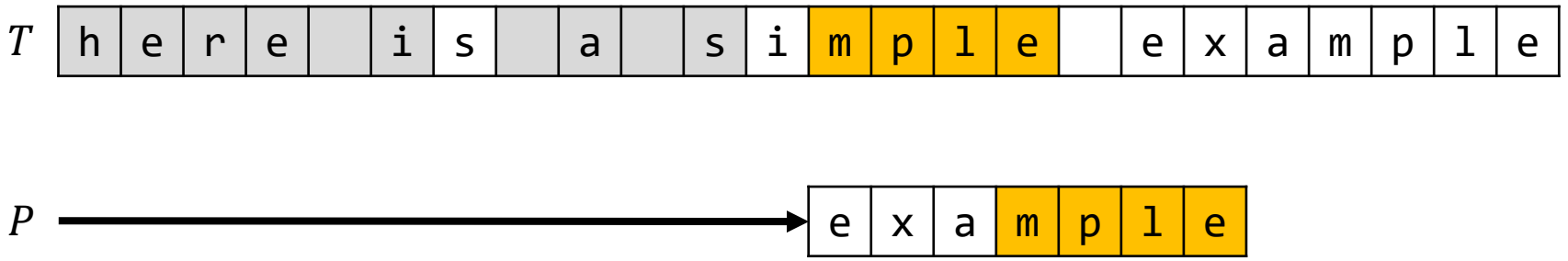
Match P from **right-to-left**

- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

Yet Another New Strategy



Match P from **right-to-left**

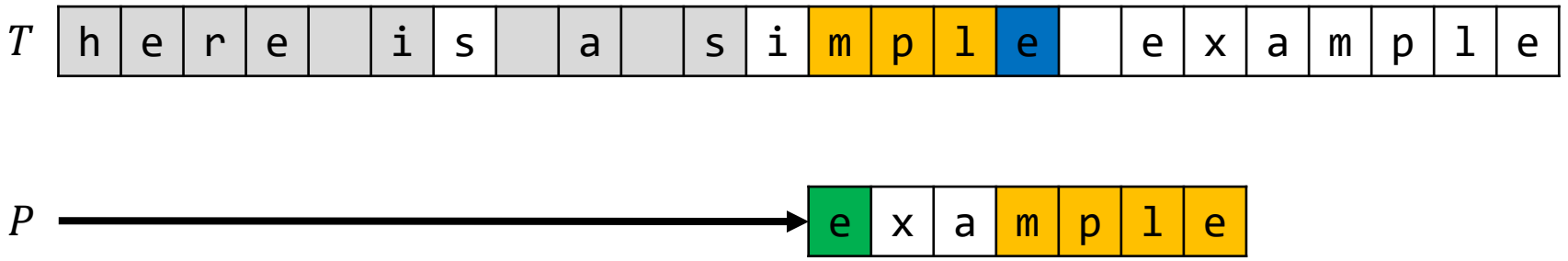
- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Yet Another New Strategy



Match P from **right-to-left**

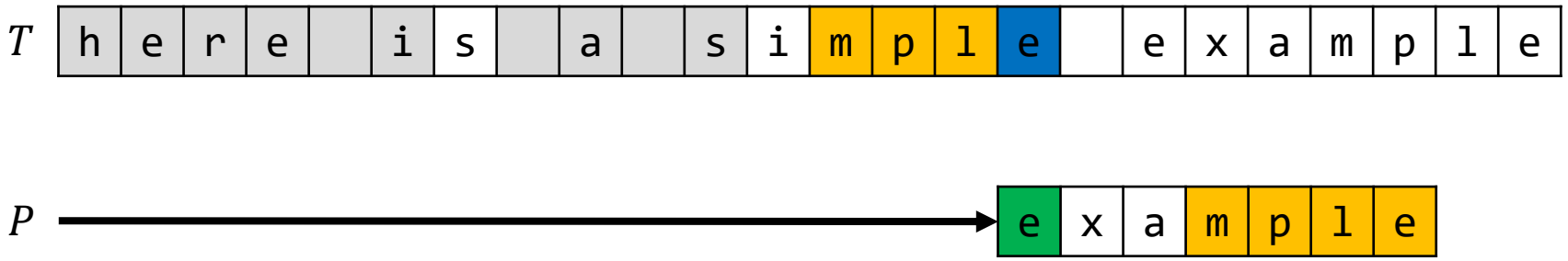
- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Yet Another New Strategy



Match P from **right-to-left**

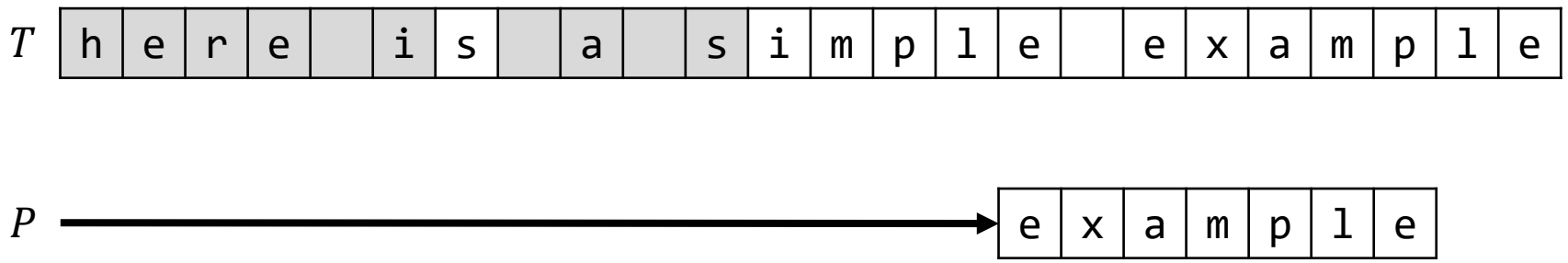
- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Yet Another New Strategy



Match P from **right-to-left**

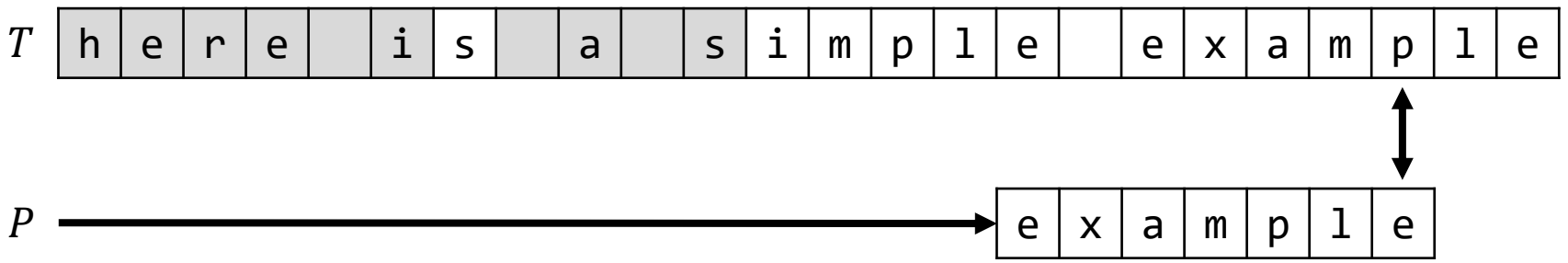
- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Yet Another New Strategy



Match P from **right-to-left**

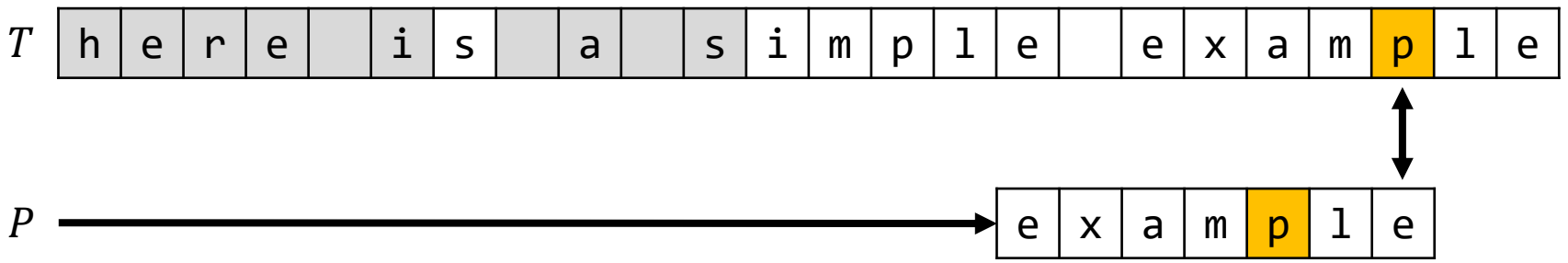
- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Yet Another New Strategy



Match P from **right-to-left**

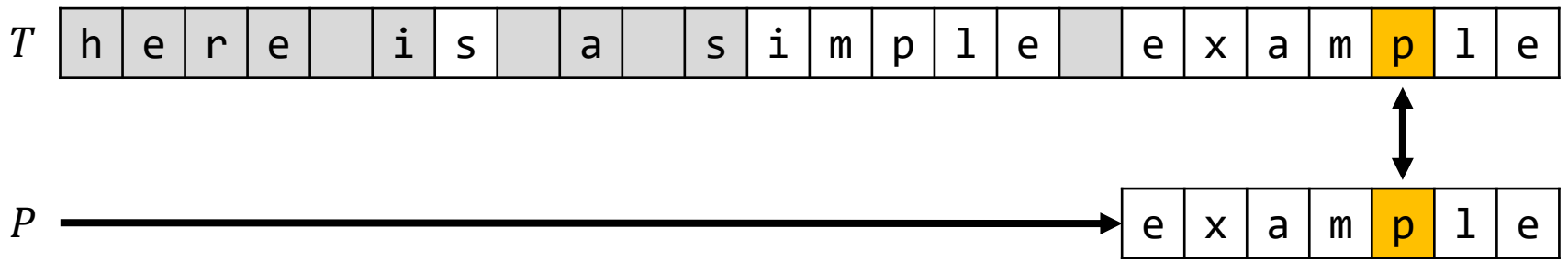
- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Yet Another New Strategy



Match P from **right-to-left**

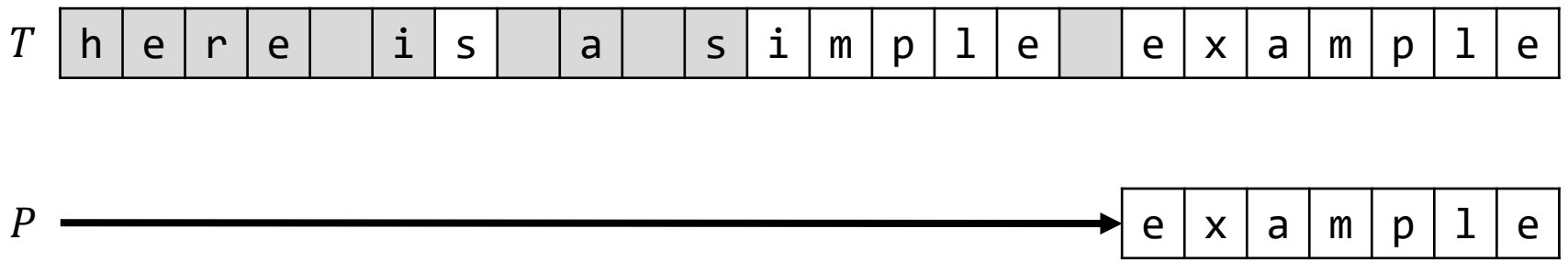
- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Yet Another New Strategy



Match P from **right-to-left**

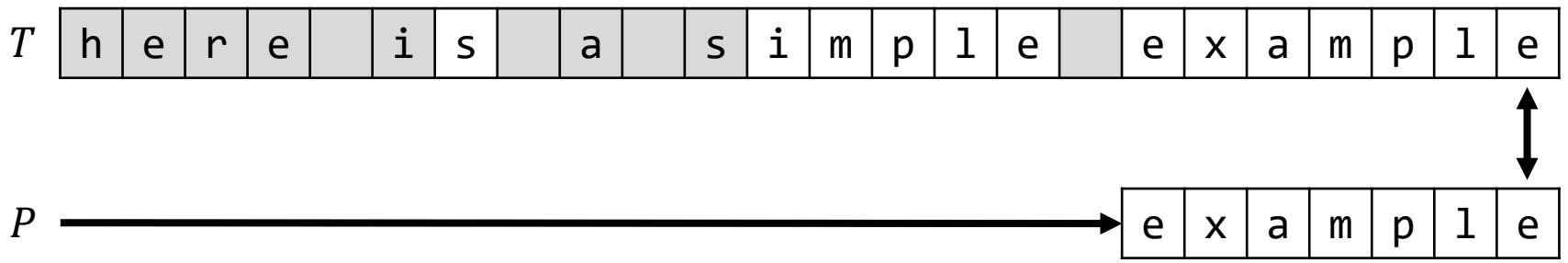
- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Yet Another New Strategy



Match P from **right-to-left**

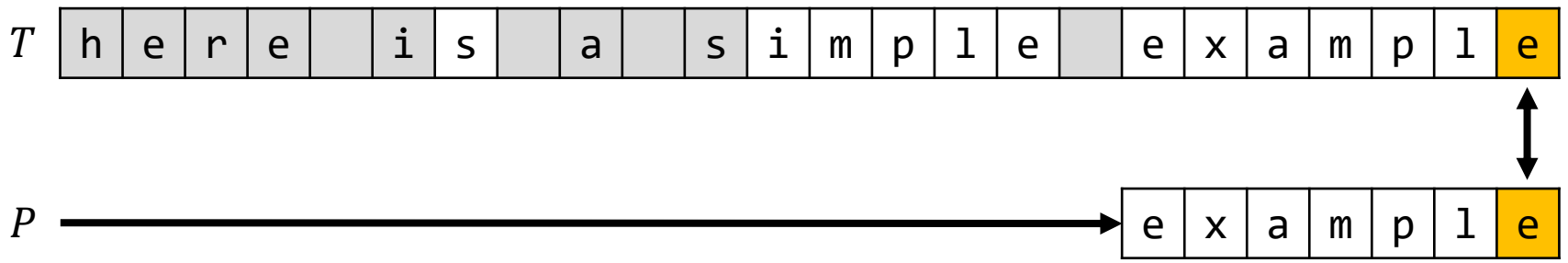
- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Yet Another New Strategy



Match P from **right-to-left**

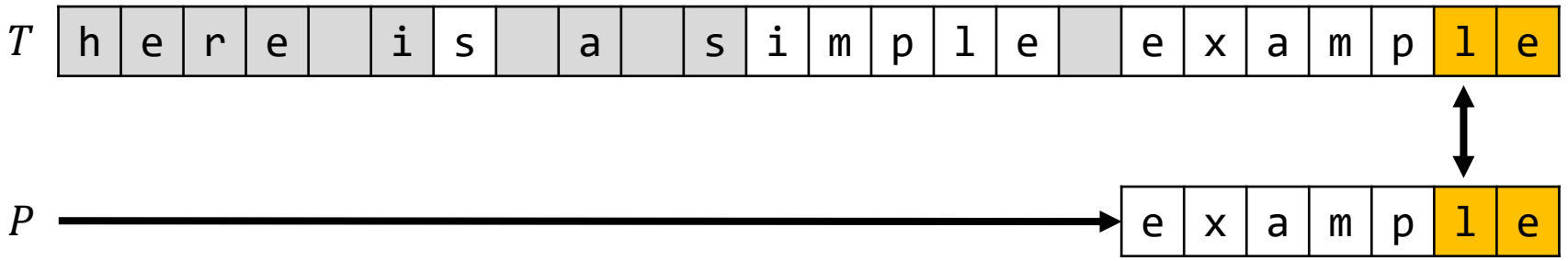
- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Yet Another New Strategy



Match P from **right-to-left**

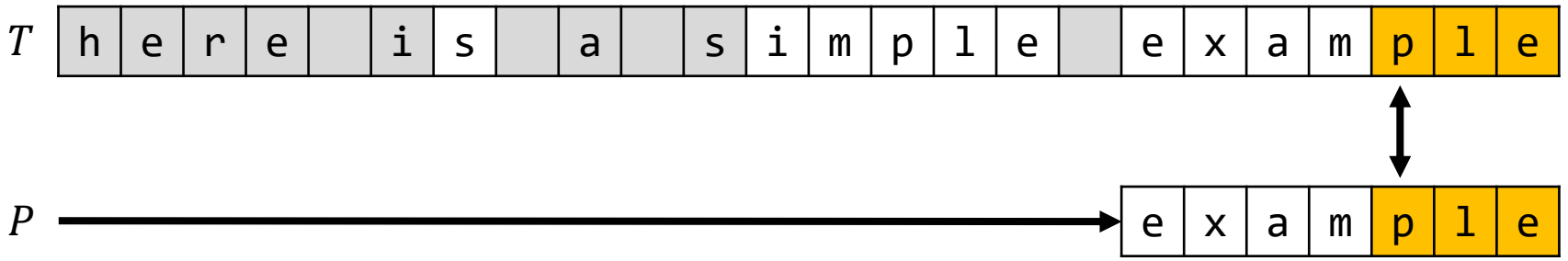
- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Yet Another New Strategy



Match P from **right-to-left**

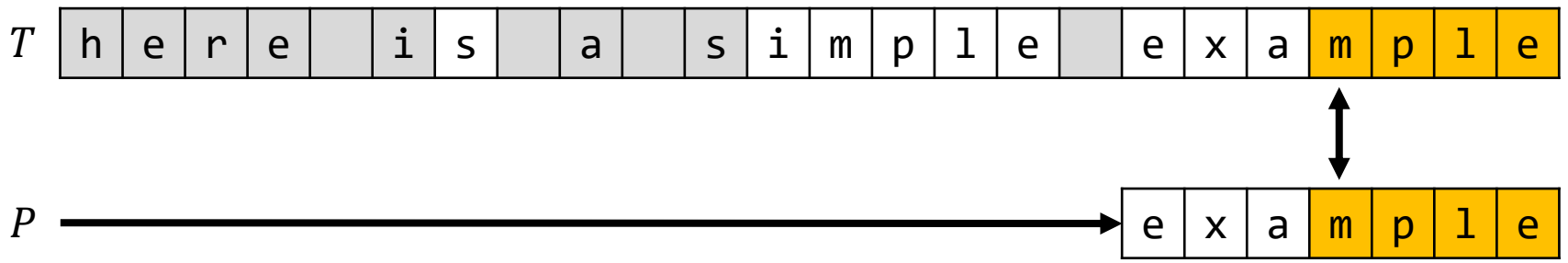
- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Yet Another New Strategy



Match P from **right-to-left**

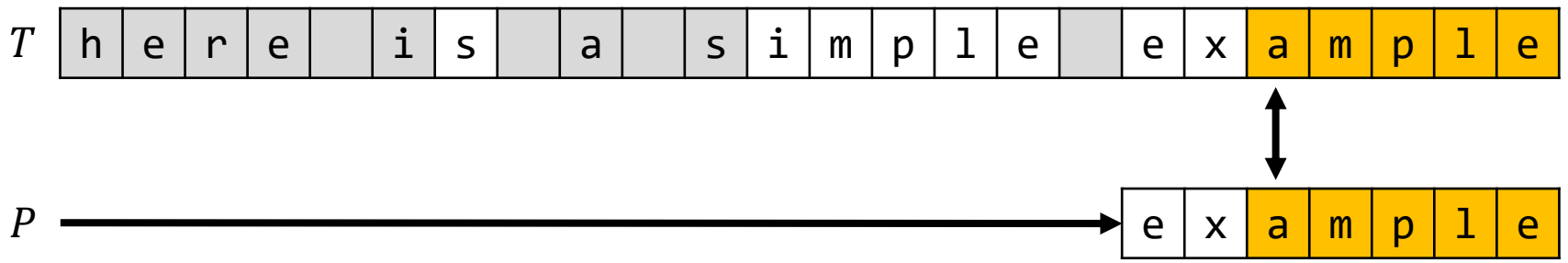
- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Yet Another New Strategy



Match P from **right-to-left**

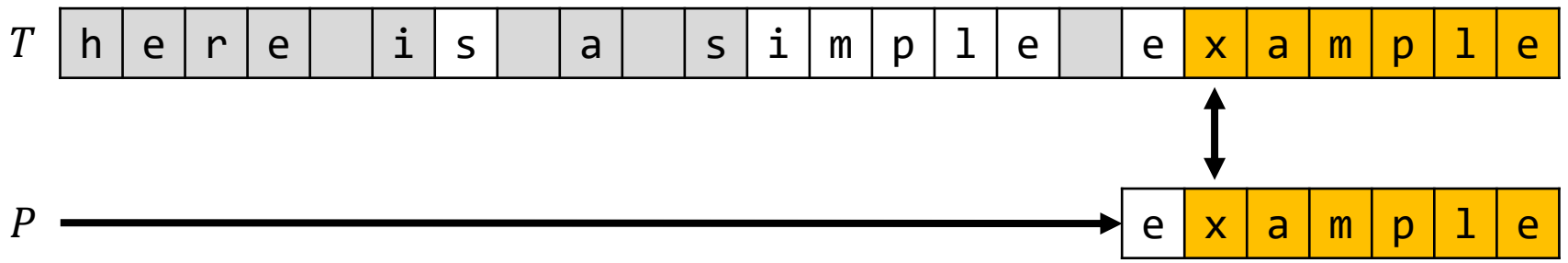
- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Yet Another New Strategy



Match P from **right-to-left**

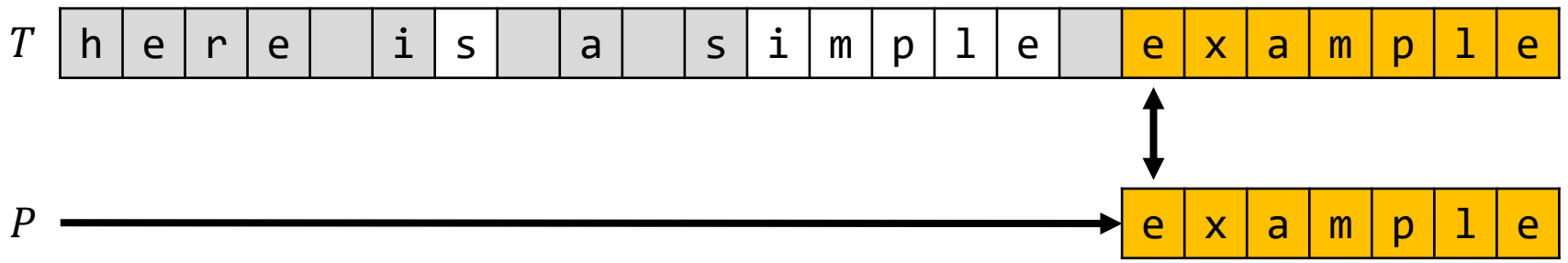
- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Yet Another New Strategy



Match P from **right-to-left**

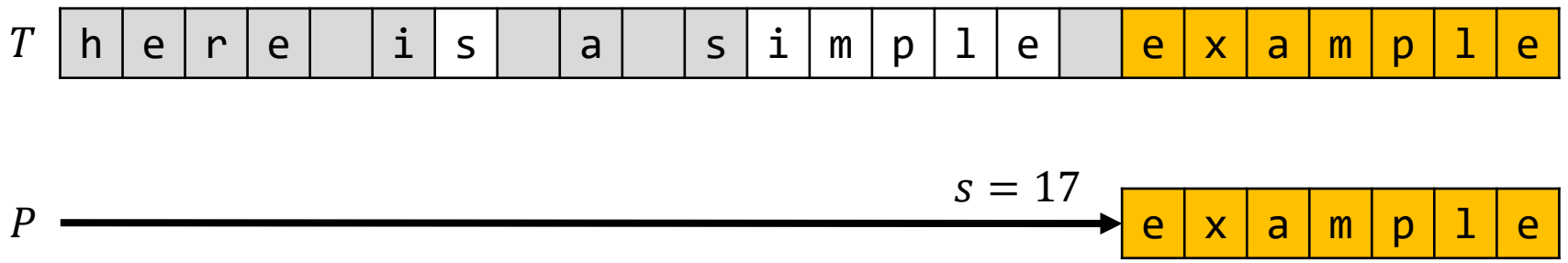
- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Yet Another New Strategy



Match P from **right-to-left**

- If there is a **bad character** β in T, then **shift**

  so that P **skips** β, if β is not in P

  so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

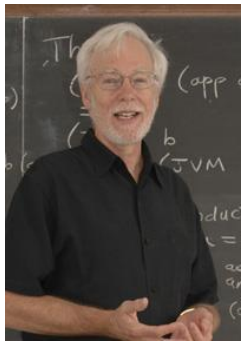
We skipped 11 out of 24 characters in text T.

Boyer-Moore Matching

Strategy of Boyer-Moore Matching

- shift P as far right as possible, thereby maximising the number of skipped comparisons
- **Bad character rule (BCR)**: current character in T does not match current character in P
- **Good suffix rule (GSR)**: a suffix of P has been matched in T up to a bad character in T
- Shift the maximum indicated by BCR or GSR
- Amount of shift is dependent only on P, so can be **precomputed** and stored (trading space for time)

Robert Boyer



J Moore

Shift Rules

Match P from right-to-left

- If there is a **bad character** β in T, then **shift**

  a) so that P **skips** β, if β is not in P

  b) so that P is **aligned** with the right-most occurrence of β in P, if β is in P

  c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Bad Character Rule

If there is a **bad character** β in T, then **shift**

a) so that P **skips** β, if β is not in P

b) so that P is **aligned** with the right-most occurrence of β in P, if β is in P

```
BCR-TABLE(P)
 1: n = Σ.length
 2: m = P.length
 3: let bcr[1..n] be a new array
 4: for i = 1 to n
 5:     bcr[i] = 0
 6: for j = 1 to m
 7:     bcr[P[j]] = j
 8: return bcr
```

Bad Character Rule

If there is a **bad character** β in T, then **shift**

a) so that P **skips** β, if β is not in P

b) so that P is **aligned** with the right-most occurrence of β in P, if β is in P

```
BCR-TABLE(P)
 1: n = Σ.length
 2: m = P.length
 3: let bcr[1..n] be a new array
 4: for i = 1 to n
 5:     bcr[i] = 0
 6: for j = 1 to m
 7:     bcr[P[j]] = j
 8: return bcr
```

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a

c	a	b	c
$bcr[c]$			

Bad Character Rule

If there is a **bad character** β in T, then **shift**

a) so that P **skips** β, if β is not in P

b) so that P is **aligned** with the right-most occurrence of β in P, if β is in P

```
BCR-TABLE(P)
 1: n = Σ.length
 2: m = P.length
 3: let bcr[1..n] be a new array
 4: for i = 1 to n
 5:     bcr[i] = 0
 6: for j = 1 to m
 7:     bcr[P[j]] = j
 8: return bcr
```

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a

c	a	b	c
$bcr[c]$	7		

Bad Character Rule

If there is a **bad character** β in T, then **shift**

a) so that P **skips** β, if β is not in P

b) so that P is **aligned** with the right-most occurrence of β in P, if β is in P

```
BCR-TABLE(P)
 1: n = Σ.length
 2: m = P.length
 3: let bcr[1..n] be a new array
 4: for i = 1 to n
 5:     bcr[i] = 0
 6: for j = 1 to m
 7:     bcr[P[j]] = j
 8: return bcr
```

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a

c	a	b	c
$bcr[c]$	7	4	

Bad Character Rule

If there is a **bad character** β in T, then **shift**

a) so that P **skips** β, if β is not in P

b) so that P is **aligned** with the right-most occurrence of β in P, if β is in P

```
BCR-TABLE(P)
 1: n = Σ.length
 2: m = P.length
 3: let bcr[1..n] be a new array
 4: for i = 1 to n
 5:     bcr[i] = 0
 6: for j = 1 to m
 7:     bcr[P[j]] = j
 8: return bcr
```

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a

c	a	b	c
$bcr[c]$	7	4	6

Bad Character Rule

If there is a **bad character** β in T, then **shift**

a) so that P **skips** β, if β is not in P

b) so that P is **aligned** with the right-most occurrence of β in P, if β is in P

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

P	c	a	b	a	b

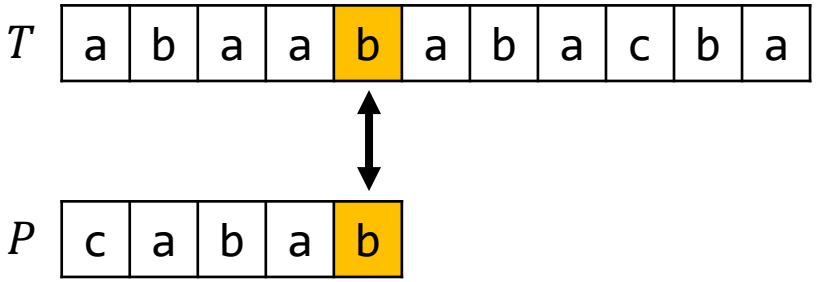
c	a	b	c
$bcr[c]$	4	5	1

Bad Character Rule

If there is a **bad character** β in T, then **shift**

a) so that P **skips** β, if β is not in P

b) so that P is **aligned** with the right-most occurrence of β in P, if β is in P



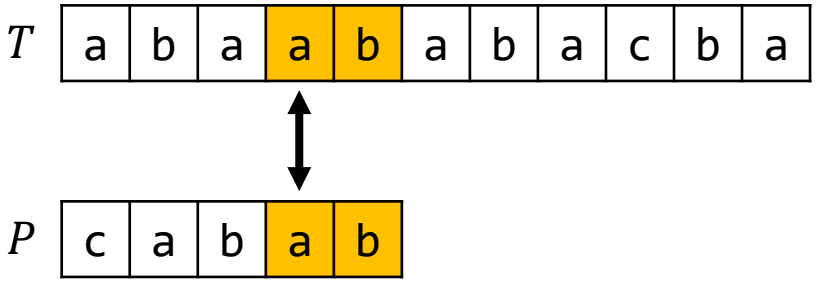
c	a	b	c
$bcr[c]$	4	5	1

Bad Character Rule

If there is a **bad character** β in T, then **shift**

a) so that P **skips** β, if β is not in P

b) so that P is **aligned** with the right-most occurrence of β in P, if β is in P



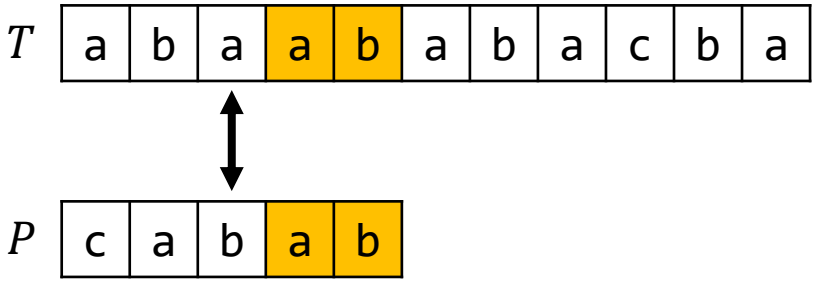
c	a	b	c
$bcr[c]$	4	5	1

Bad Character Rule

If there is a **bad character** β in T, then **shift**

a) so that P **skips** β, if β is not in P

b) so that P is **aligned** with the right-most occurrence of β in P, if β is in P



c	a	b	c
$bcr[c]$	4	5	1

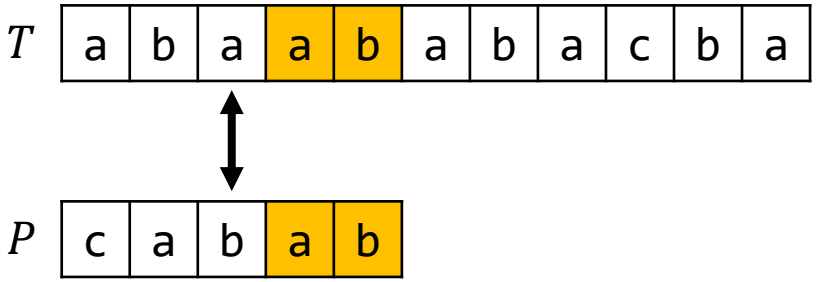
What does the BCR suggest?

Bad Character Rule

If there is a **bad character** β in T, then **shift**

a) so that P **skips** β, if β is not in P

b) so that P is **aligned** with the right-most occurrence of β in P, if β is in P



c	a	b	c
$bcr[c]$	4	5	1

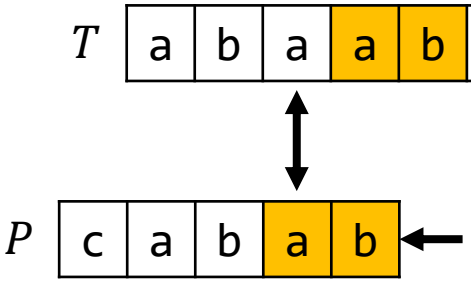
What does the BCR suggest?

Bad Character Rule

If there is a **bad character** β in T, then **shift**

a) so that P **skips** β, if β is not in P

b) so that P is **aligned** with the right-most occurrence of β in P, if β is in P



c	a	b	c
$bcr[c]$	4	5	1

What does the BCR suggest?

Negative shift!

Shift Rules

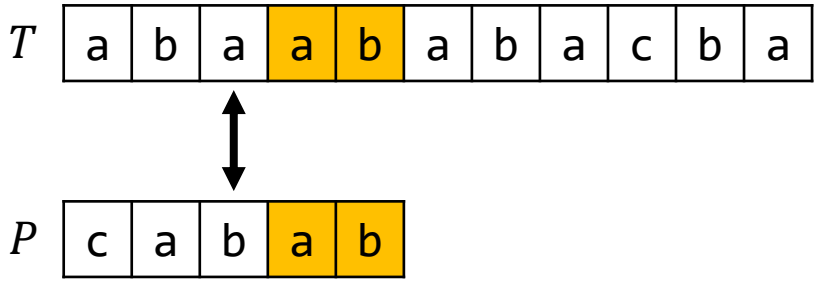
Match P from right-to-left

- If there is a **bad character** β in T, then **shift**
  a) so that P **skips** β, if β is not in P
  b) so that P is **aligned** with the right-most occurrence of β in P, if β is in P
  c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

Good Suffix Rule

If there is a **bad character** β in T, then **shift**

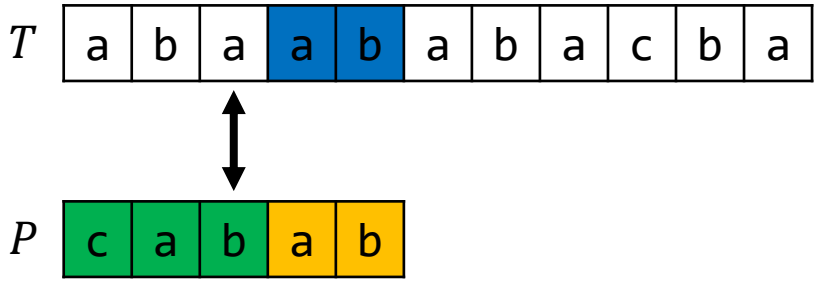
c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T



Good Suffix Rule

If there is a **bad character** β in T, then **shift**

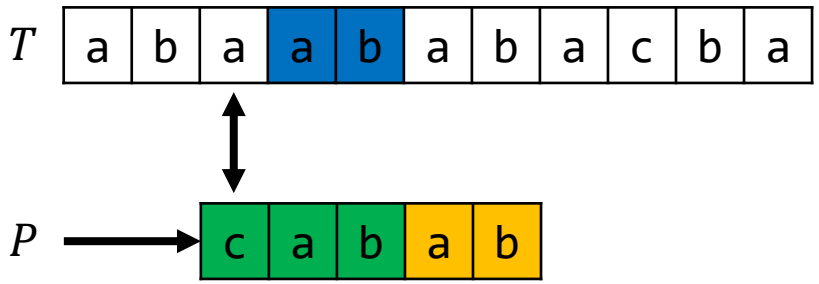
c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T



Good Suffix Rule

If there is a **bad character** β in T, then **shift**

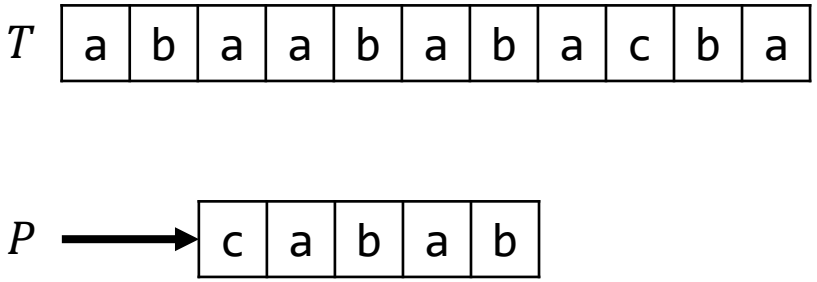
c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T



Good Suffix Rule

If there is a **bad character** β in T, then **shift**

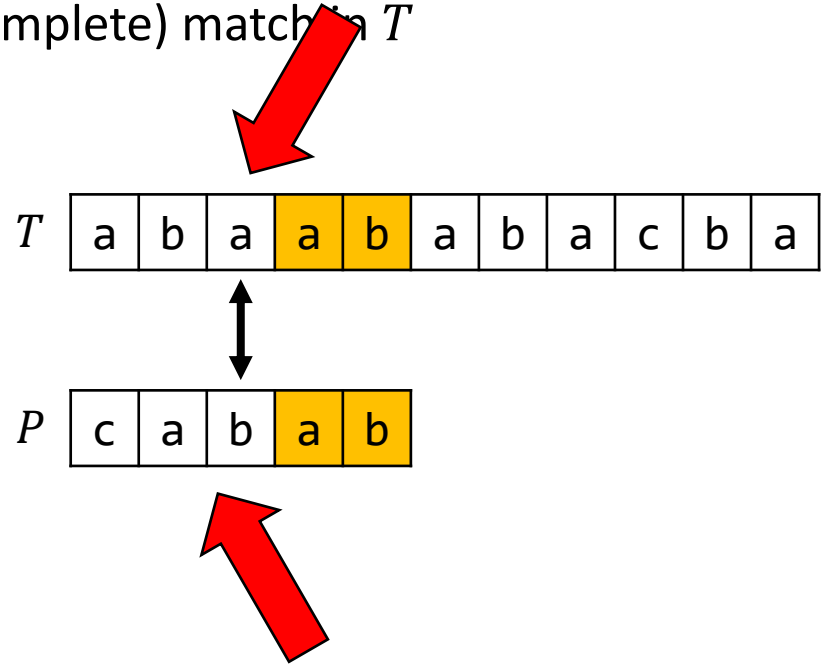
c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T



Good Suffix Rule

If there is a **bad character** β in T, then **shift**

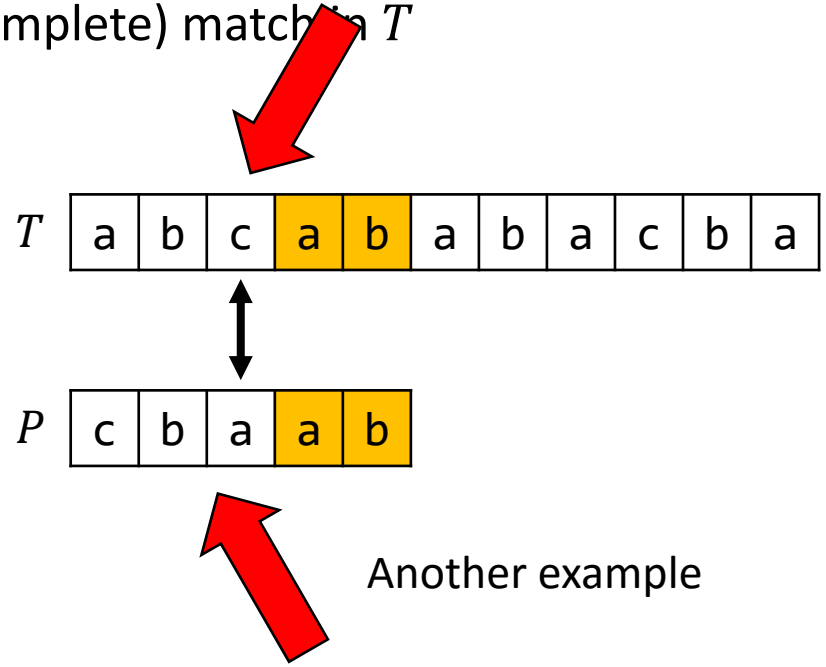
c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T



Good Suffix Rule

If there is a **bad character** β in T, then **shift**

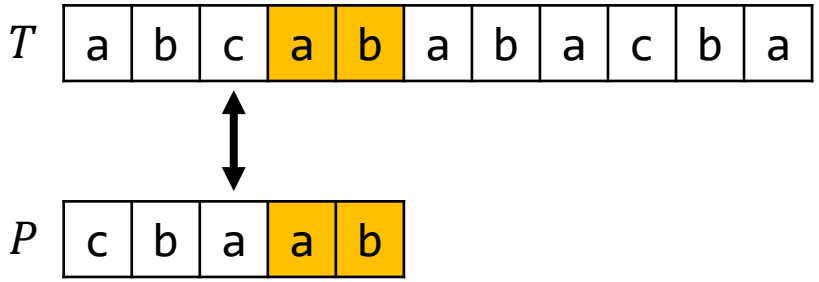
c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T



Good Suffix Rule

If there is a **bad character** β in T, then **shift**

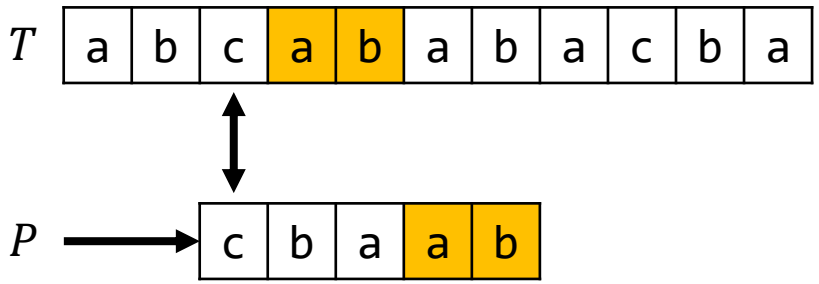
c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T



Good Suffix Rule

If there is a **bad character** β in T, then **shift**

c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T

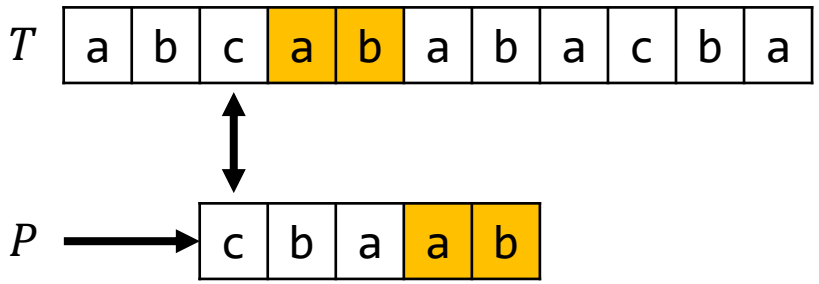


Suggested by BCR

Good Suffix Rule

If there is a **bad character** β in T, then **shift**

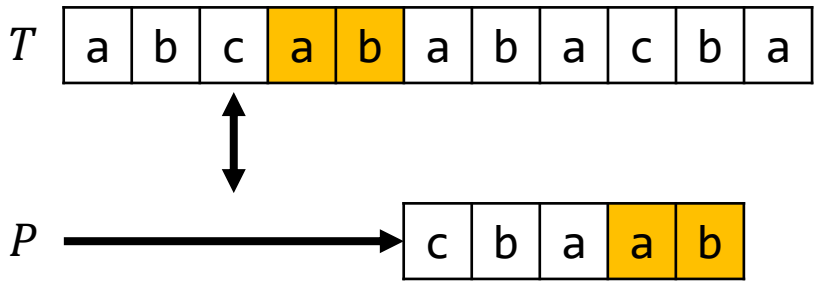
c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T, if no other occurrence of a suffix exist in P, then shift past the match



Good Suffix Rule

If there is a **bad character** β in T, then **shift**

c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T, if no other occurrence of a suffix exist in P, then shift past the match

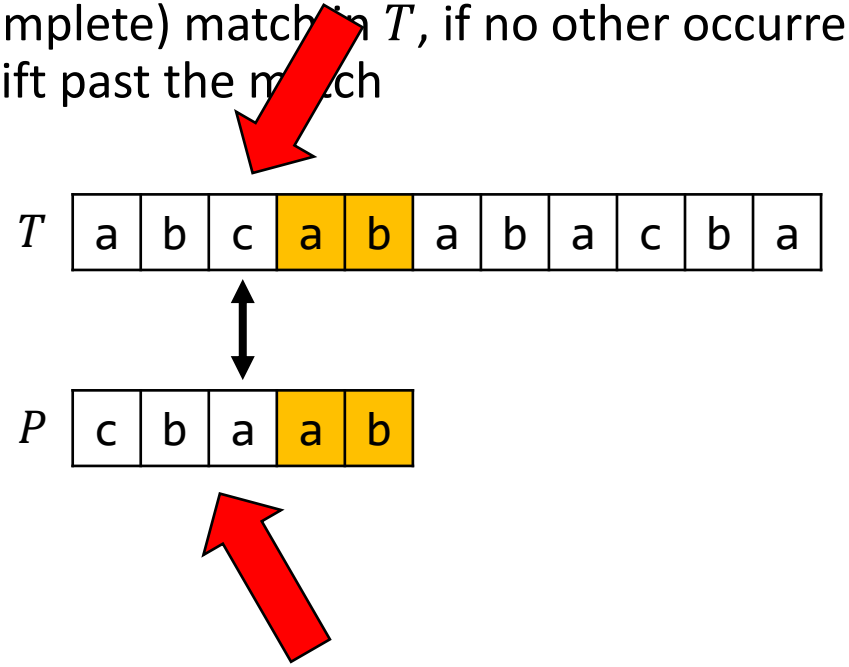


Suggested by GSR

Good Suffix Rule

If there is a **bad character** β in T, then **shift**

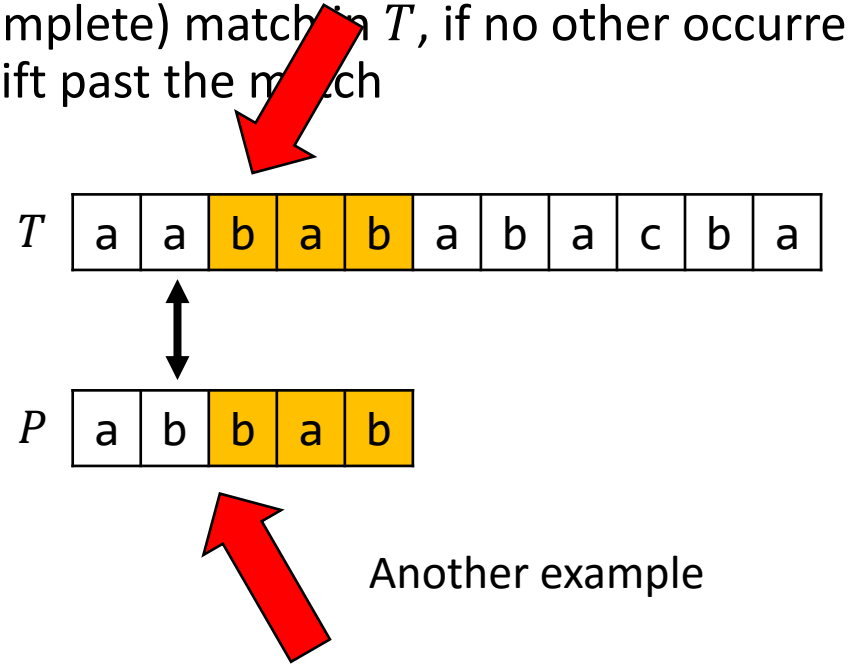
c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T, if no other occurrence of a suffix exist in P, then shift past the match



Good Suffix Rule

If there is a **bad character** β in T, then **shift**

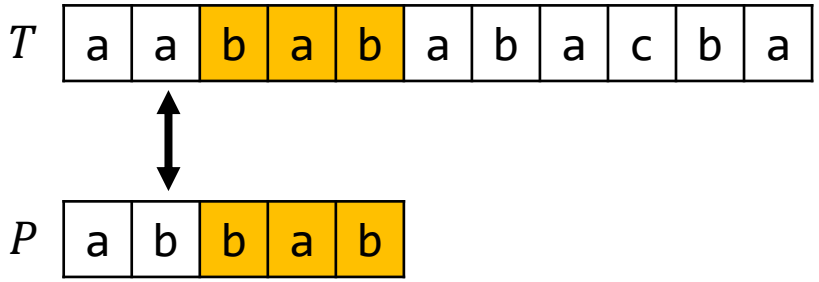
c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T, if no other occurrence of a suffix exist in P, then shift past the match



Good Suffix Rule

If there is a **bad character** β in T, then **shift**

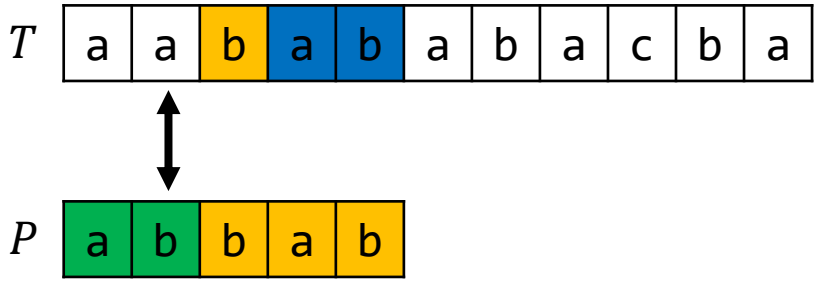
c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T, if no other occurrence of a suffix exist in P, then shift past the match



Good Suffix Rule

If there is a **bad character** β in T, then **shift**

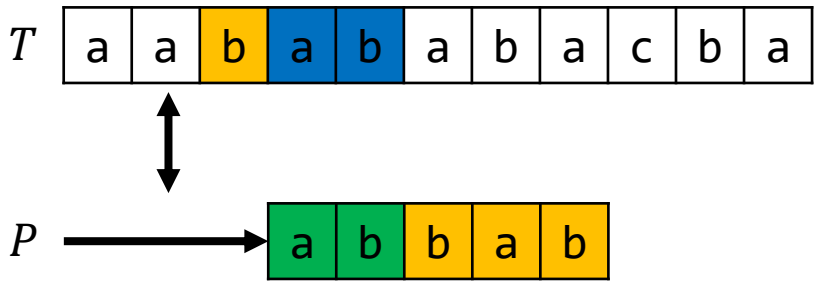
c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T, if no other occurrence of a suffix exist in P, then shift past the match



Good Suffix Rule

If there is a **bad character** β in T, then **shift**

c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T, if no other occurrence of a suffix exist in P, then shift past the match

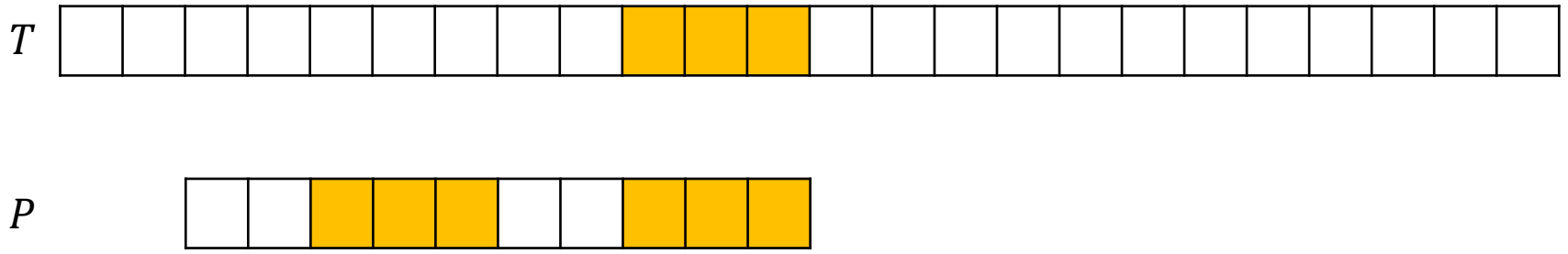


Suggested by GSR

Good Suffix Rule

Two cases to consider

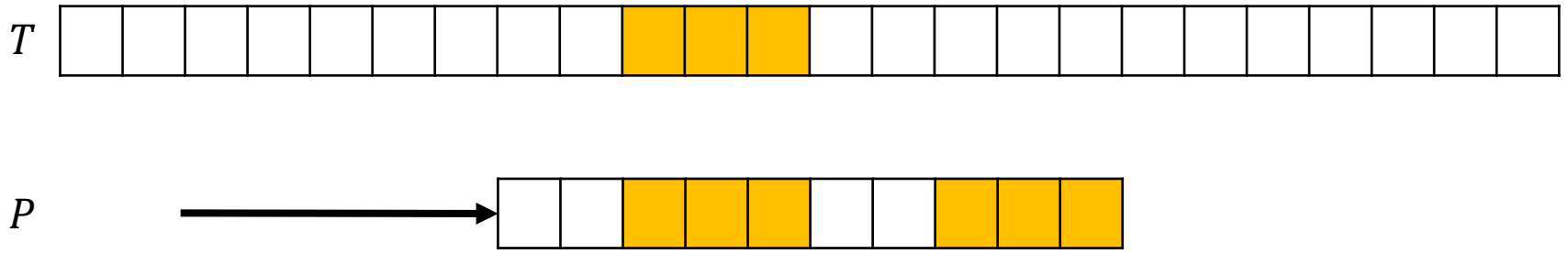
1. a) The current match occurs somewhere else in the pattern



Good Suffix Rule

Two cases to consider

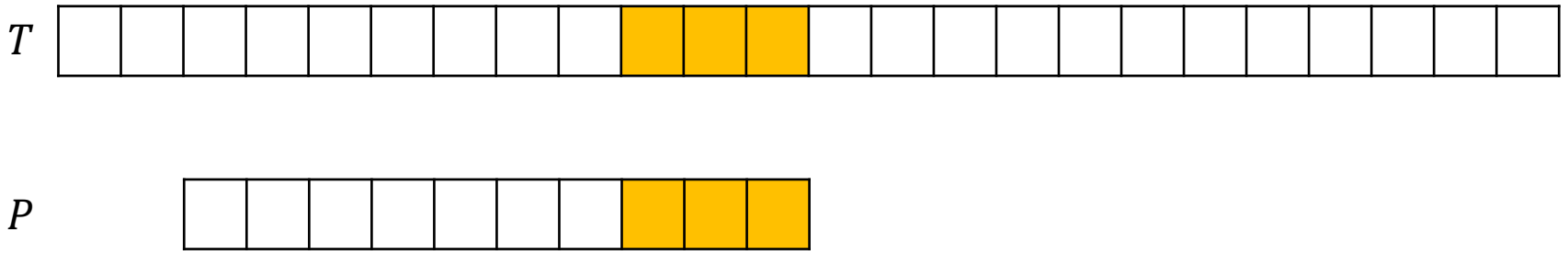
1. a) The current match occurs somewhere else in the pattern



Good Suffix Rule

Two cases to consider

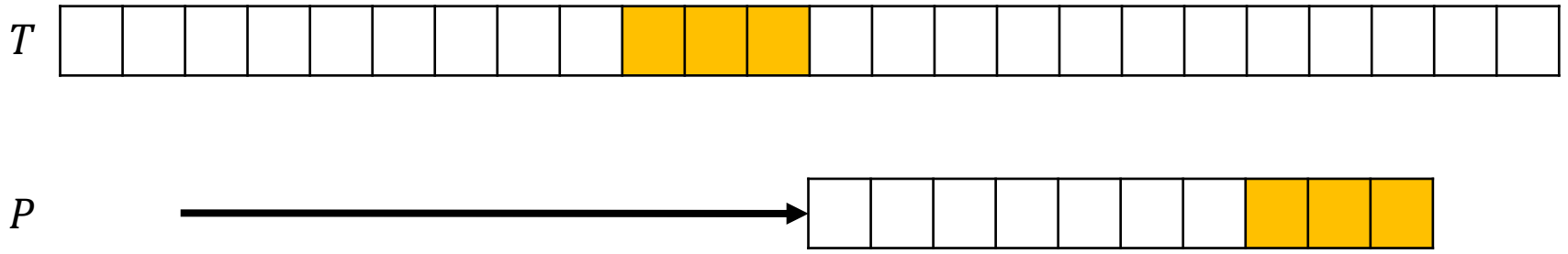
1. a) The current match occurs somewhere else in the pattern
   b) The current match occurs nowhere else in the pattern



Good Suffix Rule

Two cases to consider

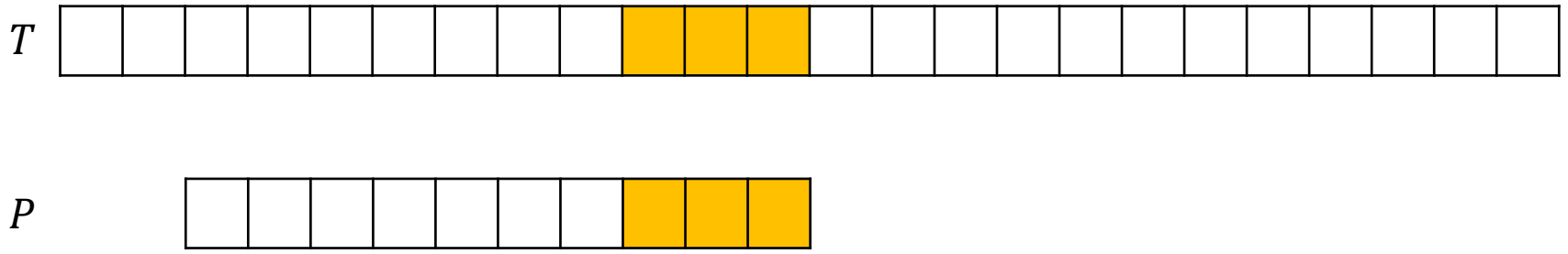
1. a) The current match occurs somewhere else in the pattern
   b) The current match occurs nowhere else in the pattern



Good Suffix Rule

Two cases to consider

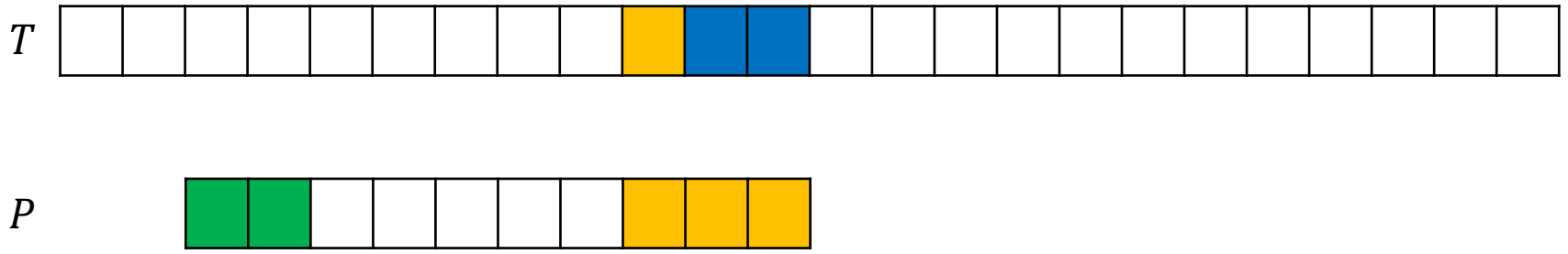
1. a) The current match occurs somewhere else in the pattern
   b) The current match occurs nowhere else in the pattern
2. A suffix of the match occurs at the beginning of the pattern



Good Suffix Rule

Two cases to consider

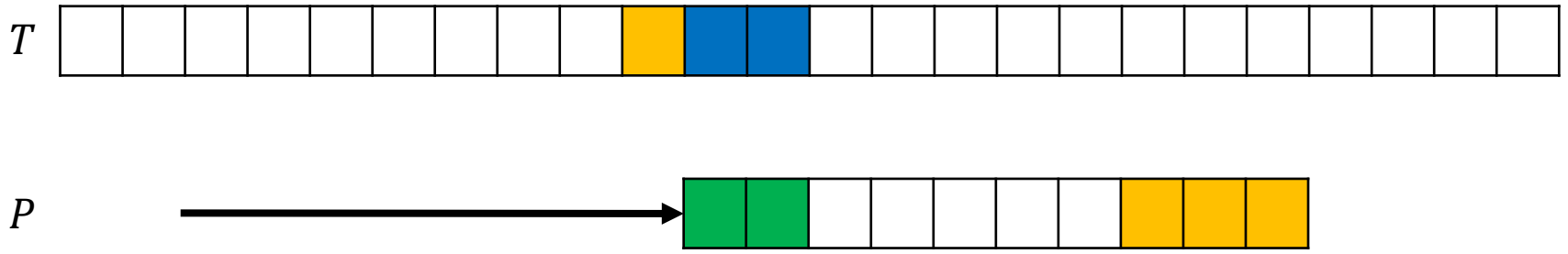
1. a) The current match occurs somewhere else in the pattern
   b) The current match occurs nowhere else in the pattern
2. A suffix of the match occurs at the beginning of the pattern



Good Suffix Rule

Two cases to consider

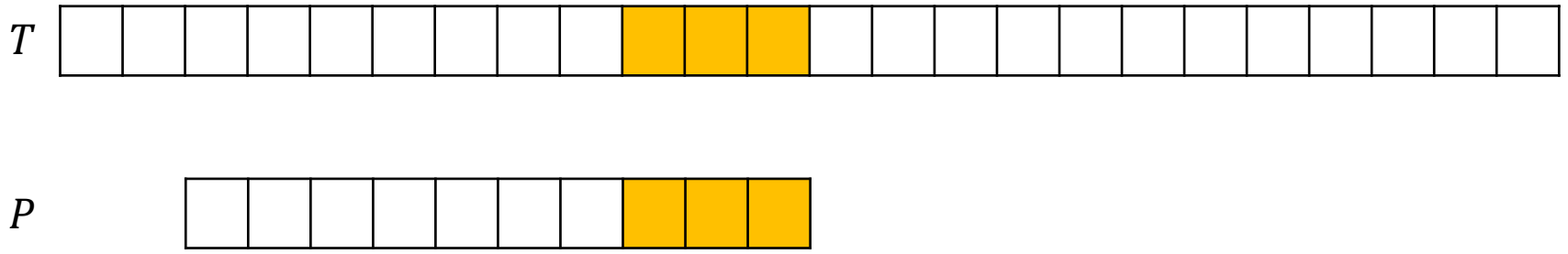
1. a) The current match occurs somewhere else in the pattern
   b) The current match occurs nowhere else in the pattern
2. A suffix of the match occurs at the beginning of the pattern



Good Suffix Rule

What about this case?

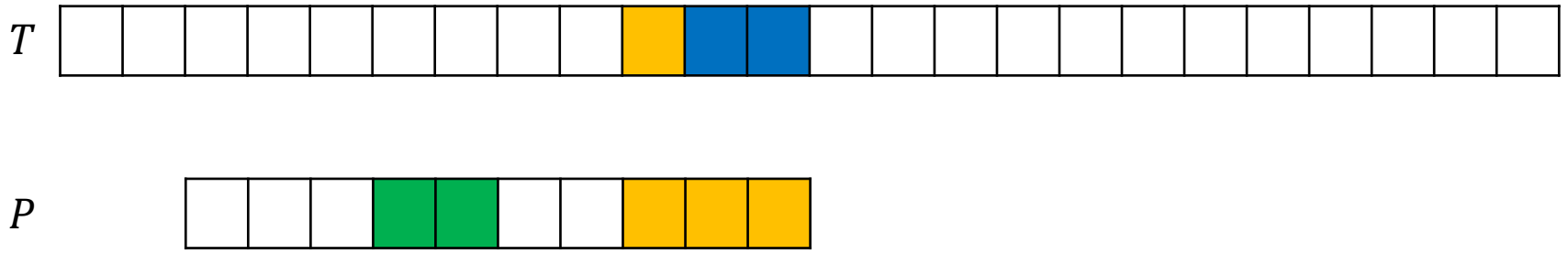
A suffix of the match occurs somewhere else in the pattern



Good Suffix Rule

What about this case?

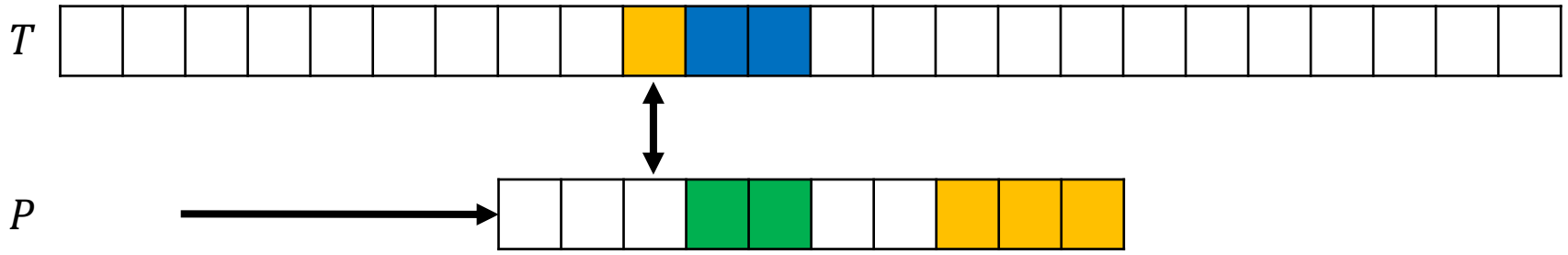
A suffix of the match occurs somewhere else in the pattern



Good Suffix Rule

What about this case?

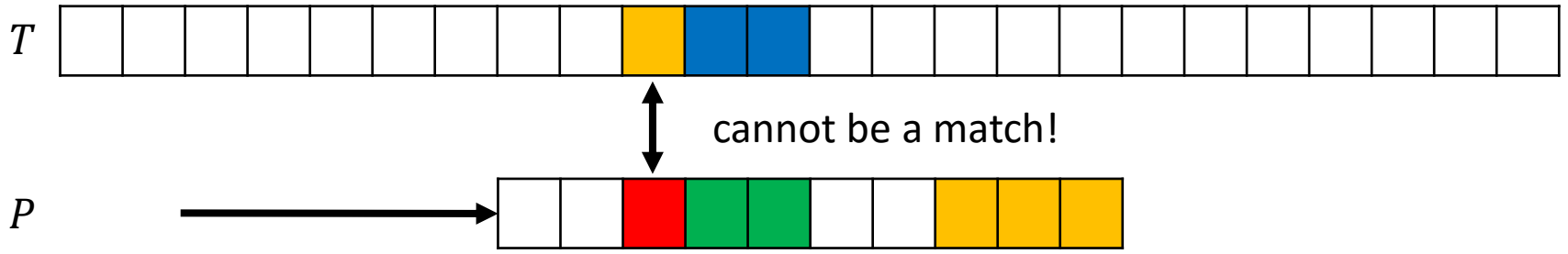
A suffix of the match occurs somewhere else in the pattern



Good Suffix Rule

What about this case?

A suffix of the match occurs somewhere else in the pattern:
Shift cannot be valid, otherwise it would correspond to case 1



Good Suffix Rule

If there is a **bad character** β in T, then **shift**

c) so that a **prefix** of P is **aligned** with a **suffix** of the current partial (or complete) match in T, if no other occurrence of a suffix exist in P, then shift past the match

```
GSR-TABLE(P)
 1: m = P.length
 2: let gsr[1..m] be a new array
 3: suffix = ε
 4: for i = m downto 1
 5:     gsr[i] = FIND-SUFFIX-SHIFT(P[i], suffix, P)
 6:     suffix = P[i] + suffix
 7: return gsr
```

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

i	1	2	3	4	5
$P[i]$	c	a	b	a	b
$gsr[i]$					

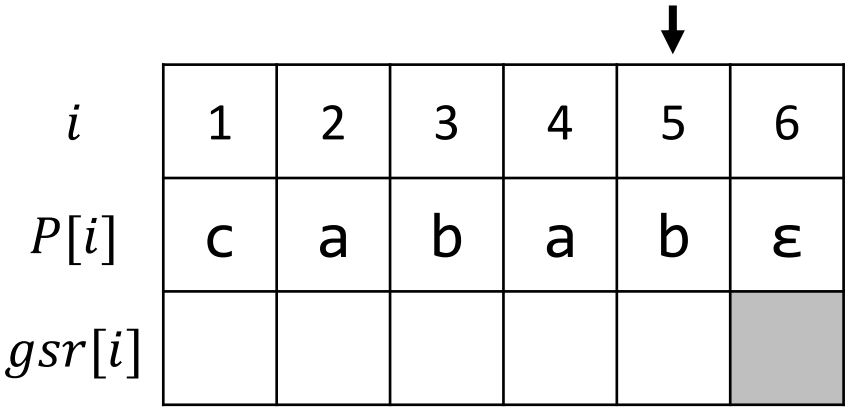
Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$						

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

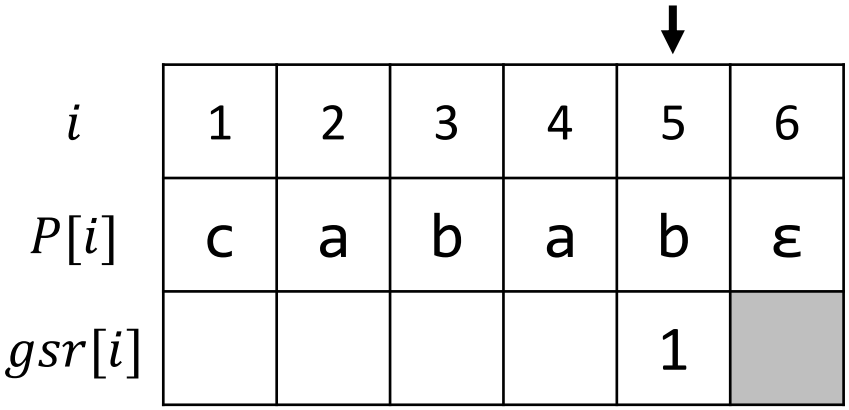


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$						

char = ‘b’ suffix = ε

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

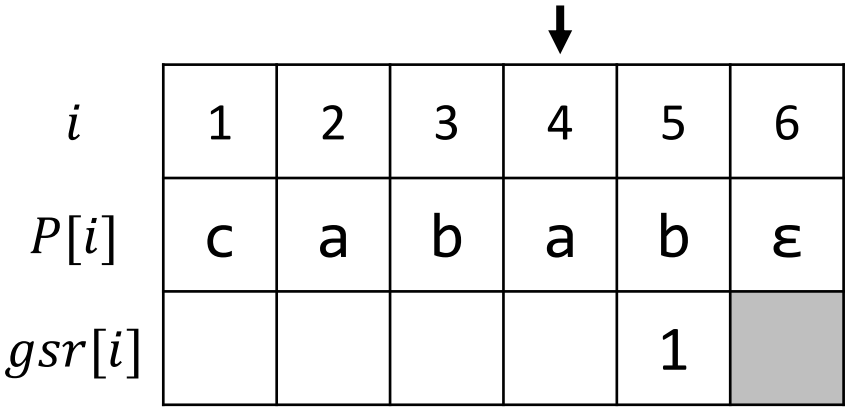


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$					1	

char = ‘b’ suffix = ε

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

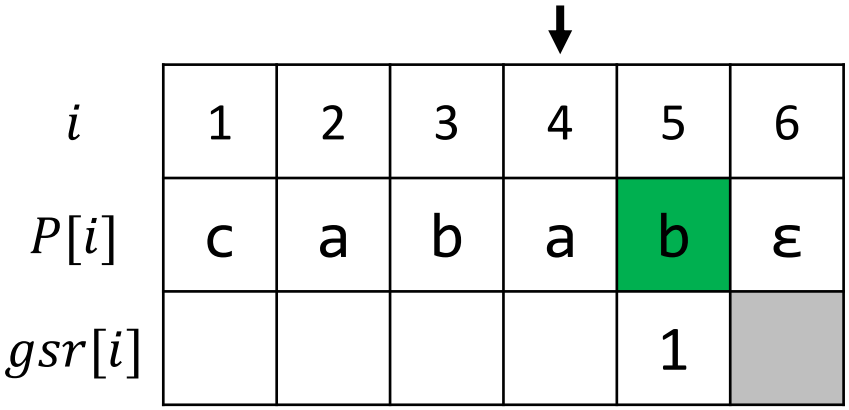


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$					1	

char = ‘a’ suffix = ‘b’

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

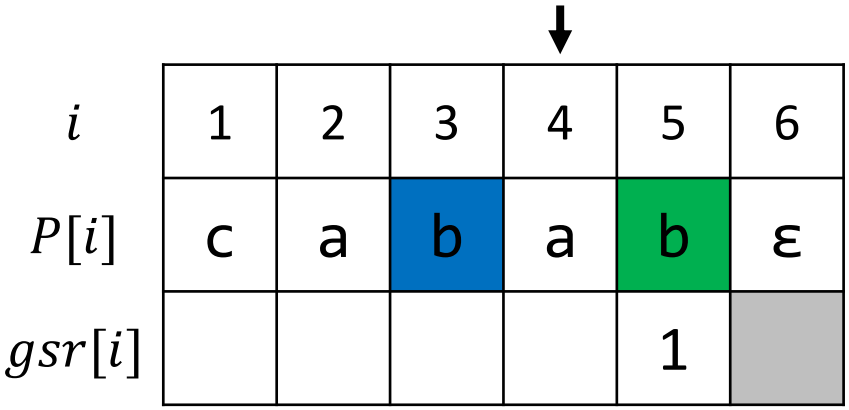


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$					1	

char = ‘a’ suffix = ‘b’

Good Suffix Rule

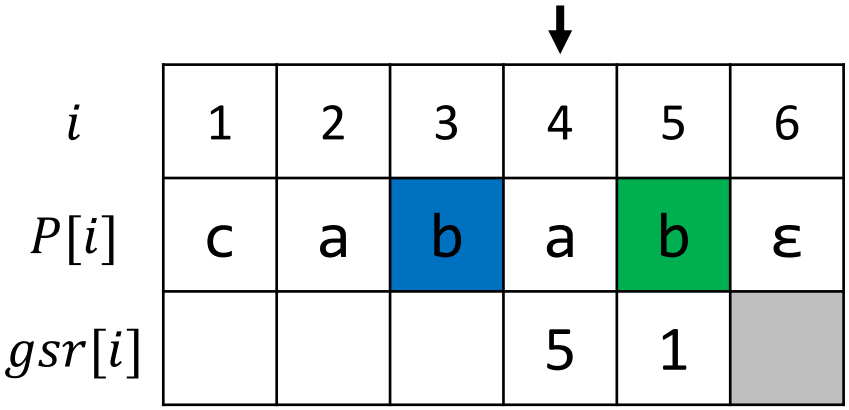
```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```



char = ‘a’ suffix = ‘b’

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

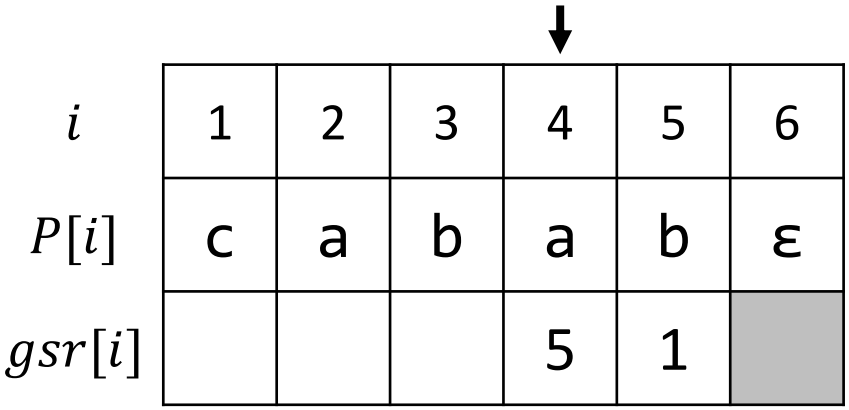


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$				5	1	

char = ‘a’ suffix = ‘b’

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

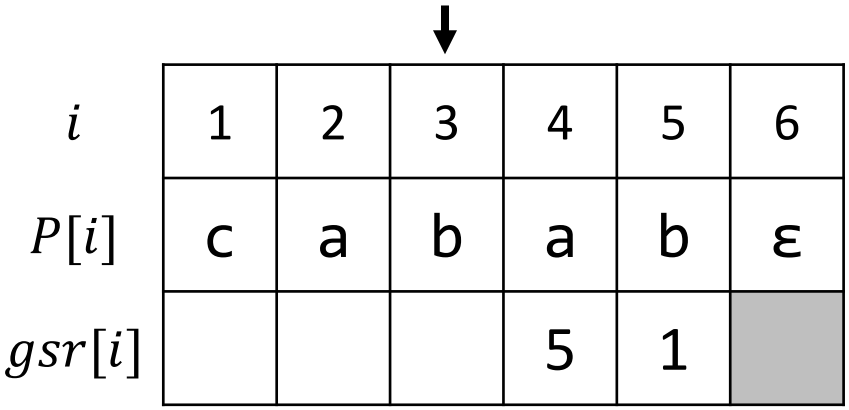


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$				5	1	

char = ‘a’ suffix = ‘b’

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

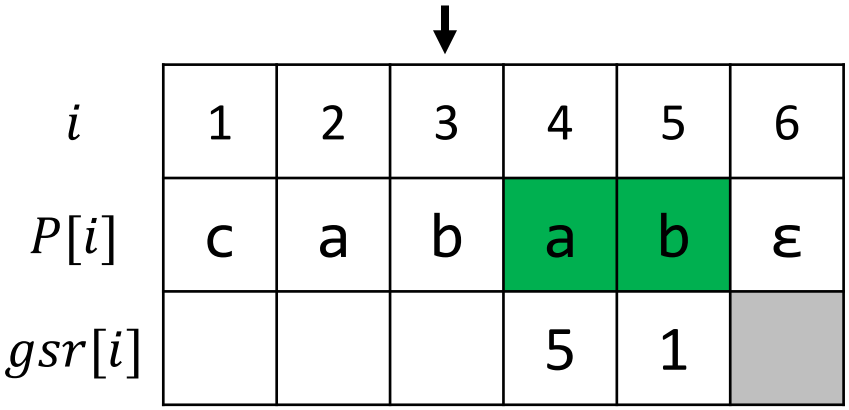


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$				5	1	

char = ‘b’ suffix = ‘ab’

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

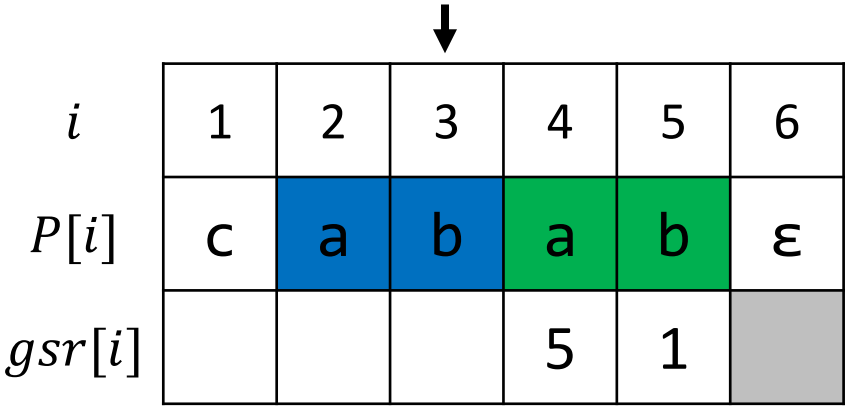


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$				5	1	

char = ‘b’ suffix = ‘ab’

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

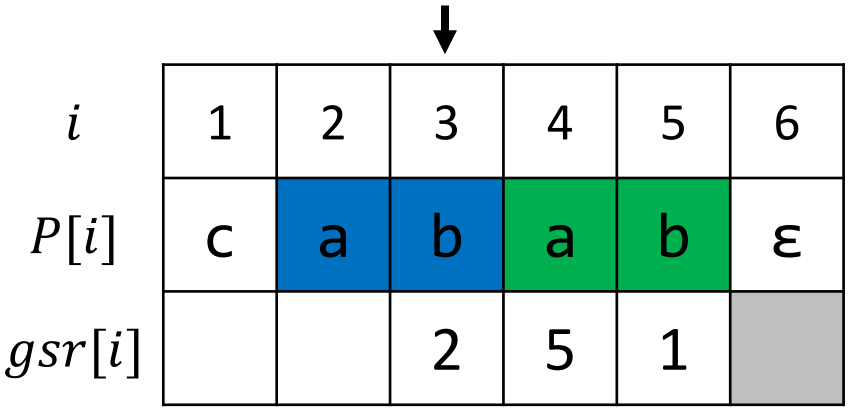


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$				5	1	

char = ‘b’ suffix = ‘ab’

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

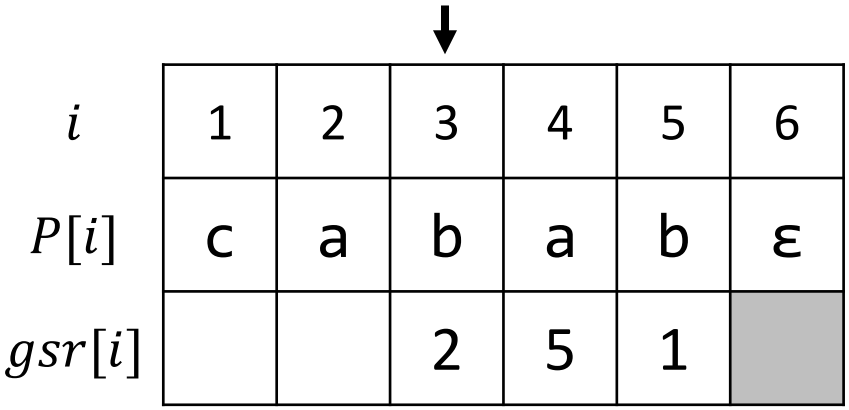


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$			2	5	1	

char = ‘b’ suffix = ‘ab’

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

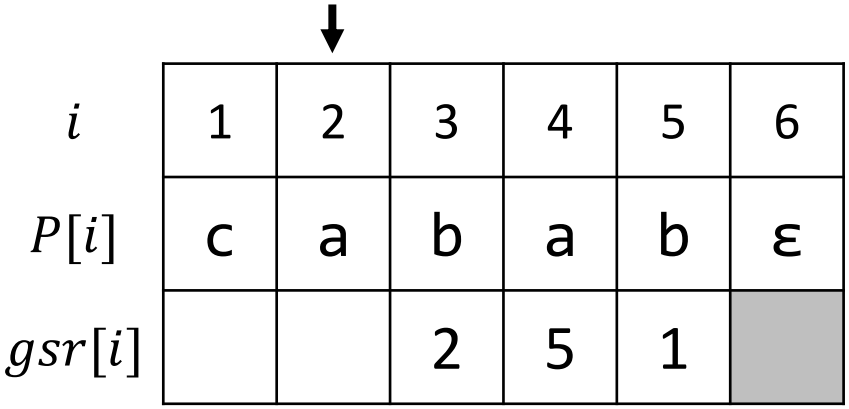


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$			2	5	1	

char = ‘b’ suffix = ‘ab’

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

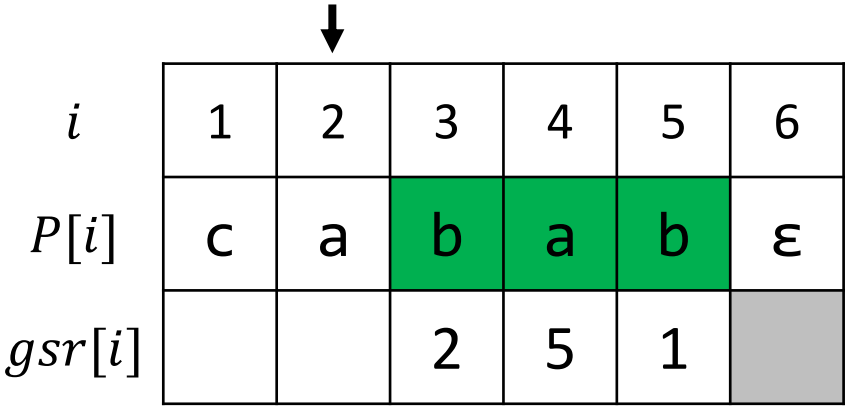


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$			2	5	1	

char = ‘a’ suffix = ‘bab’

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

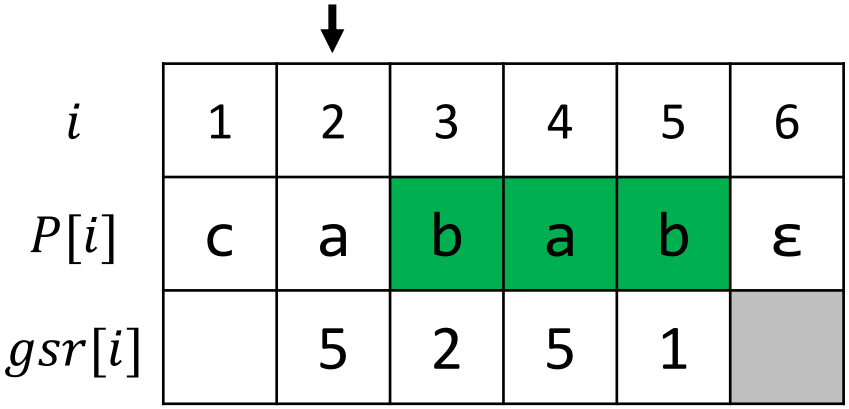


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$			2	5	1	

char = ‘a’ suffix = ‘bab’

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

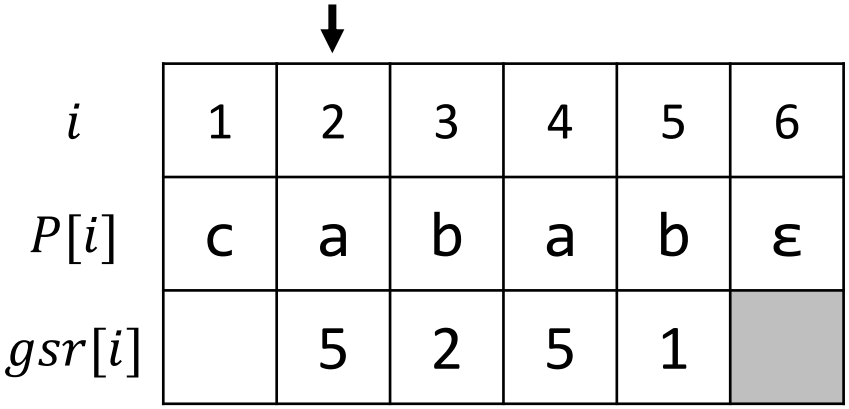


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$		5	2	5	1	

char = ‘a’ suffix = ‘bab’

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

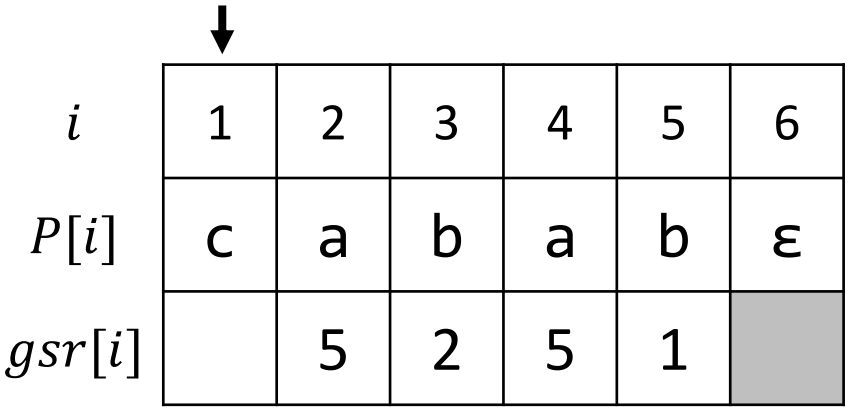


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$		5	2	5	1	

char = ‘a’ suffix = ‘bab’

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

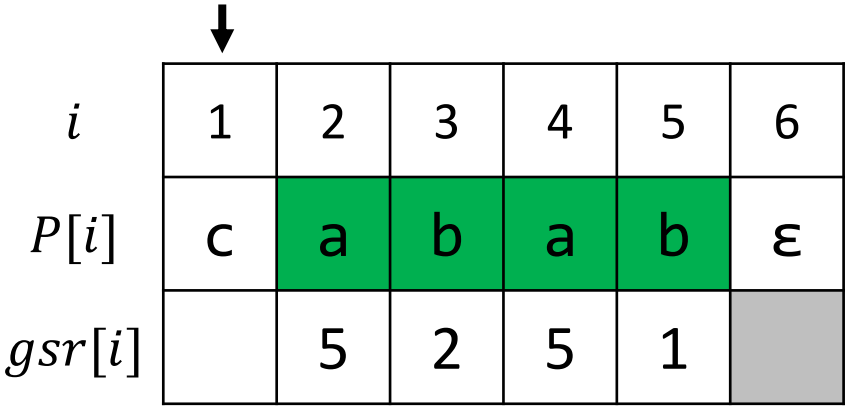


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$		5	2	5	1	

char = ‘c’ suffix = ‘abab’

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

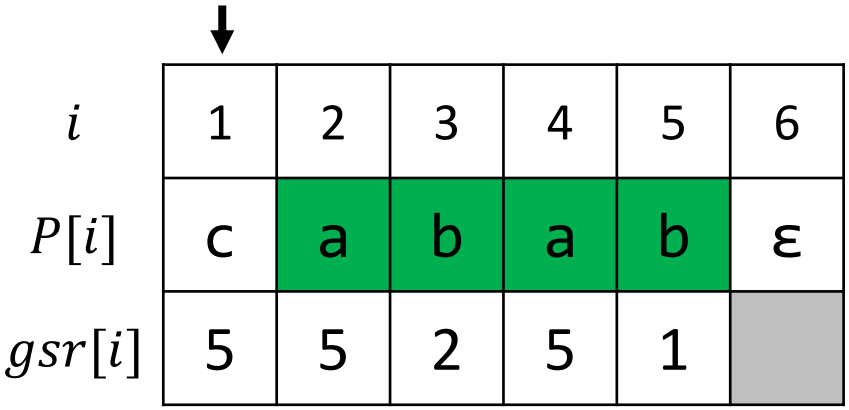


i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$		5	2	5	1	

char = ‘c’ suffix = ‘abab’

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```



i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$	5	5	2	5	1	

char = ‘c’ suffix = ‘abab’

Good Suffix Rule

```
FIND-SUFFIX-SHIFT(char, suffix, P)
 1: …
```

i	1	2	3	4	5	6
$P[i]$	c	a	b	a	b	ε
$gsr[i]$	5	5	2	5	1	

Boyer-Moore Matching

```
BM-MATCHER(T,P)
 1: n = T.length, m = P.length
 2: bcr = BCR-TABLE(P)
 3: gsr = GSR-TABLE(P)
 4: s = 0
 5: while s ≤ n-m
 6:     j = m
 7:     while j ≥ 1 and P[j] == T[s+j]
 8:         j = j-1
 9:     if j < 1
10:         PRINT(s)
11:         s = s + gsr[1]
12:     else
13:         s = s + MAX(gsr[j], j - bcr[T[s+j]])
```

Boyer-Moore with BCR Only?

```
BM-MATCHER-BCR(T,P)
 1: n = T.length, m = P.length
 2: bcr = BCR-TABLE(P)
 3: gsr = GSR-TABLE(P)
 4: s = 0
 5: while s ≤ n-m
 6:     j = m
 7:     while j ≥ 1 and P[j] == T[s+j]
 8:         j = j-1
 9:     if j < 1
10:         PRINT(s)
11:         s = s + gsr[1]
12:     else
13:         s = s + MAX(gsr[j], j - bcr[T[s+j]])
```

Boyer-Moore with BCR Only

```
BM-MATCHER-BCR(T,P)
 1: n = T.length, m = P.length
 2: bcr = BCR-TABLE(P)
 3: gsr = GSR-TABLE(P)
 4: s = 0
 5: while s ≤ n-m
 6:     j = m
 7:     while j ≥ 1 and P[j] == T[s+j]
 8:         j = j-1
 9:     if j < 1
10:         PRINT(s)
11:         s = s + 1
12:     else
13:         s = s + MAX(1, j - bcr[T[s+j]])
```

Running Time of Boyer-Moore

Algorithm	Preprocessing	Matching
Naive	0	$O(nm)$
Knuth-Morris-Pratt	$O(m)$	$O(n)$
Boyer-Moore		

Running Time of Boyer-Moore

Algorithm	Preprocessing	Matching
Naive	0	$O(nm)$
Knuth-Morris-Pratt	$O(m)$	$O(n)$
Boyer-Moore	$O(m)$	

Running Time of Boyer-Moore

Algorithm	Preprocessing	Matching
Naive	0	$O(nm)$
Knuth-Morris-Pratt	$O(m)$	$O(n)$
Boyer-Moore	$O(m)$	$O(nm)$

In practice, Boyer-Moore is often faster than Knuth-Morris-Pratt.

In addition, there exists a modification, the Galil* rule, that guarantees linear running time in worst-case.

*Zvi Galil

Knuth-Morris-Pratt vs Boyer-Moore

ROUND 1

Knuth-Morris-Pratt vs Boyer-Moore

T	h	e	r	e	␣	i	s	␣	a	␣	s	i	m	p	l	e	␣	e	x	a	m	p	l	e

KMP

P	e	x	a	m	p	l	e

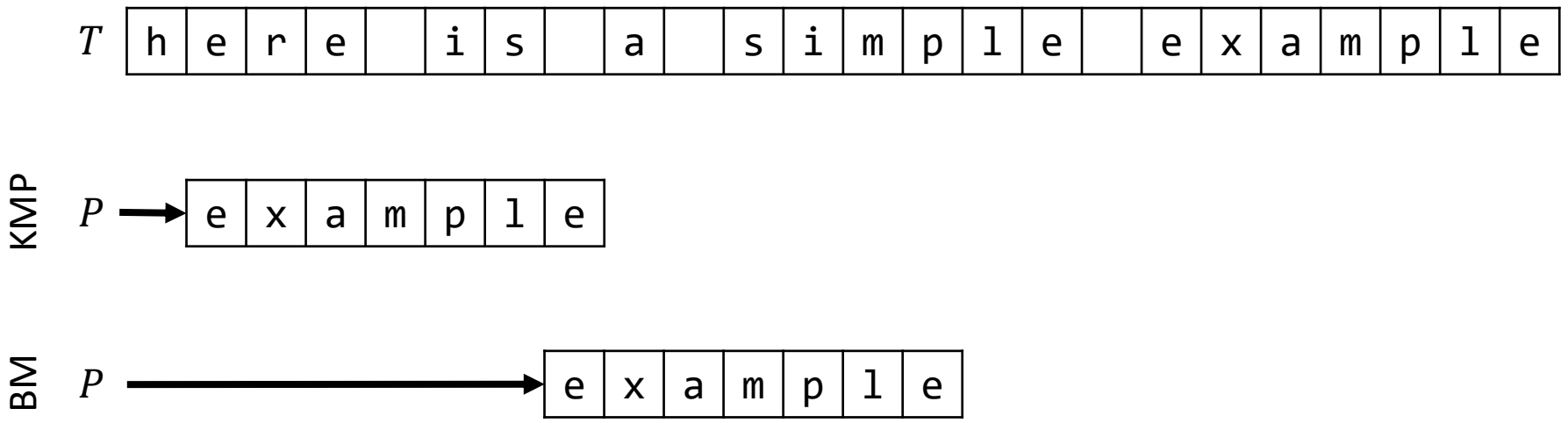
BM

P	e	x	a	m	p	l	e

i	1	2	3	4	5	6	7
$P[i]$	e	x	a	m	p	l	e
$\pi[i]$	0	0	0	0	0	0	1
$gsr[i]$	6	6	6	6	6	6	1

c	a	e	l	m	p	x
$bcr[c]$	3	7	6	4	5	2

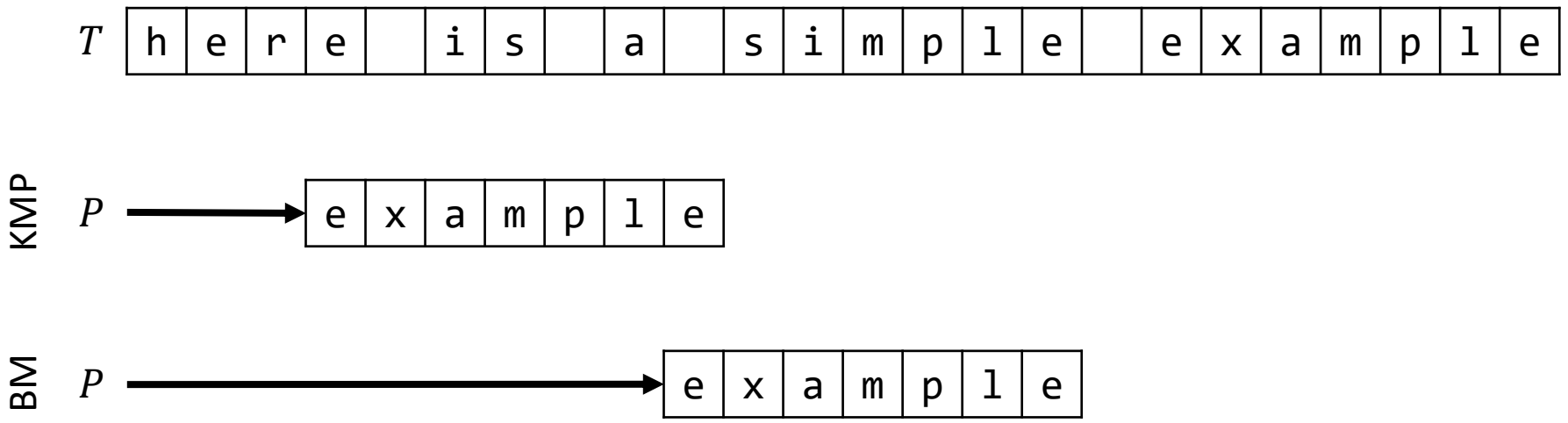
Knuth-Morris-Pratt vs Boyer-Moore



i	1	2	3	4	5	6	7
$P[i]$	e	x	a	m	p	l	e
$\pi[i]$	0	0	0	0	0	0	1
$gsr[i]$	6	6	6	6	6	6	1

c	a	e	l	m	p	x
$bcr[c]$	3	7	6	4	5	2

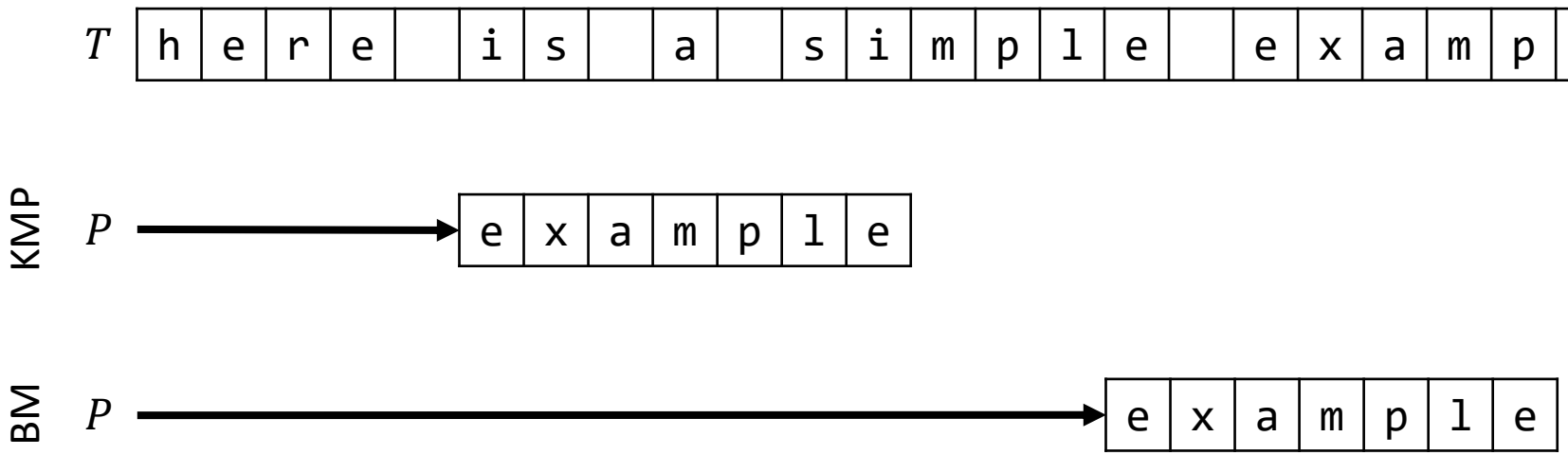
Knuth-Morris-Pratt vs Boyer-Moore



i	1	2	3	4	5	6	7
$P[i]$	e	x	a	m	p	l	e
$\pi[i]$	0	0	0	0	0	0	1
$gsr[i]$	6	6	6	6	6	6	1

c	a	e	l	m	p	x
$bcr[c]$	3	7	6	4	5	2

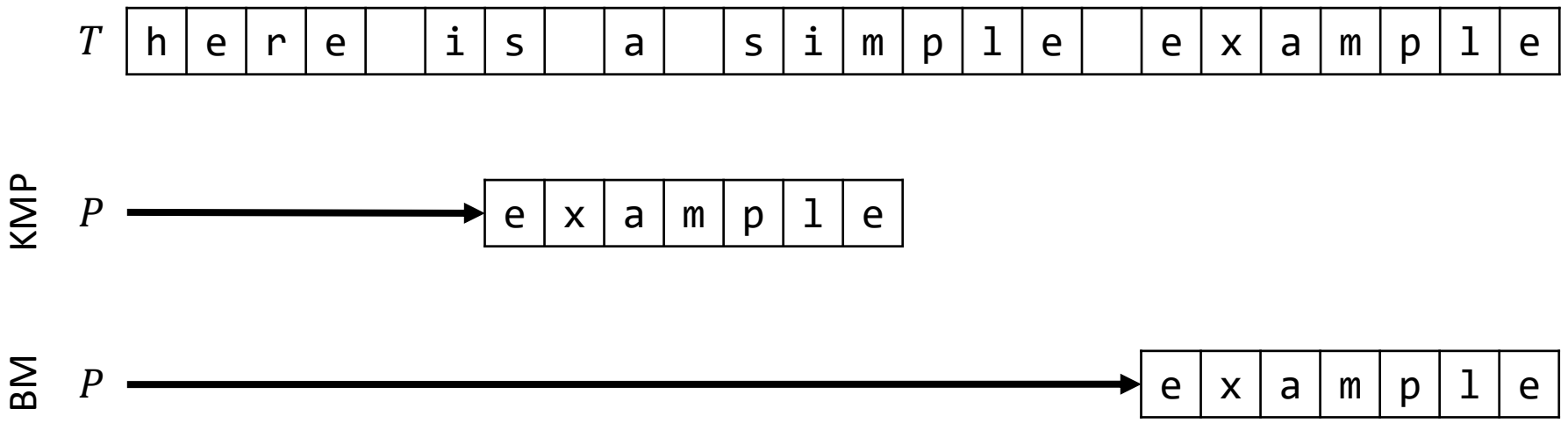
Knuth-Morris-Pratt vs Boyer-Moore



i	1	2	3	4	5	6	7
$P[i]$	e	x	a	m	p	l	e
$\pi[i]$	0	0	0	0	0	0	1
$gsr[i]$	6	6	6	6	6	6	1

c	a	e	l	m	p	x
$bcr[c]$	3	7	6	4	5	2

Knuth-Morris-Pratt vs Boyer-Moore



i	1	2	3	4	5	6	7
$P[i]$	e	x	a	m	p	l	e
$\pi[i]$	0	0	0	0	0	0	1
$gsr[i]$	6	6	6	6	6	6	1

c	a	e	l	m	p	x
$bcr[c]$	3	7	6	4	5	2

Knuth-Morris-Pratt vs Boyer-Moore

ROUND 2

Knuth-Morris-Pratt vs Boyer-Moore

T	w	o	o	d		w	o	u	l	d		a		w	o	o	d	c	h	u	c	k

KMP

P	w	o	o	d

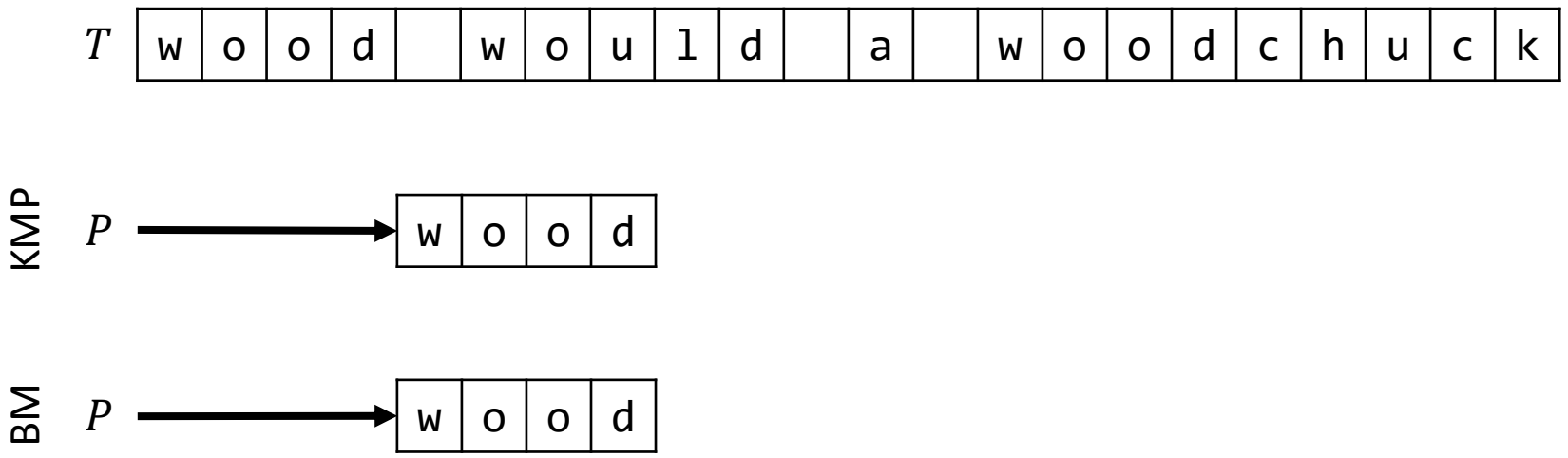
BM

P	w	o	o	d

i	1	2	3	4
$P[i]$	w	o	o	d
$\pi[i]$	0	0	0	0
$gsr[i]$	4	4	4	1

c	d	o	w
$bcr[c]$	4	3	1

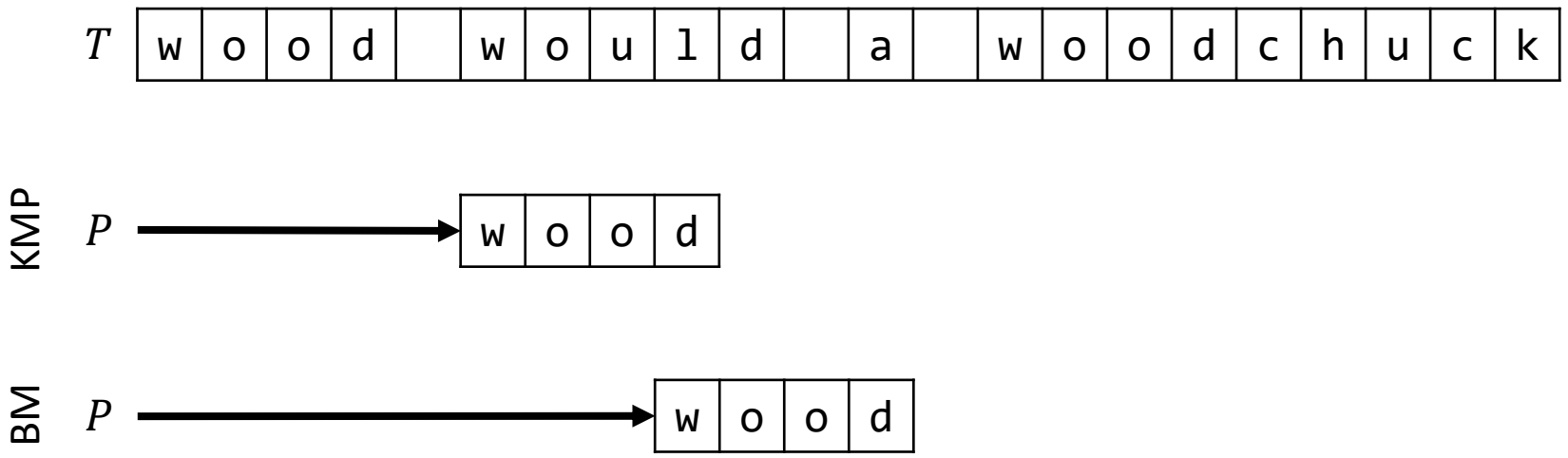
Knuth-Morris-Pratt vs Boyer-Moore



i	1	2	3	4
$P[i]$	w	o	o	d
$\pi[i]$	0	0	0	0
$gsr[i]$	4	4	4	1

c	d	o	w
$bcr[c]$	4	3	1

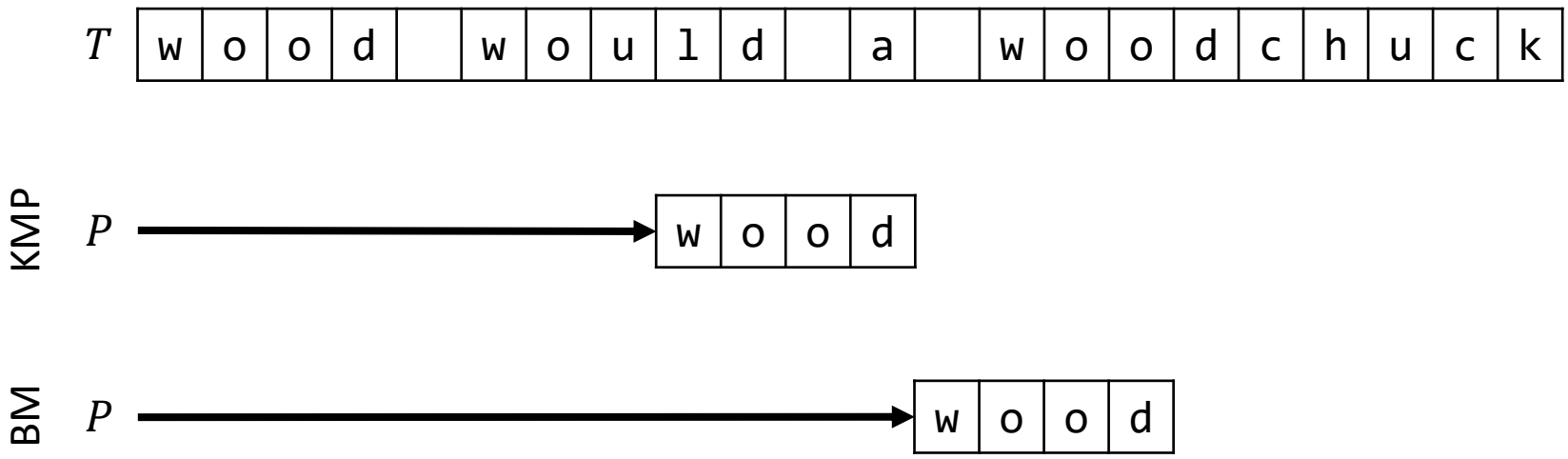
Knuth-Morris-Pratt vs Boyer-Moore



i	1	2	3	4
$P[i]$	w	o	o	d
$\pi[i]$	0	0	0	0
$gsr[i]$	4	4	4	1

c	d	o	w
$bcr[c]$	4	3	1

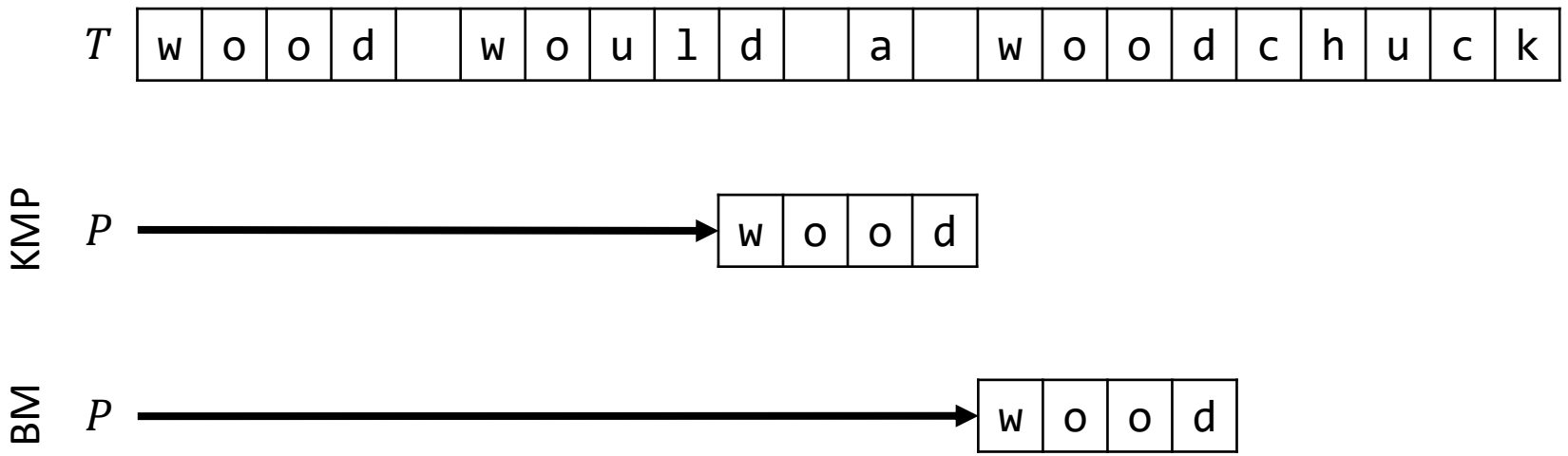
Knuth-Morris-Pratt vs Boyer-Moore



i	1	2	3	4
$P[i]$	w	o	o	d
$\pi[i]$	0	0	0	0
$gsr[i]$	4	4	4	1

c	d	o	w
$bcr[c]$	4	3	1

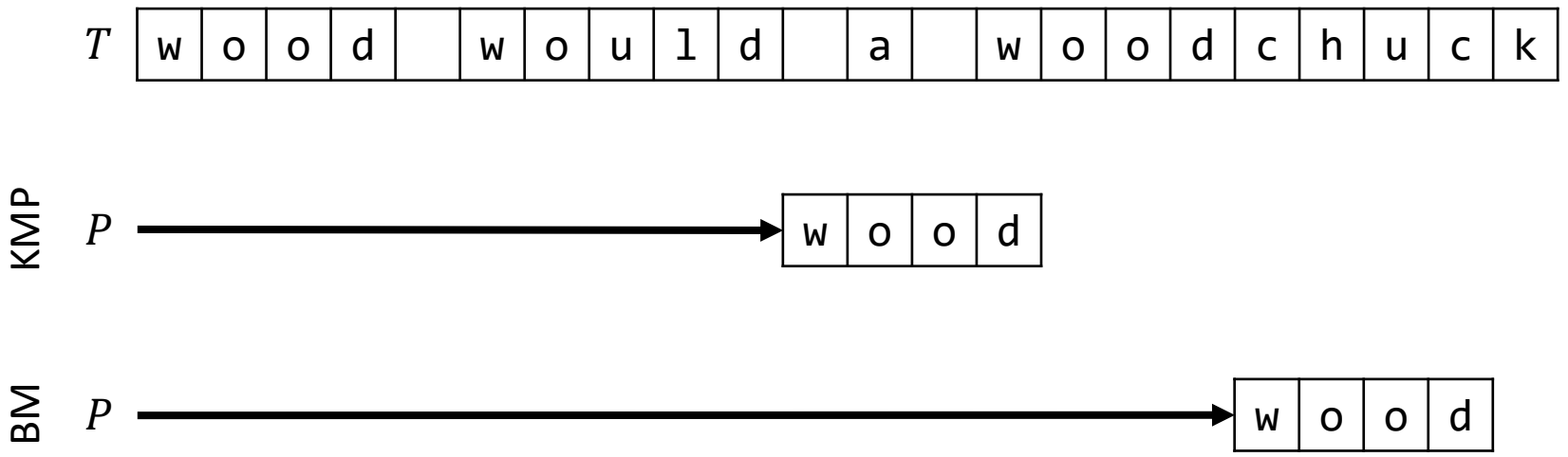
Knuth-Morris-Pratt vs Boyer-Moore



i	1	2	3	4
$P[i]$	w	o	o	d
$\pi[i]$	0	0	0	0
$gsr[i]$	4	4	4	1

c	d	o	w
$bcr[c]$	4	3	1

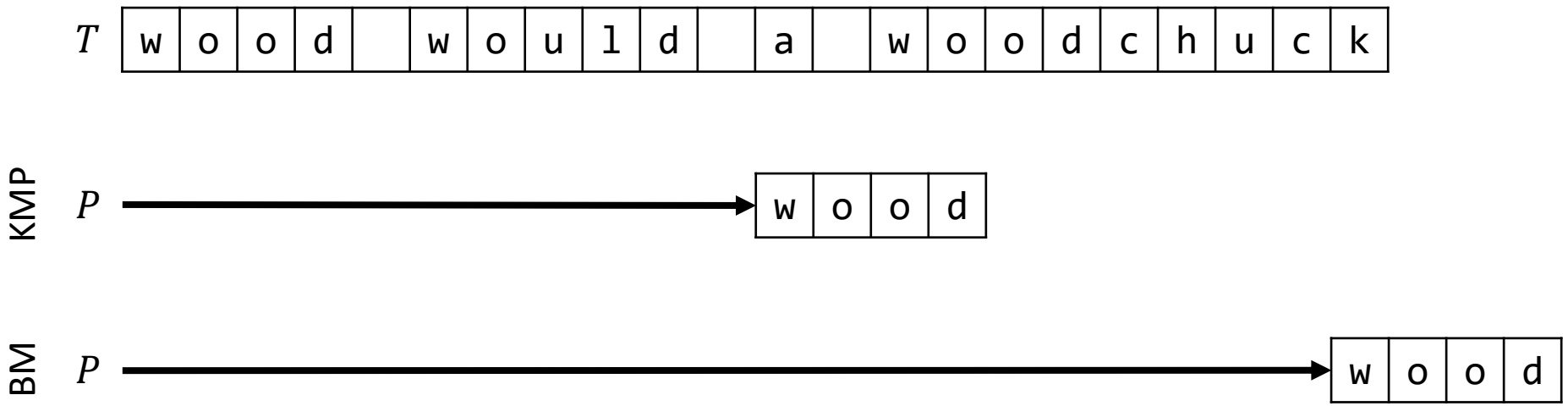
Knuth-Morris-Pratt vs Boyer-Moore



i	1	2	3	4
$P[i]$	w	o	o	d
$\pi[i]$	0	0	0	0
$gsr[i]$	4	4	4	1

c	d	o	w
$bcr[c]$	4	3	1

Knuth-Morris-Pratt vs Boyer-Moore



i	1	2	3	4
$P[i]$	w	o	o	d
$\pi[i]$	0	0	0	0
$gsr[i]$	4	4	4	1

c	d	o	w
$bcr[c]$	4	3	1

Conclusion

Does this mean Boyer-Moore is always faster than KMP?

- No. And we will see this in our empirical analysis.
- Also, we have ignored the work that is done during an iteration, and this is sometimes much less for KMP.

In practice, it depends on the **nature of the input** (in particular, on the **length of the pattern**) which algorithm performance favourably.

References

Books

- [Cormen] **Introduction to Algorithms**
  T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. MIT Press. 2009 (3rd Edition)
- [Sedgewick] **Algorithms**
  R. Sedgewick, K. Wayne. Addison-Wesley. 2011 (4th Edition)
- [Dasgupta] **Algorithms**
  S. Dasgupta, C. Papadimitriou, U. Vazirani. McGraw-Hill Higher Education. 2006

Online

- http://algs4.cs.princeton.edu/lectures/
- https://www.coursera.org/courses?query=algorithms