

◀Function Prefix▶ + ◀KMP▶

Kaio Christaldo
Fabricio Matsunaga

◀ **Function Prefix** ▶

Apresentação Problema Motivador

Maior Prefixo Repetido

Dada uma string `s`, determine o comprimento do **maior prefixo próprio** que aparece **novamente em outra posição da string** (ou seja, não apenas no início).

Um **prefixo próprio** é qualquer prefixo da string que **não seja igual à string completa**.

Seu objetivo é encontrar o **maior prefixo próprio** que também aparece em **outra posição da string** (pode estar no meio ou no final, não precisa ser sufixo).

Esse problema pode ser resolvido de forma eficiente usando a **função prefixo** (também conhecida como vetor `pi`) do algoritmo de Knuth-Morris-Pratt (KMP).

Entrada

- Uma única linha contendo a string `s` ($1 \leq |s| \leq 10^6$).
- A string contém apenas letras do alfabeto inglês (maiúsculas e/ou minúsculas).

Saída

- Um único número inteiro: o comprimento do **maior prefixo próprio** que também aparece em outra posição da string.

Exemplos

Entrada	Saída
abcdabcc	3
abcdefg	0
cabcabcd	4

Algorithm <function prefix>

- a função prefixo é uma técnica usada para comparar strings de forma eficiente. Ela calcula, para cada posição de uma string, o tamanho do maior prefixo próprio que também é um sufixo daquela parte da string.
- Dada uma string, a função prefixo gera um vetor (array) onde cada posição i guarda o comprimento do maior prefixo que também é sufixo da substring que vai do começo até a posição i .
 - Esse vetor é importante porque:
 - Ele mostra repetições dentro da string.
 - É usado pelo algoritmo de Knuth–Morris–Pratt (KMP) para buscar padrões (palavras) dentro de um texto de forma rápida (em tempo linear, ou seja, muito eficiente).

Algorithm <function prefix>

Prefixo de uma palavra ou string é uma parte inicial dela.

- **Exemplo:** Para a palavra "computador", os prefixos são:
 - "", "c", "co", "com", "comp", "compu", "comput", "computa", "computad", "computado", "computador"
- (Note que o prefixo pode ser a string toda ou até vazia.)

Sufixo é uma parte final da palavra ou string.

- **Exemplo:** Para a palavra "computador", os sufixos são:
 - "", "r", "or", "dor", "ador", "tador", "utador", "putador", "mputador", "omputador", "computador"

OBS: Quando falamos de **prefixo próprio**, significa um prefixo que não é a palavra inteira. Ou seja, para "computador", prefixos próprios são todos os prefixos exceto "computador" completo.

Algorithm <function prefix>

Exemplo prático:

- Suponha a string "aaba"aa"
- Queremos montar o vetor da função prefixo:

indice	substring	sufixo	tamanho
0	a	""	0
1	aa	"a"	1
2	aab	""	0
3	aaba	"a"	1
4	aabaa	"aa"	2

Algorithm <function prefix>

Exemplo prático:

- Vetor da função prefixo:
- $[0, 1, 0, 1, 2]$

Explicando:

- Posição 0: só tem a letra "a", não há prefixo próprio $\rightarrow 0$
- Posição 1: "aa" \rightarrow prefixo "a", sufixo "a" \rightarrow tamanho 1
- Posição 2: "aab" \rightarrow não tem prefixo = sufixo $\rightarrow 0$
- Posição 3: "aaba" \rightarrow prefixo "a", sufixo "a" \rightarrow tamanho 1
- Posição 4: "aabaa" \rightarrow prefixo "aa", sufixo "aa" \rightarrow tamanho 2

Algorithm <function prefix>

Implementação em potl e C++ – $O(n)$:

```
1 função prefixa(s: Texto): Lista
2   n <- tamanho(s)
3   pi <- [0] * n
4
5   para i de 1 até n - 1
6     j <- pi[i - 1]
7
8     enquanto j > 0 e s[i] != s[j]
9       j <- pi[j - 1]
10
11     se s[i] == s[j]
12       j <- j + 1
13
14     pi[i] <- j
15
16   retorna pi
17 fim
18
19 início
20   palavra <- "ababc"
21   resultado <- prefixa(palavra)
22   escreva("Prefixa de ", palavra, ": ", resultado)
23 fim
24
```

```
1 vector<int> prefix_function(string s) {
2   int n = s.size();
3
4   vector<int> pi(n);
5
6   for (int i = 1; i < n; i++) {
7     int j = pi[i - 1];
8
9     while (j > 0 && s[i] != s[j])
10       j = pi[j - 1];
11
12     if (s[i] == s[j])
13       j++;
14
15     pi[i] = j;
16   }
17
18   return pi;
19 }
20
```


Resolução do Problema Motivador


Maior prefixo repetido

A resolução estará disponível no Drive. Tente resolver por conta própria e, se precisar, compare com a solução! 😊

Apresentação Problema Motivador

beecrowd | 1237

Comparação de Substring

Por TopCoder*  EUA

Timelimit: 1

Encontre a maior substring comum entre as duas strings informadas. A substring pode ser qualquer parte da string, inclusive ela toda. Se não houver subsequência comum, a saída deve ser "0". A comparação é *case sensitive* ('x' != 'X').

Entrada

A entrada contém vários casos de teste. Cada caso de teste é composto por duas linhas, cada uma contendo uma string. Ambas strings de entrada contém entre 1 e 50 caracteres ('A'-'Z','a'-'z' ou espaço ' '), inclusive, ou no mínimo uma letra ('A'-'Z','a'-'z').

Saída

O tamanho da maior subsequência comum entre as duas Strings.

Exemplo de Entrada	Exemplo de Saída
abcdef	2
cdofhij	1
TWO	0
FOUR	7
abracadabra	
open	
Hey This java is hot	
Java is a new paradigm	

* Este problema é de autoria do TopCoder (www.topcoder.com/tc) e foi adaptado por Alessandro B. para utilização (autorizada) no URI OJ.

* A reprodução não autorizada deste problema sem o consentimento por escrito de TopCoder, Inc. é estritamente proibida.

1237 – Comparação de Substring

◀KMP Algorithm▶

◀KMP Algorithm▶

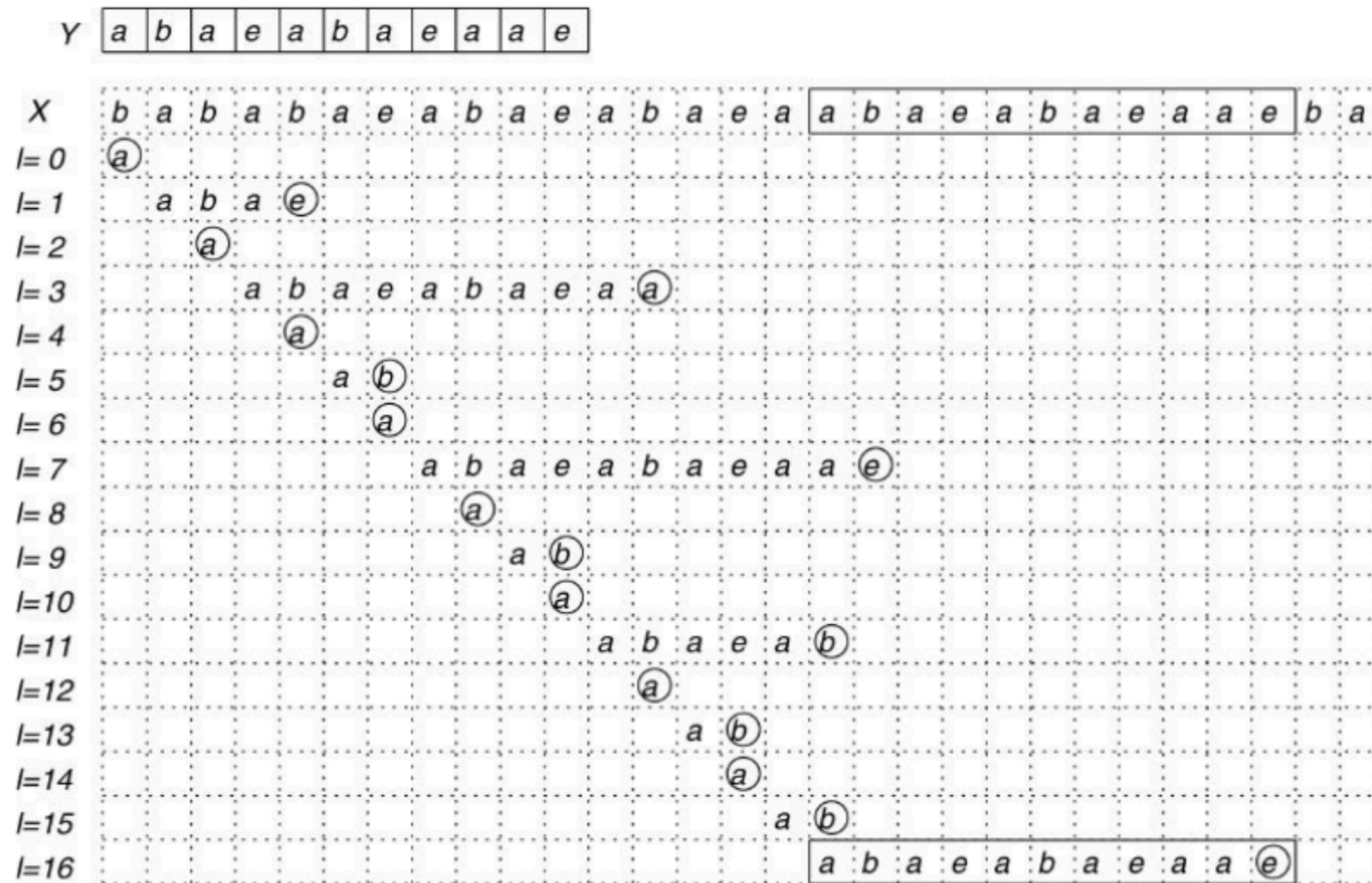
O algoritmo Knuth–Morris–Pratt (KMP) é um algoritmo de casamento de cadeia em tempo linear que eficientemente encontra ocorrências de um padrão dentro de um texto.

◀ Ideia básica do algoritmo ▶

Para descrever o algoritmo de Knuth, Morris e Pratt para o problema de casamento de cadeias deve-se, inicialmente, revisar o algoritmo de força bruta. São dadas as cadeias X e Y , com caracteres $x[i]$ e $y[j]$, $1 \leq i \leq n$ e $1 \leq j \leq m$, $m \leq n$, respectivamente. O algoritmo, para cada $L = 0, 1, \dots, n - m$, verifica se a subcadeia $X[L] + 1$ de X , de comprimento m e iniciada em $x[L] + 1$, é idêntica a Y . Se for idêntica, há casamento na posição $L + 1$. Observa-se, no entanto, que em muitos casos é desnecessário testar a identidade de $X[L] + 1$ com Y para certos valores de L , visto que o resultado é certamente negativo. O algoritmo de Knuth, Morris e Pratt explora essa propriedade.

◀ Ideia básica do algoritmo ▶

FIGURA 12.3 Comparações efetuadas pelo algoritmo da força bruta.



◀ Ideia básica do algoritmo ▶

A ideia em geral é processar a substring que queremos fazer o casamento de cadeia para determinar o array de LPS ou maior sufixos que também são prefixos.

Iniciamos dois ponteiros, uma para o texto e o outro para o padrão. Quando os caracteres de ambos os ponteiros são iguais, realizamos o incremento e continuamos a comparação. Se não forem iguais resetamos o ponteiro do padrão para para o último valor do array LPS, porque aquela porção já foi verificada.

◀ algoritmo KMP ▶

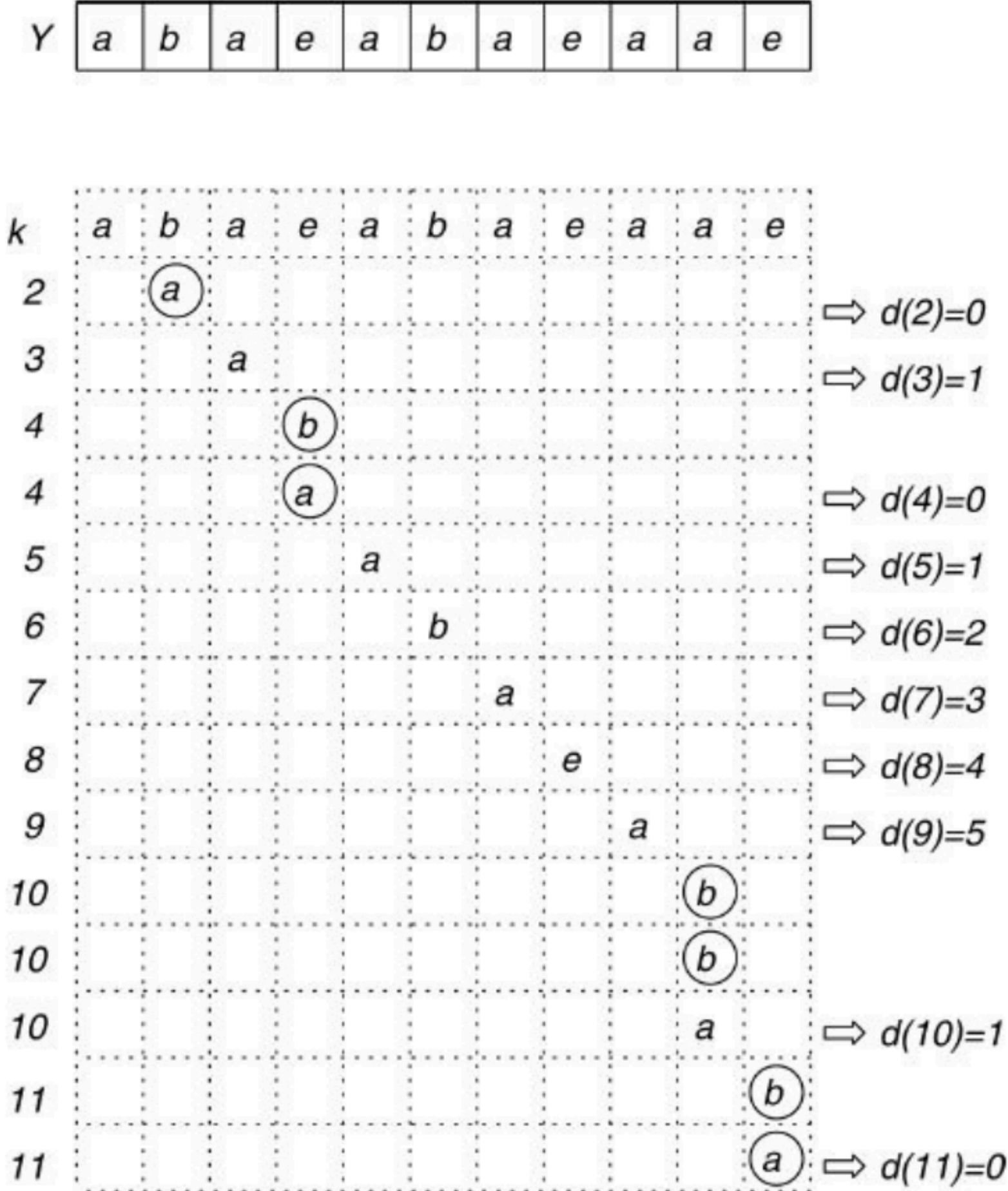


FIGURA 12.7 Comparações efetuadas pelo Algoritmo 12.3.

◀ algoritmo KMP ▶

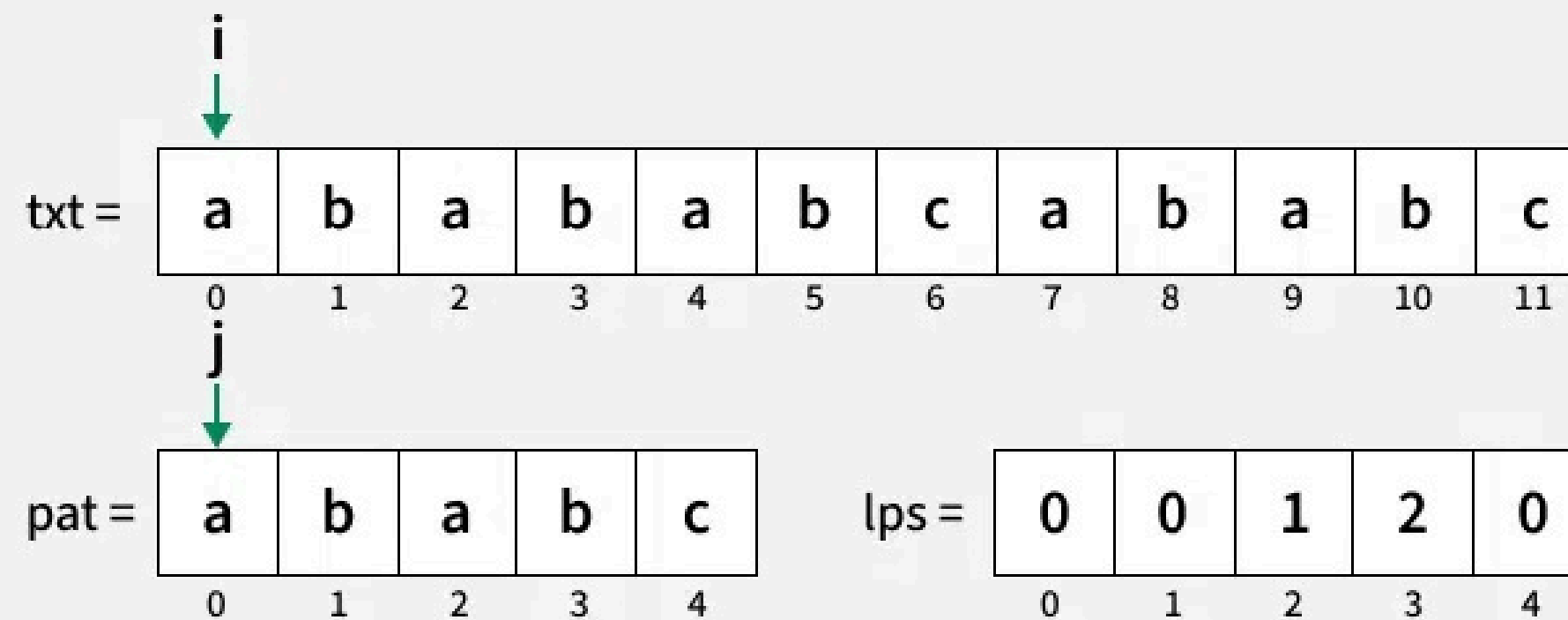
Y	a	b	a	e	a	b	a	e	a	a	e																		
X	b	a	b	a	b	a	e	a	b	a	e	a	b	a	e	a	a	b	a	e	a	b	a	e	a	a	e	b	a
$l=0$	a																												
$l=1$		a	b	a	e																								
$l=3$					b	a	e	a	b	a	e	a	a																
$l=7$													b	a	e	a	a	e											
$l=16$																		b	a	e	a	b	a	e	a	a	e		

FIGURA 12.8 Comparações efetuadas pelo Algoritmo 12.2.

◀ algoritmo KMP ▶

01
Step

Initialize pointers at the beginning of both text and pattern.
The lps for the given pattern would be $\text{lps}[] = \{0, 0, 1, 2, 0\}$



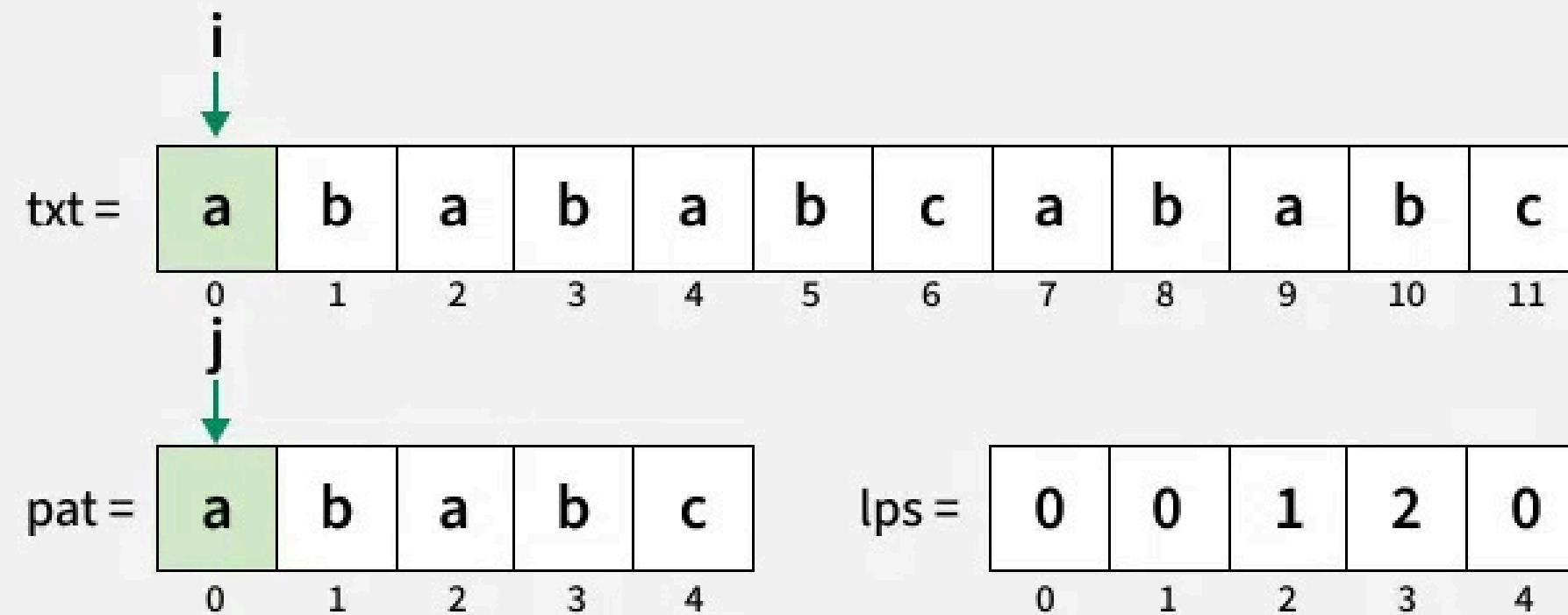
Result = []

KMP Algorithm for Pattern Searching

◀ algoritmo KMP ▶

02
Step

Since $\text{txt}[i]$ and $\text{pat}[j]$ match, increment both i and j .



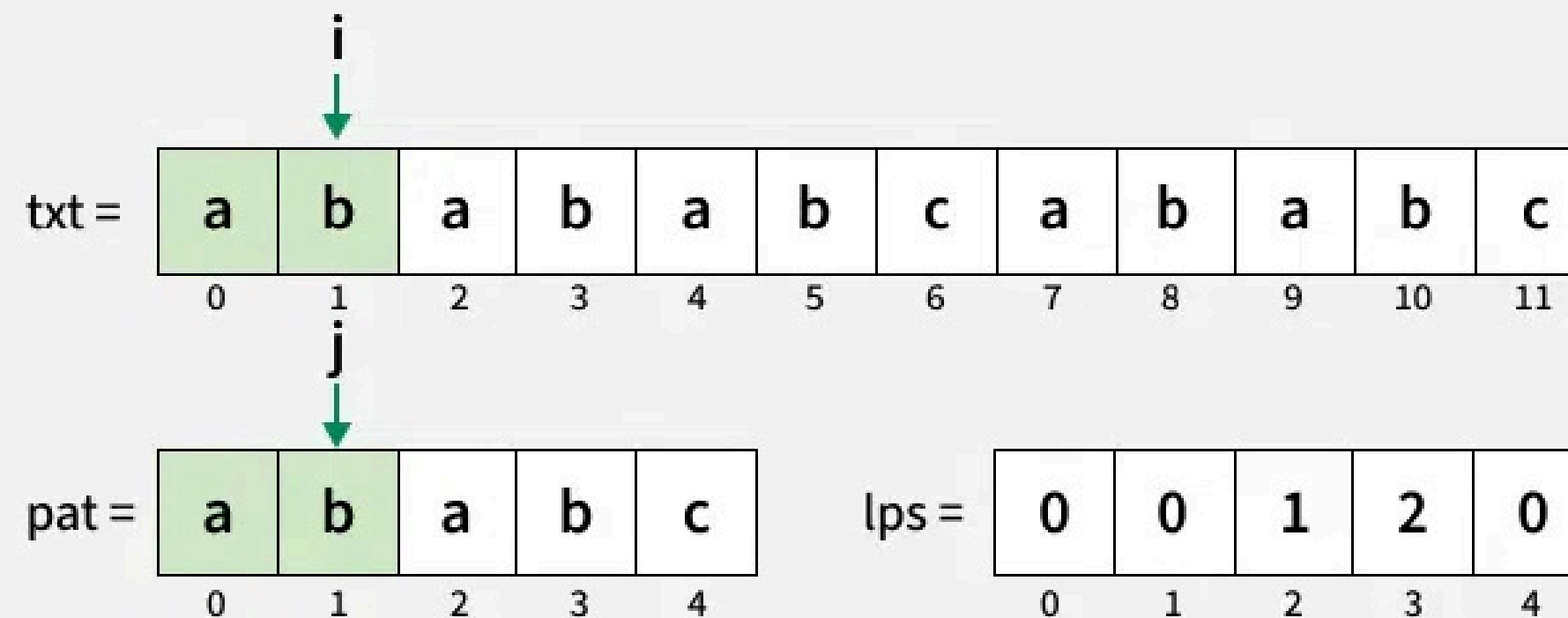
Result = []

KMP Algorithm for Pattern Searching

◀ algoritmo KMP ▶

03
Step

Since $\text{txt}[i]$ and $\text{pat}[j]$ match, increment both i and j .



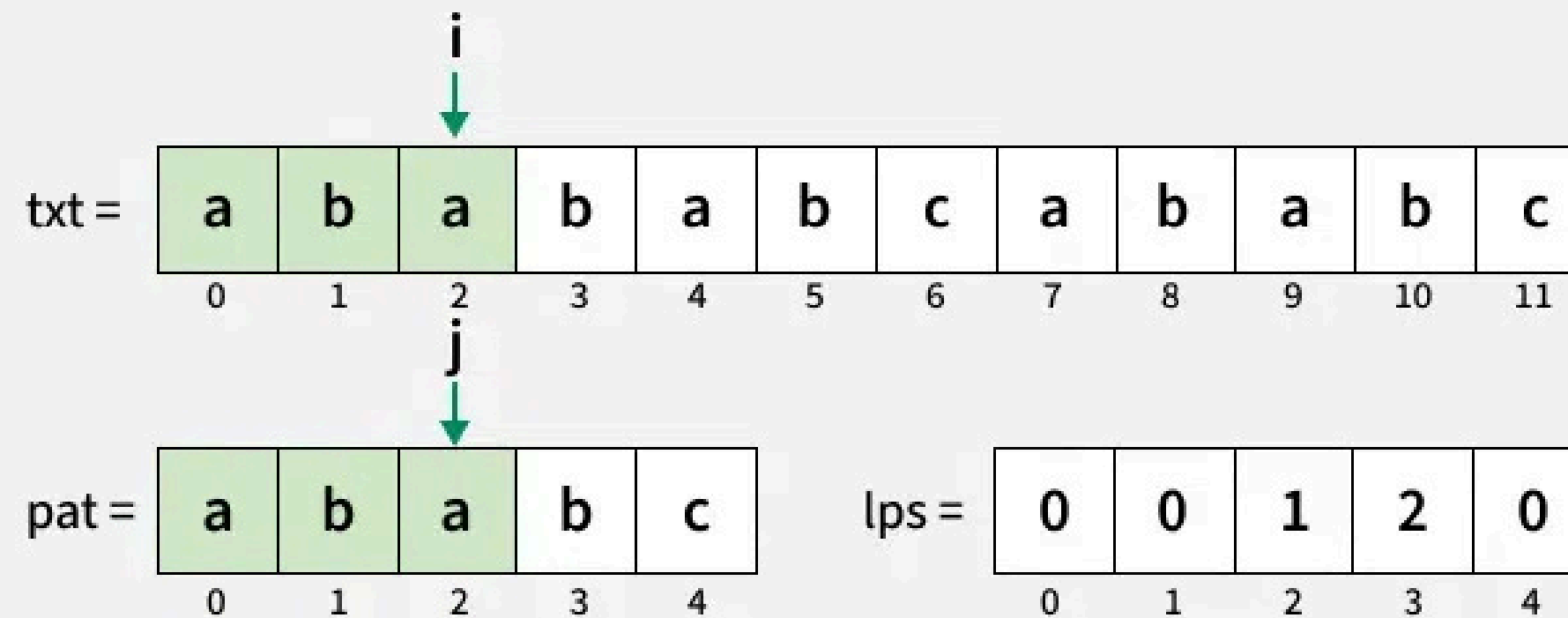
Result = []

KMP Algorithm for Pattern Searching

◀ algoritmo KMP ▶

04
Step

Since $\text{txt}[i]$ and $\text{pat}[j]$ match, increment both i and j .



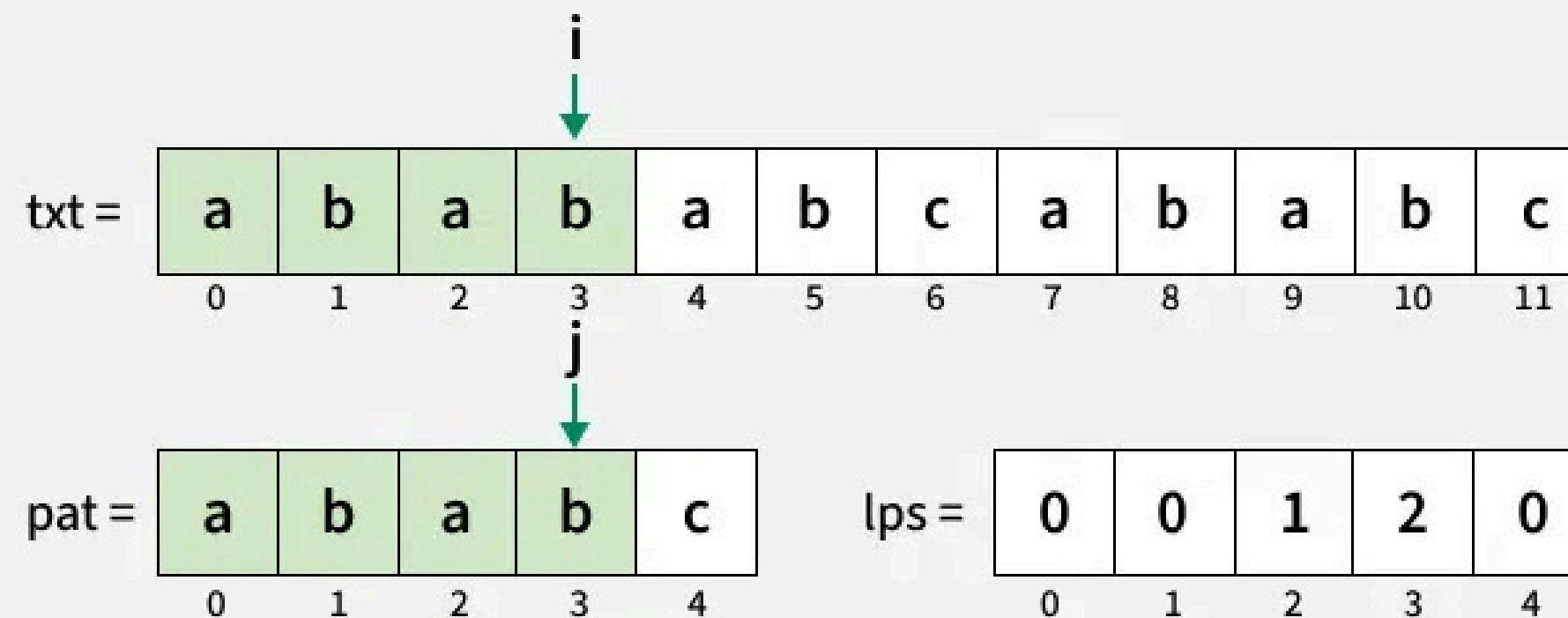
Result = []

KMP Algorithm for Pattern Searching

◀ algoritmo KMP ▶

05
Step

Since $\text{txt}[i]$ and $\text{pat}[j]$ match, increment both i and j .



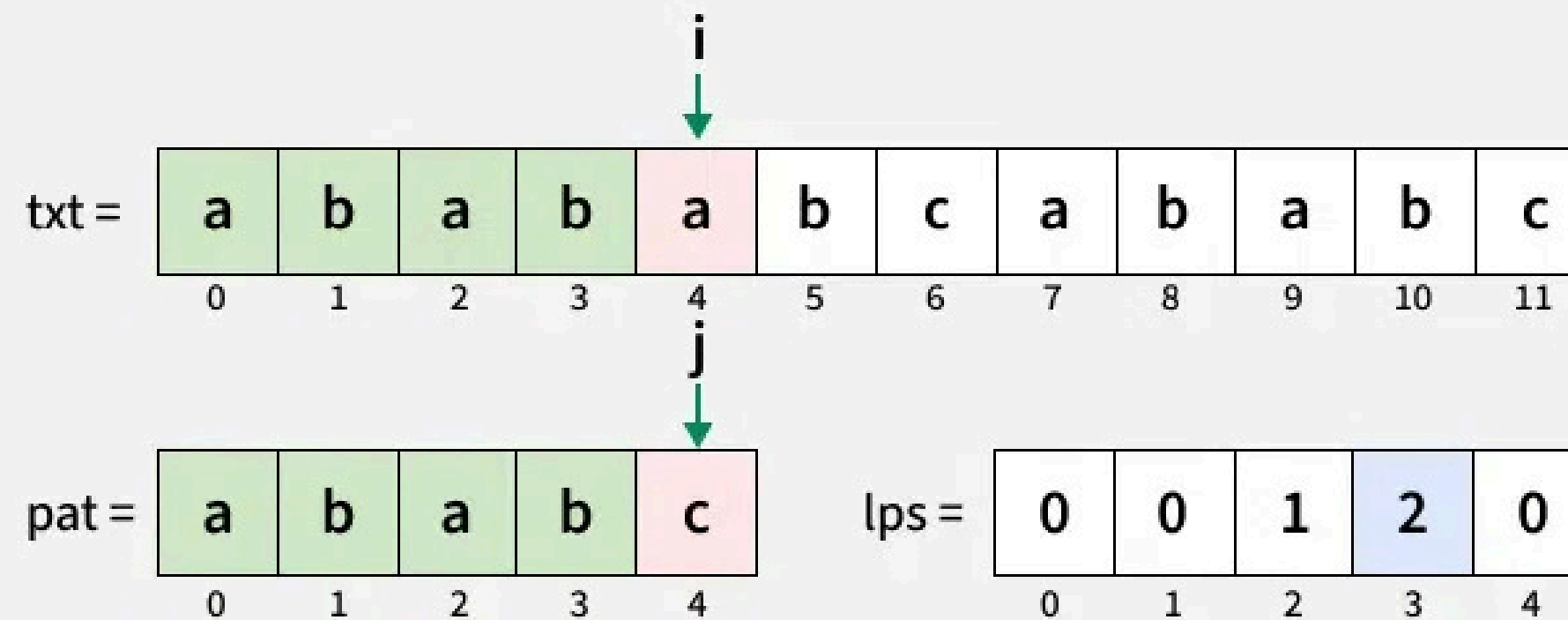
Result = []

KMP Algorithm for Pattern Searching

◀ algoritmo KMP ▶

06
Step

Since $\text{txt}[i]$ and $\text{pat}[j]$ do not match, update j to $\text{lps}[j-1] = 2$



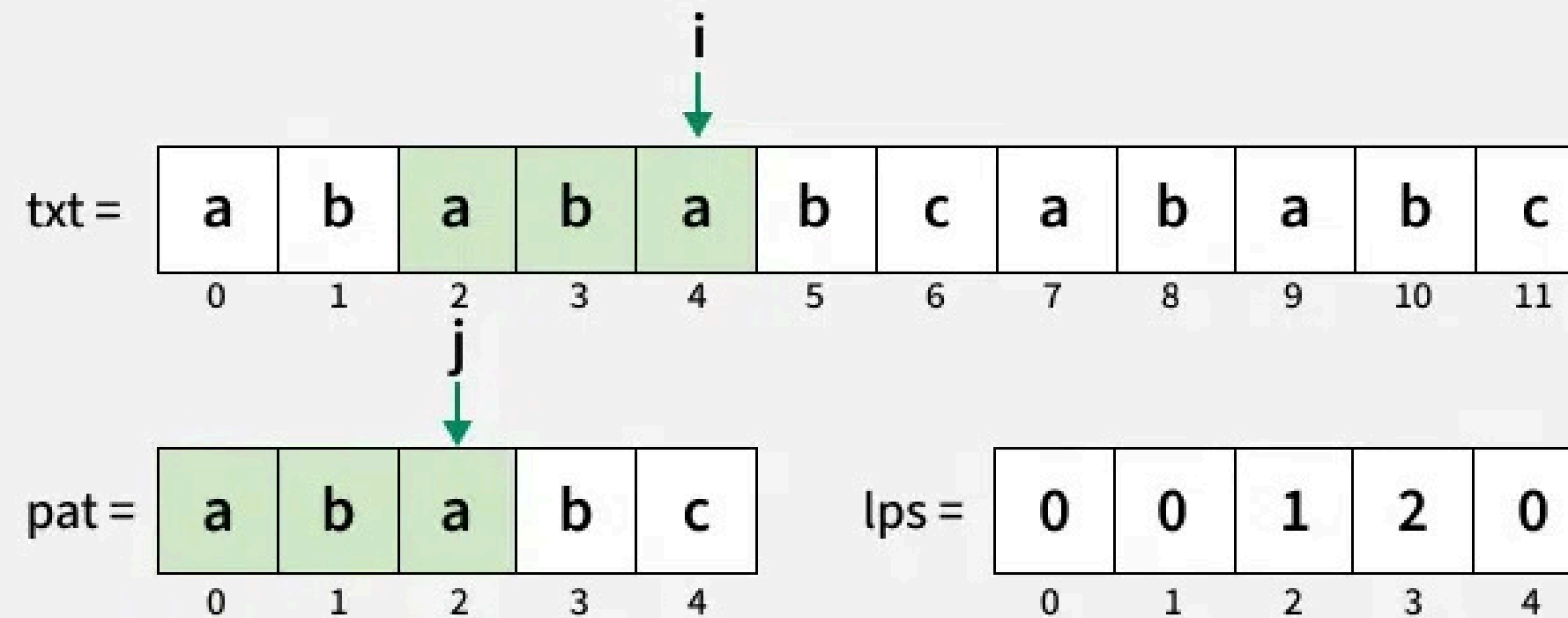
Result = []

KMP Algorithm for Pattern Searching

◀ algoritmo KMP ▶

07
Step

Since $\text{txt}[i]$ and $\text{pat}[j]$ match, increment both i and j .



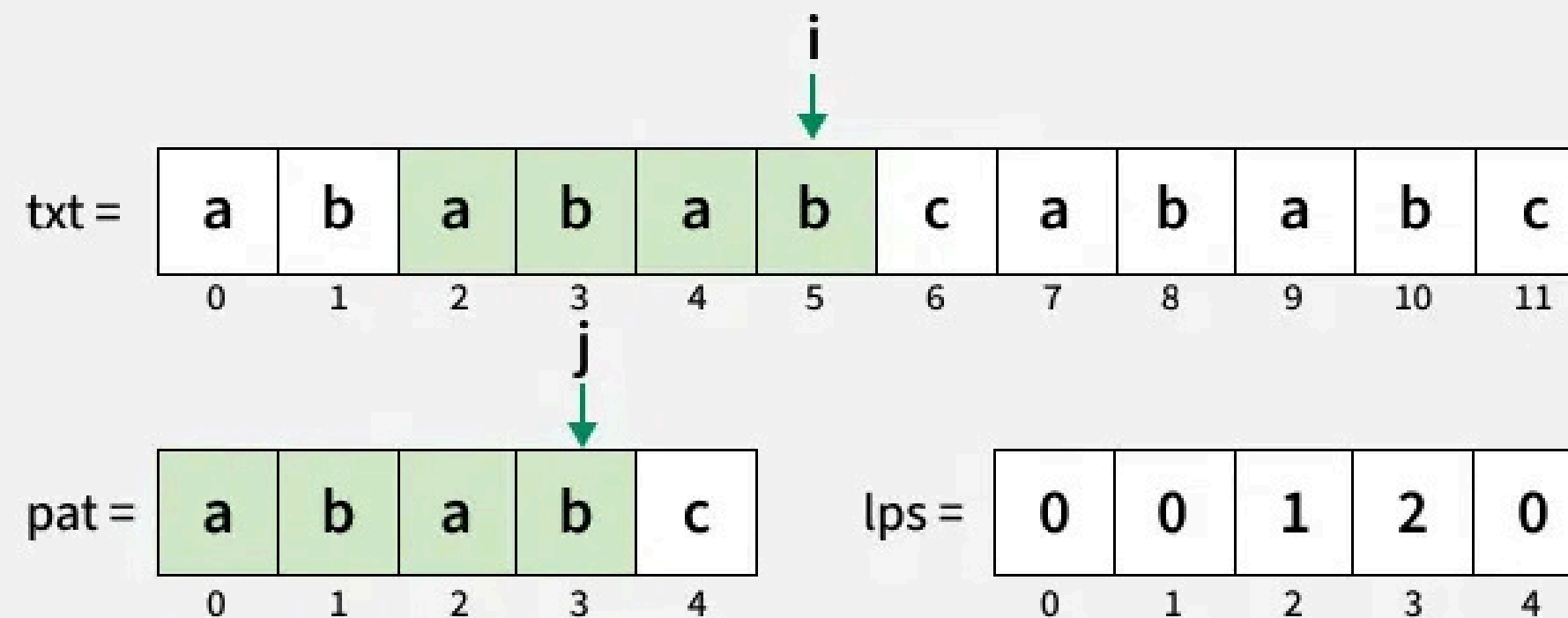
Result = []

KMP Algorithm for Pattern Searching

◀ algoritmo KMP ▶

08
Step

Since $\text{txt}[i]$ and $\text{pat}[j]$ match, increment both i and j .



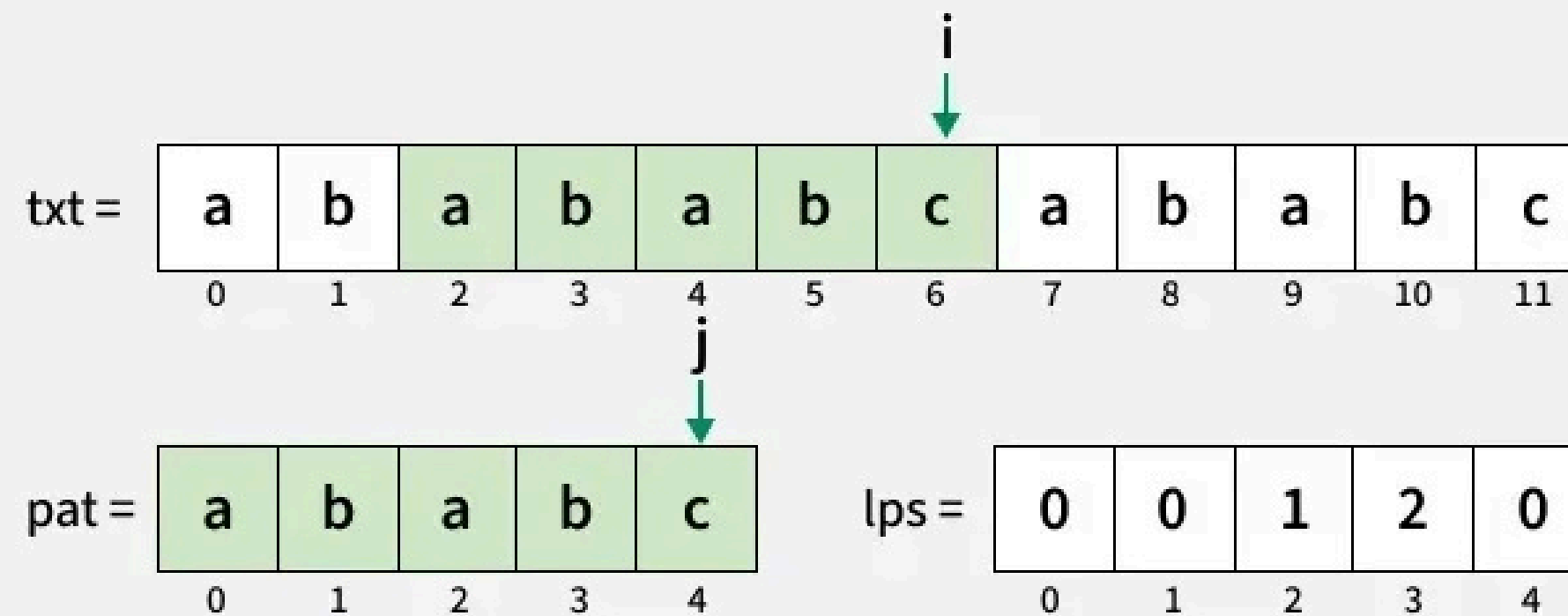
Result = []

KMP Algorithm for Pattern Searching

◀algoritmo KMP▶

09
Step

Since $\text{txt}[i]$ and $\text{pat}[j]$ match, increment both i and j .
Entire pattern has been traversed add the starting index ($i-j$) in the result
And update j to $\text{lps}[j-1] = 0$.



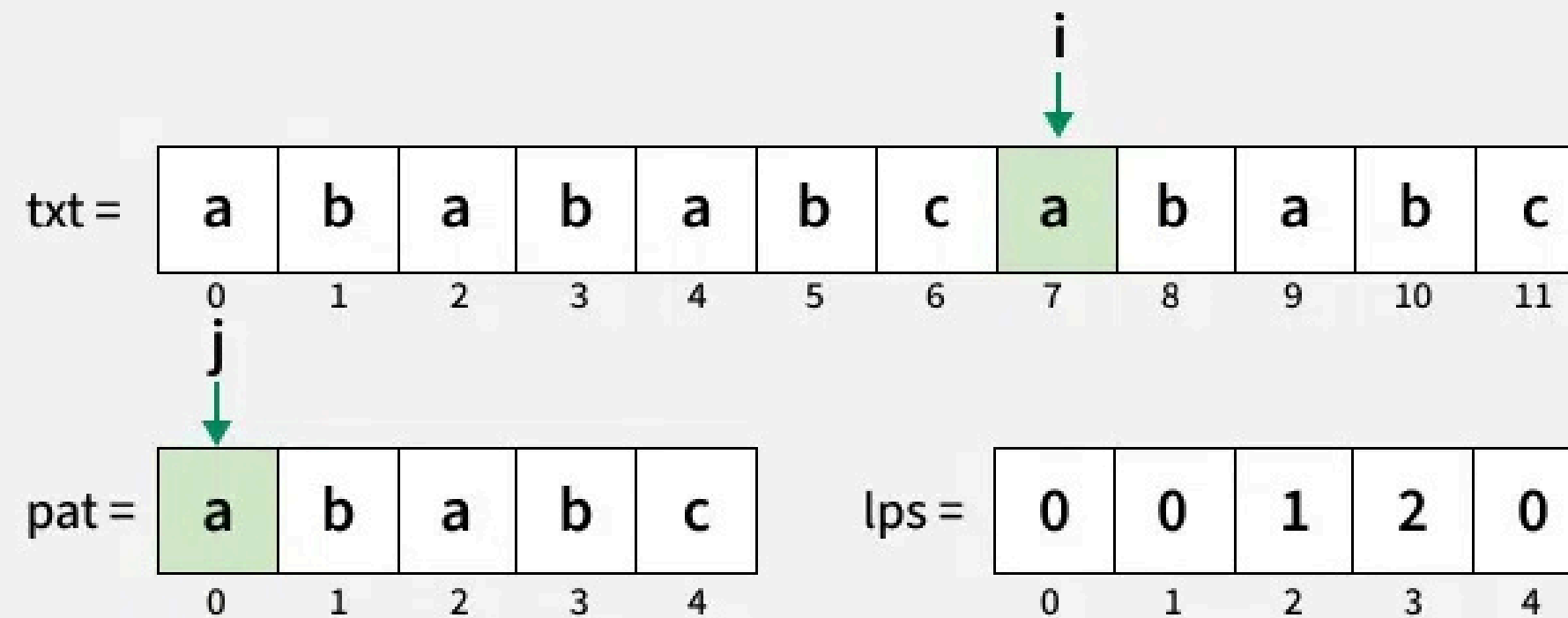
Result = [2]

KMP Algorithm for Pattern Searching

◀ algoritmo KMP ▶

10
Step

Since $\text{txt}[i]$ and $\text{pat}[j]$ match, increment both i and j .



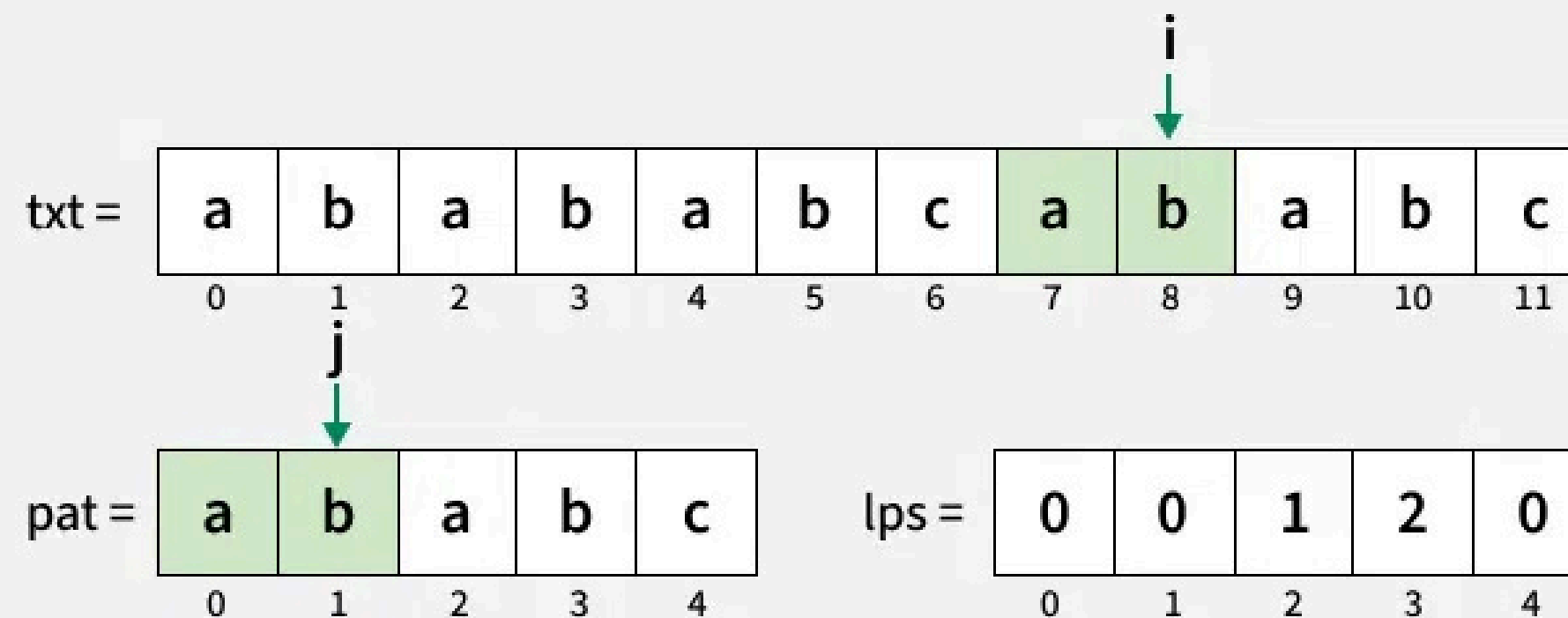
Result = [2]

KMP Algorithm for Pattern Searching

◀ algoritmo KMP ▶

11
Step

Since $\text{txt}[i]$ and $\text{pat}[j]$ match, increment both i and j .



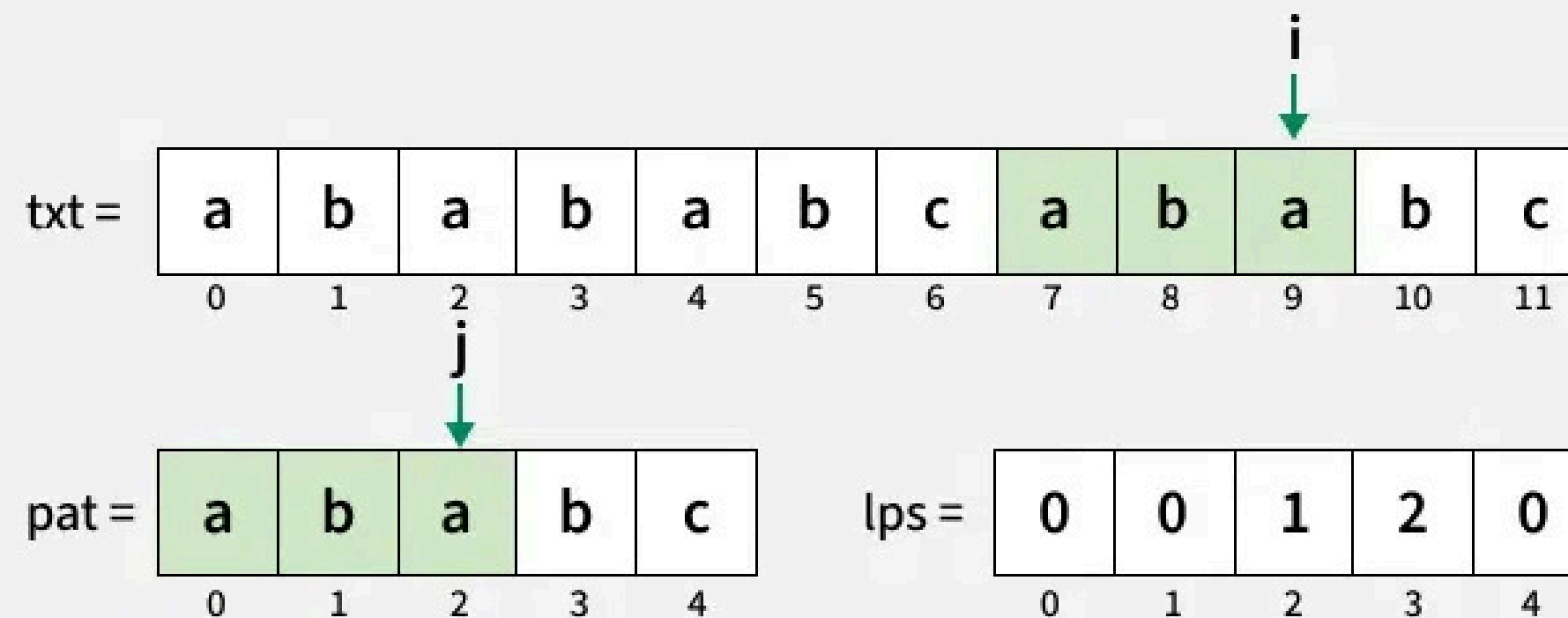
Result = [2]

KMP Algorithm for Pattern Searching

◀ algoritmo KMP ▶

12
Step

Since $\text{txt}[i]$ and $\text{pat}[j]$ match, increment both i and j .



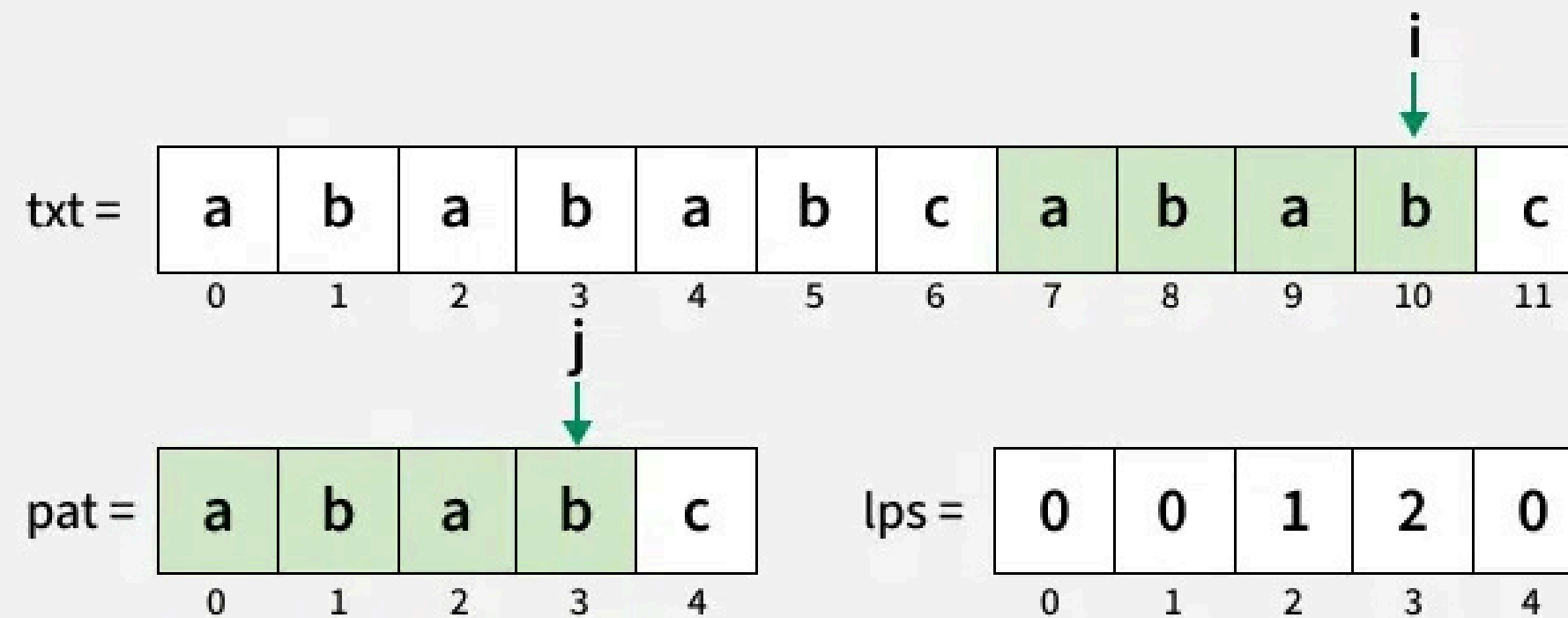
Result = [2]

KMP Algorithm for Pattern Searching

◀ algoritmo KMP ▶

13
Step

Since $\text{txt}[i]$ and $\text{pat}[j]$ match, increment both i and j .



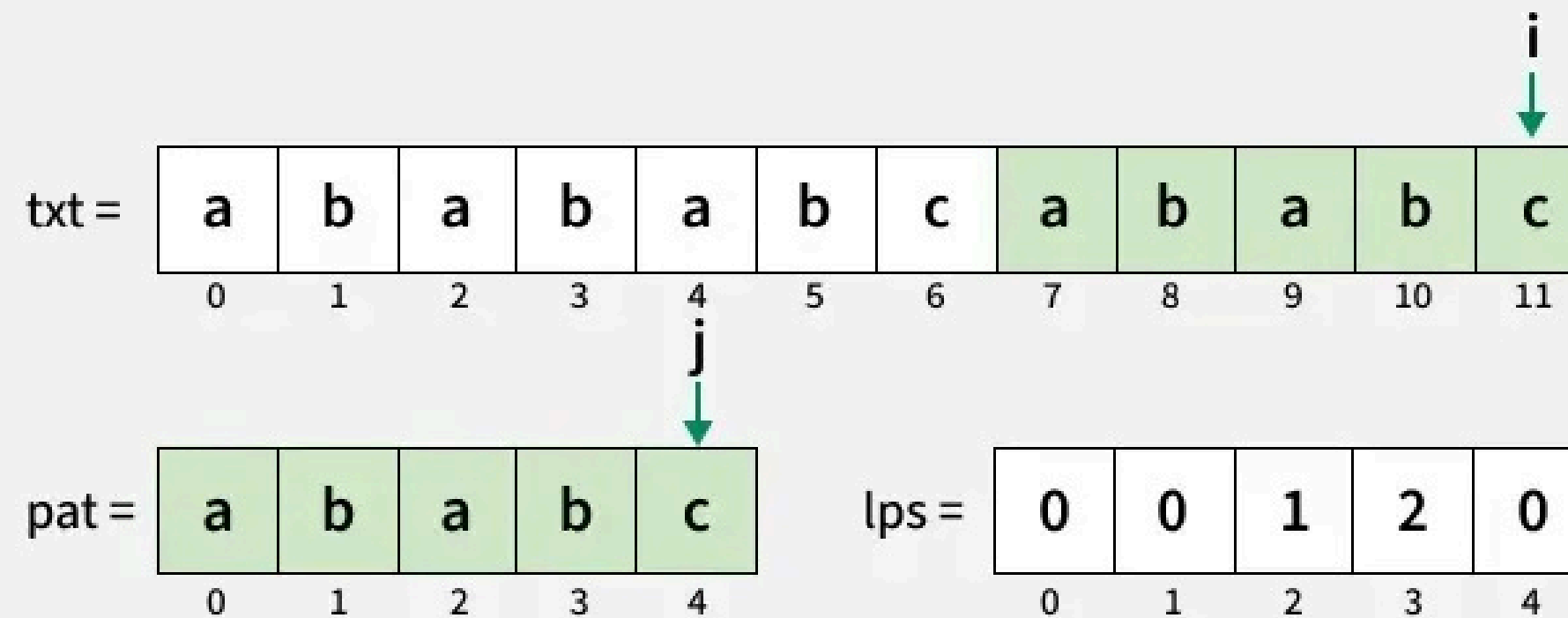
Result = [2]

KMP Algorithm for Pattern Searching

◀ algoritmo KMP ▶

14
Step

Since $\text{txt}[i]$ and $\text{pat}[j]$ match, increment both i and j .
Entire pattern has been traversed add the starting index $(i-j)$ in the result.



Result = [2, 7]

KMP Algorithm for Pattern Searching

Resolução do Problema Motivador

1237 – Comparação de Substring

A resolução estará disponível no Drive. Tente resolver por conta própria e, se precisar, compare com a solução! 😊

Lista de Exercícios

1237 – Comparação de Substring

2087 – Conjuntos Bons e Ruins



Se tiver alguma dúvida ou dificuldade na resolução de algum exercício, sinta-se à vontade para perguntar! 😊

Referências

[1] GEEKSFORGEEKS. Prefix Function and KMP Algorithm for Competitive Programming. GeeksforGeeks, [s. l.], [s. d.]. Disponível em: <https://www.geeksforgeeks.org/prefix-function-and-kmp-algorithm-for-competitive-programming/>. Acesso em: 25 maio 2025.

[2] BRUNO MONTEIRO. Algoritmo de KMP[vídeo]. YouTube, 29 jul. 2022. Disponível em: <https://www.youtube.com/watch?v=RXISWaGmYW8>. Acesso em: 25 maio 2025