LU4: Brute Force and KMP – String Searching

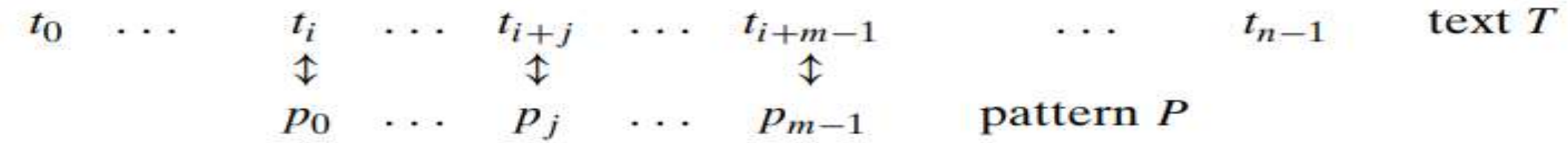
Brute Force– String Searching

Brute-Force : String Searching

- It is the process of **finding pattern in the larger text**. Also called as ***String Matching***.
    - *pattern*: a string of m characters to search for.
    - *text*: a (longer) string of n characters to search in.
- *problem*: given a string of n characters called the ***text*** and a string of m characters $(m \leq n)$ called the ***pattern,*** find a substring of the text that matches the pattern.
- Example:
    - Pattern: **Hi**
    - Text: Hello, Good Morning. **Hi**, Nice meeting you

Brute-Force : String Searching (Contd.,)

- Goal: To find i—the index of the leftmost character of the first matching substring in the text.



- such that

ti = p0, ti+1=p1, ...,ti+j = pj , ... ,ti+m−1 = pm−1

- Solution: If match occur, return index i. Otherwise return -1.

Brute-Force : String Searching (Contd.,)

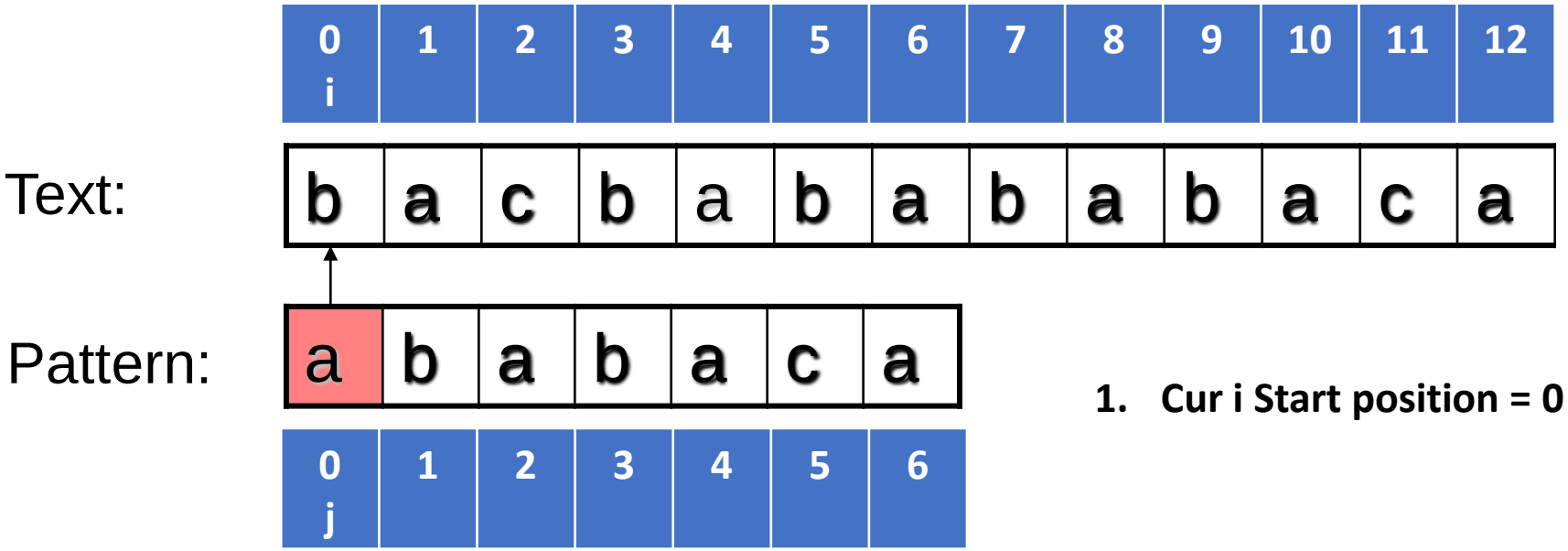
Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until

- all characters are found to match (successful search); or
- a first mismatch is detected

Step 3 While pattern is not found and the text is not yet exhausted, **realign pattern one position to the right** and repeat Step 2

Brute-Force : String Searching-Example

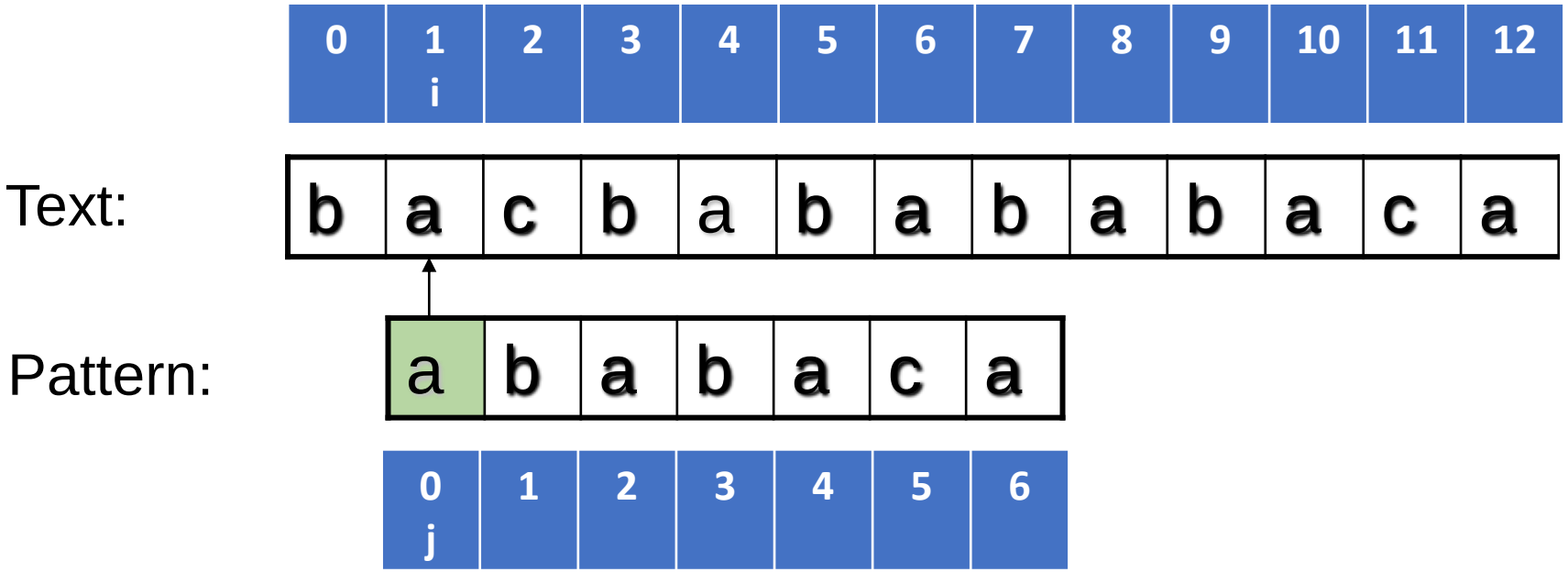


1. Cur i Start position = 0

Process:

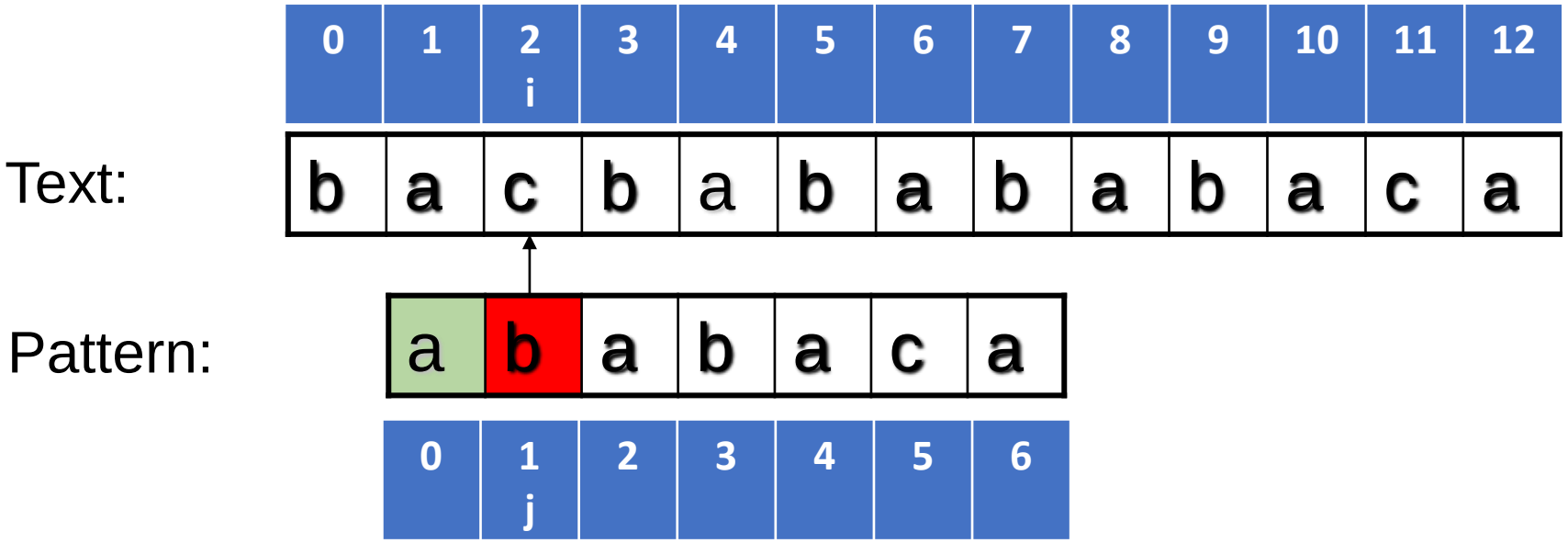
1. If there is a match, move i as well as j
2. Else
    1. If there is a mismatch, j resets to 0 in the next step
    2. Move i to the next position of previously started index

Brute-Force : String Searching-Example (Contd.,)



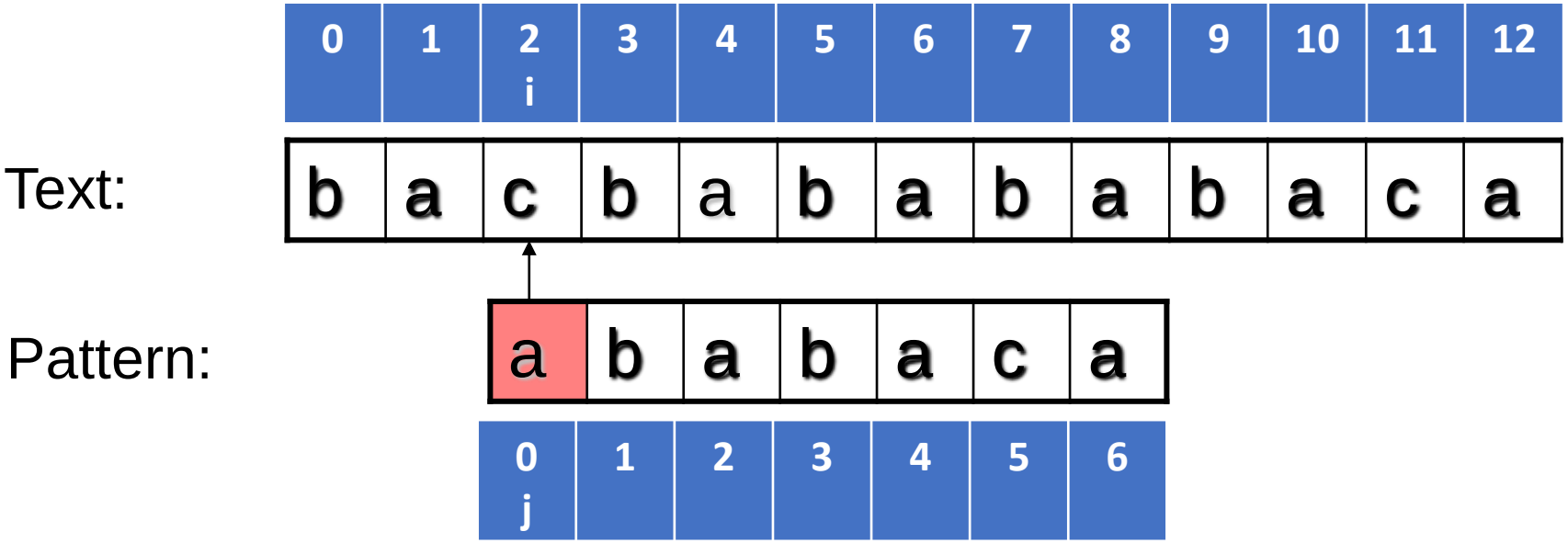
1. **Cur i Start position = 1**
2. **There is a match, move i as well as j**

Brute-Force : String Searching-Example (Contd.,)



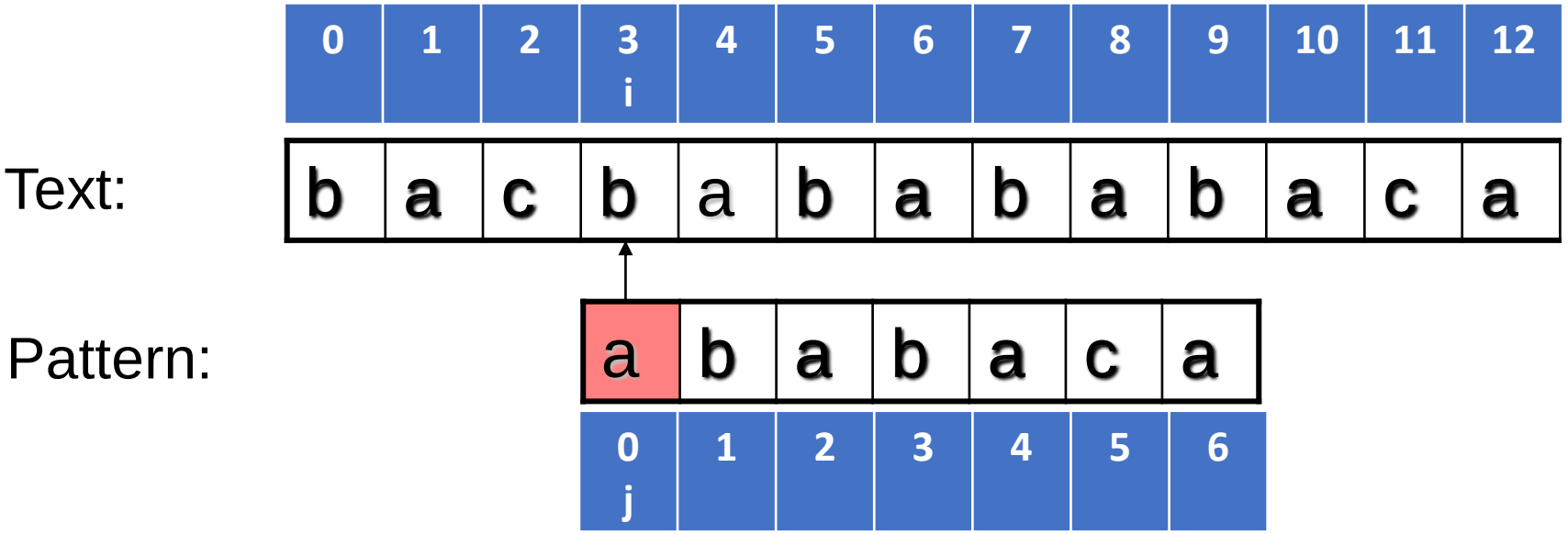
1. There is a mismatch, j resets to 0 in the next step
2. Move i to the next position = 2 (Cur. start position of i = 1, next position = Cur + 1 = 2)

Brute-Force : String Searching-Example (Contd.,)



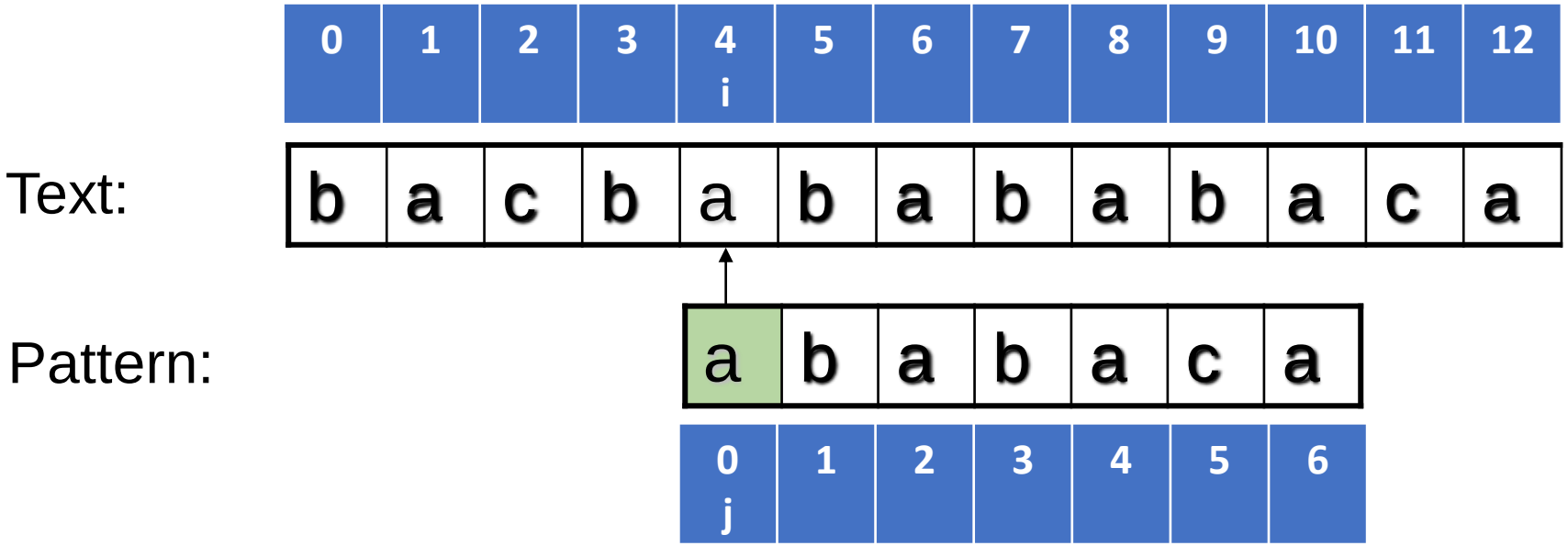
1. There is a mismatch, j resets to 0 in the next step
2. Move i to the next position = 3 (Cur. start position of i = 2, next position = cur + 1 = 3)

Brute-Force : String Searching-Example (Contd.,)



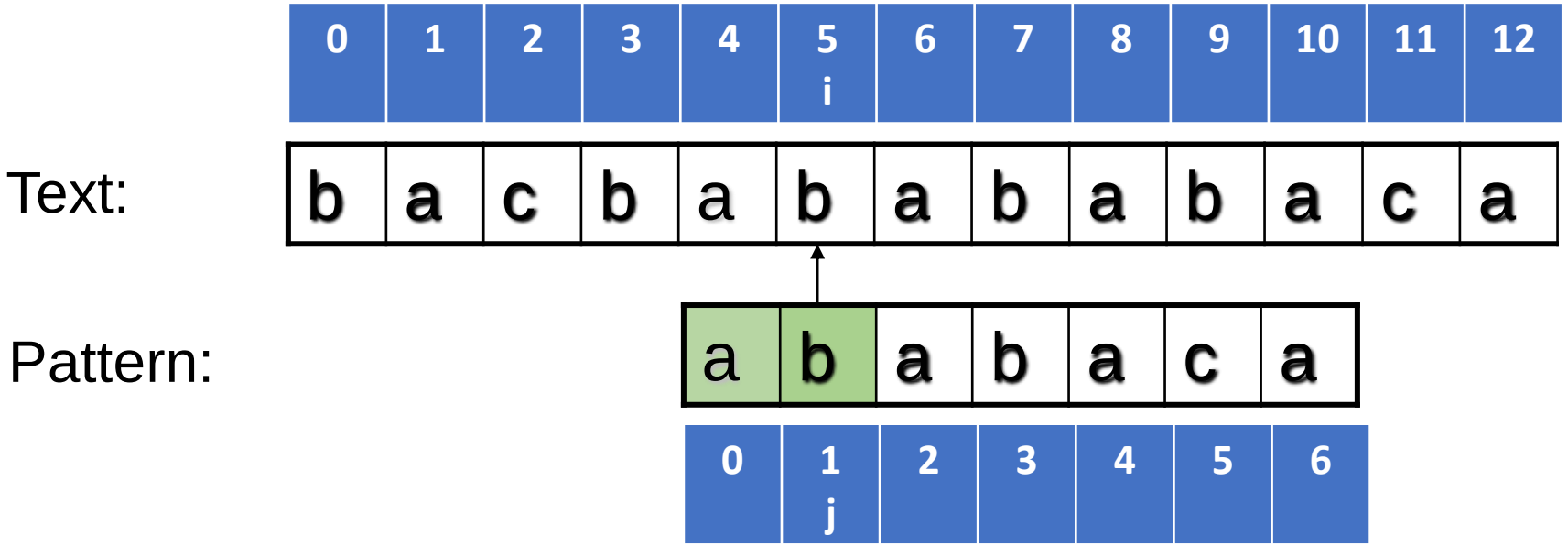
1. There is a mismatch, j resets to 0 in the next step
2. Move i to the next position = 4 (Prev. start position of i = 3, next position = prev + 1 = 4)

Brute-Force : String Searching-Example (Contd.,)



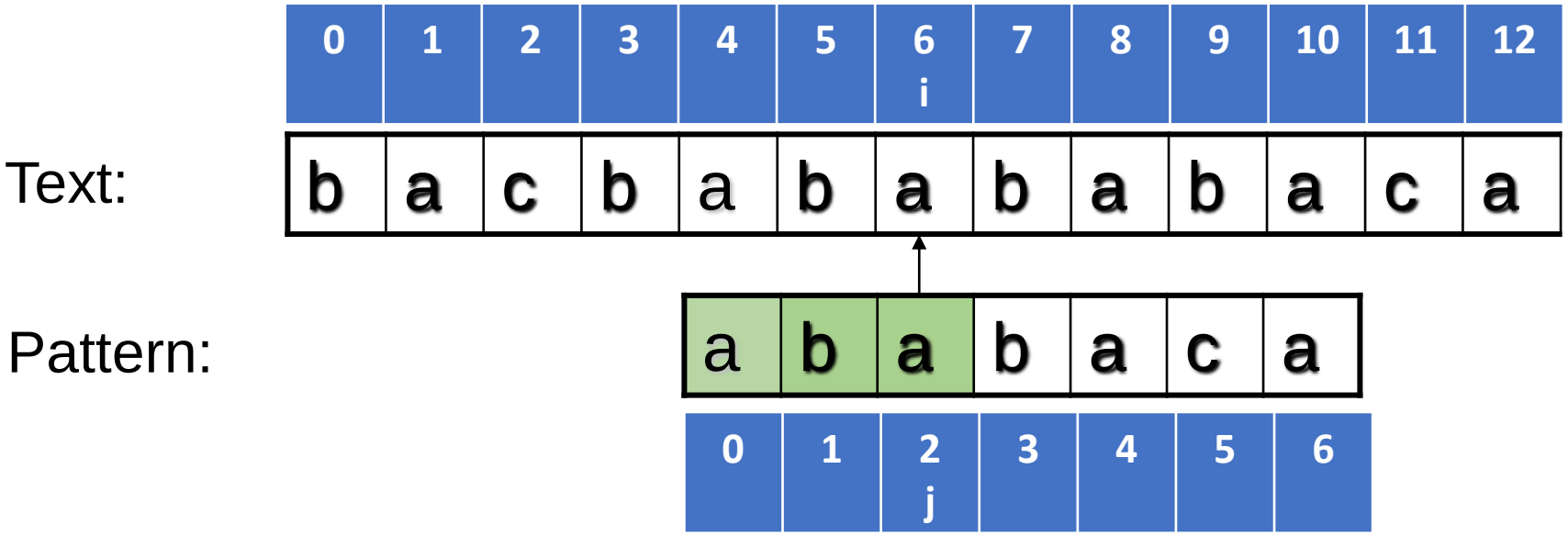
1. **Cur i Start position = 4**
2. **There is a match, move i as well as j**

Brute-Force : String Searching-Example (Contd.,)



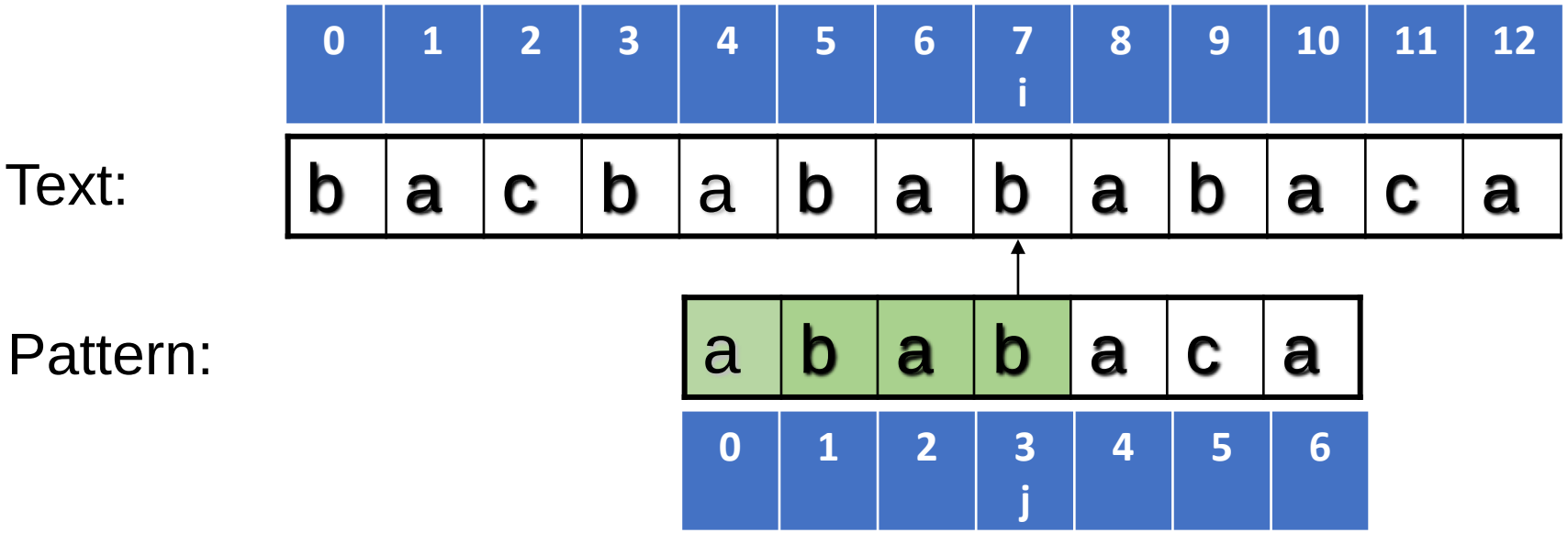
1. **Cur i Start position = 4**
2. **There is a match, move i as well as j**

Brute-Force : String Searching-Example (Contd.,)



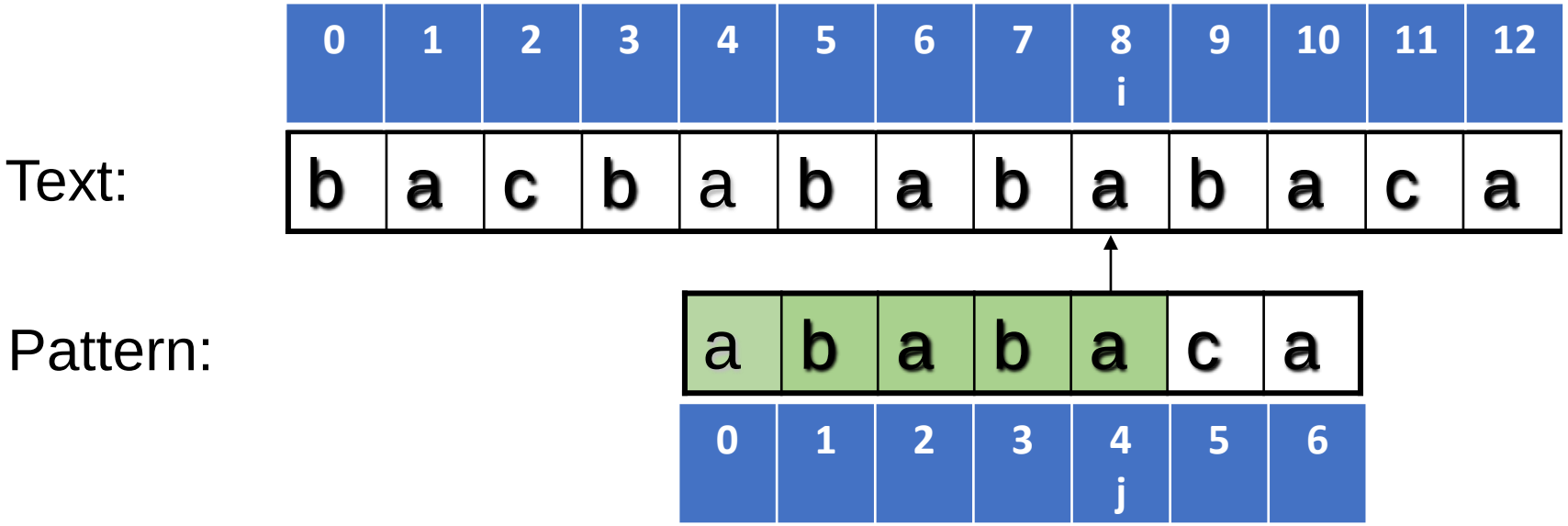
1. **Cur i Start position = 4**
2. **There is a match, move i as well as j**

Brute-Force : String Searching-Example (Contd.,)



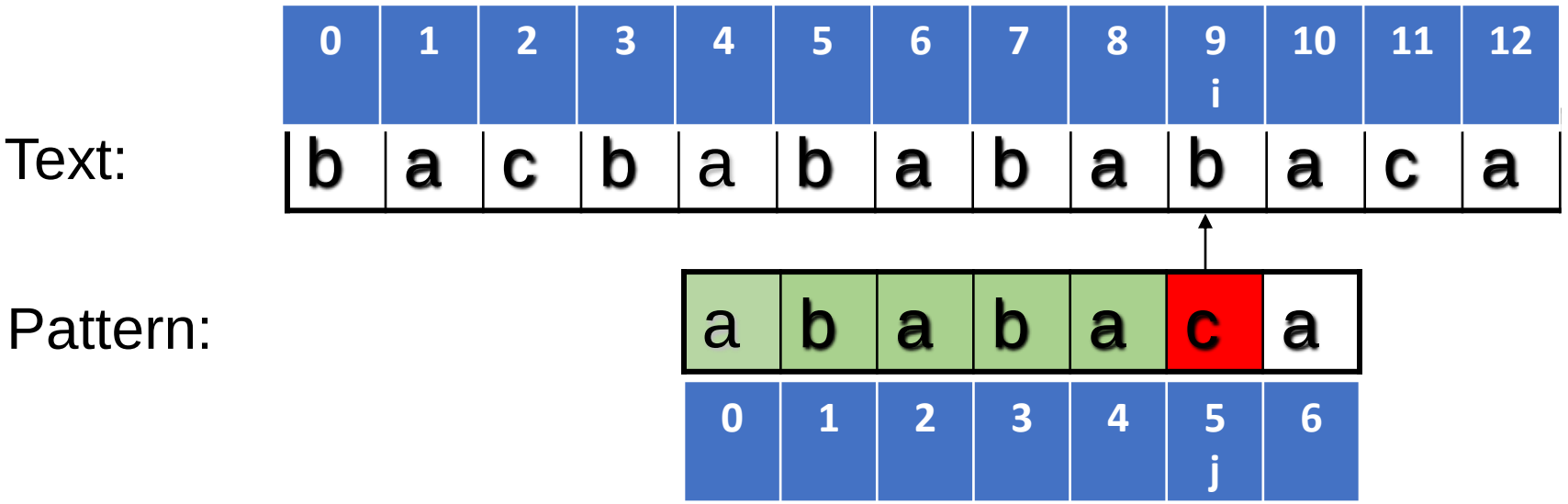
1. **Cur i Start position = 4**
2. **There is a match, move i as well as j**

Brute-Force : String Searching-Example (Contd.,)



1. **Cur i Start position = 4**
2. **There is a match, move i as well as j**

Brute-Force : String Searching-Example (Contd.,)

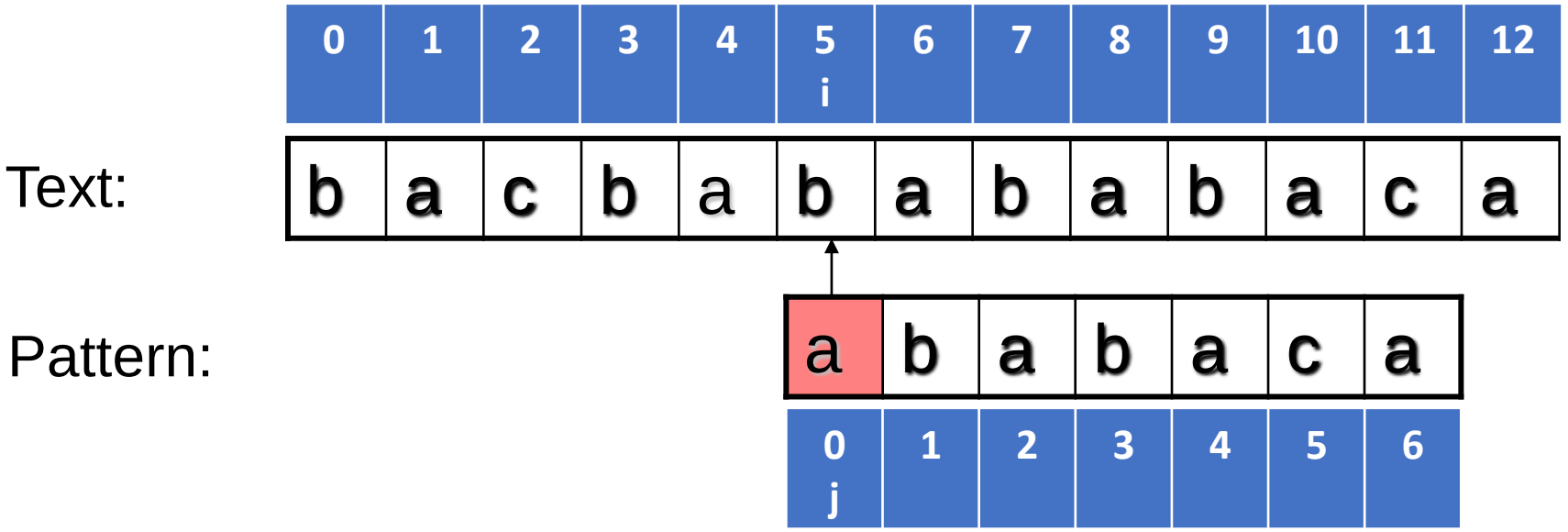


1. **There is a mismatch, j resets to 0 in the next step**
2. **Move i to the next position = 5 (Cur. start position of i = 4, next position = cur + 1 = 5)**

Drawback of **Bruteforce**:
Already i has reached **9**, when it finds mismatch, j sets to 0, but again i has to reset its index (**shifted back**) to earlier visited one . i.e. here the next shift of I will be **5**.

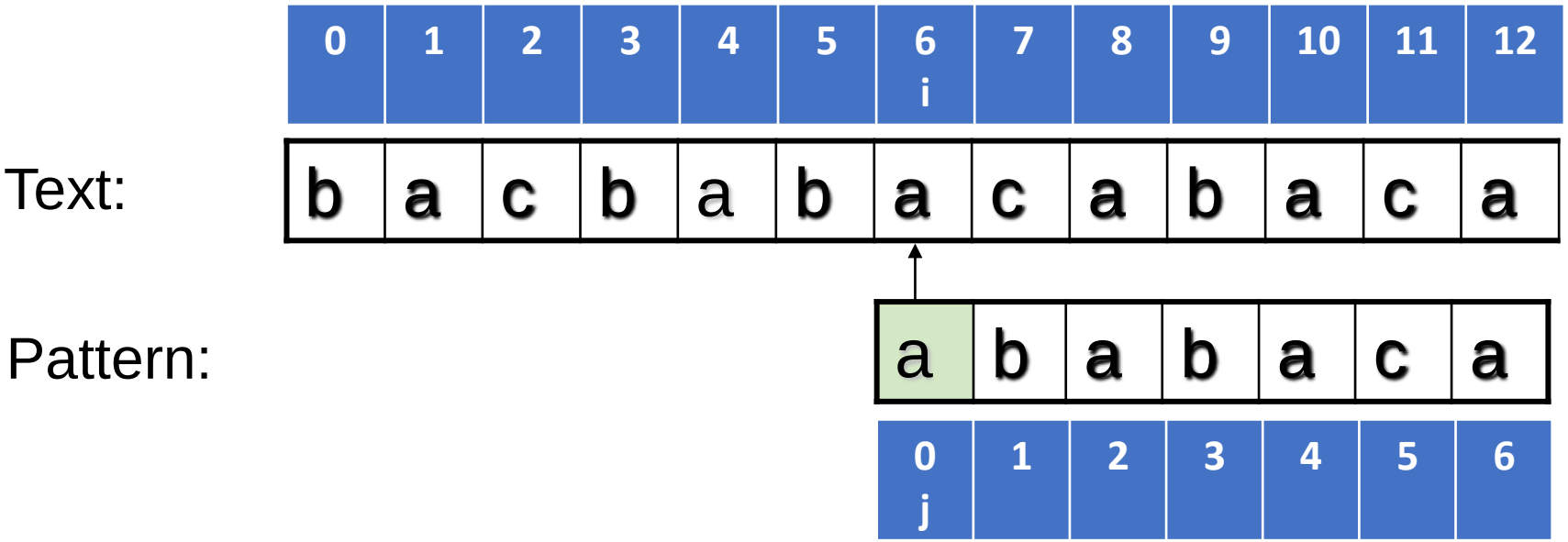
Here, I is shifting in the string. This is extra work in basic algorithm.

Brute-Force : String Searching-Example (Contd.,)



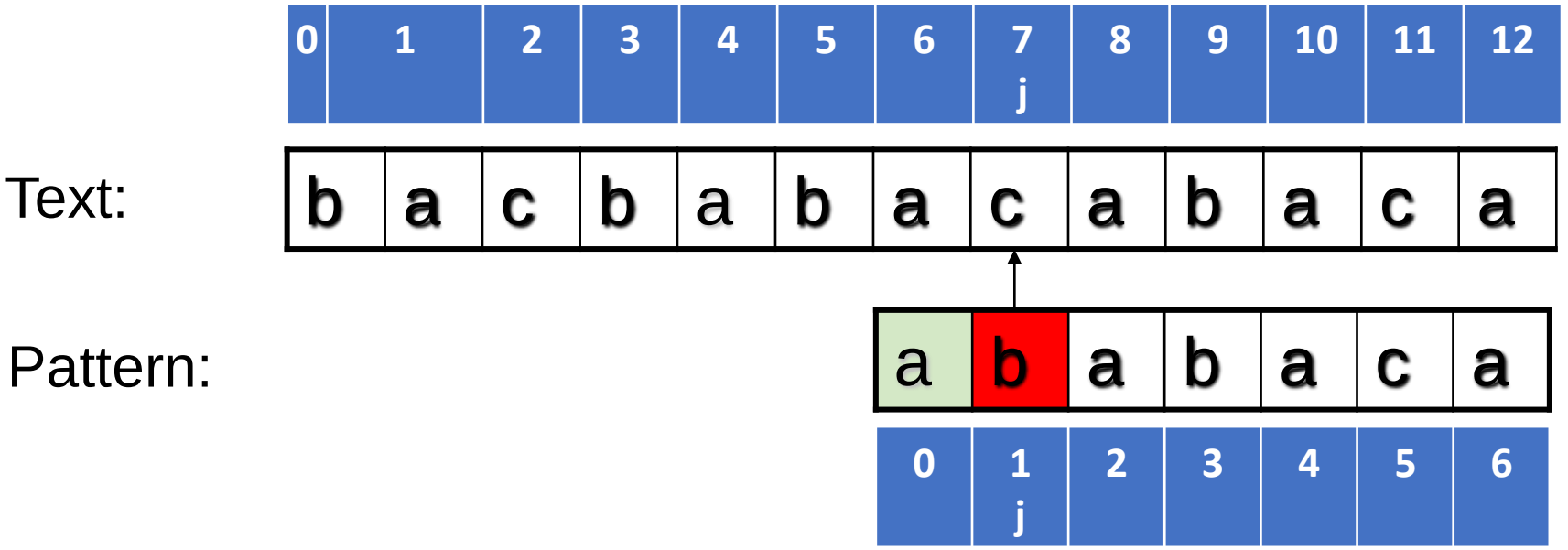
1. There is a mismatch, j resets to 0 in the next step
2. Move i to the next position = 6 (Cur. start position of i = 5, next position = cur + 1 = 6)

Brute-Force : String Searching-Example (Contd.,)



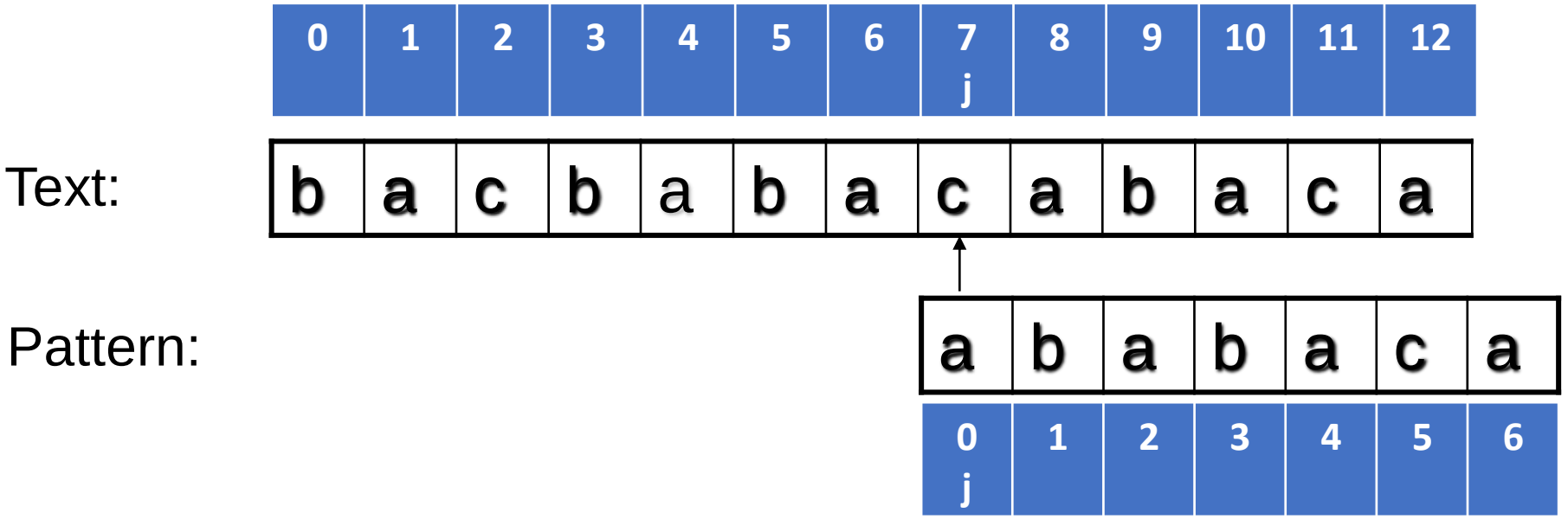
1. **Cur i Start position = 6**
2. **There is a match, move i as well as j**

Brute-Force : String Searching-Example (Contd.,)



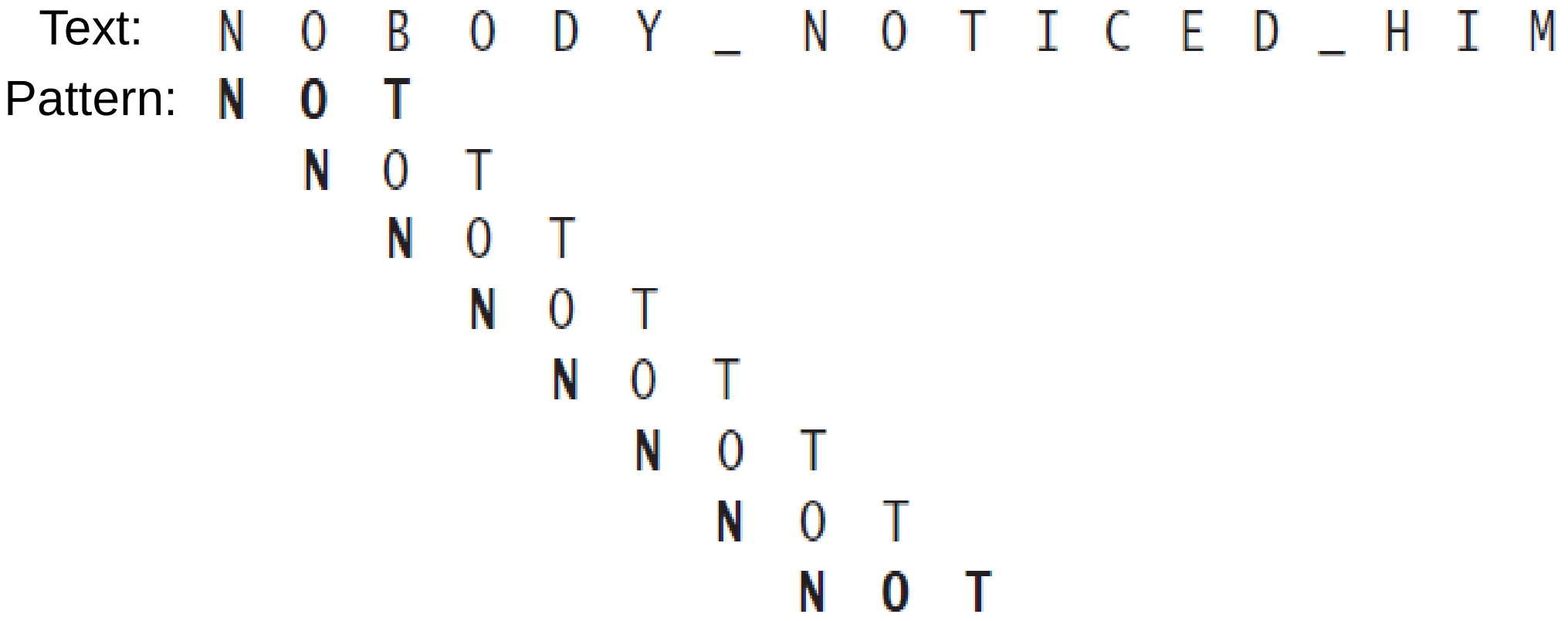
1. There is a mismatch, j resets to 0 in the next step
2. Move i to the next position = 7 (Cur. start position of i = 6, next position = cur + 1 = 7)

Brute-Force : String Searching-Example (Contd.,)



1. There is a mismatch, j resets to 0 in the next step
2. Move i to the next position = 7 (Cur. start position of i = 6, next position = cur + 1 = 7)
3. Since the length of the pattern exceeds the length of the remaining text to be compared, the process can be terminated.

Brute-Force : String Searching-Example (Contd.,)



Brute-Force : String Searching - Algorithm

ALGORITHM *BruteForceStringMatch*$(T[0..n-1],\ P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

Brute-Force : String Searching – Algorithm (Contd.,)

Mathematical Analysis of algorithm:

- **Input Size:** n (as m<<n)
- **Basic Operation:** Comparison
- Does number of times basic operation execution depend only on input size?
  - No!!!!. It also depends on the specifics of the input.
  - Hence best, worst and average case efficiencies has to be examined.
- **Best Case: Ω(m)** as pattern of length m matches in the beginning of the text.
- **Average Case: Ω(n+m)** [with equal probability assumption, the average number of comparisons is proportional to the sum of the lengths of the text and the pattern]

  O(n) [for searching random text, efficiency is considerably better]

Brute-Force : String Searching – Algorithm (Contd.,)

Mathematical Analysis of algorithm: (Contd.,)

- **Worst Case**:

$$C(n) = \sum_{i=0}^{n-m} \sum_{j=0}^{m} 1$$

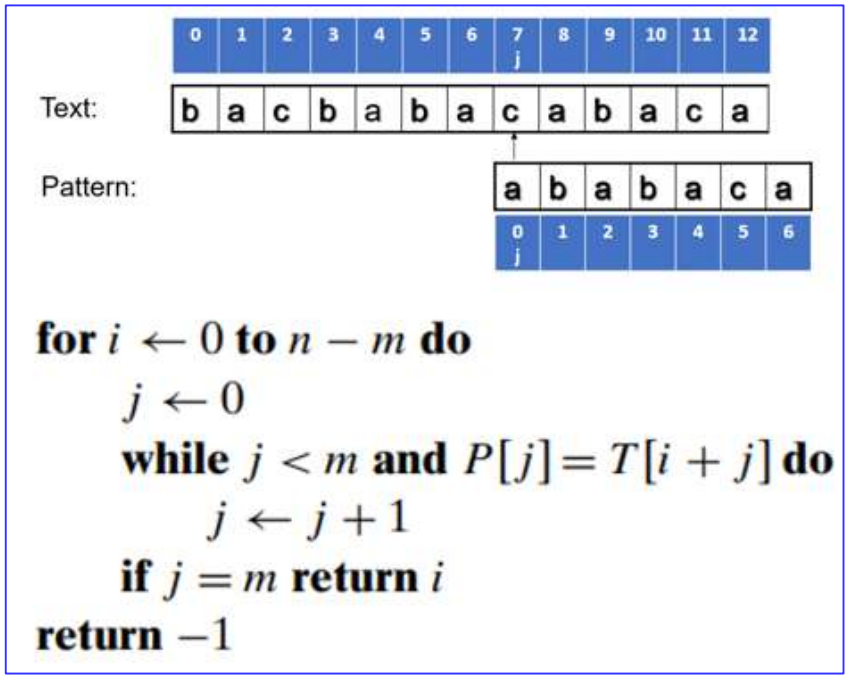
$$= \sum_{i=0}^{n-m} (m - 0 + 1)$$

$$= \text{m.} \sum_{i=0}^{n-m} 1$$

$$= \text{m(n-m-0+1)}$$

$$= \text{m.(n-m+1)}$$

Therefore, C(n) = O(nm)



It means that, it may have to make all *m* comparisons before shifting the pattern, and this can happen for (n–m+1) possible starting positions in the input text.

Exercises

- **Pattern:** 001011
  Text: 10010101101001100101111010

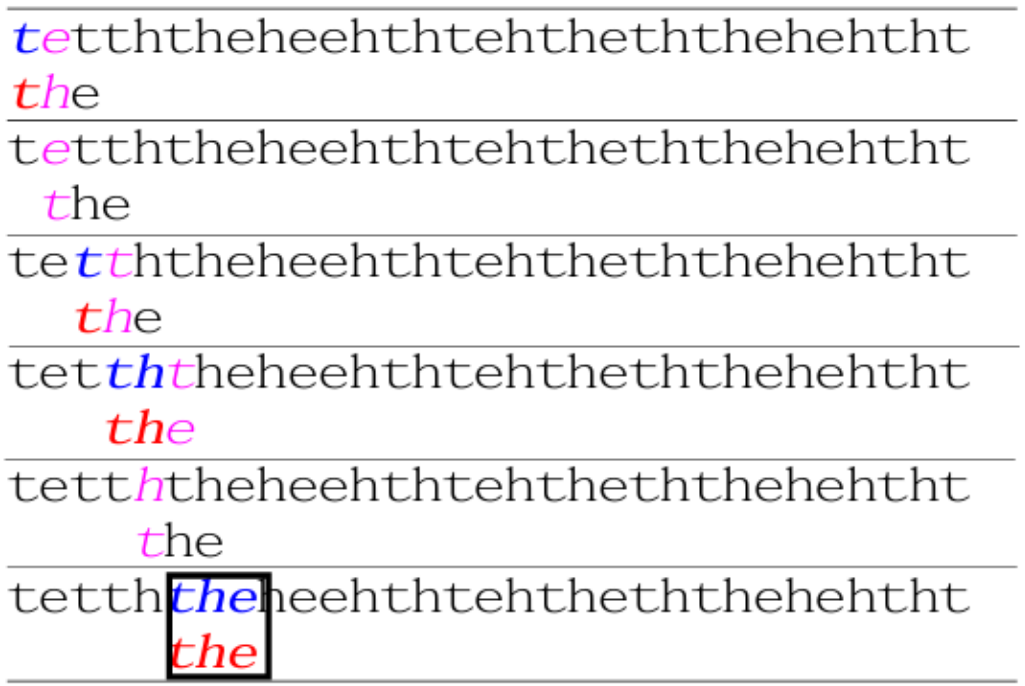
- **Pattern:** happy

  Text: It is never too late to have a happy childhood.

Brute Force Pseudo-Code

- Here's the pseudo-code

```
do
  if (text letter == pattern letter)
    compare next letter of pattern to next
    letter of text
  else
    move pattern down text by one letter
while (entire pattern found or end of text)
```



Brute Force-Complexity

- Given a pattern M characters in length, and a text N characters in length...
- **Worst case**: compares pattern to each substring of text of length M. For example, M=5.

1) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
   AAAAH **5 comparisons made**
2) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
    AAAAH **5 comparisons made**
3) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
     AAAAH **5 comparisons made**
4) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
      AAAAH **5 comparisons made**
5) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
       AAAAH **5 comparisons made**

   ....

N) AAAAAAAAAAAAAAAAAAAAA*AAAAH*
   5 comparisons made *AAAAH*

- Total number of comparisons: M (N-M+1)
- Worst case time complexity: O(MN)

Brute Force-Complexity(cont.)

- Given a pattern M characters in length, and a text N characters in length...
- **Best case if pattern found**: Finds pattern in first M positions of text. For example, M=5.

1) *AAAAA*AAAAAAAAAAAAAAAAAAAAAH
   AAAAA **5 comparisons made**

- Total number of comparisons: M
- Best case time complexity: O(M)

Brute Force-Complexity(cont.)

- Given a pattern M characters in length, and a text N characters in length...
- **Best case if pattern not found**: Always mismatch on first character. For example, M=5.

1) *A*AAAAAAAAAAAAAAAAAAAAAAAAH
   *O*OOOH **1 comparison made**
2) A*A*AAAAAAAAAAAAAAAAAAAAAAAH
   *O*OOOH **1 comparison made**
3) AA*A*AAAAAAAAAAAAAAAAAAAAAAH
   *O*OOOH **1 comparison made**
4) AAA*A*AAAAAAAAAAAAAAAAAAAAAH
   *O*OOOH **1 comparison made**
5) AAAA*A*AAAAAAAAAAAAAAAAAAAAH
   *O*OOOH **1 comparison made**

   ...

N) AAAAAAAAAAAAAAAAAAAAAAA*A*AAAH
   1 comparison made *O*OOOH

- Total number of comparisons: N
- Best case time complexity: O(N)

Needle in a Haystack

- Search for the 10-symbol "needle" "helen hunt" in the Internet "haystack" with many TBs of data
- The brute force algorithm amounts to: "Look in this corner, now in this other corner, then over there, and so on."
- The Internet holds some 1^+ trillion pages, growing by billions a day; each page on average contains in excess of 10 KB, say

$m \cong 10$

$n \cong 10^{12} \times 10^4 = 10^{16}$ B = 10 PB (Petabyte) = 0.01 EB (Exabyte)

$O(mn) \cong 10^{17}$ comparisons $\Rightarrow 10^8$ s (> 3 years), with 10^9 comparisons / s

Needle in a Haystack

- The brute force matcher is inefficient because it totally ignores information gained about the text T for one value of the shift s when it considers other values of s.
- Such information can be quite valuable, however.
- For example:
  - say pattern P = "aaab"
  - and we find that s = 0 is valid
  - then can the shift s = 1 be valid?
  - what about shifts s = 2 or s = 3?
  - none are valid, since T[4] = b.

Needle in a Haystack

- For a particular pattern and unpredictable data strings, preprocess the pattern so that searching for it in various data strings becomes faster

Analogy: Magnetize the needle

To improve Brute force method on String matching

KMP method - Knuth Morris-Pratt Algorithm

Why is KMP a Dynamic Programming Algorithm?

No, **KMP (Knuth-Morris-Pratt) algorithm is not a Dynamic Programming (DP) algorithm**. Instead, it is a **greedy** string matching algorithm that uses preprocessing to achieve efficient pattern searching.

However, **the LPS (Longest Prefix Suffix) table construction in KMP has similarities to DP**, because:

- It builds the solution **incrementally** for each prefix of the pattern.
- It stores previously computed values (prefix lengths) and reuses them, avoiding redundant computations.

Why is KMP a Dynamic Programming Algorithm?

But **KMP does not follow the classical DP structure** because:

1. **No explicit recurrence relation**: DP problems typically have a recurrence relation (e.g., `dp[i]` depends on previous states in a well-defined way). KMP's LPS table updates based on conditions rather than a recurrence formula.
2. **No optimal substructure property**: In DP, solutions to subproblems contribute optimally to the final solution. In KMP, we only use the LPS table for efficient skipping, not for building an optimal solution.
3. **No overlapping subproblems**: DP typically solves subproblems repeatedly and stores results to avoid recomputation. KMP's LPS table does not recompute values—it builds the prefix function iteratively in a single pass.

Why is KMP a Dynamic Programming Algorithm?

KMP is a **preprocessing-based greedy algorithm**, not a dynamic programming algorithm. However, the way the **LPS table** is built has a **DP-like feel** because it stores and reuses information.

KMP – Knuth Morris-Pratt Algorithm

KMP – Knuth Morris-Pratt Algorithm

- For a given pattern, we can find prefix or suffix of the pattern
- What is the **prefix**?
- Need to take subset of the alphabets given in pattern. But we must **start from the beginning** of the pattern only
- How many characters that we can take for a subset?
  - **as many as u want.**

Pattern:

a	b	c	d	a	b	c

Prefix: a, ab, abc, abcd, and more

KMP – Knuth Morris-Pratt Algorithm

- For a given pattern, we can find prefix or suffix of the pattern
- What is the **Suffix**?
- Need to take subset of the alphabets given in pattern. But we must start from the **right hand side** of the pattern only
- How many characters that we can take for a subset?
  - **as many as u want.**

Pattern:

a	b	c	d	a	b	c

Suffix: c, bc, abc, dabc and more

KMP – Knuth Morris-Pratt Algorithm

- Idea of KMP: inside the pattern,
- Is there any prefix same as suffix?

Pattern:

a	b	c	d	a	b	c

Prefix: a, ab, abc, abcd, and more

Suffix : c, bc, abc, dabc and more

KMP – Knuth Morris-Pratt Algorithm

- Idea of KMP: inside the pattern,
- Is there any prefix same as suffix?

Pattern:

a	b	c	d	a	b	c

Prefix: a, ab, abc, abcd, and more

Suffix: c, bc, abc, dabc and more

This is the prefix (abc) that also appears in suffix.

Idea of KMP: This is the observation what KMP algorithm will do on a pattern to avoid the no. of comparisons
i.e. Need to observe:- Is the beginning part of the pattern appearing again anywhere else in the pattern or not?

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

	0	1	2	3	4	5	6	7	8	9
	j	i								
P1:	a	b	c	d	a	b	e	a	b	f
LPS/failure():	0									

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

	0	1	2	3	4	5	6	7	8	9
	j	i								
P1:	a	b	c	d	a	b	e	a	b	f
LPS/failure():	0	0								

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
j		i							

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0								

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
j		i							

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0							

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
j			i						

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0							

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i **Index** always starts from 1:

0	1	2	3	4	5	6	7	8	9
j			i						

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0						

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i **Index** always starts from 1:

0	1	2	3	4	5	6	7	8	9
j				i					

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0						

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
	j			i					

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0						

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
	j			i					

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0	1					

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

	0	1	2	3	4	5	6	7	8	9
		j				i				
P1:	a	b	c	d	a	b	e	a	b	f
LPS/failure():	0	0	0	0	1					

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i **Index** always starts from 1:

	0	1	2	3	4	5	6	7	8	9
			j			i				
P1:	a	b	c	d	a	b	e	a	b	f
LPS/failure():	0	0	0	0	1					

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
		j			i				

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0	1	2				

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
		j				i			

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0	1	2				

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
		j				i			

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0	1	2				

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
		j				i			

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0	1	2				

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
j						i			

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0	1	2				

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
j						i			

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0	1	2	0			

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
j							i		

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0	1	2	0			

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
	j						i		

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0	1	2	0			

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
	j						i		

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0	1	2	0	1		

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
	j							i	

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0	1	2	0	1		

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
		j						i	

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0	1	2	0	1		

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
		j						i	

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0	1	2	0	1	2	

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
		j							i

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0	1	2	0	1	2	

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
		j							i

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0	1	2	0	1	2	

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
j									i

P1:

a	b	c	d	a	b	e	a	b	f

LPS/failure():

0	0	0	0	1	2	0	1	2	

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i Index always starts from 1:

0	1	2	3	4	5	6	7	8	9
j									i

P1:	a	b	c	d	a	b	e	a	b	f

LPS/failure():	0	0	0	0	1	2	0	1	2	0

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

i **Index** always starts from 1:

	0	1	2	3	4	5	6	7	8	9
	j									i
P1:	a	b	c	d	a	b	e	a	b	f
LPS/failure():	0	0	0	0	1	2	0	1	2	0

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.

```
if P[i] = P[j] then:
    j ← j + 1
    f[i] ← j
    i ← i + 1
else if j > 0 then:
    j ← f[j - 1]  // Backtrack using failure function
else:
    f[i] ← 0
    i ← i + 1
```

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

Index always **starts** from **1**:

	1	2								
P1:	a	b	c	d	a	b	e	a	b	f
LPS:	0	0	0	0	1	2	0	1	2	0

Short way to prepare LPS table

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.
0 – first time appears, since no match
The 2nd "ab" matches with 1 and 2 indices.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

Index always **starts** from **1**:

1	2								

P1:

a	b	c	d	a	b	e	a	b	f

LPS/failure():

0	0	0	0	1	2	0	1	2	0

Short way to prepare LPS table

The failure function $f(j)$ is computed as follows:

1. $f(0) = 0$ → Always 0
2. $f(1) = 0$ → No proper prefix which is also a suffix
3. $f(2) = 0$ → No proper prefix which is also a suffix
4. $f(3) = 0$ → No proper prefix which is also a suffix
5. $f(4) = 1$ → Prefix "**a**" is also a suffix
6. $f(5) = 2$ → Prefix "**ab**" is also a suffix
7. $f(6) = 0$ → No proper prefix which is also a suffix
8. $f(7) = 1$ → Prefix "**a**" is also a suffix
9. $f(8) = 2$ → Prefix "**ab**" is also a suffix
10. $f(9) = 0$ → No proper prefix which is also a suffix

KMP Approach For Pattern Search

```
Algorithm KMPFailureFunction(P):
    Input: String P (pattern) with m characters
    Output: Failure function f for P

    f[0] ← 0   // First character has failure value 0
    j ← 0      // Tracks length of the longest prefix suffix

    i ← 1
    while i < m do:
        if P[i] = P[j] then:
            j ← j + 1
            f[i] ← j
            i ← i + 1
        else if j > 0 then:
            j ← f[j - 1]  // Backtrack using failure function
        else:
            f[i] ← 0
            i ← i + 1
```

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

Index always starts from 1:	1	2	3							
P2:	a	b	c	d	e	a	b	f	a	b	c
LPS/failure():	0	0	0	0	0	1	2	0	1	2	3

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.
0 – first time appears, since no match
The 2nd "ab" matches with 1 and 2 indices.

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

Index always starts from 1:

1	2	3								

P2:

a	b	c	d	e	a	b	f	a	b	c

LPS/failure():

0	0	0	0	0	1	2	0	1	2	3

1. $f(0) = 0 \rightarrow$ Always 0
2. $f(1) = 0 \rightarrow$ No proper prefix which is also a suffix
3. $f(2) = 0 \rightarrow$ No proper prefix which is also a suffix
4. $f(3) = 0 \rightarrow$ No proper prefix which is also a suffix
5. $f(4) = 0 \rightarrow$ No proper prefix which is also a suffix
6. $f(5) = 1 \rightarrow$ Prefix "**a**" is also a suffix
7. $f(6) = 2 \rightarrow$ Prefix "**ab**" is also a suffix
8. $f(7) = 0 \rightarrow$ No proper prefix which is also a suffix
9. $f(8) = 1 \rightarrow$ Prefix "**a**" is also a suffix
10. $f(9) = 2 \rightarrow$ Prefix "**ab**" is also a suffix
11. $f(10) = 3 \rightarrow$ Prefix "**abc**" is also a suffix

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

Index always starts from 1:	1	2	3						
P3:	a	a	b	c	a	d	a	a	b	e
LPS/failure():	0	1	0	0	1	0	1	2	3	0

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.
0 – first time appears, since no match

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

Index always starts from 1:	1	2	3							
P3:	a	a	b	c	a	d	a	a	b	e
LPS/failure():	0	1	0	0	1	0	1	2	3	0

1. $f(0) = 0 \rightarrow$ Always 0
2. $f(1) = 1 \rightarrow$ Prefix "**a**" is also a suffix
3. $f(2) = 0 \rightarrow$ No proper prefix which is also a suffix
4. $f(3) = 0 \rightarrow$ No proper prefix which is also a suffix
5. $f(4) = 0 \rightarrow$ No proper prefix which is also a suffix
6. $f(5) = 1 \rightarrow$ Prefix "**a**" is also a suffix
7. $f(6) = 0 \rightarrow$ No proper prefix which is also a suffix
8. $f(7) = 1 \rightarrow$ Prefix "**a**" is also a suffix
9. $f(8) = 2 \rightarrow$ Prefix "**aa**" is also a suffix
10. $f(9) = 3 \rightarrow$ Prefix "**aab**" is also a suffix
11. $f(10) = 0 \rightarrow$ No proper prefix which is also a suffix

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

Index always starts from 1:	1	2	3						
P4:	a	a	a	a	b	a	a	c	d
LPS/failure():	0	1	2	3	0	1	2	0	0

3rd index a => matches with first “aa”
4th index a => matches with first “aaa”

a is appearing anywhere ?
Together with a, b is also appearing.
ab is appearing 3 times.

What do we do now?
Prepare a LPS table.
0 – first time appears, since no match

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

Index always starts from 1:	1	2	3						
P4:	a	a	a	a	b	a	a	c	d
LPS/failure():	0	1	2	3	0	1	2	0	0

3rd index a => matches with first “aa”
4th index a => matches with first “aaa”

1. $f(0) = 0 \rightarrow$ Always 0
2. $f(1) = 1 \rightarrow$ Prefix "**a**" is also a suffix
3. $f(2) = 2 \rightarrow$ Prefix "**aa**" is also a suffix
4. $f(3) = 3 \rightarrow$ Prefix "**aaa**" is also a suffix
5. $f(4) = 0 \rightarrow$ No proper prefix which is also a suffix
6. $f(5) = 1 \rightarrow$ Prefix "**a**" is also a suffix
7. $f(6) = 2 \rightarrow$ Prefix "**aa**" is also a suffix
8. $f(7) = 0 \rightarrow$ No proper prefix which is also a suffix
9. $f(8) = 0 \rightarrow$ No proper prefix which is also a suffix

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

- Example:
  - value of the KMP failure function:

j	0	1	2	3	4	5
$P[j]$	a	b	a	b	a	c
$f(j)$						

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

- Example:
  - value of the KMP failure function:

j	0	1	2	3	4	5
$P[j]$	a	b	a	b	a	c
$f(j)$	0	0	1	2	3	0

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

- Example:
  - value of the KMP failure function:

j	0	1	2	3	4	5
P[*j*]	a	b	a	b	a	c
f(*j*)	0	0	1	2	3	0

The failure function $f(j)$ (or LPS array) is computed as follows:

1. $f(0) = 0$ (Always 0)
2. $f(1) = 0$ (No proper prefix which is also a suffix)
3. $f(2) = 1$ (Prefix "**a**" is also a suffix)
4. $f(3) = 2$ (Prefix "**ab**" is also a suffix)
5. $f(4) = 3$ (Prefix "**aba**" is also a suffix)
6. $f(5) = 0$ (No proper prefix which is also a suffix for "**ababac**")

KMP – Knuth Morris-Pratt Algorithm

- In KMP algor, For a pattern, we generate a π table or LPS(Longest Prefix/Suffix) table

- Example:
  - value of the KMP failure function:

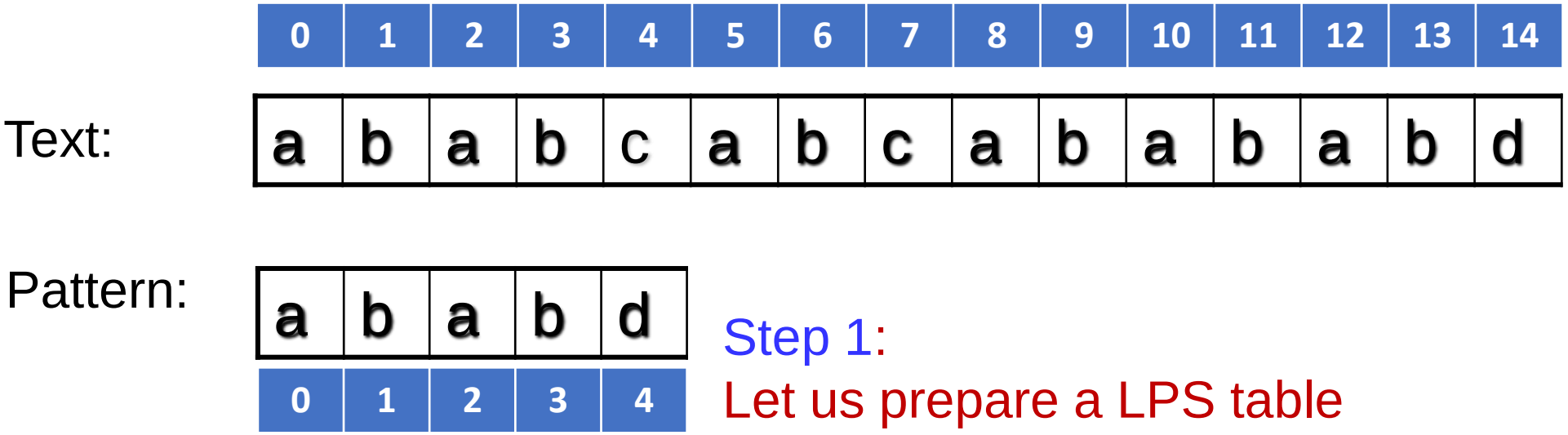
j	0	1	2	3	4	5
P[*j*]	a	b	a	b	a	c
f(*j*)	0	0	1	2	3	0

The failure function $f(j)$ (or LPS array) is computed as follows:

1. $f(0) = 0$ (Always 0)
2. $f(1) = 0$ (No proper prefix which is also a suffix)
3. $f(2) = 1$ (Prefix "**a**" is also a suffix)
4. $f(3) = 2$ (Prefix "**ab**" is also a suffix)
5. $f(4) = 3$ (Prefix "**aba**" is also a suffix)
6. $f(5) = 0$ (No proper prefix which is also a suffix for "**ababac**")

Working of KMP – Knuth Morris-Pratt Algorithm

- Explanation is based on tracing of algorithm.

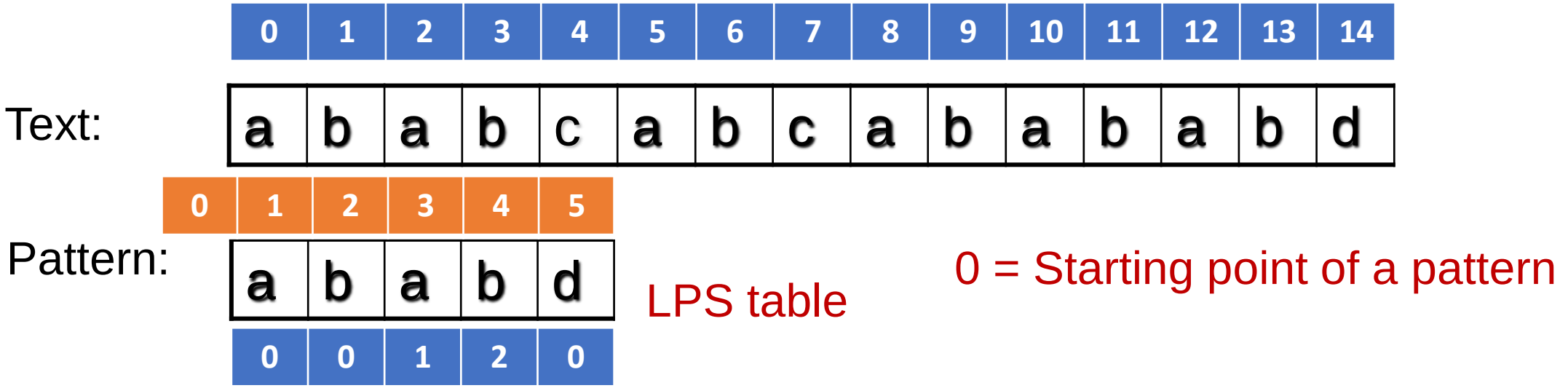


Step 1:

Let us prepare a LPS table

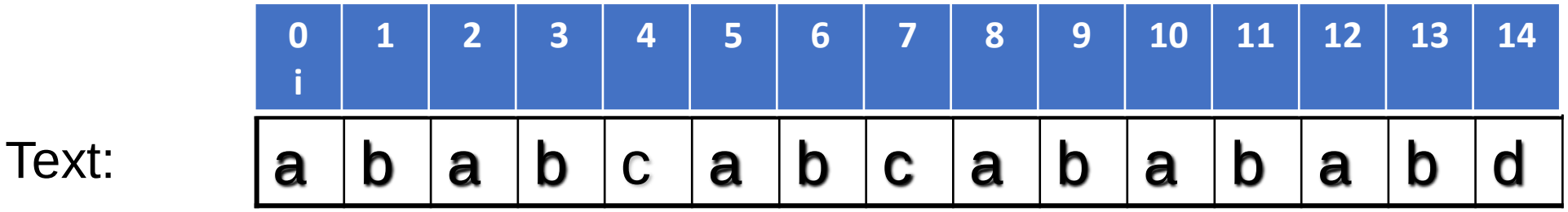
Working of KMP – Knuth Morris-Pratt Algorithm

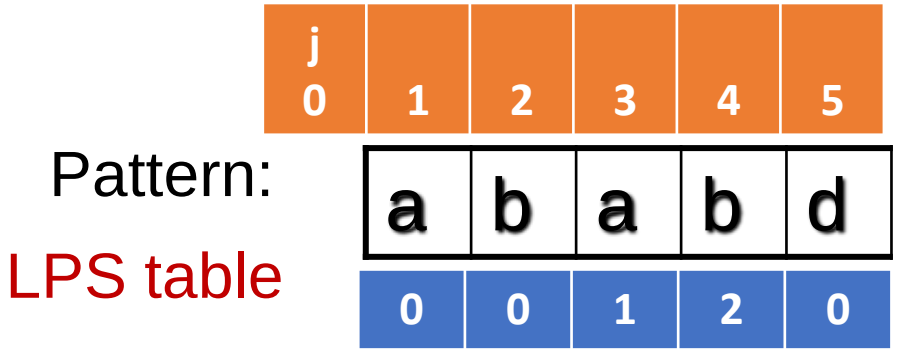
- Explanation is based on tracing of algorithm.



Working of KMP – Knuth Morris-Pratt Algorithm

- Explanation is based on tracing of algorithm.

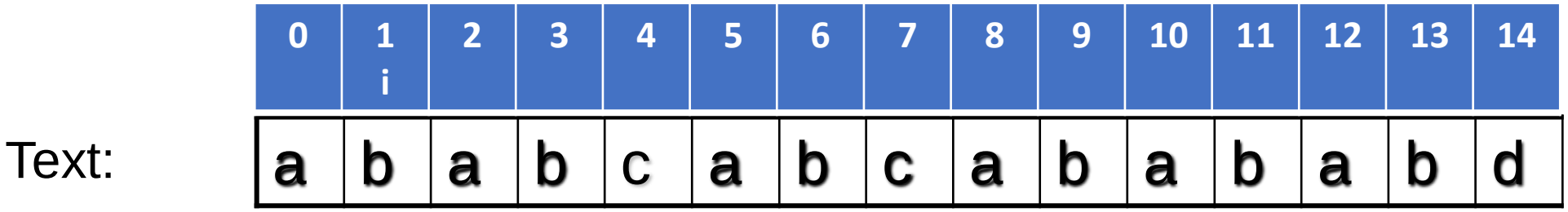


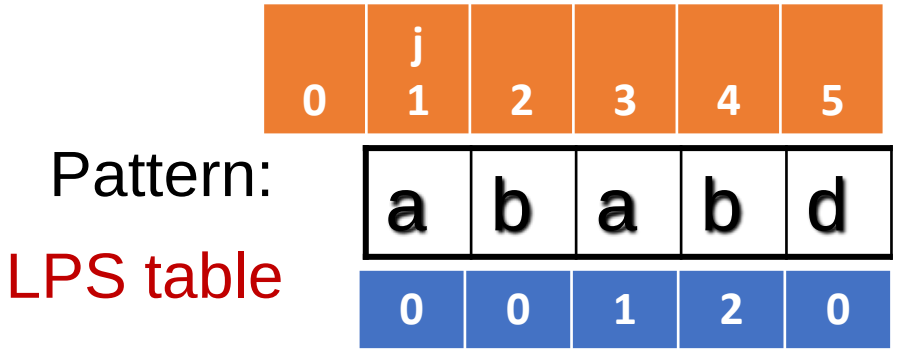


Compare char @(i) and char@(j+1)
If match, inc i and j

Working of KMP – Knuth Morris-Pratt Algorithm

- Explanation is based on tracing of algorithm.

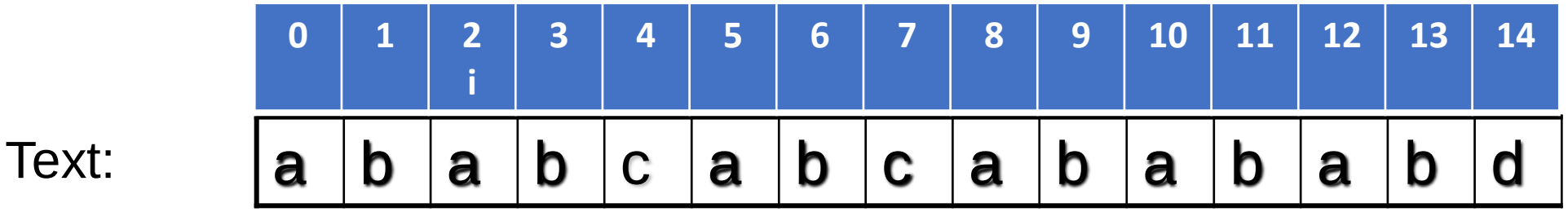


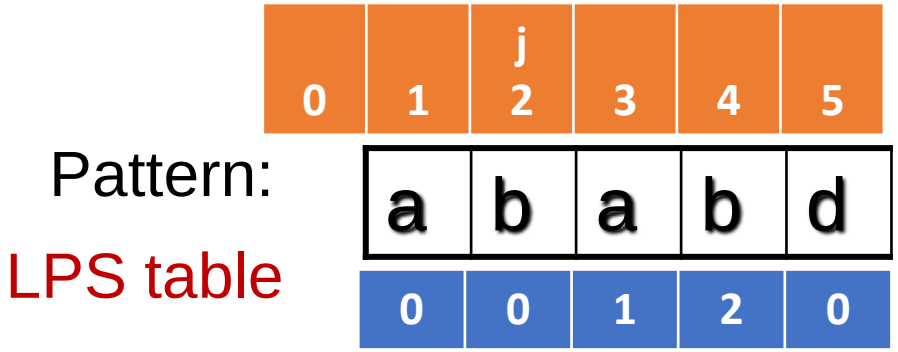


Compare char @(i) and char@(j+1)
If match, inc i and j

Working of KMP – Knuth Morris-Pratt Algorithm

- Explanation is based on tracing of algorithm.

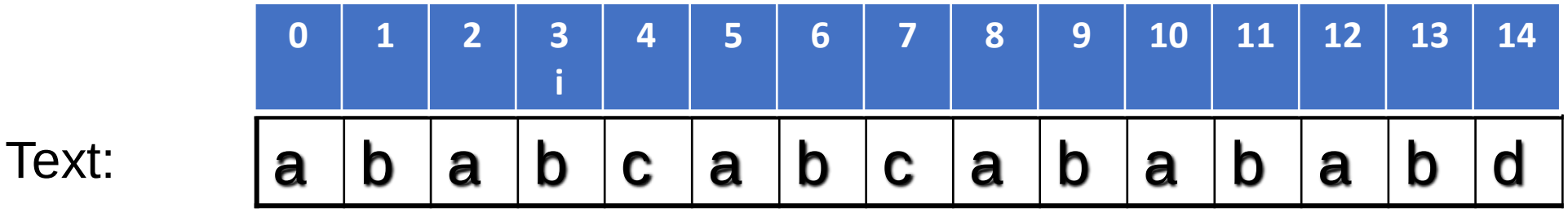


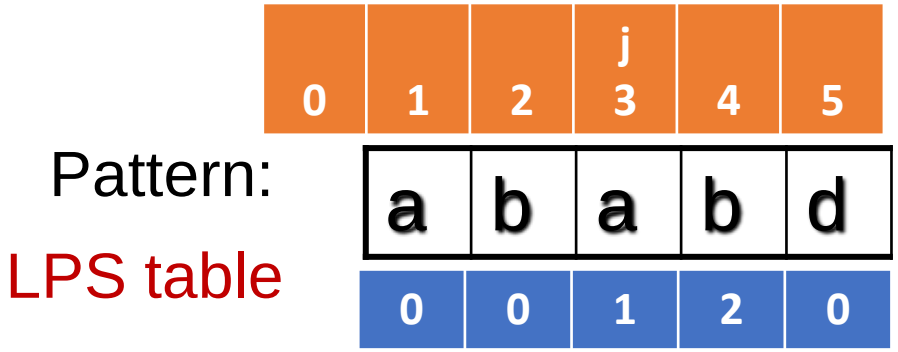


Compare char @(i) and char@(j+1)
If match, inc i and j

Working of KMP – Knuth Morris-Pratt Algorithm

- Explanation is based on tracing of algorithm.

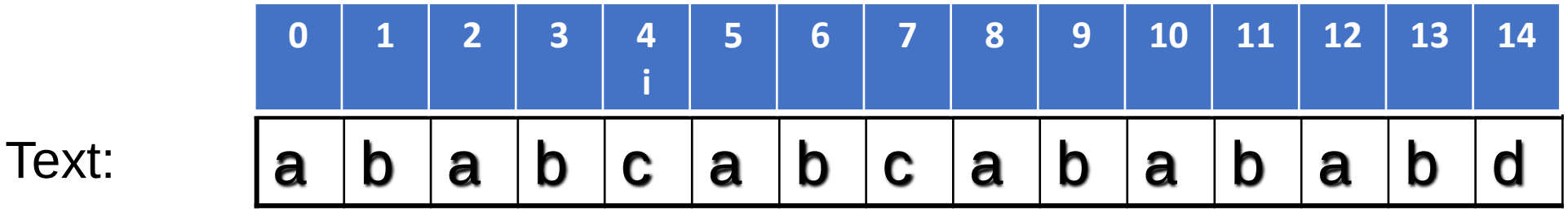


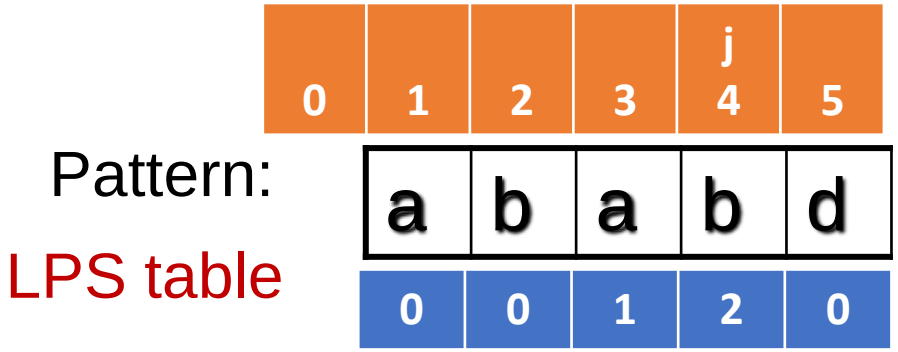


Compare char @(i) and char@(j+1)
If match, inc i and j

Working of KMP – Knuth Morris-Pratt Algorithm

- Explanation is based on tracing of algorithm.





Compare char @(i) and char@(j+1)
If there is mismatch, then
Move j to table index (here = 2)

Working of KMP – Knuth Morris-Pratt Algorithm

- Explanation is based on tracing of algorithm.

Text:

0	1	2	3	4 i	5	6	7	8	9	10	11	12	13	14
a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern:

0	1	j 2	3	4	5
a	b	a	b	d	

LPS table

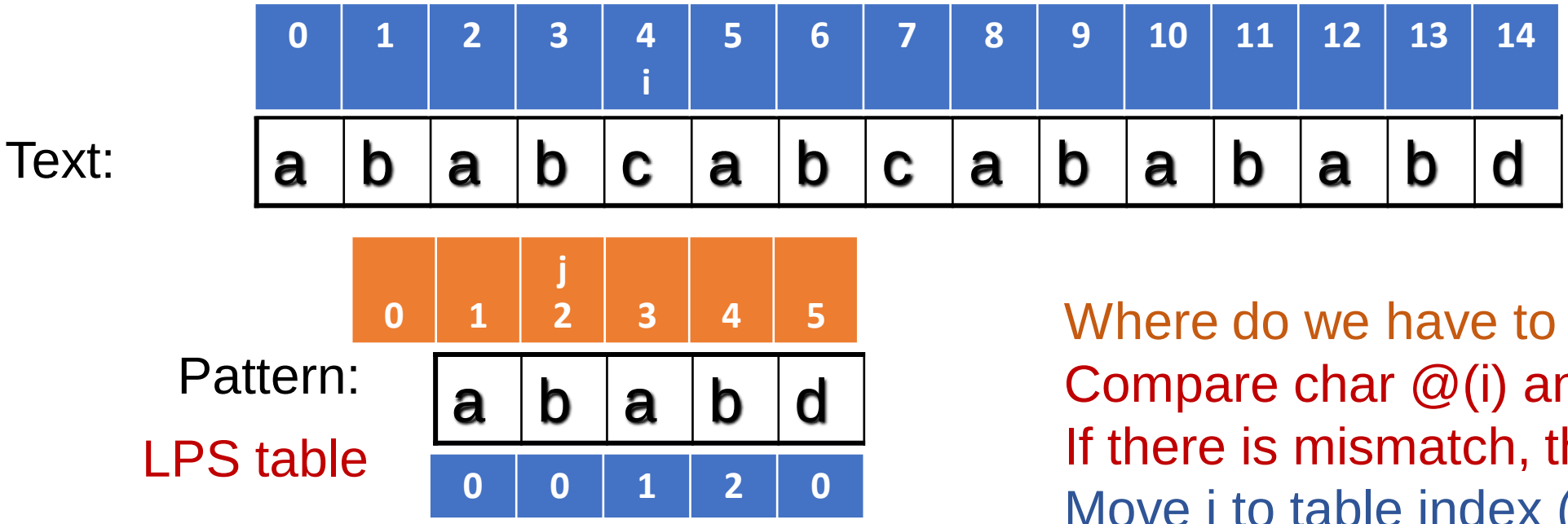
0	0	1	2	0

Compare chat @(i) and char@(j+1)
If there is mismatch, then
Move j to table index (here = 2).

Means, “abab” is repeating. “ab” came 2 times. Already we checked 1 time.
Where do we have to start?

Working of KMP – Knuth Morris-Pratt Algorithm

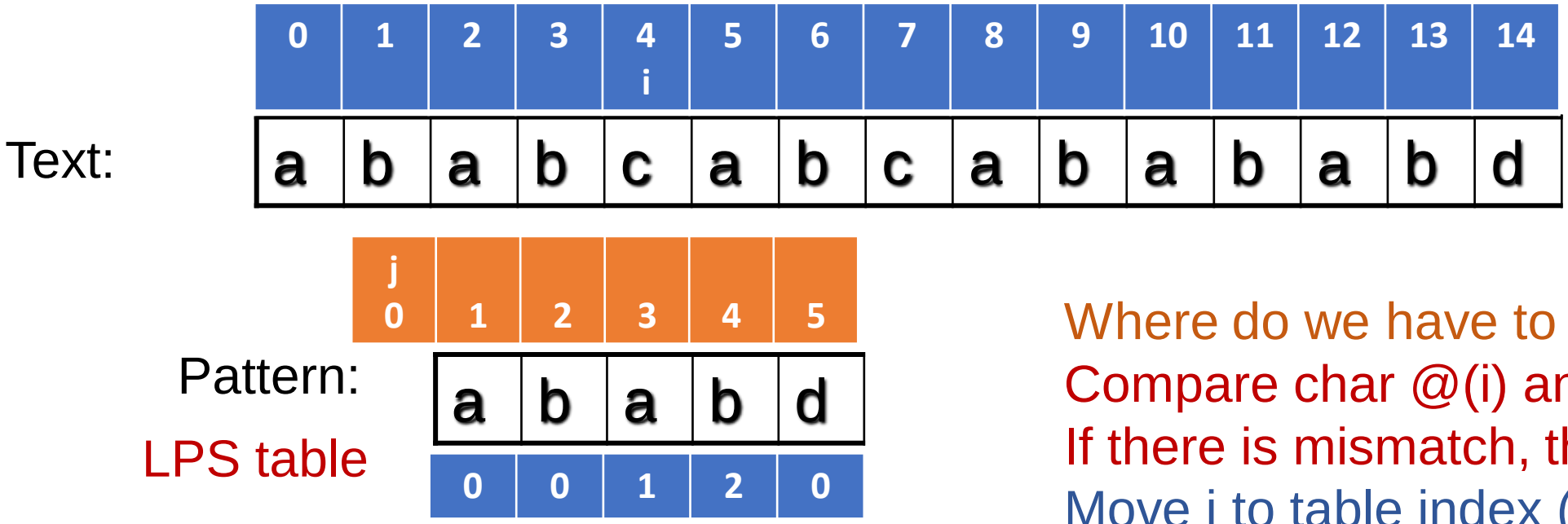
- Explanation is based on tracing of algorithm.



Where do we have to start?
Compare char @(i) and char@(j+1)
If there is mismatch, then
Move j to table index (here = **0**).

Working of KMP – Knuth Morris-Pratt Algorithm

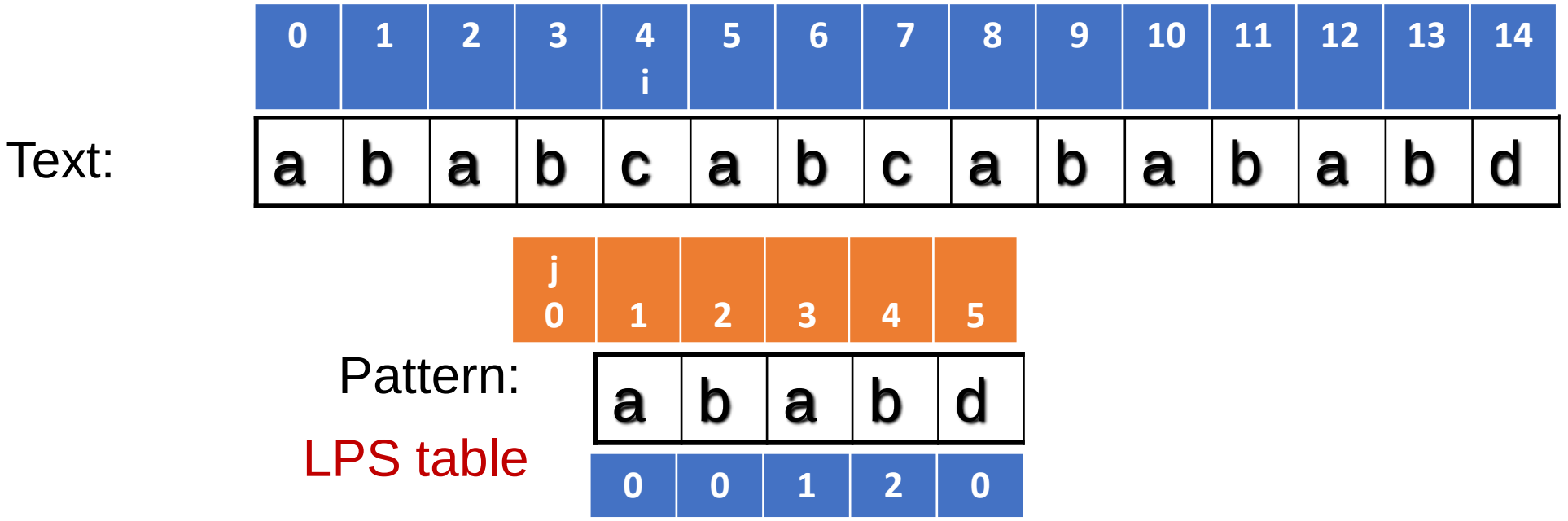
- Explanation is based on tracing of algorithm.



Where do we have to start?
Compare char @(i) and char@(j+1)
If there is mismatch, then
Move j to table index (here = **0**).

Working of KMP – Knuth Morris-Pratt Algorithm

- Explanation is based on tracing of algorithm.



Where do we have to start?

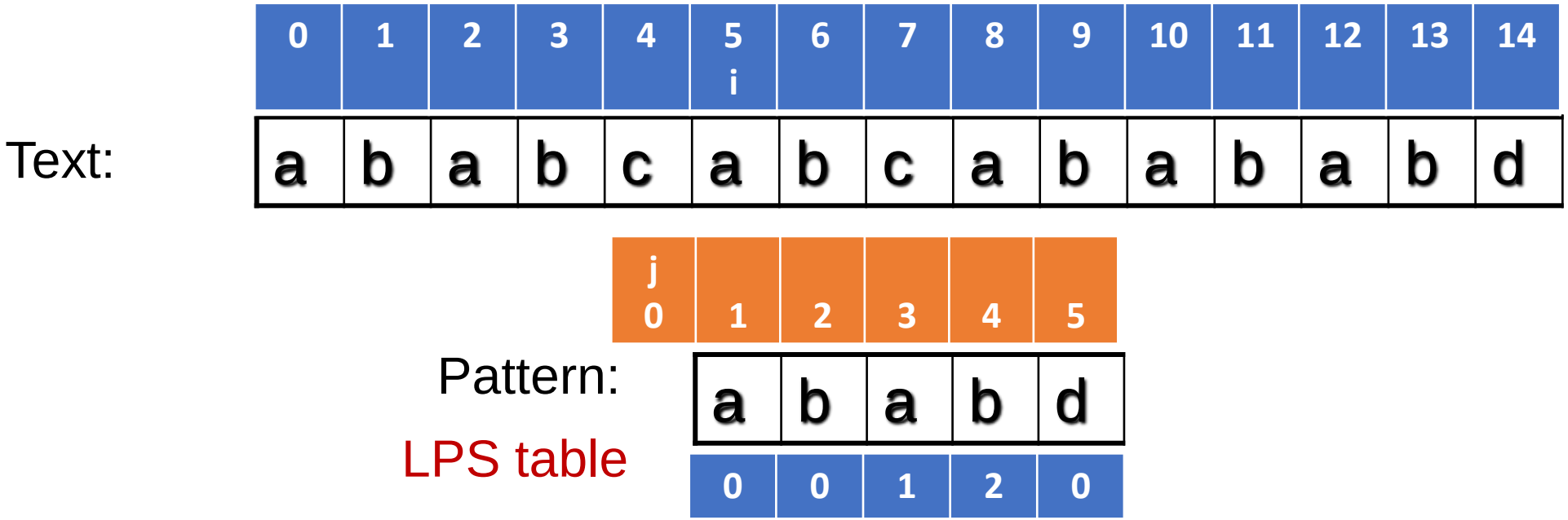
Compare char @(i) and char@(j+1)

Still there is a Mismatch, But **j** is on 0. So we can not move j to its left anymore.

So move **i** to its right.

Working of KMP – Knuth Morris-Pratt Algorithm

- Explanation is based on tracing of algorithm.



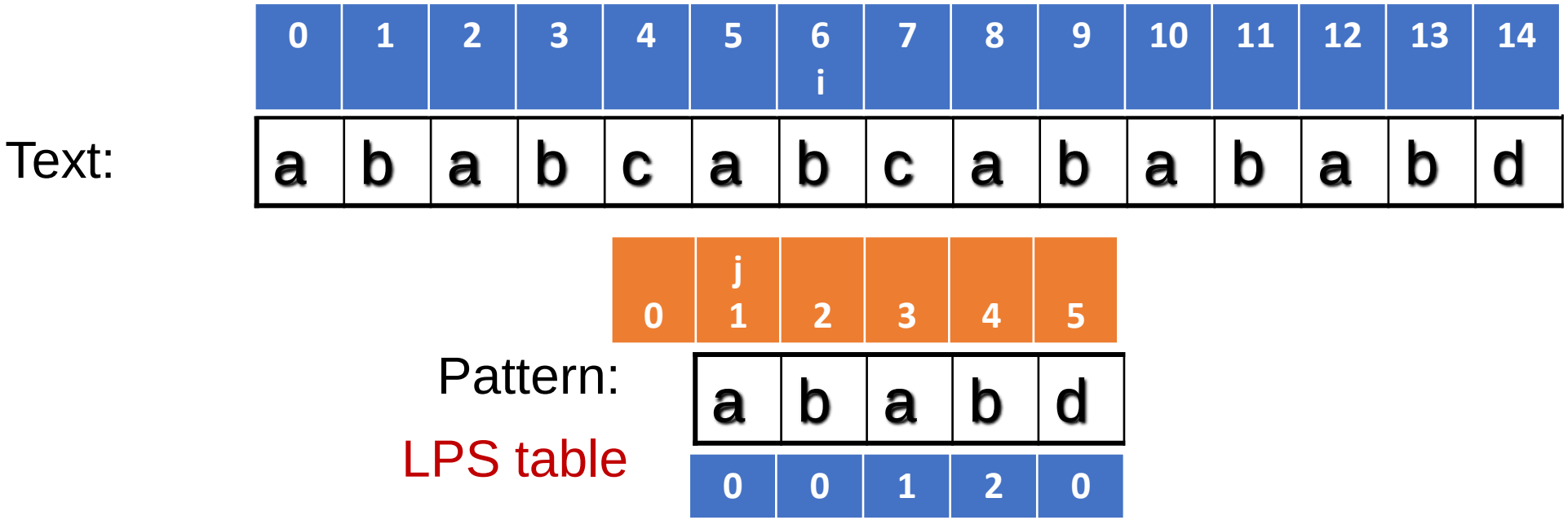
In case of mismatch, Only j is moving and i is not backtracking. i.e. i has moved to the next alphabet.

Compare char @(i) and char@(j+1)

Match found. So move i and j

Working of KMP – Knuth Morris-Pratt Algorithm

- Explanation is based on tracing of algorithm.



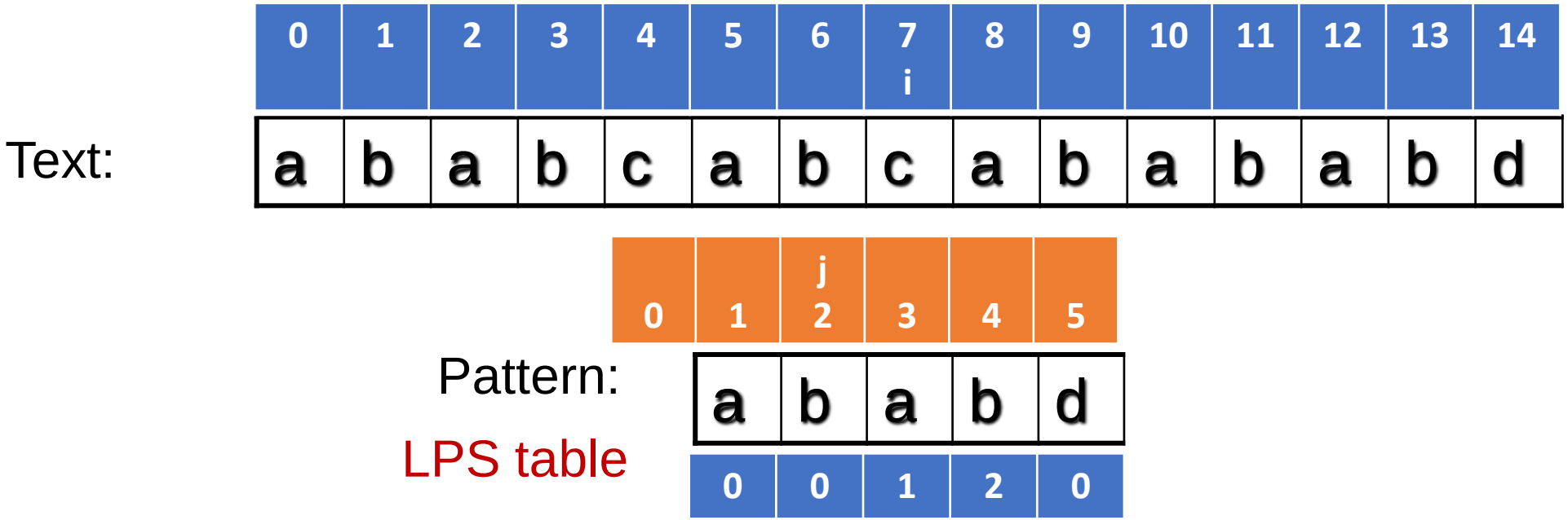
In case of mismatch, Only j is moving and i is not backtracking. i.e. i has moved to the next alphabet.

Compare char @(i) and char@(j+1)

Again, Match found. So move i and j

Working of KMP – Knuth Morris-Pratt Algorithm

- Explanation is based on tracing of algorithm.



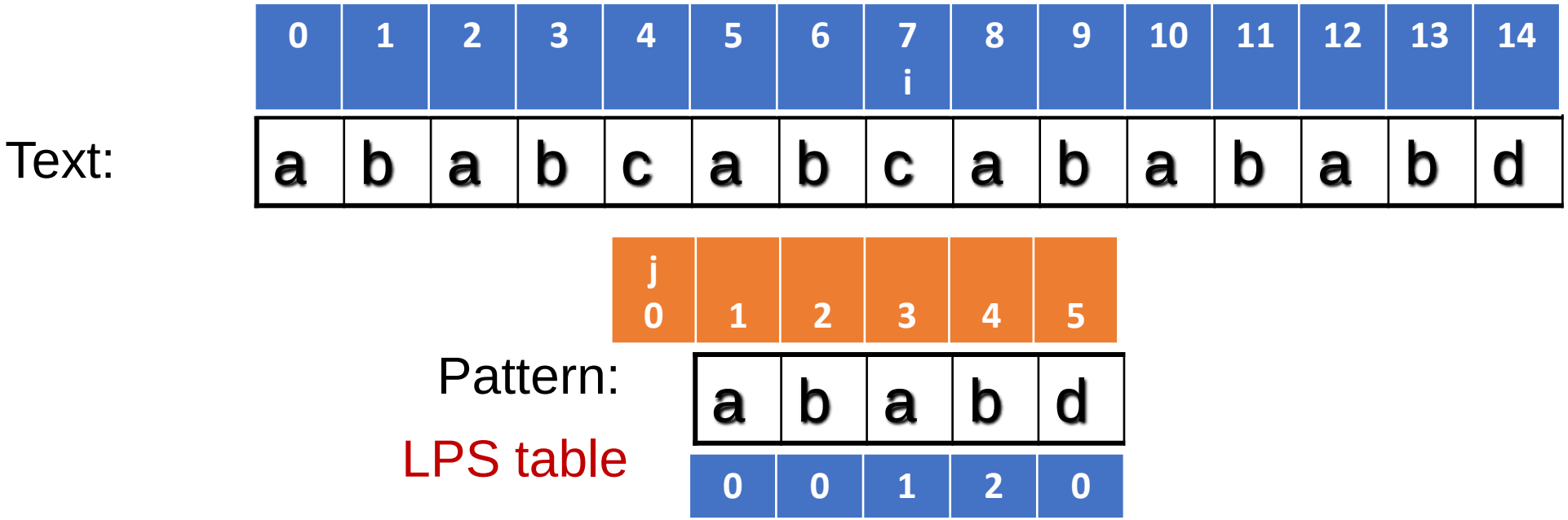
In case of mismatch, Only j is moving and i is not backtracking. i.e. i has moved to the next alphabet.

Compare char @(i) and char@(j+1)

MisMatch found. So move j to table index (here = 0)

Working of KMP – Knuth Morris-Pratt Algorithm

- Explanation is based on tracing of algorithm.



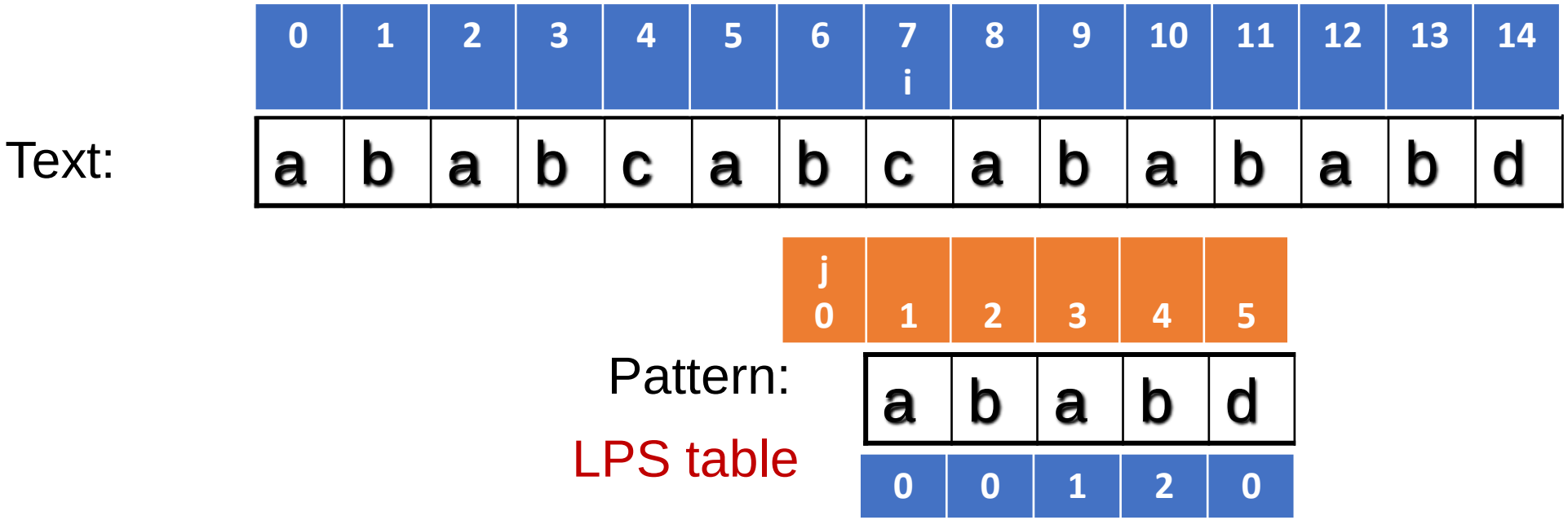
In case of mismatch, Only j is moving and i is not backtracking. i.e. i has moved to the next alphabet.

Compare char @(i) and char@(j+1)

MisMatch found. So move j to table index (here = 0)

Working of KMP – Knuth Morris-Pratt Algorithm

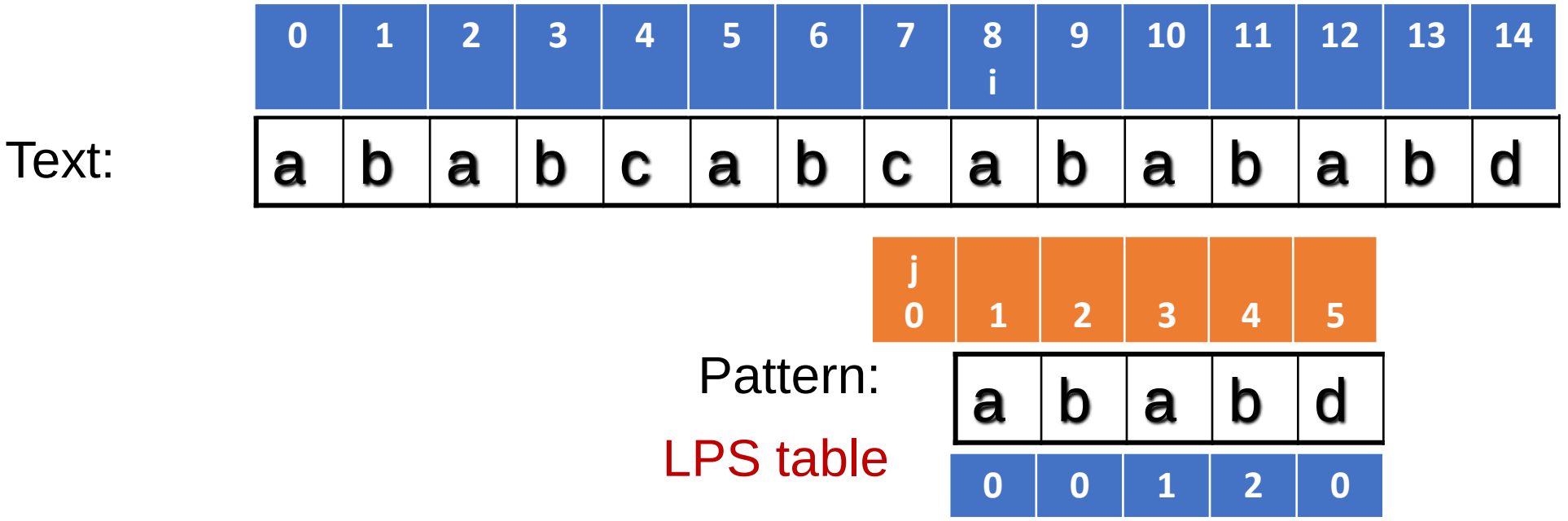
- Explanation is based on tracing of algorithm.



Compare char @(i) and char@(j+1)
MisMatch found. But **j** is on 0. So we can not move j to its left anymore.
So move **i** to its right.

Working of KMP – Knuth Morris-Pratt Algorithm

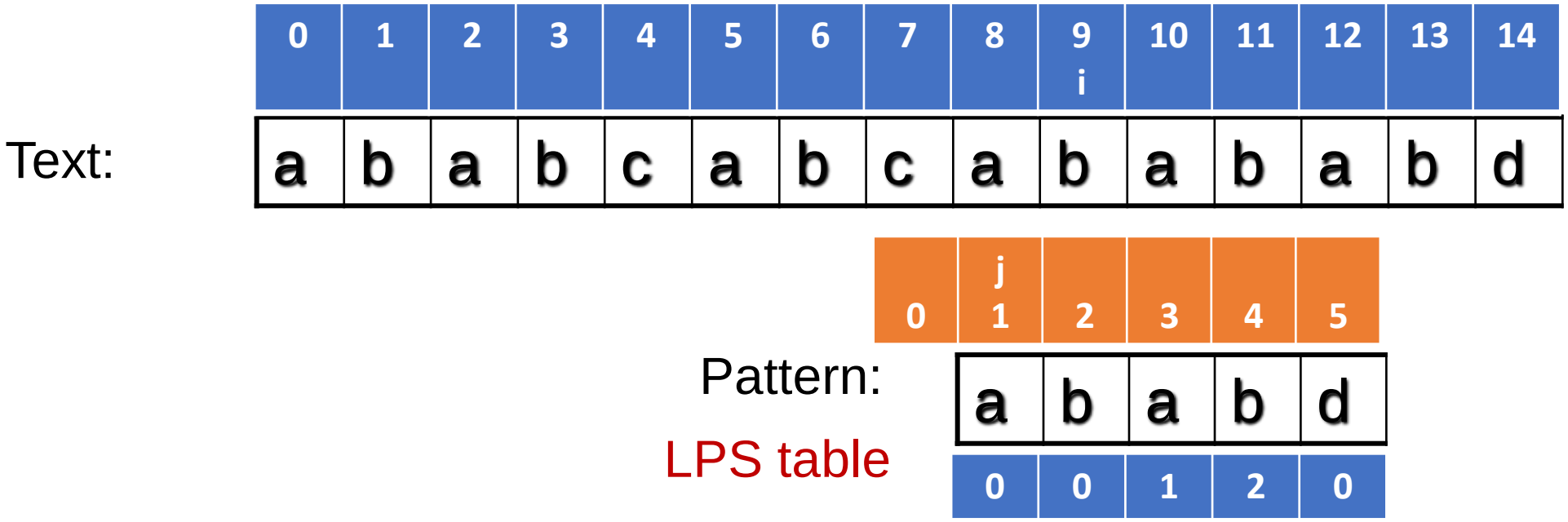
- Explanation is based on tracing of algorithm.



Compare char @(i) and char@(j+1)
Match found. Move i and j

Working of KMP – Knuth Morris-Pratt Algorithm

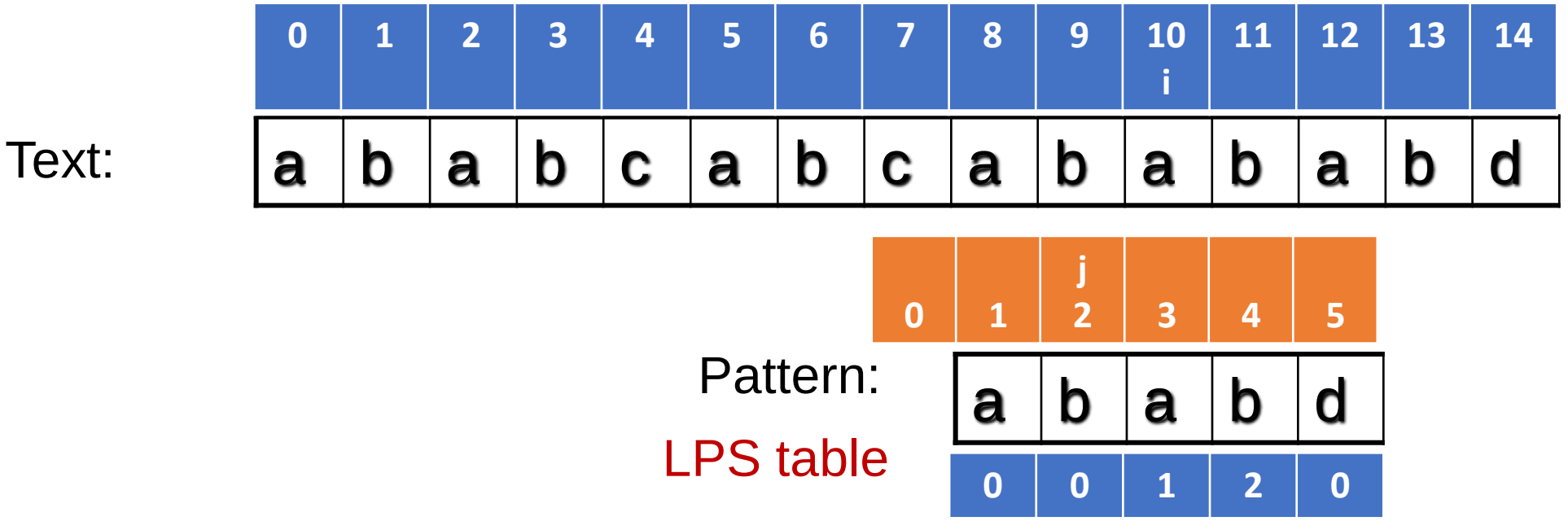
- Explanation is based on tracing of algorithm.



Compare char @(i) and char@(j+1)
Match found. Move i and j

Working of KMP – Knuth Morris-Pratt Algorithm

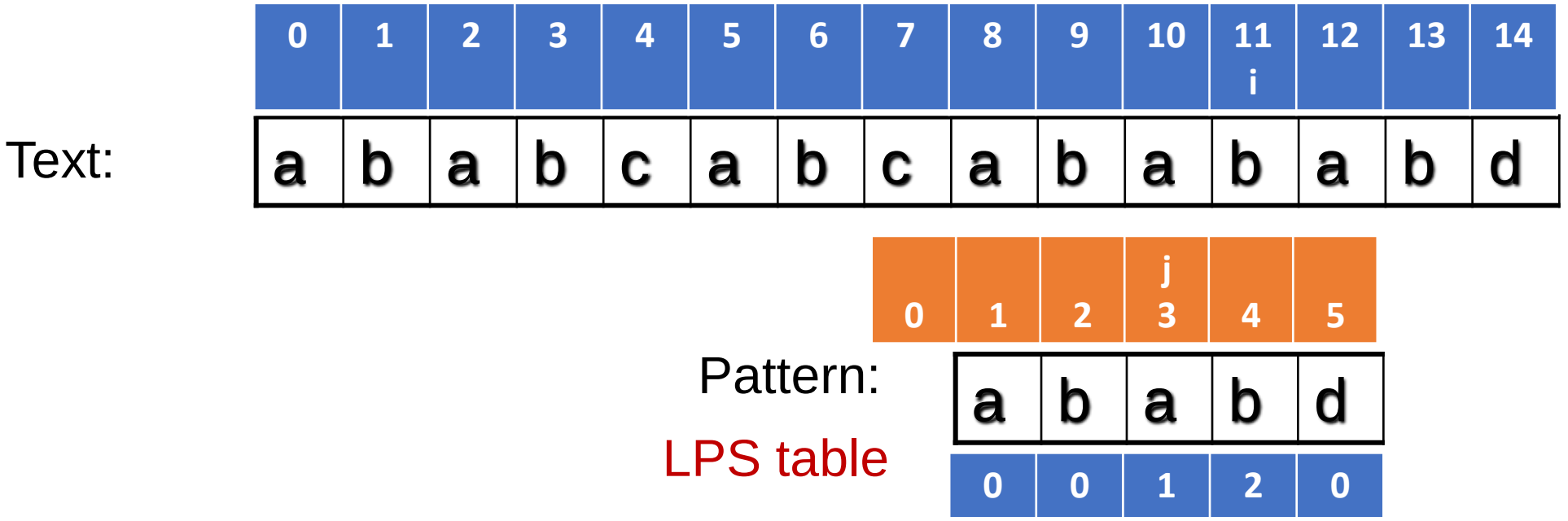
- Explanation is based on tracing of algorithm.



Compare char @(i) and char@(j+1)
Match found. Move i and j

Working of KMP – Knuth Morris-Pratt Algorithm

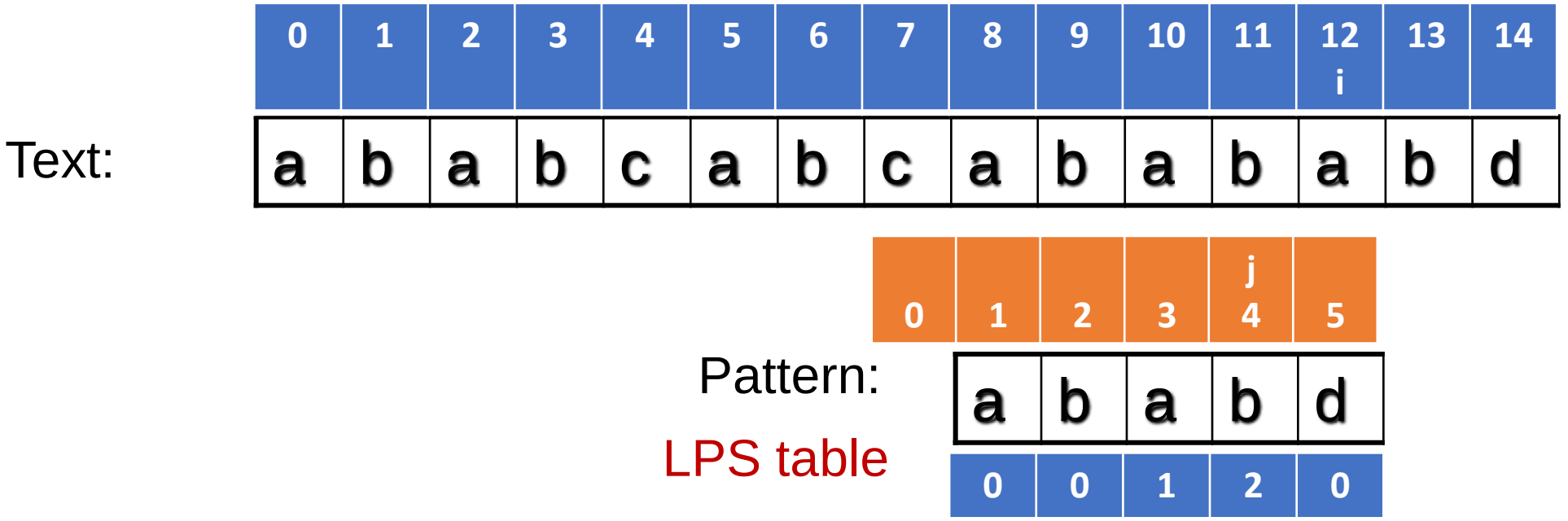
- Explanation is based on tracing of algorithm.



Compare char @(i) and char@(j+1)
Match found. Move i and j

Working of KMP – Knuth Morris-Pratt Algorithm

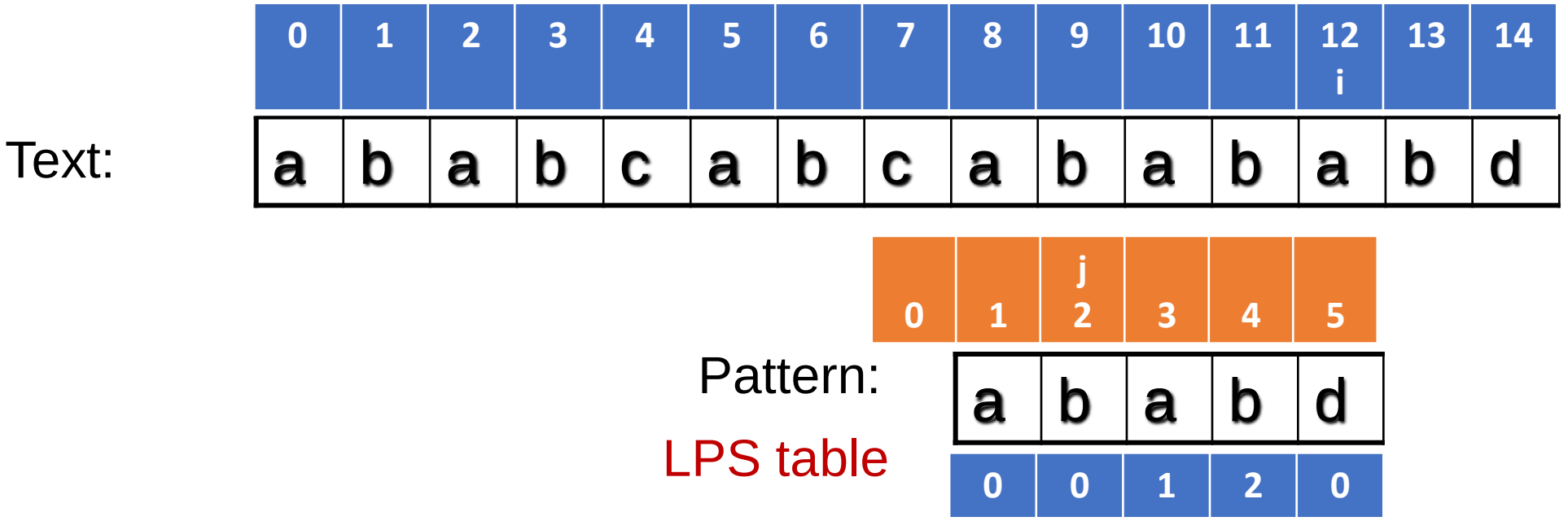
- Explanation is based on tracing of algorithm.



Compare char @(i) and char@(j+1)
MisMatch found. So move j to table index (here = 2)

Working of KMP – Knuth Morris-Pratt Algorithm

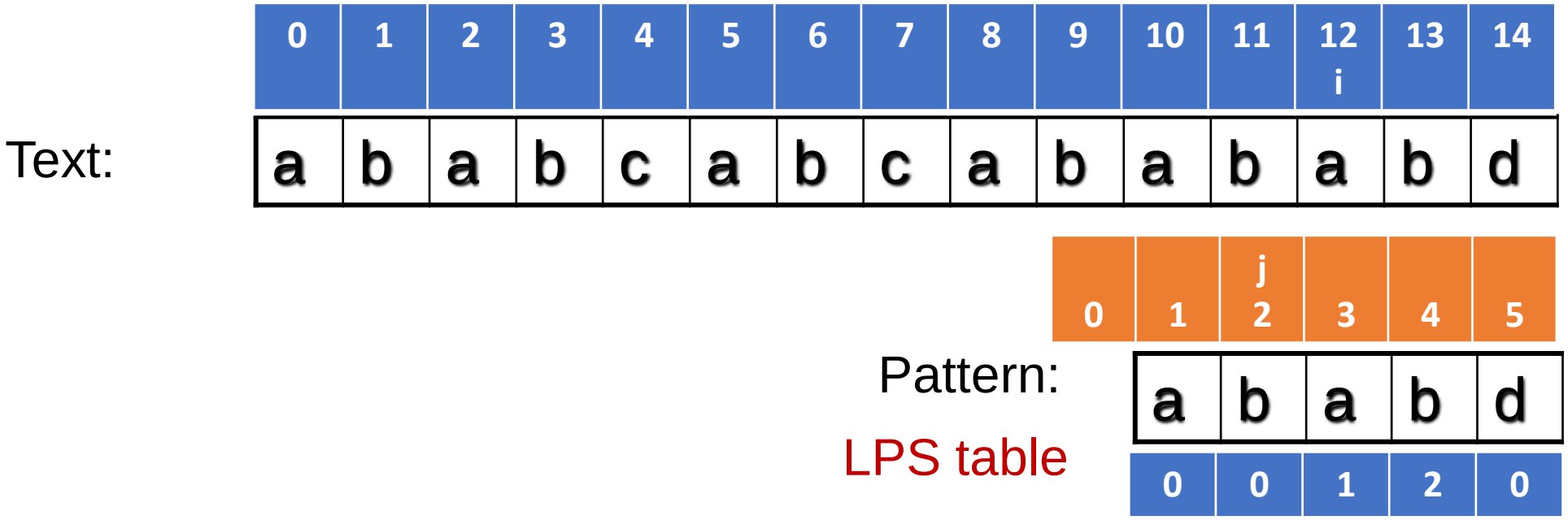
- Explanation is based on tracing of algorithm.



Compare char @(i) and char@(j+1)

Working of KMP – Knuth Morris-Pratt Algorithm

- Explanation is based on tracing of algorithm.

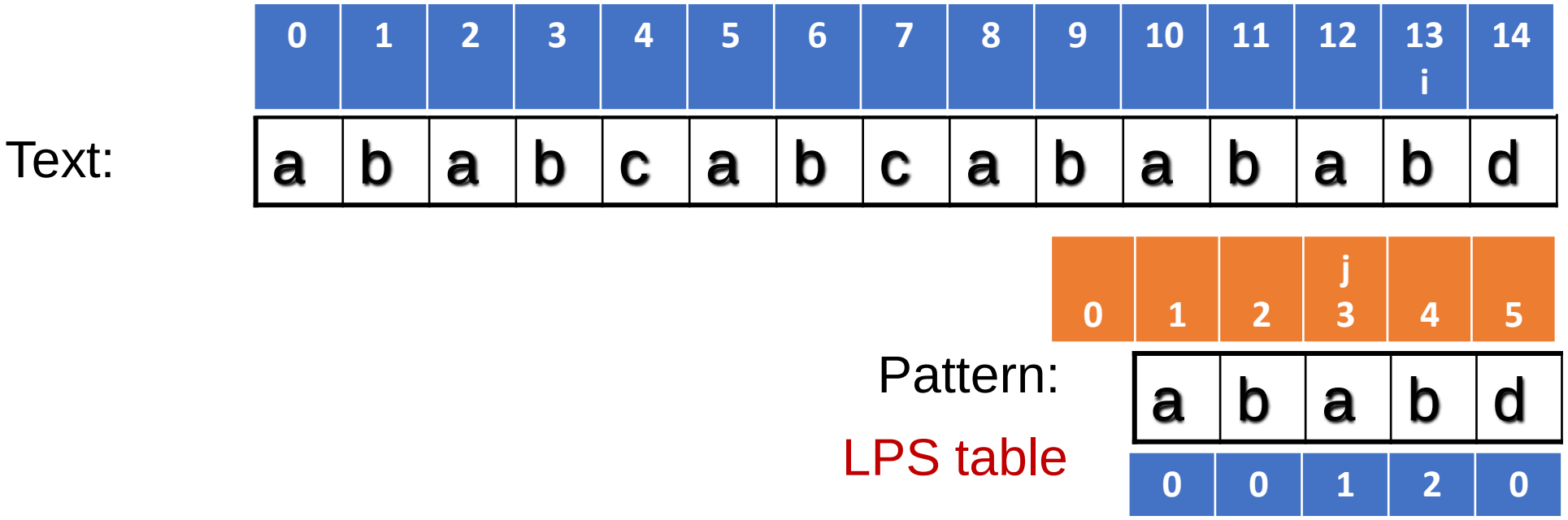


Since “ab” have come 2 times, we can skip the first time, we can start searching from the 2nd time onwards

Compare char @(i) and char@(j+1)
Match found. Move i and j

Working of KMP – Knuth Morris-Pratt Algorithm

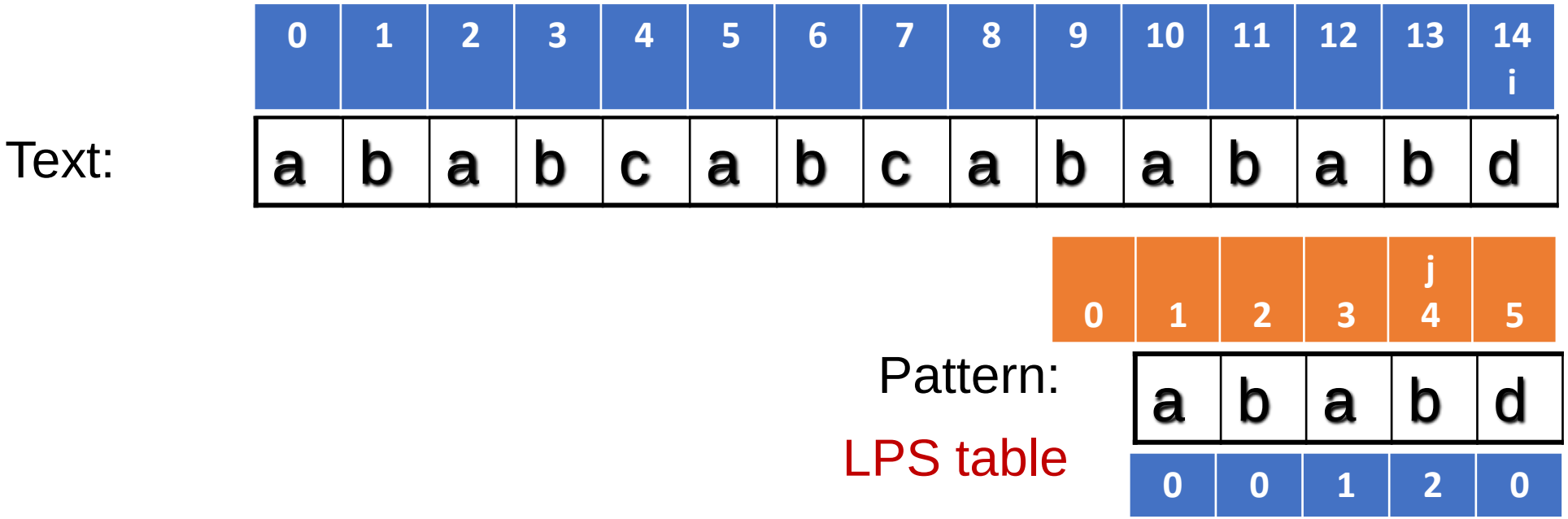
- Explanation is based on tracing of algorithm.



Compare char @(i) and char@(j+1)
Match found. Move i and j

Working of KMP – Knuth Morris-Pratt Algorithm

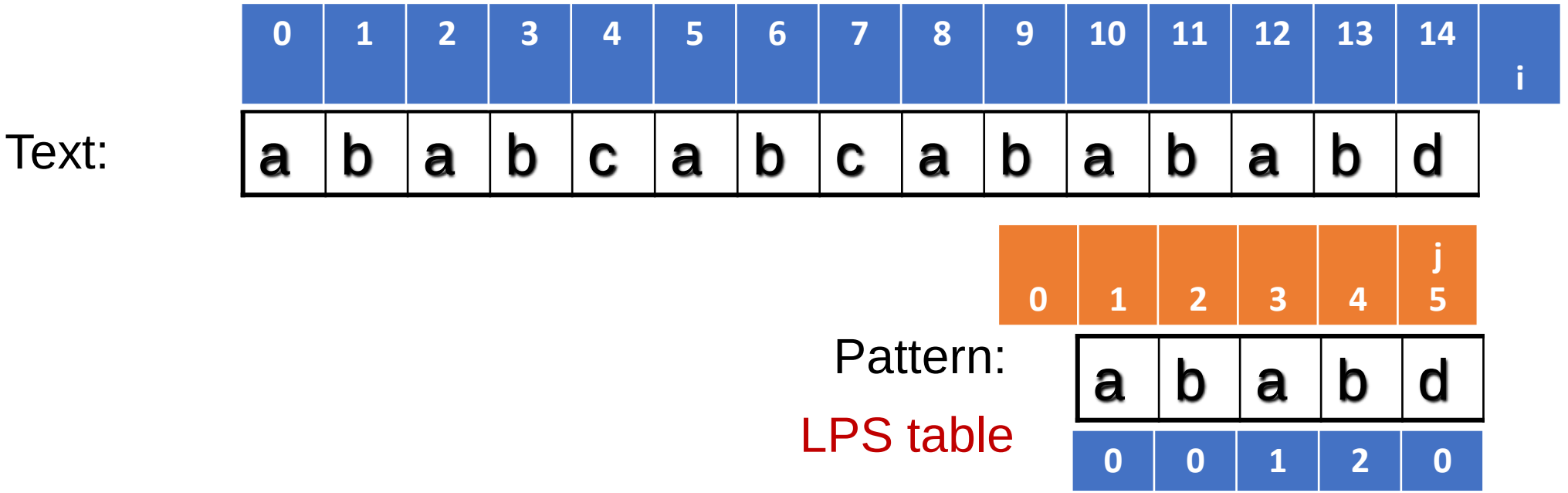
- Explanation is based on tracing of algorithm.



Compare char @(i) and char@(j+1)
Match found. Move i and j

Working of KMP – Knuth Morris-Pratt Algorithm

- Explanation is based on tracing of algorithm.



Since ‘i' has reached the end of the string,
Also, we reached the end of the pattern, So now Pattern available inside the string.

Idea behind KMP

- If the “beginning part” of the string is appearing again somewhere else in the string, then **comparison of characters** can be avoided (don’t compare the characters again)
- ‘i’ need not to be moved back again (Don’t move ‘i’ back again). [‘i’ is moved only in the basic naïve brute force method]

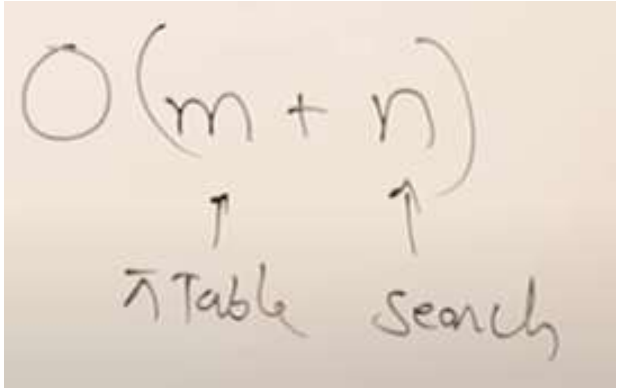
- Moving i only in the forward direction , never backtrack i

KMP - TimeComplexity

	0	1	2	3	4	5	6	7	8	9	10	11	12	13 i	14
Text:	a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Pattern:	a	b	a	b	d
	0	0	1	2	0

1. How many times that you are passing Text? 1 time
2. How many alphabets ? **n**
3. For preparing a table, parse through the pattern.
   How many alphabets in pattern? **m**



Naive Approach(Brute Force) For Pattern Search

```
void Patternsearch(char* pat, char* str) {
    int M = strlen(pat);
    int N = strlen(str);
    for (int i = 0; i <= N - M; i++) {
        int j;

        for (j = 0; j < M; j++)
            if (str[i + j] != pat[j])
                break;

        if (j == M)
            printf("Pattern found at index %d \n", i);
    }
}
```

KMP Approach For Pattern Search

```
Algorithm KMPFailureFunction(P):
    Input: String P (pattern) with m characters
    Output: Failure function f for P

    f[0] ← 0   // First character has failure value 0
    j ← 0      // Tracks length of the longest prefix suffix

    i ← 1
    while i < m do:
        if P[i] = P[j] then:
            j ← j + 1
            f[i] ← j
            i ← i + 1
        else if j > 0 then:
            j ← f[j - 1]  // Backtrack using failure function
        else:
            f[i] ← 0
            i ← i + 1
```

KMP Approach For Pattern Search (cont…)

```
Algorithm KMPMatch(T, P)
Input:
    - String T (text) of length n
    - String P (pattern) of length m
Output:
    - Index of the first occurrence of P in T, or -1 if P is not a substring of T.

1. Compute failure function f ← KMPFailureFunction(P)
2. i ← 0  (index for T)
3. j ← 0  (index for P)
4. while i < n do:
      a. if P[j] = T[i] then:
            - if j = m - 1 then:
                  return i - m + 1   {Pattern found}
            - i ← i + 1
            - j ← j + 1
      b. else if j > 0 then:
            - j ← f[j - 1]  {Use failure function to skip comparisons}
      c. else:
            - i ← i + 1
5. return -1  {Pattern not found}
```

KMP Algorithm : C++ Implementation

```
// Compute the LPS array (Unchanged)
void computeLPSArray(string pattern, vector<int>& lps) {
    int m = pattern.length();
    int length = 0;  // Length of the previous longest prefix
suffix
    lps[0] = 0;      // First element is always 0
    int i = 1;
```

```
while (i < m) {
        if (pattern[i] == pattern[length]) {
            length++;
            lps[i] = length;
            i++;
        } else {
            if (length != 0) {
                length = lps[length - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}
```

KMP Algorithm : C++ Implementation (cont…)

```
// Modified KMP Search function with j+1 indexing
void KMPSearch(string text, string pattern) {
    int n = text.length();
    int m = pattern.length();
    vector<int> lps(m, 0);

    // Compute LPS array
    computeLPSArray(pattern, lps);

    int i = 1, j = 0;  // Start i at 1, j at 0

while (i < n) {
        while (j > 0 && pattern[j] != text[i]) {
            j = lps[j - 1];  // Backtrack using LPS
        }

        if (pattern[j] == text[i]) {
            j++;  // Move j forward
        }

        if (j == m) {  // Full match found
            cout << "Pattern found at index " << (i - j
+ 1) << endl;
            j = lps[j - 1];  // Correctly update j
        }
        i++; // Move to next character in text
    }
}
```

Drawbacks of Brute force

1. Redundant Comparisons

- **Brute Force:**
  - After a mismatch, it **resets the search** and compares characters again from the beginning of the pattern.
- **KMP:**
  - Uses the **LPS table** to skip unnecessary comparisons, avoiding redundant checks.

📌 **Example:**

Searching `"abab"` in `"abababab"`

- Brute force checks every position.
- KMP skips over repeated calculations using LPS.

Drawbacks of Brute force (contn...)

2. High Time Complexity in Worst Case

- **Brute Force:**
  - Worst-case time complexity: **O(n × m)** (where `n` is text length, `m` is pattern length).
  - In the worst case (e.g., searching `"aaaa"` in `"aaaaaaaaaaaa"`), the algorithm **restarts the comparison** from the next position every time a mismatch occurs.
- **KMP:**
  - Worst-case time complexity: **O(n + m)**.
  - Uses the **LPS (Longest Prefix Suffix) table** to avoid redundant comparisons, making it much faster.

📌 **Example:**

Searching `"abc"` in `"aaaaabc"`

- **Brute Force:** Will retry all comparisons when a mismatch happens.
- **KMP:** Uses previous comparisons to intelligently shift the pattern.

Drawbacks of Brute force (contn…)

3. Inefficiency with Large Text and Pattern

- **Brute Force:**
  - Performs **unoptimized shifting**, leading to **slow performance on large inputs**.
- **KMP:**
  - Uses **preprocessing (LPS table)**, making it **faster on large texts**.

📌 **Example:**

Searching `"ABABAC"` in `"ABABABABABABABAC"`

- **Brute Force:** May perform nearly `O(n × m)` comparisons.
- **KMP:** Will intelligently shift the pattern based on LPS values.

Drawbacks of Brute force (contn…)

4. Not Suitable for Real-World Large Text Searches

- **Brute Force:**
  - **Inefficient** for searching in **large texts** (e.g., DNA sequences, logs, books).
- **KMP:**
  - **Efficient** for searching in large datasets (e.g., substring search in documents, bioinformatics, and search engines).

5. Brute Force Cannot Utilize Previous Knowledge

- **Brute Force:**
  - Does **not** use any preprocessing. Every mismatch forces the algorithm to start fresh.
- **KMP:**
  - Precomputes **LPS table** once, then reuses it to **optimize pattern shifts**.

Final Comparison Table

Feature	Brute Force	KMP Algorithm
Worst-case Time Complexity	O(n × m)	O(n + m)
Redundant Comparisons	Yes (Resets search on mismatch)	No (Uses LPS table)
Efficiency for Large Text	Slow	Fast
Preprocessing Required	No	Yes (LPS table)
Best Use Case	Small inputs	Large texts & repeated searches

Example 2: O(n) Case (KMP Algorithm - Skipping Comparisons)

This occurs when the algorithm **uses a prefix table** to avoid unnecessary rechecking of characters.

Example:

- **Text (n = 10):** `"aaaaaaaaaa"`
- **Pattern (m = 3):** `"aaa"`
- **Algorithm:** Knuth-Morris-Pratt (KMP)

Modified KMP to get O(n) Time Complexity

Example 2: O(n) Case (KMP Algorithm - Skipping Comparisons)

This occurs when the algorithm **uses a prefix table** to avoid unnecessary rechecking of characters.

Example:

- **Text (n = 10):** `"aaaaaaaaaa"`
- **Pattern (m = 3):** `"aaa"`
- **Algorithm:** Knuth-Morris-Pratt (KMP)

Step	Text Window Being Checked	Pattern Matched?	Comparisons Made	Pattern Shift
1	"aaa"aaaaaaa	✅ Match	3 comparisons	Shift by 2 (from prefix table)
2	a"aaa"aaaaaa	✅ Match	1 comparison (skipped 2)	Shift by 2
3	aa"aaa"aaaaa	✅ Match	1 comparison (skipped 2)	Shift by 2
4	aaa"aaa"aaaa	✅ Match	1 comparison (skipped 2)	Shift by 2
5	aaaa"aaa"aaa	✅ Match	1 comparison (skipped 2)	Shift by 2
6	aaaaa"aaa"aa	✅ Match	1 comparison (skipped 2)	Shift by 2
7	aaaaaa"aaa"a	✅ Match	1 comparison (skipped 2)	Shift by 2
8	aaaaaaa"aaa"	✅ Match	1 comparison (skipped 2)	-
Total	-	Multiple matches	O(n) = 10 comparisons	-

Example 1: O(n + m) Case (Naïve Algorithm)

This occurs when the algorithm checks **every position** in the text, comparing all characters of the pattern at each step.

Example:

- **Text (n = 10):** `"abcdefghij"`
- **Pattern (m = 3):** `"abc"`
- **Algorithm:** Naïve String Matching

Step	Text Window Being Checked	Pattern Match?	Comparisons Made
1	`"abc" defghij`	✅ **Match found**	3 comparisons (all matched)
2	`a "bcd" efghij`	❌ No match	3 comparisons
3	`ab "cde" fghij`	❌ No match	3 comparisons
4	`abc "def" ghij`	❌ No match	3 comparisons
5	`abcd "efg" hij`	❌ No match	3 comparisons
6	`abcde "fgh" ij`	❌ No match	3 comparisons
7	`abcdef "ghi" j`	❌ No match	3 comparisons
8	`abcdefg "hij"`	❌ No match	3 comparisons
Total	-	**1 match found**	**O(n + m) = 10 + 3 = 13 comparisons**

Example 2: O(n) Case (KMP Algorithm - Skipping Comparisons)

✅ **Why O(n)?**

- Instead of rechecking already matched characters, **KMP skips comparisons** using a prefix table, ensuring the time complexity stays **O(n)**.

KMP algorithm

- The **Knuth-Morris-Pratt (KMP) Algorithm** optimizes pattern matching by using a **prefix table** (also called the **lps table** – longest proper prefix which is also a suffix). This helps avoid unnecessary rechecking of characters, allowing the algorithm to **shift the pattern efficiently**.
- O(n) Case (KMP Algorithm - Skipping Comparisons)
- This occurs when the algorithm **uses a prefix table** to avoid unnecessary rechecking of characters.

How Does the Prefix Table Work?

- The **prefix table (lps array)** stores information about the longest prefix of the pattern that is also a suffix.
- When a mismatch occurs, instead of shifting by 1 (like in the naïve approach), **KMP uses the lps table to determine how far it can shift**.
- This reduces redundant comparisons, making the algorithm run in **O(n) time** instead of **O(n + m).**

Example 3: O(n) Case (Boyer-Moore Algorithm - Large Jumps)

This occurs when the algorithm makes **large shifts** based on mismatches.

Example:

- **Text (n = 10):** "abcdefghij"
- **Pattern (m = 3):** "xyz"
- **Algorithm:** Boyer-Moore

Step	Text Window Being Checked	Last Character in Pattern	Mismatch Found?	Pattern Shift
1	`"abc"defghij`	`'c'` (not in pattern)	❌ Yes	Shift by **3**
2	`abc"def"ghij`	`'f'` (not in pattern)	❌ Yes	Shift by **3**
3	`abcdef"ghi"j`	`'i'` (not in pattern)	❌ Yes	Shift by **3**
4	`abcdefgh"ij"`	`'j'` (not in pattern)	❌ Yes	Shift by **3**
Total	-	-	**No matches found**	**O(n) = 4 comparisons**

Example 3: O(n) Case (Boyer-Moore Algorithm - Large Jumps)

This occurs when the algorithm makes **large shifts** based on mismatches.

Example:

- **Text (n = 10):** "abcdefghij"
- **Pattern (m = 3):** "xyz"
- **Algorithm:** Boyer-Moore

✅ **Why O(n)?**

- Boyer-Moore makes **large jumps** on mismatches, reducing the number of comparisons **to a fraction of n.**

Step	Text Window Being Checked	Last Character in Pattern	Mismatch Found?	Pattern Shift
1	`"abc"defghij`	`'c'` (not in pattern)	❌ Yes	Shift by **3**
2	`abc"def"ghij`	`'f'` (not in pattern)	❌ Yes	Shift by **3**
3	`abcdef"ghi"j`	`'i'` (not in pattern)	❌ Yes	Shift by **3**
4	`abcdefgh"ij"`	`'j'` (not in pattern)	❌ Yes	Shift by **3**
Total	-	-	**No matches found**	**O(n) = 4 comparisons**

Final Summary of Complexity Differences

Case	Algorithm	Pattern Matching Behavior	Time Complexity
O(n + m) (Slowest Case)	Naïve Algorithm	Checks every character in the text sequentially	**O(n + m)**
O(n) (Skipping Comparisons)	KMP Algorithm	Uses prefix table to skip unnecessary comparisons	**O(n)**
O(n) (Large Jumps on Mismatch)	Boyer-Moore Algorithm	Uses bad-character rule to jump ahead	**O(n)**

Summary

- **O(n + m) happens when the pattern is rechecked at every position.**
- **O(n) is possible when the algorithm efficiently skips unnecessary comparisons** (KMP) or **makes large jumps** on mismatches (Boyer-Moore).
- **Optimized algorithms reduce time complexity by avoiding redundant operations, leading to O(n) in practice.**