13 String Matching Algorithms

For COMSC 132

Sam Bowne

Nov 30, 2024

Topics

- String notations
- Pattern matching
- Brute force algorithm
- Rabin-Karp algorithm
- KMP algorithm
- Boyer-Moore algorithm

String notations

String notations

- **String** example: "Hello, world!"
- **Substring** example: "Hello"
- Prefix **p** is the start of the string, like "Hello"
- The remainder is called **u**, ", World!"
- Suffix **d** is the end, like "World!"

endswith() and startswith()

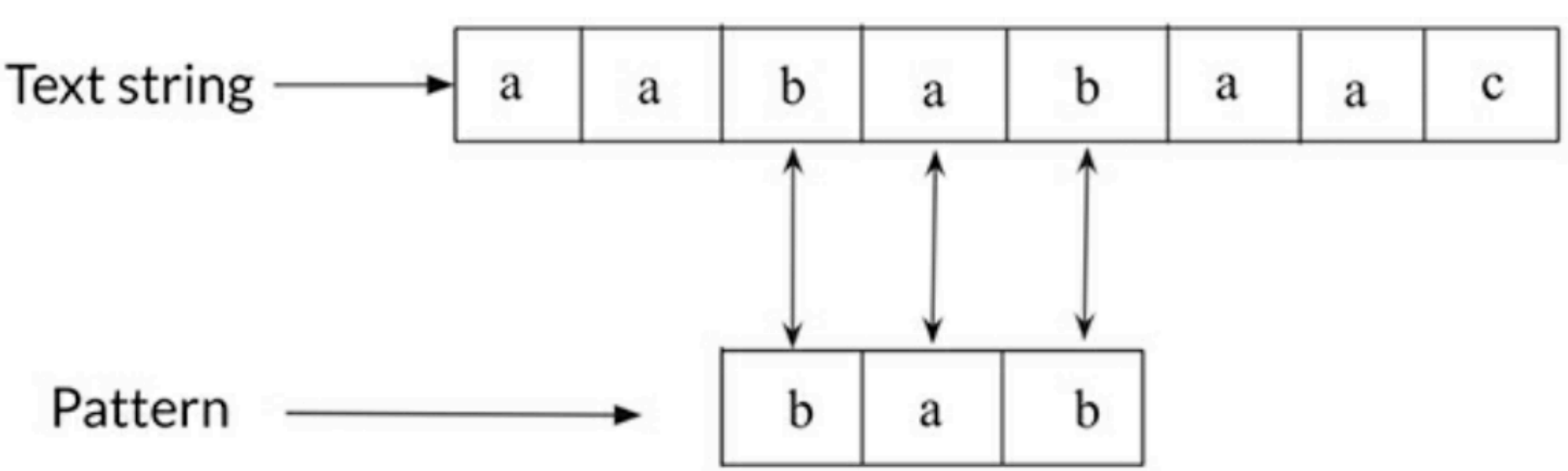
```
string = "this is data structures book by packt publisher"
suffix = "publisher"
prefix = "this"
print(string.endswith(suffix))  #Check if string contains given suffix.
print(string.startswith(prefix)) #Check if string starts with given prefix.
```

- Both of these statements are True

Pattern matching

Pattern matching

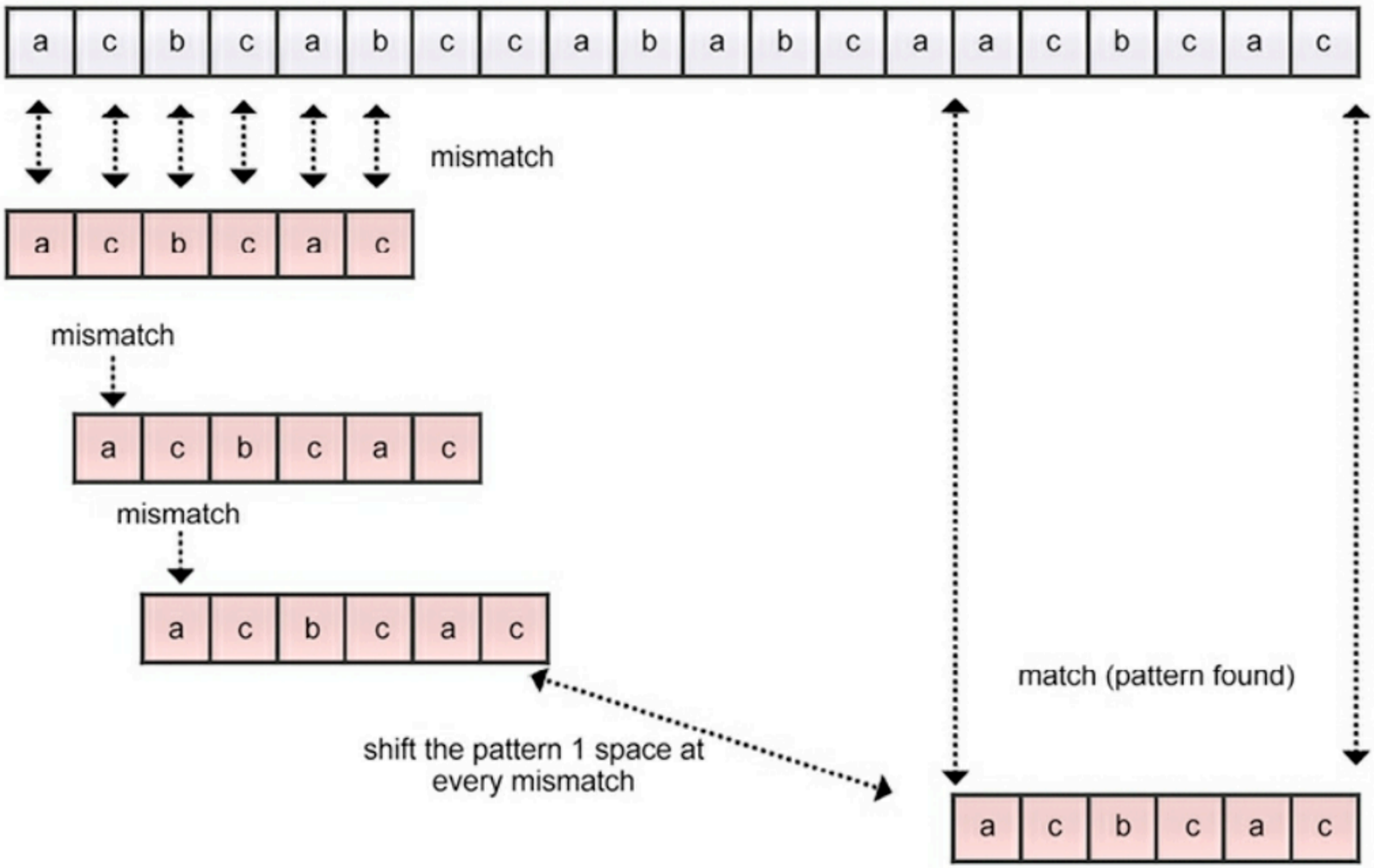
- If pattern is found, return its index location
- Otherwise, return "not found" (usually -1)



Brute force algorithm

Brute force

- Compare pattern to the start of the string
  - index = 0
- If that doesn't match, increment index and try again
- Continue until the last possible index value


a c b c a b c c a b a b c a a c b c a c
mismatch
a c b c a c
mismatch
a c b c a c
mismatch
a c b c a c
shift the pattern 1 space at
every mismatch
match (pattern found)
a c b c a c

Brute force complexity

- String has length ***n***
- Target has length ***m***
- Best case complexity is O(***n***)
  - When the first character of Target is not present in the string at all
  - Ex: find "FAA" in "AABBCCDDEE"

Brute force complexity

- Worst case complexity is O(***m****(***n***-***m***+1))
  - When all the characters in the Target and String are the same and we want to find all matches
    - Ex: find "AAA" in "AAAAAAAAAAA"

Rabin-Karp algorithm

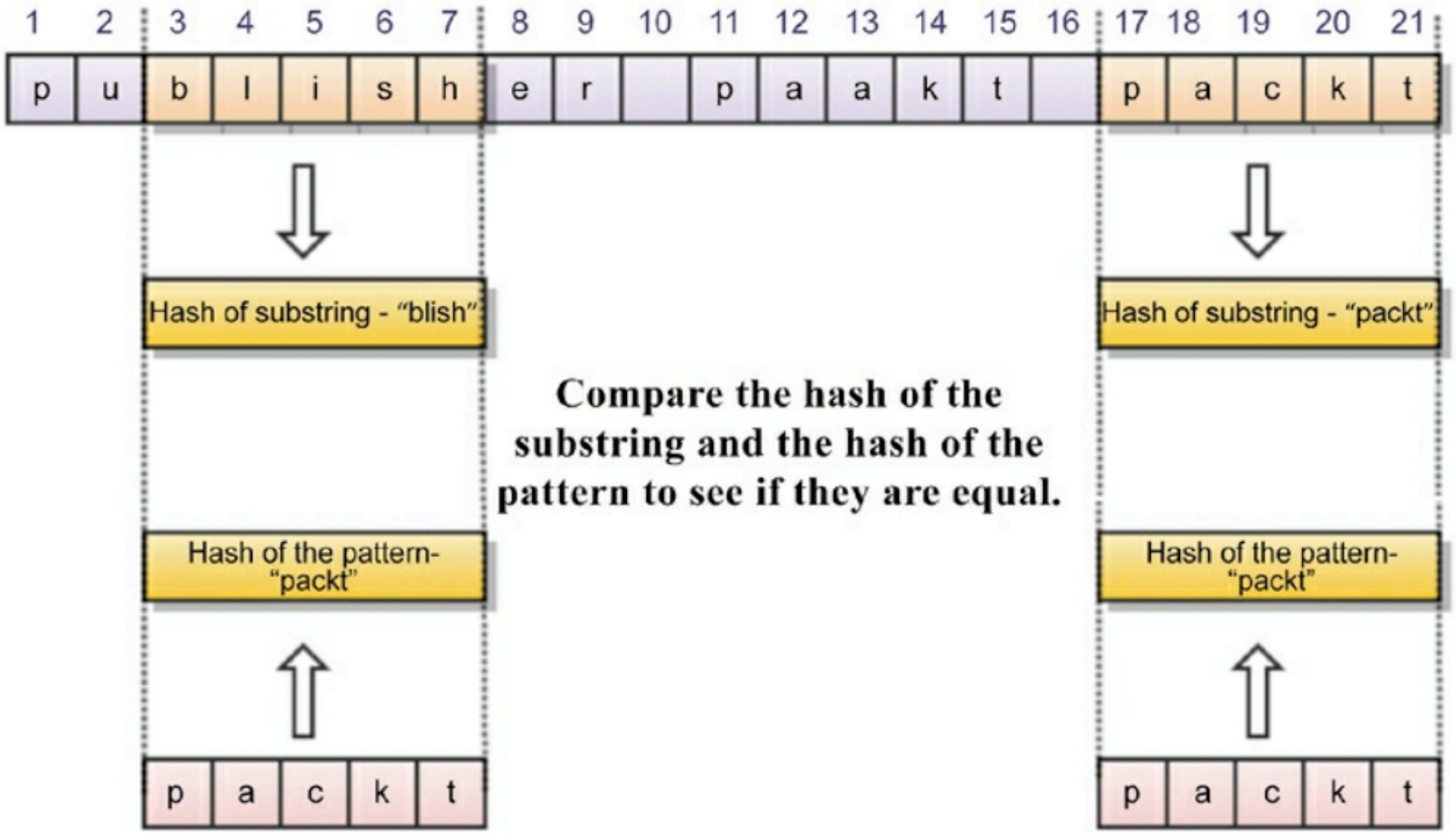
Rabin-Karp algorithm

- Reduce the number of comparisons using hashing
- A single comparison of hashes replaces comparing the characters one by one
- Since hashes can collide, verify a match by comparing the characters one by one

Rabin-Karp algorithm

1. Compute hash of pattern (length ***m***) and of all possible substrings of the text with length ***m***
2. Compare hash of pattern to the substring hashes, one by one
3. If there's a match, verify by comparing the characters

Rabin-Karp algorithm



KMP algorithm

KMP algorithm

- Use the structure of the target to exclude some shift values
- Avoiding unnecessary comparisons

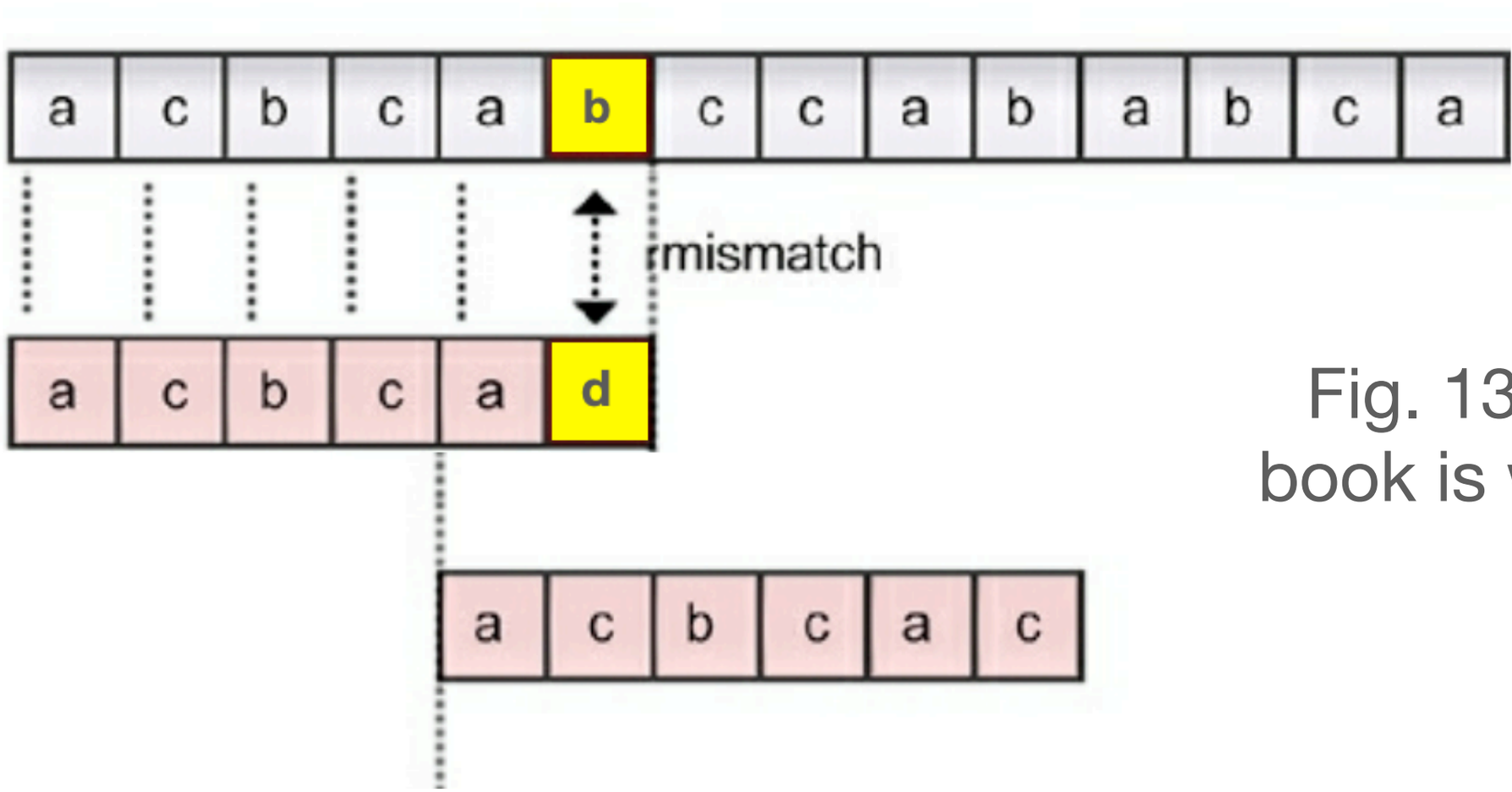
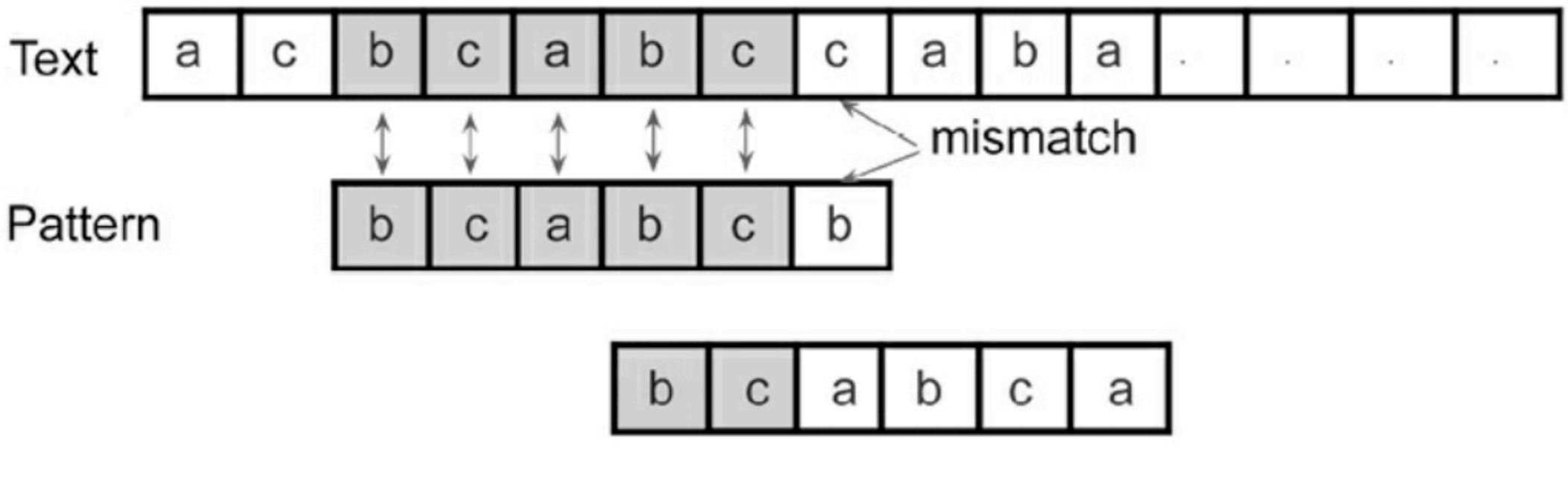


Fig. 13.4 in book is wrong

KMP algorithm



- In this case, there's a possible match 3 characters further

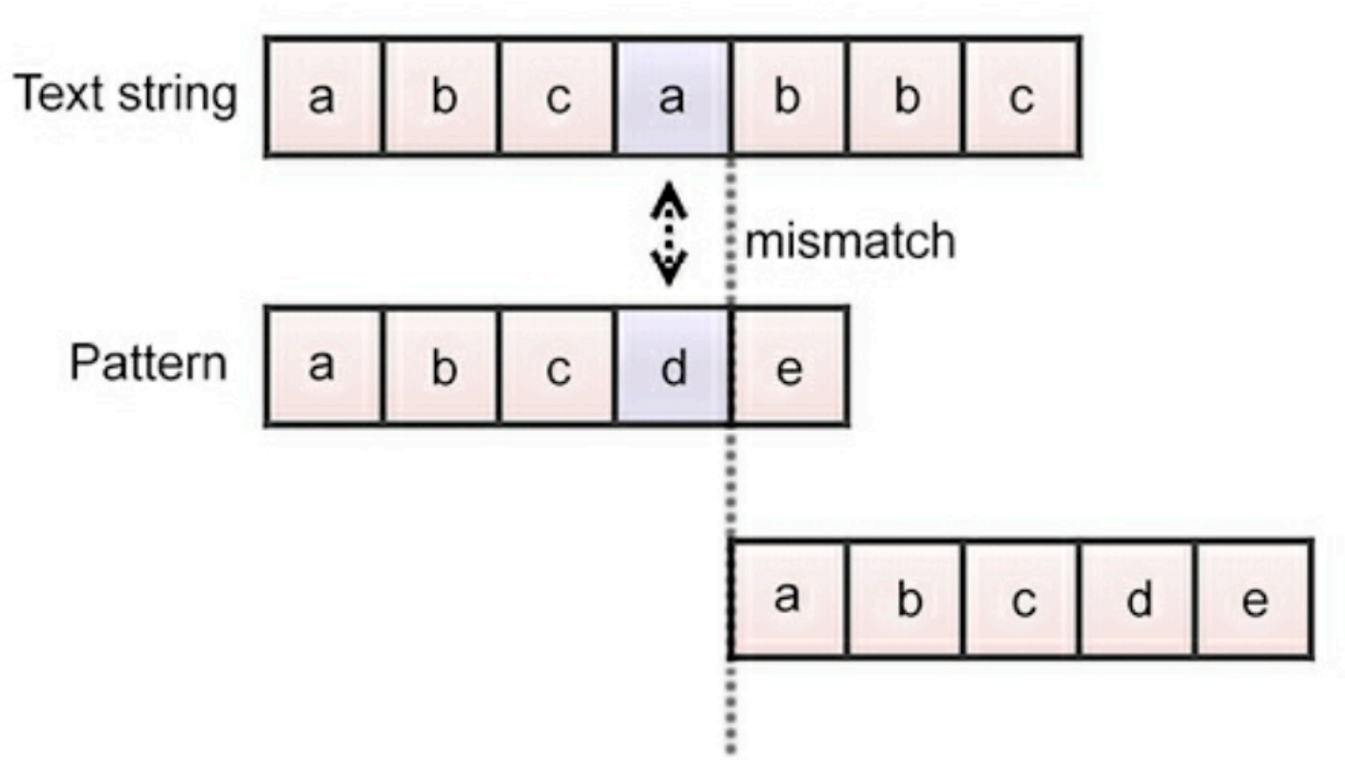
The prefix function

- Finds a pattern within the pattern
- Measures how much benefit can we get from the previous character comparisons
- Value is the # of characters from the start of the function that match at this point

The prefix function

- All characters in the pattern are different
- prefix() always 0
- Jump to the location of the first mismatched character

Index	1	2	3	4	5
Pattern	a	b	c	d	e
Prefix_function	0	0	0	0	0



The prefix function

- First 3 characters are all different
  - So prefix_function is 0

Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0						

The prefix function

- Character 4 repeats character 1
  - Prefix_function = 1
  - Next, try with index 1 shifted to here

Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0	1					

The prefix function

- Character 5 repeats character 2, forming the pattern **ab**
  - Prefix_function = 2
  - Next, try with index=2 here

Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0	1	2				

Beginning of KMP-matcher

```
def KMP_matcher(text, pattern):
    """
    Knuth-Morris-Pratt (KMP) string matching algorithm.

    Args:
    text (str): The text to search in.
    pattern (str): The pattern to search for.

Returns:
    list: A list of starting indices of all occurrences of the pattern in the text.
"""
```

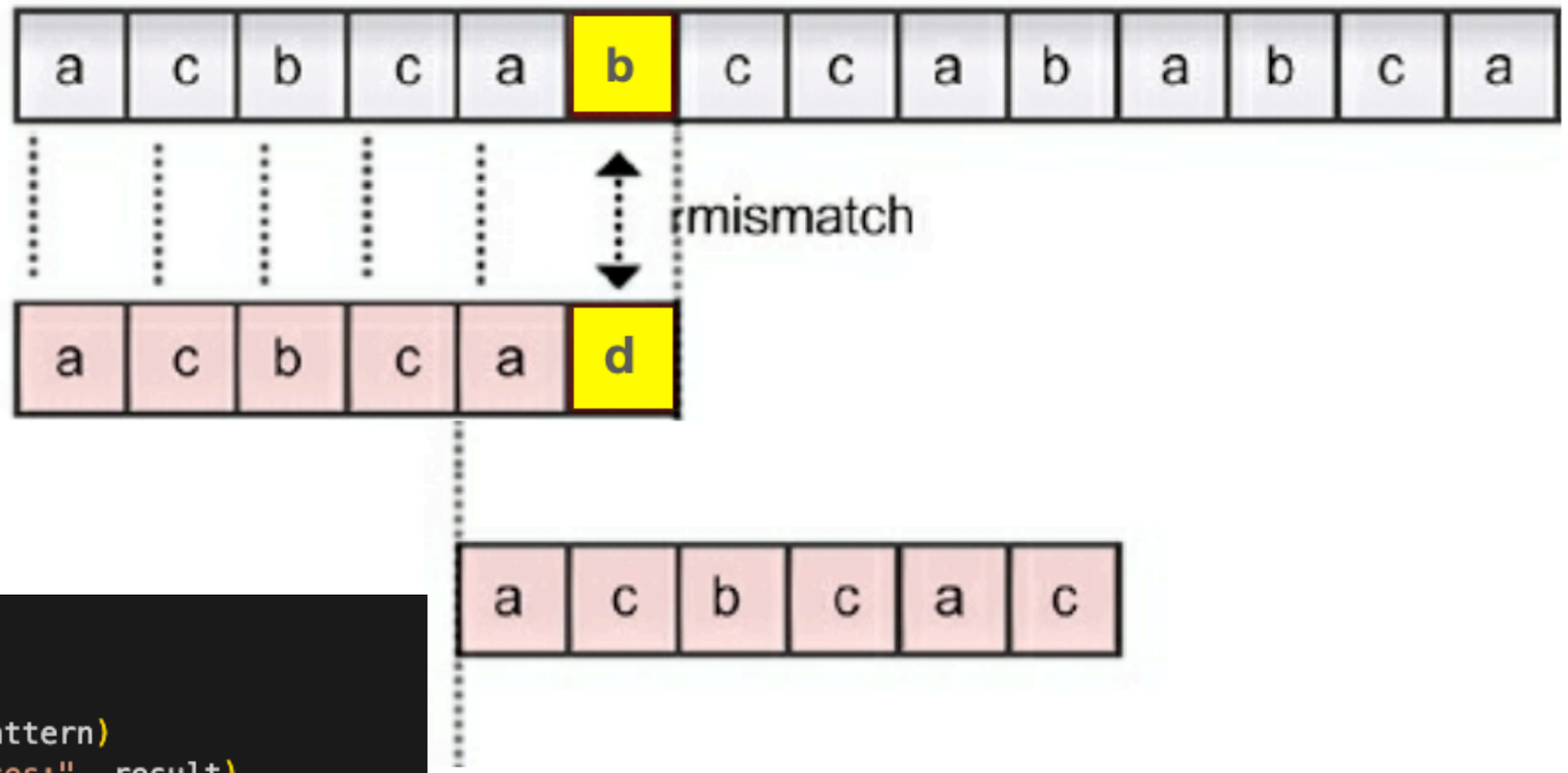
The prefix function

```
def compute_lps(pattern):
    """
    Computes the longest proper prefix-suffix (LPS) array for the given pattern.
    """
    m = len(pattern)
    lps = [0] * m
    length = 0
    i = 1

    while i < m:
    if pattern[i] == pattern[length]:
        length += 1
        lps[i] = length
        i += 1
    else:
        if length != 0:
            length = lps[length - 1]
        else:
            lps[i] = 0
            i += 1
return lps
```

The rest of KMP_matcher

```
n = len(text)
m = len(pattern)
lps = compute_lps(pattern)
print("Prefix function:", lps)

i = 0  # index for text
j = 0  # index for pattern
occurrences = []

while i < n:
    print("Comparing pattern[", j,"]", pattern[j], " to text[:", i, "]", text[i], end = " ")
    if pattern[j] == text[i]:
      print("Match!")
    else:
      print("Mismatch!")
    if pattern[j] == text[i]:
        i += 1
        j += 1

        if j == m:
            occurrences.append(i - j)
            print("Pattern found at index:", i - j)
            j = lps[j - 1]
    else:
        if j != 0:
            j = lps[j - 1]
        else:
            i += 1

return occurrences
```

Result




```
66 # Example usage
67 text = "acbcabccababca"
68 pattern = "acbcad"
69 result = KMP_matcher(text, pattern)
70 print("Pattern found at indices:", result)
```

```
Prefix function: [0, 0, 0, 0, 1, 0]
Comparing pattern[ 0 ] a  to text[: 0 ] a Match!
Comparing pattern[ 1 ] c  to text[: 1 ] c Match!
Comparing pattern[ 2 ] b  to text[: 2 ] b Match!
Comparing pattern[ 3 ] c  to text[: 3 ] c Match!
Comparing pattern[ 4 ] a  to text[: 4 ] a Match!
Comparing pattern[ 5 ] d  to text[: 5 ] b Mismatch!
Comparing pattern[ 1 ] c  to text[: 5 ] b Mismatch!
Comparing pattern[ 0 ] a  to text[: 5 ] b Mismatch!
Comparing pattern[ 0 ] a  to text[: 6 ] c Mismatch!
Comparing pattern[ 0 ] a  to text[: 7 ] c Mismatch!
Comparing pattern[ 0 ] a  to text[: 8 ] a Match!
Comparing pattern[ 1 ] c  to text[: 9 ] b Mismatch!
Comparing pattern[ 0 ] a  to text[: 9 ] b Mismatch!
Comparing pattern[ 0 ] a  to text[: 10 ] a Match!
Comparing pattern[ 1 ] c  to text[: 11 ] b Mismatch!
Comparing pattern[ 0 ] a  to text[: 11 ] b Mismatch!
Comparing pattern[ 0 ] a  to text[: 12 ] c Mismatch!
Comparing pattern[ 0 ] a  to text[: 13 ] a Match!
Pattern found at indices: []
```

Boyer-Moore algorithm

Boyer-Moore algorithm

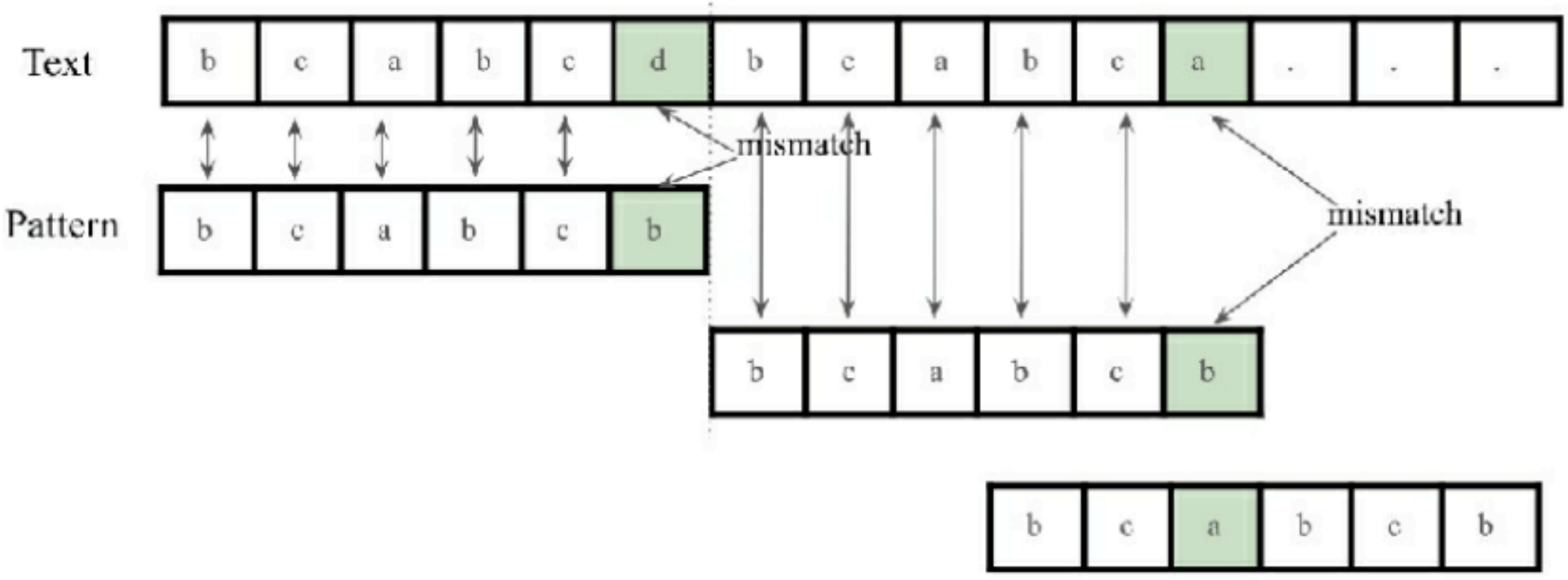
1. In this algorithm, we shift the pattern in the direction from left to right, similar to the KMP algorithm.
2. We compare the characters of the pattern and the text string from right to left, which is the opposite of what we do in the case of the KMP algorithm.
3. The algorithm skips the unnecessary comparisons by using the good suffix and bad character shift heuristics. These heuristics themselves find the possible number of comparisons that can be skipped. We slide the pattern over the given text with the greatest offsets suggested by both of these heuristics.

Boyer-Moore algorithm

- Uses two heuristics to determine the maximum possible shift after a mismatch
  - Bad character heuristic
  - Good suffix heuristic
  - Shift the pattern by the longer of the two distances given by those heuristics

Boyer-Moore algorithm

- Since **d** is not in the pattern, shift the entire pattern length
- **a** is in the pattern, so shift to match it

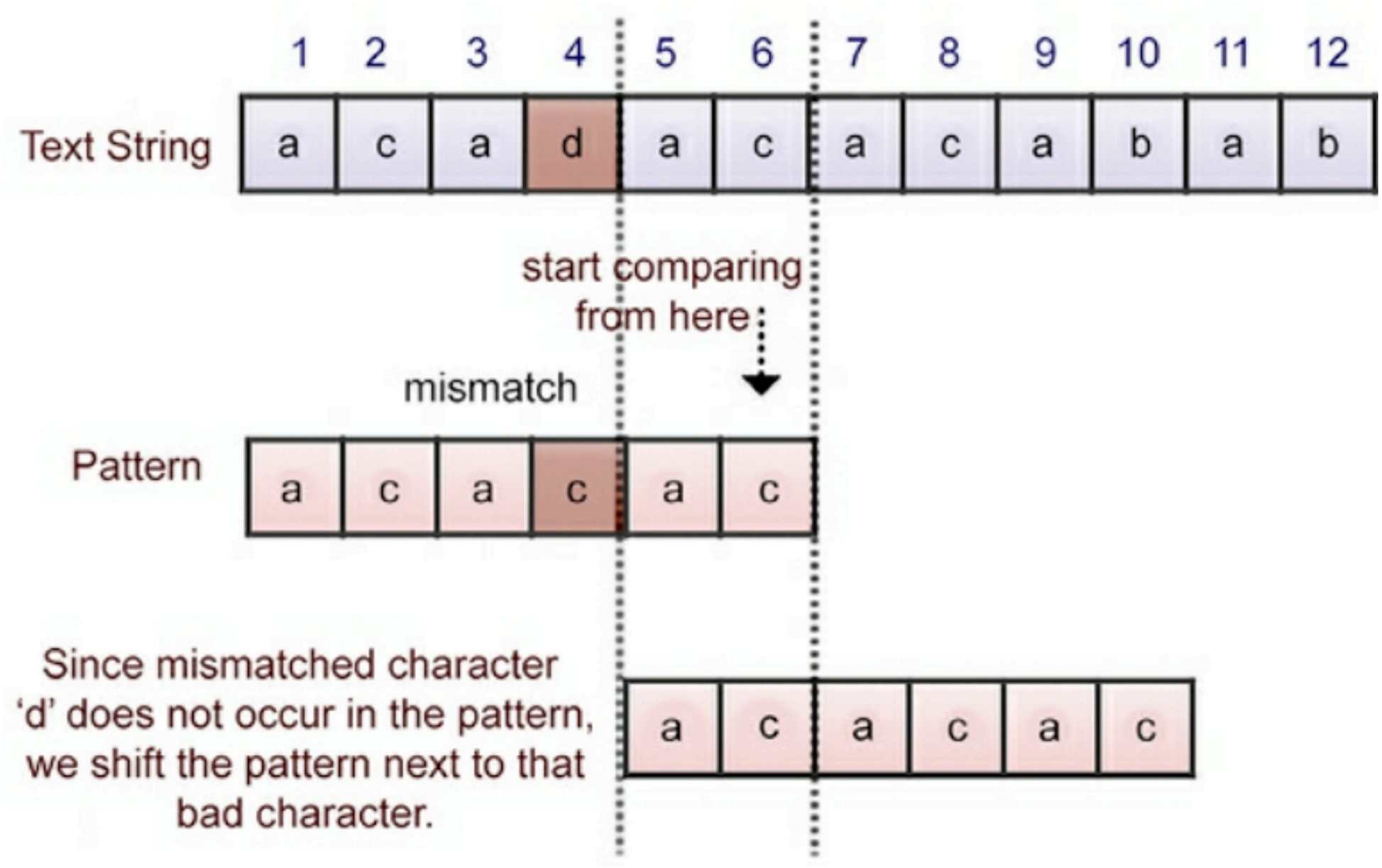


Bad character heuristic

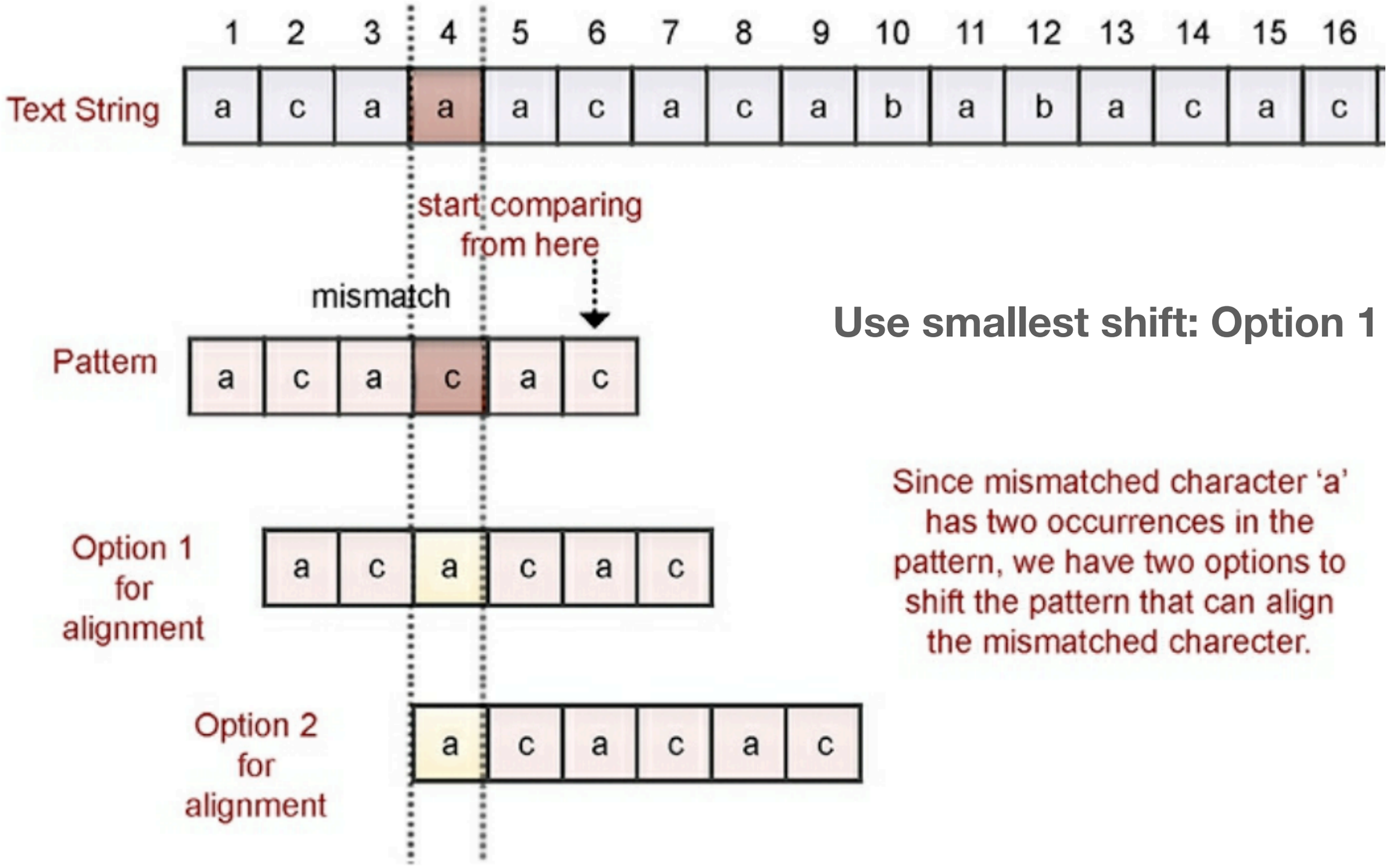
1. If the mismatched character of the text does not occur in the pattern, then we shift the pattern next to the mismatched character.
2. If the mismatched character has one occurrence in the pattern, then we shift the pattern in such a way that we align with the mismatched character.
3. If the mismatched character has more than one occurrence in the pattern, then we make the most minimal shift possible to align the pattern with that character.

Bad character heuristic

- Mismatched character is called "bad"



Bad character heuristic



Use smallest shift: Option 1

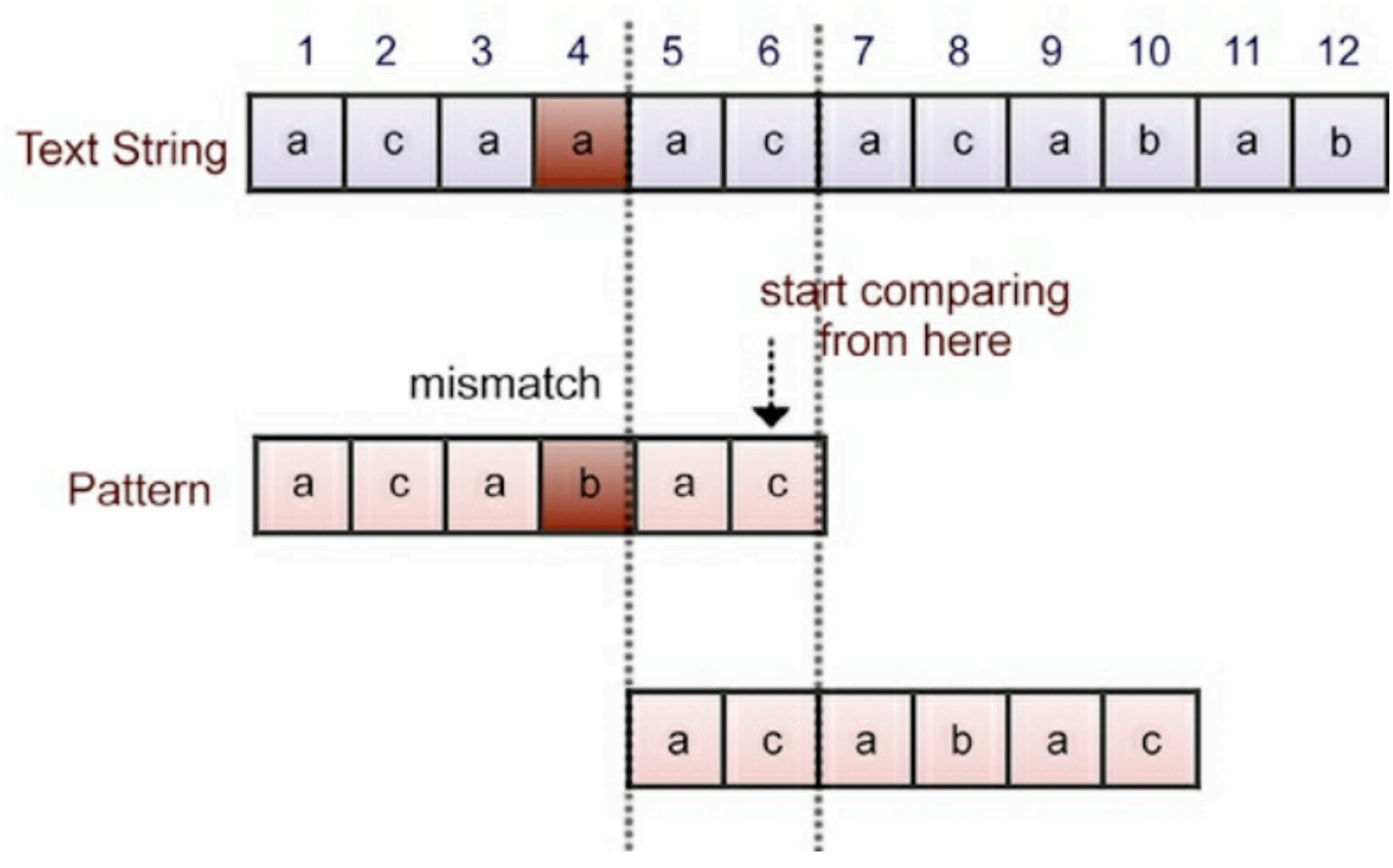
Since mismatched character 'a' has two occurrences in the pattern, we have two options to shift the pattern that can align the mismatched charecter.

Good suffix heuristic

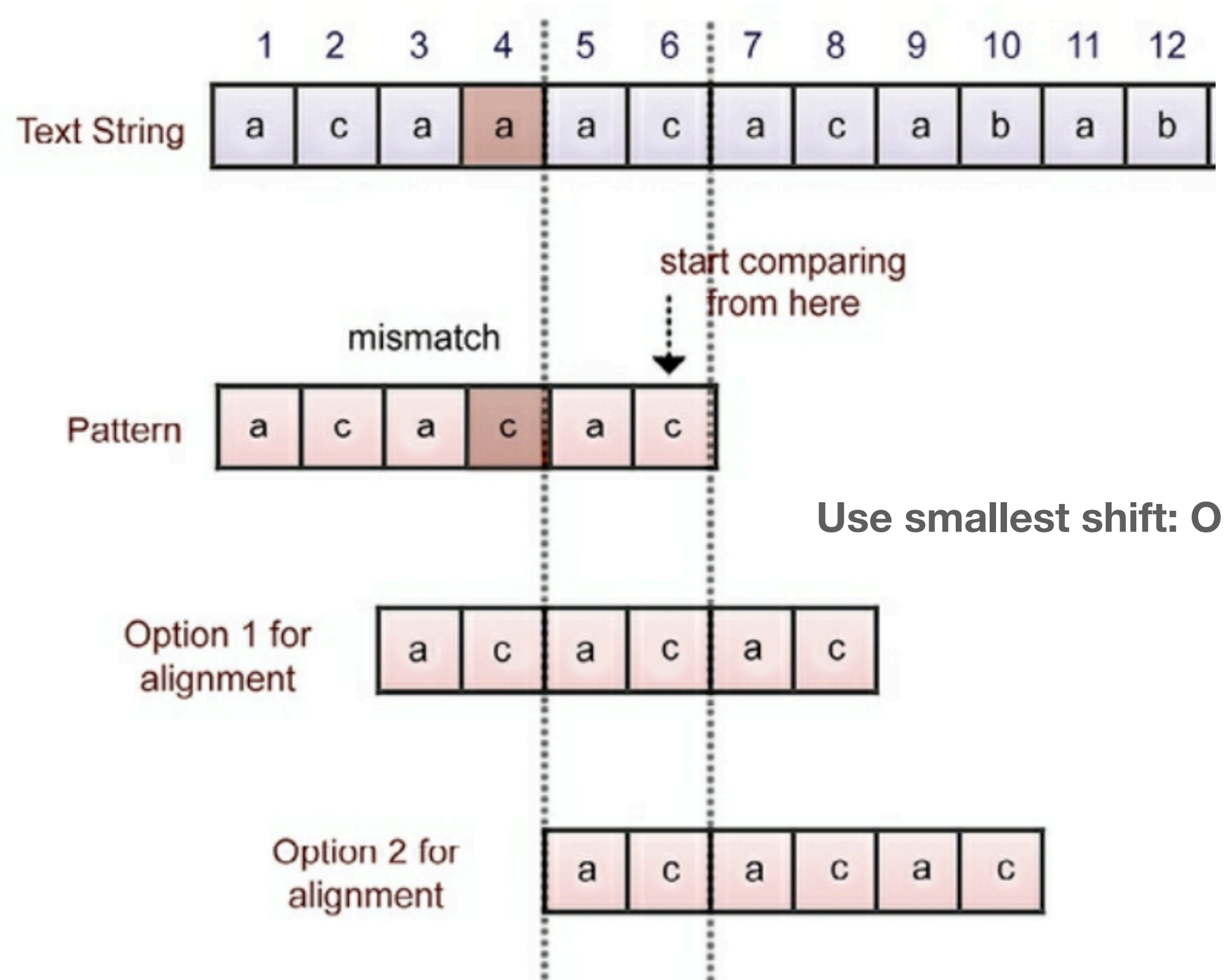
1. The matching suffix has one or more occurrences in the pattern
2. Some part of the matching suffix is present at the start of the pattern (this means that the suffix of the matched suffix exists as the prefix of the pattern)

Good suffix heuristic

- Shift to match other occurrence of the **good suffix**



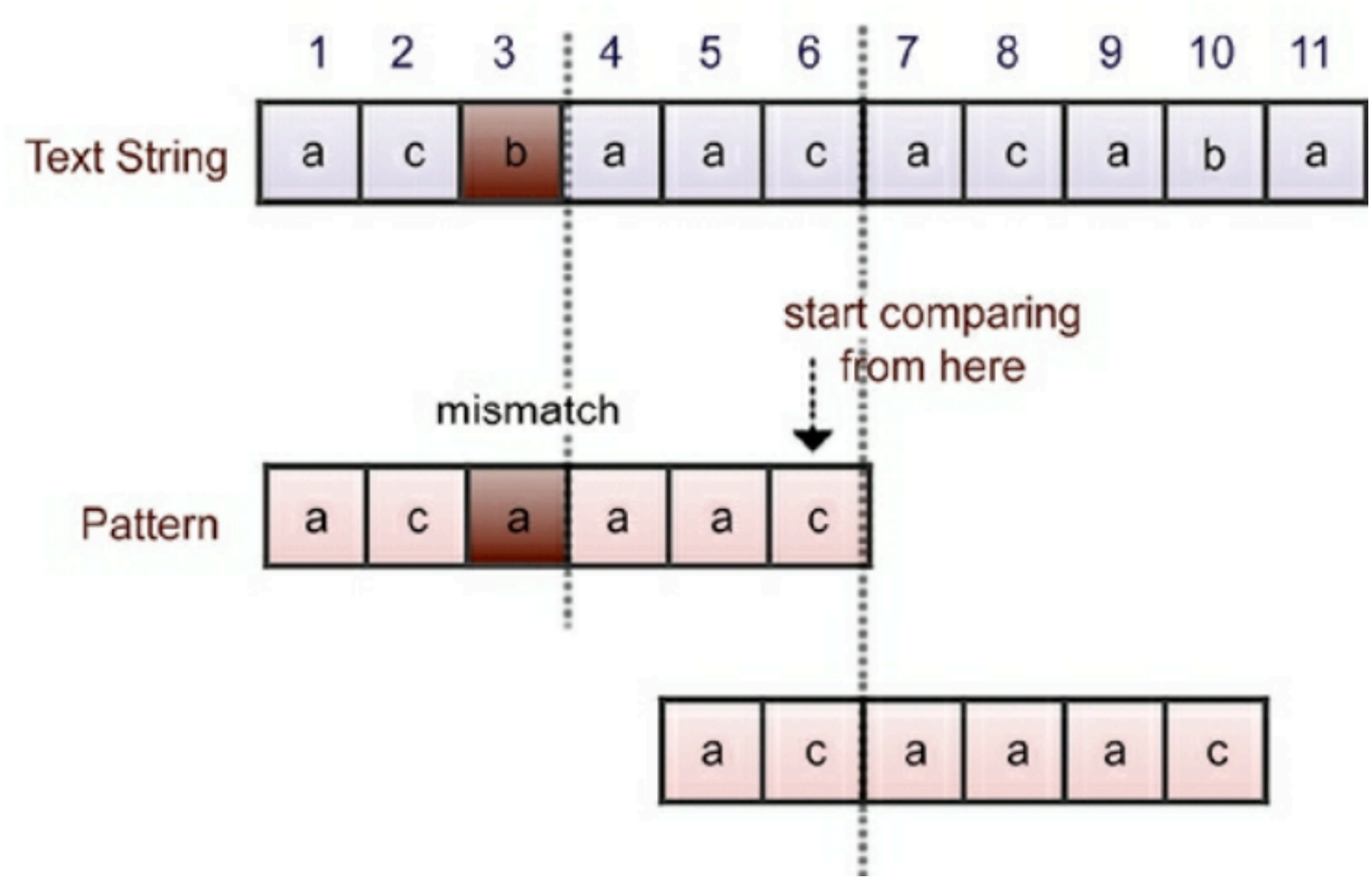
Good suffix heuristic



Use smallest shift: Option 1

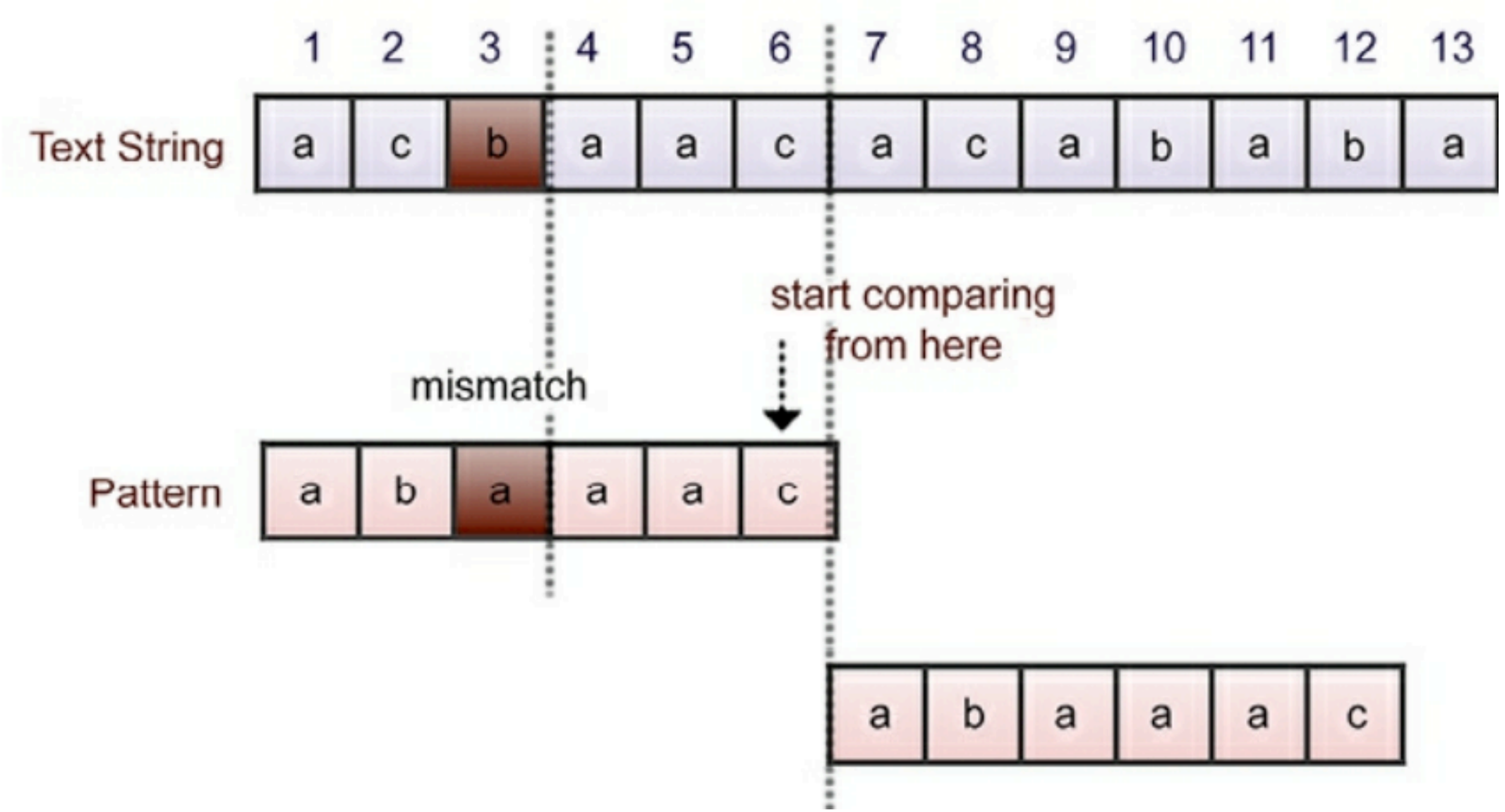
Good suffix heuristic

- Good suffix is **aac** but only **ac** has a match



Good suffix heuristic

- Good suffix is **aac** but there's no earlier match in the pattern, even of **c**



Boyer-Moore algorithm

- Complexity:
  - O(***m***) for preprocessing the pattern
  - O(***mn***) for searching in the worst case
  - BUT with an optimization called the "Galli rule" it's linear in all cases
- ***m*** is the length of the pattern
- ***n*** is the length of the text
  - ***https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string-search_algorithm***

Summary

- **Brute force** is inefficient
- **Rabin-Karp** uses hashing
- **KMP** uses overlapping substrings in the pattern to avoid redundant comparisons
- **Boyer-Moore** is very efficient with the text and pattern are long
  - Most popular algorithm in practice

Ch 13