Algorithmics

Lecture 4

Dr. Oana Andrei

School of Computing Science
University of Glasgow

oana.andrei@glasgow.ac.uk

Section 2 – Strings and text algorithms

Text compression

- Huffman encoding
- LZW compression/decompression

String comparison

- string distance

String/pattern search

- brute force algorithm
- KMP algorithm
- BM algorithm

String search – Brute force algorithm

Worst case is no better than O(mn)

- e.g. search for s = aa … ab in t = aa ... aaaa … ab

- m character comparisons needed at each n-(m+1) positions in the text before the text/pattern is found

Typically, the number of comparisons from each point will be small

- often just 1 comparison needed to show a mismatch
- so we can expect O(n) on average

Challenges: can we find a solution that is…

1. linear, i.e. O(m+n) in the worst case?
2. (much) faster than brute force on average?

String search – KMP algorithm

The Knuth-Morris-Pratt (KMP) algorithm

- addresses first challenge: linear (O(m+n)) in the worst case

About Donald Knuth

Credit photo: blog.typekit.com

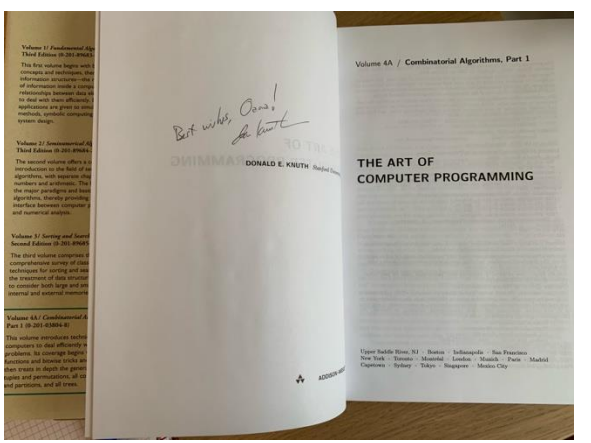
Known as the "father of the analysis of algorithms"

Recipient of the ACM Turing Award in 1974 (aka Nobel Prize of Computer Science)

Honorary degree from the University of Glasgow 2011

Wrote "The Art of Computer Programming"

- volumes 1-7
- when type setting he came up with TeX - the predecessor or LaTeX



Extra: https://cacm.acm.org/news/175194-twenty-questions-for-donald-knuth

String search – KMP algorithm

The Knuth-Morris-Pratt (KMP) algorithm

- addresses first challenge: linear (`O(m+n)`) in the worst case

It is an on-line algorithm

- i.e., it removes the need to back-up in the text
- involves pre-processing the string to build a border table
- border table: an array `b` with entry `b[j]` for each position `j` of the string

If we get a mismatch at position `j` in the string/pattern

- we remain on the current text character (do not back-up)
- the border table tells us which string character should next be compared with the current text character

String search – KMP algorithm

A substring of string s is a sequence of consecutive characters of s

- if s has length n, then s[i..j] is a substring for i and j with $0 \le i \le j \le n-1$

A prefix of s is a substring that begins at position 0

- i.e. s[0..j] for any j with $0 \le j \le n-1$

A suffix of s is a substring that ends at position n-1

- i.e. s[i..n-1] for any i with $0 \le i \le n-1$

A border of a string s is a substring that is both a prefix and a suffix and cannot be the string itself

- e.g. s = a c a c g a t a c a c
- a c and a c a c are borders

String search – KMP algorithm

A substring of string s is a sequence of consecutive characters of s

- if s has length n, then s[i..j] is a substring for i and j with $0 \le i \le j \le n-1$

A prefix of s is a substring that begins at position 0

- i.e. s[0..j] for any j with $0 \le j \le n-1$

A suffix of s is a substring that ends at position n-1

- i.e. s[i..n-1] for any i with $0 \le i \le n-1$

A border of a string s is a substring that is both a prefix and a suffix and cannot be the string itself

- e.g. s = **a c** a c g a t a c **a c**
- **a c** and a c a c are borders

String search – KMP algorithm

A substring of string s is a sequence of consecutive characters of s

- if s has length n, then s[i..j] is a substring for i and j with $0 \le i \le j \le n-1$

A prefix of s is a substring that begins at position 0

- i.e. s[0..j] for any j with $0 \le j \le n-1$

A suffix of s is a substring that ends at position n-1

- i.e. s[i..n-1] for any i with $0 \le i \le n-1$

A border of a string s is a substring that is both a prefix and a suffix and cannot be the string itself

- e.g. s = **a c a c** g a t **a c a c**
- a c and **a c a c** are borders

String search – KMP algorithm

A substring of string `s` is a sequence of consecutive characters of `s`

- if `s` has length `n`, then `s[i..j]` is a substring for `i` and `j` with $0 \le i \le j \le n-1$

A prefix of `s` is a substring that begins at position `0`

- i.e. `s[0..j]` for any `j` with $0 \le j \le n-1$

A suffix of `s` is a substring that ends at position `n-1`

- i.e. `s[i..n-1]` for any `i` with $0 \le i \le n-1$

A border of a string `s` is a substring that is both a prefix and a suffix and cannot be the string itself

- e.g. `s = a c a c g a t a c a c`
- `a c` and `a c a c` are borders and `a c a c` is the longest border

String search – KMP algorithm

A substring of string s is a sequence of consecutive characters of s

- if s has length n, then s[i..j] is a substring for i and j with $0 \le i \le j \le n-1$

A prefix of s is a substring that begins at position 0

- i.e. s[0..j] for any j with $0 \le j \le n-1$

A suffix of s is a substring that ends at position n-1

- i.e. s[i..n-1] for any i with $0 \le i \le n-1$

A border of a string s is a substring that is both a prefix and a suffix and cannot be the string itself

- e.g. s = a c a c g a t a c a c
- a c and a c a c are borders and a c a c is the longest border

Many strings have no border

- we then say that the empty string ε (of length 0) is the longest border

String search – Border table

KMP algorithm requires the border table of the string/pattern

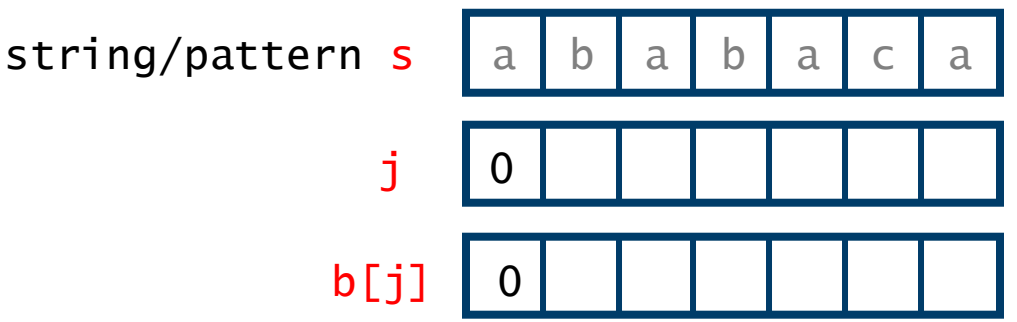
- a border of a string s is a substring that is both a prefix and a suffix and cannot be the string itself

Border table b: array which has the same size as the string

- b[j] = the length of the longest border of s[0..j-1]

  = max { k | s[0..k-1] = s[j-k..j-1] ∧ k<j }

Example



- no common prefix/suffix of empty string (when j=0) so set b[0] to 0

String search – Border table

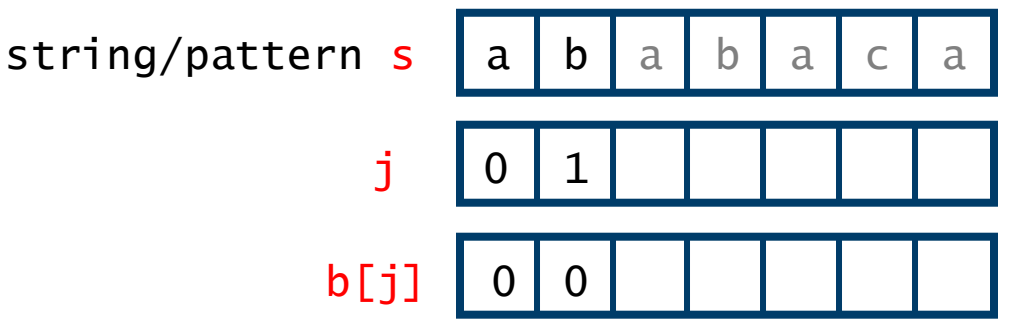
KMP algorithm requires the border table of the string pattern

- a border of a string s is a substring that is both a prefix and a suffix and cannot be the string itself

Border table b: array which has the same size as the string

- b[j] = the length of the longest border of s[0..j-1]
  = max { k | s[0..k-1] = s[j-k..j-1] ∧ k<j }

Example



- no common prefix/suffix of a (no border for single-char string), so b[i] is set to 0

String search – Border table

KMP algorithm requires the border table of the string pattern

- a border of a string s is a substring that is both a prefix and a suffix and cannot be the string itself

Border table b: array which has the same size as the string

- b[j] = the length of the longest border of s[0..j-1]
  = max { k | s[0..k-1] = s[j-k..j-1] ∧ k<j }

Example

string/pattern s	a	b	a	b	a	c	a
j	0	1	2				
b[j]	0	0	0				

- no common prefix/suffix of ab so set to 0

String search – Border table

KMP algorithm requires the border table of the string pattern

- a border of a string s is a substring that is both a prefix and a suffix and cannot be the string itself

Border table b: array which has the same size as the string

- b[j] = the length of the longest border of s[0..j-1]
  = max { k | s[0..k-1] = s[j-k..j-1] ∧ k<j }

Example

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3			
b[j]	0	0	0	1			

- a is the longest prefix and suffix of aba

String search – Border table

KMP algorithm requires the border table of the string pattern

- a border of a string s is a substring that is both a prefix and a suffix and cannot be the string itself

Border table b: array which has the same size as the string

- b[j] = the length of the longest border of s[0..j-1]
  = max { k | s[0..k-1] = s[j-k..j-1] ∧ k<j }

Example

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4		
b[j]	0	0	0	1	2		

- ab is the longest prefix and suffix of abab

String search – Border table

KMP algorithm requires the border table of the string pattern

- a border of a string s is a substring that is both a prefix and a suffix and cannot be the string itself

Border table b: array which has the same size as the string

- b[j] = the length of the longest border of s[0..j-1]
  = max { k | s[0..k-1] = s[j-k..j-1] ∧ k<j }

Example

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	
b[j]	0	0	0	1	2	3	

- aba is the longest prefix and suffix of ababa

String search – Border table

KMP algorithm requires the border table of the string pattern

- a border of a string s is a substring that is both a prefix and a suffix and cannot be the string itself

Border table b: array which has the same size as the string

- b[j] = the length of the longest border of s[0..j-1]
  = max { k | s[0..k-1] = s[j-k..j-1] ∧ k<j }

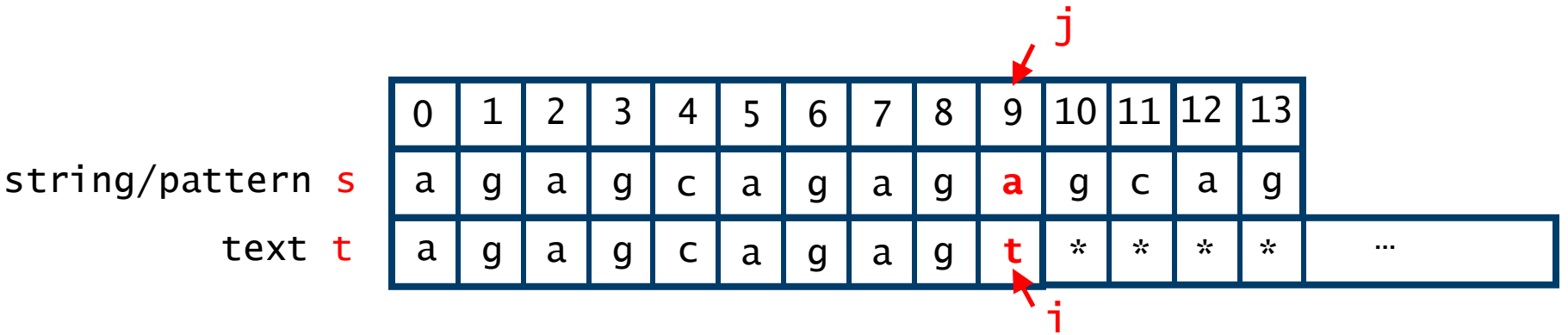
Example

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

- no common prefix/suffix of ababac so set to 0

String search – Brute force versus KMP

Example – Mismatch between s and t at position 9 in s

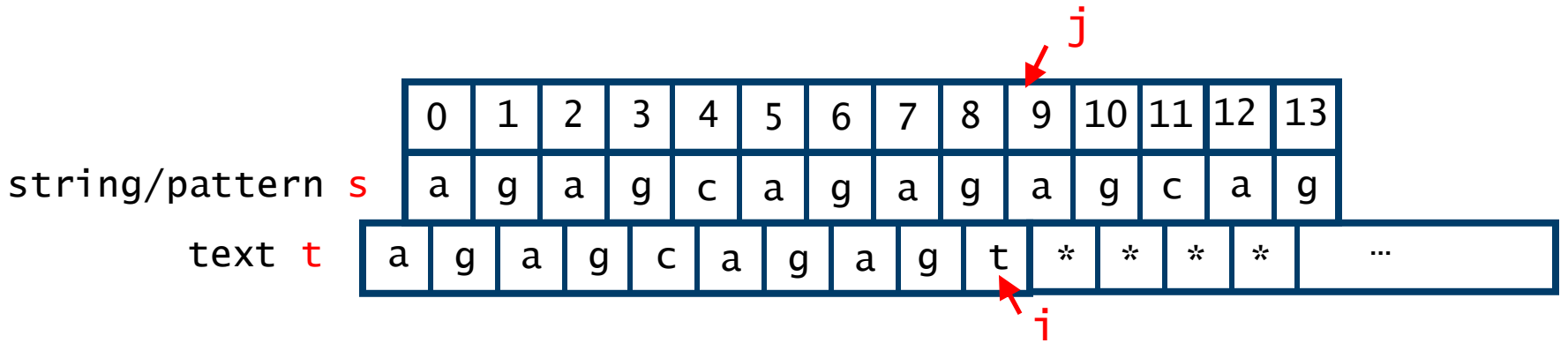


Applying the brute force algorithm, after the mis-match:

- s has to be 'moved along' one position relative to t

String search – Brute force versus KMP

Example – Mismatch between s and t at position 9 in s

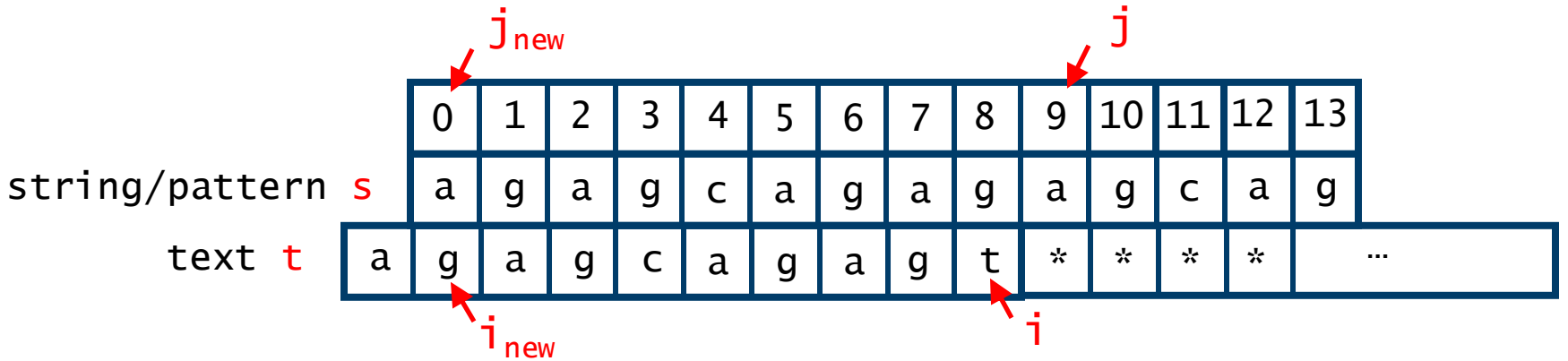


Applying the brute force algorithm, after the mis-match:

- s has to be 'moved along' one position relative to t

String search – Brute force versus KMP

Example – Mismatch between s and t at position 9 in s

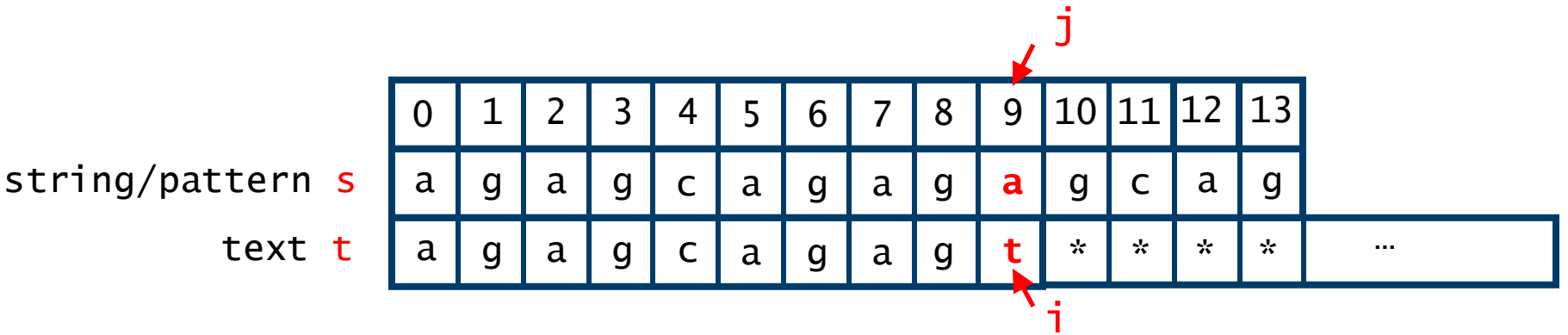


Applying the brute force algorithm, after the mis-match:

- s has to be 'moved along' one position relative to t
- then we start again at position 0 in s and jump back j-1 positions in t

String search – Brute force versus KMP

Example – Mismatch between **s** and **t** at position **9** in **s**

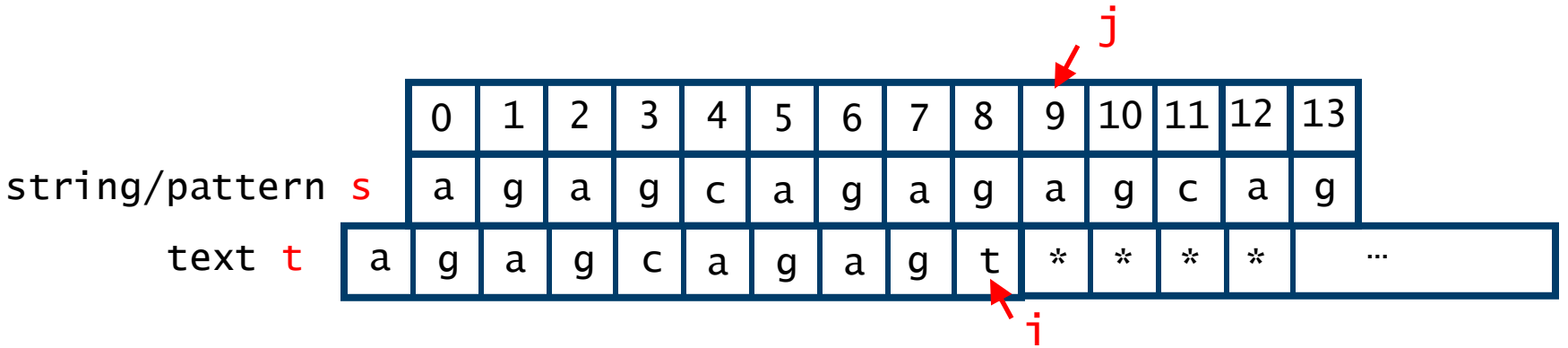


Applying the KMP algorithm, after the mis-match:

- s has to be 'moved along' until the characters to the left of i again match

String search – Brute force versus KMP

Example – Mismatch between s and t at position 9 in s

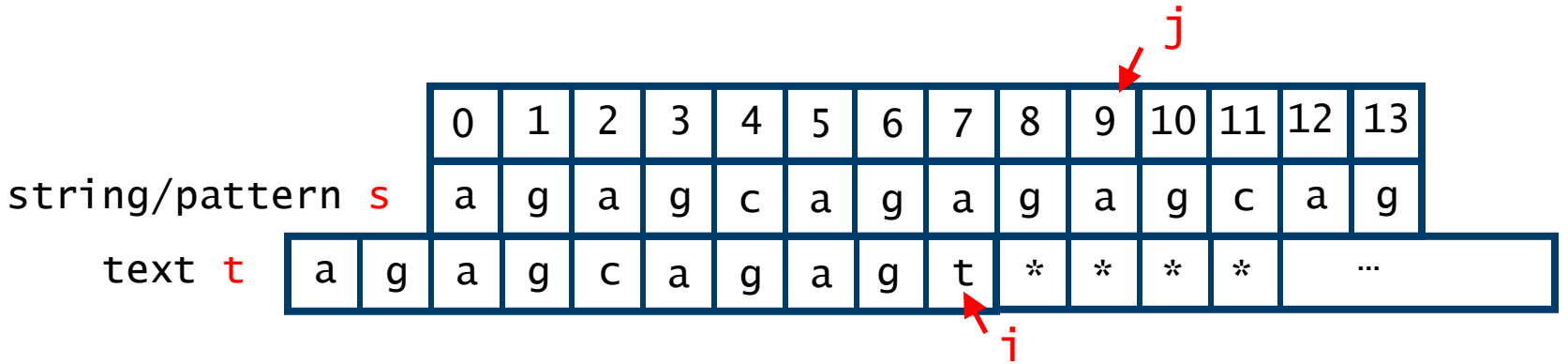


Applying the KMP algorithm, after the mis-match:

- s has to be 'moved along' until the characters to the left of i again match

String search – Brute force versus KMP

Example – Mismatch between s and t at position 9 in s

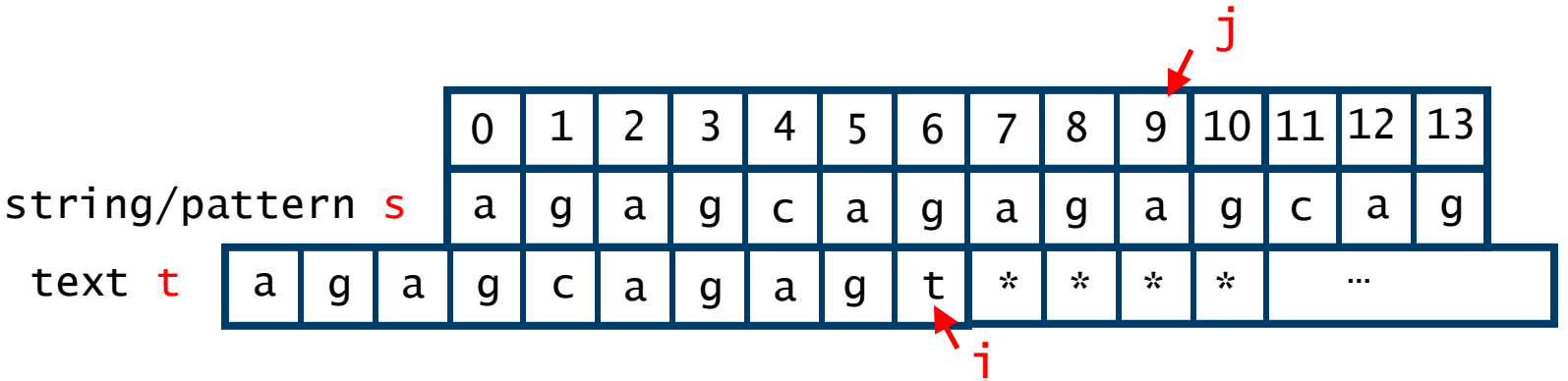


Applying the KMP algorithm, after the mis-match:

- s has to be 'moved along' until the characters to the left of i again match

String search – Brute force versus KMP

Example – Mismatch between s and t at position 9 in s

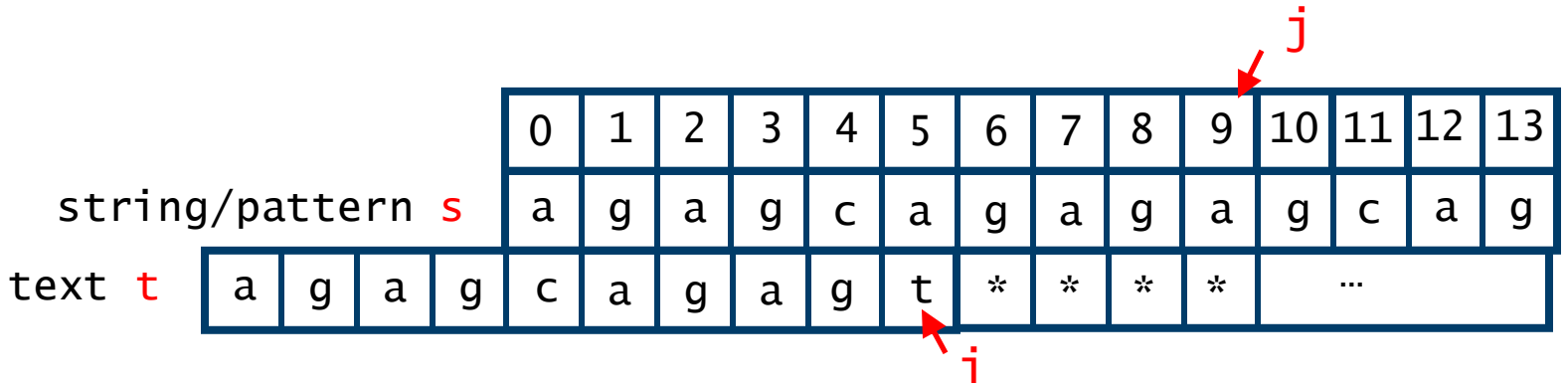


Applying the KMP algorithm, after the mis-match:

- s has to be 'moved along' until the characters to the left of i again match

String search – Brute force versus KMP

Example – Mismatch between s and t at position 9 in s

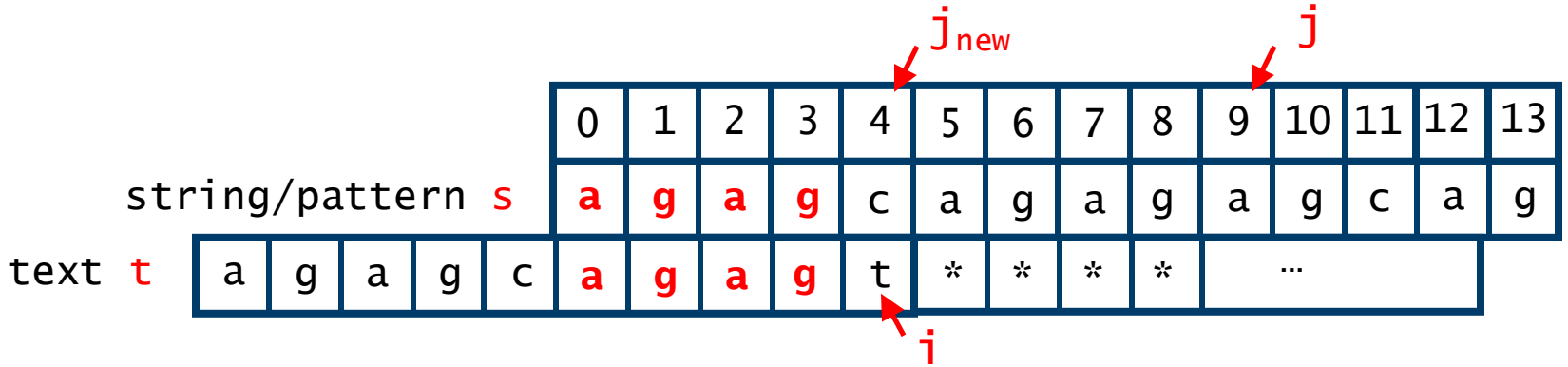


Applying the KMP algorithm, after the mis-match:

- s has to be 'moved along' until the characters to the left of i again match

String search – Brute force versus KMP

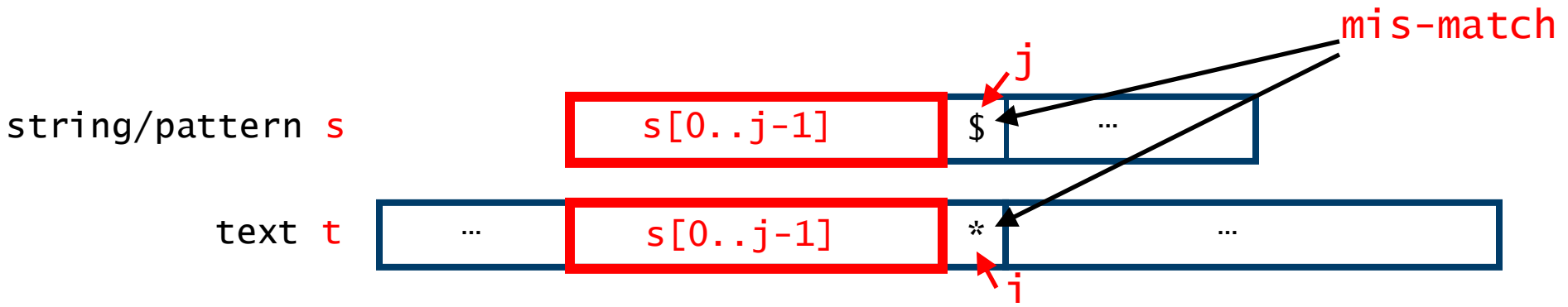
Example – Mismatch between **s** and **t** at position **9** in **s**



Applying the KMP algorithm, after the mis-match:

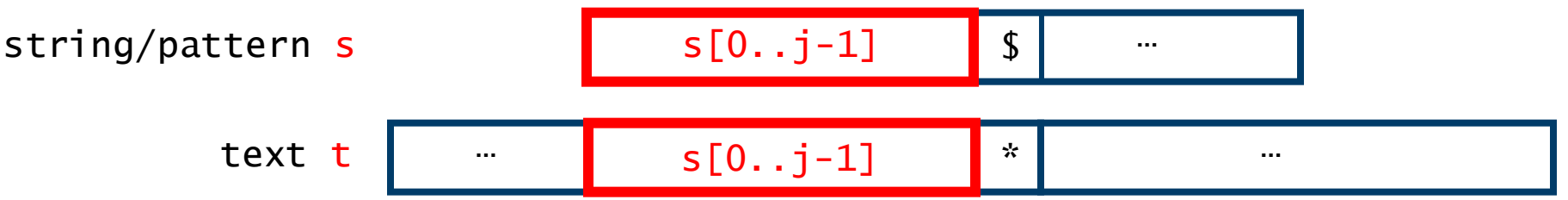
- s has to be 'moved along' until the characters to the left of i again match
- this determines the new value of j, the value of i is unchanged

String search – Brute force versus KMP

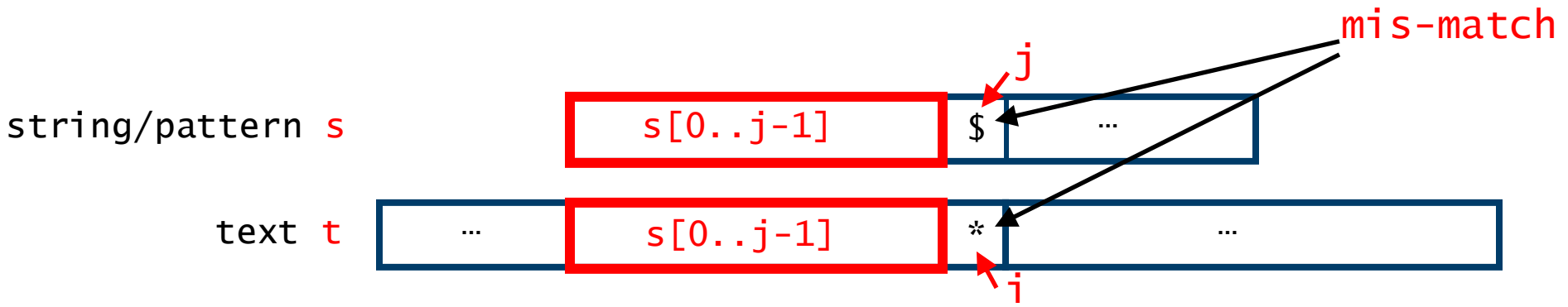


Need to move s along until the characters to the left of i match

- i.e. move s as follows:

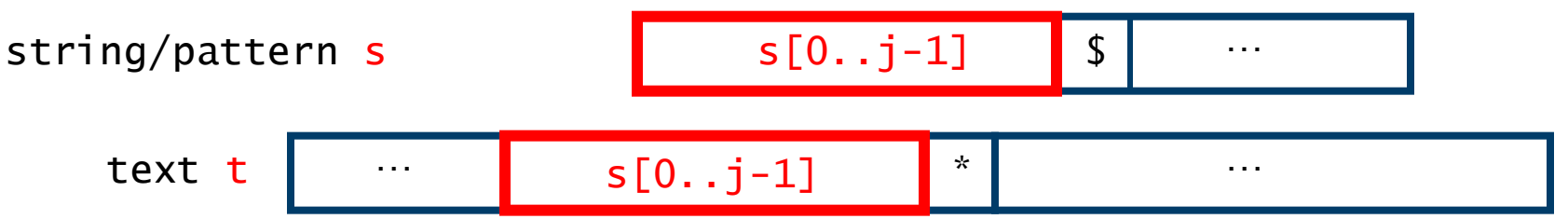


String search – Brute force versus KMP

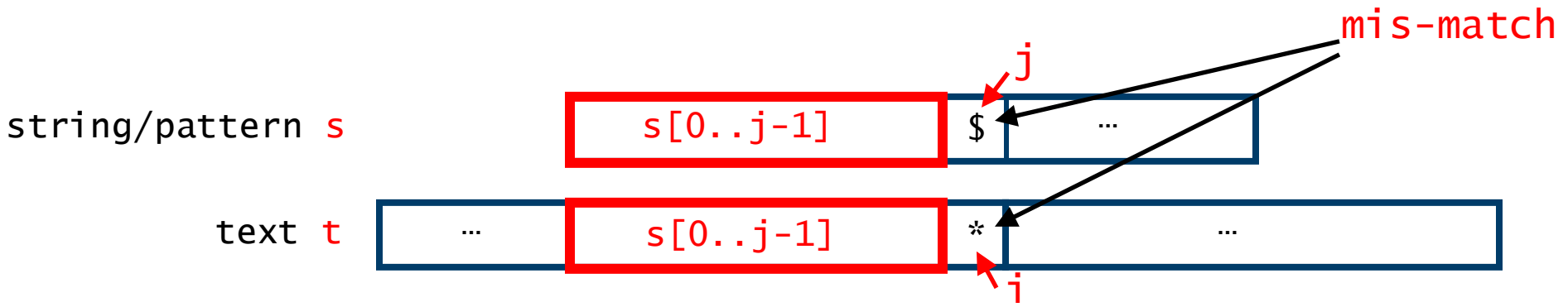


Need to move s along until the characters to the left of i match

- i.e. move s as follows:

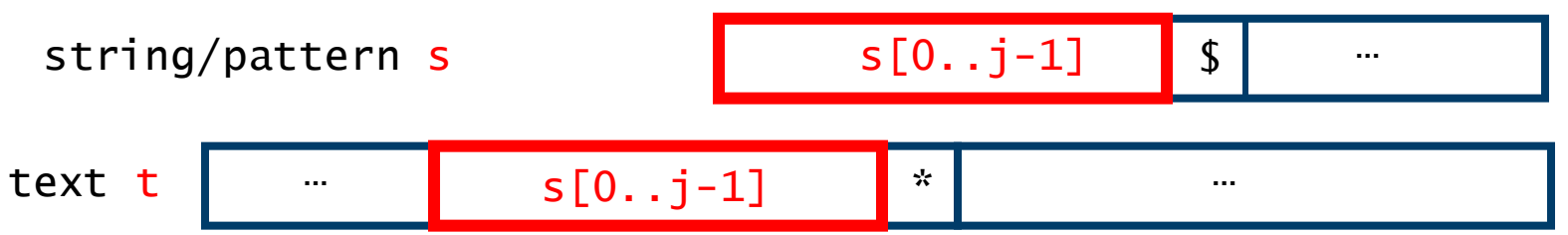


String search – Brute force versus KMP

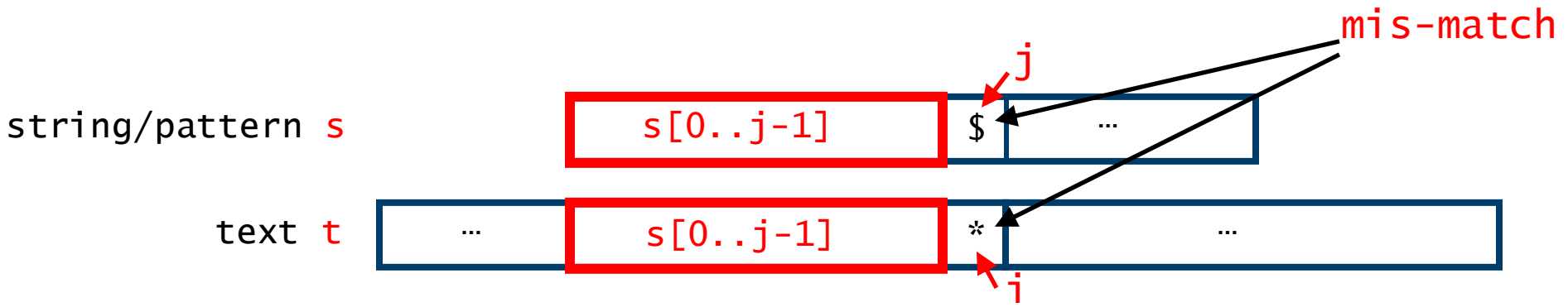


Need to move s along until the characters to the left of i match

- i.e. move s as follows:

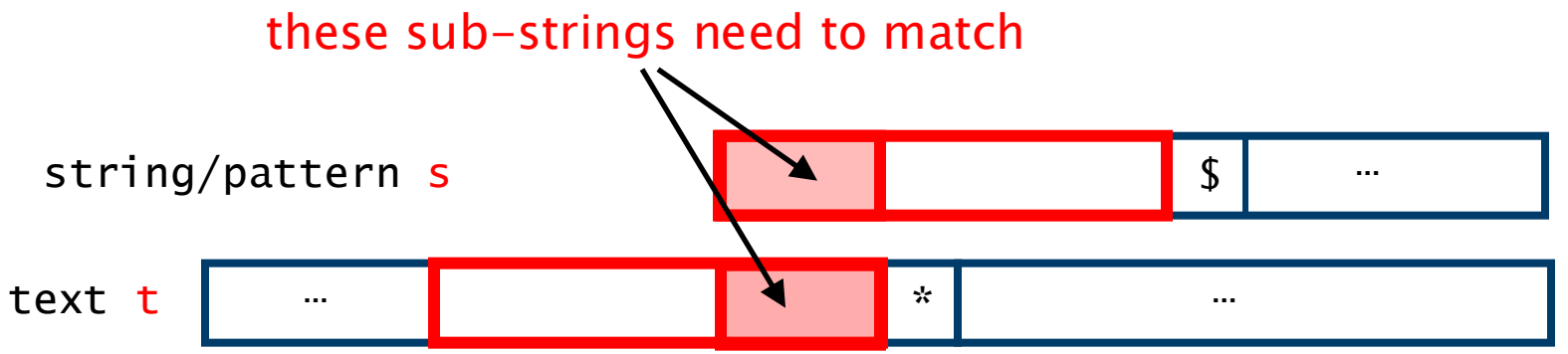


String search – Brute force versus KMP

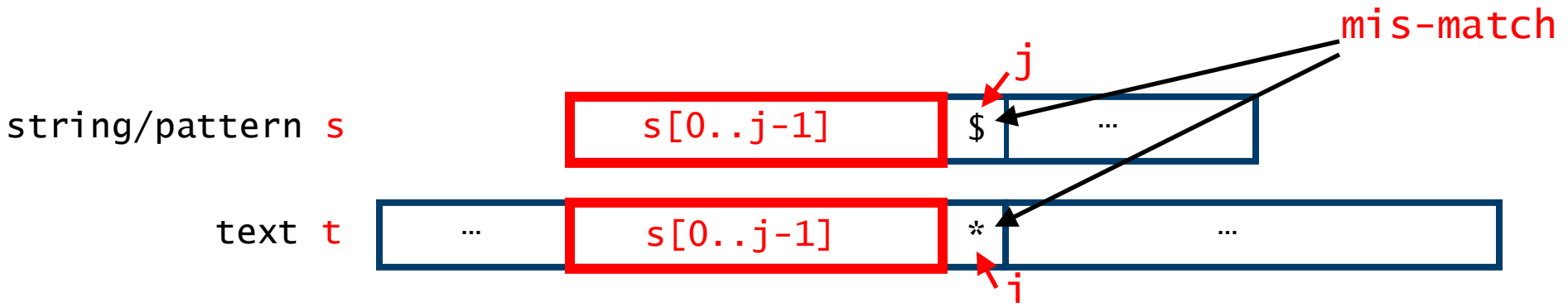


Need to move **s** along until the characters to the left of **i** match

therefore need start of **s[0..j-1]** to match end of **s[0..j-1]**

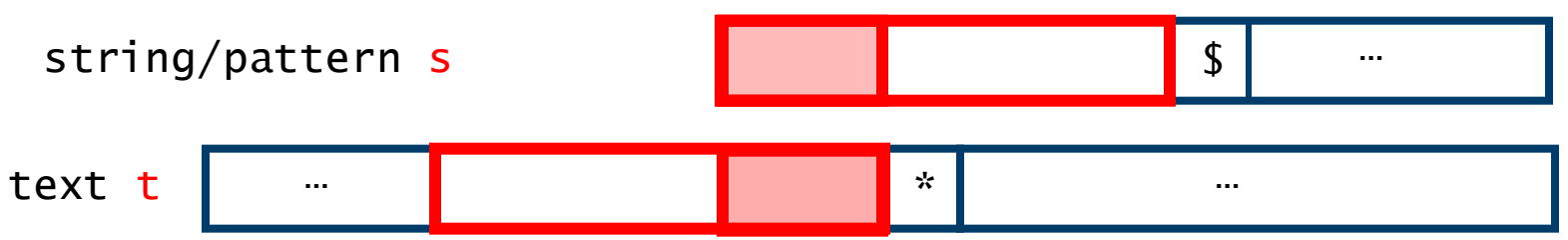


String search – Brute force versus KMP

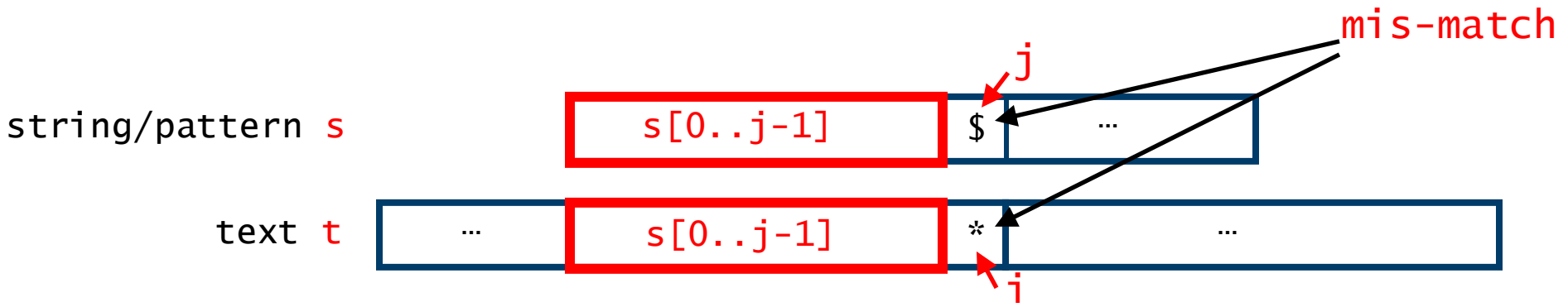


Need to move `s` along until the characters to the left of `i` match
therefore need start of `s[0..j-1]` to match end of `s[0..j-1]`

- therefore use longest border of `s[0..j-1]`
- i.e. longest substring that is both a prefix and a suffix of `s[0..j-1]`

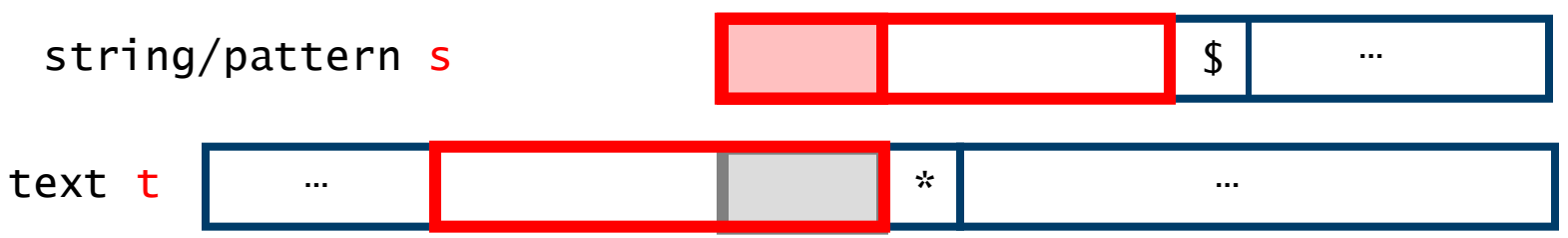


String search – Brute force versus KMP

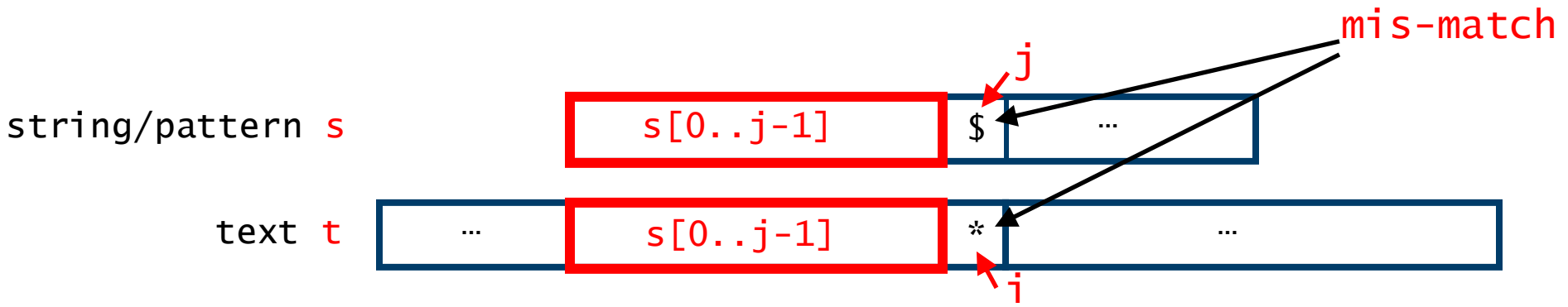


Need to move `s` along until the characters to the left of `i` match
therefore need start of `s[0..j-1]` to match end of `s[0..j-1]`

- therefore use longest border of `s[0..j-1]`
- i.e. longest substring that is both a **prefix** and a suffix of `s[0..j-1]`

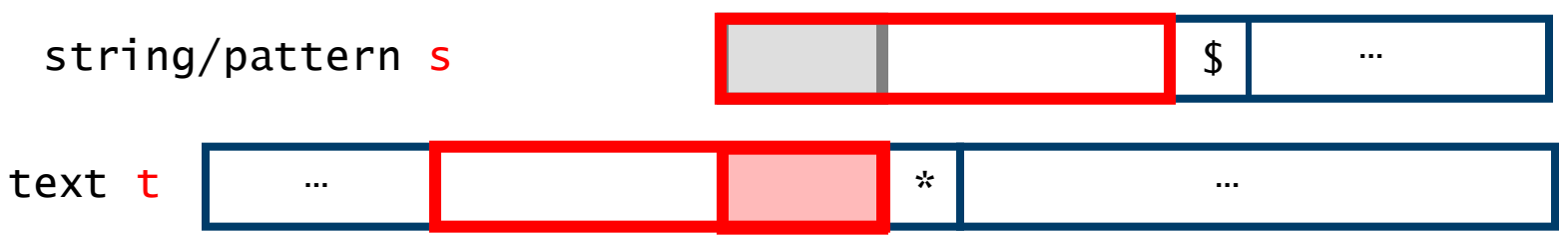


String search – Brute force versus KMP



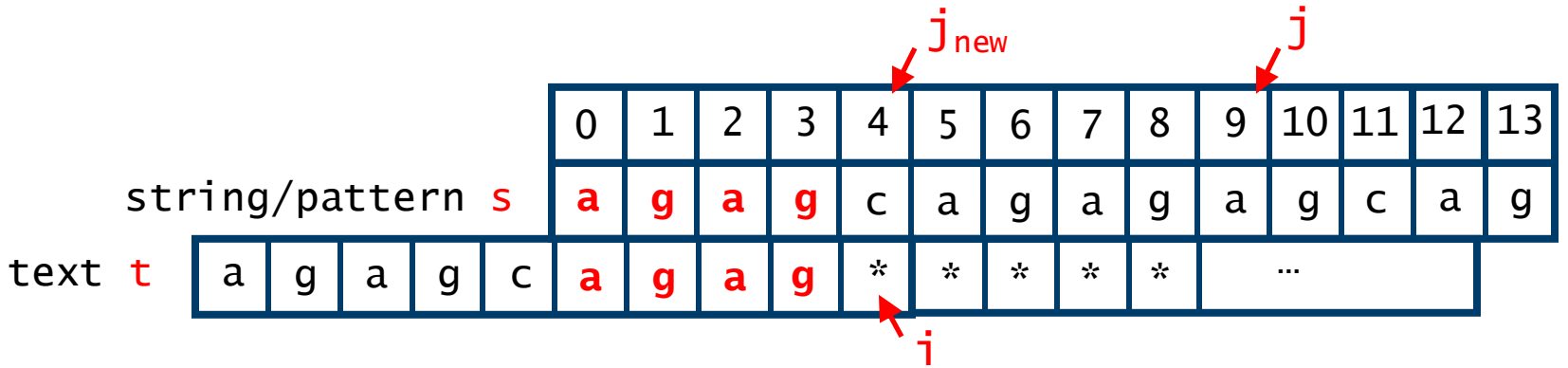
Need to move `s` along until the characters to the left of `i` match
therefore need start of `s[0..j-1]` to match end of `s[0..j-1]`

- therefore use longest border of `s[0..j-1]`
- i.e. longest substring that is both a `prefix` and a **`suffix`** of `s[0..j-1]`



String search – Brute force versus KMP

Example – Mismatch between **s** and **t** at position **9** in **s**

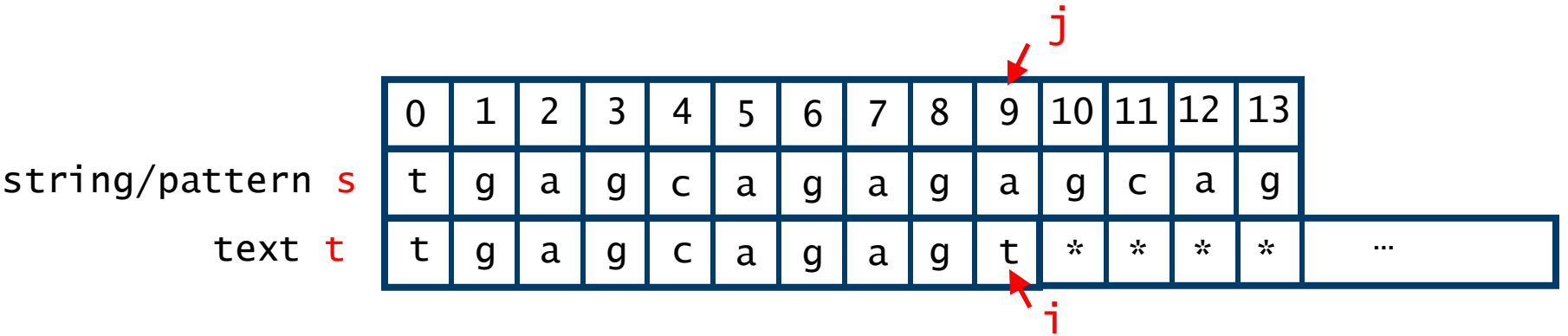


Applying the KMP algorithm, after the mis-match:

- s has to be 'moved along' until the characters to the left of i again match
- this determines the new value of j, the value of i is unchanged
- length of the longest border of s[0..j-1] is 4 in this case
  - i.e. longest substring that is both a prefix and a suffix of s[0..j-1]
- so the new value of j is 4

String search – Brute force versus KMP

Example – Mismatch between **s** and **t** at position **9** in **s**



Applying the KMP algorithm, after the mis-match:

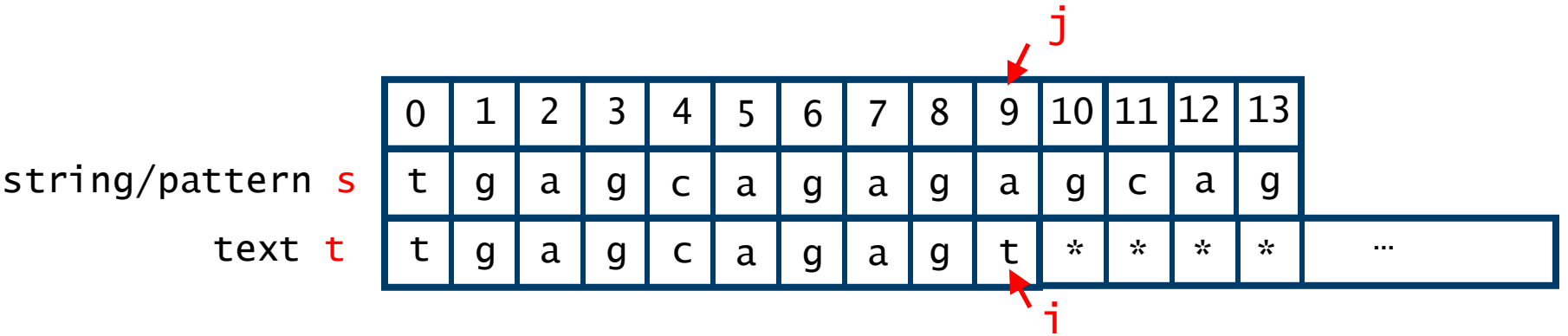
- s has to be 'moved along' until the characters to the left of i again match

If we cannot move **s** along to get a match, then we need to

- reset j (i.e. return to the start of the string) and i remains unchanged

String search – Brute force versus KMP

Example – Mismatch between **s** and **t** at position **9** in **s**



Applying the KMP algorithm, after the mis-match:

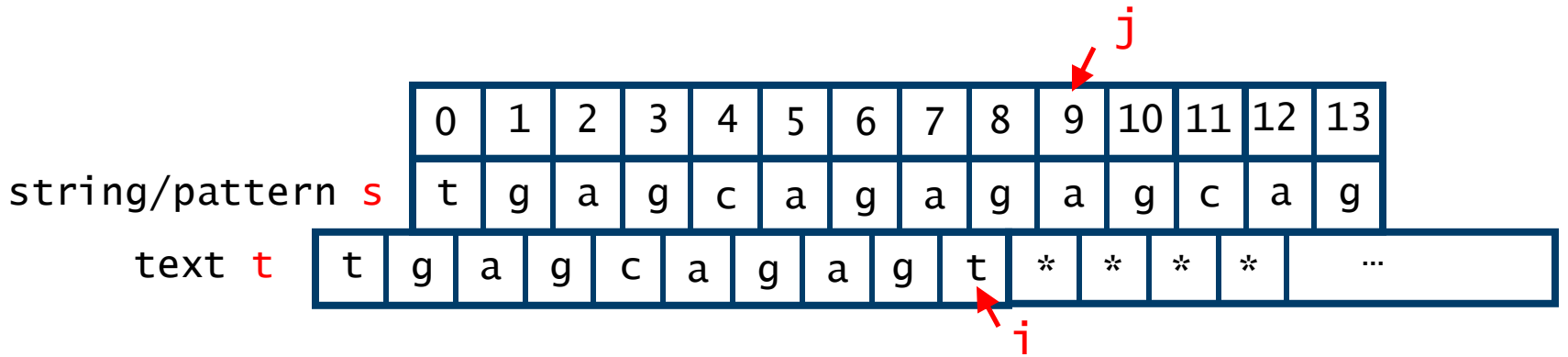
- s has to be 'moved along' until the characters to the left of i again match

If we cannot move **s** along to get a match, then we need to

- reset j (i.e. return to the start of the string) and i remains unchanged

String search – Brute force versus KMP

Example – Mismatch between **s** and **t** at position **9** in **s**



Applying the KMP algorithm, after the mis-match:

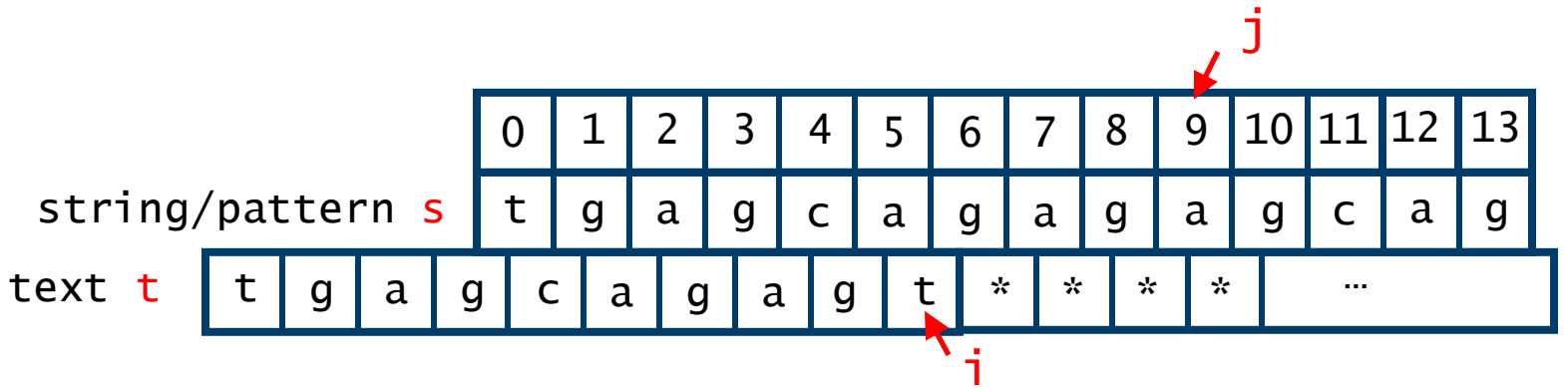
- `s` has to be 'moved along' until the characters to the left of `i` again match

If we cannot move **s** along to get a match, then we need to

- reset `j` (i.e. return to the start of the string) and `i` remains unchanged

String search – Brute force versus KMP

Example – Mismatch between **s** and **t** at position **9** in **s**



Applying the KMP algorithm, after the mis-match:

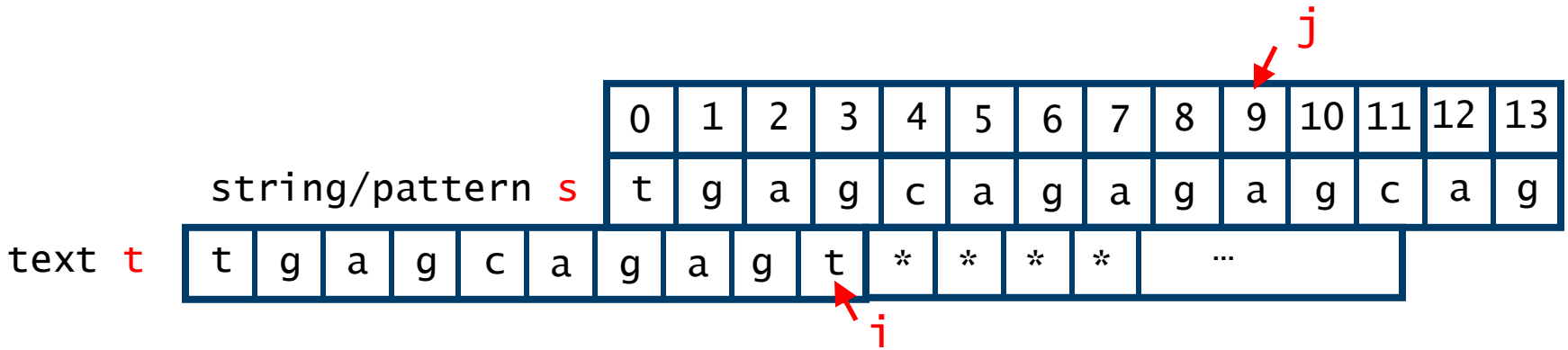
- s has to be 'moved along' until the characters to the left of i again match

If we cannot move **s** along to get a match, then we need to

- reset j (i.e. return to the start of the string) and i remains unchanged

String search – Brute force versus KMP

Example – Mismatch between s and t at position 9 in s



Applying the KMP algorithm, after the mis-match:

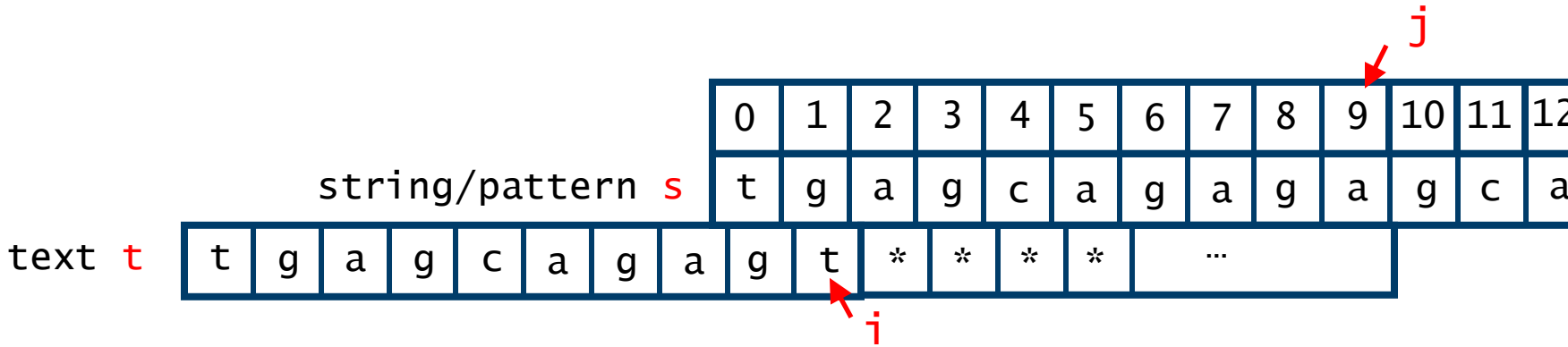
- s has to be 'moved along' until the characters to the left of i again match

If we cannot move s along to get a match, then we need to

- reset j (i.e. return to the start of the string) and i remains unchanged

String search – Brute force versus KMP

Example – Mismatch between **s** and **t** at position **9** in **s**



Applying the KMP algorithm, after the mis-match:

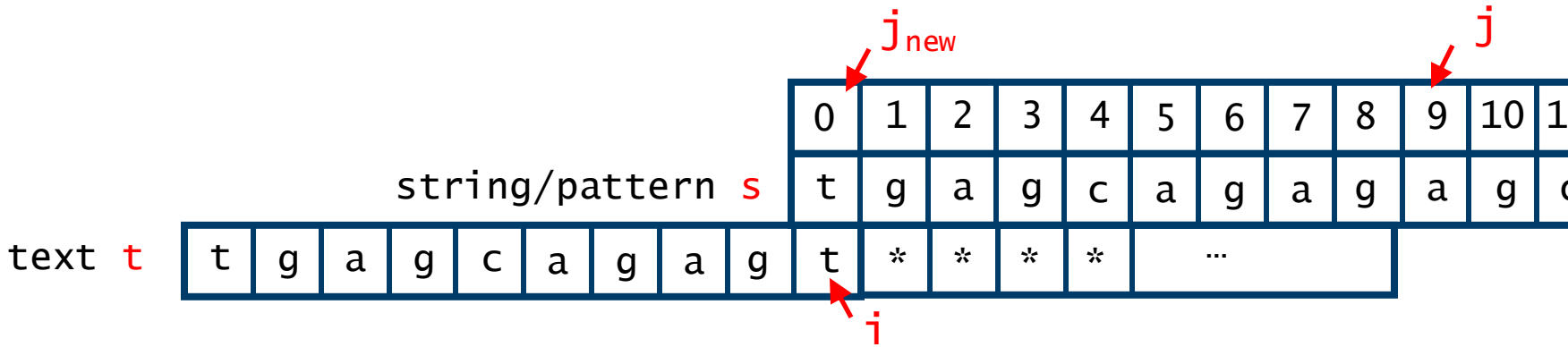
- s has to be 'moved along' until the characters to the left of i again match

If we cannot move **s** along to get a match, then we need to

- reset j (i.e. return to the start of the string) and i remains unchanged

String search – Brute force versus KMP

Example – Mismatch between **s** and **t** at position **9** in **s**



Applying the KMP algorithm, after the mis-match:

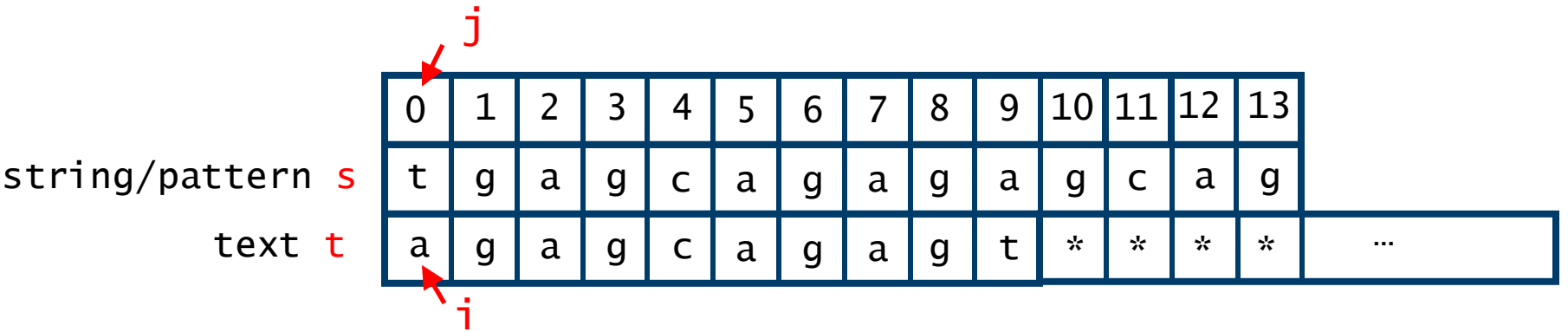
- s has to be 'moved along' until the characters to the left of i again match

If we cannot move **s** along to get a match, then we need to

- reset j (i.e. return to the start of the string) and i remains unchanged

String search – Brute force versus KMP

Example – Mismatch between s and t at position 0 in s



Applying the KMP algorithm, after the mis-match:

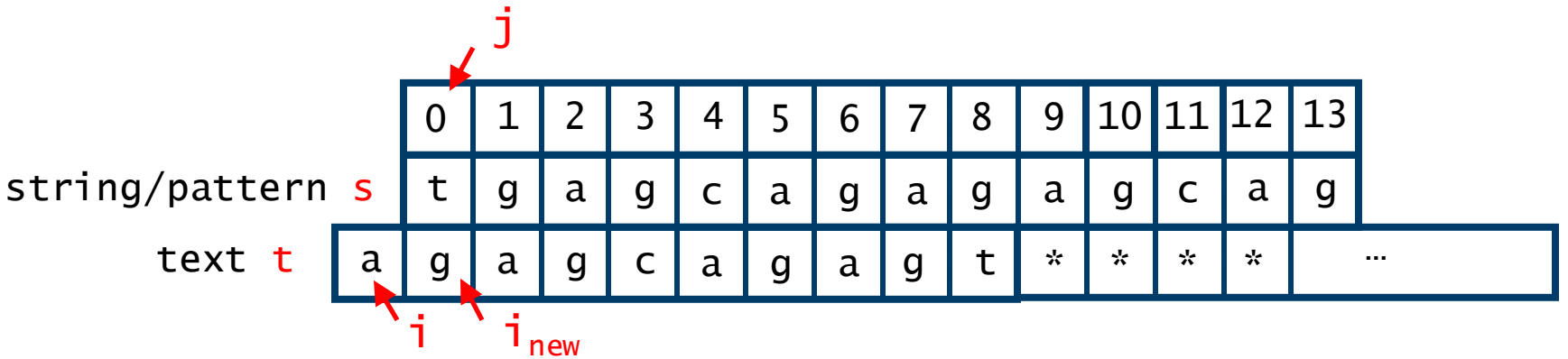
- s has to be 'moved along' until the characters to the left of i again match

If we cannot move s along to get a match, then we need to

- reset j (i.e. return to the start of the string) and i remains unchanged
- unless j is already 0 and in this case increment i

String search – Brute force versus KMP

Example – Mismatch between s and t at position 0 in s



Applying the KMP algorithm, after the mis-match:

- s has to be 'moved along' until the characters to the left of i again match

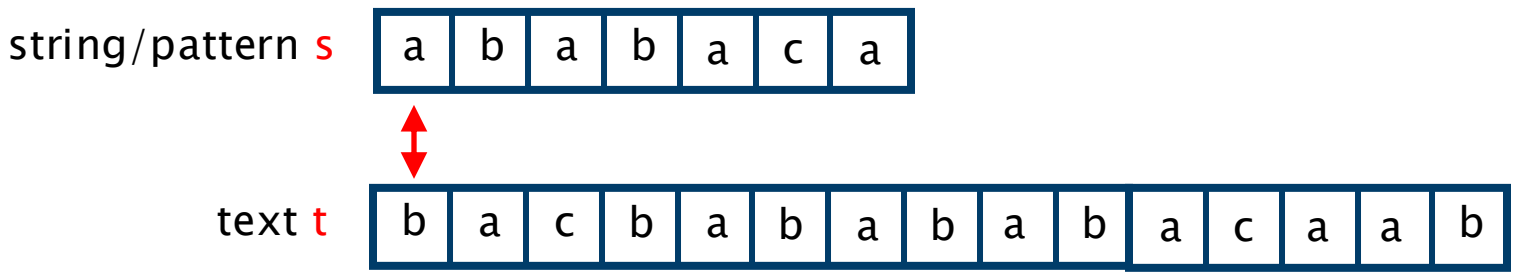
If we cannot move s along to get a match, then we need to

- reset j (i.e. return to the start of the string) and i remains unchanged
- unless j is already 0 and in this case increment i

KMP search – Implementation

```
/** return smallest k such that s occurs from position k in t or -1 if no k exists */
public int kmp(char[] t, char[] s) {
   int m = s.length; // length of string/pattern
   int n = t.length; // length of text
   int i = 0; // current position in text
   int j = 0; // current position in string s
   int [] b = new int[m]; // create border table
   setUp(b); // set up the border table
   while (i <= n) { // not reached end of text
      if (t[i] == s[j]){ // if positions match
         i++; // move on in text
         j++; // move on in string
        if (j = m) return i - j; // reached end of string so a match
      } else { // mismatch adjust current position in string using the border table
          if (b[j] > 0) // there is a common prefix/suffix
              j = b[j]; // change position in string (position in text unchanged)
          else { // no common prefix/suffix
              if (j = 0) i++; // move forward one position in text if not advanced
              else j = 0; // else start from beginning of the string
          }
      }
   }
   return -1; // no occurrence
}
```

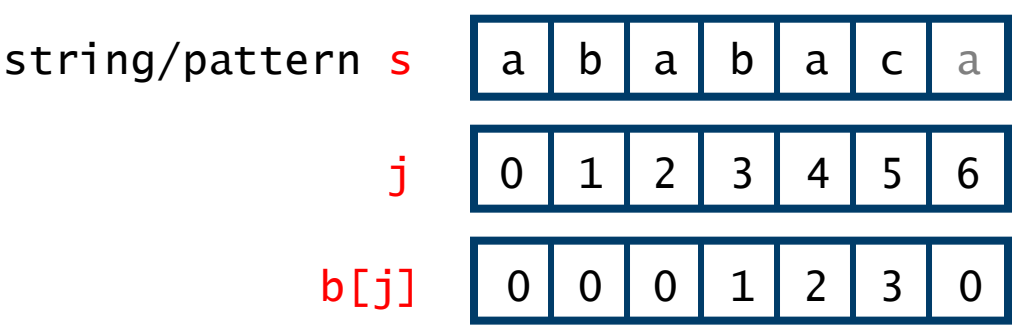
KMP – Example



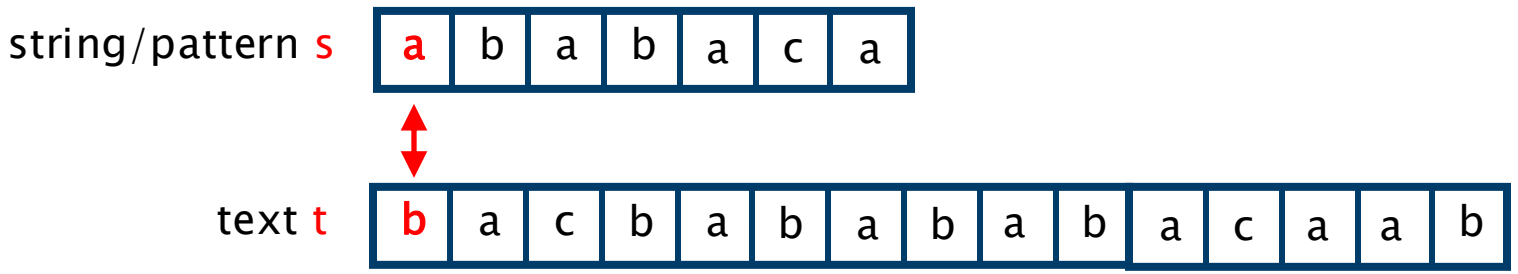
Starting position:

- start of text and string

position in string j=0



KMP – Example

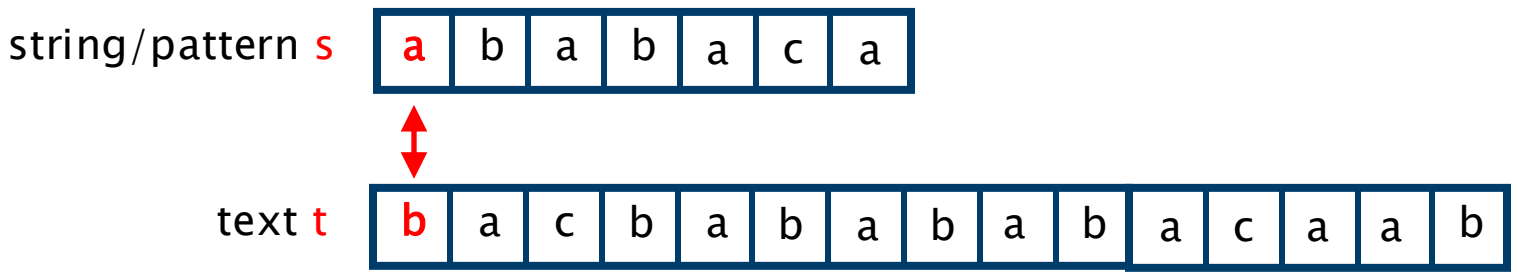


Compare characters in text and string

position in string j=0

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

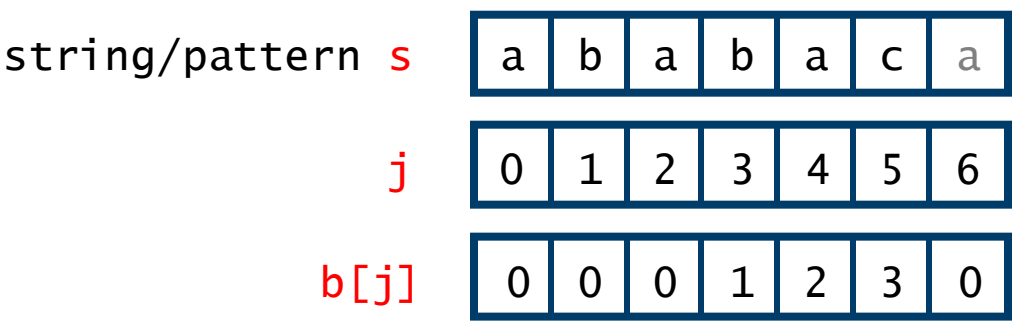
KMP – Example



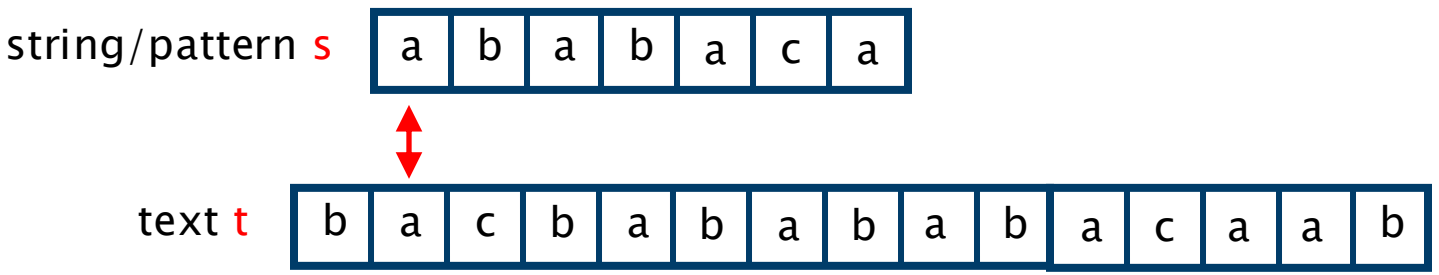
Characters do not match

position in string j=0

- and border table for b[j]=b[0] is 0
- j=0 so increment position text



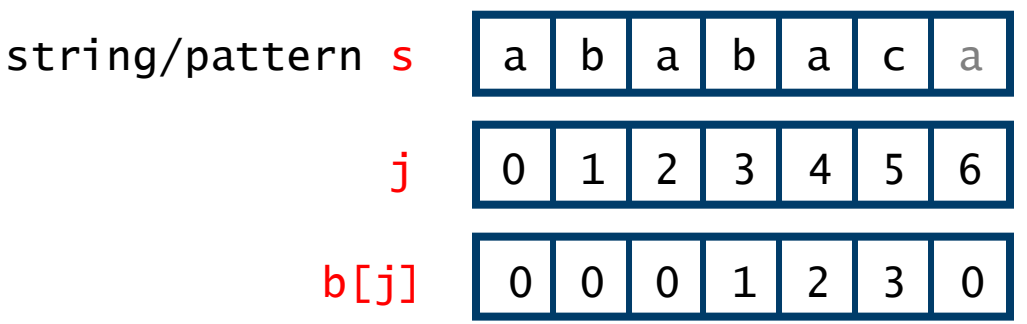
KMP – Example



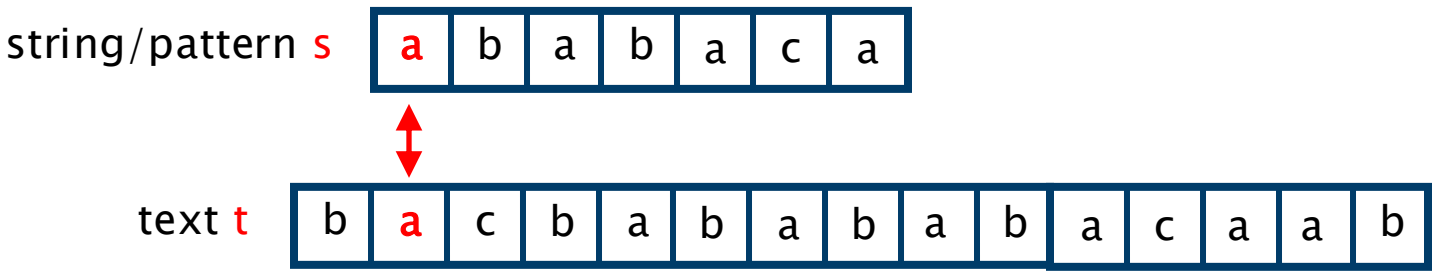
Characters do not match

position in string j=0

- and border table for b[j]=b[0] is 0
- j=0 so increment position text

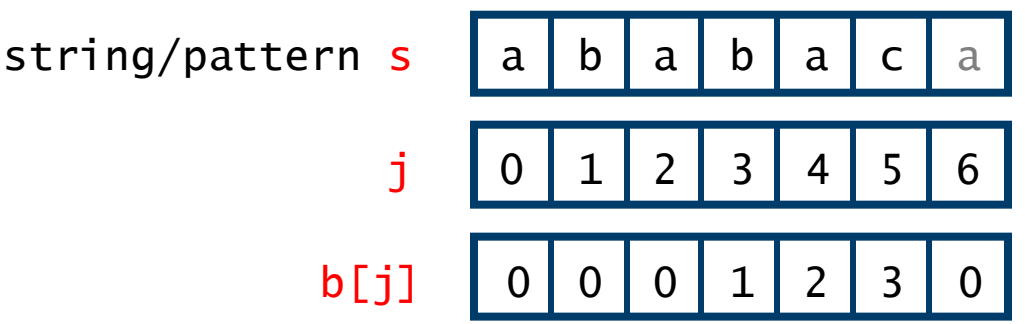


KMP – Example

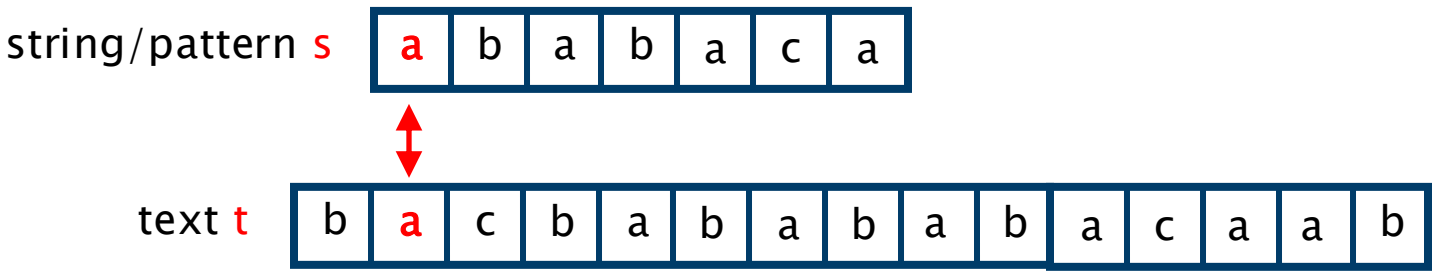


Compare characters in text and string

position in string j=0



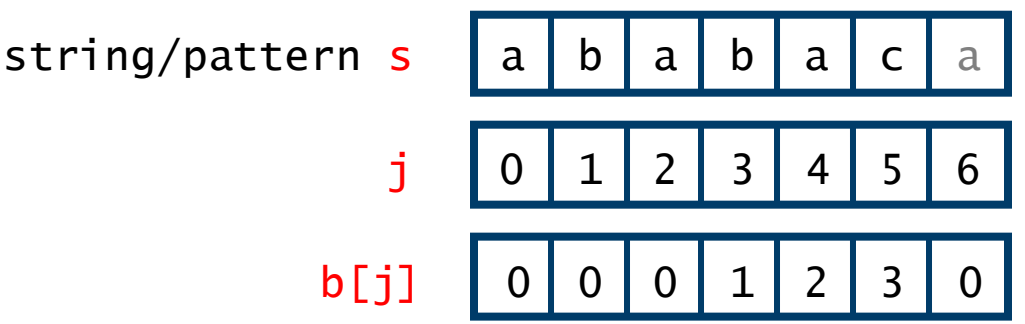
KMP – Example



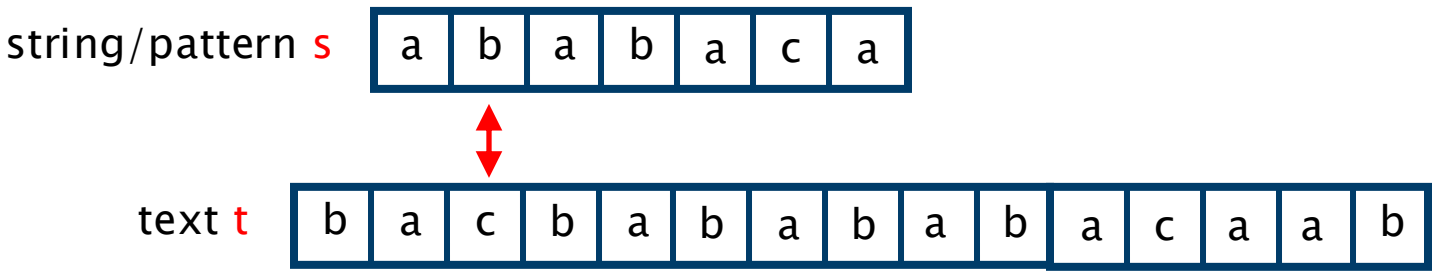
Characters match

position in string j=0

- increment position in text and string



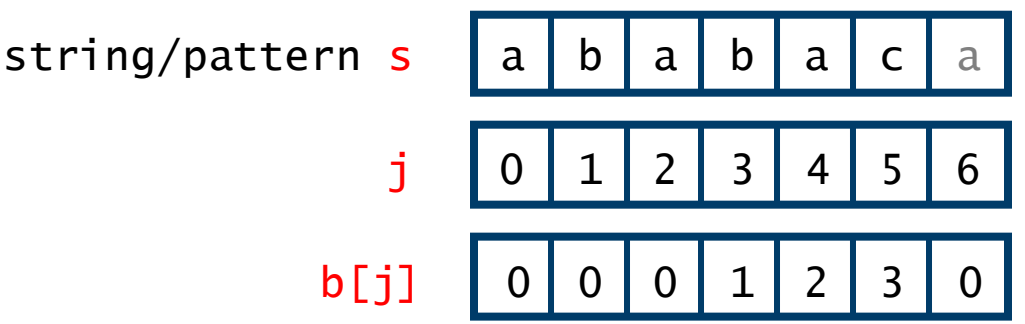
KMP – Example



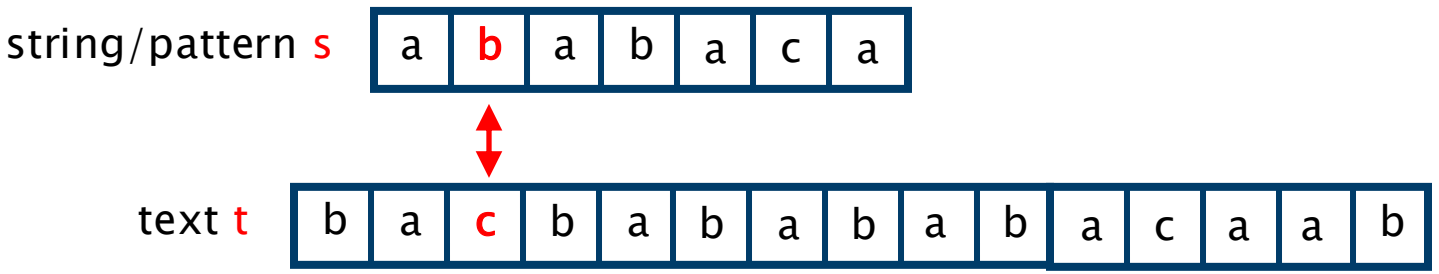
Characters match

position in string j=0

- increment position in text

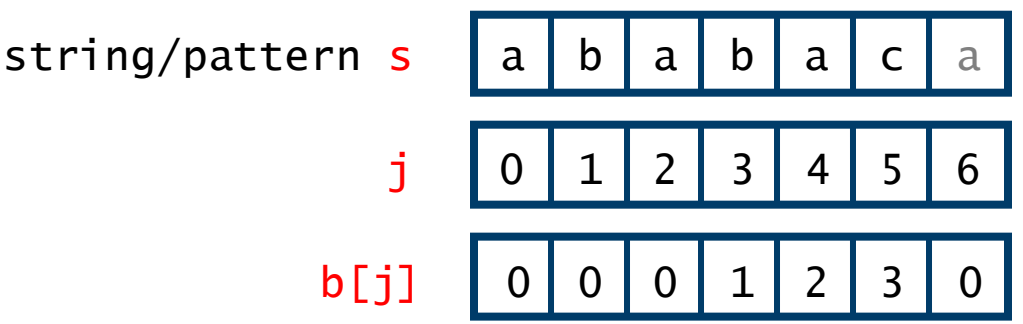


KMP – Example

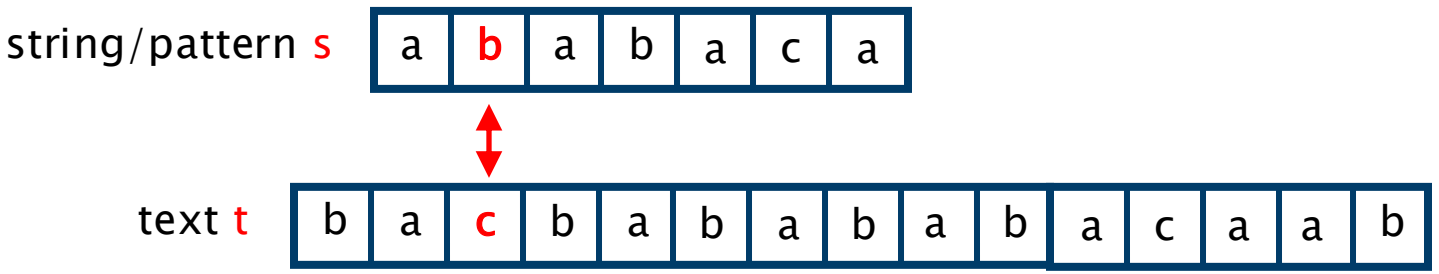


Compare characters in text and string

position in string j=1



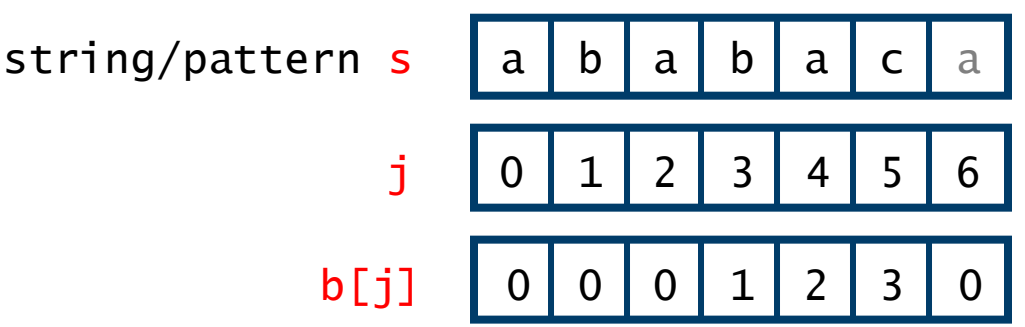
KMP – Example



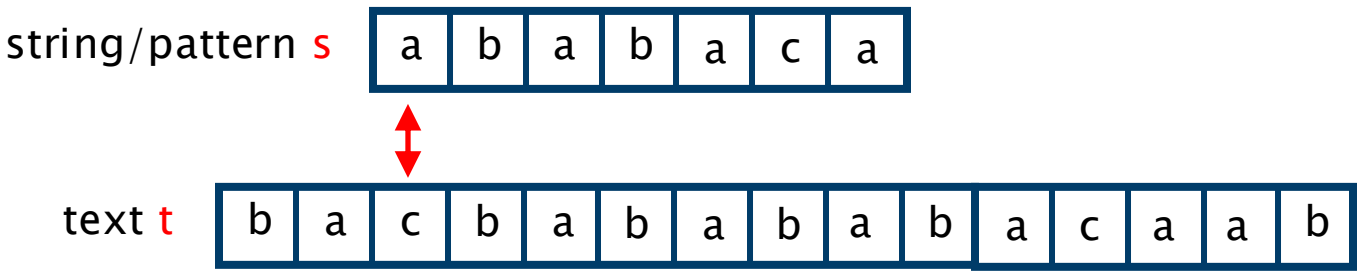
Characters do not match

position in string j=1

- border table for b[j]=b[1] is 0
- j>0 so start again in string



KMP – Example



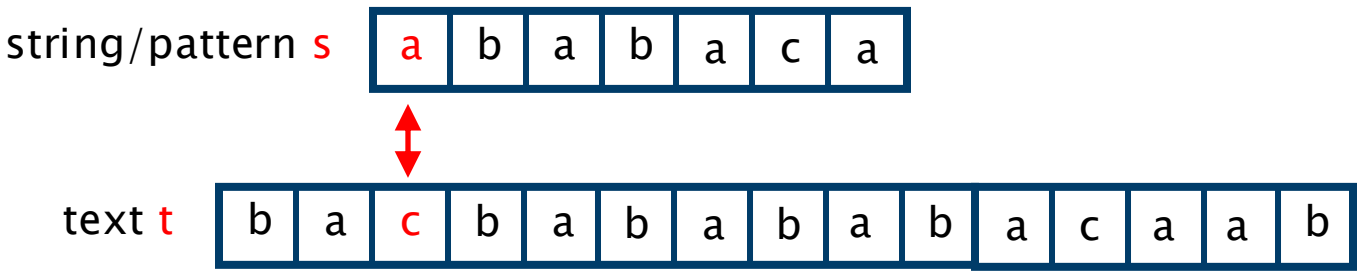
Characters do not match

position in string j=0

- border table for b[j]=b[1] is 0
- j>0 so start again in string

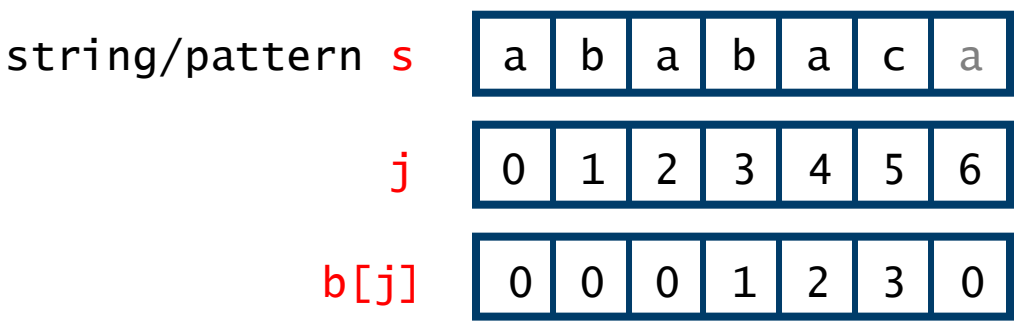
string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

KMP – Example

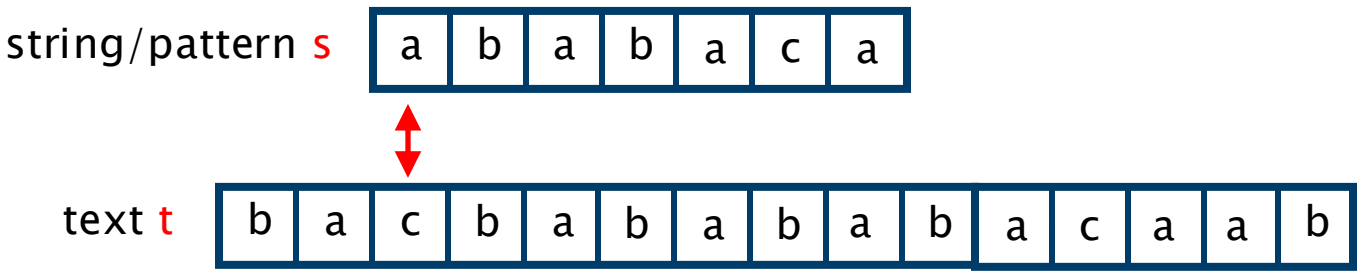


Compare characters in text and string

position in string j=0



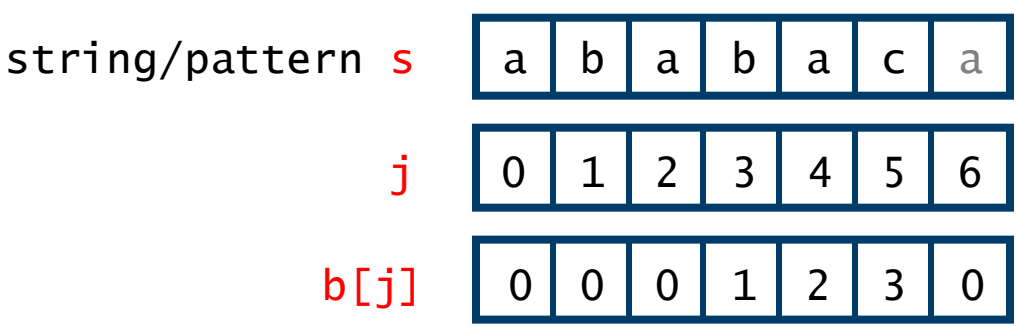
KMP – Example



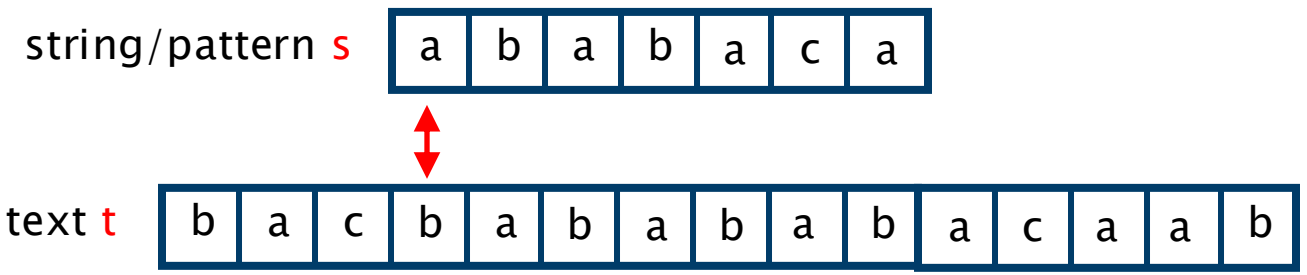
Compare characters in text and string

position in string j=0

- and border table for b[j]=b[0] is 0
- j=0 so increment position text



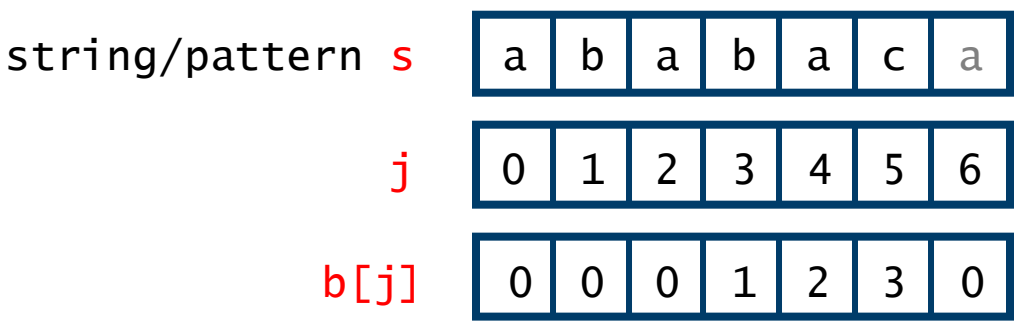
KMP – Example



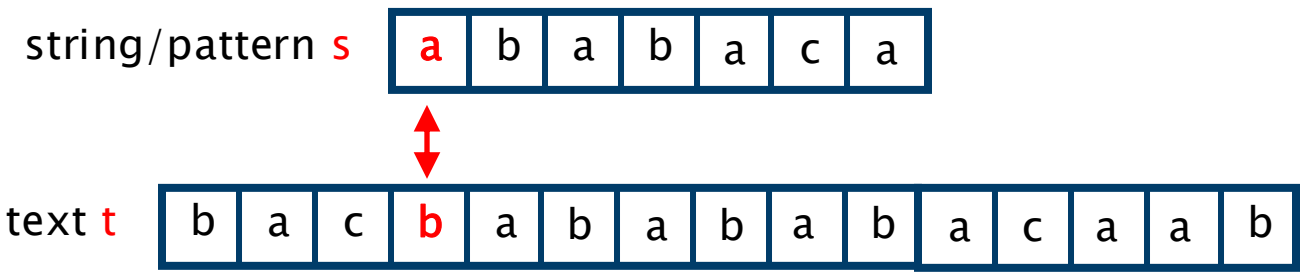
Compare characters in text and string

position in string j=0

- and border table for b[j]=b[0] is 0
- j=0 so increment position text

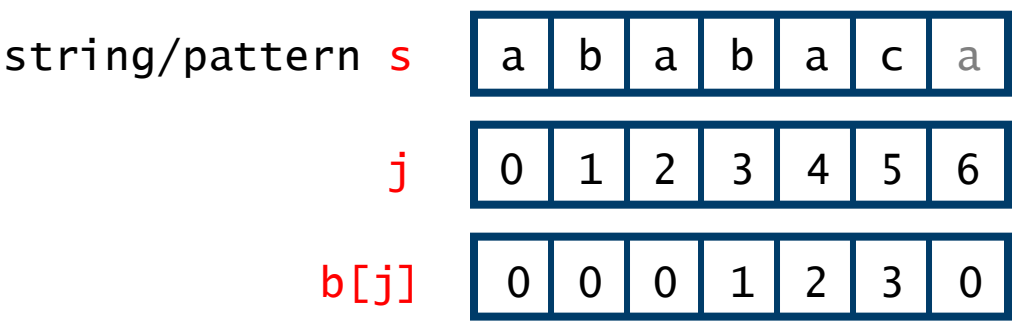


KMP – Example

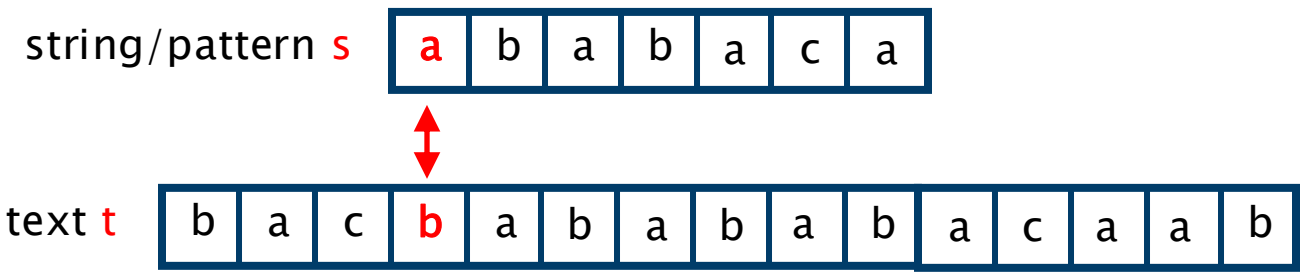


Compare characters in text and string

position in string j=0



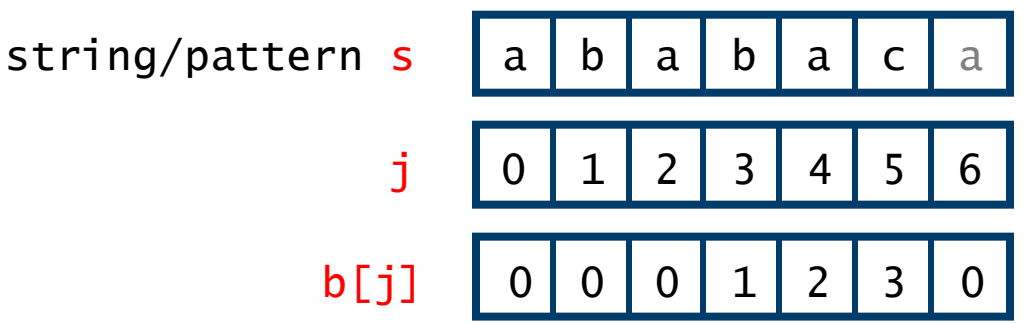
KMP – Example



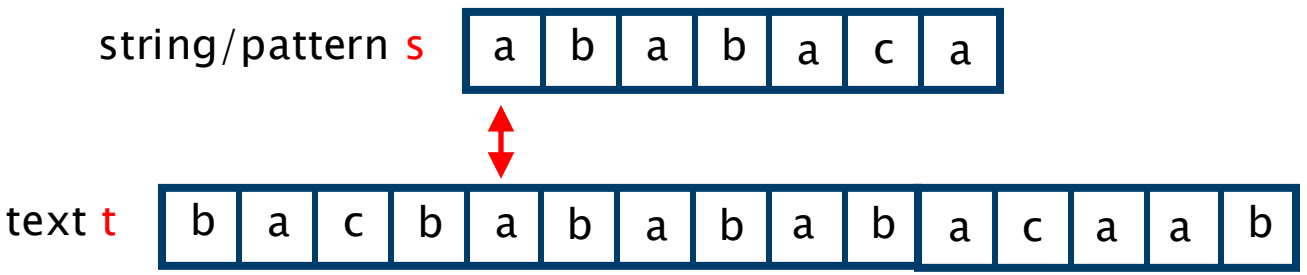
Characters do not match

position in string j=0

- and border table for b[j]=b[0] is 0
- j=0 so increment position text



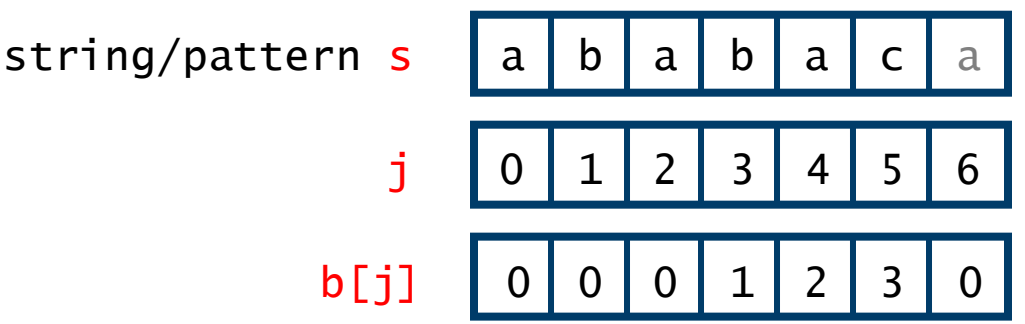
KMP – Example



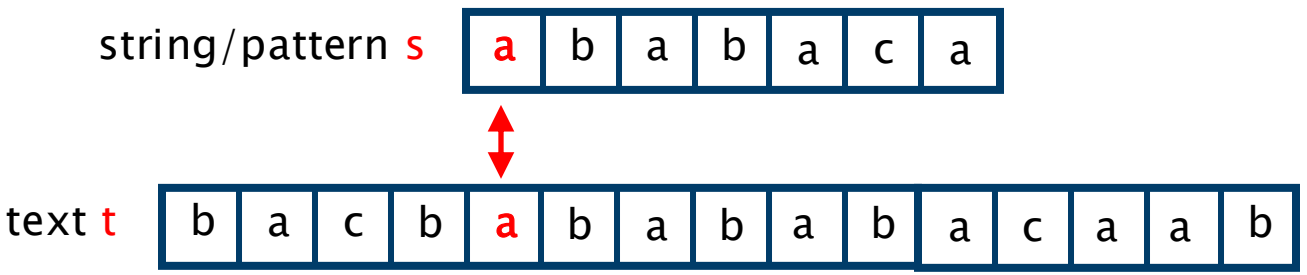
Characters do not match

position in string j=0

- and border table for b[j]=b[0] is 0
- j=0 so increment position text

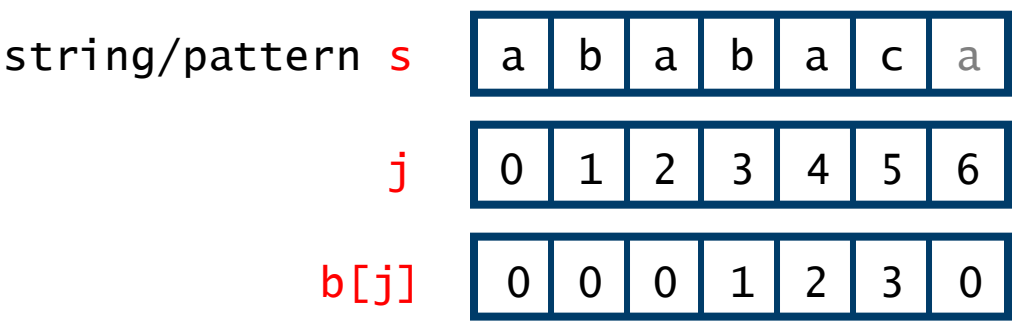


KMP – Example

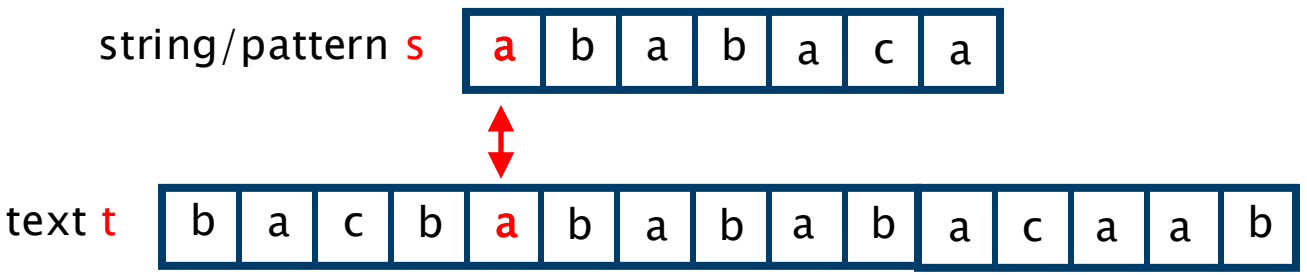


Compare characters in text and string

position in string j=0



KMP – Example



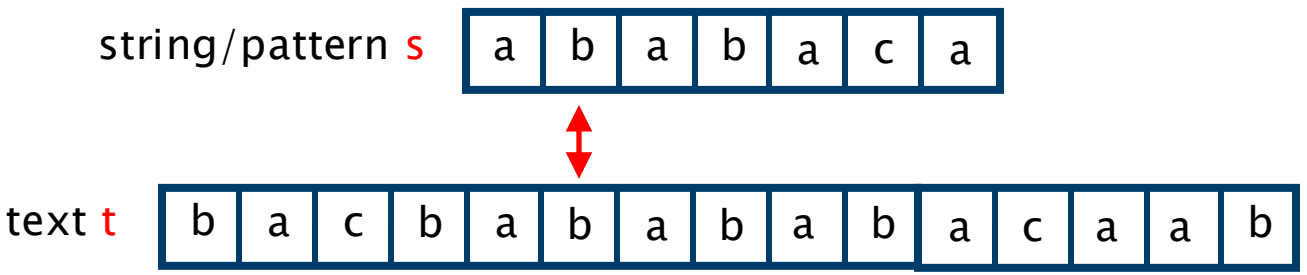
Characters match

position in string j=0

– increment position in text and string

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

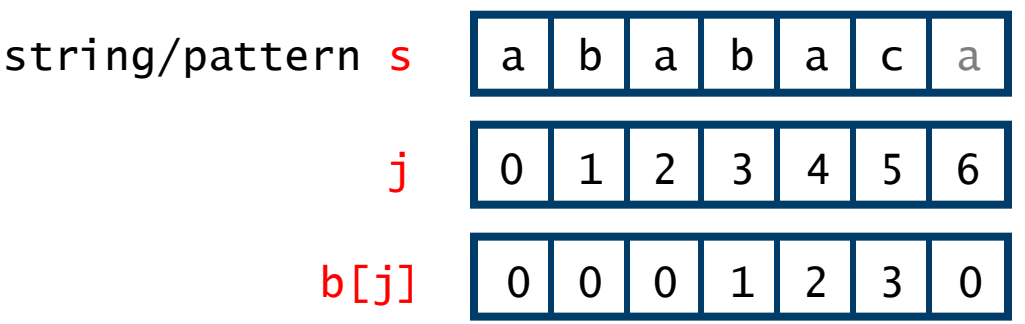
KMP – Example



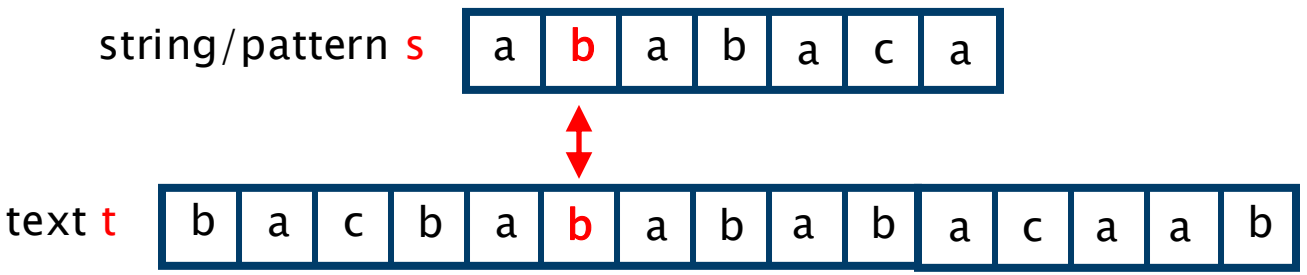
Characters match

position in string j=1

- increment position in text and string



KMP – Example

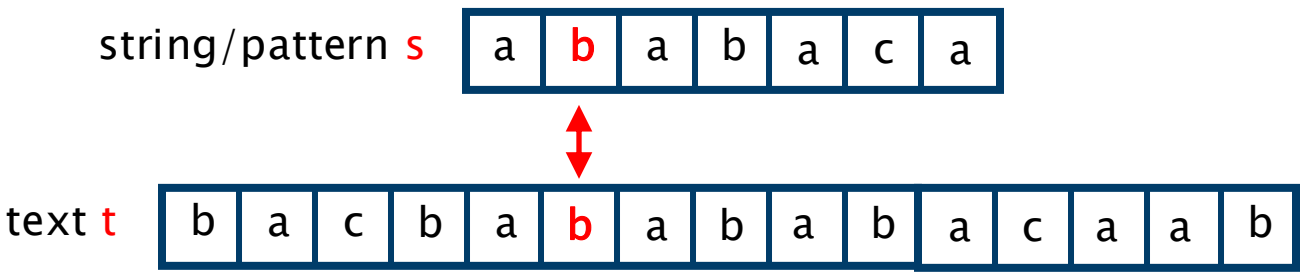


Compare characters in text and string

position in string j=1

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

KMP – Example



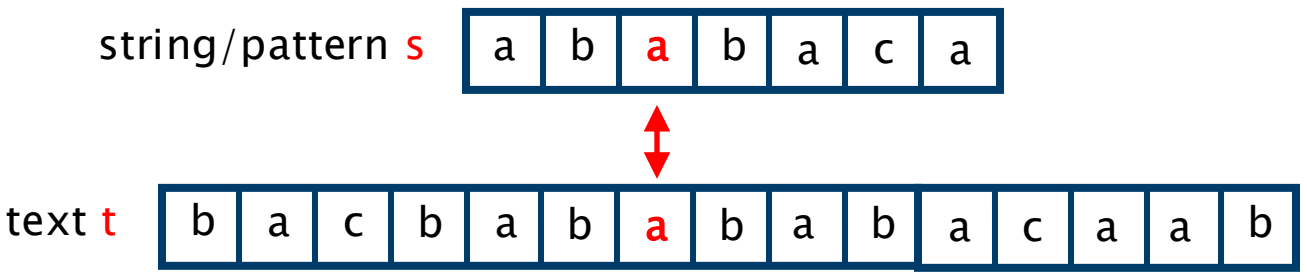
Characters match

position in string j=1

– increment position in text and string

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

KMP – Example



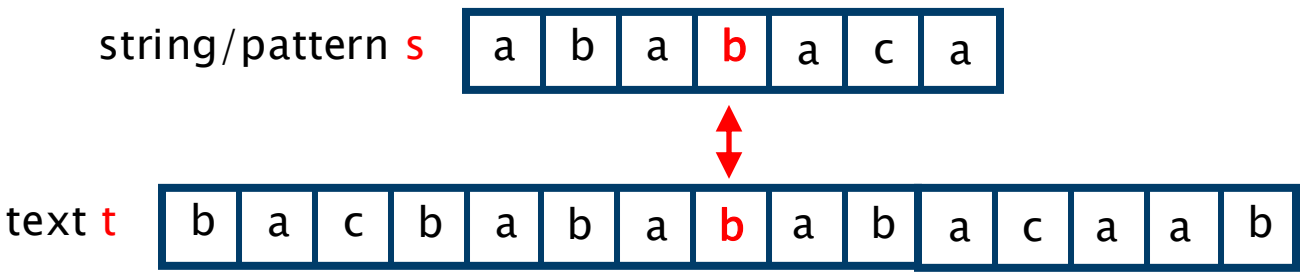
Characters continue to match so

position in string j=2

- increment position in text and string

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

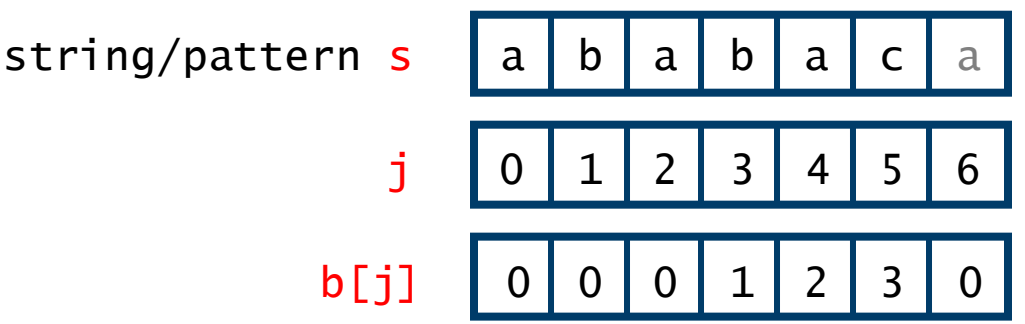
KMP – Example



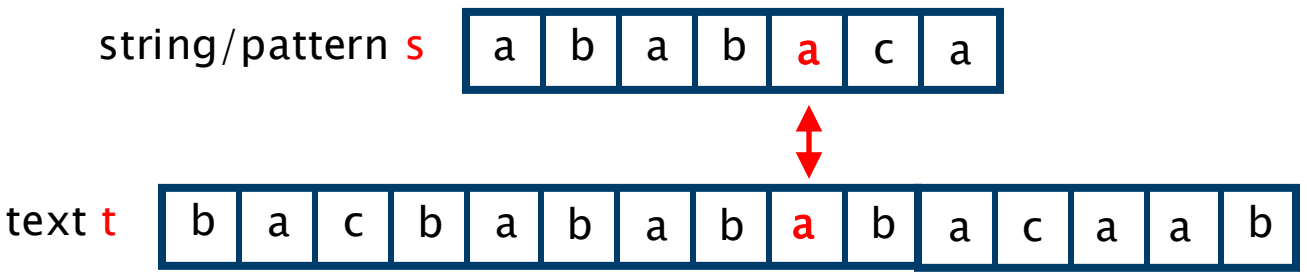
Characters continue to match so

– increment position in text and string

position in string j=3



KMP – Example



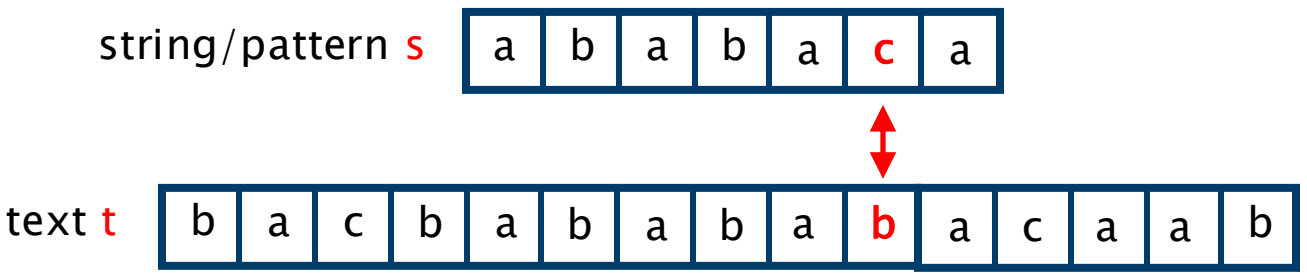
Characters continue to match so

position in string j=4

- increment position in text and string

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

KMP – Example



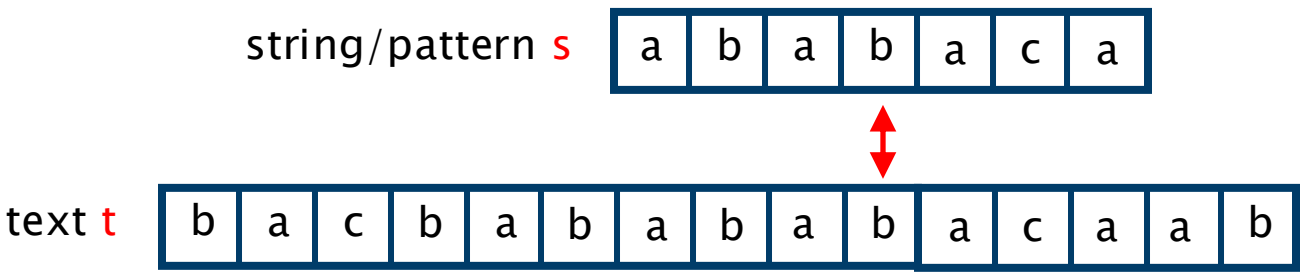
Characters do not match

position in string j=5

- and border table for b[j]=b[5] is 3
- therefore
  - j set equal to b[j]=3
  - position in text unchanged

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

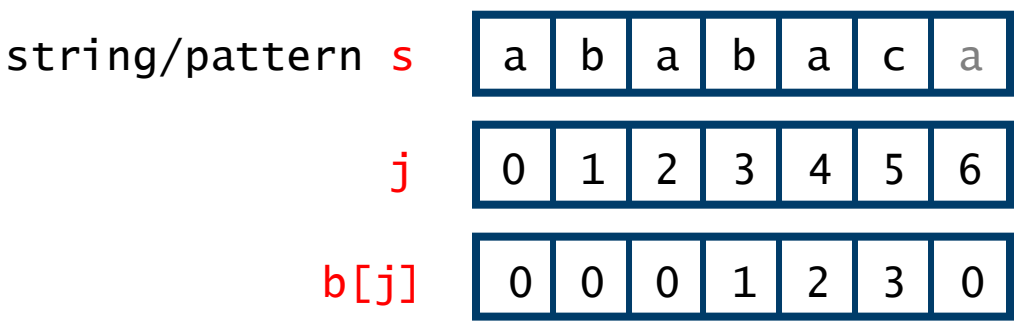
KMP – Example



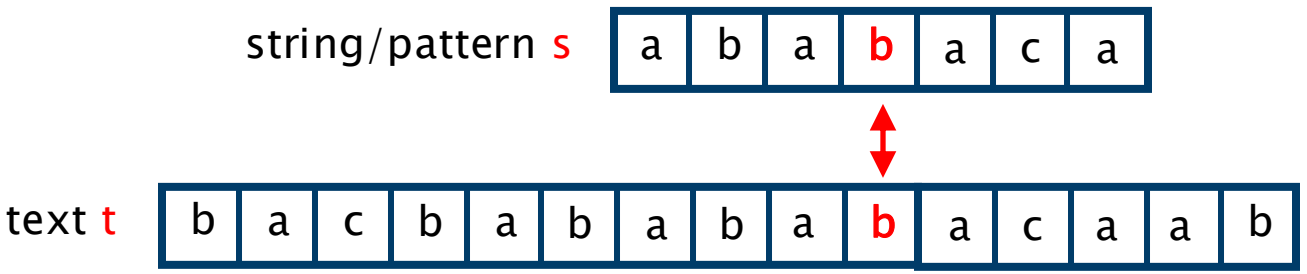
Characters do not match

position in string j=3

- and border table for b[j]=b[5] is 3
- therefore
  - j set equal to b[j]=3
  - position in text unchanged



KMP – Example

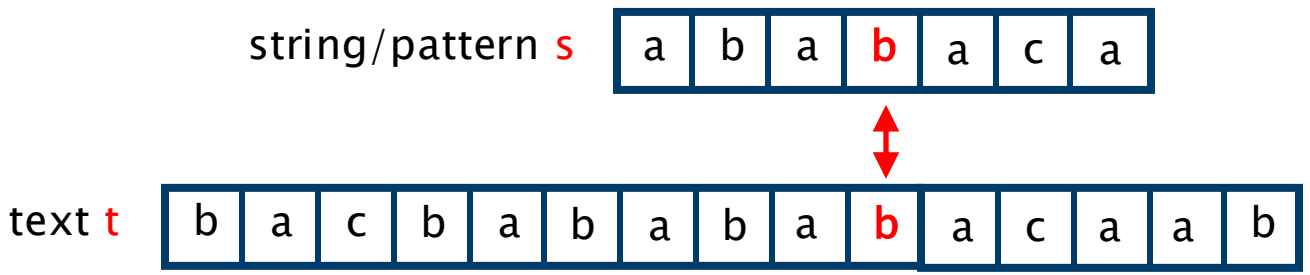


Compare characters in text and string

position in string j=3

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

KMP – Example



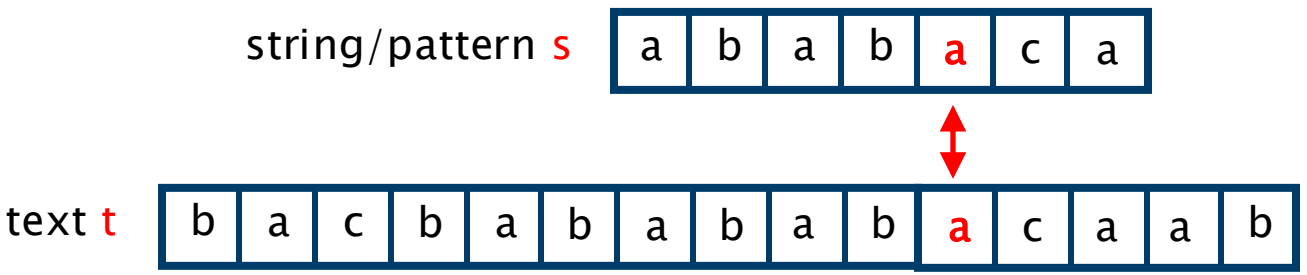
Characters match

position in string j=3

- increment position in text and string

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

KMP – Example



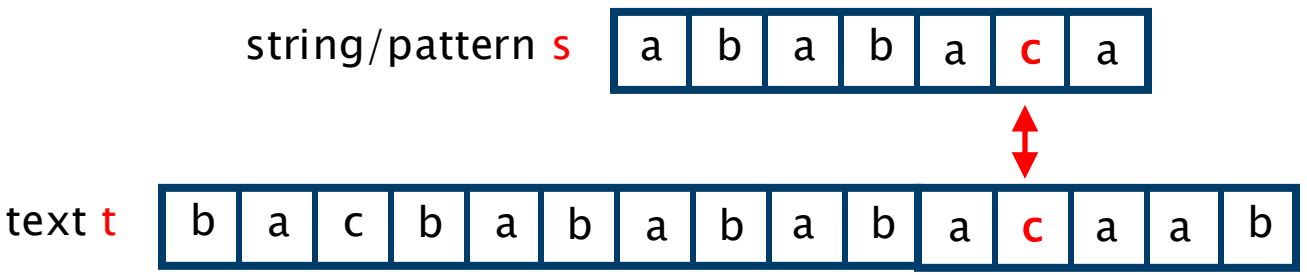
Characters continue to match so

position in string j=4

- increment position in text and string

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

KMP – Example



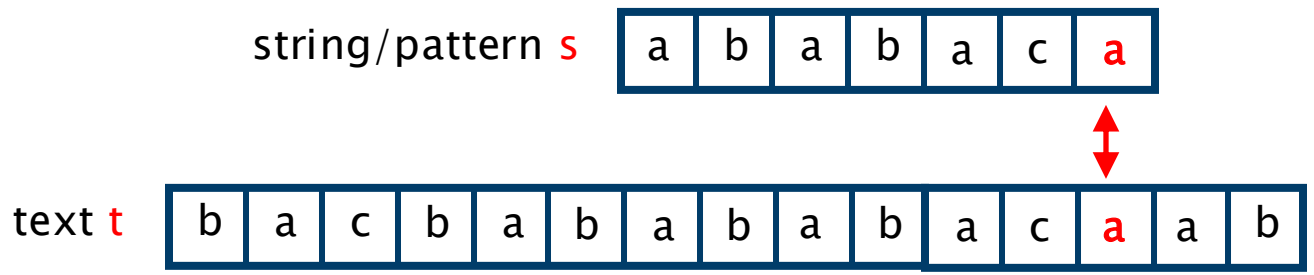
Characters continue to match so

position in string j=5

- increment position in text and string

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

KMP – Example



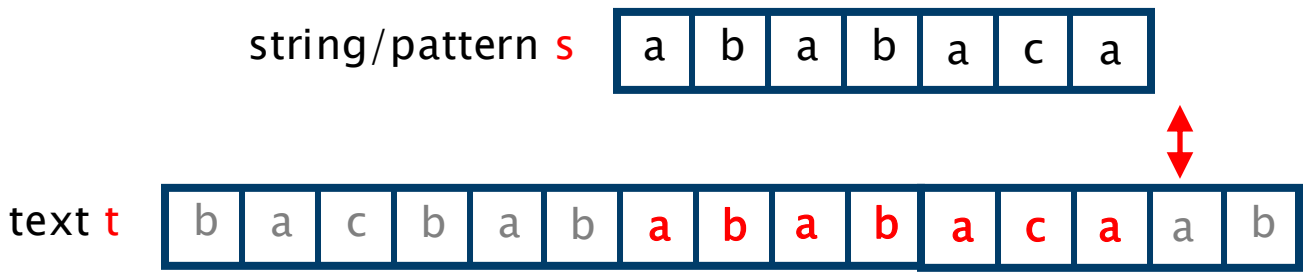
Characters continue to match so

position in string j=6

- increment position in text and string

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

KMP – Example



String/pattern has been found

position in string j=7

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

KMP search – Analysis

```
while (i<n)
   if (t[i] == s[j]){
       i++; j++;
   }
   else {
       if (b[j]>0) j = b[j];
       else {
           if (j=0) i++;
           else j = 0;
       }
    }
```

For the complexity we need to know the number of loop iterations

Consider values of **i** and **k** (where **k=i-j**) during the iterations

- clearly $i \leq n$ and since j is never negative we also have $k \leq n$
- in each iteration either i or k is incremented and neither is decremented

KMP search – Analysis

```
while (i<n)
    if (t[i] == s[j]){
        i++; j++;
    }
    else {
        if (b[j]>0) j = b[j];
        else {
            if (j=0) i++;
            else j = 0;
        }
     }
```

For the complexity we need to know the number of loop iterations

Consider values of **i** and **k** (where **k=i-j**) during the iterations

- clearly $i \leq n$ and since j is never negative we also have $k \leq n$
- in each iteration either i or k is incremented and neither is decremented
  - `i++ > i` and `(i++)-(j++) = i-j`

KMP search – Analysis

```
while (i<n)
   if (t[i] == s[j]){
       i++; j++;
   }
   else {
       if (b[j]>0) j = b[j];
       else {
           if (j=0) i++;
           else j = 0;
       }
    }
```

For the complexity we need to know the number of loop iterations

Consider values of **i** and **k** (where **k=i-j**) during the iterations

- clearly $i \leq n$ and since j is never negative we also have $k \leq n$
- in each iteration either i or k is incremented and neither is decremented
  - $i = i$ and $i-b[j] > i-j$
    - since $b[j]<j$ as $b[j]$ longest border in a string of length j

KMP search – Analysis

```
while (i<n)
   if (t[i] == s[j]){
       i++; j++;
   }
   else {
       if (b[j]>0) j = b[j];
       else {
           if (j=0) i++;
           else j = 0;
       }
    }
```

For the complexity we need to know the number of loop iterations

Consider values of **i** and **k** (where **k=i-j**) during the iterations

- clearly $i \leq n$ and since j is never negative we also have $k \leq n$
- in each iteration either i or k is incremented and neither is decremented
  - $i++ > i$ and $(i++)-j > i-j$

KMP search – Analysis

```
while (i<n)
   if (t[i] == s[j]){
       i++; j++;
   }
   else {
       if (b[j]>0) j = b[j];
       else {
           if (j=0) i++;
           else j = 0;
       }
    }
```

For the complexity we need to know the number of loop iterations

Consider values of **i** and **k** (where **k=i-j**) during the iterations

- clearly $i \leq n$ and since j is never negative we also have $k \leq n$
- in each iteration either i or k is incremented and neither is decremented
  - $i = i$ and $i-0 > i-j$
    - since $j>0$ must hold for the else case to be taken

KMP search – Analysis

```
while (i<n)
   if (t[i] == s[j]){
       i++; j++;
   }
   else {
       if (b[j]>0) j = b[j];
       else {
           if (j=0) i++;
           else j = 0;
       }
    }
```

For the complexity we need to know the number of loop iterations

Consider values of **i** and **k** (where **k=i-j**) during the iterations

- clearly $i \leq n$ and since j is never negative we also have $k \leq n$
- in each iteration either i or k is incremented and neither is decremented
- so the number of iterations of the loop is at most $2n$

Hence KMP is **O(n)** in the worst case for the search part

KMP search – Analysis

KMP search is $O(n)$ in the worst case

Creating the border table

- naïve method requires $O(j^2)$ steps to evaluate `b[j]` giving $O(m^3)$ overall
  - requires testing all possible borders
  - $1 + 2 + \ldots + j-1 = (j-1)j/2 = O(j^2)$ operations

KMP search – Analysis

KMP search is $O(n)$ in the worst case

Creating the border table

- naïve method requires $O(j^2)$ steps to evaluate `b[j]` giving $O(m^3)$ overall
- a more efficient method is possible that requires just $O(m)$ steps in total involves a subtle application of the KMP algorithm (details are omitted)

Overall complexity of KMP search

- KMP can be implemented to run in $O(m+n)$ time
- $O(m)$ for setting up the border table
- $O(n)$ for conducting the search

Have addressed challenge 1

- KMP algorithm is linear (i.e. $O(m+n)$)

Section 2 – Strings and text algorithms

Text compression

- Huffman encoding
- LZW compression/decompression

String comparison

- string difference

String/pattern search

- brute force algorithm
- KMP algorithm
- Boyer-Moore algorithm

Boyer-Moore Algorithm

Challenge 1: can we find a solution that is linear in the worst case?

Yes: KMP

Challenge 2: can we find a solution that is (much) faster than brute force on average?

Boyer-Moore: almost always faster than brute force or KMP

- variants are used in many real-world applications (used in GNU's grep)
- typically, many text characters are skipped without even being checked
- the string/pattern is scanned right-to-left
- text character involved in a mismatch is used to decide next comparison

Boyer-Moore Algorithm – Example

Search for '**pill**' in '**the caterpillar**'

```
the caterpillar
pill
   ^
```

Search for string from right to left

- start by comparing m^{th} element of text with last character of string
  - m is the length of the string, i.e. equals 4
  - i.e. we line them up on their first characters

Boyer-Moore Algorithm – Example

Search for '**pill**' in '**the caterpillar**'

```
the caterpillar
pill
   ^
```

Search for string from right to left

- start by comparing 4^{th} element of text with last (4^{th}) character of string

Boyer-Moore Algorithm – Example

Search for '**pill**' in '**the caterpillar**'

```
the caterpillar
pill
   ^
```

Search for string from right to left

- start by comparing 4^{th} element of text with last (4^{th}) character of string
- 'l' and ' ' (space) do not match
- since there is no ' ' in the string, we can shift string along by its length i.e. by 4 places

Boyer-Moore Algorithm – Example

Search for '**pill**' in '**the caterpillar**'

```
the caterpillar
 pill
```

Search for string from right to left

- start by comparing 4^{th} element of text with last (4^{th}) character of string
- 'l' and ' ' (space) do not match
- since there is no ' ' in the string, we can shift string along by its length i.e. by 4 places

Boyer-Moore Algorithm – Example

Search for '**pill**' in '**the caterpillar**'

```
the caterpillar
  pill
```

Search for string from right to left

- start by comparing 4^{th} element of text with last (4^{th}) character of string
- 'l' and ' ' (space) do not match
- since there is no ' ' in the string, we can shift string along by its length i.e. by 4 places

Boyer–Moore Algorithm – Example

Search for '**pill**' in '**the caterpillar**'

```
the caterpillar
   pill
```

Search for string from right to left

- start by comparing 4^{th} element of text with last (4^{th}) character of string
- 'l' and ' ' (space) do not match
- since there is no ' ' in the string, we can shift string along by its length i.e. by 4 places

Boyer-Moore Algorithm – Example

Search for '**pill**' in '**the caterpillar**'

```
the caterpillar
    pill
```

Search for string from right to left

- start by comparing 4^{th} element of text with last (4^{th}) character of string
- 'l' and ' ' (space) do not match
- since there is no ' ' in the string, we can shift string along by its length i.e. by 4 places

Boyer-Moore Algorithm – Example

Search for '**pill**' in '**the caterpillar**'

```
the caterpillar
    pill
       ^
```

Search for string from right to left

- start by comparing 4th element of text with last (4th) character of string
- 'l' and ' ' (space) do not match
- since there is no ' ' in the string can shift string along by its length
  i.e. by 4 places
  and continue the search from last position in the string

Boyer-Moore Algorithm – Example

Search for '**pill**' in '**the caterpillar**'

```
the caterpillar
    pill
       ^
```

Search for string from right to left

- continue search from the last position in the string
- 'l' and 'e' do not match
- since there is no 'e' in the string, we can again shift string along by its length
  - i.e. by 4, and continue search from the last position in the string
  - i.e. we shift the string all the way past 'e' in the text

Boyer-Moore Algorithm – Example

Search for '`pill`' in '`the caterpillar`'

```
the caterpillar
       pill
          ^
```

Search for string from right to left

- continue search from the last position in the string

Boyer-Moore Algorithm – Example

Search for '**pill**' in '**the caterpillar**'

```
the caterpillar
       pill
          ^
```

Search for string from right to left

- continue search from the last position in the string
- 'l' matches so continue trying to match right to left

Boyer-Moore Algorithm – Example

Search for '`pill`' in '`the caterpillar`'

```
the caterpillar
       pill
         ^
```

Search for string from right to left

- continue search from the last position in the string
- '`i`' and '`l`' do not match
- however there is an '`i`' in the string so move string along so that the '`i`'s line up

Boyer-Moore Algorithm – Example

Search for '`pill`' in '`the caterpillar`'

```
the caterpillar
        pill
```

Search for string from right to left

- continue search from the last position in the string
- '`i`' and '`l`' do not match
- however there is an '`i`' in the string so move string along so that the '`i`'s line up

Boyer-Moore Algorithm – Example

Search for '`pill`' in '`the caterpillar`'

```
the caterpillar
        pill
           ^
```

Search for string from right to left

- continue search from the last position in the string
- '`i`' and '`l`' do not match
- however there is an '`i`' in the string so move string along so that the '`i`'s line up
  and continue search from the last position in the string

Boyer-Moore Algorithm – Example

Search for '`pill`' in '`the caterpillar`'

```
the caterpillar
        pill
           ^
```

Search for string from right to left

- continue search from the last position in the string
- '`l`' matches so continue trying to match right to left

Boyer-Moore Algorithm – Example

Search for '**pill**' in '**the caterpillar**'

```
the caterpillar
        pill
          ^
```

Search for string from right to left

- continue search from the last position in the string
- 'l' matches so continue trying to match right to left

Boyer-Moore Algorithm – Example

Search for '**pill**' in '**the caterpillar**'

```
the caterpillar
        pill
         ^
```

Search for string from right to left

- continue search from the last position in the string
- '**i**' matches so continue trying to match right to left

Boyer-Moore Algorithm – Example

Search for '**pill**' in '**the caterpillar**'

```
the caterpillar
        pill
        ^
```

Search for string from right to left

- continue search from the last position in the string
- 'p' matches and we have found the string in the text

Boyer-Moore Algorithm - Simplified version

The string is scanned right-to-left

- text character involved in a mismatch is used to decide next comparison
- involves pre-processing the string to record the position of the last occurrence of each character `c` in the alphabet
- therefore the alphabet must be fixed in advance of the search

Last occurrence position of character **c** in the string **s**

- equals `max{ k | s[k]=c }` if such a `k` exists and `-1` otherwise

Want to store last occurrence position of **c** in an array element **p[c]**

- but in Java we can not index an array by characters
- instead can use the static method `Character.getNumericValue(c)`
- to compute an appropriate array index

Simplified version (often called the Boyer-Moore-Horspool algorithm)

Boyer–Moore Algorithm – Simplified version

In our pseudocode we assume an array `p[c]` indexed by characters

- the characters range over the underlying alphabet of the text
- `p[c]` records the position in the string of the last occurrence of char `c`
- if the character `c` is absent from the string `s`, then let `p[c]=-1`

Assume ASCII character set (128 characters)

- for Unicode (more than `107,000` characters), `p` would be a large array

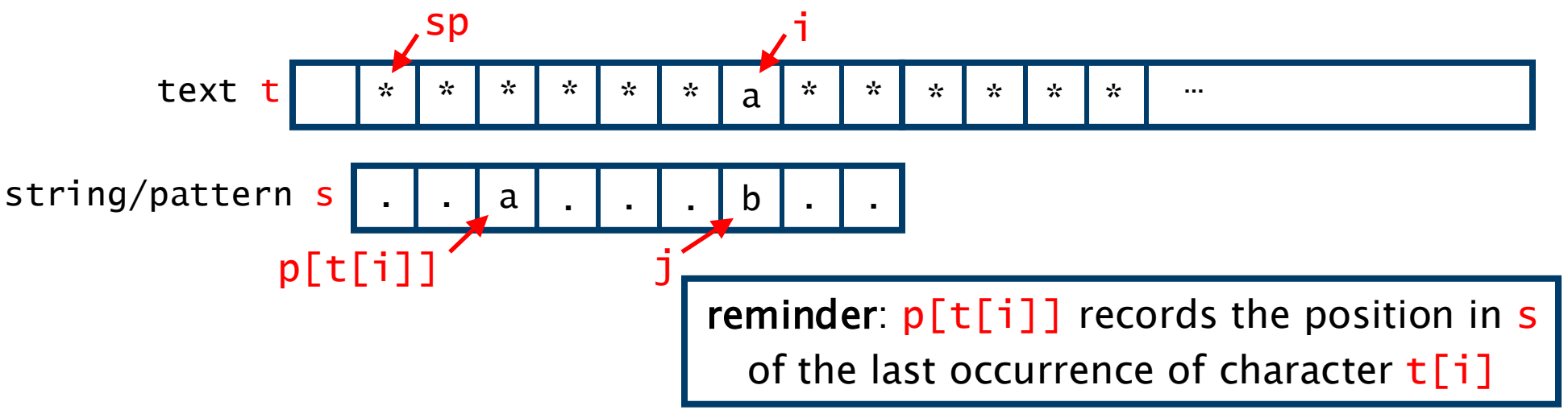
On finding a mismatch there is a jump step in the algorithm

- if the mismatch is between `s[j]` and `t[i]`
- 'slide `s` along' so that position `p[t[i]]` of `s` aligns with `t[i]`
  - i.e. align last position in `s` of character `t[i]` with position `i` of `t`
- if this moves `s` in the 'wrong direction', instead move `s` one position right
- if `t[i]` does not appear in string, 'slide string' passed `t[i]`
  - i.e. align position `-1` of `s` with position `i` of `t`

Boyer-Moore Algorithm – Jump step case 1

Assume a mismatch between position `s[j]` and position `t[i]`

Case 1: the last position of character `t[i]` in `s` is before position `j`



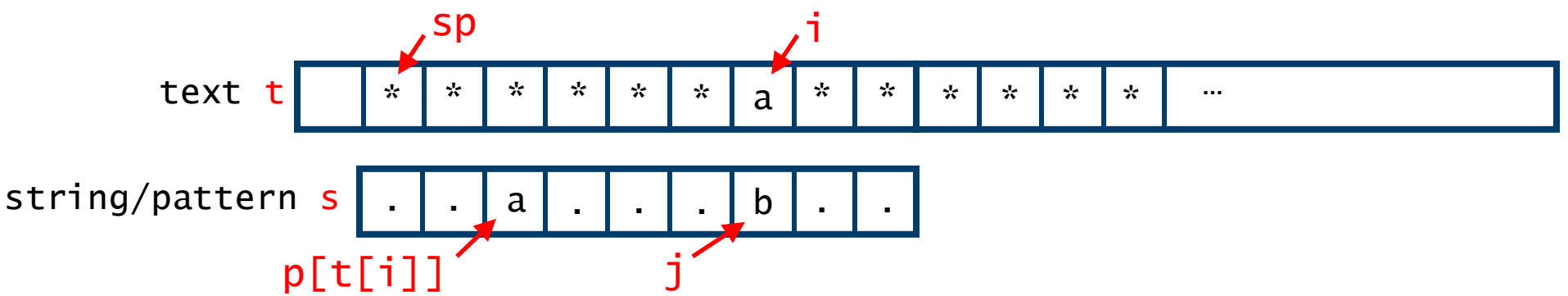
reminder: `p[t[i]]` records the position in `s` of the last occurrence of character `t[i]`

- `i` records the current position in the text we are checking
- `j` records the current position in the string we are checking
- `sp` records the current starting position of string in the text

Boyer-Moore Algorithm – Jump step case 1

Assume a mismatch between position `s[j]` and position `t[i]`

Case 1: the last position of character `t[i]` in `s` is before position `j`



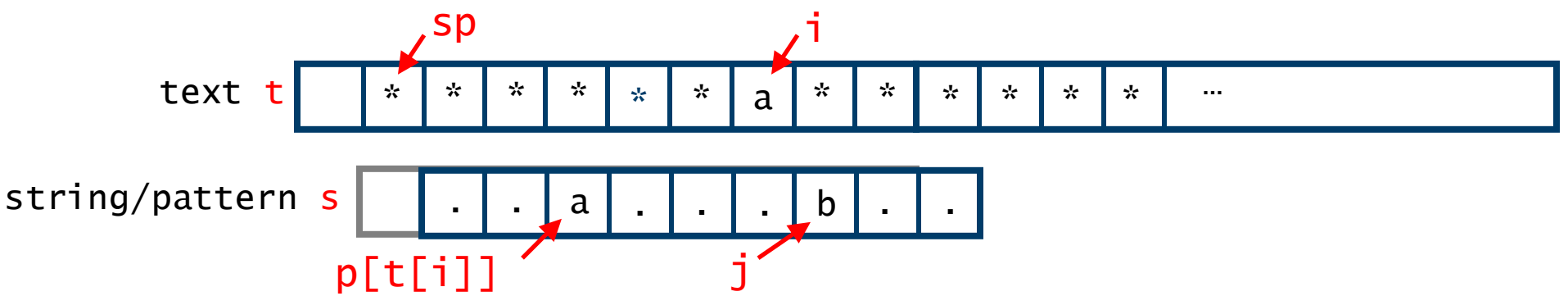
move `s` so char at position `p[t[i]]` is aligned with that of `t` that was mismatched

- `i` records the current position in the text we are checking
- `j` records the current position in the string we are checking
- `sp` records the current starting position of string in the text

Boyer-Moore Algorithm – Jump step case 1

Assume a mismatch between position `s[j]` and position `t[i]`

Case 1: the last position of character `t[i]` in `s` is before position `j`



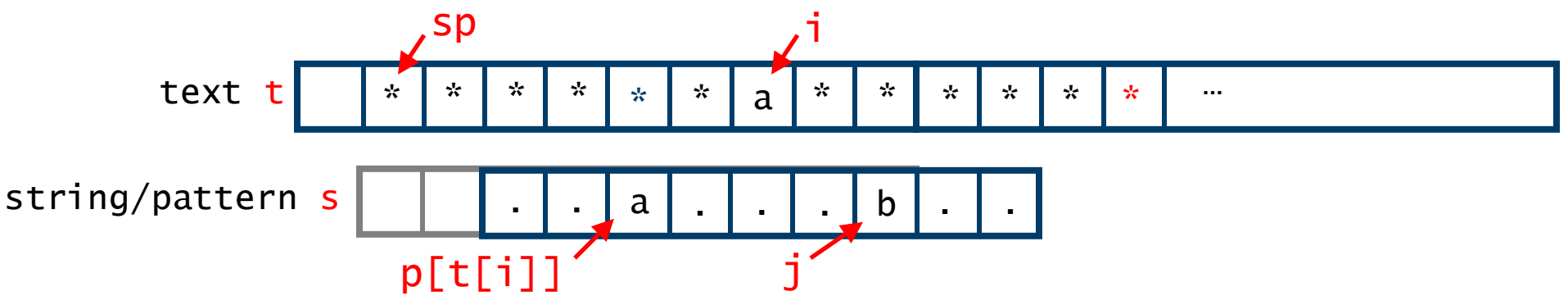
move `s` so char at position `p[t[i]]` is aligned with that of `t` that was mismatched

- `i` records the current position in the text we are checking
- `j` records the current position in the string we are checking
- `sp` records the current starting position of string in the text

Boyer-Moore Algorithm – Jump step case 1

Assume a mismatch between position **s[j]** and position **t[i]**

Case 1: the last position of character **t[i]** in **s** is before position **j**



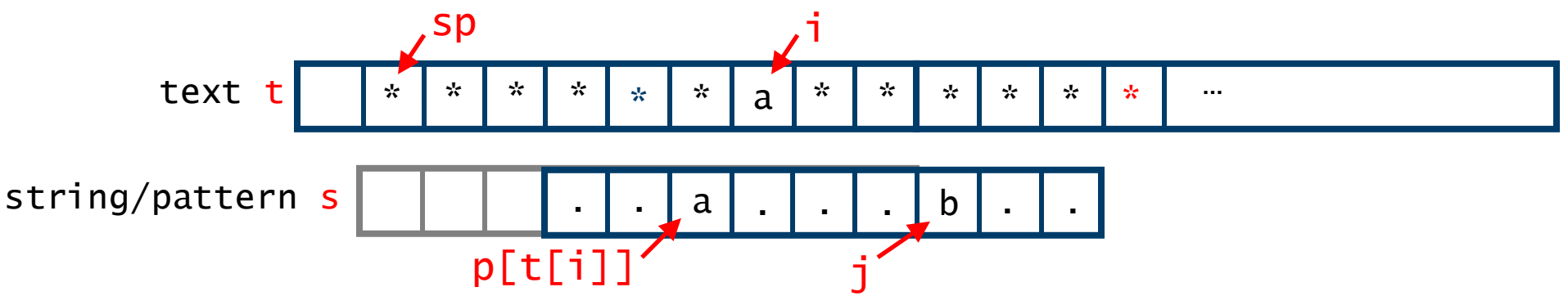
move **s** so char at position **p[t[i]]** is aligned with that of **t** that was mismatched

- **i** records the current position in the text we are checking
- **j** records the current position in the string we are checking
- **sp** records the current starting position of string in the text

Boyer-Moore Algorithm – Jump step case 1

Assume a mismatch between position **s[j]** and position **t[i]**

Case 1: the last position of character **t[i]** in **s** is before position **j**



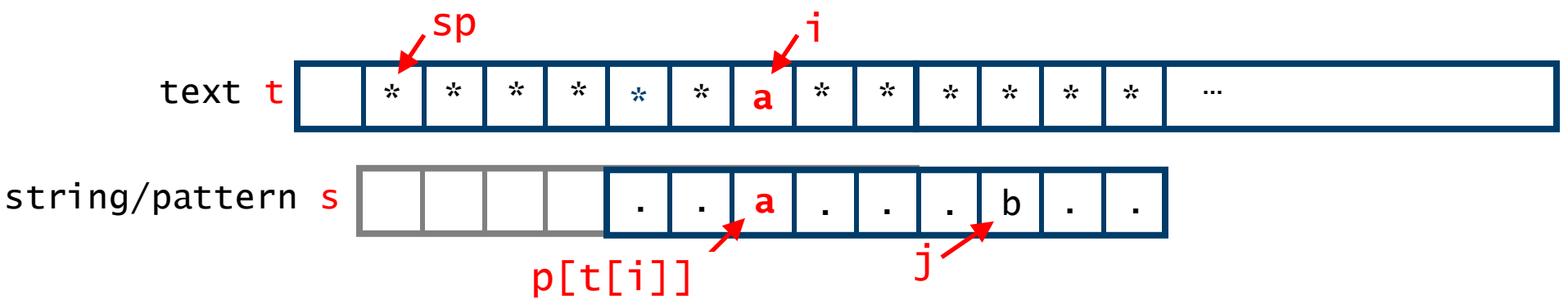
move **s** so char at position **p[t[i]]** is aligned with that of **t** that was mismatched

- **i** records the current position in the text we are checking
- **j** records the current position in the string we are checking
- **sp** records the current starting position of string in the text

Boyer-Moore Algorithm – Jump step case 1

Assume a mismatch between position **s[j]** and position **t[i]**

Case 1: the last position of character **t[i]** in **s** is before position **j**



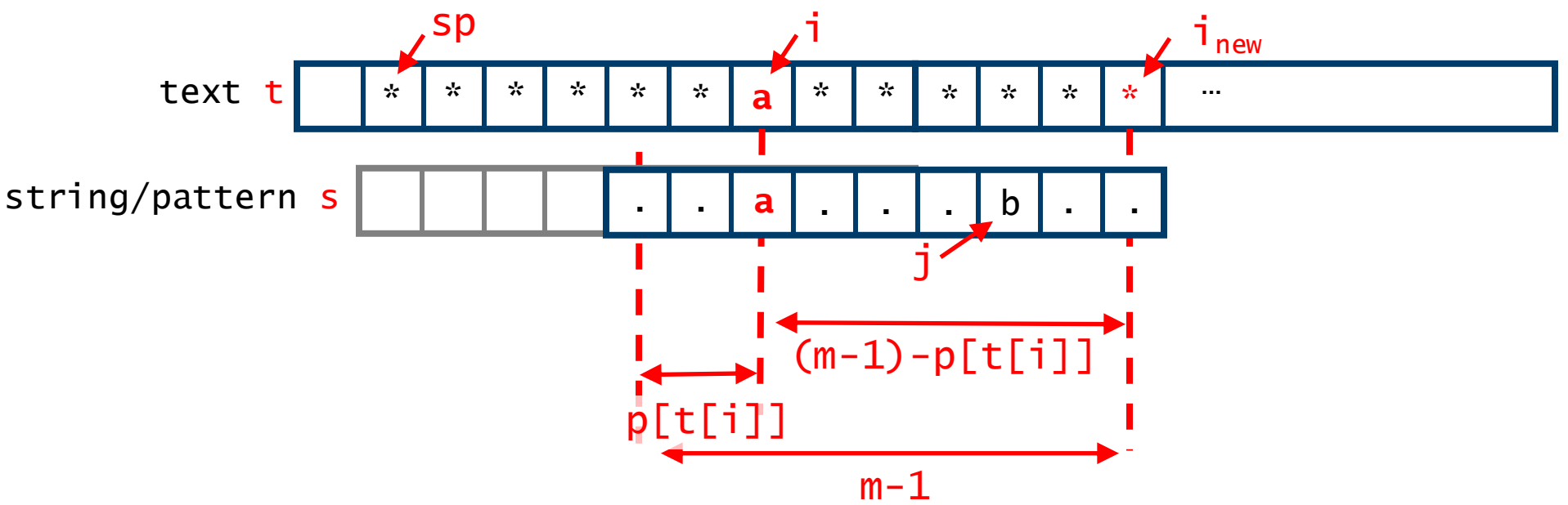
move s so char at position p[t[i]] is aligned with that of t that was mismatched

- i records the current position in the text we are checking
- j records the current position in the string we are checking
- sp records the current starting position of string in the text

Boyer–Moore Algorithm – Jump step case 1

Assume a mismatch between position **s[j]** and position **t[i]**

Case 1: the last position of character **t[i]** in **s** is before position **j**

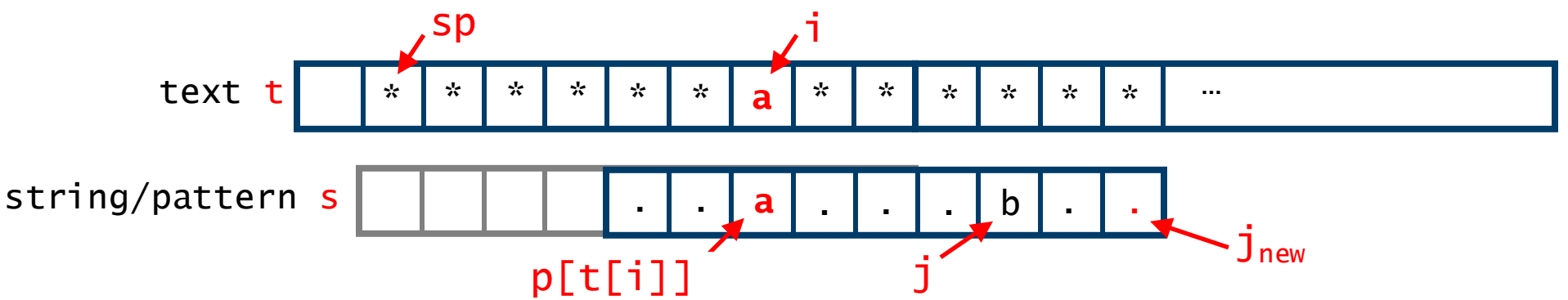


- i records the current position in the text we are checking
- new value of i equals i+(m-1)-p[t[i]]

Boyer–Moore Algorithm – Jump step case 1

Assume a mismatch between position **s[j]** and position **t[i]**

Case 1: the last position of character **t[i]** in **s** is before position **j**

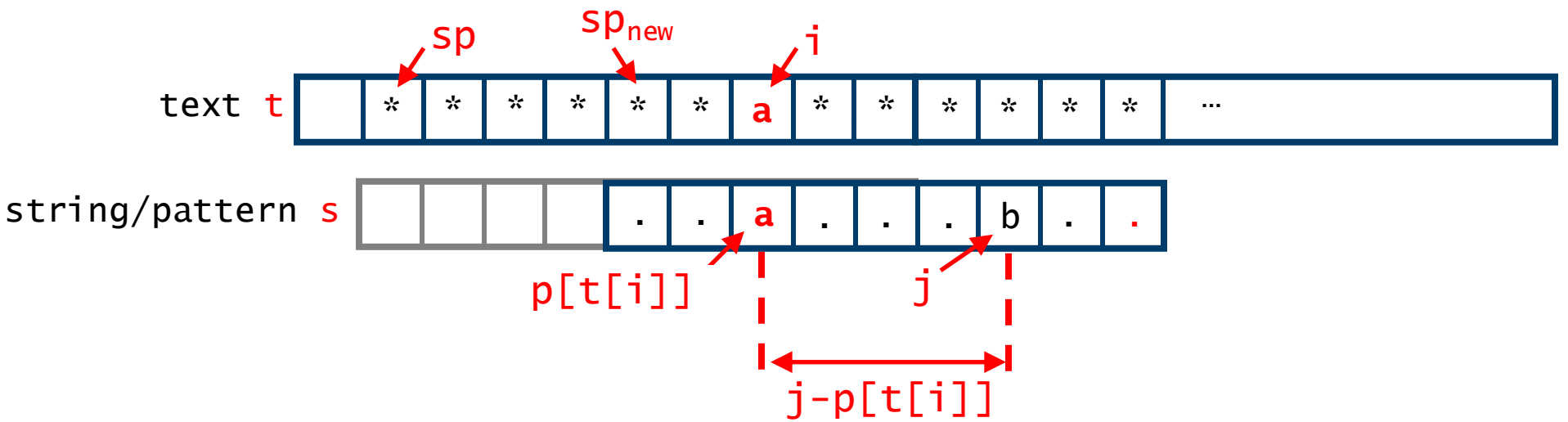


- j records the current position in the string we are checking
- new value of j equals m-1 (start again from the end of the string/pattern)

Boyer–Moore Algorithm – Jump step case 1

Assume a mismatch between position **s[j]** and position **t[i]**

Case 1: the last position of character **t[i]** in **s** is before position **j**

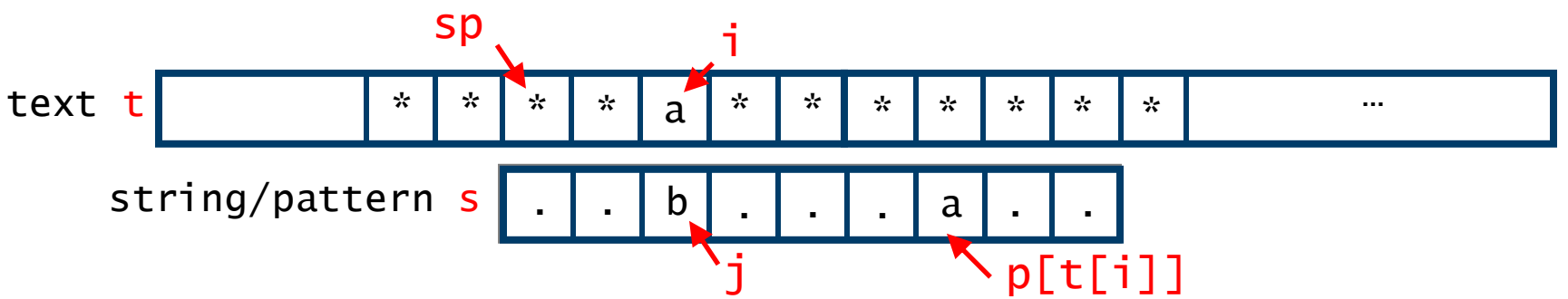


- sp records the current starting position of string in the text
- new value of sp equals sp+j-p[t[i]] as this is the amount the pattern/ string has been moved forward

Boyer–Moore Algorithm – Jump step case 2

Assume a mismatch between position **s[j]** and position **t[i]**

Case 2: last position of character **t[i]** in **s** is at least at position **j**



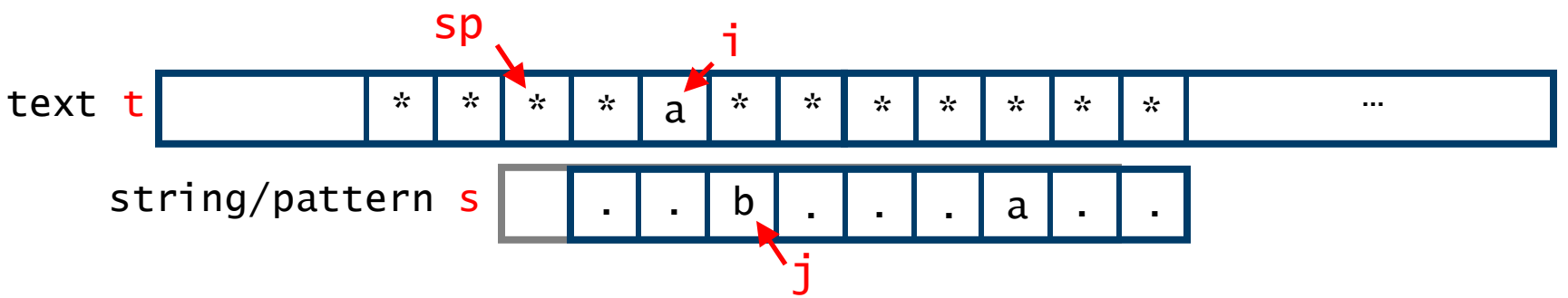
move string along by one place and start again from the end of the string

- i records the current position in the text we are checking
- j records the current position in the string we are checking
- sp records the current starting position of string in the text

Boyer–Moore Algorithm – Jump step case 2

Assume a mismatch between position `s[j]` and position `t[i]`

Case 2: last position of character `t[i]` in `s` is at least at position `j`



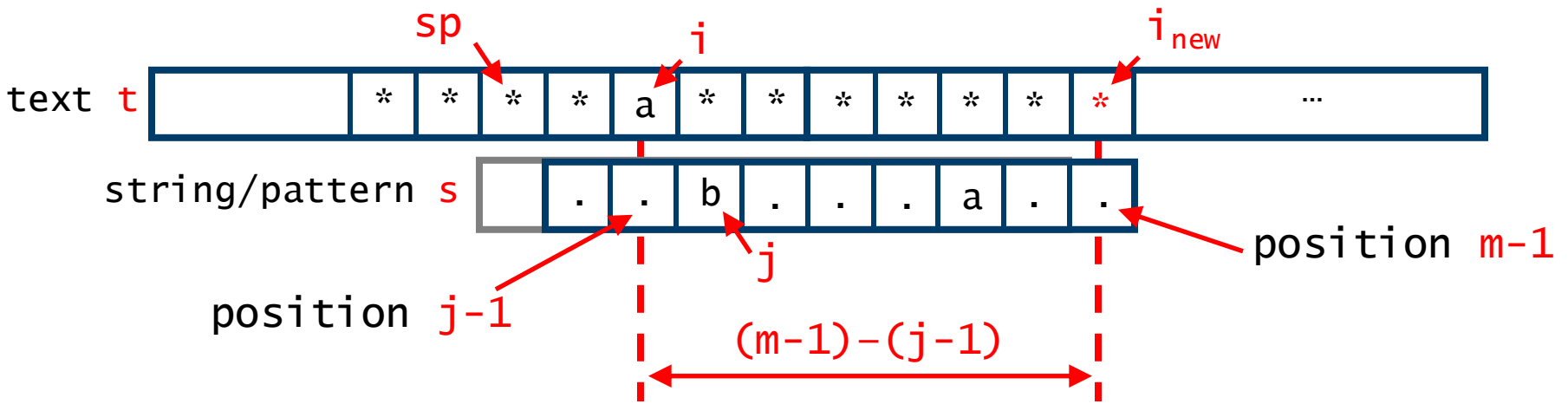
move string along by one place and start again from the end of the string

- `i` records the current position in the text we are checking
- `j` records the current position in the string we are checking
- `sp` records the current starting position of string in the text

Boyer–Moore Algorithm – Jump step case 2

Assume a mismatch between position `s[j]` and position `t[i]`

Case 2: last position of character `t[i]` in `s` is at least at position `j`

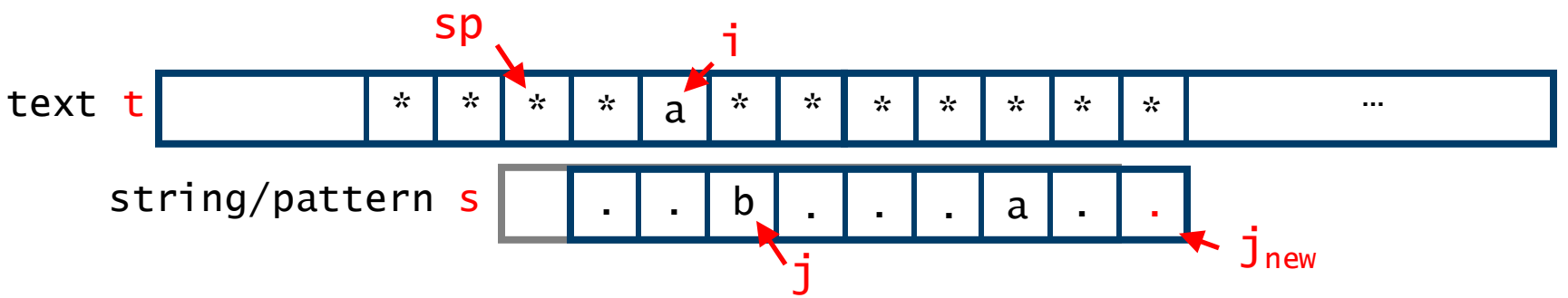


- `i` records the current position in the text we are checking
- new value of `i` equals `i+(m-1)-(j-1) = i+(m-j)`

Boyer–Moore Algorithm – Jump step case 2

Assume a mismatch between position `s[j]` and position `t[i]`

Case 2: last position of character `t[i]` in `s` is at least at position `j`

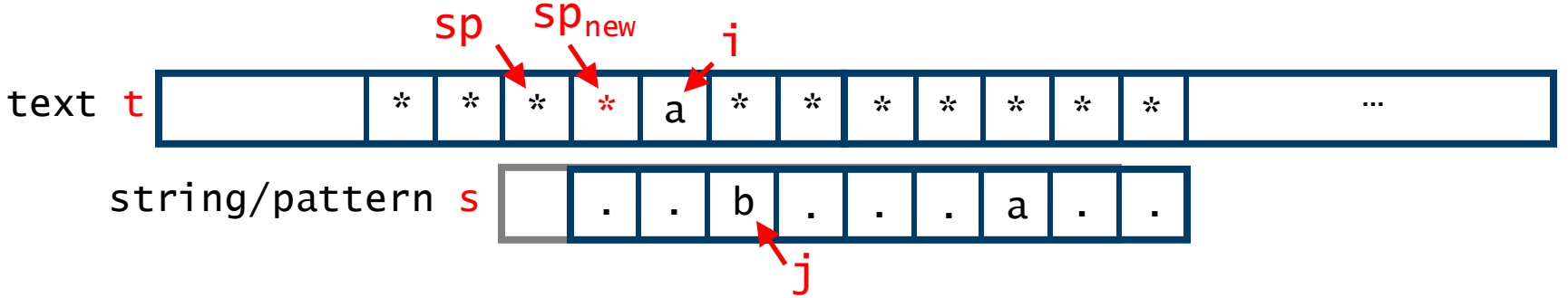


- `j` records the current position in the string we are checking
- new value of `j` equals `m-1`

Boyer–Moore Algorithm – Jump step case 2

Assume a mismatch between position **s[j]** and position **t[i]**

Case 2: last position of character **t[i]** in **s** is at least at position **j**

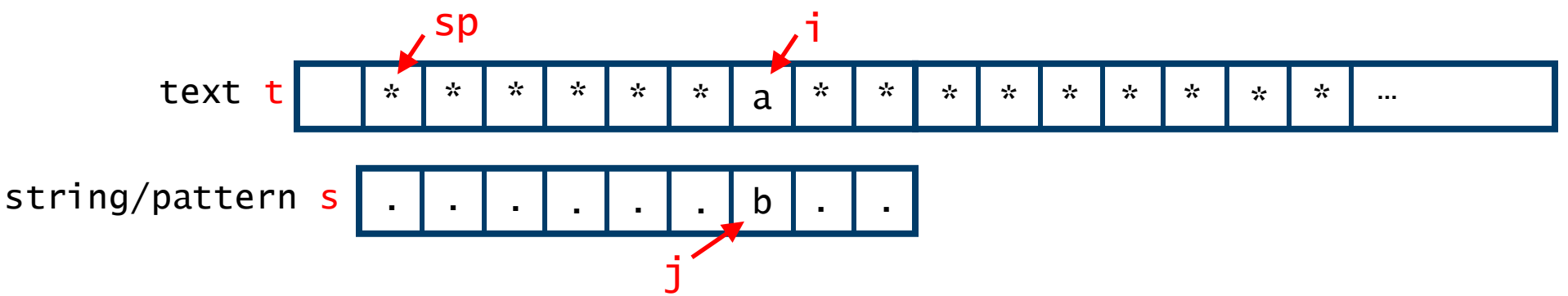


- sp records the current starting position of string in the text
- new value of sp equals sp+1

Boyer-Moore Algorithm – Jump step case 3

Assume a mismatch between position **s[j]** and position **t[i]**

Case 3: character **t[i]** does not appear in **s** (i.e. we have **p[j]=-1**)

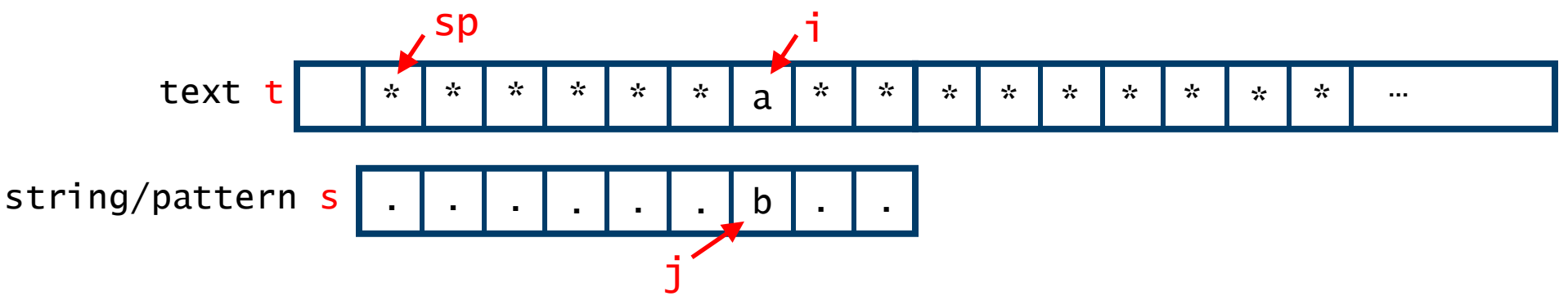


- i records the current position in the text we are checking
- j records the current position in the string we are checking
- sp records the current starting position of string in the text

Boyer-Moore Algorithm – Jump step case 3

Assume a mismatch between position **s[j]** and position **t[i]**

Case 3: character **t[i]** does not appear in **s** (i.e. we have **p[j]=-1**)



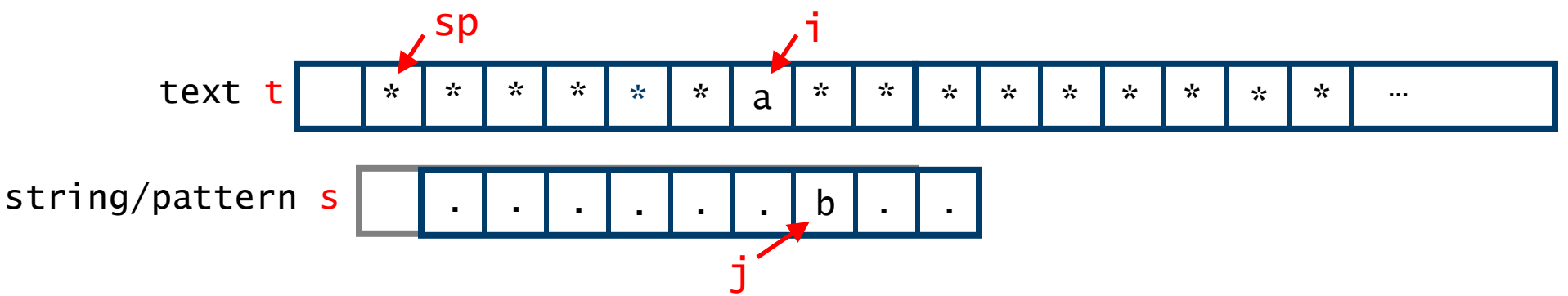
move s past the character that does not appear in the string

- i records the current position in the text we are checking
- j records the current position in the string we are checking
- sp records the current starting position of string in the text

Boyer-Moore Algorithm – Jump step case 3

Assume a mismatch between position `s[j]` and position `t[i]`

Case 3: character `t[i]` does not appear in `s` (i.e. we have `p[j]=-1`)



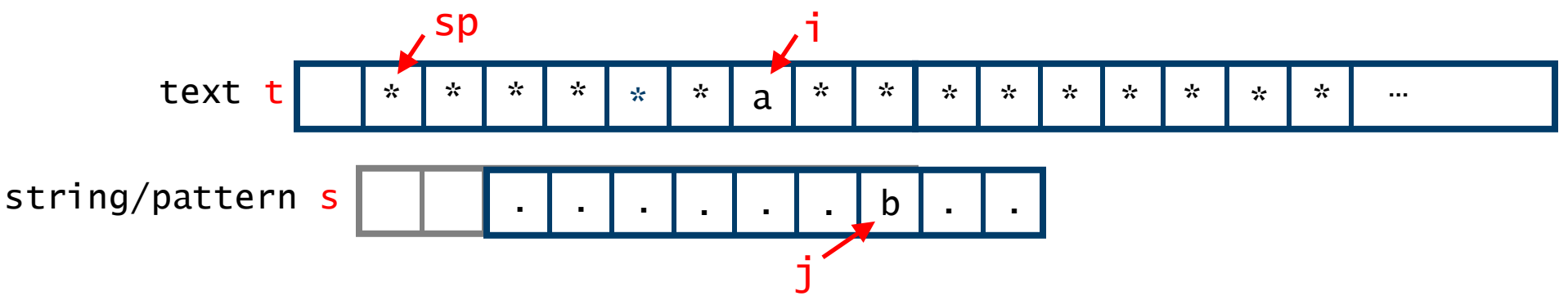
move `s` past the character that does not appear in the string

- `i` records the current position in the text we are checking
- `j` records the current position in the string we are checking
- `sp` records the current starting position of string in the text

Boyer-Moore Algorithm – Jump step case 3

Assume a mismatch between position `s[j]` and position `t[i]`

Case 3: character `t[i]` does not appear in `s` (i.e. we have `p[j]=-1`)



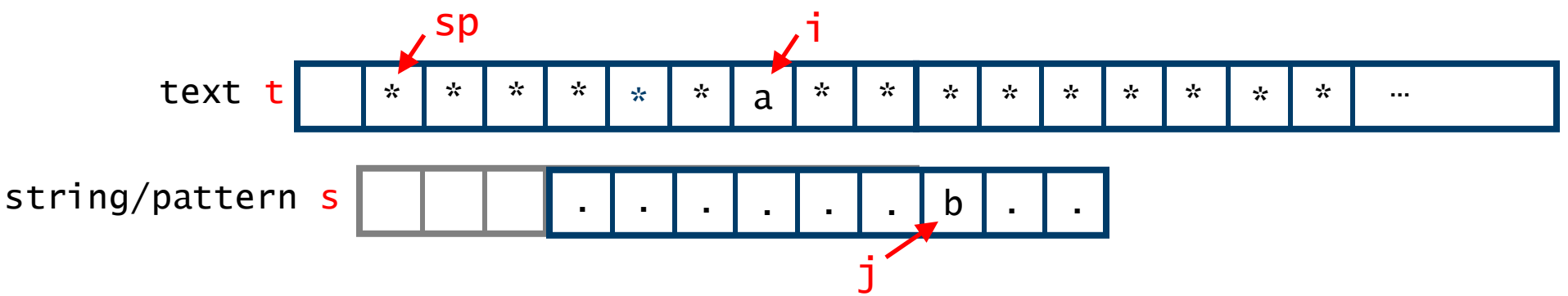
move `s` past the character that does not appear in the string

- `i` records the current position in the text we are checking
- `j` records the current position in the string we are checking
- `sp` records the current starting position of string in the text

Boyer-Moore Algorithm – Jump step case 3

Assume a mismatch between position **s[j]** and position **t[i]**

Case 3: character **t[i]** does not appear in **s** (i.e. we have **p[j]=-1**)



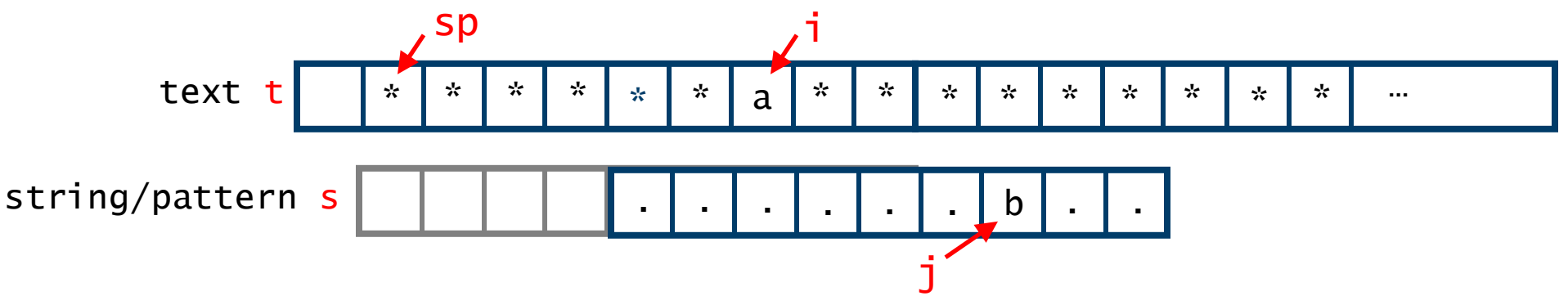
move s past the character that does not appear in the string

- i records the current position in the text we are checking
- j records the current position in the string we are checking
- sp records the current starting position of string in the text

Boyer-Moore Algorithm – Jump step case 3

Assume a mismatch between position **s[j]** and position **t[i]**

Case 3: character **t[i]** does not appear in **s** (i.e. we have **p[j]=-1**)



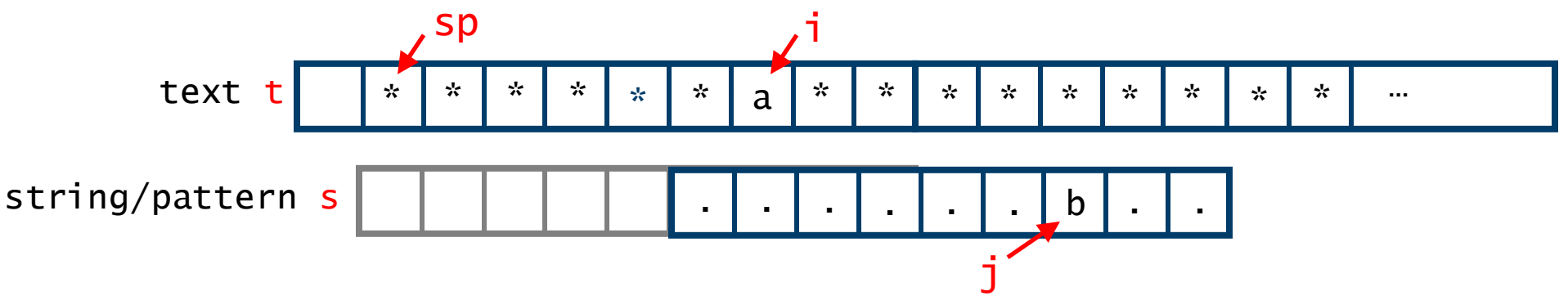
move s past the character that does not appear in the string

- i records the current position in the text we are checking
- j records the current position in the string we are checking
- sp records the current starting position of string in the text

Boyer-Moore Algorithm – Jump step case 3

Assume a mismatch between position **s[j]** and position **t[i]**

Case 3: character **t[i]** does not appear in **s** (i.e. we have **p[j]=-1**)



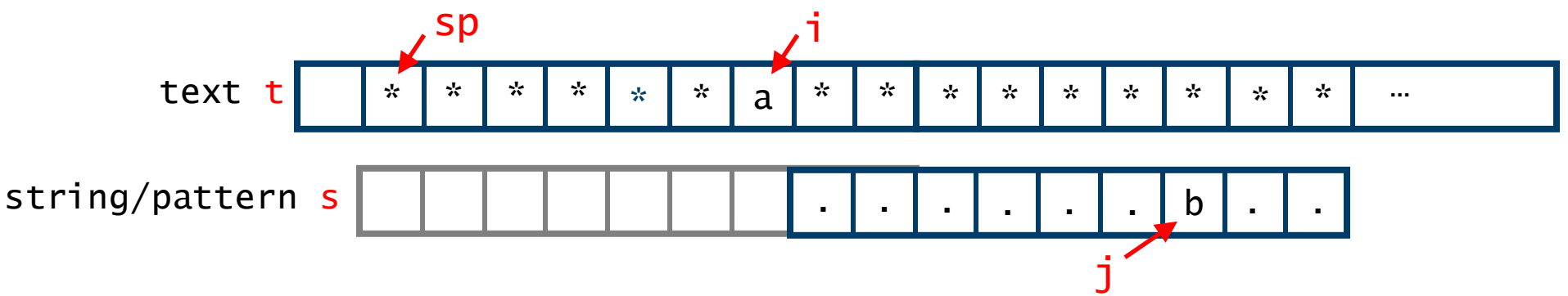
move s past the character that does not appear in the string

- i records the current position in the text we are checking
- j records the current position in the string we are checking
- sp records the current starting position of string in the text

Boyer-Moore Algorithm – Jump step case 3

Assume a mismatch between position **s[j]** and position **t[i]**

Case 3: character **t[i]** does not appear in **s** (i.e. we have **p[j]=-1**)



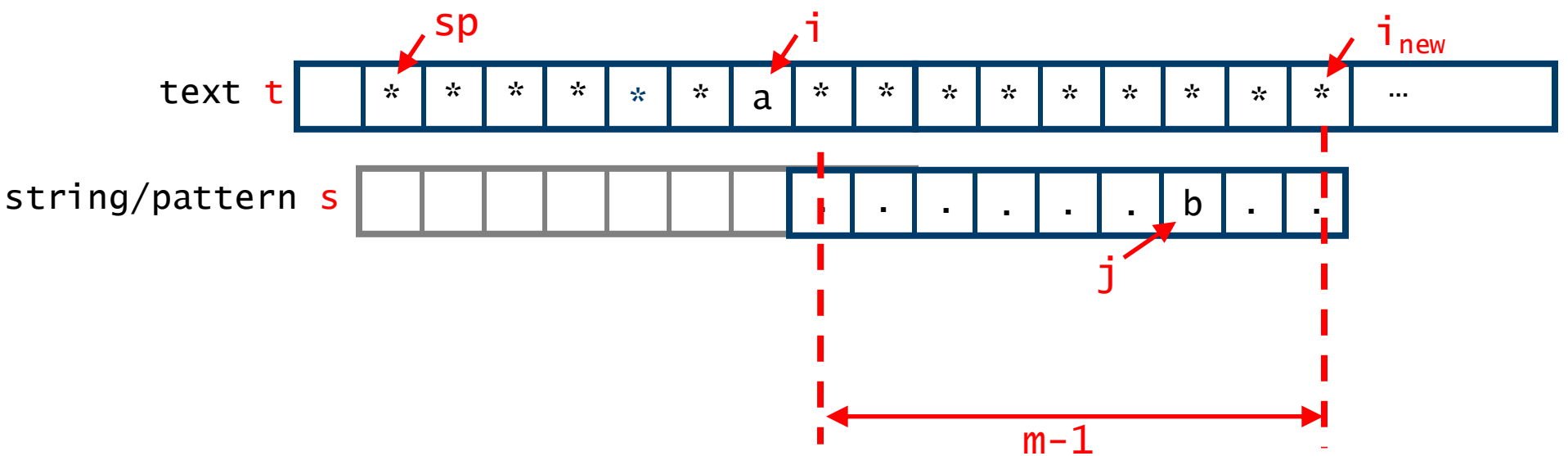
move s past the character that does not appear in the string

- i records the current position in the text we are checking
- j records the current position in the string we are checking
- sp records the current starting position of string in the text

Boyer-Moore Algorithm – Jump step case 3

Assume a mismatch between position **s[j]** and position **t[i]**

Case 3: character **t[i]** does not appear in **s** (i.e. we have **p[j]=-1**)

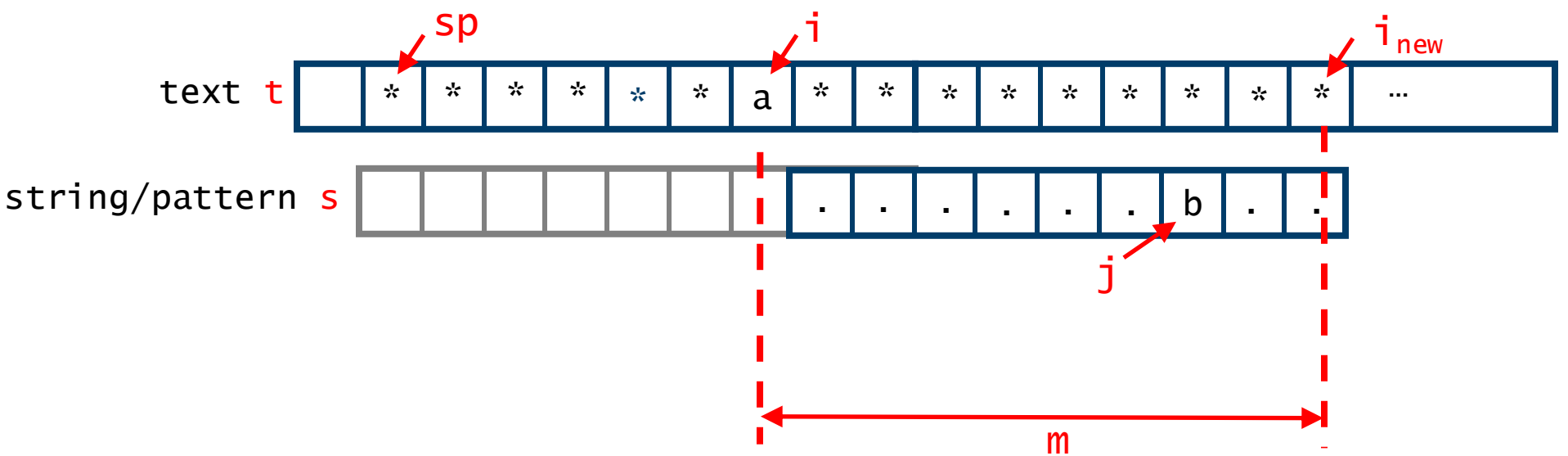


- i records the current position in the text we are checking
- new value of i equals i+m

Boyer-Moore Algorithm – Jump step case 3

Assume a mismatch between position **s[j]** and position **t[i]**

Case 3: character **t[i]** does not appear in **s** (i.e. we have **p[j]=-1**)

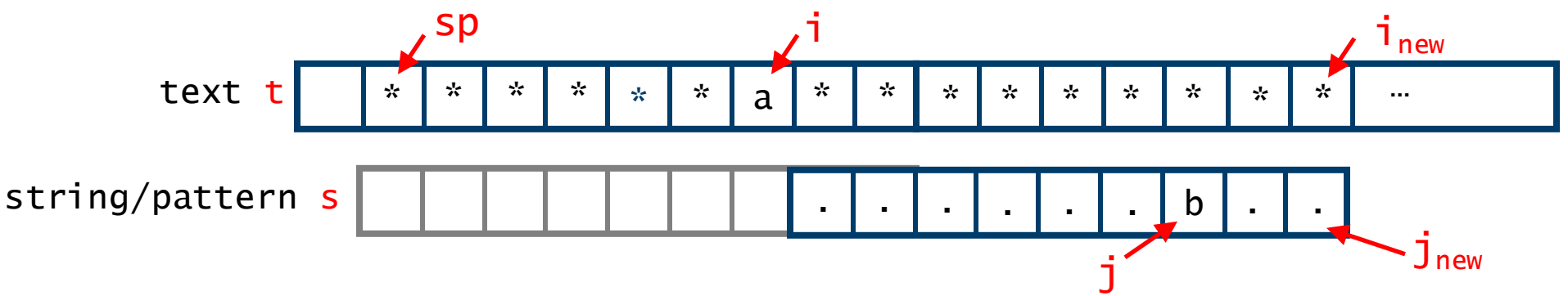


- i records the current position in the text we are checking
- new value of i equals i+m

Boyer-Moore Algorithm – Jump step case 3

Assume a mismatch between position `s[j]` and position `t[i]`

Case 3: character `t[i]` does not appear in `s` (i.e. we have `p[j]=-1`)

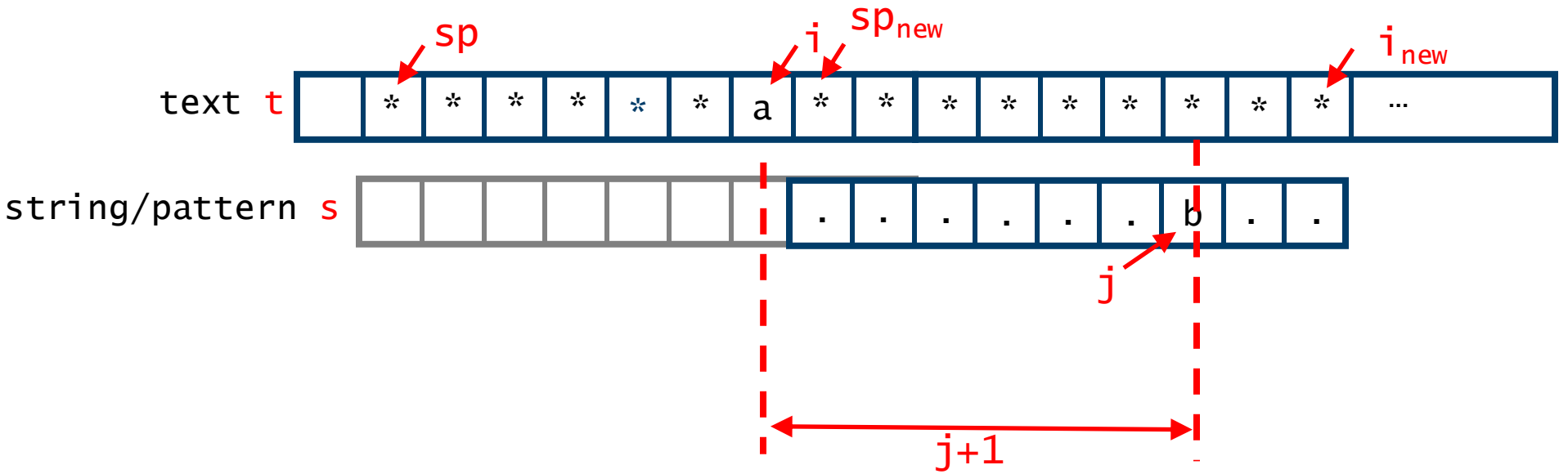


- `j` records the current position in the string we are checking
- new value of `j` equals `m-1` (start again from the end of the string/pattern)

Boyer-Moore Algorithm – Jump step case 3

Assume a mismatch between position **s[j]** and position **t[i]**

Case 3: character **t[i]** does not appear in **s** (i.e. we have **p[j]=-1**)



- sp records the current starting position of string in the text
- new value of sp equals sp+(j+1) as this is the amount the pattern/ string has been moved forward

Boyer–Moore Algorithm – All cases

Case 1: `p[t[i]]<j` and `p[t[i]]≥0`

- new value of `i` equals `i+m-1-p[t[i]]`
- new value of `j` equals `m-1`

Case 2: `p[t[i]]>j`

- new value of `i` equals `i+m-j`
- new value of `j` equals `m-1`
- new value of `sp` equals `sp+1`

Case 3: `p[t[i]]=-1`

- new value of `i` equals `i+m`
- new value of `j` equals `m-1`
- new value of `sp` equals `sp+j+1`

Note `p[t[i]]` cannot equal `j` as `p[t[i]]` last position of character `t[i]` in `s` and mismatch between `t[i]` and `s[j]`

Boyer-Moore Algorithm – All cases

We find that we can express these updates as follows:

- new value of `i` equals `i + m - min(1+p[t[i]],j)`
- new value of `j` equals `m-1`
- new value of `sp` equals `sp + max(j-p[t[i]],1)`

You do not need to learn these updates, just how the algorithm works

- this is sufficient for running it on an example (as you saw)
- and for working out what the updates are if needed (again as you saw)

Boyer-Moore Algorithm – Implementation

```
/** return smallest k such that s occurs at k in t or -1 if no k exists */
public int bm(char[] t, char[] s) {
   int m = s.length; // length of string/pattern
   int n = t.length; // length of text
   int sp = 0; // current starting position of string in text
   int i = m-1; // current position in text
   int j = m-1; // current position in string/pattern
   // declare a suitable array p
   setUp(s, p);   // set up the last occurrence array
   while (sp <= n-m && j >= 0) {
      if (t[i] == s[j]){ // current characters match
         i--; // move back in text
         j--; // move back in string
      } else { // current characters do not match
         sp += max(1, j - p[t[i]]);
         i += m - min(j, 1 + p[t[i]]);
         j = m-1;   // return to end of string
      }
   }
   if (j < 0) return sp; else return -1; // occurrence found yes/no
}
```

Boyer-Moore Algorithm – Complexity

Worst case is no better than O(mn)

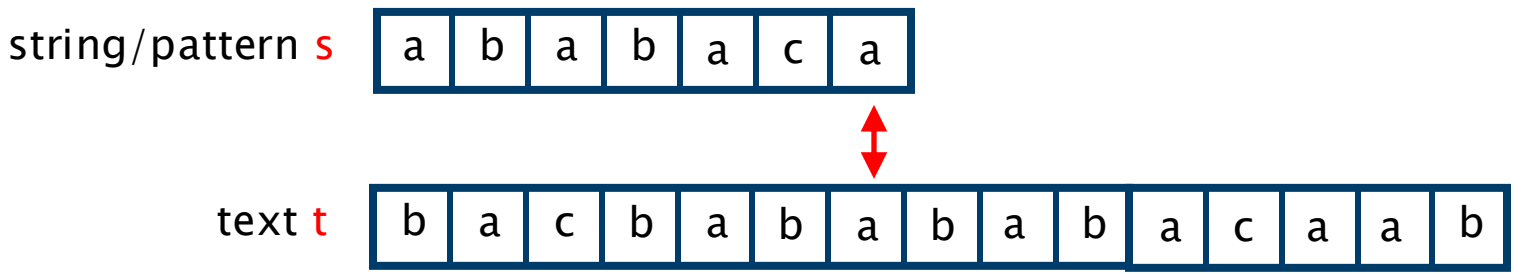
- e.g. search for s = ba … aa in t = aa … aaaa … aa

- m character comparisons needed at each n-(m+1) positions in the text before the text/pattern is found
  - start from the right and all matches until we get to b
  - since the last occurrence comes afterwards, only shift by one position each time
  - we end up doing the same number of operations at each step
- hence worst case behaviour similar to brute force

There is an extended version which is linear, i.e. O(m+n)

- this uses "the good suffix rule" – look it up yourselves

Boyer Moore – Example

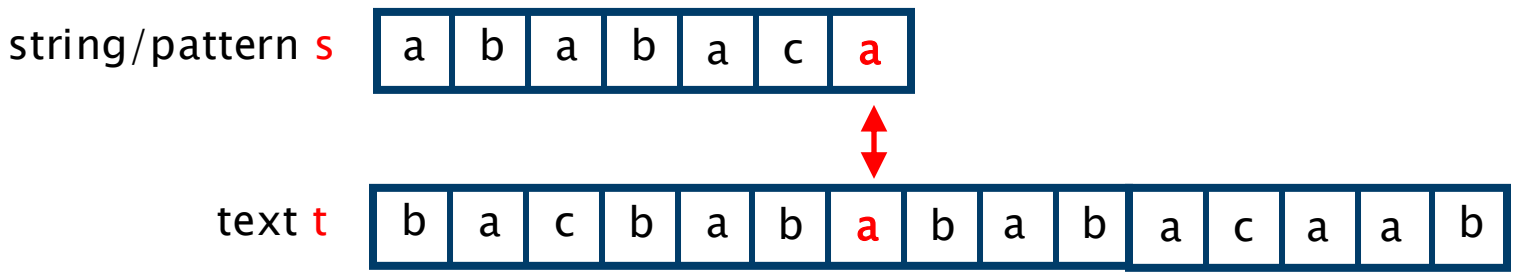


Starting position:

position in string j=6

- start of text and end of string

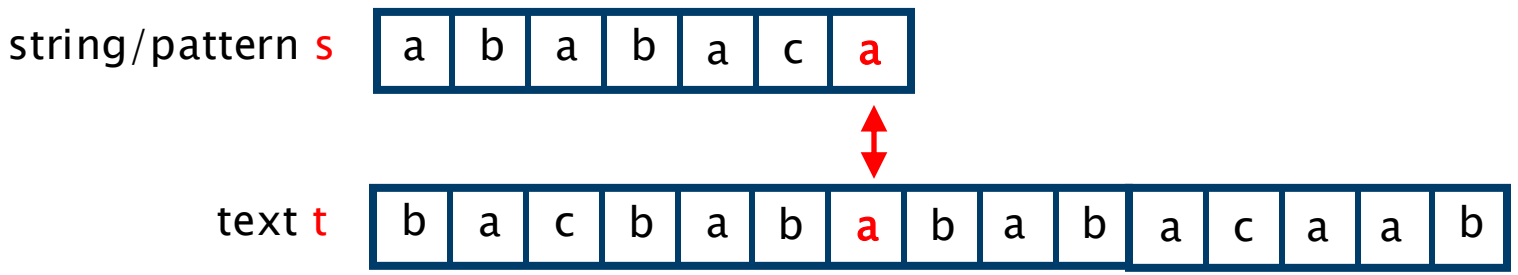
Boyer Moore – Example



Compare characters in text and string

position in string j=6

Boyer Moore – Example

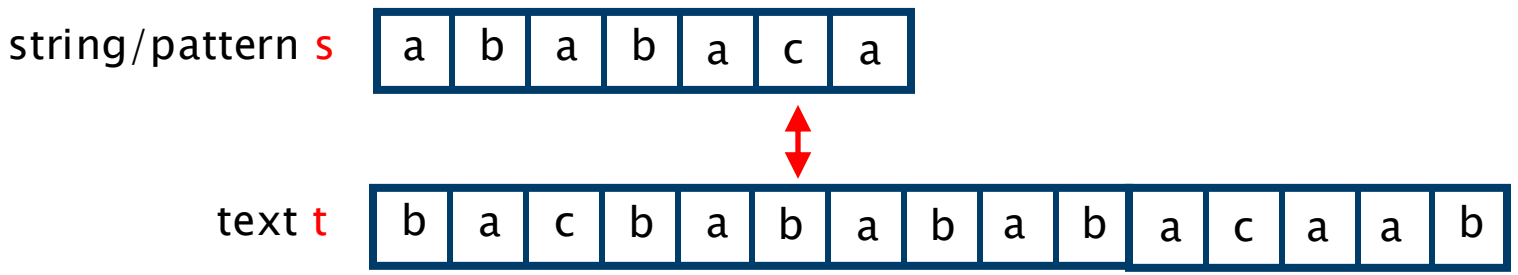


Characters match

position in string j=6

- decrement position in text and string

Boyer Moore – Example

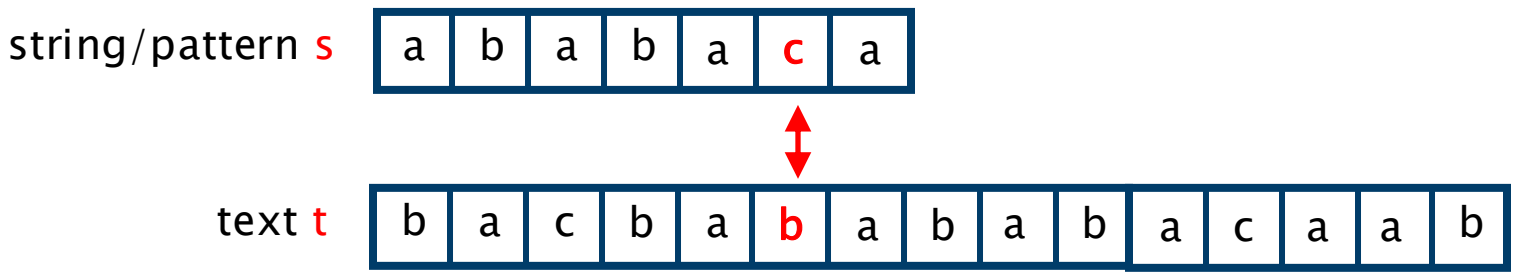


Characters match

position in string j=5

- decrement position in text and string

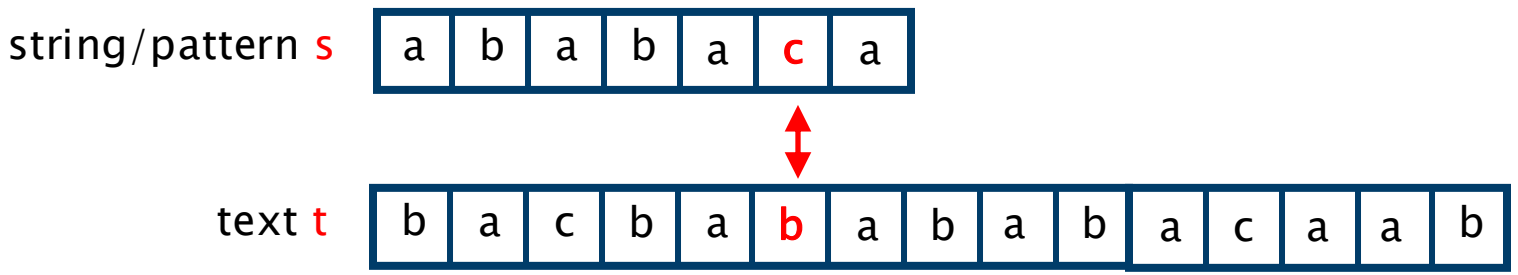
Boyer Moore – Example



Compare characters in text and string

position in string j=5

Boyer Moore – Example

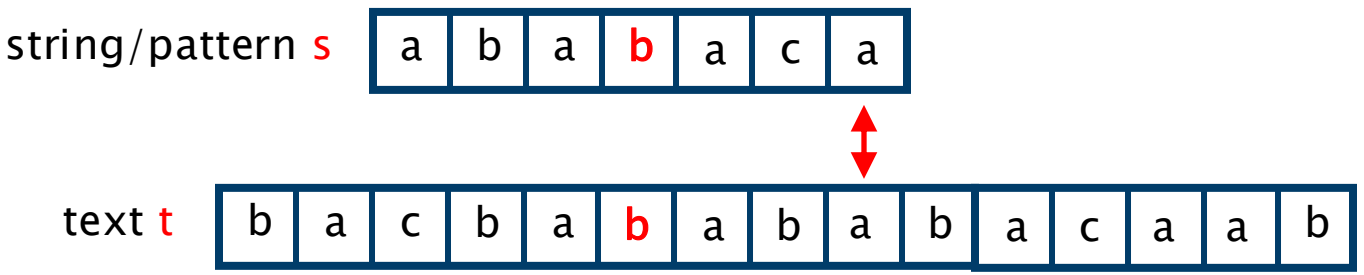


Characters do not match

position in string j=5

- case 1: the last position of character b in s is before position j
- move s so last b in s is aligned with that of t that was mismatched
- and start comparisons from end of the string

Boyer Moore – Example

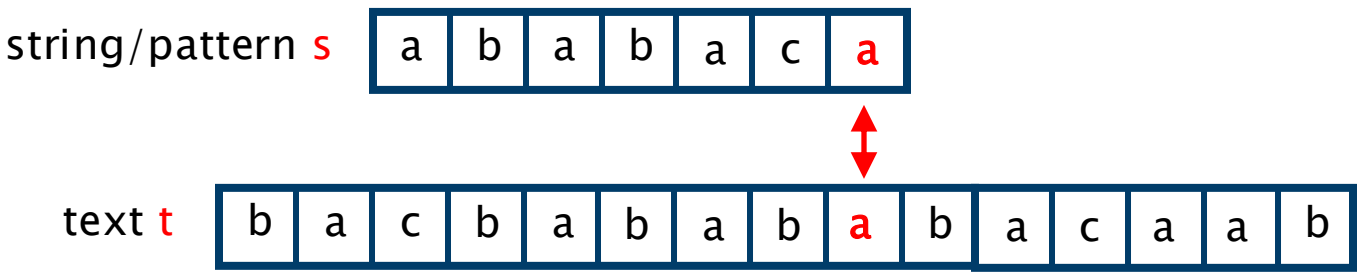


Characters do not match

position in string j=6

- case 1: the last position of character b in s is before position j
- move s so last b in s is aligned with that of t that was mismatched
- and start comparisons from end of the string

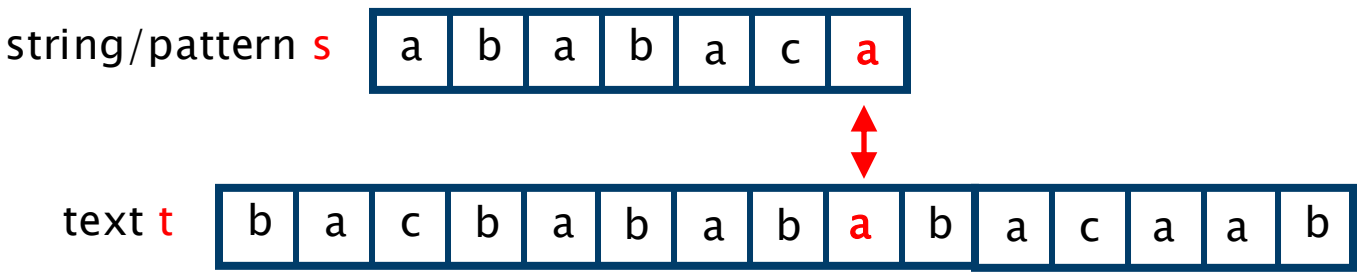
Boyer Moore – Example



Compare characters in text and string

position in string j=6

Boyer Moore – Example

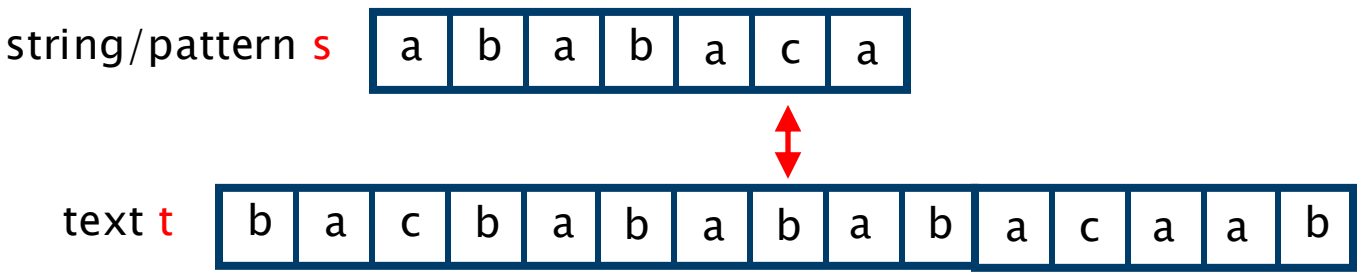


Characters match

position in string j=6

- decrement position in text and string

Boyer Moore – Example

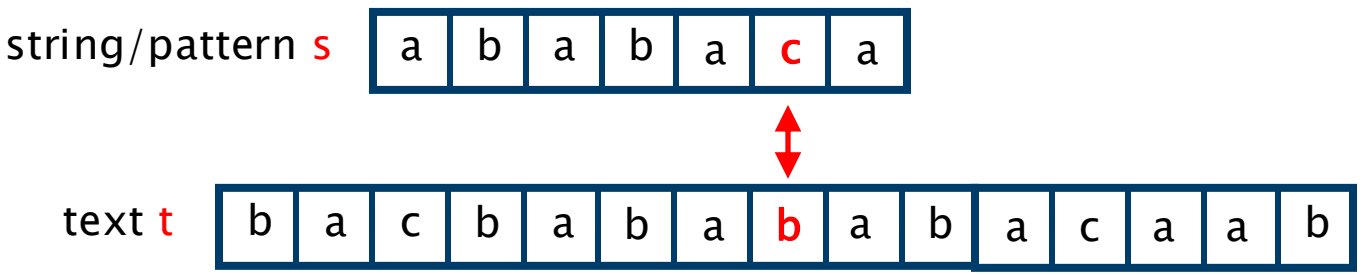


Characters match

position in string j=5

- decrement position in text and string

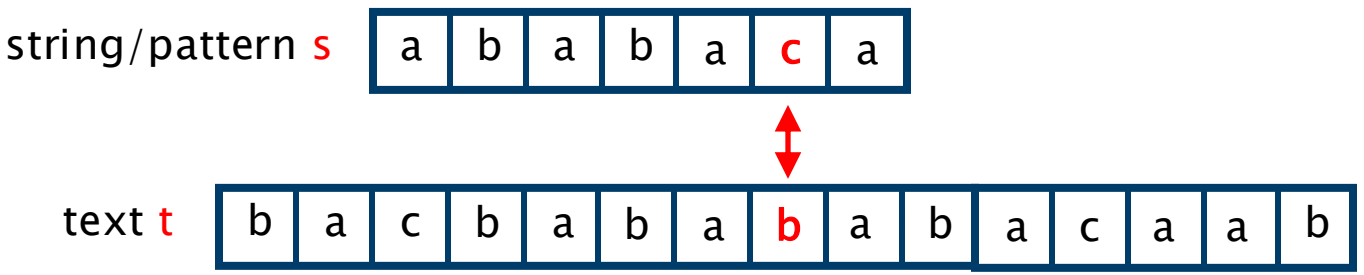
Boyer Moore – Example



Compare characters in text and string

position in string j=5

Boyer Moore – Example

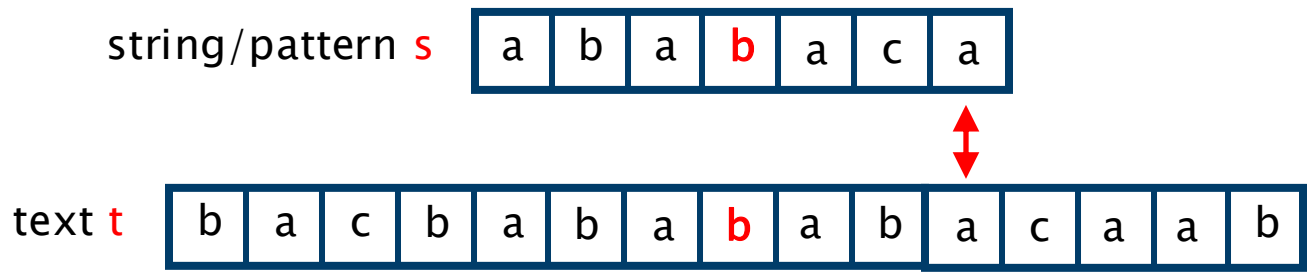


Characters do not match

position in string j=5

- case 1: the last position of character b in s is before position j
- move s so last b in s is aligned with that of t that was mismatched
- and start comparisons from end of the string

Boyer Moore – Example

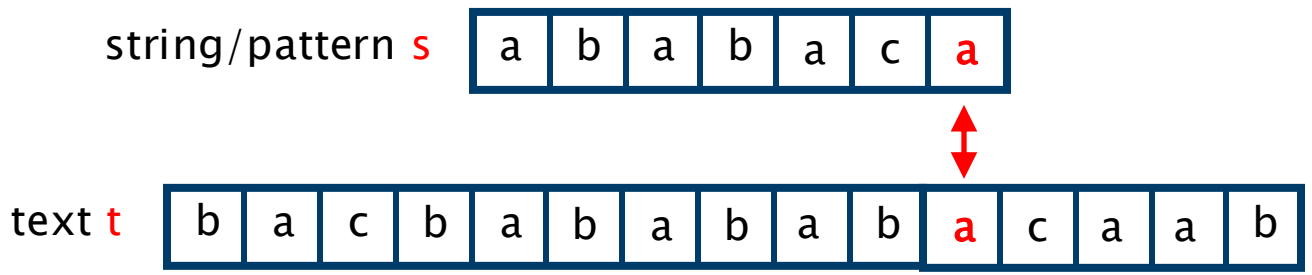


Characters do not match

position in string j=6

- case 1: the last position of character b in s is before position j
- move s so last b in s is aligned with that of t that was mismatched
- and start comparisons from end of the string

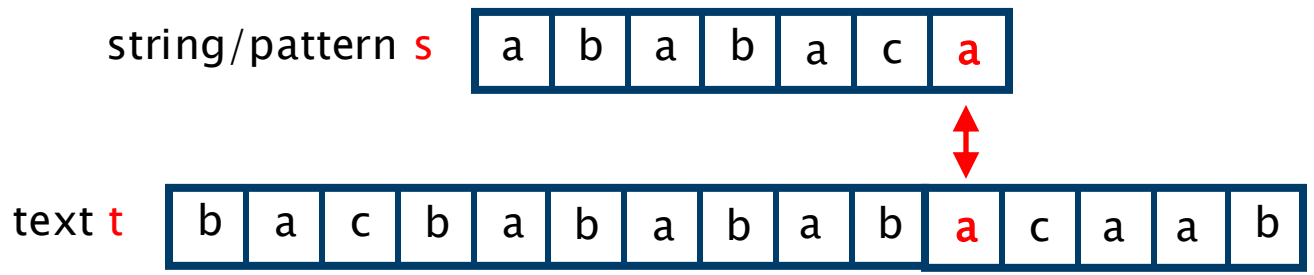
Boyer Moore – Example



Compare characters in text and string

position in string j=6

Boyer Moore – Example

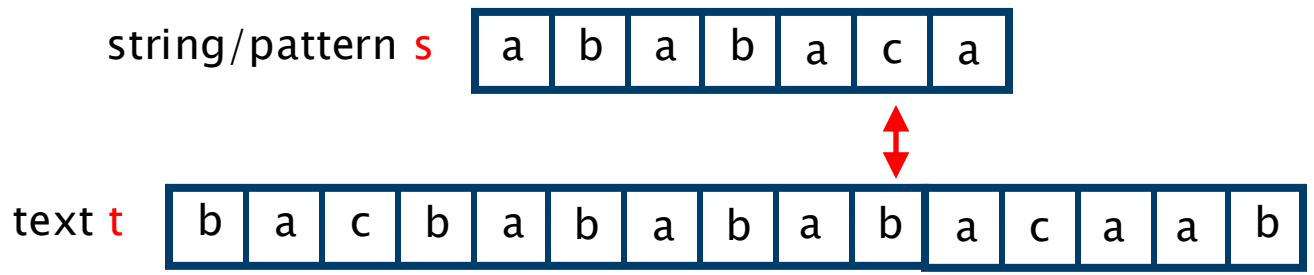


Characters match

position in string j=6

- decrement position in text and string

Boyer Moore – Example

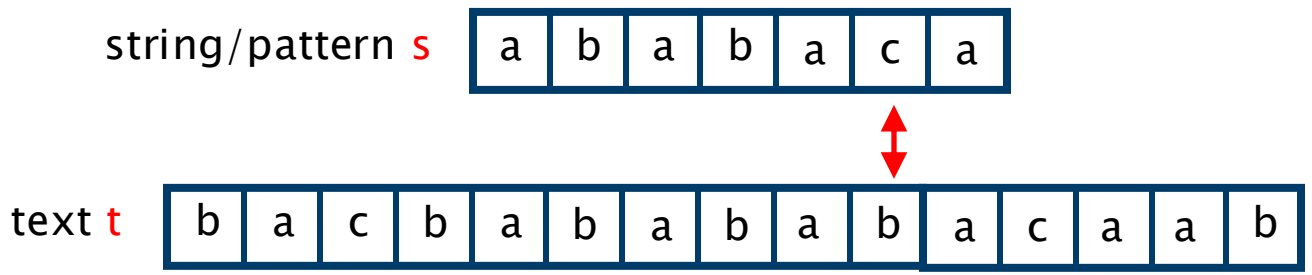


Characters match

position in string j=5

- decrement position in text and string

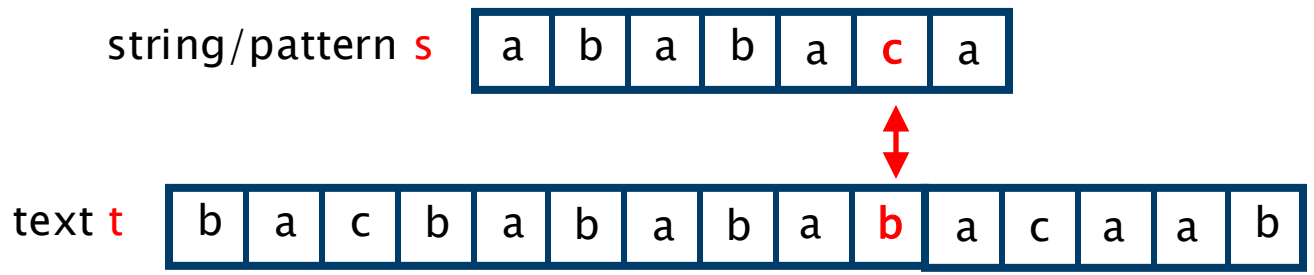
Boyer Moore – Example



Compare characters in text and string

position in string j=5

Boyer Moore – Example

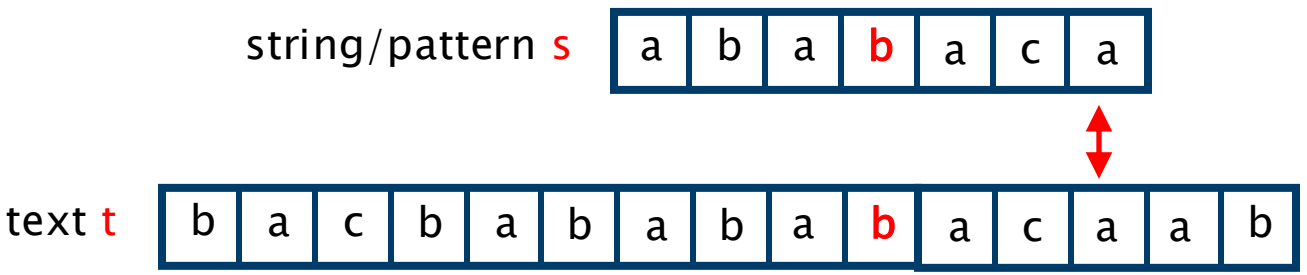


Characters do not match

position in string j=5

- case 1: the last position of character b in s is before position j
- move s so last b in s is aligned with that of t that was mismatched
- and start comparisons from end of the string

Boyer Moore – Example

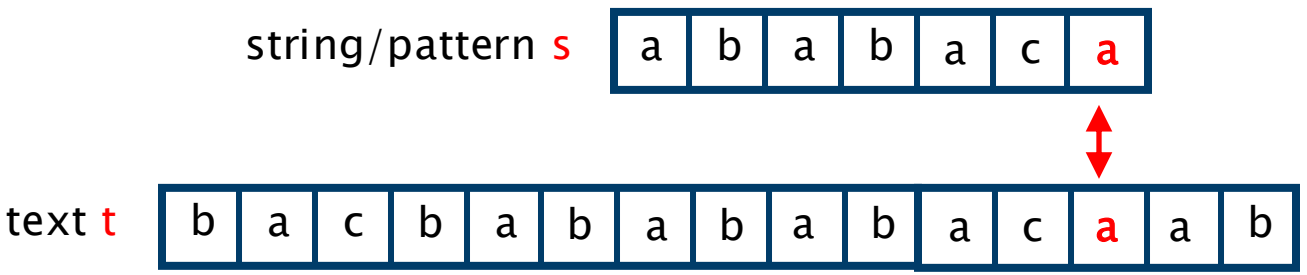


Characters do not match

position in string j=6

- case 1: the last position of character b in s is before position j
- move s so last b in s is aligned with that of t that was mismatched
- and start comparisons from end of the string

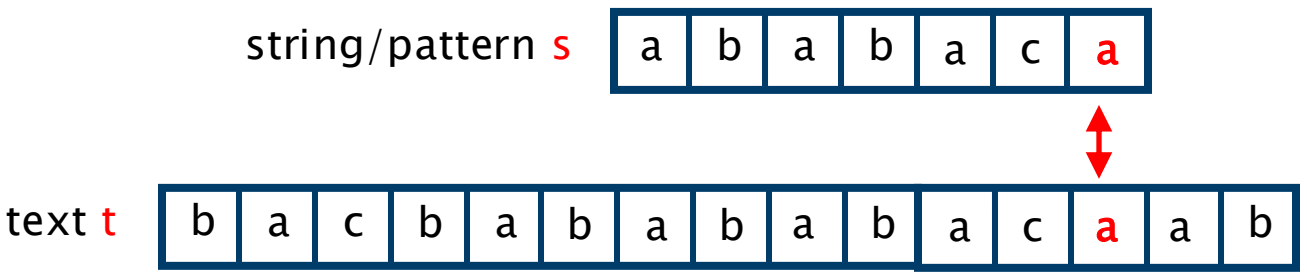
Boyer Moore – Example



Compare characters in text and string

position in string j=6

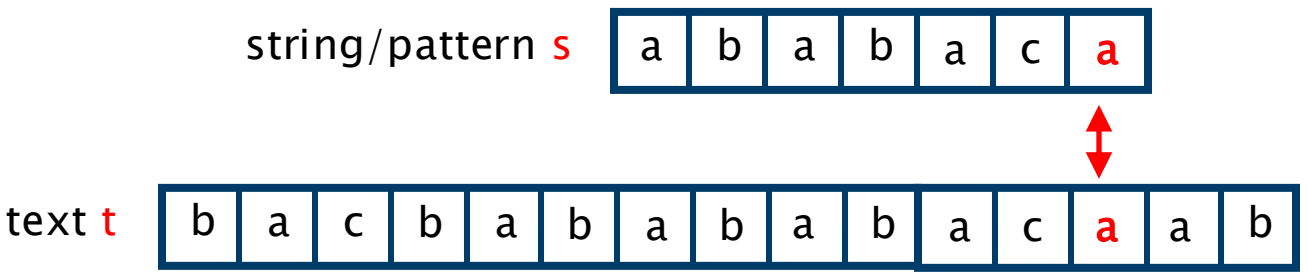
Boyer Moore – Example



Compare characters in text and string

position in string j=6

Boyer Moore – Example

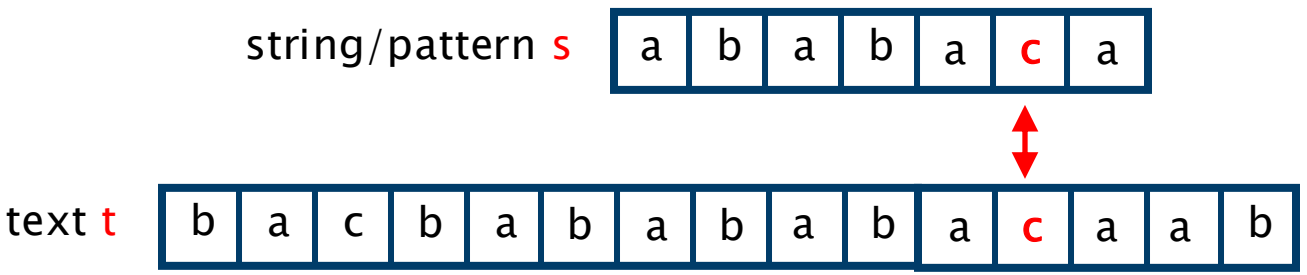


Characters match

position in string j=6

- decrement position in text and string

Boyer Moore – Example

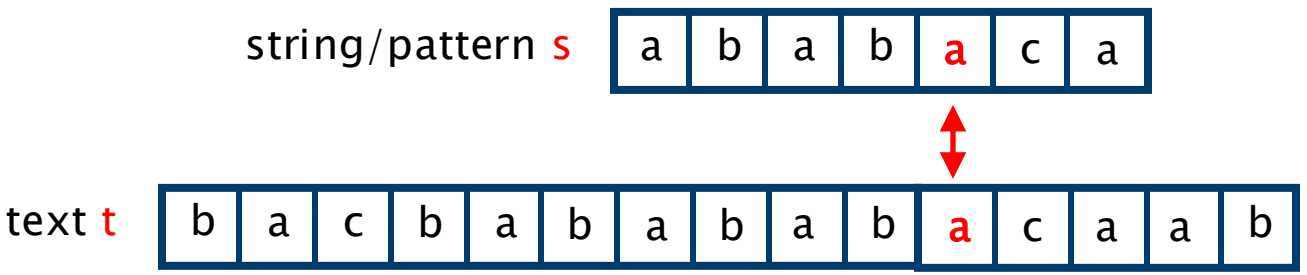


Characters continue to match so

– decrement position in text and string

position in string j=5

Boyer Moore – Example

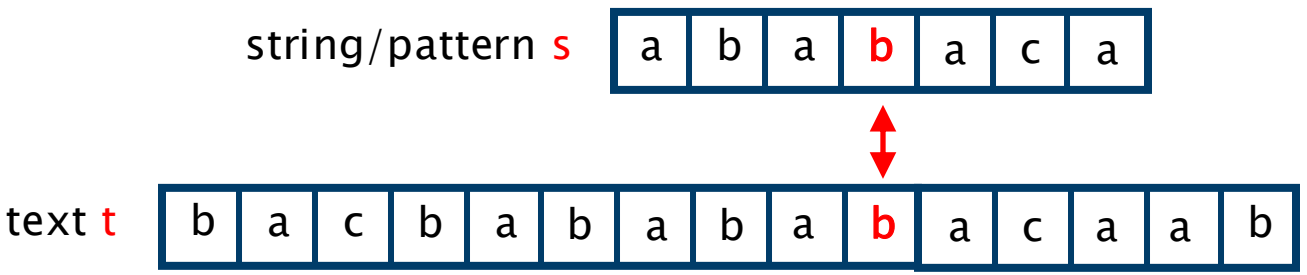


Characters continue to match so

– decrement position in text and string

position in string j=4

Boyer Moore – Example

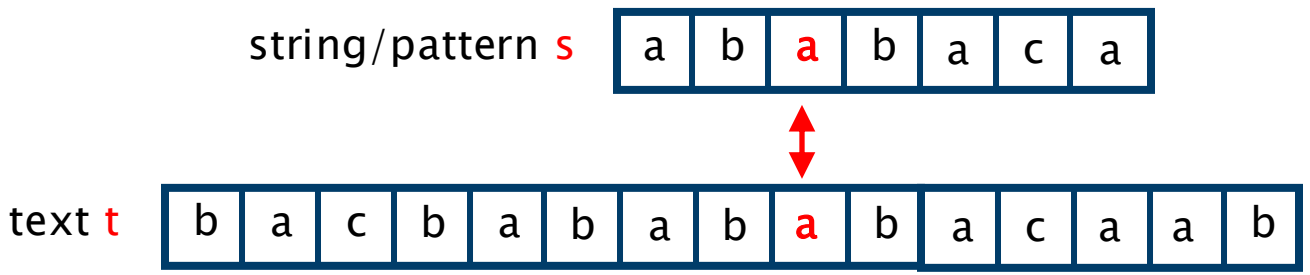


Characters continue to match so

– decrement position in text and string

position in string j=3

Boyer Moore – Example

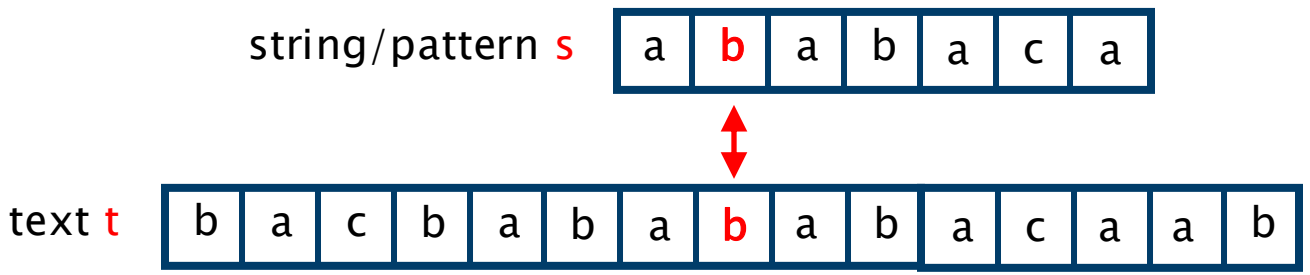


Characters continue to match so

– decrement position in text and string

position in string j=2

Boyer Moore – Example

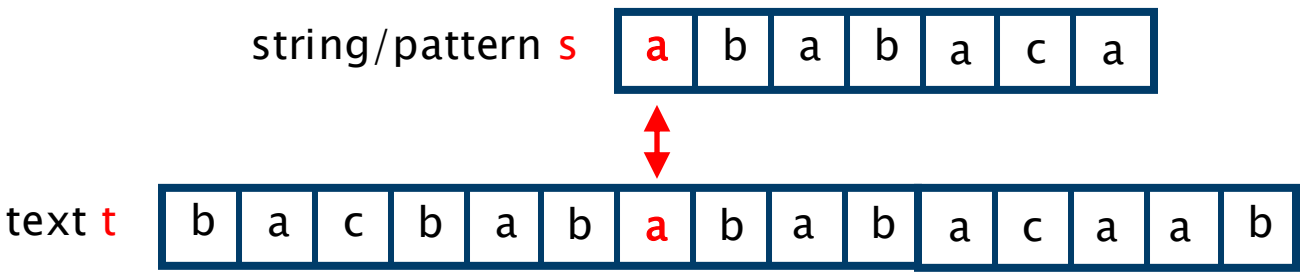


Characters continue to match so

- decrement position in text and string

position in string j=1

Boyer Moore – Example

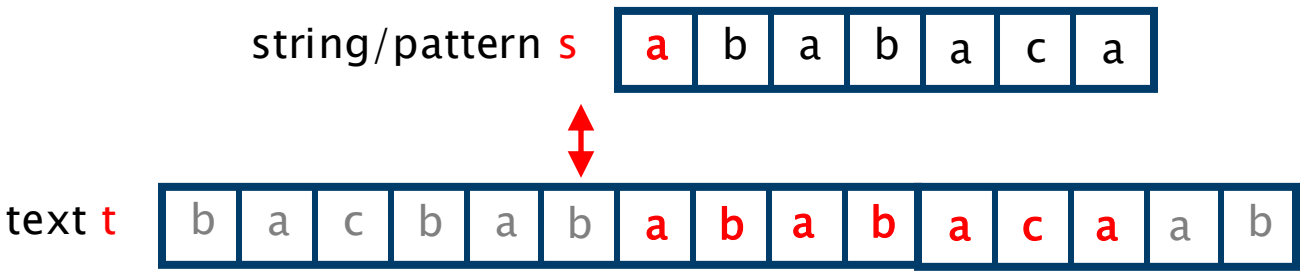


Characters continue to match so

- decrement position in text and string

position in string j=0

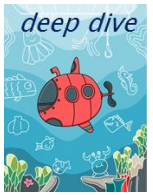
Boyer Moore – Example



String/pattern has been found

position in string `j=-1`

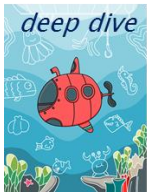
Boyer Moore – application example



Detecting malware signatures in network traffic

One of the critical challenges in cybersecurity is to quickly and accurately detect malware signatures within the vast amount of data passing through a network. As cyber threats evolve, malware becomes more sophisticated, often embedding itself in legitimate network traffic to avoid detection. Security systems must inspect packets of data in real-time to identify these threats without significantly impacting network performance.

Boyer Moore – application example



Detecting malware signatures in network traffic

Why use the BM algorithm:

- efficient string matching:
  - use it for scanning network packets for known malware signatures
  - skipping over non-matching portions of the packets allows for rapid analysis, minimising the impact on network performance
- enhanced accuracy:
  - the algorithm can be tuned to match exact strings (signatures) of various malware, reducing the chances of false positives and negatives
- scalability with network demand:
  - the algorithm's performance advantage grows with the length of the text (or packet size) being searched hence making it a scalable solution for malware detection in high-speed networks

Section 2 – Strings and text algorithms

Text compression

- Huffman encoding
- LZW compression/decompression

String comparison

- string distance

String/pattern search

- brute force algorithm
- KMP algorithm
- BM algorithm

Outline of course

Section 0: Quick recap on algorithm analysis

Section 1: Sorting algorithms

Section 2: Strings and text algorithms

Section 3: Graphs and graph algorithms – next week!

Section 4: An introduction to NP completeness

Section 5: A (very) brief introduction to computability

Remember!

Quiz for Week 1 opens up today at 12:00 up until tomorrow 23:59

Tutorial session tomorrow 2 hours to catch up with the tutorial exercises of this week