Algorithmics

Lecture 3

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

Strings – Notation

For a string $s=s_0s_1\ldots s_{m-1}$

- m is the length of the string
- $s[i]$ is the $(i+1)$th element of the string, i.e. s_i
- $s[i..j]$ is the substring from the ith to jth position, i.e. $s_is_{i+1}\ldots s_j$

Prefixes and suffixes

- jth prefix is the first j characters of s denoted $s[0..j-1]$
  - i.e. $s[0..j-1] = s_0s_1\ldots s_{j-1}$
  - $s[0..0-1]=s[0..-1]$ (the 0th prefix) is the empty string
- jth suffix is the last j characters of s denoted $s[m-j..m-1]$
  - i.e. $s[m-j..m-1] = s_{m-j}s_{m-j+1}\ldots s_{m-1}$
  - $s[m-0..m-1]=s[m..m-1]$ (the 0th suffix) is the empty string

String comparison

Fundamental question: how similar, or how different, are 2 strings?

- applications include:
    - biology (DNA and protein sequences)
    - file comparison (diff in Unix, and other similar file utilities)
    - spelling correction, speech recognition,…

A more precise formulation:

given strings $s=s_0s_2\ldots s_{m-1}$ and $t=t_0t_2\ldots t_{n-1}$ of lengths m and n,
what is the smallest number of basic operations needed to transform s to t?

'Basic' operations for transforming strings:

- **insert** a single character
- **delete** of a single character
- **substitute** one character by another

String comparison – String distance

The distance between s and t is defined to be the smallest number of basic operations needed to transform s to t

- for example consider the strings s and t

s:	a	b	a	d	c	d	b	
t:	a	c	b	a	c	a	c	b

- we can show an alignment between s and t that illustrates how 4 steps would suffice to transform s into t
- hence the distance between s and t is less than or equal to 4

insert 'c' delete 'd' substitute 'a' for 'd' insert 'c'

s:	a	~~c~~	b	a	~~d~~	c	a	~~c~~	b
t:	a	c	b	a	–	c	a	c	b

String comparison – String distance

The **distance** between **s** and **t** is defined to be the smallest number of basic operations needed to transform **s** into **t**

- for example for the strings

s:	a	b	a	d	c	d	b	
t:	a	c	b	a	c	a	c	b

the distance between s and t is less than or equal to 4

s:	a	-	b	a	d	c	d	-	b
t:	a	c	b	a	-	c	a	c	b

But could it be done in 3 steps?

- the answer is no, proof later based on our algorithm to find the distance for any two strings

String comparison – String distance

The distance between s and t is defined to be the smallest number of basic operations needed to transform s into t

– for example for the strings

s:	a	b	a	d	c	d	b	
t:	a	c	b	a	c	a	c	b

the distance between s and t is less than or equal to 4

s:	a	-	b	a	d	c	d	-	b
t:	a	c	b	a	-	c	a	c	b

But could it be done in 3 steps?

– the answer is no, proof later based on our algorithm to find the distance for any two strings, so above alignment is an optimal alignment

String comparison – String distance

More complex models are possible

- e.g., we can allocate a cost to each basic operation
- our methods adapt easily but we will stick to the unit-cost model

String comparison algorithms use dynamic programming

- the problem is solved by building up solutions to sub-problems of ever increasing size
- often called the tabular method (it builds up a table of relevant values)
- eventually, one of the values in the table gives the required answer

The dynamic programming technique has applications to many different problems

String distance – Dynamic programming

Recall the i^{th} prefix of string s is the first i characters of s

- let d(i,j) be the distance between i^{th} prefix of s and the j^{th} prefix of t
- distance between s and t is then d(m,n)
  (since s and t of lengths m and n)

The basis of dynamic programming method is a recurrence relation

- more precisely we define the distance d(i,j) between i^{th} prefix of s and the j^{th} prefix of t in terms of the distance between shorter prefixes
  - i.e., in terms of the distances d(i-1,j-1), d(i,j-1) and d(i-1,j)

- in the base cases we set d(i,0)=i and d(0,j)=j for all i≤n and j≤m
- since the distance from/to an empty string to/from a string of length k is equal to k (we require k insertions/deletions)

String distance – Dynamic programming

In an optimal alignment of the `i`th prefix of `s` with the `j`th prefix of `t` the last position of the alignment must either be of the form:

$\boxed{\begin{matrix} * \\ * \end{matrix}}$ if `s[i-1] = t[j-1]` and $\boxed{\begin{matrix} - \\ * \end{matrix}}$, $\boxed{\begin{matrix} * \\ - \end{matrix}}$ or $\boxed{\begin{matrix} * \\ \$ \end{matrix}}$ otherwise

where – is a gap, while * and $ are arbitrary but different characters

String distance – Dynamic programming

In an optimal alignment of the `i`th prefix of `s` with the `j`th prefix of `t` the last position of the alignment must either be of the form:

$\boxed{\begin{array}{c} * \\ * \end{array}}$ if `s[i-1] = t[j-1]` and $\boxed{\begin{array}{c} - \\ * \end{array}}$, $\boxed{\begin{array}{c} * \\ - \end{array}}$ or $\boxed{\begin{array}{c} * \\ \$ \end{array}}$ otherwise

where – is a gap, while * and $ are arbitrary but different characters

In this case, no operations are required and the distance is given by that between the `i-1`th and `j-1`th prefixes of `s` and `t`

String distance – Dynamic programming

In an optimal alignment of the **i**th prefix of **s** with the **j**th prefix of **t** the last position of the alignment must either be of the form:

$\begin{bmatrix} * \\ * \end{bmatrix}$ if `s[i-1] = t[j-1]` and $\begin{bmatrix} - \\ * \end{bmatrix}$, $\begin{bmatrix} * \\ - \end{bmatrix}$ or $\begin{bmatrix} * \\ \$ \end{bmatrix}$ otherwise

where – is a gap, while * and $ are arbitrary but different characters

In this case, no operations are required and the distance is given by that between the **i-1**th and **j-1**th prefixes of **s** and **t**

$$\texttt{d(i,j)} = \begin{cases} \texttt{d(i-1,j-1)} & \text{if } \texttt{s[i-1]=t[j-1]} \\ & \text{otherwise} \end{cases}$$

String distance – Dynamic programming

In an optimal alignment of the i^{th} prefix of **s** with the j^{th} prefix of **t** the last position of the alignment must either be of the form:

$\boxed{\begin{array}{c} * \\ * \end{array}}$ if s[i-1] = t[j-1] and $\boxed{\begin{array}{c} - \\ * \end{array}}$, $\boxed{\begin{array}{c} * \\ - \end{array}}$ or $\boxed{\begin{array}{c} * \\ \$ \end{array}}$ otherwise

where – is a gap, while * and $ are arbitrary but different characters

In this case, **insert** element into **s** and distance given by **1** (for the insertion) plus distance between i^{th} prefix of **s** and $i-1^{th}$ prefix of **t**

$$d(i,j) = \begin{cases} d(i-1,j-1) & \text{if } s[i-1]=t[j-1] \\ 1 + \min\{\ d(i,j-1)\ \} & \text{otherwise} \end{cases}$$

String distance – Dynamic programming

In an optimal alignment of the **i**th prefix of **s** with the **j**th prefix of **t** the last position of the alignment must either be of the form:

$\boxed{\begin{matrix}*\\ *\end{matrix}}$ if s[i-1] = t[j-1] and $\boxed{\begin{matrix}-\\ *\end{matrix}}$, $\boxed{\begin{matrix}*\\ -\end{matrix}}$ or $\boxed{\begin{matrix}*\\ \$\end{matrix}}$ otherwise

where – is a gap, while * and $ are arbitrary but different characters

In this case, **delete** an element from **s** and distance given by **1** plus distance between **i-1**th prefix of **s** and **i**th prefix of **t**

$$
d(i,j) = \begin{cases} d(i-1,j-1) & \text{if } s[i-1]=t[j-1] \\ 1 + \min\{\, d(i,j-1),\ d(i-1,j),\ \} & \text{otherwise} \end{cases}
$$

String distance – Dynamic programming

In an optimal alignment of the i^{th} prefix of **s** with the j^{th} prefix of **t** the last position of the alignment must either be of the form:

$\boxed{\begin{array}{c} * \\ * \end{array}}$ if s[i-1] = t[j-1] and $\boxed{\begin{array}{c} - \\ * \end{array}}$, $\boxed{\begin{array}{c} * \\ - \end{array}}$ or $\boxed{\begin{array}{c} * \\ \$ \end{array}}$ otherwise

where – is a gap, while * and $ are arbitrary but different characters

In this case, **substitute** an element in **s** and distance given by **1** plus distance between **i-1**th prefix of **s** and **i-1**th prefix of **t**

$$d(i,j) = \begin{cases} d(i-1,j-1) & \text{if } s[i-1]=t[j-1] \\ 1 + \min\{\, d(i,j-1),\ d(i-1,j),\ d(i-1,j-1) \,\} & \text{otherwise} \end{cases}$$

String distance – Dynamic programming

In an optimal alignment of the **i**th prefix of **s** with the **j**th prefix of **t** the last position of the alignment must either be of the form:

$\begin{bmatrix} * \\ * \end{bmatrix}$ if s[i-1] = t[j-1] and $\begin{bmatrix} - \\ * \end{bmatrix}$, $\begin{bmatrix} * \\ - \end{bmatrix}$ or $\begin{bmatrix} * \\ \$ \end{bmatrix}$ otherwise

where – is a gap, while * and $ are arbitrary but different characters

We take the minimum when **s[i-1]≠t[j-1]** as we want the optimal (minimal) distance

$$d(i,j) = \begin{cases} d(i-1,j-1) & \text{if } s[i-1]=t[j-1] \\ 1 + \min\{\, d(i,j-1),\ d(i-1,j),\ d(i-1,j-1) \,\} & \text{otherwise} \end{cases}$$

String distance – Dynamic programming

The complete recurrence relation is given by:

$$
\texttt{d(i,j)} = \begin{cases} \texttt{d(i-1,j-1)} & \text{if } \texttt{s[i-1]=t[j-1]} \\ \texttt{1+min\{ d(i,j-1),d(i-1,j),d(i-1,j-1)\}} & \text{otherwise} \end{cases}
$$

subject to `d(i,0)=i` and `d(0,j)=j` for all `i≤n-1` and `j≤m-1`

String distance – Example

s\t		0	1	2	3	4	5	6	7	8
			a	c	b	a	c	a	c	b
0										
1	a									
2	b									
3	a									
4	d									
5	c									
6	d									
7	b									

String distance – Example

s\t		0	1	2	3	4	5	6	7	8
			a	c	b	a	c	a	c	b
0		0	1	2	3	4	5	6	7	8
1	a	1								
2	b	2								
3	a	3								
4	d	4								
5	c	5								
6	d	6								
7	b	7								

Table initialised by filling row **0** and column **0** using initial conditions of recurrence relation (**d(i,0)=i** and **d(0,j)=j** for **i≤n** and **j≤m**)

String distance – Example

s\t		0	1	2	3	4	5	6	7	8
			a	c	b	a	c	a	c	b
0		0	1	2	3	4	5	6	7	8
1	a	1	0							
2	b	2								
3	a	3								
4	d	4								
5	c	5								
6	d	6								
7	b	7								

The entries are calculated one by one by application of the formula

– `d(1,1)=0` since `s[1-1]=t[1-1]` and `d(0,0)=0`

String distance – Example

s\t		0	1	2	3	4	5	6	7	8
			a	c	b	a	c	a	c	b
0		0	1	2	3	4	5	6	7	8
1	a	1	0	1						
2	b	2								
3	a	3								
4	d	4								
5	c	5								
6	d	6								
7	b	7								

The entries are calculated one by one by application of the formula

– `d(1,2)=1+0` since `s[1-1]≠t[2-1]` and `min(d(1,1),d(0,2),d(0,1))=0`)

String distance – Example

s\t		0	1	2	3	4	5	6	7	8
			a	c	b	a	c	a	c	b
0		0	1	2	3	4	5	6	7	8
1	a	1	0	1	2					
2	b	2								
3	a	3								
4	d	4								
5	c	5								
6	d	6								
7	b	7								

The entries are calculated one by one by application of the formula

- `d(1,3)=1+1` since `s[1-1]≠t[3-1]` and `min(d(0,2),d(0,3),d(1,2))=1`

String distance – Example

s\t		0	1	2	3	4	5	6	7	8
			a	c	b	a	c	a	c	b
0		0	1	2	3	4	5	6	7	8
1	a	1	0	1	2	3	4	5	6	7
2	b	2	1							
3	a	3								
4	d	4								
5	c	5								
6	d	6								
7	b	7								

The entries are calculated one by one by application of the formula

– `d(2,1)=1+0=1` since `s[2-1]≠t[1-1]` and `min(d(2,0),d(1,1),d(1,0))=0`

String distance – Example

s\t		0	1	2	3	4	5	6	7	8
			a	c	**b**	a	c	a	c	b
0		0	1	2	3	4	5	6	7	8
1	a	1	0	1	2	3	4	5	6	7
2	b	2	1	**1**	**1**	2	3	4	5	6
3	**a**	3	2	2	**2**					
4	d	4								
5	c	5								
6	d	6								
7	b	7								

The entries are calculated one by one by application of the formula

– `d(3,3)=1+1=2` since `s[3-1]≠t[3-1]` and `min(d(3,2),d(2,3),d(2,2))=1`

String distance – Example

s\t		0	1	2	3	4	5	6	7	8
			a	c	b	a	c	a	c	b
0		0	1	2	3	4	5	6	7	8
1	a	1	0	1	2	3	4	5	6	7
2	b	2	1	1	1	2	3	4	5	6
3	a	3	2	2	2	1	2	3	4	5
4	d	4	3	3	3	2	2	3	4	5
5	c	5	4	3	4	3	2	3	3	4
6	d	6	5	4	4	4	3	3	**4**	4
7	b	7	6	5	4	5	4	4	4	**4**

The entries are calculated one by one by application of the formula

– `d(7,8)=4` since `s[7-1]=t[8-1]` and `d(6,7)=4`

String distance – Example

s\t		0	1	2	3	4	5	6	7	8
			a	c	b	a	c	a	c	b
0		0	1	2	3	4	5	6	7	8
1	a	1	0	1	2	3	4	5	6	7
2	b	2	1	1	1	2	3	4	5	6
3	a	3	2	2	2	1	2	3	4	5
4	d	4	3	3	3	2	2	3	4	5
5	c	5	4	3	4	3	2	3	3	4
6	d	6	5	4	4	4	3	3	4	4
7	b	7	6	5	4	5	4	4	4	4

The entries are calculated one by one by application of the formula

- the final table: d(7,8)=4 so the string distance is 4

String distance – Dynamic programming

The dynamic programming algorithm for string distance comes immediately from the formula

- fill in the entries of an $m \times n$ table row by row, and column by column

Time and space complexity both $O(m \cdot n)$

- a consequence of the size of the table
- can easily reduce the space complexity to $O(m+n)$
- just keep the most recent entry in each column of the table

But what about obtaining an optimal alignment?

- can use a ‘traceback’ in the table (see next slides)
- less obvious how this can be done using only $O(m+n)$ space
- but in fact it turns out that it's still possible (Hirschberg's algorithm)

String distance – Dynamic programming

The traceback phase used to construct an optimal alignment

- trace a path in the table from bottom right to top left
- draw an arrow from an entry to the entry that led to its value

Interpretation

- vertical steps as `deletions`
- horizontal steps as `insertions`
- diagonal steps as `matches` or `substitutions`
  - a `match` if the distance does not change and a `substitution` otherwise

The traceback is not necessarily unique

- since there can be more than one optimal alignment

String distance – Example (traceback)

s\t		0	1	2	3	4	5	6	7	8
			a	c	b	a	c	a	c	b
0		0	1	2	3	4	5	6	7	8
1	a	1	0	1	2	3	4	5	6	7
2	b	2	1	1	1	2	3	4	5	6
3	a	3	2	2	2	1	2	3	4	5
4	d	4	3	3	3	2	2	3	4	5
5	c	5	4	3	4	3	2	3	3	4
6	d	6	5	4	4	4	3	3	4	4
7	b	7	6	5	4	5	4	4	4	4

– `d(7,8)=d(6,7)` since `s[7-1]=t[8-1]`

String distance – Example (traceback)

s\t		0	1	2	3	4	5	6	7	8
			a	c	b	a	c	a	c	b
0		0	1	2	3	4	5	6	7	8
1	a	1	0	1	2	3	4	5	6	7
2	b	2	1	1	1	2	3	4	5	6
3	a	3	2	2	2	1	2	3	4	5
4	d	4	3	3	3	2	2	3	4	5
5	c	5	4	3	4	3	2	3	3	4
6	d	6	5	4	4	4	3	3	4	4
7	b	7	6	5	4	5	4	4	4	4

– `d(6,7)=1+d(5,7)` since `s[6-1]≠t[7-1]` and `d(5,7)=min(d(5,7),d(6,6),d(5,6))` could have also taken horizontal or diagonal step

String distance – Example (traceback)

s\t		0	1	2	3	4	5	6	7	8
			a	c	b	a	c	a	c	b
0		0	1	2	3	4	5	6	7	8
1	a	1	0	1	2	3	4	5	6	7
2	b	2	1	1	1	2	3	4	5	6
3	a	3	2	2	2	1	2	3	4	5
4	d	4	3	3	3	2	2	3	4	5
5	c	5	4	3	4	3	2	3	3	4
6	d	6	5	4	4	4	3	3	4	4
7	b	7	6	5	4	5	4	4	4	4

– `d(5,7)=1+d(4,6)` since `s[5-1]=t[7-1]`

String distance – Example (traceback)

s\t		0	1	2	3	4	5	6	7	8
			a	c	b	a	c	a	c	b
0		0	1	2	3	4	5	6	7	8
1	a	1	0	1	2	3	4	5	6	7
2	b	2	1	1	1	2	3	4	5	6
3	a	3	2	2	2	1	2	3	4	5
4	d	4	3	3	3	2	2	3	4	5
5	c	5	4	3	4	3	2	3	3	4
6	d	6	5	4	4	4	3	3	4	4
7	b	7	6	5	4	5	4	4	4	4

– d(4,6)=1+d(4,5) since s[4-1]≠t[6-1] and d(4,5)=min(d(3,6),d(4,5),d(3,5)) could have also taken the diagonal step

String distance – Example (traceback)

s\t		0	1	2	3	4	5	6	7	8
			a	c	b	a	c	a	c	b
0		0	1	2	3	4	5	6	7	8
1	a	1	0	1	2	3	4	5	6	7
2	b	2	1	1	1	2	3	4	5	6
3	a	3	2	2	2	1	2	3	4	5
4	d	4	3	3	3	2	2	3	4	5
5	c	5	4	3	4	3	2	3	3	4
6	d	6	5	4	4	4	3	3	4	4
7	b	7	6	5	4	5	4	4	4	4

String distance – Example (traceback)

s\t		0	1	2	3	4	5	6	7	8
			a	c	b	a	c	a	c	b
0		0	1	2	3	4	5	6	7	8
1	a	1	0	1	2	3	4	5	6	7
2	b	2	1	1	1	2	3	4	5	6
3	a	3	2	2	2	1	2	3	4	5
4	d	4	3	3	3	2	2	3	4	5
5	c	5	4	3	4	3	2	3	3	4
6	d	6	5	4	4	4	3	3	4	4
7	b	7	6	5	4	5	4	4	4	4

Corresponding alignment:

```
   s: a   -   b   a   d   -   c   d   b
   t: a   c   b   a   c   a   c   -   b
step: d ← h ← d ← d ← d ← h ← d ← v ← d
```

(d=diagonal, v = vertical, h = horizontal)

String distance – Example (traceback)

Corresponding alignment:

match insert ‘c’ match match substitute ‘c’ for ‘d’ insert ‘a’ match delete ‘d’ match

s:	a	c	b	a	c	a	c	-	b
t:	a	c	b	a	c	a	c	-	b

step: d ← h ← d ← d ← d ← h ← d ← v ← d

Interpretation

- v (vertical) steps deletions
- h (horizontal) steps insertions
- d (diagonal) steps as substitutions or matches

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/pattern search

Searching a (long) text for a (short) string/pattern

- many applications including
  - information retrieval
  - text editing
  - computational biology

Many variants, such as exact or approximate matches

- first occurrence or all occurrences
- one text and many strings/patterns
- many texts and one string/pattern

We describe three different solutions to the basic problem:

- given a text `t` (of length `n`) and a string/pattern `s` (of length `m`)
- find the position of the first occurrence (if it exists) of `s` in `t`
- usually `n` is large and `m` is small

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
- BM algorithm

String search – Brute force algorithm

Given a text **t** (of length **n**) and a string/pattern **s** (of length **m**) find the position of the first occurrence (if any) of **s** in **t**

The naive **brute force** algorithm

- also known as exhaustive search (as we simply test all possible positions)
- set the current starting position in the text to be zero
- compare text and string characters left-to-right until the entire string is matched or a character mismatches
- in the case of a mismatch

  advance the starting position in the text by 1 and repeat
- continue until a match is found or the text is exhausted

Algorithms expressed with **char arrays** rather than **strings** in Java

String search – Brute force algorithm

```
/** return smallest k such that s occurs in t starting at position k */
public int bruteForce (char[] s, char[] t){
   int m = s.length; // length of string/pattern
   int n = t.length; // length of text
   int sp = 0;  // starting position in text t
   int i = 0;   // curr position in text
   int j = 0;   // curr position in string/pattern s
   while (sp <= n-m && j < m) { // not reached end of text/string
      if (t[i] == s[j]){ // chars match
         i++;     // move on in text
         j++;     // move on in string/pattern
      } else {    // a mismatch
         j = 0;   // start again in string
         sp++;    // advance starting position
         i = sp;  // back up in text to new starting position
       }
   }
   if (j == m) return sp; // occurrence found (reached end of string)
   else return -1; // no occurrence (reached end of text)
}
```

Brute Force – Example

string/pattern s

a	b	a	b	a	c	a

text t

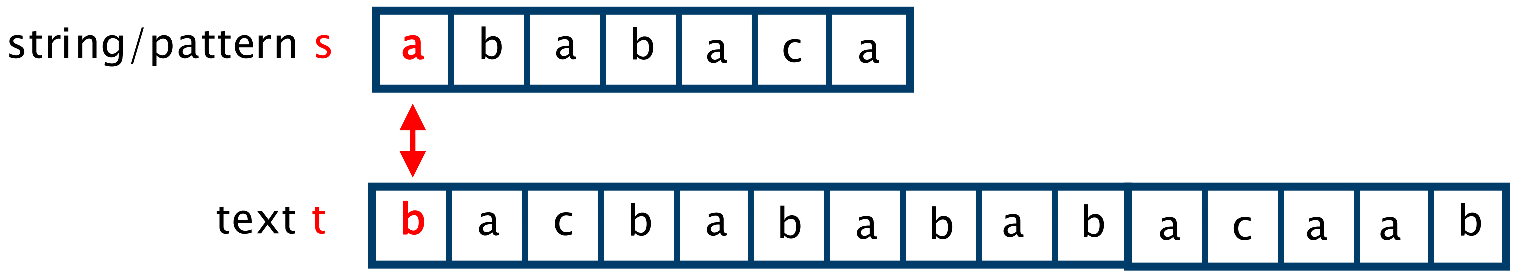
b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

Starting position:

– start of text and string

position in string $j=0$

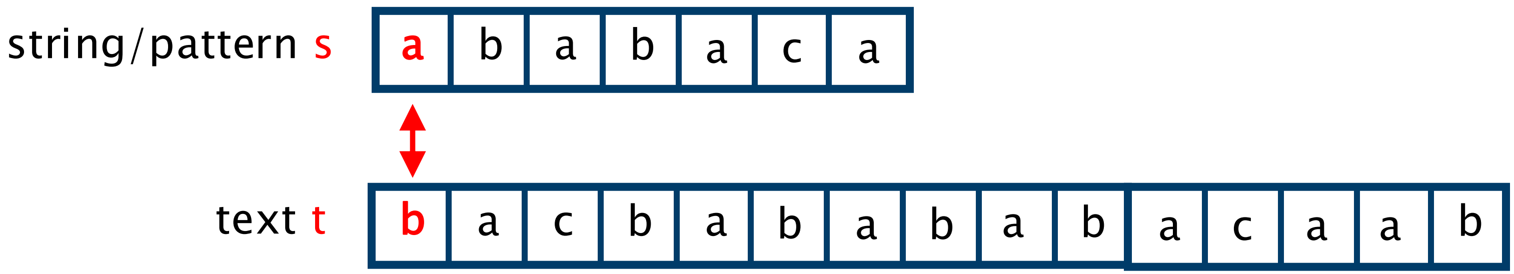
Brute Force – Example



Compare characters in text and string

position in string j=0

Brute Force – Example



Characters do not match

position in string j=0

– advance the starting position in the text by 1 and start comparison again

Brute Force – Example

string/pattern s

a	b	a	b	a	c	a

text t

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

Characters do not match

position in string j=0

– advance the starting position in the text by 1 and start comparison again

Brute Force – Example

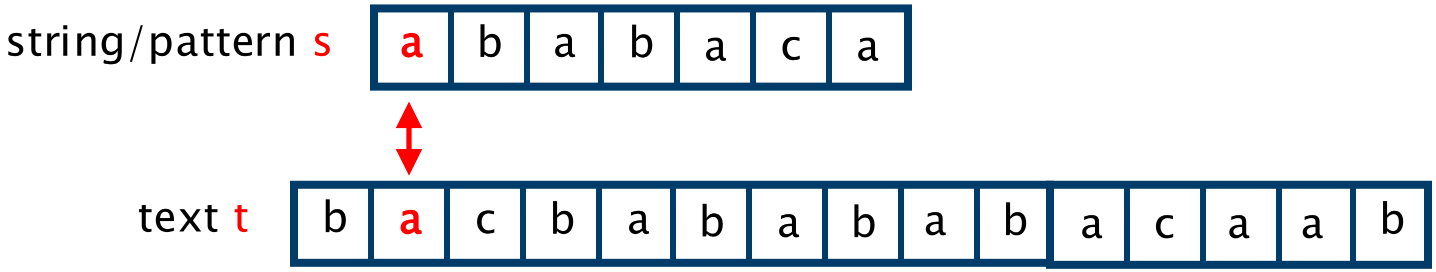
string/pattern s	a	b	a	b	a	c	a

text t	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

Compare characters in text and string

position in string j=0

Brute Force – Example

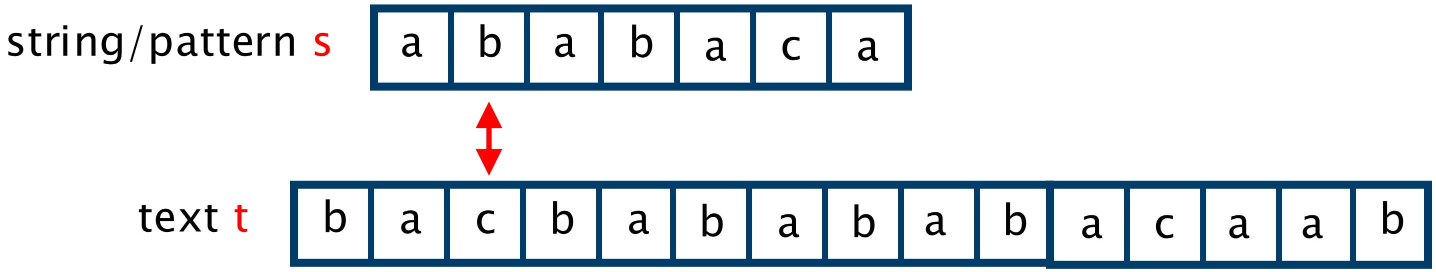


Characters match

position in string $j=0$

– increment position in text and string

Brute Force – Example



Characters match

position in string $j=1$

- increment position in text and string

Brute Force – Example

string/pattern s

a	b	a	b	a	c	a

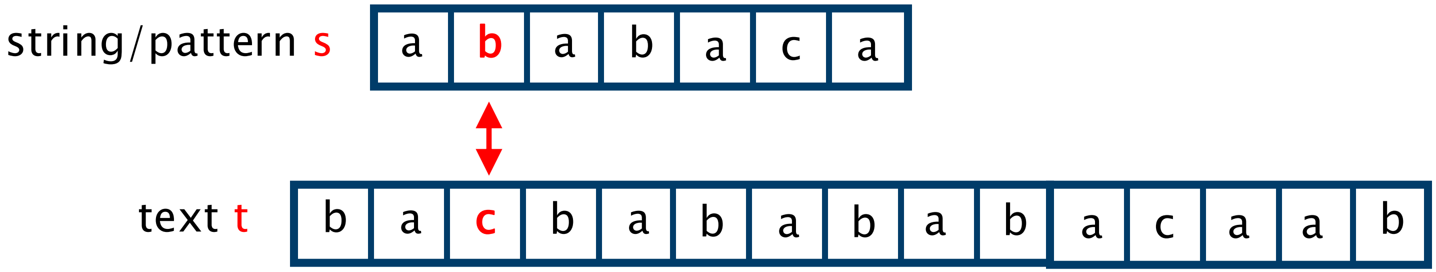
text t

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

Compare characters in text and string

position in string j=1

Brute Force – Example

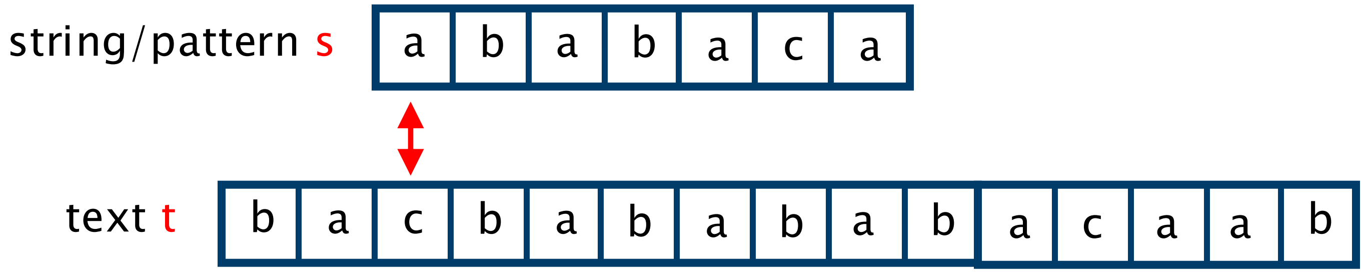


Characters do not match

position in string j=1

- reset the position in the string to 0 and start comparison again

Brute Force – Example

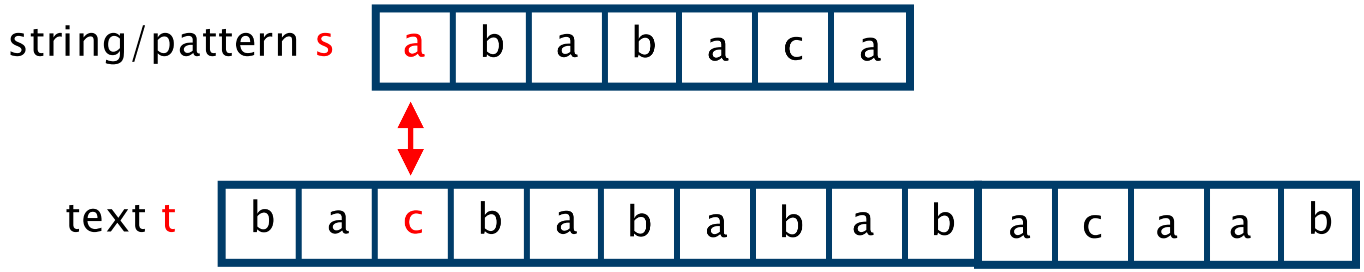


Characters do not match

position in string j=0

– reset the position in the string to 0 and start comparison again

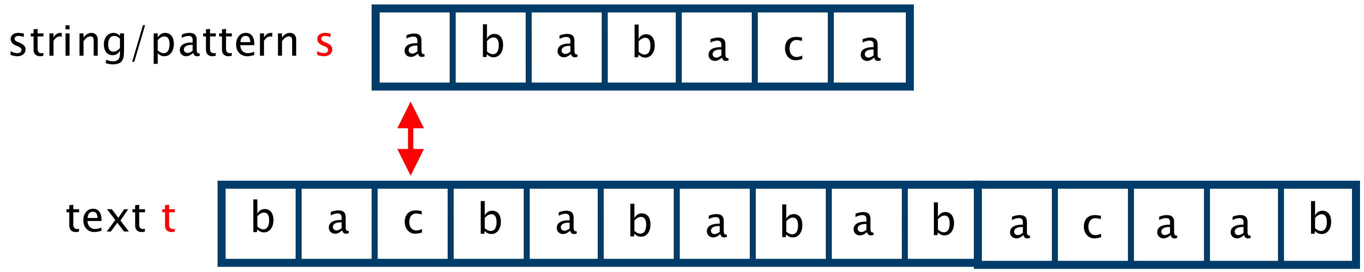
Brute Force – Example



Compare characters in text and string

position in string $j=0$

Brute Force – Example

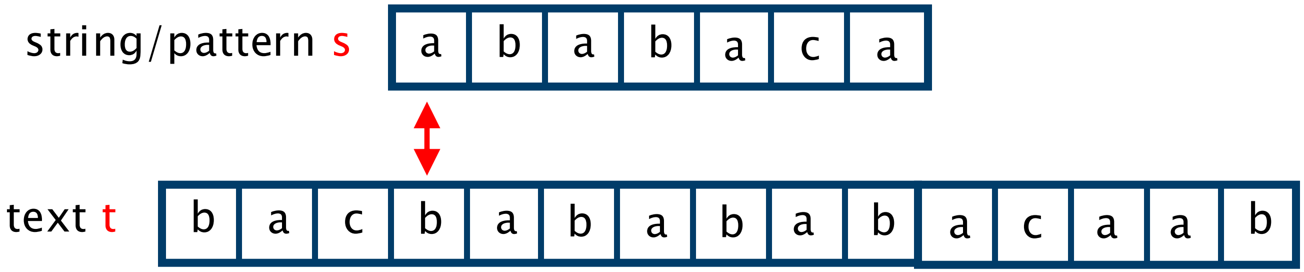


Compare characters in text and string

position in string j=0

– advance the starting position in the text by 1 and start comparison again

Brute Force – Example

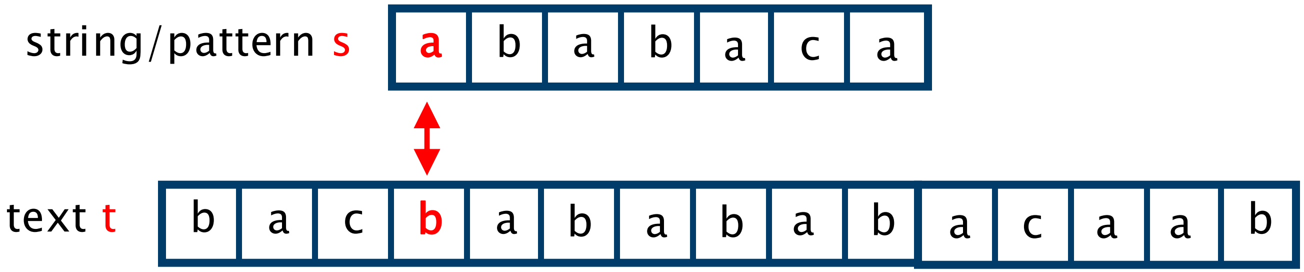


Compare characters in text and string

position in string $j=0$

– advance the starting position in the text by 1 and start comparison again

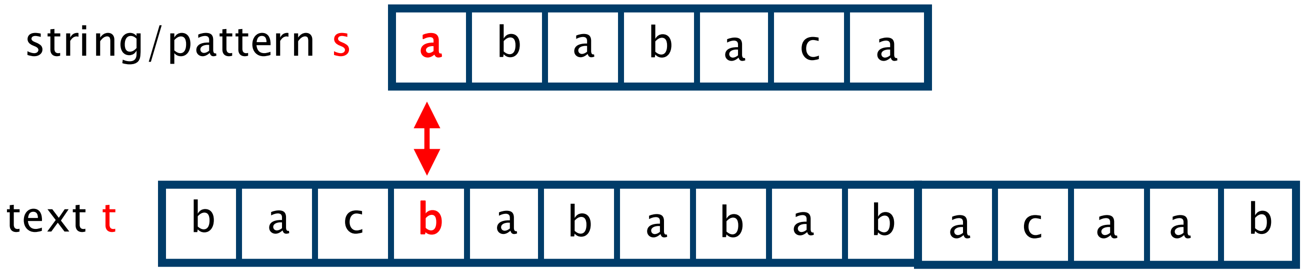
Brute Force – Example



Compare characters in text and string

position in string j=0

Brute Force – Example

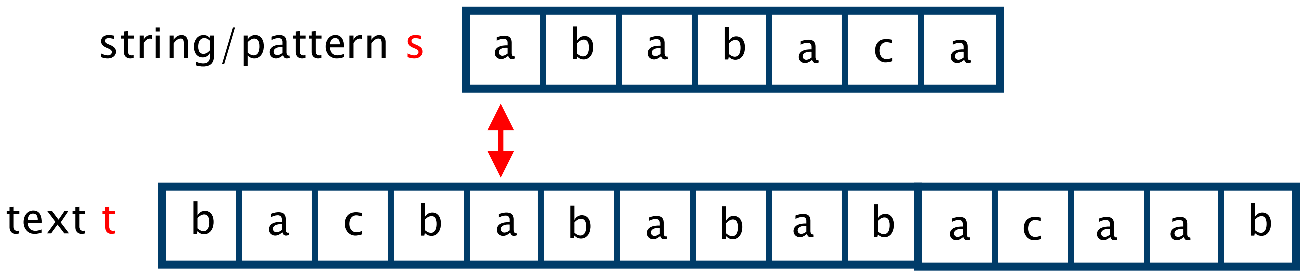


Characters do not match

position in string j=0

– advance the starting position in the text by 1 and start comparison again

Brute Force – Example

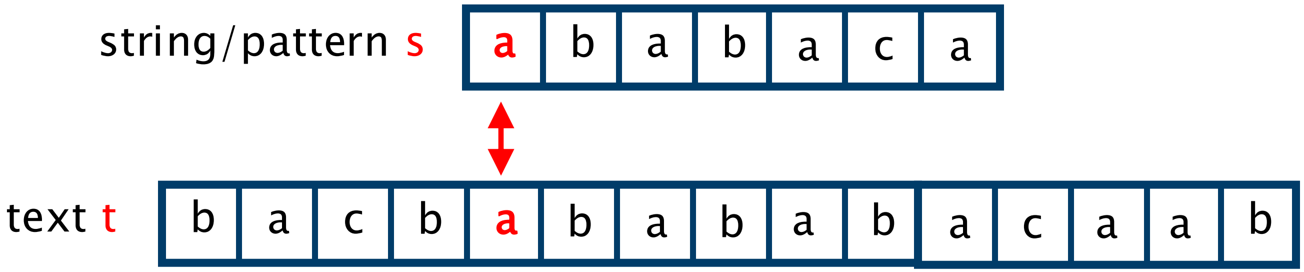


Characters do not match

position in string j=0

– advance the starting position in the text by 1 and start comparison again

Brute Force – Example



Compare characters in text and string

position in string j=0

Brute Force – Example

string/pattern s

a	b	a	b	a	c	a

text t

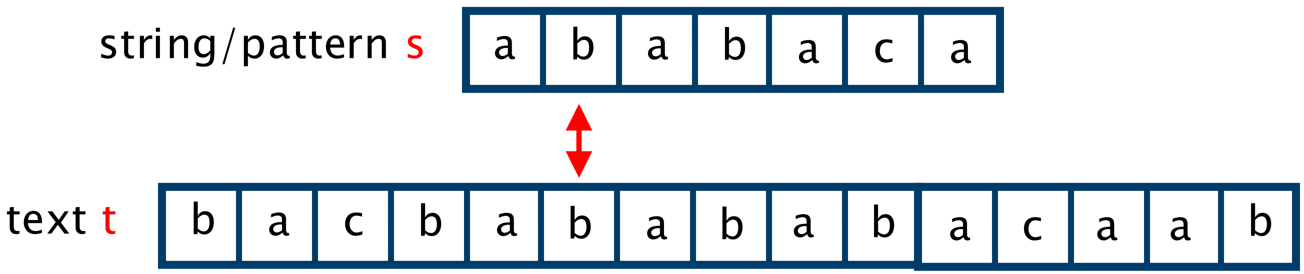
b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

position in string j=0

Characters match

– increment position in text and string

Brute Force – Example



Characters match

position in string $j=1$

- increment position in text and string

Brute Force – Example

string/pattern s	a	b	a	b	a	c	a								
text t	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

Compare characters in text and string

position in string j=1

Brute Force – Example

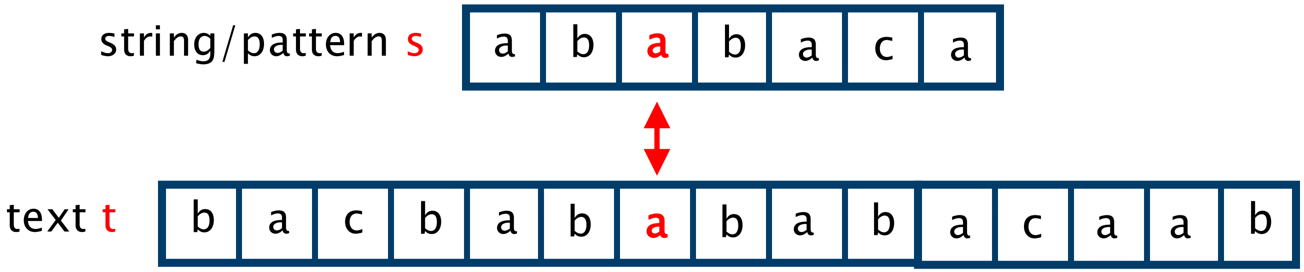
string/pattern s	a	b	a	b	a	c	a								
text t	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

Characters match

position in string $j=1$

- increment position in text and string

Brute Force – Example



Characters continue to match so

position in string j=2

- increment position in text and string

Brute Force – Example

string/pattern s

a	b	a	b	a	c	a

text t

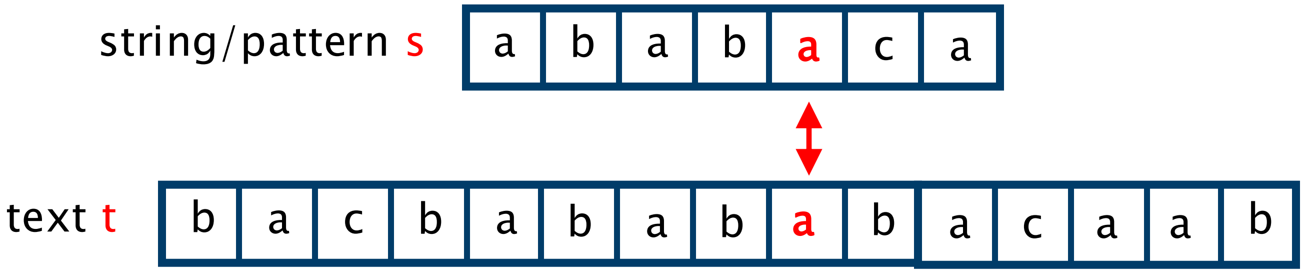
b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

Characters continue to match so

position in string j=3

– increment position in text and string

Brute Force – Example

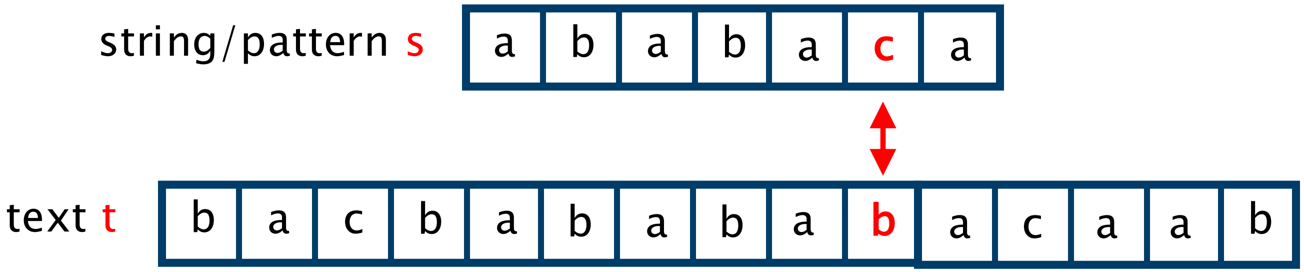


Characters continue to match so

position in string j=4

– increment position in text and string

Brute Force – Example



Characters do not match

position in string j=5

– advance the starting position in the text by 1 and start comparison again

Brute Force – Example

string/pattern s	a	b	a	b	a	c	a

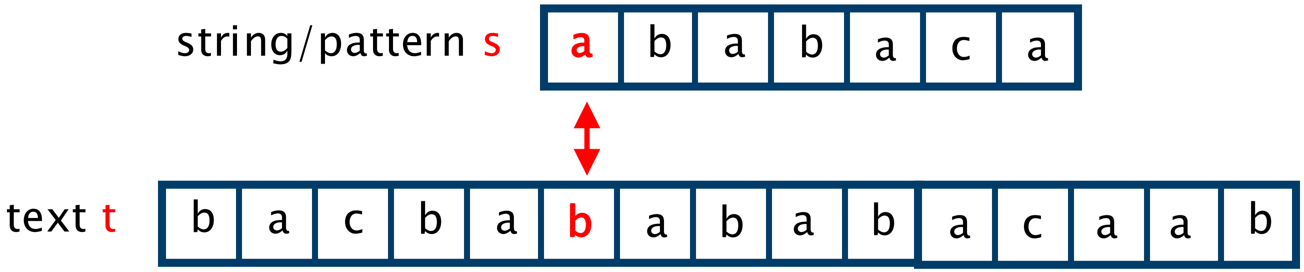
text t	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

Characters do not match

position in string j=0

– advance the starting position in the text by 1 and start comparison again

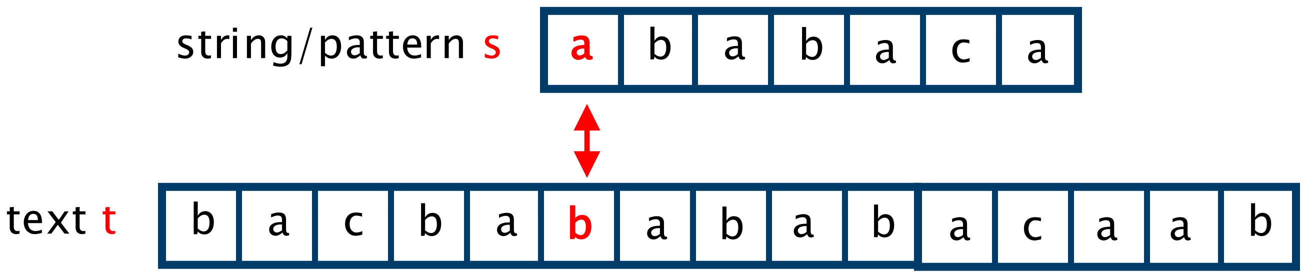
Brute Force – Example



Compare characters in text and string

position in string $j=0$

Brute Force – Example

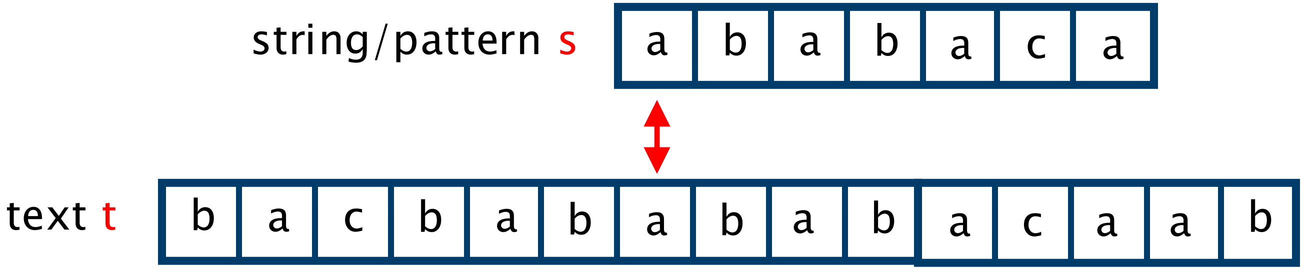


Characters do not match

position in string $j=0$

– advance the starting position in the text by 1 and start comparison again

Brute Force – Example

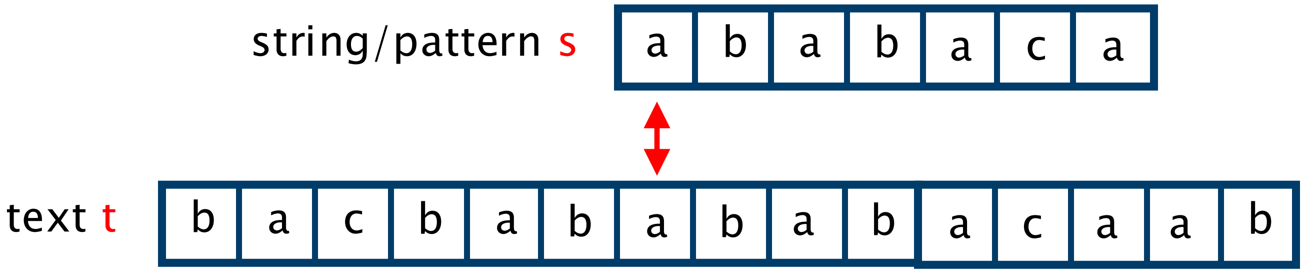


Characters do not match

position in string $j=0$

– advance the starting position in the text by 1 and start comparison again

Brute Force – Example

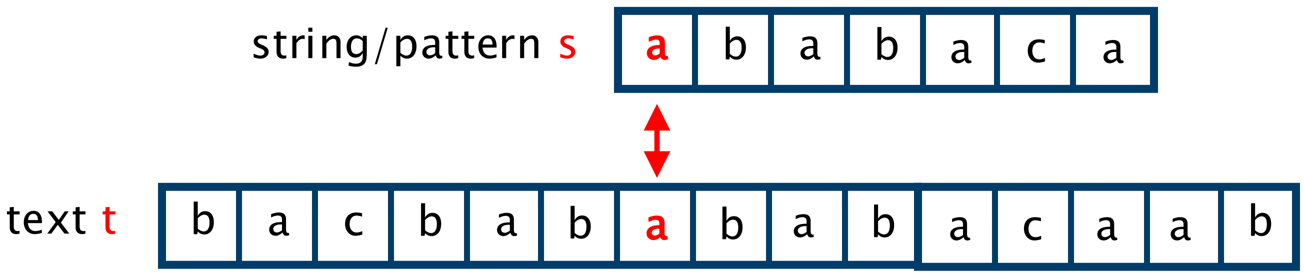


Characters do not match

position in string $j=0$

– advance the starting position in the text by 1 and start comparison again

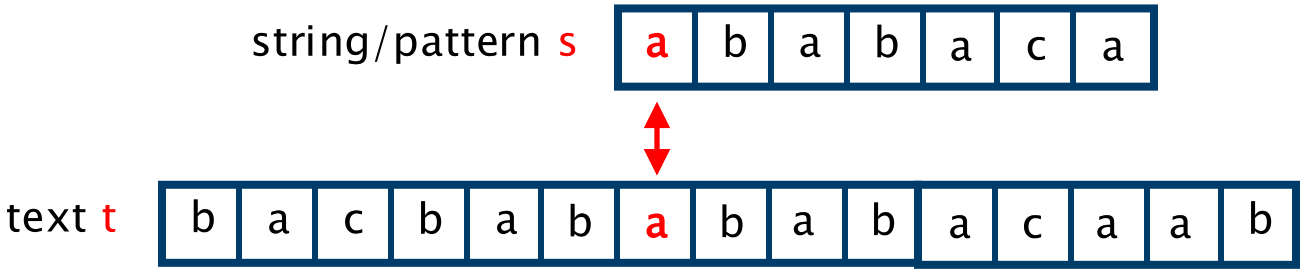
Brute Force – Example



Compare characters in text and string

position in string j=0

Brute Force – Example

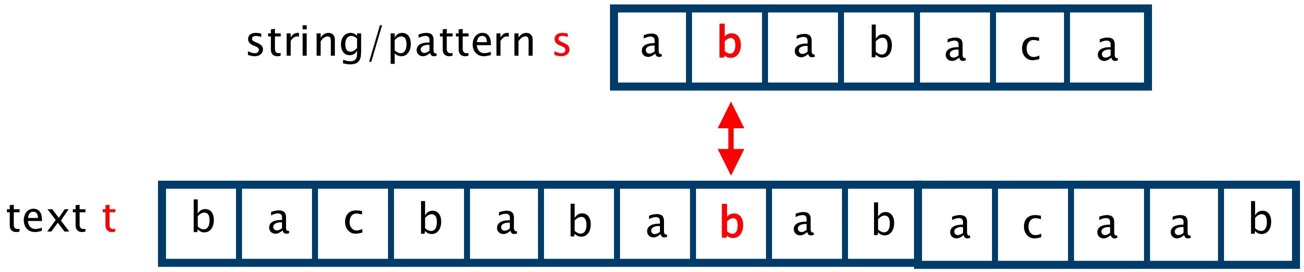


Compare characters in text and string

position in string j=0

– increment position in text and string

Brute Force – Example



Characters continue to match so

position in string j=1

- increment position in text and string

Brute Force – Example

string/pattern s	a	b	a	b	a	c	a

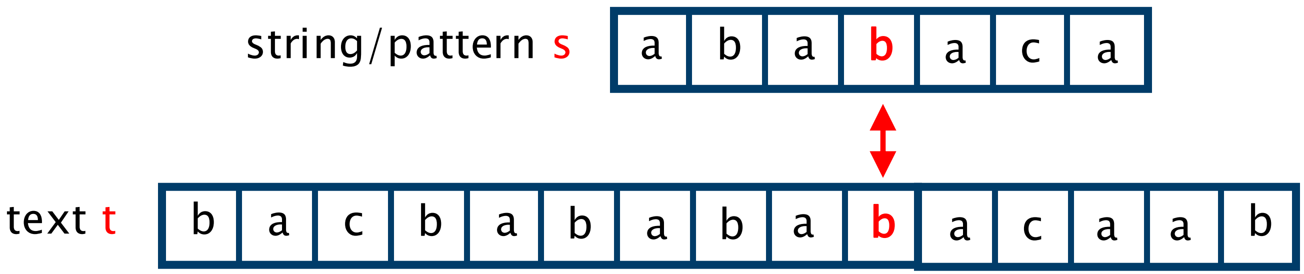
text t	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

position in string j=2

Characters continue to match so

- increment position in text and string

Brute Force – Example

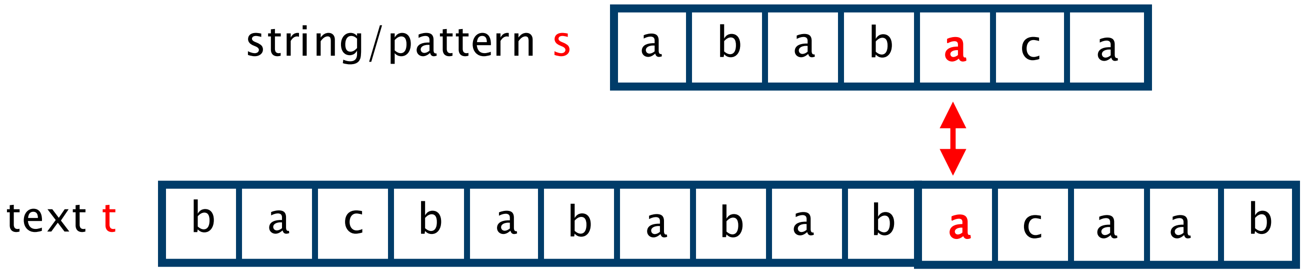


Characters continue to match so

position in string $j=3$

- increment position in text and string

Brute Force – Example

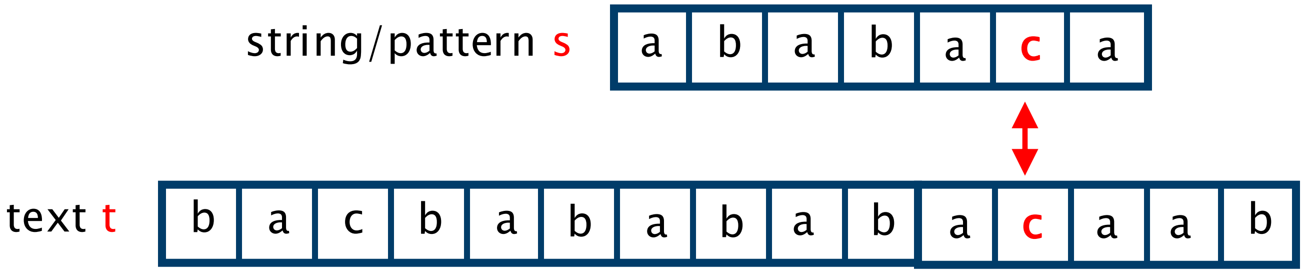


position in string $j=4$

Characters continue to match so

- increment position in text and string

Brute Force – Example

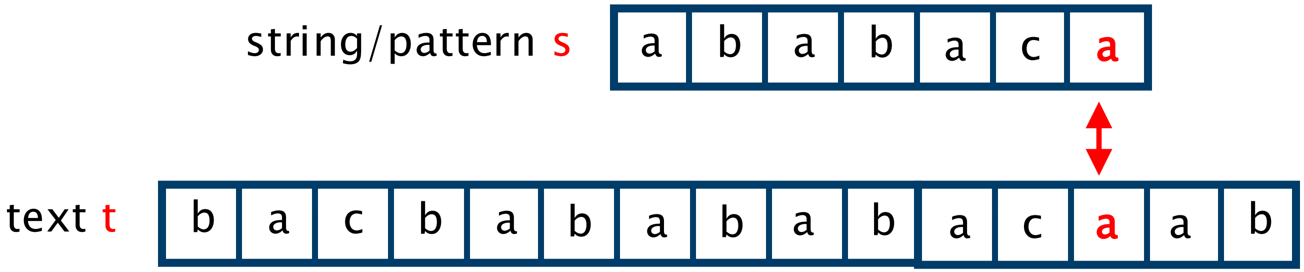


position in string j=5

Characters continue to match so

– increment position in text and string

Brute Force – Example



Characters continue to match so

- increment position in text and string

position in string j=6

Brute Force – Example

string/pattern s

a	b	a	b	a	c	a

text t

b	a	c	b	a	b	**a**	**b**	**a**	**b**	**a**	**c**	**a**	a	b

String/pattern has been found

position in string j=7

String search – Brute force algorithm

Worst case is no better than **O(mn)**

– e.g., search for $s = \underbrace{\texttt{aa … ab}}_{\text{length } m}$ in $t = \underbrace{\texttt{aa ... aaaa … ab}}_{\text{length } n}$

– m character comparisons needed at each n−(m+1) position 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?

Next lecture

Text compression

- Huffman encoding
- LZW compression/decompression

String comparison

- string distance

String/pattern search

- brute force algorithm
- KMP algorithm
- BM algorithm

Why learning how these algorithms work?

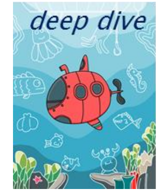
Yes, there are pre-existing functions/methods and libraries for each algorithm

Understanding how these algorithms work fosters a deeper comprehension of computational thinking and problem-solving

Understanding the "how" and "why" behind algorithms

- enables you to craft solutions that are not only functional
- but also efficient and adaptable to future technological advancements

Why learning how these algorithms work?



Yes, there are pre-existing functions/methods and libraries for each algorithm

Understanding how these algorithms work fosters a deeper comprehension of computational thinking and problem-solving

- to innovate and adapt solutions to new or evolving problems, or optimised for particular needs
- to make informed decisions about which algorithm or data structure is most appropriate for a given scenario, considering factors like time complexity, space complexity, and scalability
- to identify whether bugs/issues stem from incorrect usage, limitations of the algorithm itself, or other factors
- to create more comprehensive test cases by understanding edge cases, potential failure points, and performance bottlenecks
- classical algorithms are the bedrock upon which new technologies and frameworks are built, the underlying principles remain constant