Lecture 17,18: Introduction to String Matching Algorithms

BT 3051 – Data Structures and Algorithms for Biology

Karthik Raman
Department of Biotechnology

Indian Institute of Technology Madras

Introduction

String Matching

Problem Definition

- Given the text

  We should continually be striving to transform every art into a science: in the process, we advance the art.

  Find the string "science"

- More challenging: How many times does CATCC appear in

```
GATAAGGACCAGCGCAGTGGTAATTTAGGAACAAAGTATAGAAAT
GCTCATTTGAATTCGTAAGTTCAACTATAGTGTTCTCAGGATCGGTATCT
GAAAGGATCGATGCCTACTGAGTAATACGCGTGTGGACTGGTCGACCCTC
ATACGCTGCCACATCTCACACTTGTTAAGATGTTGGGCCTATGGG
TCAGCAACCATTGGATCGGAAACTTGAACAAGTCTGCGCCTCAGGTACGG
AACCCCCCACCTTCTGGCGTGGCACTCGGTGACTGTGTTCAGAGCCGGAA
GTCAGGGTCGGTACTCCGCCGCGGTGTCGATACGTAACACACAGACATTC
AGTCTCTTAGTCTCCAAACCATGAGGAAATGTTGCCGCCGAGGCTTTTTT
```

String Matching
Problem Definition

- Given the text

  We should continually be striving to transform every art into a science: in the process, we advance the art.

  Find the string "science"

- More challenging: How many times does `CATCC` appear in

```
GATAAGGACCAGCGCAGTGGTAATTTAGGAACACATCCAAGTATAGAAAT
GCTCATTTGAATTCGTAAGTTCAACTATAGTGTTCTCAGGATCGGTATCT
GAAAGGATCGATGCCTACTGAGTAATACGCGTGTGGACTGGTCGACCCTC
ATACGCTGCCACATCTCACACTTGTTAACATCCGATGTTGGGCCTATGGG
TCAGCAACCATTGGATCGGAAACTTGAACAAGTCTGCGCCTCAGGTACGG
AACCCCCCACCTTCTGGCGTGGCACTCGGTGACTGTGTTCAGAGCCGGAA
GTCAGGGTCGGTACTCCGCCGCGGTGTCGATACGTAACACACAGACATTC
AGTCTCTTAGTCTCCAAACCATGAGGAAATGTTGCCGCCGAGGCTTTTTT
```

String Matching
Problem Definition

- Given the text

  We should continually be striving to transform every art into a science: in the process, we advance the art.

  Find the string "science"

- More challenging: How many times does `CATCC` appear in

  ```
  GATAAGGACCAGCGCAGTGGTAATTTAGGAACACATCCAAGTATAGAAAT
  GCTCATTTGAATTCGTAAGTTCAACTATAGTGTTCTCAGGATCGGTATCT
  GAAAGGATCGATGCCTACTGAGTAATACGCGTGTGGACTGGTCGACCCTC
  ATACGCTGCCACATCTCACACTTGTTAACATCCGATGTTGGGCCTATGGG
  TCAGCAACCATTGGATCGGAAACTTGAACAAGTCTGCGCCTCAGGTACGG
  AACCCCCCACCTTCTGGCGTGGCACTCGGTGACTGTGTTCAGAGCCGGAA
  GTCAGGGTCGGTACTCCGCCGCGGTGTCGATACGTAACACACAGACATTC
  AGTCTCTTAGTCTCCAAACCATGAGGAAATGTTGCCGCCGAGGCTTTTTT
  ```

String Matching

Formal Definition

- Given a text T
  - $T \in \Sigma^*$: finite alphabet Σ
  - $|T| = n$
- and a pattern P
  - $P \in \Sigma^*$: same finite alphabet Σ
  - $|P| = m$
- Assuming both T and P can be represented using arrays
  - $T[1 \ldots n]$ and $P[1 \ldots m]$
- Pattern P occurs with shift s in T iff
  - $0 \leq s \leq n - m$
  - $T[s + i] = P[i]$ for all positions $1 \leq i \leq m$
  - Problem: Find all s

String Matching
Formal Definition

- Given a text T
  - $T \in \Sigma^*$: finite alphabet Σ
  - $|T| = n$
- and a pattern P
  - $P \in \Sigma^*$: same finite alphabet Σ
  - $|P| = m$
- Assuming both T and P can be represented using arrays
  - $T[1 \ldots n]$ and $P[1 \ldots m]$
- Pattern P occurs with shift s in T iff
  - $0 \leq s \leq n - m$
  - $T[s + i] = P[i]$ for all positions $1 \leq i \leq m$
  - Problem: Find all s

String Matching
Formal Definition

- Given a text T
  - $T \in \Sigma^*$: finite alphabet Σ
  - $|T| = n$
- and a pattern P
  - $P \in \Sigma^*$: same finite alphabet Σ
  - $|P| = m$
- Assuming both T and P can be represented using arrays
  - $T[1 \ldots n]$ and $P[1 \ldots m]$
- Pattern P occurs with shift s in T iff
  - $0 \leq s \leq n - m$
  - $T[s + i] = P[i]$ for all positions $1 \leq i \leq m$
  - Problem: Find all s

String Matching

Formal Definition

- Given a text T
  - $T \in \Sigma^*$: finite alphabet Σ
  - $|T| = n$
- and a pattern P
  - $P \in \Sigma^*$: same finite alphabet Σ
  - $|P| = m$
- Assuming both T and P can be represented using arrays
  - $T[1 \dots n]$ and $P[1 \dots m]$
- Pattern P occurs with shift s in T iff
  - $0 \leq s \leq n - m$
  - $T[s + i] = P[i]$ for all positions $1 \leq i \leq m$
  - Problem: Find all s

String Matching

Formal Definition

- Problem: find all s such that
  - $0 \leq s \leq n - m$
  - $T[s+i] = P[i]$ for $1 \leq i \leq m$

String T

A	T	C	G	A	T	C	A	G	A	T	C	G	A	A

$n = 15$

Pattern P

String Matching
Formal Definition

- Problem: find all s such that
  - $0 \leq s \leq n - m$
  - $T[s + i] = P[i]$ for $1 \leq i \leq m$

String T

A	T	C	G	A	T	C	A	G	A	T	C	G	A	A

$n = 15$

Pattern P

String Matching
Formal Definition

- Problem: find all s such that
  - $0 \leq s \leq n - m$
  - $T[s + i] = P[i]$ for $1 \leq i \leq m$

String T

A	T	C	G	A	T	C	A	G	A	T	C	G	A	A

$n = 15$

Pattern P

A	T	C

$m = 3$

$s = 0$

- Hits: [0]

String Matching

Formal Definition

- Problem: find all s such that
  - $0 \leq s \leq n - m$
  - $T[s + i] = P[i]$ for $1 \leq i \leq m$

String T

A	T	C	G	A	T	C	A	G	A	T	C	G	A	A

$n = 15$

Pattern P

A	T	C

$s = 4$

$m = 3$

- Hits: $[0, 4]$

String Matching

Formal Definition

- Problem: find all s such that
  - $0 \leq s \leq n - m$
  - $T[s + i] = P[i]$ for $1 \leq i \leq m$

String T

A	T	C	G	A	T	C	A	G	A	T	C	G	A	A

$n = 15$

Pattern P

A	T	C

$s = 9$

$m = 3$

- Hits: $[0, 4, 9]$

String Matching
Applications

- String matching is universally used in several applications
- Searching words in a document
- Searching for genes in an organism
- Spell-check
- ...

String Matching
Applications

- String matching is universally used in several applications
- Searching words in a document
- Searching for genes in an organism
- Spell-check
- ...

String Matching
Applications

- String matching is universally used in several applications
- Searching words in a document
- Searching for genes in an organism
- Spell-check
- ...

String Matching

Applications

- String matching is universally used in several applications
- Searching words in a document
- Searching for genes in an organism
- Spell-check
- ...

String Matching

Applications

- String matching is universally used in several applications
- Searching words in a document
- Searching for genes in an organism
- Spell-check
- ...

String Matching

Algorithm Design Overview (Skiena)

- Is the search pattern/text short? *Naïve matching*
- Is the search pattern/text very long? *Knuth–Morris–Pratt*
- Do we expect to find the pattern or not? *Boyer–Moore*
- Will we perform multiple queries on the same text? *Suffix trees*
- Will we search many texts using the same pattern? *Complex algorithms ...*
- What if the input contains a spelling error? *Approximate string matching*

String Matching

Algorithm Design Overview (Skiena)

- Is the search pattern/text short? *Naïve matching*
- Is the search pattern/text very long? *Knuth–Morris–Pratt*
- Do we expect to find the pattern or not? *Boyer–Moore*
- Will we perform multiple queries on the same text? *Suffix trees*
- Will we search many texts using the same pattern? *Complex algorithms ...*
- What if the input contains a spelling error? *Approximate string matching*

String Matching

Algorithm Design Overview (Skiena)

- Is the search pattern/text short? *Naïve matching*
- Is the search pattern/text very long? *Knuth–Morris–Pratt*
- Do we expect to find the pattern or not? *Boyer–Moore*
- Will we perform multiple queries on the same text? *Suffix trees*
- Will we search many texts using the same pattern? *Complex algorithms ...*
- What if the input contains a spelling error? *Approximate string matching*

String Matching

Algorithm Design Overview (Skiena)

- Is the search pattern/text short? *Naïve matching*
- Is the search pattern/text very long? *Knuth–Morris–Pratt*
- Do we expect to find the pattern or not? *Boyer–Moore*
- Will we perform multiple queries on the same text? *Suffix trees*
- Will we search many texts using the same pattern? *Complex algorithms ...*
- What if the input contains a spelling error? *Approximate string matching*

String Matching

Algorithm Design Overview (Skiena)

- Is the search pattern/text short? *Naïve matching*
- Is the search pattern/text very long? *Knuth–Morris–Pratt*
- Do we expect to find the pattern or not? *Boyer–Moore*
- Will we perform multiple queries on the same text? *Suffix trees*
- Will we search many texts using the same pattern? *Complex algorithms ...*
- What if the input contains a spelling error? *Approximate string matching*

String Matching

Algorithm Design Overview (Skiena)

- Is the search pattern/text short? *Naïve matching*
- Is the search pattern/text very long? *Knuth–Morris–Pratt*
- Do we expect to find the pattern or not? *Boyer–Moore*
- Will we perform multiple queries on the same text? *Suffix trees*
- Will we search many texts using the same pattern? *Complex algorithms ...*
- What if the input contains a spelling error? *Approximate string matching*

String Matching

Algorithm Design Overview (Skiena)

- Is the search pattern/text short? *Naïve matching*
- Is the search pattern/text very long? *Knuth–Morris–Pratt*
- Do we expect to find the pattern or not? *Boyer–Moore*
- Will we perform multiple queries on the same text? *Suffix trees*
- Will we search many texts using the same pattern? *Complex algorithms …*
- What if the input contains a spelling error? *Approximate string matching*

String Matching

Algorithm Design Overview (Skiena)

- Is the search pattern/text short? *Naïve matching*
- Is the search pattern/text very long? *Knuth–Morris–Pratt*
- Do we expect to find the pattern or not? *Boyer–Moore*
- Will we perform multiple queries on the same text? *Suffix trees*
- Will we search many texts using the same pattern? *Complex algorithms …*
- What if the input contains a spelling error? *Approximate string matching*

String Matching

Algorithm Design Overview (Skiena)

- Is the search pattern/text short? *Naïve matching*
- Is the search pattern/text very long? *Knuth–Morris–Pratt*
- Do we expect to find the pattern or not? *Boyer–Moore*
- Will we perform multiple queries on the same text? *Suffix trees*
- Will we search many texts using the same pattern? *Complex algorithms ...*
- What if the input contains a spelling error? *Approximate string matching*

String Matching

Algorithm Design Overview (Skiena)

- Is the search pattern/text short? *Naïve matching*
- Is the search pattern/text very long? *Knuth–Morris–Pratt*
- Do we expect to find the pattern or not? *Boyer–Moore*
- Will we perform multiple queries on the same text? *Suffix trees*
- Will we search many texts using the same pattern? *Complex algorithms ...*
- What if the input contains a spelling error? *Approximate string matching*

String Matching

Algorithm Design Overview (Skiena)

- Is the search pattern/text short? *Naïve matching*
- Is the search pattern/text very long? *Knuth–Morris–Pratt*
- Do we expect to find the pattern or not? *Boyer–Moore*
- Will we perform multiple queries on the same text? *Suffix trees*
- Will we search many texts using the same pattern? *Complex algorithms ...*
- What if the input contains a spelling error? *Approximate string matching*

String Matching

Algorithm Design Overview (Skiena)

- Is the search pattern/text short? *Naïve matching*
- Is the search pattern/text very long? *Knuth–Morris–Pratt*
- Do we expect to find the pattern or not? *Boyer–Moore*
- Will we perform multiple queries on the same text? *Suffix trees*
- Will we search many texts using the same pattern? *Complex algorithms ...*
- What if the input contains a spelling error? *Approximate string matching*

String Matching

Algorithm Design Overview (Skiena)

- Is the search pattern/text short? *Naïve matching*
- Is the search pattern/text very long? *Knuth–Morris–Pratt*
- Do we expect to find the pattern or not? *Boyer–Moore*
- Will we perform multiple queries on the same text? *Suffix trees*
- Will we search many texts using the same pattern? *Complex algorithms ...*
- What if the input contains a spelling error? *Approximate string matching*

Algorithms

Naïve algorithm

```
def NaiveStringMatcher(Text,Pattern):
m = len(Pattern)
n = len(Text)

matches = []
for s in range(n-m):
if Text[s:s+m]==Pattern:
matches.append(s)

return matches

if __name__ == '__main__':
print(NaiveStringMatcher('ATCGATCAGATCGAA','ATC'
                         ))
```

What is the complexity?
Can we do better?

Naïve algorithm

```
def NaiveStringMatcher(Text,Pattern):
m = len(Pattern)
n = len(Text)

matches = []
for s in range(n-m):
if Text[s:s+m]==Pattern:
matches.append(s)

return matches

if __name__ == '__main__':
print(NaiveStringMatcher('ATCGATCAGATCGAA','ATC'
                         ))
```

What is the complexity?

Can we do better?

Naïve algorithm

```
def NaiveStringMatcher(Text,Pattern):
m = len(Pattern)
n = len(Text)

matches = []
for s in range(n-m):
if Text[s:s+m]==Pattern:
matches.append(s)

return matches

if __name__ == '__main__':
print(NaiveStringMatcher('ATCGATCAGATCGAA','ATC'
                    ))
```

What is the complexity?
Can we do better?

Improvement Strategy

- Observation

T														
	A	T	C	A	A	T	A	A	T	A	T	A	C	A

P

- What now?

Improvement Strategy

- Observation

T	A	T	C	A	A	T	A	A	T	A	T	A	C	A

P	A	T	A

- What now?
  - the naive algorithm goes *back to the second position in T and starts from the beginning of P*
  - can't we simply move along through T?

Improvement Strategy

- Observation

T	A	T	C	A	A	T	A	A	T	A	T	A	C	A
	=													
P	A	T	A											

- What now?

Improvement Strategy

- Observation

T	A	T	C	A	A	T	A	A	T	A	T	A	C	A
	=	=												
P	A	T	A											

- What now?

Improvement Strategy

- Observation

$$T \; \boxed{A}\boxed{T}\boxed{C}\boxed{A}\boxed{A}\boxed{T}\boxed{A}\boxed{A}\boxed{T}\boxed{A}\boxed{T}\boxed{A}\boxed{C}\boxed{A}$$

$$= \;\; = \;\; \neq$$

$$P \; \boxed{A}\boxed{T}\boxed{A}$$

- What now?
  - the naïve algorithm *goes back to the second position in T* and *starts from the beginning of P*
  - can't we simply move along through T?

Improvement Strategy

- Observation

T	A	T	C	A	A	T	A	A	T	A	T	A	C	A
	$=$	$=$	$\neq$											
P	A	T	A											

- What now?
  - the naïve algorithm *goes back to the second position in T* and *starts from the beginning of P*
  - can't we simply move along through T?

Improvement Strategy

- Observation

T	A	T	C	A	A	T	A	A	T	A	T	A	C	A
	$=$	$=$	$\neq$											
P	A	T	A											

- What now?
  - the naïve algorithm *goes back to the second position in T* and *starts from the beginning of P*
  - can't we simply move along through T?

Improvement Strategy

- Observation

T	A	T	C	A	A	T	A	A	T	A	T	A	C	A
	$=$	$=$	$\neq$											
P	A	T	A											

- What now?
  - the naïve algorithm *goes back to the second position in T* and *starts from the beginning of P*
  - can't we simply move along through T?

Careless String Matcher

```
def CarelessStringMatcher(Text, Pattern):
n = len(Text)
m = len(Pattern)
q = s = 0
while s < n:
if Text[s] == Pattern[q]:
q += 1 #increase match length
if q == m: #matched `m' characters
print (s - m + 1)
q = 0 #reset match length
else:
q = 0 #found a mismatch
s += 1 #move further in the text
```

Is there a bug?

Careless String Matcher

```
def CarelessStringMatcher(Text, Pattern):
n = len(Text)
m = len(Pattern)
q = s = 0
while s < n:
if Text[s] == Pattern[q]:
q += 1 #increase match length
if q == m: #matched `m' characters
print (s - m + 1)
q = 0 #reset match length
else:
q = 0 #found a mismatch
s += 1 #move further in the text
```

Is there a bug?

Boyer-Moore Algorithm

Smart Heuristics

- Main idea: improve running time of brute-force algorithm by adding two potentially time-saving heuristics
- Roughly stated, these heuristics are:
  - *Looking-Glass Heuristic*: When testing a possible placement of P against T, begin comparisons from the end of P and move backward to the front of P
  - *Character-Jump Heuristic*: During the testing of a possible placement of P within T, a mismatch of text character $T[i] = c$ with the corresponding pattern character $P[k]$ is handled as follows:

Boyer-Moore Algorithm

Smart Heuristics

- Main idea: improve running time of brute-force algorithm by adding two potentially time-saving heuristics
- Roughly stated, these heuristics are:
  - *Looking-Glass Heuristic*: When testing a possible placement of P against T, begin comparisons from the end of P and move backward to the front of P
  - *Character-Jump Heuristic*: During the testing of a possible placement of P within T, a mismatch of text character $T[i] = c$ with the corresponding pattern character $P[\mathrm{k}]$ is handled as follows:
    - If c is not contained anywhere in P, then shift P completely past $T[i]$ (for it cannot match any character in P)
    - Otherwise, shift P until an occurrence of character c in P gets aligned with $T[i]$

Boyer-Moore Algorithm

Smart Heuristics

- Main idea: improve running time of brute-force algorithm by adding two potentially time-saving heuristics
- Roughly stated, these heuristics are:
  - *Looking-Glass Heuristic*: When testing a possible placement of P against T, begin comparisons from the end of P and move backward to the front of P
  - *Character-Jump Heuristic*: During the testing of a possible placement of P within T, a mismatch of text character $T[i] = c$ with the corresponding pattern character $P[\mathrm{k}]$ is handled as follows:
    - If c is not contained anywhere in P, then shift P completely past $T[i]$ (for it cannot match any character in P)
    - Otherwise, shift P until an occurrence of character c in P gets aligned with $T[i]$

Boyer-Moore Algorithm

Smart Heuristics

- Main idea: improve running time of brute-force algorithm by adding two potentially time-saving heuristics
- Roughly stated, these heuristics are:
  - *Looking-Glass Heuristic*: When testing a possible placement of P against T, begin comparisons from the end of P and move backward to the front of P
  - *Character-Jump Heuristic*: During the testing of a possible placement of P within T, a mismatch of text character $T[i] = c$ with the corresponding pattern character $P[\mathrm{k}]$ is handled as follows:
    - If c is not contained anywhere in P, then shift P completely past $T[i]$ (for it cannot match any character in P)
    - Otherwise, shift P until an occurrence of character c in P gets aligned with $T[i]$

Boyer-Moore Algorithm
Smart Heuristics

- Main idea: improve running time of brute-force algorithm by adding two potentially time-saving heuristics
- Roughly stated, these heuristics are:
  - *Looking-Glass Heuristic*: When testing a possible placement of P against T, begin comparisons from the end of P and move backward to the front of P
  - *Character-Jump Heuristic*: During the testing of a possible placement of P within T, a mismatch of text character $T[i] = c$ with the corresponding pattern character $P[k]$ is handled as follows:
    - If c is not contained anywhere in P, then shift P completely past $T[i]$ (for it cannot match any character in P)
    - Otherwise, shift P until an occurrence of character c in P gets aligned with $T[i]$

Boyer-Moore Algorithm

Smart Heuristics

- Main idea: improve running time of brute-force algorithm by adding two potentially time-saving heuristics
- Roughly stated, these heuristics are:
  - *Looking-Glass Heuristic*: When testing a possible placement of P against T, begin comparisons from the end of P and move backward to the front of P
  - *Character-Jump Heuristic*: During the testing of a possible placement of P within T, a mismatch of text character $T[i] = c$ with the corresponding pattern character $P[\mathrm{k}]$ is handled as follows:
    - If c is not contained anywhere in P, then shift P completely past $T[i]$ (for it cannot match any character in P)
    - Otherwise, shift P until an occurrence of character c in P gets aligned with $T[i]$

Boyer–Moore Algorithm

h	e	r	e		i	s		a		s	i	m	p	l	e		e	x	a	m	p	l	e

- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

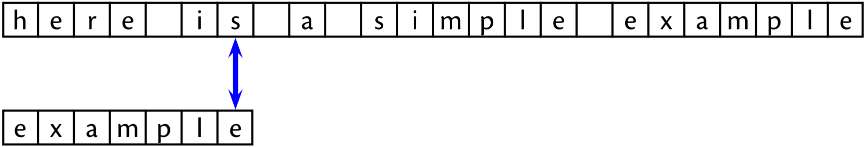
Boyer–Moore Algorithm

h	e	r	e		i	s		a		s	i	m	p	l	e		e	x	a	m	p	l	e

e	x	a	m	p	l	e

- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
    - so that the pattern skips α, if α is not in the pattern

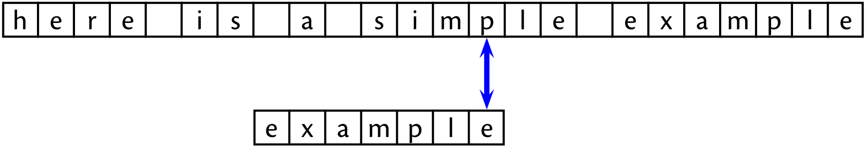
Boyer–Moore Algorithm

h	e	r	e		i	s		a		s	i	m	p	l	e		e	x	a	m	p	l	e

e	x	a	m	p	l	e

- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

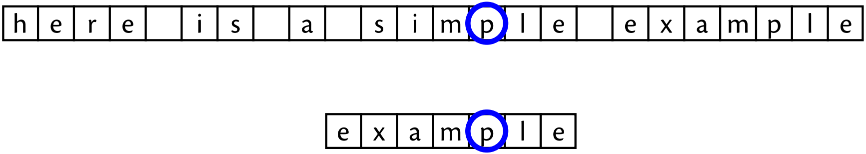
Boyer–Moore Algorithm

h	e	r	e		i	s		a		s	i	m	p	l	e		e	x	a	m	p	l	e

e	x	a	m	p	l	e

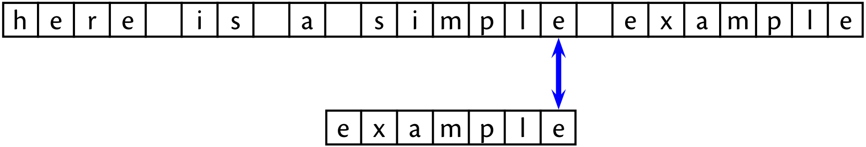
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



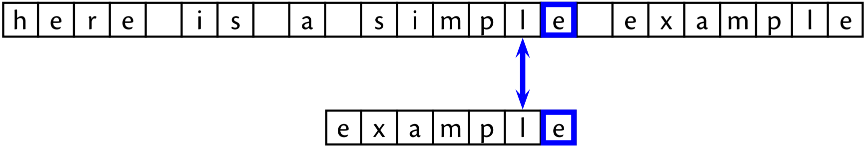
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



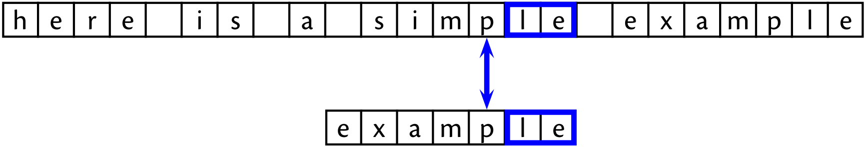
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



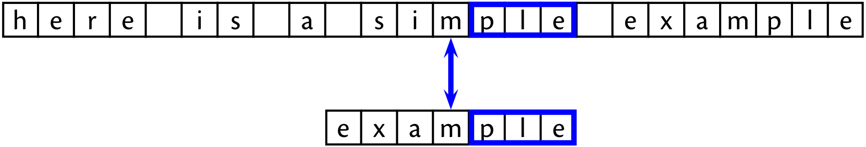
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



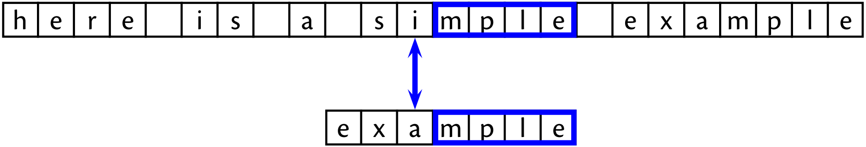
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm

h	e	r	e		i	s		a		s	i	m	p	l	e		e	x	a	m	p	l	e

e	x	a	m	p	l	e

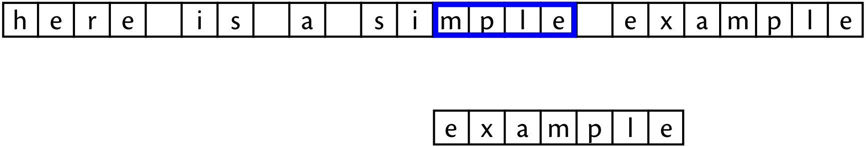
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm

h	e	r	e		i	s		a		s	i	m	p	l	e		e	x	a	m	p	l	e
												e	x	a	m	p	l	e					

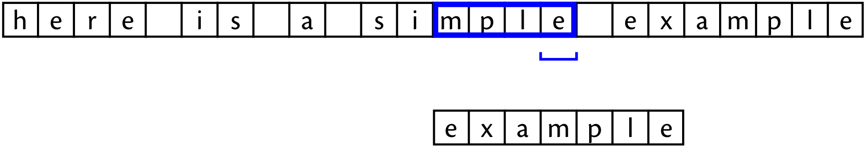
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



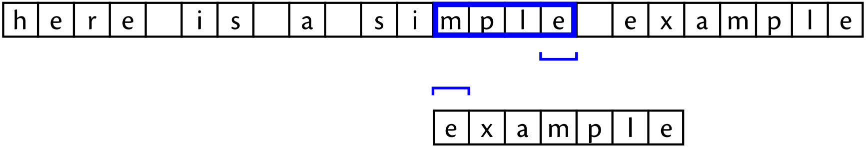
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



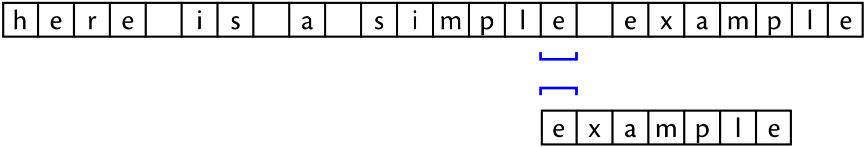
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



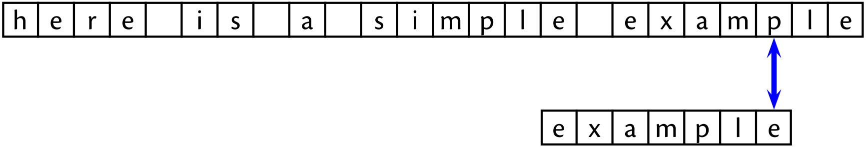
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



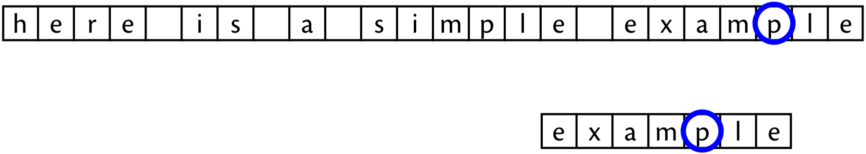
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



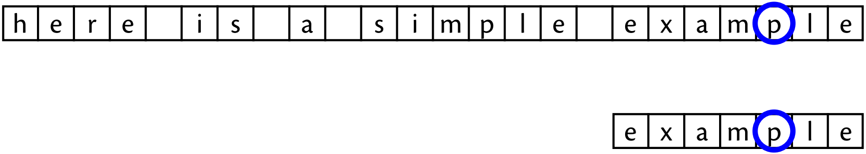
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



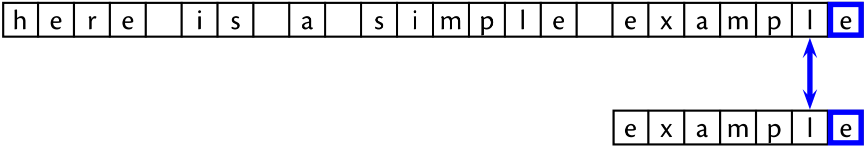
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm

h	e	r	e		i	s		a		s	i	m	p	l	e		e	x	a	m	p	l	e
																	e	x	a	m	p	l	e

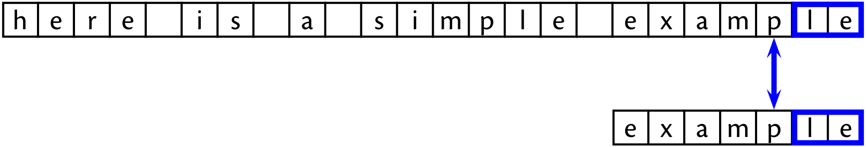
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



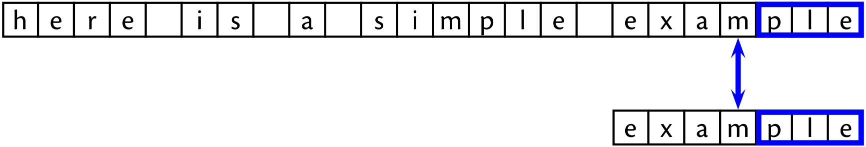
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



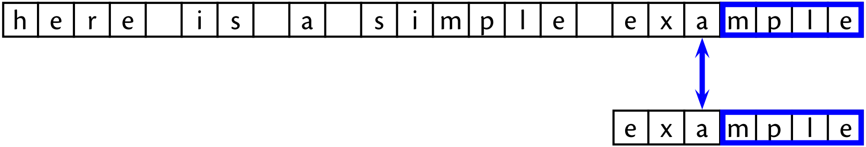
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



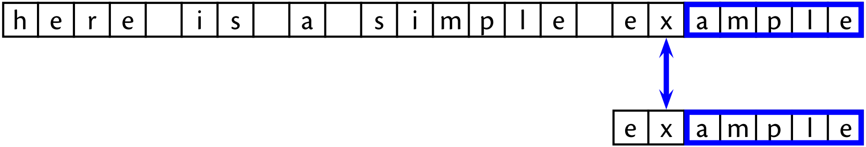
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



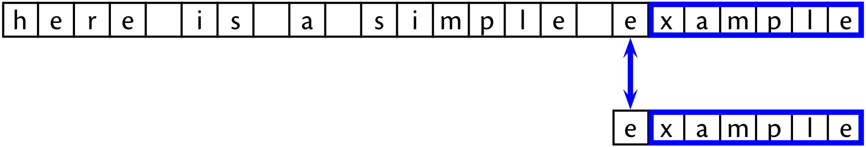
- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm



- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm

h	e	r	e		i	s		a		s	i	m	p	l	e		e	x	a	m	p	l	e
																	e	x	a	m	p	l	e

- We match the pattern right-to-left
- If we find a bad character α in the text, we can shift
  - so that the pattern skips α, if α is not in the pattern
  - so that the pattern lines up with the rightmost occurrence of α in the pattern, if the pattern contains α
  - so that a pattern prefix lines up with a suffix of the current partial (or complete) match

Boyer–Moore Algorithm

- Boyer–Moore includes a pre-processing phase: $O(m)$
- The search phase is $O(nm)$
- The search phase can be as low as $O(n/m)$ in common cases
- *In practice, Boyer-Moore is the fastest string-matching algorithm for most applications*

Boyer–Moore Algorithm

- Boyer–Moore includes a pre-processing phase: $O(m)$
- The search phase is $O(nm)$
- The search phase can be as low as $O(n/m)$ in common cases
- *In practice, Boyer-Moore is the fastest string-matching algorithm for most applications*

Boyer–Moore Algorithm

- Boyer–Moore includes a pre-processing phase: $O(m)$
- The search phase is $O(nm)$
- The search phase can be as low as $O(n/m)$ in common cases
- *In practice, Boyer-Moore is the fastest string-matching algorithm for most applications*

Boyer–Moore Algorithm

- Boyer–Moore includes a pre-processing phase: $O(m)$
- The search phase is $O(nm)$
- The search phase can be as low as $O(n/m)$ in common cases
- *In practice, Boyer-Moore is the fastest string-matching algorithm for most applications*

Knuth–Morris–Pratt Algorithm

Knuth–Morris–Pratt Algorithm

- What is a major inefficiency of Brute-Force/Boyer-Moore (in the worst cases)?
- For a certain alignment of the pattern, if we find several matching characters but then detect a mismatch, we ignore all the information gained by the successful comparisons after restarting with the next incremental placement of the pattern!
- *What can we do instead?*

Knuth–Morris–Pratt Algorithm

- What is a major inefficiency of Brute-Force/Boyer-Moore (in the worst cases)?
- For a certain alignment of the pattern, if we find several matching characters but then detect a mismatch, we ignore all the information gained by the successful comparisons after restarting with the next incremental placement of the pattern!
- *What can we do instead?*

Knuth–Morris–Pratt Algorithm

- What is a major inefficiency of Brute-Force/Boyer-Moore (in the worst cases)?
- For a certain alignment of the pattern, if we find several matching characters but then detect a mismatch, we ignore all the information gained by the successful comparisons after restarting with the next incremental placement of the pattern!
- *What can we do instead?*