

6 String Matching – What's behind Ctrl+F?

17 November 2025

Prof. Dr. Sebastian Wild

Learning Outcomes

Unit 6: *String Matching*

1. Know and use typical notions for *strings* (substring, prefix, suffix, etc.).
2. Understand principles and implementation of the *KMP*, *BM*, and *RK* algorithms.
3. Know the *performance characteristics* of the KMP, BM, and RK algorithms.
4. Be able to solve simple *stringology problems* using the *KMP failure function*.

Outline

6 String Matching

6.1 String Notation

6.2 Brute Force

6.3 String Matching with Finite Automata

6.4 Constructing String Matching Automata

6.5 The Knuth-Morris-Pratt algorithm

6.6 Beyond Optimal? The Boyer-Moore Algorithm

6.7 The Rabin-Karp Algorithm

6.1 String Notation

Ubiquitous strings

string = sequence of characters

► universal data type for . . . everything!

- natural language texts
- programs (source code)
- websites
- XML documents
- DNA sequences
- bitstrings
- . . . a computer's memory ↗ ultimately any data is a string

↗ many different tasks and algorithms

Ubiquitous strings

string = sequence of characters

- ▶ universal data type for . . . everything!
 - ▶ natural language texts
 - ▶ programs (source code)
 - ▶ websites
 - ▶ XML documents
 - ▶ DNA sequences
 - ▶ bitstrings
 - ▶ . . . a computer's memory ↗ ultimately any data is a string
- ↗ many different tasks and algorithms
- ▶ This unit: finding (exact) **occurrences of a pattern text**.
 - ▶ Ctrl+F
 - ▶ grep
 - ▶ computer forensics (e. g. find signature of file on disk)
 - ▶ virus scanner
- ▶ basis for many advanced applications

Notation

$$\Sigma = \{0..9\}$$

- *alphabet* Σ : finite set of allowed **characters**; $\sigma = |\Sigma|$ “a string over alphabet Σ ”

- letters (Latin, Greek, Arabic, Cyrillic, Asian scripts, ...)

- “what you can type on a keyboard”, Unicode characters

- $\{0,1\}$; nucleotides $\{A, C, G, T\}$; ...

comprehensive standard character set
including emoji and all known symbols

Notation

- *alphabet* Σ : finite set of allowed **characters**; $\sigma = |\Sigma|$ “*a string over alphabet Σ* ”
 - letters (Latin, Greek, Arabic, Cyrillic, Asian scripts, ...)
 - “what you can type on a keyboard”, Unicode characters
 - $\{0,1\}$; nucleotides $\{A, C, G, T\}$; ... comprehensive standard character set including emoji and all known symbols
- $\Sigma^n = \Sigma \times \dots \times \Sigma$: strings of **length $n \in \mathbb{N}_0$** (n -tuples)
- $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$: set of **all** (finite) strings over Σ
- $\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n$: set of **all** (finite) **nonempty** strings over Σ
- $\varepsilon \in \Sigma^0$: the *empty* string (same for all alphabets)

Notation

- **alphabet** Σ : finite set of allowed **characters**; $\sigma = |\Sigma|$ “a string over alphabet Σ ”
 - letters (Latin, Greek, Arabic, Cyrillic, Asian scripts, ...)
 - “what you can type on a keyboard”, Unicode characters
 - $\{0,1\}$; nucleotides $\{A, C, G, T\}$; ... comprehensive standard character set including emoji and all known symbols
- $\Sigma^n = \Sigma \times \dots \times \Sigma$: strings of **length** $n \in \mathbb{N}_0$ (n -tuples)
- $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$: set of **all** (finite) strings over Σ
- $\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n$: set of **all** (finite) **nonempty** strings over Σ
- $\varepsilon \in \Sigma^0$: the *empty* string (same for all alphabets)
- for $S \in \Sigma^n$, write $S[i]$ (other sources: S_i) for **i**th character ($0 \leq i < n$) zero-based (like arrays)!
- for $S, T \in \Sigma^*$, write $ST = S \cdot T$ for **concatenation** of S and T
- for $S \in \Sigma^n$, write $S[i..j]$ or $S_{i,j}$ for the **substring** $S[i] \cdot S[i+1] \cdots S[j]$ ($0 \leq i \leq j < n$)
 - $S[i..j] = S[i..j-1]$ (endpoint exclusive) \rightsquigarrow $S = S[0..n]$
 - $S[0..j]$ is a **prefix** of S ; $S[i..n-1]$ is a **suffix** of S

Clicker Question



True or false: $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$

A

True

B

False



→ *sli.do/cs566*

Clicker Question



True or false: $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$

A True ✓

B False



→ *sli.do/cs566*

String matching – Definition

Search for a string (pattern) in a large body of text

► **Input:**

- $T \in \Sigma^n$: The *text* (haystack) being searched within
- $P \in \Sigma^m$: The *pattern* (needle) being searched for; typically $n \gg m$

► **Output:**

- the *first occurrence (match)* of P in T : $\min\{i \in [0..n-m) : \underbrace{T[i..i+m)}_P = P\}$
- or *NO_MATCH* if there is no such i (“ P does not occur in T ”)

► Variant: Find **all** occurrences of P in T .

~~ Can do that iteratively (update T to $T[i+1..n]$ after match at i)

► **Example:**

- $T = \text{"Where is he?"}$
- $P_1 = \text{"he"}$ ~~ $i = 1$
- $P_2 = \text{"who"}$ ~~ *NO_MATCH*

► string matching is implemented in Java in `String.indexOf`, in Python as `str.find`

6.2 Brute Force

Abstract idea of algorithms

String matching algorithms typically use *guesses* and *checks*:

- ▶ A **guess** is a position i such that P might start at $T[i]$.
Possible guesses (initially) are $0 \leq i \leq n - m$.
- ▶ A **check** of a guess is a comparison of $T[i + j]$ to $P[j]$.

Abstract idea of algorithms

String matching algorithms typically use *guesses* and *checks*:

- ▶ A **guess** is a position i such that P might start at $T[i]$.
Possible guesses (initially) are $0 \leq i \leq n - m$.
- ▶ A **check** of a guess is a comparison of $T[i + j]$ to $P[j]$.
- ▶ Note: need all m checks to verify a single *correct* guess i ,
but it may take (many) fewer checks to recognize an *incorrect* guess.
- ▶ Cost measure: #character comparisons
~~> #checks $\leq n \cdot m$ (number of possible checks)

Brute-force method

```
1 procedure bruteForceSM( $T[0..n]$ ,  $P[0..m]$ ):  
2   for  $i := 0, \dots, n - m - 1$  do  
3     for  $j := 0, \dots, m - 1$  do  
4       if  $T[i + j] \neq P[j]$  then break inner loop  
5       if  $j == m$  then return  $i$   
6   return NO_MATCH
```

- ▶ try all guesses i
- ▶ check each guess (left to right); stop early on mismatch
- ▶ essentially the implementation in Java! (`String.indexOf`)

▶ Example:

$T = \text{abbbababbbab}$

$P = \text{abba}$

a	b	b	b	a	b	a	b	b	a	b

Brute-force method

```
1 procedure bruteForceSM( $T[0..n]$ ,  $P[0..m]$ ):  
2   for  $i := 0, \dots, n - m - 1$  do  
3     for  $j := 0, \dots, m - 1$  do  
4       if  $T[i + j] \neq P[j]$  then break inner loop  
5       if  $j == m$  then return  $i$   
6   return NO_MATCH
```

- ▶ try all guesses i
- ▶ check each guess (left to right); stop early on mismatch
- ▶ essentially the implementation in Java! (`String.indexOf`)

▶ Example:

$T = abbbabababb$

$P = abba$

~ 15 char cmps
(vs $n \cdot m = 44$)
not too bad!

a	b	b	b	a	b	a	b	b	a	b
		a								
			a							
				a	b	b				
					a					
						a	b	b	a	

Brute-force method – Discussion

👍 Brute-force method can be good enough

- ▶ typically works well for natural language text
- ▶ also for random strings

👎 but: can be as bad as it gets!

a	a	a	a	a	a	a	a	a	a	a
a	a	a	b							
	a	a	a	b						
		a	a	a	b					
			a	a	a	b				
				a	a	a	b			
					a	a	a	b		
						a	a	a	b	

- ▶ Worst possible input: $P = a^{m-1}b$,
 $T = a^n$
- ▶ Worst-case performance: $(n - m + 1) \cdot m$
~~ for $m \leq n/2$ that is $\Theta(mn)$

Brute-force method – Discussion

👍 Brute-force method can be good enough

- ▶ typically works well for natural language text
- ▶ also for random strings

👎 but: can be as bad as it gets!

a	a	a	a	a	a	a	a	a	a	a
a	a	a	b							
	a	a	a	b						
		a	a	a	b					
			a	a	a	b				
				a	a	a	b			
					a	a	a	b		
						a	a	a	b	

- ▶ Worst possible input: $P = a^{m-1}b$,
 $T = a^n$
- ▶ Worst-case performance: $(n - m + 1) \cdot m$
~~ for $m \leq n/2$ that is $\Theta(mn)$

- ▶ Bad input: lots of self-similarity in T ! ~~ can we exploit that?
- ▶ brute force does ‘obviously’ stupid repetitive comparisons ~~ can we avoid that?

Roadmap

- ▶ **Approach 1** (this week): Use *preprocessing* on the **pattern P** to eliminate guesses (avoid ‘obvious’ redundant work)
 - ▶ Deterministic finite automata (**DFA**)
 - ▶ **Knuth-Morris-Pratt** algorithm
 - ▶ **Boyer-Moore** algorithm
 - ▶ **Rabin-Karp** algorithm
- ▶ **Approach 2** (\rightsquigarrow Unit 13): Do *preprocessing* on the **text T**
Can find matches in time *independent of text size(!)*
 - ▶ **inverted indices**
 - ▶ **Suffix trees**
 - ▶ **Suffix arrays**

6.3 String Matching with Finite Automata

Clicker Question



Do you know what regular expressions, NFAs and DFAs are, and how to convert between them?

- A** Never heard of this; are these new emoji?
- B** Heard the terms, but don't remember conversion methods.
- C** Had that in my undergrad course (memories fading a bit).
- D** Sure, I could do that blindfolded!



→ *sli.do/cs566*

Theoretical Computer Science to the rescue!

- ▶ string matching = deciding whether $T \in \Sigma^* \cdot P \cdot \Sigma^*$
- ▶ $\Sigma^* \cdot P \cdot \Sigma^*$ is *regular* formal language
 - ~~ \exists deterministic finite automaton (DFA) to recognize $\Sigma^* \cdot P \cdot \Sigma^*$
 - ~~ can check for occurrence of P in $|T| = n$ steps!

Theoretical Computer Science to the rescue!

- ▶ string matching = deciding whether $T \in \Sigma^* \cdot P \cdot \Sigma^*$
- ▶ $\Sigma^* \cdot P \cdot \Sigma^*$ is *regular* formal language
 - ~~ \exists deterministic finite automaton (DFA) to recognize $\Sigma^* \cdot P \cdot \Sigma^*$
 - ~~ can check for occurrence of P in $|T| = n$ steps!



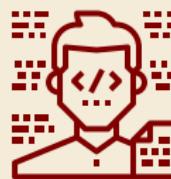
Job done!

Theoretical Computer Science to the rescue!

- ▶ string matching = deciding whether $T \in \Sigma^* \cdot P \cdot \Sigma^*$
- ▶ $\Sigma^* \cdot P \cdot \Sigma^*$ is *regular* formal language
 - ~~ \exists deterministic finite automaton (DFA) to recognize $\Sigma^* \cdot P \cdot \Sigma^*$
 - ~~ can check for occurrence of P in $|T| = n$ steps!



Job done!



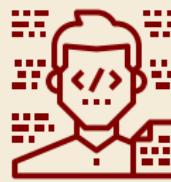
WTF!?

Theoretical Computer Science to the rescue!

- ▶ string matching = deciding whether $T \in \Sigma^* \cdot P \cdot \Sigma^*$
- ▶ $\Sigma^* \cdot P \cdot \Sigma^*$ is *regular* formal language
 - ~~ \exists deterministic finite automaton (DFA) to recognize $\Sigma^* \cdot P \cdot \Sigma^*$
 - ~~ can check for occurrence of P in $|T| = n$ steps!



Job done!



WTF!?

We are not quite done yet.

- ▶ (Problem 0: programmer might not know automata and formal languages . . .)
- ▶ Problem 1: existence alone does not give an algorithm!
- ▶ Problem 2: automaton could be very big!

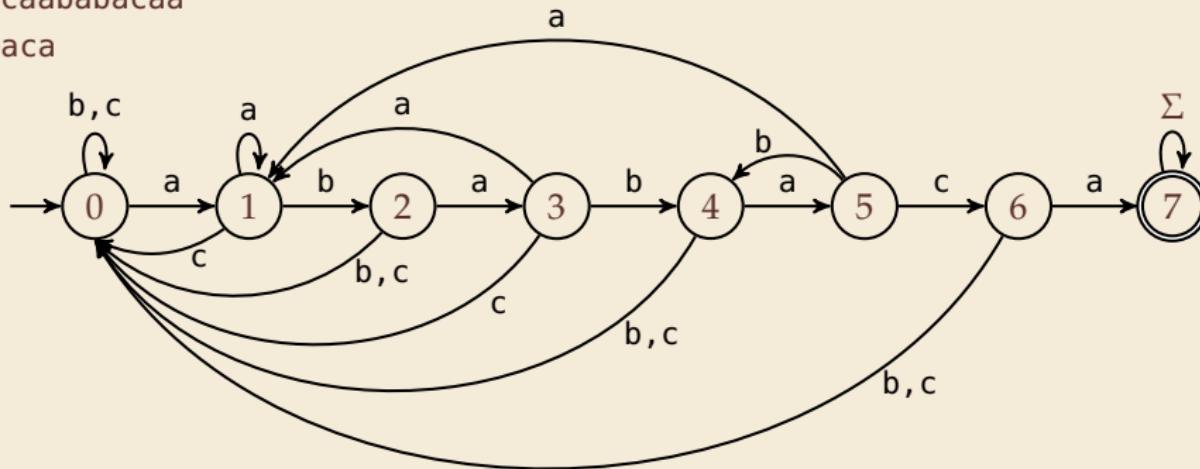
String matching with DFA

- ▶ Assume first, we already have a deterministic automaton
- ▶ How does string matching work?

Example:

$T = \text{aabacaababacaa}$

$P = \text{ababaca}$



text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	/	1	2	3	0	1	1	2	3	4	5	6	7	7

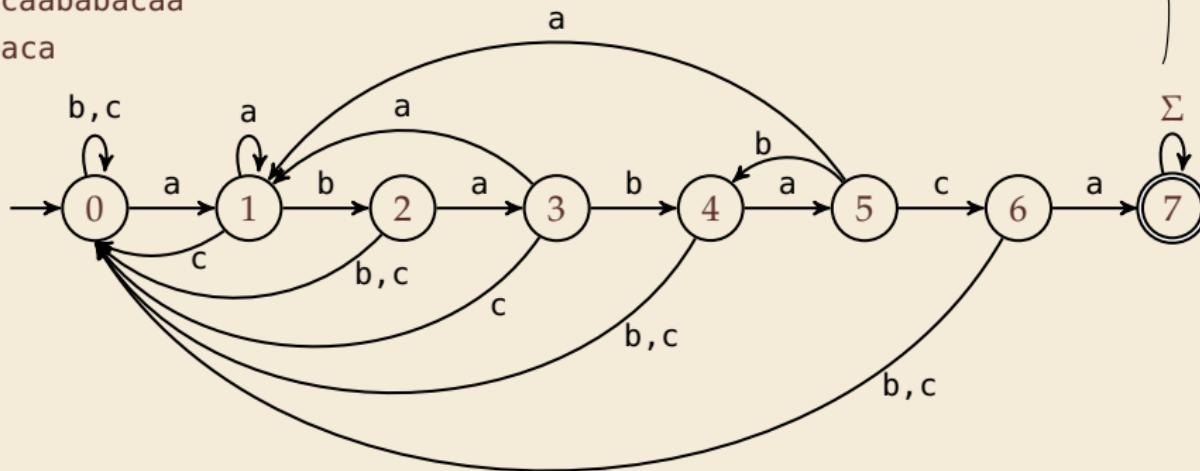
String matching with DFA

- ▶ Assume first, we already have a deterministic automaton
- ▶ How does string matching work?

Example:

$T = \text{aabacaababacaa}$

$P = \text{ababaca}$



text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3	0	1	1	2	3	4	5	6	7	7

$\delta(q, a)$	c	q	a	b	c
q		0	1	0	0
0		1	1	2	0
1		1	1	2	0

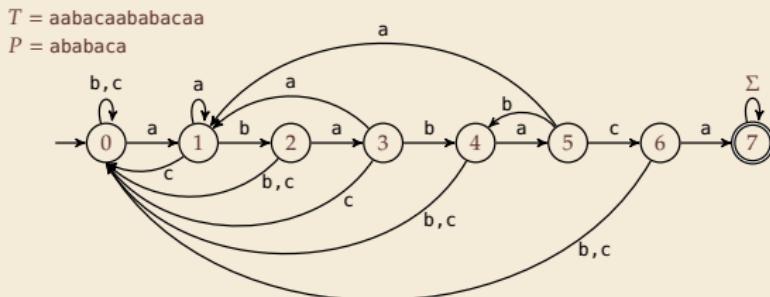
String matching DFA – Intuition

Why does this work?

- ## ► Main insight:

State q means:

*"we have seen $P[0..q)$ until here
(but not any longer prefix of P)"*

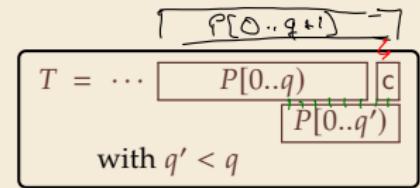


$$c = P\{q\} \approx q + 1$$

text:	a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3	0	1	1	2	3	4	5	6	7

- If the next text character *c* does not match, we know:

- (i) text seen so far ends with $P[0\dots q] \cdot c$
 - (ii) $P[0\dots q] \cdot c$ is not a prefix of P
 - (iii) without reading c , $P[0\dots q]$ was the longest prefix of P that ends here.



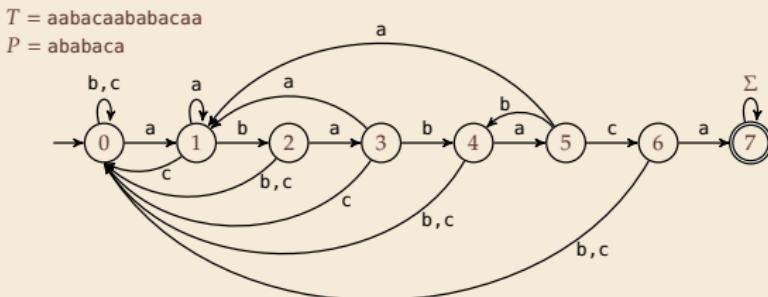
String matching DFA – Intuition

Why does this work?

- ▶ Main insight:

State q means:

*“we have seen $P[0..q]$ until here
(but not any longer prefix of P)”*



text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3	0	1	1	2	3	4	5	6	7	7

- ▶ If the next text character c does not match, we know:

- text seen so far ends with $P[0..q] \cdot c$
- $P[0..q] \cdot c$ is not a prefix of P
- without reading c , $P[0..q]$ was the *longest* prefix of P that ends here.

$T = \dots \boxed{P[0..q]} \boxed{c} \boxed{P[0..q']}$
with $q' < q$

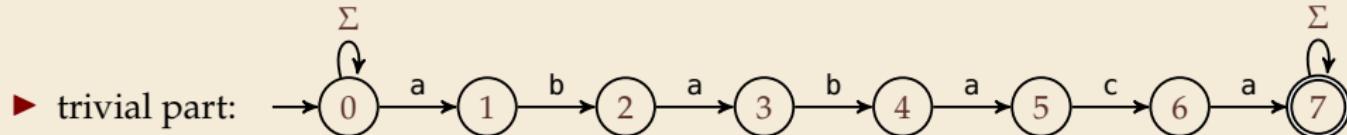
↝ New longest matched prefix will be (weakly) shorter than q

↝ All information about the text needed to determine it is contained in $P[0..q] \cdot c$!

6.4 Constructing String Matching Automata

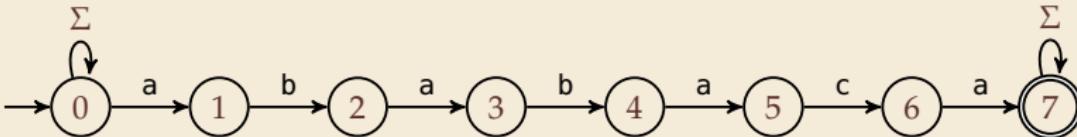
NFA instead of DFA?

It remains to *construct* the DFA.



NFA instead of DFA?

It remains to *construct* the DFA.

- trivial part: 
- that actually is a *nondeterministic finite automaton* (NFA) for $\Sigma^* P \Sigma^*$

~~ We *could* use the NFA directly for string matching:

- at any point in time, we are in a **set of states**
- accept when one of them is final state

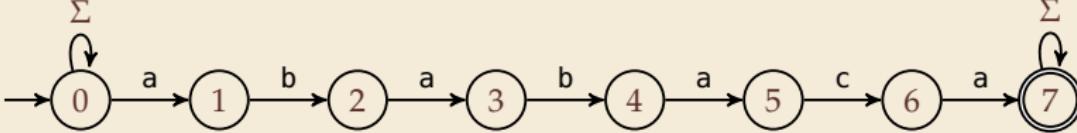
Example:

text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	0,1	0,1	0,2	0,1,3	0	0,1	0,1	0,2	0,1,3	0,2,4	0,1,3,5	0,6	0,1,7	

But maintaining a whole set makes this slow ...

NFA instead of DFA?

It remains to *construct* the DFA.

- trivial part: 
- that actually is a *nondeterministic finite automaton* (NFA) for $\Sigma^* P \Sigma^*$

~~ We *could* use the NFA directly for string matching:

- at any point in time, we are in a **set of states**
- accept when one of them is final state

Example:

text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	0,1	0,1	0,2	0,1,3	0	0,1	0,1	0,2	0,1,3	0,2,4	0,1,3,5	0,6	0,1,7	0,1,7

But maintaining a whole set makes this slow ...

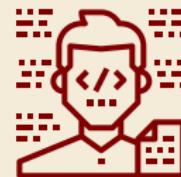
Computing DFA directly



You have an NFA and want a DFA?

Simply apply the power-set construction
(and maybe DFA minimization)!

The powerset method has exponential state blow up!
I guess I might as well use brute force string matching ...



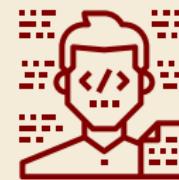
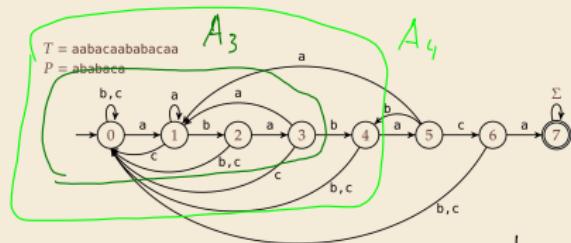
Computing DFA directly



You have an NFA and want a DFA?

Simply apply the power-set construction
(and maybe DFA minimization)!

$j = 3$



aba a
aba c

The powerset method has exponential state blow up!

I guess I might as well use brute force string matching ...



Ingenious algorithm by Knuth, Morris, and Pratt: construct DFA *inductively*:

Suppose we add character $P[j]$ to automaton A_j for $P[0..j)$ to construct A_{j+1}

- ▶ add new state and matching transition \rightsquigarrow easy $\textcircled{j} \xrightarrow{P[j+1]} \textcircled{j+1}$
- ▶ for each $c \neq P[j]$, we need $\delta(j, c)$ (transition from \textcircled{j} when reading c)

Computing DFA directly

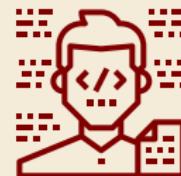


You have an NFA and want a DFA?

Simply apply the power-set construction
(and maybe DFA minimization)!

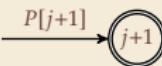
The powerset method has exponential state blow up!

I guess I might as well use brute force string matching ...



Ingenious algorithm by Knuth, Morris, and Pratt: construct DFA *inductively*:

Suppose we add character $P[j]$ to automaton A_j for $P[0..j)$ to construct A_{j+1}

- ▶ add new state and matching transition \rightsquigarrow easy 
- ▶ for each $c \neq P[j]$, we need $\delta(j, c)$ (transition from j when reading c)
- ▶ $\delta(j, c) =$ length of the longest prefix of $P[0..j)c$ that is a suffix of $P[1..j)c$
= state of automaton after reading $P[1..j)c$
 $\leq j \rightsquigarrow$ can use known automaton A_j for that!

State q means:
"we have seen $P[0..q)$ until here
(but not any longer prefix of P)"

Computing DFA directly



You have an NFA and want a DFA?

Simply apply the power-set construction
(and maybe DFA minimization)!

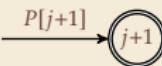
The powerset method has exponential state blow up!

I guess I might as well use brute force string matching ...



Ingenious algorithm by Knuth, Morris, and Pratt: construct DFA *inductively*:

Suppose we add character $P[j]$ to automaton A_j for $P[0..j)$ to construct A_{j+1}

- ▶ add new state and matching transition \rightsquigarrow easy 
- ▶ for each $c \neq P[j]$, we need $\delta(j, c)$ (transition from j when reading c)
- ▶ $\delta(j, c) =$ length of the longest prefix of $P[0..j)c$ that is a suffix of $P[1..j)c$
= state of automaton after reading $P[1..j)c$
 $\leq j \rightsquigarrow$ can use known automaton A_j for that!

\rightsquigarrow can directly compute A_{j+1} from A_j !

 seems to require simulating automata $m \cdot \sigma$ times

State q means:
"we have seen $P[0..q)$ until here
(but not any longer prefix of P)"

Computing DFA efficiently

- ▶ KMP's second insight: simulations in one step differ only in last symbol
 - ↝ simply maintain state x , the state after reading $P[1..j]$.
 - ▶ copy its transitions
 - ▶ update x by following transitions for $P[j]$

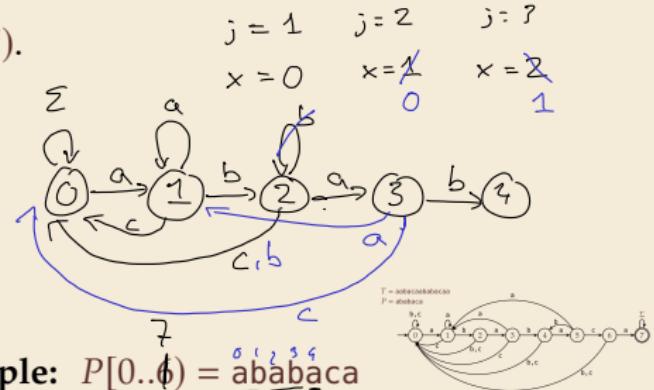
Computing DFA efficiently

- KMP's second insight: simulations in one step differ only in last symbol

~ simply maintain state x , the state after reading $P[1..j]$.

- copy its transitions
- update x by following transitions for $P[j]$

```
1 procedure constructDFA( $P[0..m]$ ):  
2     //  $\delta[q][c] = \text{target state when reading } c \text{ in state } q$   
3     for  $c \in \Sigma$  do  
4          $\delta[0][c] := 0$   
5          $\delta[0][P[0]] := 1$   
6          $x := 0$   
7         for  $j = 1, \dots, m - 1$  do  
8             for  $c \in \Sigma$  do // copy transitions  
9                  $\delta[j][c] := \delta[x][c]$   
10                 $\delta[j][P[j]] := j + 1$  // match edge  
11                 $x := \delta[x][P[j]]$  // update  $x$ 
```



$\delta(c, q)$	0	1	2	3	4	5	6
a	1	1	3	1			
b	0	2	0	4			
c	0	0	0	0			

Computing DFA efficiently

- ▶ KMP's second insight: simulations in one step differ only in last symbol
 - ↝ simply maintain state x , the state after reading $P[1..j]$.
 - ▶ copy its transitions
 - ▶ update x by following transitions for $P[j]$

```
1 procedure constructDFA( $P[0..m]$ ):  
2     //  $\delta[q][c]$  = target state when reading  $c$  in state  $q$   
3     for  $c \in \Sigma$  do  
4          $\delta[0][c] := 0$   
5          $\delta[0][P[0]] := 1$   
6          $x := 0$   
7         for  $j = 1, \dots, m - 1$  do  
8             for  $c \in \Sigma$  do // copy transitions  
9                  $\delta[j][c] := \delta[x][c]$   
10                 $\delta[j][P[j]] := j + 1$  // match edge  
11                 $x := \delta[x][P[j]]$  // update  $x$ 
```

Example: $P[0..6] = ababaca$

$\delta(c, q)$	0	1	2	3	4	5	6
a	1	1	3	1	5	1	7
b	0	2	0	4	0	4	0
c	0	0	0	0	0	6	0

String matching with DFA – Discussion

► Time:

- Matching: n table lookups for DFA transitions
- building DFA: $\Theta(m\sigma)$ time (constant time per transition edge).
 $\leadsto \Theta(m\sigma + n)$ time for string matching.

► Space:

- $\Theta(m\sigma)$ space for transition matrix.



fast matching time actually: hard to beat!

Unicode $\sigma \approx 150k$



total time asymptotically optimal for small alphabet (for $\sigma = O(n/m)$)



substantial **space overhead**, in particular for large alphabets

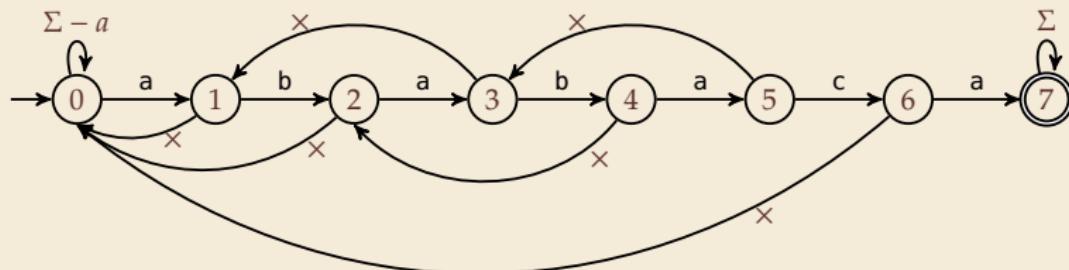
6.5 The Knuth-Morris-Pratt algorithm

Failure Links

- ▶ Recall: String matching with is DFA fast,
but needs table of $m \times \sigma$ transitions.
- ▶ in fast DFA construction, we used that all simulations differ only by *last* symbol
- ↝ **KMP's third insight:** do this last step of simulation from state x during *matching*!
... but how?

Failure Links

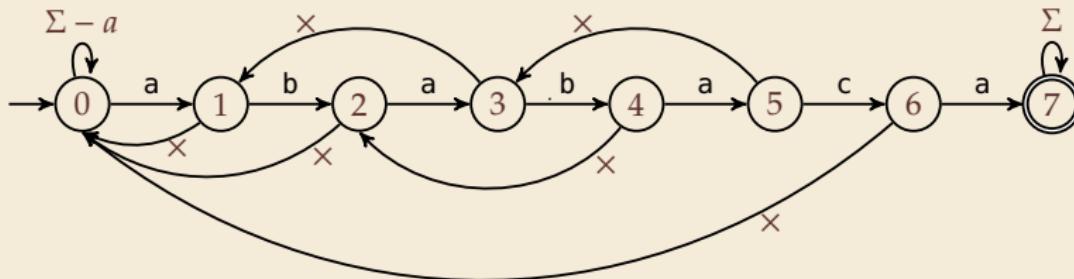
- ▶ Recall: String matching with is DFA fast, but needs table of $m \times \sigma$ transitions.
- ▶ in fast DFA construction, we used that all simulations differ only by *last symbol*
- ~~ **KMP's third insight:** do this last step of simulation from state x during *matching!* ... but how?
- ▶ **Answer:** Use a new type of transition: \times , the *failure links*
 - ▶ Use this transition (only) if no other one fits.
 - ▶ \times does not consume a character. ~~ might follow several failure links



~~ Computations are deterministic (but automaton is not a real DFA.)

Failure link automaton – Example

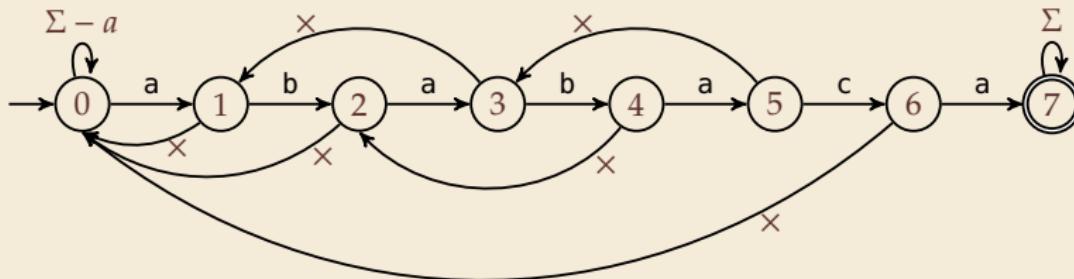
Example: $T = abababaaaaca$, $P = ababaca$



$T :$	a	b	a	b	a	b	a	a	b	a	b
	1	2	3	4	5,3	4	5,3,1,0	1	2	3	4

Failure link automaton – Example

Example: $T = abababaaaaca$, $P = ababaca$



T :	a	b	a	b	a	b	a	a	a	b	a	b
P :	a	b	a	b	a	x						
			(a)	(b)	(a)	b	a		x			
								a		b	a	b

to state 3
to state 1

q :	1	2	3	4	5	3,4	5	3,1,0,1	2	3	4
-------	---	---	---	---	---	-----	---	---------	---	---	---

(after reading this character)

Clicker Question



What is the worst-case time to process one character in a failure-link automaton for $P[0..m]$?

A $\Theta(1)$

B $\Theta(\log m)$

C $\Theta(m)$

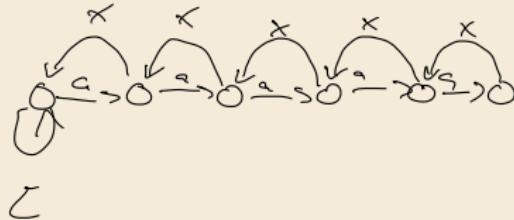
D $\Theta(m^2)$



→ *sli.do/cs566*

Clicker Question

$$P = a^m$$



What is the worst-case time to process one character in a failure-link automaton for $P[0..m]$?

A ~~$\Theta(1)$~~

C $\Theta(m)$ ✓

B ~~$\Theta(\log m)$~~

D ~~$\Theta(m^2)$~~



→ *sli.do/cs566*

The Knuth-Morris-Pratt Algorithm

```
1 procedure KMP( $T[0..n]$ ,  $P[0..m]$ ):
2    $fail[0..m] :=$  failureLinks( $P$ )
3    $i := 0$  // current position in  $T$ 
4    $q := 0$  // current state of KMP automaton
5   while  $i < n$  do
6     if  $T[i] == P[q]$  then
7        $i := i + 1$ ;  $q := q + 1$ 
8       if  $q == m$  then
9         return  $i - q$  // occurrence found
10    else // i.e.  $T[i] \neq P[q]$ 
11      if  $q \geq 1$  then
12         $q := fail[q]$  // follow one ×
13      else
14         $i := i + 1$ 
15  end while
16  return NO_MATCH
```

- ▶ only need single array $fail$ for failure links
- ▶ (failureLinks on next slide)

The Knuth-Morris-Pratt Algorithm

```
1 procedure KMP( $T[0..n]$ ,  $P[0..m]$ ):
2    $fail[0..m] :=$  failureLinks( $P$ )
3    $i := 0$  // current position in  $T$ 
4    $q := 0$  // current state of KMP automaton
5   while  $i < n$  do
6     if  $T[i] == P[q]$  then
7        $i := i + 1$ ;  $q := q + 1$ 
8       if  $q == m$  then
9         return  $i - q$  // occurrence found
10    else // i.e.  $T[i] \neq P[q]$ 
11      if  $q \geq 1$  then
12         $q := fail[q]$  // follow one ×
13      else
14         $i := i + 1$ 
15  end while
16  return NO_MATCH
```

- ▶ only need single array $fail$ for failure links
- ▶ (failureLinks on next slide)

Analysis: (matching part)

- ▶ always have $fail[j] < j$ for $j \geq 1$
 - ~~ in each iteration
 - ▶ either advance position in text ($i := i + 1$)
 - ▶ or shift pattern forward (guess $i - q$)
 - ▶ each can happen at most n times
 - ~~ $\leq 2n$ symbol comparisons!

Computing failure links

- ▶ failure links point to error state x (from DFA construction)
 - ~~~ run same algorithm, but store $fail[j] := x$ instead of copying all transitions

```
1 procedure failureLinks( $P[0..m]$ ):  
2      $fail[0] := 0$     // dummy  
3      $x := 0$   
4     for  $j := 1, \dots, m - 1$  do  
5          $fail[j] := x$   
6         // update failure state using failure links:  
7         while  $P[x] \neq P[j]$   
8             if  $x == 0$  then  
9                  $x := -1$ ; break  
10            else  
11                 $x := fail[x]$   
12            end while  
13             $x := x + 1$   
14        end for
```

Computing failure links

- ▶ failure links point to error state x (from DFA construction)
 - ~~ run same algorithm, but store $fail[j] := x$ instead of copying all transitions

```
1 procedure failureLinks( $P[0..m]$ ):  
2      $fail[0] := 0$   
3      $x := 0$   
4     for  $j := 1, \dots, m - 1$  do  
5          $fail[j] := x$   
6         // update failure state using failure links:  
7         while  $P[x] \neq P[j]$   
8             if  $x == 0$  then  
9                  $x := -1$ ; break  
10            else  
11                 $x := fail[x]$   
12            end while  
13             $x := x + 1$   
14        end for
```

Analysis:

- ▶ m iterations of for loop
- ▶ while loop always decrements x
- ▶ x is incremented only once per iteration of for loop
 - ~~ $\leq m$ iterations of while loop *in total*
 - ~~ $\leq 2m$ symbol comparisons

Knuth-Morris-Pratt – Discussion

► Time:

- $\leq 2n + 2m = O(n + m)$ character comparisons
- clearly must at least *read* both T and P
- ~ KMP has optimal worst-case complexity!

► Space:

- $\Theta(m)$ space for failure links

 total time asymptotically optimal (for any alphabet size)

 reasonable extra space

Clicker Question

What are the main advantages of the KMP string matching (using the failure-link automaton) over string matching with DFAs? Check all that apply.



- A** faster preprocessing on pattern
- B** faster matching in text
- C** fewer character comparisons
- D** uses less space
- E** makes running time independent of σ
- F** I don't have to do automata theory



→ *sli.do/cs566*

Clicker Question

What are the main advantages of the KMP string matching (using the failure-link automaton) over string matching with DFAs? Check all that apply.



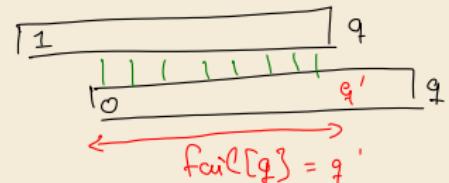
- A** faster preprocessing on pattern ✓
- B** ~~faster matching in text~~
- C** ~~fewer character comparisons~~
- D** uses less space ✓
- E** makes running time independent of σ ✓
- F** ~~I don't have to do automata theory~~



→ *sli.do/cs566*

The KMP prefix function

- ▶ It turns out that the failure links are useful beyond KMP
- ▶ a slight variation is (more?) widely used: (for historic reasons)
the (KMP) *prefix function* $F : [1..m - 1] \rightarrow [0..m - 1]$:
 $F[j]$ is the length of the longest prefix of $P[0..j]$
that is a suffix of $P[1..j]$.
- ▶ Can show: $fail[j] = F[j - 1]$ for $j \geq 1$, and hence



$fail[q] = \text{length of the}$
 $\text{longest prefix of } P[0..q)$
 $\text{that is a suffix of } P[1..q)$.

memorize this!

- ▶ EAA Buch: String indices are 1-based, but definition of failure links matches! $\Pi_P(q) = fail[q]$
 $\Pi_P : [1..m] \rightarrow [0..m - 1]$ with $\Pi_P(q) = \max\{k \in \mathbb{N}_0 : k < q \wedge P[0..k) \sqsupseteq P[0..q]\} = fail[q]$

6.6 Beyond Optimal? The Boyer-Moore Algorithm

Motivation

- ▶ KMP is an optimal algorithm, isn't it? What else could we hope for?

Motivation

- ▶ KMP is an optimal algorithm, isn't it? What else could we hope for?
 - ▶ KMP is “only” optimal in the worst-case (and up to constant factors)
 - ▶ how many comparisons do we need for the following instance?
 $T = \text{aaaaaaaaaaaaaaaaa}$, $P = \text{xxxxx}$
 - ▶ there are no matches
 - ▶ we can *certify* the correctness of that output with only 4 comparisons:

~~ We did *not* even read most characters!

Boyer-Moore Algorithm

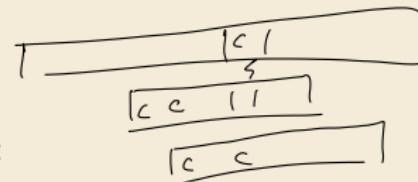
- ▶ Let's check guesses *from right to left*!
- ▶ If we are lucky, we can eliminate several shifts in one shot!

Boyer-Moore Algorithm

- ▶ Let's check guesses *from right to left*!
- ▶ If we are lucky, we can eliminate several shifts in one shot!



must avoid (excessive) redundant checks, e. g., for $T = a^n$, $P = ba^{m-1}$



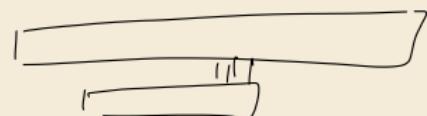
↝ New rules:

- ▶ **Bad character jumps:** Upon mismatch at $T[i] = c$:
 - ▶ If P does not contain c , shift P entirely past i !
 - ▶ Otherwise, shift P to align the *last occurrence* of c in P with $T[i]$.
- ▶ **Good suffix jumps:**

Upon a mismatch, shift so that the already matched *suffix* of P aligns with a previous occurrence of that suffix (or part of it) in P .

(Details follow; ideas similar to KMP failure links)

↝ two possible shifts (next guesses); use larger jump.



Boyer-Moore Algorithm – Code

```
1 procedure boyerMoore( $T[0..n]$ ,  $P[0..m]$ ):
2      $\lambda$  := computeLastOccurrences( $P$ )
3      $\gamma$  := computeGoodSuffixes( $P$ )
4      $i$  := 0 // current guess
5     while  $i \leq n - m$ 
6          $j$  :=  $m - 1$  // next position in  $P$  to check
7         while  $j \geq 0 \wedge P[j] == T[i + j]$  do
8              $j$  :=  $j - 1$ 
9             if  $j == -1$  then
10                return  $i$ 
11            else
12                 $i$  :=  $i + \max\{j - \lambda[T[i + j]], \gamma[j]\}$ 
13    return NO_MATCH
```

- ▶ λ and γ explained below
- ▶ shift forward is larger of two heuristics
- ▶ shift is always positive (see below)

Bad character examples

$P = a l d o$

$T = w h e r e i s w a l d o$

				o									

$P = m o o r e$

$T = b o y e r m o o r e$

Bad character examples

P = a l d o

T = w h e r e i s w a l d o

P = *m o o r e*

T = *b* *o* *v* *e* *r* *m* *o* *o* *r* *e*

Bad character examples

P = a l d o

T = *w h e r e i s w a l d o*

P = m o o r e

T = *b* *o* *y* *e* *r* *m* *o* *o* *r* *e*

Bad character examples

P = a l d o

T = *w h e r e i s w a l d o*

P = *m o o r e*

T = *b o y e r m o o r e*

Bad character examples

$P = a l d o$

$T = w h e r e i s w a l d o$

$P = m o o r e$

$T = b o y e r m o o r e$

Bad character examples

$P = a l d o$

$T = w h e r e i s w a l d o$

$P = m o o r e$

$T = b o y e r m o o r e$

Bad character examples

$P = a l d o$

$T = w h e r e i s w a l d o$

↝ 6 characters not looked at

$P = m o o r e$

$T = b o y e r m o o r e$

Bad character examples

$P = a l d o$

$T = w h e r e i s w a l d o$

↝ 6 characters not looked at

$P = m o o r e$

$T = b o y e r m o o r e$

Bad character examples

$P = a l d o$

$T = w h e r e i s w a l d o$

↝ 6 characters not looked at

$P = m o o r e$

$T = b o y e r m o o r e$

Bad character examples

$P = a l d o$

$T = w h e r e i s w a l d o$

↝ 6 characters not looked at

$P = m o o r e$

$T = b o y e r m o o r e$

Bad character examples

$P = a l d o$

$T = w h e r e i s w a l d o$

↝ 6 characters not looked at

$P = m o o r e$

$T = b o y e r m o o r e$

Bad character examples

$P = a l d o$

$T = w h e r e i s w a l d o$

↝ 6 characters not looked at

$P = m o o r e$

$T = b o y e r m o o r e$

↝ 4 characters not looked at

Last-Occurrence Function

- ▶ Preprocess pattern P and alphabet Σ
- ▶ *last-occurrence function* $\lambda[c]$ defined as
 - ▶ the largest index i such that $P[i] = c$ or
 - ▶ -1 if no such index exists

Last-Occurrence Function

- ▶ Preprocess pattern P and alphabet Σ
- ▶ *last-occurrence function* $\lambda[c]$ defined as
 - ▶ the largest index i such that $P[i] = c$ or
 - ▶ -1 if no such index exists

- ▶ **Example:** $P = \text{moore}$

c	m	o	r	e	all others
$\lambda[c]$	0	2	3	4	-1

P	=	m	o	o	r	e					
T	=	b	o	y	e	r	m	o	o	r	e
						e					

						(r)	e				

$$i = 0, j = 4, T[i + j] = r, \lambda[r] = 3$$

↷ shift by $j - \lambda[T[i + j]] = 1$

- ▶ λ computed in $O(m + \sigma)$ time.
- ▶ store as array $\lambda[0..\sigma)$.

```
1 procedure computeLastOccurrences( $P[0..m]$ ):  
2    $\lambda[0..\sigma)$  := array initialized to 0  
3   for  $j = 0, \dots, m - 1$   
4      $\lambda[P[j]] := j$   
5   return  $\lambda$ 
```

Good suffix examples

1. $P = \text{sell}_s \text{ shells}$

s	h	e	i	l	a	<u>s</u>	e	l	l	s	<u>s</u>	h	e	l	l	s

Good suffix examples

1. $P = \text{sell}_s \text{ shells}$

s	h	e	i	l	a	u	s	e	l	l	s	u	s	h	e	l	l	s
							h	e	l	l	s							

Good suffix examples

1. $P = \underline{\text{sell}}s \cup \underline{\text{shells}}$

s	h	e	i	l	a	u	s	e	l	l	s	u	s	h	e	l	l	s
							h	e	l	l	s							

Below the second row, corresponding to the highlighted 'h' in the first row, are the characters (e), (l), (l), (s) in parentheses, indicating the good suffix 'ells'.

Good suffix examples

1. $P = \text{sell}_s \text{ shells}$

s	h	e	i	l	a	\sqcup	s	e	l	l	s	\sqcup	s	h	e	l	l	s
							h	e	l	l	s							
							(e)	(l)	(l)	(s)								

2. $P = \underline{o} \underline{det} \underline{o} \underline{food}$

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o
					o	f	o	o	d									

Good suffix examples

1. $P = \text{sell}_s \text{ shells}$

s	h	e	i	l	a	u	s	e	l	l	s	u	s	h	e	l	l	s
							h	e	l	l	s							
							(e)	(l)	(l)	(s)								

2. $P = \underline{o} \underline{det} \underline{o} \underline{food}$

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o
				o	f	o	o	d										
							(o)	(d)										

Good suffix examples

1. $P = \text{sell}_s \text{ shells}$

s	h	e	i	l	a	\sqcup	s	e	l	l	s	\sqcup	s	h	e	l	l	s
							h	e	l	l	s							
							(e)	(l)	(l)	(s)								

2. $P = \text{ode to food}$

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o
				o	f	o	o	d										
						(o)	(d)											

matched suffix

► Crucial ingredient: longest suffix of $P[j+1..m]$ that occurs earlier in P .

► 2 cases (as illustrated above)

1. complete suffix occurs in $P \rightsquigarrow$ characters left of suffix are *not* known to match
2. part of suffix occurs at beginning of P

Good suffix jumps

- ▶ Precompute *good suffix jumps* $\gamma[0..m]$:
 - ▶ For $0 \leq j < m$, $\gamma[j]$ stores shift if search failed at $P[j]$
 - ▶ At this point, had $T[i+j+1..i+m] = P[j+1..m]$, but $T[i] \neq P[j]$

Good suffix jumps

- ▶ Precompute *good suffix jumps* $\gamma[0..m]$:
 - ▶ For $0 \leq j < m$, $\gamma[j]$ stores shift if search failed at $P[j]$
 - ▶ At this point, had $T[i+j+1..i+m] = P[j+1..m]$, but $T[i] \neq P[j]$
- ~~ $\gamma[j]$ is the shift $m - \ell$ for the *largest* ℓ such that
 - ▶ $P[j+1..m]$ is a suffix of $P[0..\ell]$ and $P[j] \neq P[j-(m - \ell)]$

								h	e	l	l	s									
								×	(e)	(l)	(l)	(s)									

-OR-

- ▶ $P[0..\ell]$ is a suffix of $P[j+1..m]$

				o	f	o	o	d													
								(o)	(d)												

Good suffix jumps

- ▶ Precompute *good suffix jumps* $\gamma[0..m]$:
 - ▶ For $0 \leq j < m$, $\gamma[j]$ stores shift if search failed at $P[j]$
 - ▶ At this point, had $T[i+j+1..i+m] = P[j+1..m]$, but $T[i] \neq P[j]$
- ~~ $\gamma[j]$ is the shift $m - \ell$ for the *largest* ℓ such that
 - ▶ $P[j+1..m]$ is a suffix of $P[0..\ell]$ and $P[j] \neq P[j-(m - \ell)]$

								h	e	l	l	s											
								×	(e)	(l)	(l)	(s)											

-OR-

- ▶ $P[0..\ell]$ is a suffix of $P[j+1..m]$

				o	f	o	o	d															
								(o)	(d)														

- ▶ Computable (similar to KMP failure function) in $\Theta(m)$ time.

Good suffix jumps – Efficient Computation

exam

```
1 procedure computeGoodSuffixes( $P[0..m]$ ):  
2    $fail[0..m]$  := failureLinks( $P$ )  
3    $revFail[0..m]$  := failureLinks(reverseString( $P$ ))  
4    $\gamma[0..m]$  := new array initialized to  $m - fail[m]$   
5   for  $\ell := 1, \dots, m$   
6      $j := m - revFail[\ell] - 1$   
7     if  $\gamma[j] > \ell - fail[\ell]$   
8        $\gamma[j] := \ell - revFail[\ell]$   
9     end if  
10   end for  
11   return  $\gamma$ 
```

- ▶ Reuses failureLinks function from KMP
 - ▶ on both P and the reversed pattern!
- ▶ Correctness not obvious ...
Requires careful analysis of all possible cases
- ▶ Clearly $\Theta(m)$ time

Boyer-Moore algorithm – Discussion

-  Worst-case running time $\in O(n + m + \sigma)$ if P does *not* occur in T .
(follows from not at all obvious analysis!)
-  As given, worst-case running time $\Theta(nm)$ if we want to report all occurrences
 - ▶ To avoid that, have to keep track of implied matches.
(tricky because they can be in the “middle” of P)
 - ▶ Note: KMP reports all matches in $O(n + m)$ without modifications!
-  On typical *English text*, Boyer Moore probes only approx. 25% of the characters in T !
 - ~~ Faster than KMP on English text.
-  requires moderate extra space $\Theta(m + \sigma)$