

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 $\{A, C, G, T\}; \dots$
 - ▶ $\Sigma^n = \Sigma \times \dots \times \Sigma$: 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** $n \in \mathbb{N}_0$ (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 **ith** 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:**

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

► 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
 $\rightsquigarrow \#checks \leq n \cdot m$ (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:

$T = abbbabababb$

$P = abba$

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!



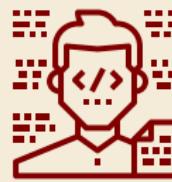
Job done!

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!



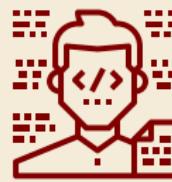
WTF!?

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!

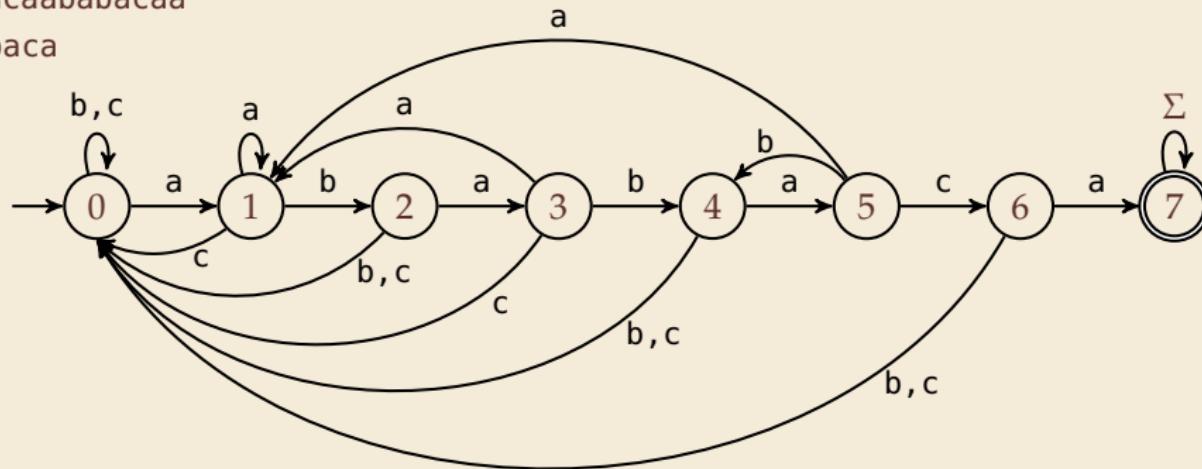
String matching with DFA

- ▶ 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

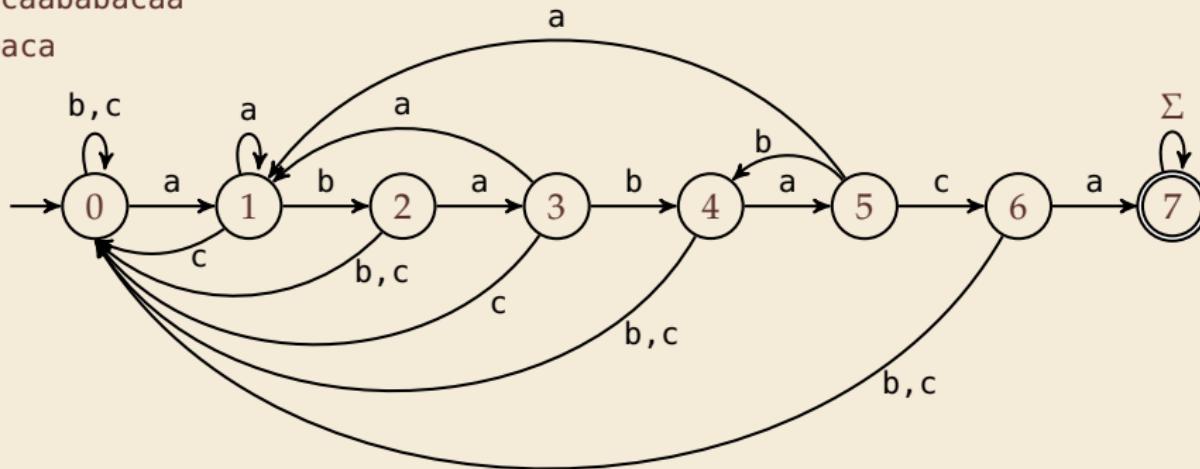
String matching with DFA

- ▶ 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