

# 4

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

*24 February 2020*

Sebastian Wild

## 4 String Matching

- 4.1 Introduction
- 4.2 Brute Force
- 4.3 String Matching with Finite Automata
- 4.4 The Knuth-Morris-Pratt algorithm
- 4.5 Beyond Optimal? The Boyer-Moore Algorithm
- 4.6 The Rabin-Karp Algorithm

## 4.1 Introduction

# 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  $\rightsquigarrow$  ultimately any data is a string

$\rightsquigarrow$  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  $\rightsquigarrow$  ultimately any data is a string


$\rightsquigarrow$  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

# Notations

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

# Notations

- ▶ *alphabet*  $\Sigma$ : finite set of allowed **characters**;  $|\Sigma|$  “a string over alphabet  $\Sigma$ ”
  - ▶ 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
- ▶  $\Sigma^n = \Sigma \times \cdots \times \Sigma$ : strings of **length**  $n \in \mathbb{N}_0$  ( $n$ -tuples)
- ▶  $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$ : set of **all** (finite) strings over  $\Sigma$
- ▶  $\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n$ : set of **all** (finite) **nonempty** strings over  $\Sigma$
- ▶  $\varepsilon$   $\in \Sigma^0$ : the *empty* string (same for all alphabets)

# Notations

- ▶ *alphabet*  $\Sigma$ : finite set of allowed **characters**;  $\sigma = |\Sigma|$  “a string over alphabet  $\Sigma$ ”
  - ▶ 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
- ▶  $\Sigma^n = \Sigma \times \cdots \times \Sigma$ : strings of **length**  $n \in \mathbb{N}_0$  ( $n$ -tuples)
- ▶  $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$ : set of **all** (finite) strings over  $\Sigma$
- ▶  $\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n$ : set of **all** (finite) **nonempty** strings over  $\Sigma$
- ▶  $\varepsilon \in \Sigma^0$ : the *empty* string (same for all alphabets)
  - $\swarrow$   $\Sigma^0 \dots \Sigma^n - \{\}$
- ▶ for  $\underline{S \in \Sigma^n}$ , 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**  $S[i] \cdot S[i+1] \cdots S[j]$  ( $0 \leq i \leq j < n$ )
  - ▶  $S[0..j]$  is a **prefix** of  $S$ ;  $S[i..n-1]$  is a **suffix** of  $S$
  - ▶  $S[i..j) = S[i..j \text{ } \overline{\neq} 1]$  (endpoint exclusive)  $\rightsquigarrow S = \underline{S[0..n)}$



## Clicker Question



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

**A** True

**B** False

*[pingo.upb.de/622222](https://pingo.upb.de/622222)*

## Clicker Question



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

**A** True ✓

**B** ~~False~~

*[pingo.upb.de/622222](https://pingo.upb.de/622222)*

# 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  $T$ :  $\min\{i \in [0..n-m) : T[i..i+m) = P\}$
- 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

## 4.2 Brute Force

# Abstract idea of algorithms

Pattern matching algorithms consist of *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 pair  $(\underline{i}, \underline{j})$  where we compare  $T[i + j]$  to  $P[j]$ .
- ▶ Note: need all  $m$  checks to verify a single **correct** guess  $\underline{i}$ ,  
but it may take (many) fewer checks to recognize an **incorrect** guess.
- ▶ Cost measure: #character comparisons = #checks

$\leadsto \text{cost} \leq n \cdot m$  (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!

► Example:

$T = \text{abbbababbab}$

$P = \text{abba}$

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

[illegible]

# 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!

► **Example:**

$T = \text{abbbababbab}$

$P = \text{abba}$

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

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

▶ Worst possible input:  $P = a^{m-1}b$ ,  
 $T = a^n$

▶ Worst-case performance:  $(n - m + 1) \cdot m$

$\rightsquigarrow$  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		
							a	a	a	b	

▶ Worst possible input:  $P = a^{m-1}b$ ,  
 $T = a^n$

▶ Worst-case performance:  $(n - m + 1) \cdot m$

$\rightsquigarrow$  for  $m \leq n/2$  that is  $\Theta(mn)$

- ▶ Bad input: lots of self-similarity in  $T$ !  $\rightsquigarrow$  can we exploit that?
- ▶ brute force does ‘obviously’ stupid repetitive comparisons  $\rightsquigarrow$  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** (↪ Unit 6): Do preprocessing on the text  $T$   
Can find matches in time *independent of text size(!)*
  - ▶ inverted indices
  - ▶ Suffix trees
  - ▶ Suffix arrays

## 4.3 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!

*[pingo.upb.de/622222](https://pingo.upb.de/622222)*

# 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
- $\rightsquigarrow \exists$  *deterministic finite automaton* (DFA) to recognize  $\Sigma^* \cdot P \cdot \Sigma^*$
- $\rightsquigarrow$  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
- $\rightsquigarrow (\exists)$  *deterministic finite automaton* (DFA) to recognize  $\Sigma^* \cdot P \cdot \Sigma^*$
- $\rightsquigarrow$  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!



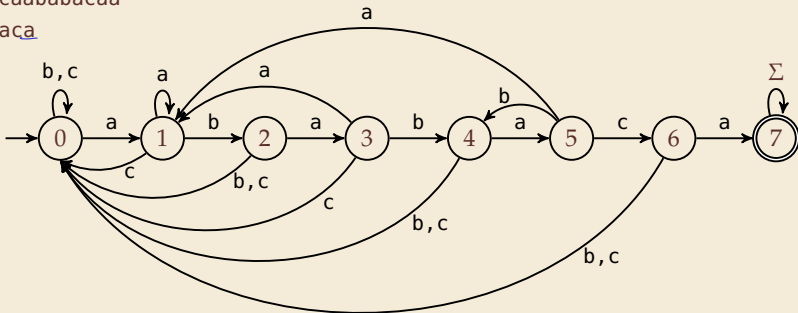
# String matching with DFA

- 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	1	2	3										

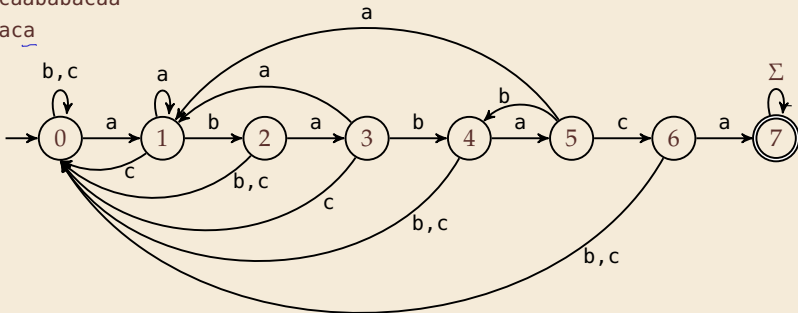
# String matching with DFA

- 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	<u>1</u>	2	3	4	5	6	7	7

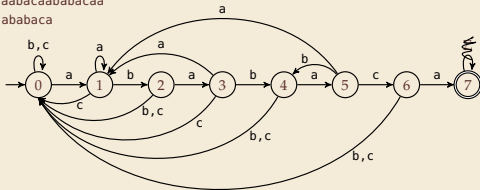
# String matching DFA – Intuition

Why does this work?

- Main insight: *Invariant*

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

$T = \text{aabacaababacaa}$   
 $P = \text{ababaca}$



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

$P[q]$   
 $\neq$

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

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

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

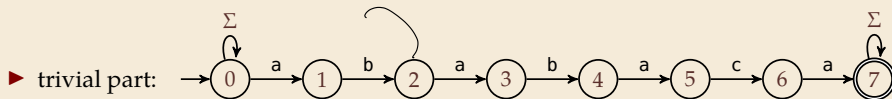
$$q' \leq q$$

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

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

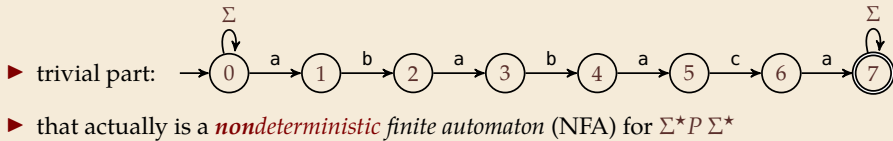
## NFA instead of DFA?

It remains to *construct* the DFA.



# NFA instead of DFA?

It remains to *construct* the DFA.



~> 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:** *Previous versions of this example were missing states; this is the correct version:*

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 ...



# 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 ...



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

Suppose we add character  $P[j]$  to automaton  $A_{j-1}$  for  $P[0..j-1]$

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



# 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 ...



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

Suppose we add character  $P[j]$  to automaton  $A_{j-1}$  for  $P[0..j-1]$

- ▶ 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  $\underline{P[1..j)c}$   
 $\leq j \rightsquigarrow$  can use known automaton  $A_{j-1}$  for that!

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



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  $P$ )”