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

20 October 2023

Sebastian Wild

COMP526 (Fall 2023)
University of Liverpool
version `2023-10-20 10:45`

Learning Outcomes

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*.

Unit 4: ***String Matching***

Outline

4 String Matching

4.1 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 $\rightsquigarrow$ 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* 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 $\simeq$ 130k
  - $\{0,1\}$; nucleotides $\{A,C,G,T\}$; . . .

comprehensive standard character set
including emoji and all known symbols

Notations

- *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
    comprehensive standard character set including emoji and all known symbols
  - $\{0, 1\}$; nucleotides $\{A, C, G, T\}$; ...
- $\Sigma^n = \Sigma \times \cdots \times \Sigma$: strings of **length** $n \in \mathbb{N}_0$ (n-tuples)
- $\Sigma^\star = \bigcup_{n \geq 0} \Sigma^n$: 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)

Notations

- *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

- $\Sigma^n = \Sigma \times \cdots \times \Sigma$: strings of **length** $n \in \mathbb{N}_0$ (n-tuples)
- $\Sigma^\star = \bigcup_{n \ge 0} \Sigma^n$: set of **all** (finite) strings over Σ
- $\Sigma^+ = \bigcup_{n \ge 1} \Sigma^n$: set of **all** (finite) **nonempty** strings over Σ
- $\varepsilon \in \Sigma^0$: the *empty* string (same for all alphabets)

zero-based (like arrays)!

- for $S \in \Sigma^n$, write $S[i]$ (other sources: S_i) for i**th** character $(0 \le i < n)$
- for $S, T \in \Sigma^\star$, 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** $S[i] \cdot S[i+1] \cdots S[j]$ $(0 \le i \le j < n)$
  - $S[0..j]$ is a **prefix** of S; $S[i..n-1]$ is a **suffix** of S
  - $S[i..j) = S[i..j-1]$ (endpoint exclusive) $\rightsquigarrow$ $S = S[0..n)$

Clicker Question

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

A True

B False

→ sli.do/comp526

Clicker Question

strings of length ≥ 0

True or false: $\underbrace{\Sigma^\star}_{} = \Sigma^+ \cup \{\varepsilon\}$

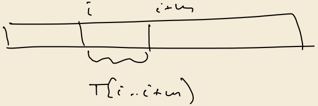
strings of length ≥ 1

A True ✓

B ~~False~~

→ sli.do/comp526

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.

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

- **Example:**
  - T = "`Where is he?`"
  - P_1 = "`he`" $\rightsquigarrow$ $i = 1$
  - P_2 = "`who`" $\rightsquigarrow$ `NO_MATCH`
- string matching is implemented in Java in `String.indexOf`, in Python as `str.find`

Clicker Question

0 1 2 3 4 5 6 7 8 9 ..

Let T = `COMP526␣is␣fun.`

What is $T[3..7)$?

→ *sli.do/comp526*

Clicker Question

Let T = `COMP526␣is␣fun.`
What is $T[3..7)$?

```
012345678901234
COMP526␣is␣fun.
```

→ *sli.do/comp526*

4.2 Brute Force

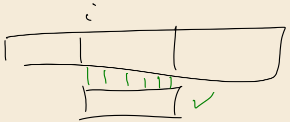
Abstract idea of algorithms

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 $0 \le i \le n - m$.
- 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 $\le n \cdot m$ (number of possible checks)

Brute-force method

```
procedure bruteForceSM(T[0..n), P[0..m))
    for i := 0, ..., n − m − 1 do
        for j := 0, ..., m − 1 do
            if T[i + j] ≠ P[j] then break inner loop
        if j == m then return i
    return NO_MATCH
```

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

- **Example:**
  T = abbbababbab
  P = abba

a	b	b	b	a	b	a	b	b	a	b
a	b	b	4 a							
	a									
		a								
			a							
				a	b	b				

Brute-force method

```
procedure bruteForceSM(T[0..n), P[0..m))
    for i := 0, ..., n − m − 1 do
        for j := 0, ..., m − 1 do
            if T[i + j] ≠ P[j] then break inner loop
        if j == m then return i
    return NO_MATCH
```

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

- **Example:**
  T = `abbbababbab`
  P = `abba`

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

a	b	b	b	a	b	a	b	b	a	b
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 $m \le n/2$ 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 $m \le n/2$ 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** ($\rightsquigarrow$ Unit ~~6~~ 8): 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!

→ `sli.do/comp526`

Theoretical Computer Science to the rescue!

- string matching = deciding whether $T \in \Sigma^\star \cdot P \cdot \Sigma^\star$

first occurrence ?

- $\Sigma^\star \cdot P \cdot \Sigma^\star$ is *regular* formal language

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

automata / formal languages 101

$\rightsquigarrow$ can check for occurrence of P in $|T| = n$ steps!

Theoretical Computer Science to the rescue!

- string matching = deciding whether $T \in \Sigma^\star \cdot P \cdot \Sigma^\star$
- $\Sigma^\star \cdot P \cdot \Sigma^\star$ is *regular* formal language

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

$\rightsquigarrow$ can check for occurrence of P in $|T| = n$ steps!

Job done!

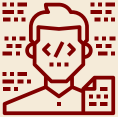
Theoretical Computer Science to the rescue!

- string matching $=$ deciding whether $T \in \Sigma^\star \cdot P \cdot \Sigma^\star$
- $\Sigma^\star \cdot P \cdot \Sigma^\star$ is *regular* formal language

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

$\rightsquigarrow$ can check for occurrence of P in $|T| = n$ steps!

Job done!



`WTF!?`

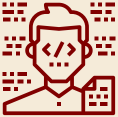
Theoretical Computer Science to the rescue!

- string matching $=$ deciding whether $T \in \Sigma^\star \cdot P \cdot \Sigma^\star$
- $\Sigma^\star \cdot P \cdot \Sigma^\star$ is *regular* formal language

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

$\rightsquigarrow$ 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

Σ = alphabet
Q = set of states

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

exactly 1 arc foreach pair (q, c)

transition function

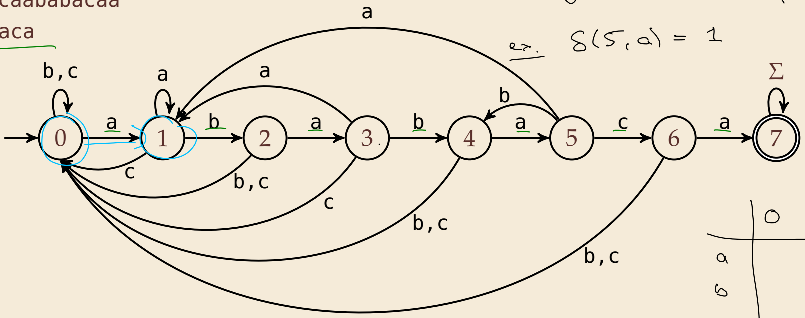
$\delta : Q \times \Sigma \to Q$

ex. $\delta(5, a) = 1$

Example:

T = aabacaababacaa

P = ababaca



	0	1	2	3	4	5	6	7
a						1		
b								
c								

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	

String matching with DFA

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

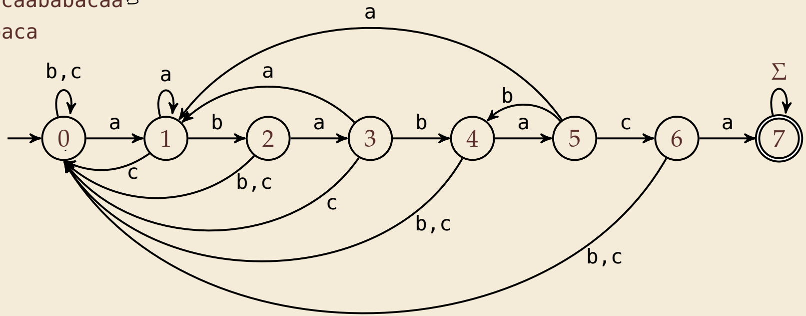
time to find first occurrence

$O(n)$

Example:

T = aabacaababacaab

P = ababaca

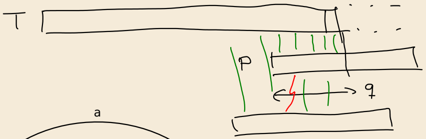


text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3	0	1	1	2	3	4	5	6	7	7

b

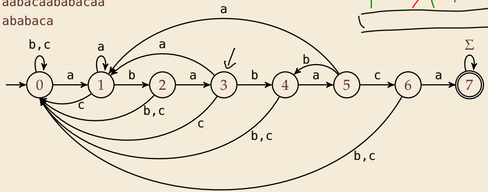
7 7 7

String matching DFA – Intuition



Why does this work?

T = aabacaababacaa
P = ababaca

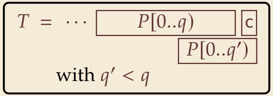


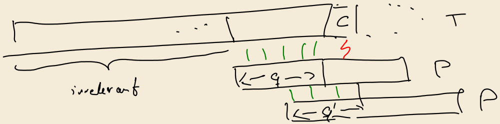
- Main insight:

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

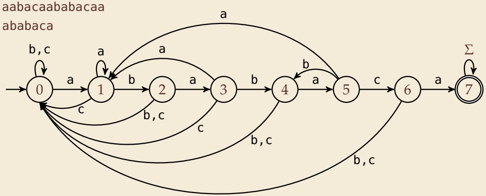




String matching DFA – Intuition

Why does this work?

T = aabacaababacaa
P = ababaca

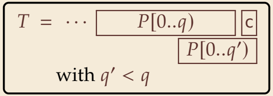


- Main insight:

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



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

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

Note: our automata stay in state m forever once they found the first occurrence

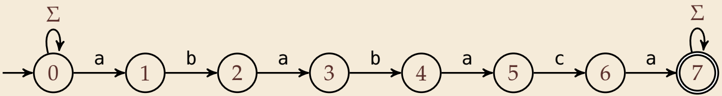
one can also give edges to m to keep finding occurrences

$\Rightarrow$ DFA can find all occurrences in time $O(n)$

4.4 Constructing String Matching Automata

NFA instead of DFA?

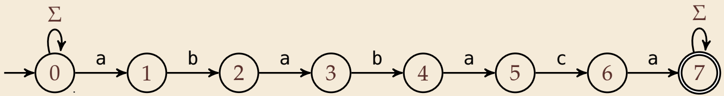
It remains to *construct* the DFA.

- trivial part: 

NFA instead of DFA?

It remains to *construct* the DFA.

- trivial part:



- that actually is a ***nondeterministic** finite automaton* (NFA) for $\Sigma^\star P\,\Sigma^\star$

⇝ 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:

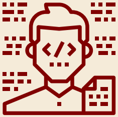
text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	0, 1	0, 1	0, 2	0, 1, 3	0	0, 1	0, 1	0, 2	0, 1, 3	0, 2, 4	0, 1, 3, 5	0, 6	0, 1, 7	0, 1, 7

But maintaining a whole set makes this slow . . . $\Theta(n \cdot m)$ w.c.

Computing DFA directly

You have an NFA and want a DFA?
Simply apply the power-set construction
(and maybe DFA minimization)!

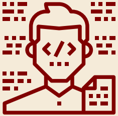
`The powerset method has exponential state blow up!`
`I guess I might as well use brute force ...`



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 ...
```



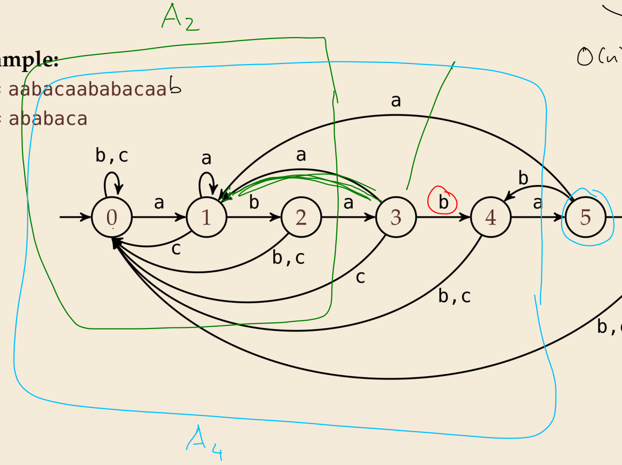
Ingenious algorithm by Knuth, Morris, and Pratt: construct DFA *inductively*:
Suppose we add character $P[j]$ to automaton A_{j-1} for $P[0..j)$

- 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)

Example:

T = aabacaababacaab

P = ababaca



$\delta(3, a) = 1$

get this by

$O(n)$ reading ~~a~~baa

without first letter

T

a b a

a b a b

a b a

a b a

a b

$5 \xrightarrow{c} 6$

$\delta(5, b) = 4$

$\delta(5, a) = 1$

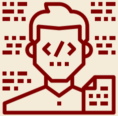
~~a~~babab

a

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 $P[0..j)$

- 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-1} for that!

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

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

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

Computing DFA efficiently

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

$\rightsquigarrow$ 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

$\rightsquigarrow$ simply maintain state x, the state after reading $P[1..j)$.

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

```
procedure constructDFA(P[0..m))
    // δ[q][c] = target state when reading c in state q
    for c ∈ Σ do
        δ[0][c] := 0
    δ[0][P[0]] := 1
    x := 0
    for j = 1, . . . , m − 1 do
        for c ∈ Σ do // copy transitions
            δ[j][c] := δ[x][c]
        δ[j][P[j]] := j + 1 // match edge
        x := δ[x][P[j]] // update x
```

Example: $P[0..6) =$ `ababac`

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

$x = 3$

Computing DFA efficiently

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

$\rightsquigarrow$ simply maintain state x, the state after reading $P[1..j)$.

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

```
procedure constructDFA(P[0..m))
    // δ[q][c] = target state when reading c in state q
    for c ∈ Σ do
        δ[0][c] := 0
    δ[0][P[0]] := 1
    x := 0
    for j = 1, ..., m − 1 do
        for c ∈ Σ do // copy transitions
            δ[j][c] := δ[x][c]
        δ[j][P[j]] := j + 1 // match edge
        x := δ[x][P[j]] // update x
```

Example: $P[0..6) =$ `ababac`

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

String matching with DFA – Discussion

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

Oct 2022

Unicode $\sigma = 149.186$

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

👍 **fast matching** time   actually: hard to beat!

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

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

4.5 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

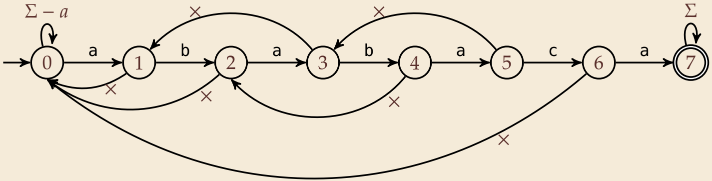
$\rightsquigarrow$ **KMP's third insight:** do this last step of simulation from state x during *matching*!
. . . but how?

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?

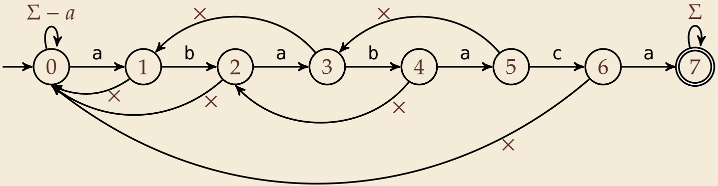
- **Answer:** Use a new type of transition, the *failure links*
  - Use this transition (only) if no other one fits.
  - $\times$ *does not consume a character.* ⇝ might follow several failure links



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

Failure link automaton – Example

Example: $T = \texttt{abababaaaca}$, $P = \texttt{ababaca}$



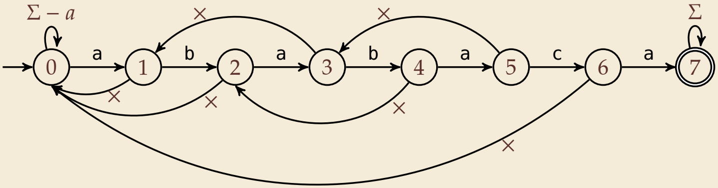
$T:$ a b a b a b a a b a b

1 2 3 4 5

3 4

Failure link automaton – Example

Example: T = ababababaaaca, P = ababaca



T :	a	b	a	b	a	b	a	a	b	a	b	
P :	a	b	a	b	a	×						to state 3
			(a)	(b)	(a)	b	a	×				to state 1
								a	b	a	b	

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 $\Theta(1)$

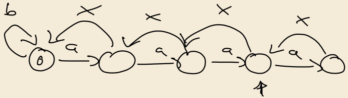
B $\Theta(\log m)$

C $\Theta(m)$

D $\Theta(m^2)$

→ sli.do/comp526

Clicker Question



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

A $\cancel{\Theta(1)}$

B $\cancel{\Theta(\log m)}$

C $\Theta(m)$ ✓

D $\cancel{\Theta(m^2)}$

→ sli.do/comp526

The Knuth-Morris-Pratt Algorithm

```
procedure KMP(T[0..n), P[0..m))
    fail[0..m] := failureLinks(P)
    i := 0 // current position in T
    q := 0 // current state of KMP automaton
    while i < n do
        if T[i] == P[q] then
            i := i + 1; q := q + 1
            if q == m then
                return i − q // occurrence found
        else // i.e. T[i] ≠ P[q]
            if q ≥ 1 then
                q := fail[q] // follow one ×
            else
                i := i + 1
    end while
    return NO_MATCH
```

- only need single array *fail* for failure links
- (procedure failureLinks later)

The Knuth-Morris-Pratt Algorithm

```
procedure KMP(T[0..n), P[0..m))
    fail[0..m] := failureLinks(P)
    i := 0 // current position in T
    q := 0 // current state of KMP automaton
    while i < n do
        if T[i] == P[q] then
            i := i + 1; q := q + 1
            if q == m then
                return i − q // occurrence found
        else // i.e. T[i] ≠ P[q]
            if q ≥ 1 then
                q := fail[q] // follow one ×
            else
                i := i + 1
    end while
    return NO_MATCH
```

$O(1)$

- only need single array *fail* for failure links
- (procedure failureLinks later)

Analysis: (matching part)

- always have $fail[j] < j$ for $j \geq 1$

$\rightsquigarrow$ in each iteration

- either advance position in text ($i := i + 1$) $\leq n$ steps
- or shift pattern forward (guess $i - q$) $\leq n$ steps

- each can happen at most n times

$\rightsquigarrow$ $\leq 2n$ symbol comparisons!

Computing failure links

- failure links point to error state x (from DFA construction)

$\rightsquigarrow$ run same algorithm, but store $\mathit{fail}[j] := x$ instead of copying all transitions

```
procedure failureLinks(P[0..m])
    fail[0] := 0
    x := 0
    for j := 1, ..., m − 1 do
        fail[j] := x
        // update failure state using failure links:
        while P[x] ≠ P[j]
            if x == 0 then
                x := −1;  break
            else
                x := fail[x]
        end while
        x := x + 1
    end for
```

Computing failure links

- failure links point to error state x (from DFA construction)

$\rightsquigarrow$ run same algorithm, but store $\mathit{fail}[j] := x$ instead of copying all transitions

```
procedure failureLinks(P[0..m])
    fail[0] := 0
    x := 0
    for j := 1, ..., m − 1 do
        fail[j] := x
        // update failure state using failure links:
        while P[x] ≠ P[j]
            if x == 0 then
                x := −1;   break
            else
                x := fail[x]  < x
        end while
        x := x + 1
    end for
```

Analysis:

- m iterations of for loop
- while loop always decrements x
- x is incremented only once per iteration of for loop

$\rightsquigarrow$ $\le m$ iterations of while loop *in total*

$\rightsquigarrow$ $\le 2m$ symbol comparisons

Knuth-Morris-Pratt – Discussion

- **Time:**
  - $\leq 2n + 2m = O(n + m)$ character comparisons
  - clearly must at least *read* both T and P
  - $\leadsto$ 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/comp526

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.

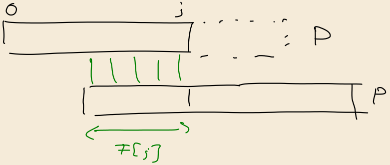
- **A** faster preprocessing on pattern ✓
- **B** ~~faster matching in text~~
- **C** ~~fewer character comparisons~~
- **D** uses less space ✓
- **E** makes running time independent of σ ✓
- **F** ~~I don't have to do automata theory~~

→ sli.do/comp526

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* $F : [1..m-1] \to [0..m-1]$:

  $F[j]$ 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 $j \ge 1$, and hence

$fail[j]$ = ***length*** *of the longest prefix of* $P[0..j)$ *that is a suffix of* $P[1..j)$.

memorize this!