

# CS 240 – Data Structures and Data Management

## Module 9: String Matching

Armin Jamshidpey, Éric Schost

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2026

# Outline

## 9 String Matching

- Introduction
- Suffix Trees
- Boyer-Moore Algorithm
- Karp-Rabin Algorithm
- Knuth-Morris-Pratt algorithm

# Outline

## 9 String Matching

- Introduction
- Suffix Trees
- Boyer-Moore Algorithm
- Karp-Rabin Algorithm
- Knuth-Morris-Pratt algorithm

# Pattern matching: Introduction

- Search for a string (pattern) in a large body of text. Useful for
  - ▶ Information Retrieval (text editors, search engines)
  - ▶ Bioinformatics
  - ▶ Data Mining

- $T[0..n-1]$  – The **text** (or **haystack**) being searched within

**Example:**  $T = \text{"Where is he?"}$

- $P[0..m-1]$  – The **pattern** (or **needle**) being searched for

**Example:**  $P_1 = \text{"he"}$        $P_2 = \text{"who"}$

- **occurrence:** index  $i$  such that  $T[i..i+m-1] = P$ , i.e.,

$$P[j] = T[i+j] \quad \text{for } 0 \leq j \leq m-1$$

- Convention: return smallest such  $i$  (leftmost occurrence)
- If  $P$  does not **occur** in  $T$ , return FAIL

# Pattern matching: Observation

## Recall:

- **Substring**  $T[i..j]$  for  $0 \leq i \leq j+1 \leq n$ : the string of length  $j - i + 1$  which consists of characters  $T[i], \dots, T[j]$  in order.
- **Prefix** of  $T$ : a substring  $T[0..i-1]$  of  $T$  for some  $0 \leq i \leq n$ .
- **Suffix** of  $T$ : a substring  $T[i..n-1]$  of  $T$  for some  $0 \leq i \leq n$ .
- The **empty string**  $\Lambda$  is also considered a substring, prefix and suffix.

# Brute-force algorithm

**Idea:** Consider every possible position where  $P$  might occur.

```
Bruteforce::pattern-matching( $T[0..n-1], P[0..m-1]$ )  
1. for  $g \leftarrow 0$  to  $n - m$  do           //  $g$ : index of guess  
2.     if strncmp( $T, P, g, m$ ) = 0  
3.         return "found at guess  $g$ "  
4. return FAIL
```

- A **guess** is a position  $g$  such that  $P$  might start at  $T[g]$ .  
Valid guesses (initially) are  $0 \leq g \leq n - m$ .
- We use *strncmp*( $T, P, g, m$ ) to compare a guess to  $P$ .
  - ▶ This compares  $m$  chars of  $T$  and  $P$ , starting at  $T[g]$ .
- Sometimes we break *strncmp* up into individual **checks**: Compare  $P[j]$  to  $T[i]$  for some  $0 \leq j < m$  and  $0 \leq i < n$ .
- *strncmp* uses  $m$  checks in the worst-case, but may use (many) fewer checks if there is a **mismatch**.

## Brute-force Algorithm: Example

We will diagram a single run of any pattern matching algorithm by a matrix of checks, where each row represents a single guess (shaded).

$P$ : a a a b

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

|   |   |   |   |   |   |   |   |   |   |  |
|---|---|---|---|---|---|---|---|---|---|--|
| a | a | a |   |   |   |   |   |   |   |  |
|   | a | a |   |   |   |   |   |   |   |  |
|   |   | a |   |   |   |   |   |   |   |  |
|   |   |   | a | a | a | b |   |   |   |  |
|   |   |   |   | a | a | a | b |   |   |  |
|   |   |   |   |   | a | a | a | b |   |  |
|   |   |   |   |   |   | a | a | a | b |  |

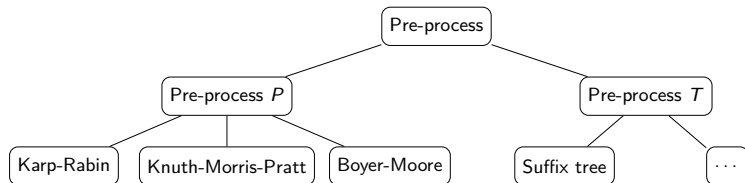
- What is the worst possible input?  $P = a^{m-1}b$ ,  $T = a^n$
- Worst case performance  $\Theta((n - m + 1) \cdot m)$
- This is too slow (quadratic if  $m \approx n/2$ ).

# How to improve?

General idea of **preprocessing**: Do work on some parts of input beforehand, so that the actual **query** (with rest of input) goes faster.

For pattern matching, we have two options:

- Do preprocessing on the pattern  $P$ 
  - ▶ We **eliminate guesses** based on characters we have seen.
- Do preprocessing on the text  $T$ 
  - ▶ We **create a data structure** to find matches easily.



Can be faster than brute-force even with only **one** query.

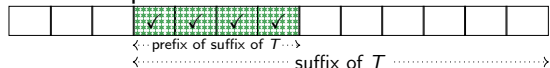
# Outline

## 9 String Matching

- Introduction
- **Suffix Trees**
- Boyer-Moore Algorithm
- Karp-Rabin Algorithm
- Knuth-Morris-Pratt algorithm

# Tries of suffixes and suffix trees

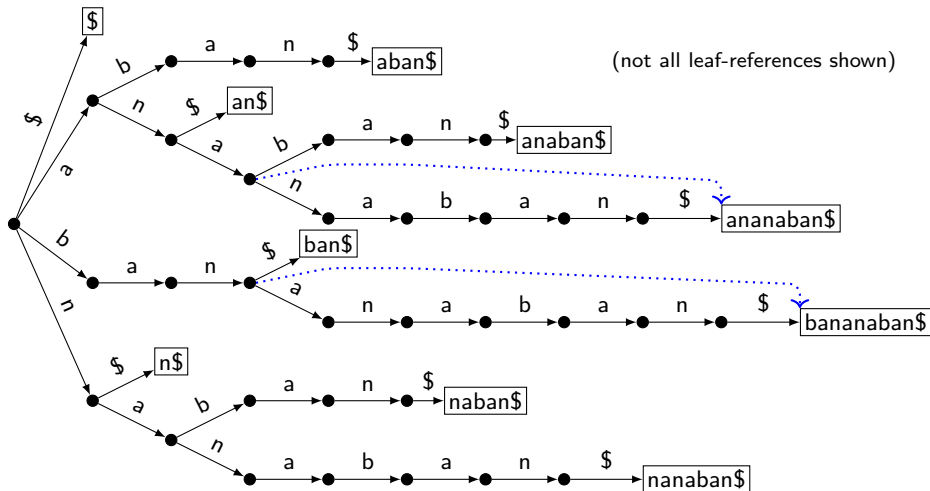
**Observation:**  $P$  occurs in  $T \Leftrightarrow P$  is a prefix of some suffix of  $T$ .



- **Idea:** Build a data structure that stores all suffixes of  $T$ .
  - ▶ So we preprocess the text  $T$  rather than the pattern  $P$
- **Naive idea:** Store the suffixes in a trie.
  - ▶  $|T| = n \Rightarrow$  the  $n+1$  suffixes together have  $\binom{n+1}{2} \in \Theta(n^2)$  characters
  - ▶ Wastes space by storing the suffixes at the leaves
- **Suffix tree** saves space in two ways:
  - ▶ Store suffixes implicitly via indices in  $T$ .
  - ▶ Use a compressed trie.
  - ▶ Then the space is  $O(n)$  since we store  $n+1$  words.

## Trie of suffixes: Example

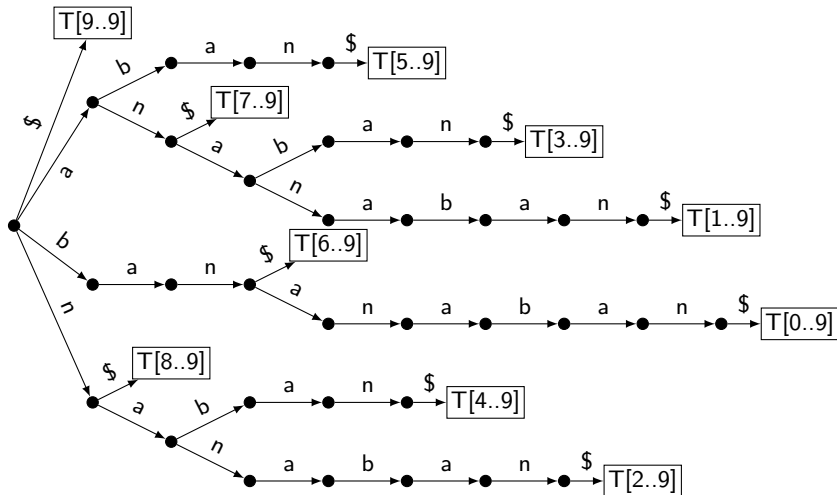
$T = \text{bananaban}\$$  has suffixes  $\{\text{bananaban}\$, \text{ananaban}\$, \text{nanaban}\$, \text{anaban}\$, \text{naban}\$, \text{aban}\$, \text{ban}\$, \text{an}\$, \text{n}\$, \$\}$



# Tries of suffixes: Improvement

Store suffixes via indices:

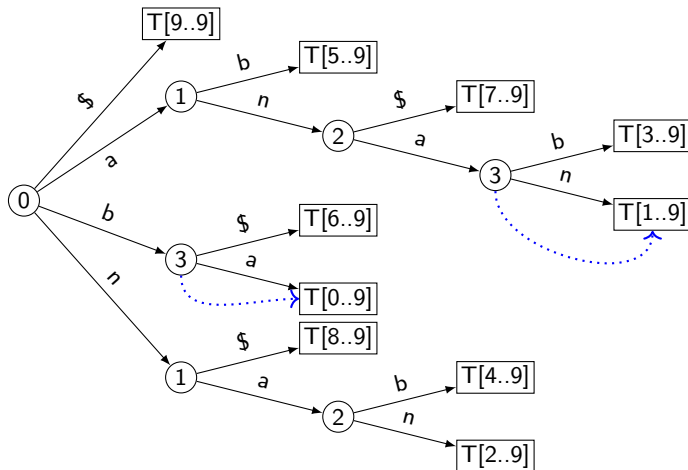
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  |
|---|---|---|---|---|---|---|---|---|----|
| b | a | n | a | n | a | b | a | n | \$ |



# Suffix tree

Compressed trie of suffixes where leaves store indices.

|       | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  |
|-------|---|---|---|---|---|---|---|---|---|----|
| $T =$ | b | a | n | a | n | a | b | a | n | \$ |



# Using suffix trees

## Pattern Matching:

- Want: “Is  $P$  a prefix of a word stored in suffix tree?”
- We know how to do this: *prefix-search* for  $P$ .
  - ▶ Recall: This needed leaf-references.
  - ▶ We defined leaf-references to use the leaf with the longest word.
  - ▶ Therefore this returns longest suffix with prefix  $P$ , hence leftmost occurrence.
- Run-time:  $O(\log(|\Sigma|)m)$  using fast dictionary for children,  $O(m)$  if we use more space and store child-links with direct addressing
- *Independent* of the text-length  $n$
- Great for repeated search in the same text

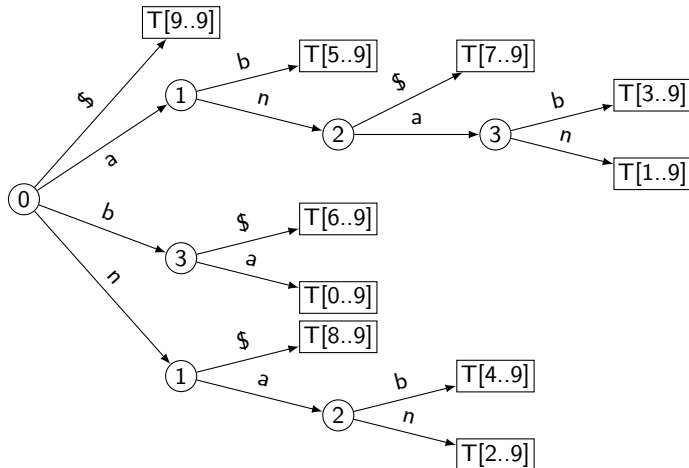
# Pattern matching in suffix trees: Example 1

$T =$ 

|   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  |
| b | a | n | a | n | a | b | a | n | \$ |

$P =$ 

|   |   |   |
|---|---|---|
| 0 | 1 | 2 |
| a | n | n |



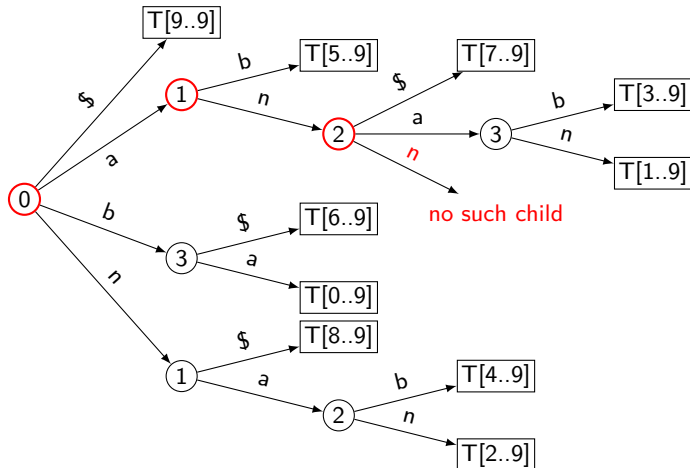
# Pattern matching in suffix trees: Example 1

$T =$ 

|   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  |
| b | a | n | a | n | a | b | a | n | \$ |

$P =$ 

|   |   |   |
|---|---|---|
| 0 | 1 | 2 |
| a | n | n |



If 'no such child' before we reach end of  $P$ : **FAIL**

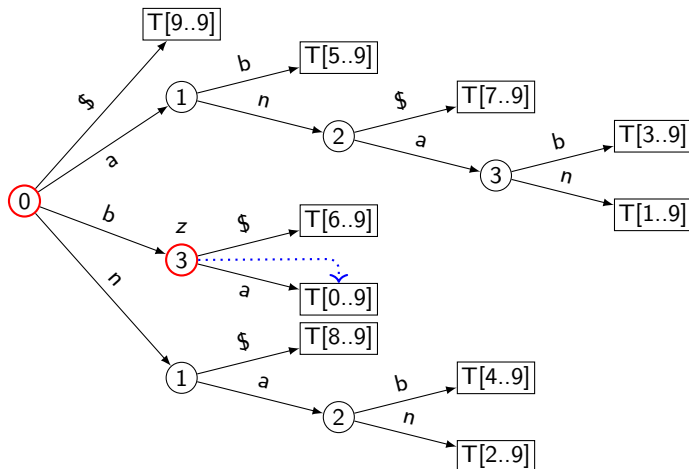
## Pattern matching in suffix trees: Example 2

$T =$ 

|   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  |
| b | a | n | a | n | a | b | a | n | \$ |

$P =$ 

|   |   |
|---|---|
| 0 | 1 |
| b | e |



If we reach node  $z$  at end of  $P$ : Compare  $P$  to  $z$ . *leaf*.

# Outline

## 9 String Matching

- Introduction
- Suffix Trees
- **Boyer-Moore Algorithm**
- Karp-Rabin Algorithm
- Knuth-Morris-Pratt algorithm

# Towards the Boyer-Moore algorithm

We do string-comparisons *backwards* (i.e., right-to-left), using characters mismatches to move the pattern

Forward-searching:

$P$ : g o o d

$T$ : g r a d i e n t

|   |   |  |  |  |  |  |  |
|---|---|--|--|--|--|--|--|
| g | o |  |  |  |  |  |  |
|   |   |  |  |  |  |  |  |

- $r$  does not occur in  $P$ .  
⇒ shift pattern past  $o$

At most  $j - 1$  guesses ruled out after  $j$  checks.

Reverse-order searching:

$P$ : g o o d

$T$ : g r a d i e n t

|  |  |   |   |  |  |  |  |
|--|--|---|---|--|--|--|--|
|  |  | o | d |  |  |  |  |
|  |  |   |   |  |  |  |  |

- $a$  does not occur in  $P$ .  
⇒ shift pattern past  $o$ .

Could rule out  $m - 1$  guesses even after only one check

- Reverse-order searching typically eliminates more guesses.
- Some characters of  $T$  were not compared at all.

## Last-occurrence heuristic

*P*: p a p e r

*T*: f e e d a l l p o o r p a r r o t s

|  |  |  |  |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  | r |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

## Last-occurrence heuristic

*P*: p a p e r

*T*: f e e d a l l p o o r p a r r o t s

|  |  |  |  |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  | r |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

Bad *T*-character is a. Shift the guess until a in *P* aligns with a in *T*

## Last-occurrence heuristic

*P*: p a p e r

*T*: f e e d a l l p o o r p a r r o t s

|  |  |  |  |   |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|---|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  | r |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|---|--|--|--|--|--|--|--|--|--|--|--|--|--|

Bad *T*-character is a. Shift the guess until a in *P* aligns with a in *T*

|  |  |  |     |  |  |   |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|-----|--|--|---|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  | (a) |  |  | r |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|-----|--|--|---|--|--|--|--|--|--|--|--|--|--|--|

## Last-occurrence heuristic

$P$ : p a p e r

$T$ : f e e d a l l **p** o o r p a r r o t s

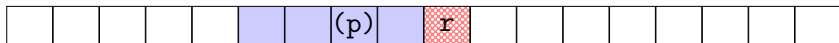


Bad  $T$ -character is **a**. Shift the guess until **a** in  $P$  aligns with **a** in  $T$



Shift the guess until **last p** in  $P$  aligns with bad  $T$ -character **p**.

Use “last” since we cannot rule out this guess.



## Last-occurrence heuristic

$P$ : p a p e r

$T$ : f e e d a l l p o o r p a r r o t s



Bad  $T$ -character is a. Shift the guess until a in  $P$  aligns with a in  $T$

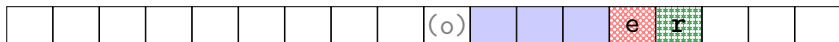


Shift the guess until last p in  $P$  aligns with bad  $T$ -character p.

Use “last” since we cannot rule out this guess.



Shift completely past o since o is not in  $P$ .



## Last-occurrence heuristic

$P$ : p a p e r

$T$ : f e e d a l l p o o r p a **r** r o t s



Bad  $T$ -character is **a**. Shift the guess until **a** in  $P$  aligns with **a** in  $T$

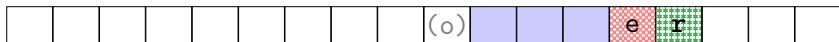


Shift the guess until **last** **p** in  $P$  aligns with bad  $T$ -character **p**.

Use “last” since we cannot rule out this guess.

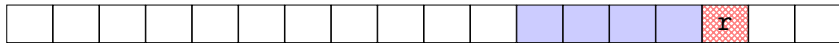


Shift completely past **o** since **o** is not in  $P$ .



The guess that aligns rightmost **r** of  $P$  has already been ruled out.

Simply shift one unit to the right.



# Helper-array for last-occurrence heuristic

- Build the helper-array  $L$  mapping  $\Sigma$  to integers
- $L[c]$  is the largest index  $i$  such that  $P[i] = c$

Pattern:

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| p | a | p | e | r |

Helper-array:

|            |     |     |     |     |            |
|------------|-----|-----|-----|-----|------------|
| char       | $p$ | $a$ | $e$ | $r$ | all others |
| $L[\cdot]$ | 2   | 1   | 3   | 4   | ?          |

- ▶ We will see: should set  $L[c] = -1$
- We can build this in time  $O(m + |\Sigma|)$  with simple for-loop

*BoyerMoore::last-occurrence-array*( $P[0..m-1]$ )

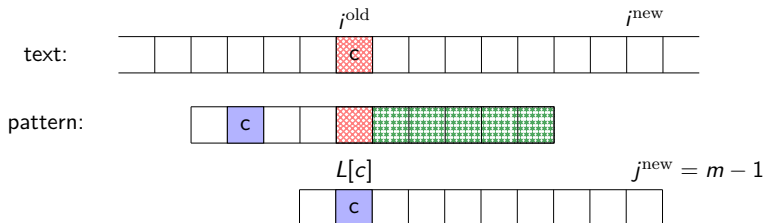
1. initialize array  $L$  indexed by  $\Sigma$  with all  $-1$
2. **for**  $j \leftarrow 0$  **to**  $m-1$  **do**  $L[P[j]] \leftarrow j$
3. **return**  $L$

# Update formula

Maintain two variables:

- $i$  = index of the character we look up in  $T$
- $j$  = index of the character we look up in  $P$

**Case 1:** Bad  $T$ -character  $c$  is in  $P$  and last occurrence left of  $j$



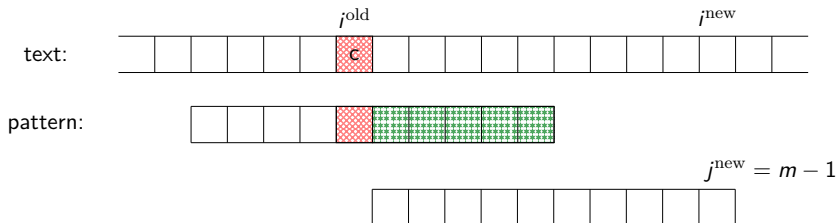
$$i^{\text{new}} - i^{\text{old}} = (m-1) - L[c] \text{ so } i^{\text{new}} = i^{\text{old}} + (m-1) - L[c]$$

# Update formula

Maintain two variables:

- $i$  = index of the character we look up in  $T$
- $j$  = index of the character we look up in  $P$

**Case 2:** Bad  $T$ -character  $c$  is not in  $P$



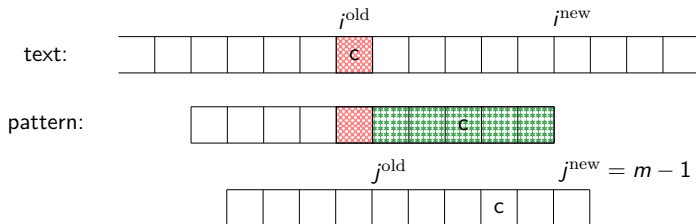
$i^{\text{new}} - i^{\text{old}} = m$  so  $i^{\text{new}} = i^{\text{old}} + (m-1) - L[c]$  if we set  $L[c] = -1$

# Update formula

Maintain two variables:

- $i$  = index of the character we look up in  $T$
- $j$  = index of the character we look up in  $P$

**Case 3:** Bad  $T$ -character  $c$  is in  $P$  and last occurrence right of  $j$



$$i^{\text{new}} - i^{\text{old}} = (m-1) - j^{\text{old}} + 1 \text{ so } i^{\text{new}} = i^{\text{old}} + (m-1) - (j^{\text{old}} - 1)$$

# Boyer-Moore Algorithm

```
Boyer-Moore::pattern-matching( $T, P$ )           // simplified version
1.  $L \leftarrow \textit{last-occurrence-array}(P)$ 
2.  $i \leftarrow m - 1, j \leftarrow m - 1$  // currently compare  $T[i]$  to  $P[j]$ 
3. while  $i < n$  do
4.     if  $P[j] = T[i]$ 
5.         if  $j = 0$  then return "found at guess  $i$ "
6.         else // go backwards
7.              $i \leftarrow i - 1, j \leftarrow j - 1$ 
8.     else
9.          $i \leftarrow i + m - 1 - \min\{L[T[i]], j - 1\}$  // unifies all cases
10.         $j \leftarrow m - 1$  // restart from right end
11. return FAIL
```

# Summary

- This is a simplified version of the Boyer-Moore algorithm
- Worst-case run-time  $O(mn)$
- In practice, works very well on English text  
(on typical English text, looks at only  $\approx 25\%$  of  $T$ )
- There are ways to ensure  $O(n)$  run-time.

# Outline

## 9 String Matching

- Introduction
- Suffix Trees
- Boyer-Moore Algorithm
- Karp-Rabin Algorithm
- Knuth-Morris-Pratt algorithm

# Karp-Rabin fingerprinting: Idea

**Idea:** Use **fingerprints** to eliminate guesses

- Need function  $h : \{\text{strings of length } m\} \rightarrow \{0, \dots, M-1\}$
- **Insight:** If  $h(P) \neq h(T[g..g+m-1])$  then guess  $g$  cannot work

$P$ : 9 2 6 5 3 has fingerprint 18

$T$ : 3 1 4 1 5 9 2 6 5 3 5

|                |                |                |                |                |                |  |  |  |  |  |
|----------------|----------------|----------------|----------------|----------------|----------------|--|--|--|--|--|
| fingerprint 84 |                |                |                |                |                |  |  |  |  |  |
|                | fingerprint 94 |                |                |                |                |  |  |  |  |  |
|                |                | fingerprint 76 |                |                |                |  |  |  |  |  |
|                |                |                | fingerprint 18 |                |                |  |  |  |  |  |
|                |                |                |                | fingerprint 95 |                |  |  |  |  |  |
|                |                |                |                |                | fingerprint 18 |  |  |  |  |  |

no *strcmp* needed

no *strcmp* needed

no *strcmp* needed

false positive

no *strcmp* needed

found

- Normally  $h$  is standard hash-function for words; we can choose  $M$ .
- In example:  $|\Sigma| = 10$ ,  $h(x_0 \dots x_4) = (x_0 x_1 x_2 x_3 x_4)_{10} \bmod 97$

## Karp-Rabin fingerprinting: First attempt

*Karp-Rabin-Simple::pattern-matching*( $T, P$ )

1. choose fingerprint function  $h$
2.  $h_P \leftarrow h(P[0..m-1])$  // end of pre-computation
3. **for**  $g \leftarrow 0$  to  $n - m$
4.      $h_T \leftarrow h(T[g..g+m-1])$  // compute fingerprint of guess
5.     **if**  $h_T = h_P$
6.         **if** *strncmp*( $T, P, g, m$ ) = 0 **then return** “found at guess  $g$ ”
7. **return** FAIL

- never misses a match
- **Problem:** line 4 is *not* constant time. Naively,  $\Omega(m)$  time per guess
- no better than brute-force

## Karp-Rabin fingerprinting: Fast update

**Idea:** Consecutive guesses share  $m-1$  characters

⇒ for suitable hash-functions, can compute next fingerprint from previous

|       |   |   |   |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|---|---|---|
| $T :$ | 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 |
|       |   |   |   |   |   |   |   |   |   |   |   |
|       |   |   |   |   |   |   |   |   |   |   |   |

**Example:**  $15926 = (41592 - 4 \cdot 10\,000) \cdot 10 + 6$

$$\begin{aligned}\underbrace{15926 \bmod 97}_{h(15926)} &= \left( \underbrace{(41592 \bmod 97)}_{\text{previous fingerprint}} - 4 \cdot \underbrace{10000 \bmod 97}_9 \text{ (pre-computed)} \right) \cdot 10 + 6 \bmod 97 \\ &= ((76 - 4 \cdot 9) \cdot 10 + 6) \bmod 97 = 18\end{aligned}$$

Next fingerprint can be computed in  $O(1)$  time after preprocessing.

# Karp-Rabin fingerprinting: Conclusion

*Karp-Rabin::pattern-matching*( $T, P$ ) // rolling hash-function

1. Choose  $M$ , radix  $R$  (for standard word-hash-function  $h$ )
2.  $h_P \leftarrow h(P[0..m-1])$
3.  $s \leftarrow R^{m-1} \bmod M$  // end of precomputation
4.  $h_T \leftarrow h(T[0..m-1])$  // leftmost fingerprint
5. **for**  $g \leftarrow 0$  to  $n - m$
6.     **if**  $h_T = h_P$
7.         **if** *strncmp*( $T, P, g, m$ ) = 0 **then return** “found at guess  $g$ ”
8.     **if**  $g < n - m$  **then**  $h_T \leftarrow ((h_T - T[g] \cdot s) \cdot R + T[g+m]) \bmod M$
9. **return** “FAIL”

- Run-time:  $O(m + n)$  operations modulo  $M$  and  $O(m \cdot (1 + \#\{\text{false positives}\}))$  for string comparisons
- Choose “table size”  $M$  to be **random** prime in  $\{2, \dots, mn^2\}$
- Can show (no details):  $Pr(\text{at least one false positive}) \in O(\frac{1}{n})$
- Then expected time for string comparisons is  $O(m)$

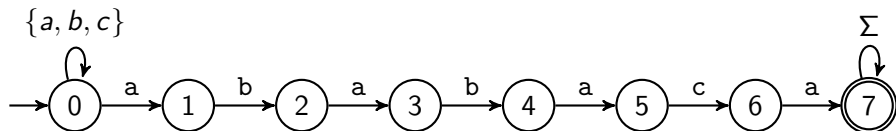
# Outline

## 9 String Matching

- Introduction
- Suffix Trees
- Boyer-Moore Algorithm
- Karp-Rabin Algorithm
- Knuth-Morris-Pratt algorithm

# Pattern matching with Finite Automata

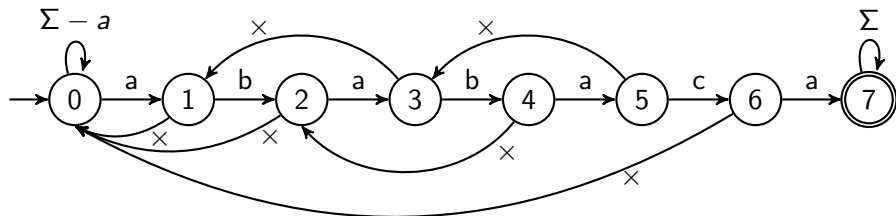
**Example:** Automaton for the pattern  $P = \text{ababaca}$



- This is a **N**on-deterministic **F**inite **A**utomaton
- **Forward-arc**  $(j) \longrightarrow (j+1)$  labelled with  $P[j]$
- State  $j$  expresses “we have  $j$  leftmost characters of  $P$ ”
- NFA accepts  $T$  if and only if  $T$  contains  $P$

But evaluating NFAs is very slow.

# The Knuth-Morris-Pratt automaton

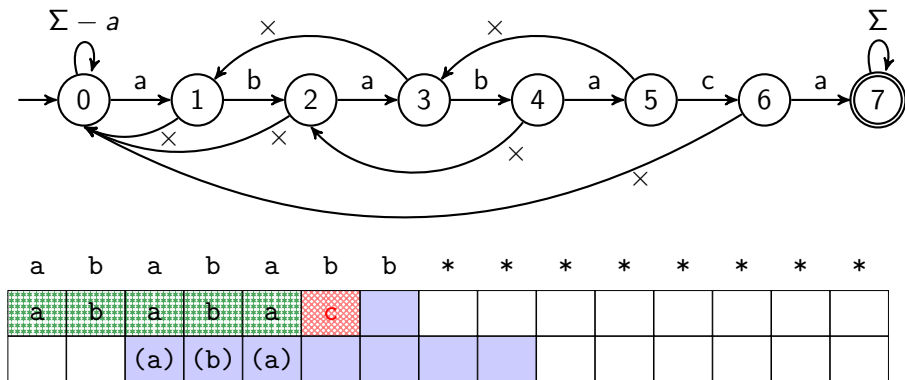


- Same states, forward-arcs, start state, accepting state
- Use a new type of transition  $\times$  (“*failure*”) but deterministic:
  - ▶ One per state  $1, \dots, m-1$ , use if the other transition is not possible.
  - ▶ Does **not** consume a character.
- We will later determine failure-arcs such that the automaton accepts  $T$  if and only if  $T$  contains ababaca

# Search for $P = \text{ababaca}$ in $T = \text{abababb} \dots$

Maintain two variables:

- $j$  = state = length of match = index of the character we look up in  $P$
- $i$  = index of the character we look up in  $T$

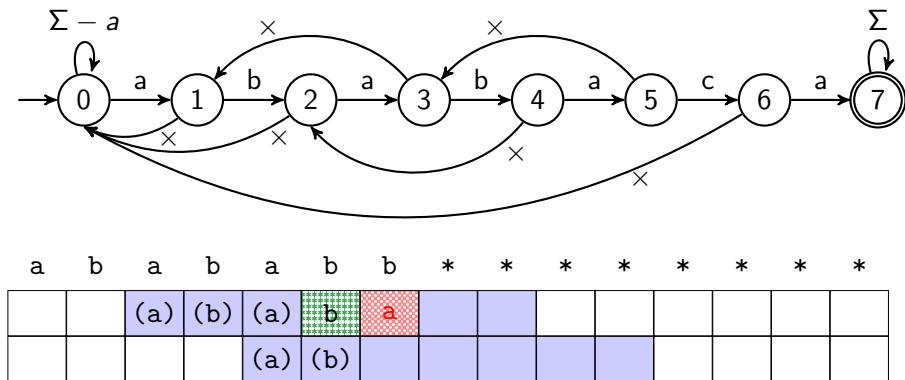


Mismatch at  $j = 5$ . In new guess, three matched characters:  $j^{\text{new}} = 3$ .  
We will next check the same character of  $T$ :  $i^{\text{new}} = i$ .

# Search for $P = \text{ababaca}$ in $T = \text{abababb} \dots$

Maintain two variables:

- $j$  = state = length of match = index of the character we look up in  $P$
- $i$  = index of the character we look up in  $T$

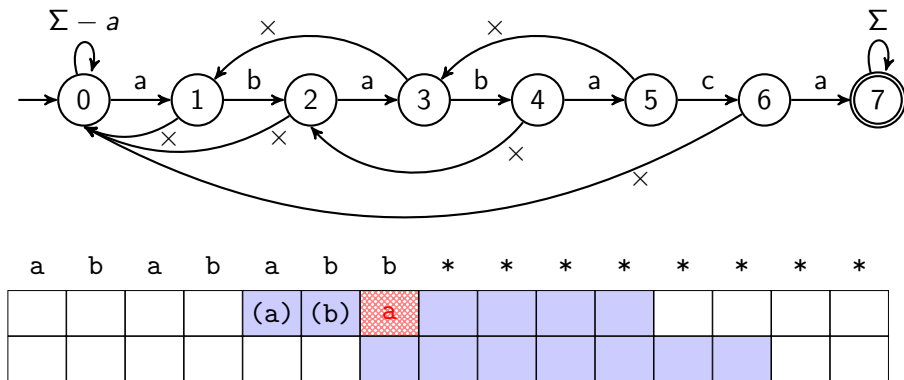


Mismatch at  $j = 4$ . In new guess, two matched characters:  $j^{\text{new}} = 2$ .  
We will next check the same character of  $T$ :  $i^{\text{new}} = i$ .

# Search for $P = \text{ababaca}$ in $T = \text{abababb} \dots$

Maintain two variables:

- $j$  = state = length of match = index of the character we look up in  $P$
- $i$  = index of the character we look up in  $T$



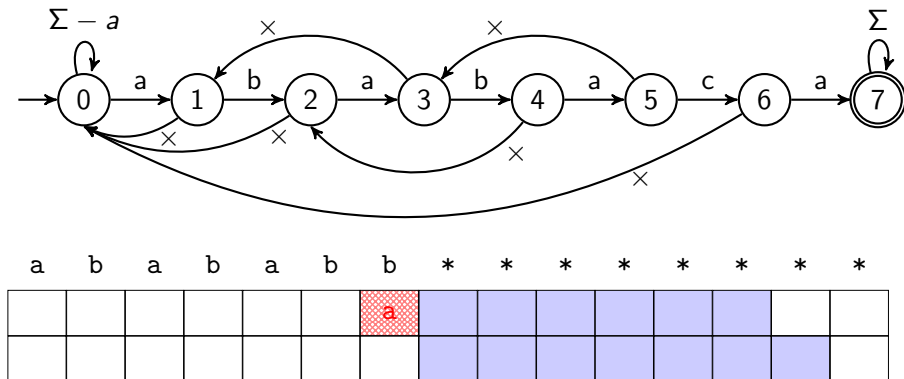
Mismatch at  $j = 2$ . In new guess, no matched characters:  $j^{\text{new}} = 0$

We will next check the same character of  $T$ :  $i^{\text{new}} = i$ .

# Search for $P = \text{ababaca}$ in $T = \text{abababb} \dots$

Maintain two variables:

- $j$  = state = length of match = index of the character we look up in  $P$
- $i$  = index of the character we look up in  $T$



Match is empty ( $j = 0$ ), shift the guess forward by 1

$$j^{\text{new}} = 0, i^{\text{new}} = i + 1.$$

# Knuth-Morris-Pratt algorithm

Store transitions in an array:  $f[j] = j^{\text{new}}$  to use if we mismatch at  $j$

|        |    |   |   |   |   |   |   |
|--------|----|---|---|---|---|---|---|
| $j$    | 0  | 1 | 2 | 3 | 4 | 5 | 6 |
| $f[j]$ | NA | 0 | 0 | 1 | 2 | 3 | 0 |

*KMP::pattern-matching*( $T, P$ )

```
1.  $f \leftarrow \text{compute-failure-array}(P)$ 
2.  $i \leftarrow 0, j \leftarrow 0$            // currently compare  $T[i]$  to  $P[j]$ 
3. while  $i < n$  do
4.   // inv:  $P[0..j-1]$  is a suffix of  $T[0..i-1]$ 
5.   if  $P[j] = T[i]$ 
6.     if  $j = m - 1$  then return "found at guess  $i - m + 1$ "
7.     else                               // check next character
8.        $i \leftarrow i + 1, j \leftarrow j + 1$ 
9.   else                               // good prefix is  $P[0..j-1]$ 
10.    if  $j = 0$  then  $i \leftarrow i + 1$ 
11.    else  $j \leftarrow f[j]$ 
```

**Warning:** the book works with an array  $F$  shifted by 1:  $f[j] = F[j - 1]$   
we don't need this complication

# Runtime

Assume we have precomputed the failure array.

- run-time is proportional to the number of arcs followed.
- every loop at 0 and forward-arc consumes a character of  $T$ , so this happens at most  $n$  times
- last state is (number of forward steps)  $- \sum$  lengths of back steps, so

$$0 \leq (\text{number of forward steps}) - \sum \text{lengths of back steps}$$

and

$$\text{num. back steps} \leq \sum \text{lengths of back steps} \leq \text{num. forward steps} \leq n$$

**Total:**  $O(n)$  worst case

## Computing the failure array

- $f[j] =$  number of re-used characters if good prefix was  $P[0..j - 1]$

|   |     |     |     |     |   |   |   |   |   |   |   |   |   |   |
|---|-----|-----|-----|-----|---|---|---|---|---|---|---|---|---|---|
| a | b   | a   | b   | a   | b | b | * | * | * | * | * | * | * | * |
| a | b   | a   | b   | a   | c |   |   |   |   |   |   |   |   |   |
|   | (a) | (b) | (a) | (b) |   |   |   |   |   |   |   |   |   |   |
|   |     | (a) | (b) | (a) |   |   |   |   |   |   |   |   |   |   |
|   |     |     | (a) | (b) |   |   |   |   |   |   |   |   |   |   |
|   |     |     |     | (a) |   |   |   |   |   |   |   |   |   |   |
|   |     |     |     |     |   |   |   |   |   |   |   |   |   |   |

- We must find a prefix of  $P$  that is a suffix of  $P[0..j - 1]$
- It should not be **all** of  $P[0..j - 1]$ , so it is a suffix of  $P[1..j - 1]$
- To be sure we don't miss any match, take the **longest** one

**Result:**  $f[j] =$  length of the longest prefix of  $P$  that is a suffix of  $P[1..j - 1]$ .

# Runtime

$f[j]$  = length of the longest prefix of  $P$  that is a suffix of  $P[1..j - 1]$ .

- Write down  $P[1..j - 1]$  for all  $j$ , and all prefixes of  $P$ .
- Check whether each prefix is a suffix of  $P[1..j - 1]$  and take the longest.

| $j$ | $P[1..j - 1]$ | Prefixes of $P$ |   |    |     |      |     | longest   | $f[j]$ |
|-----|---------------|-----------------|---|----|-----|------|-----|-----------|--------|
|     |               | $\Lambda$       | a | ab | aba | abab | ... |           |        |
| 1   | $\Lambda$     | ✓               | × | ×  | ×   | ×    | ... | $\Lambda$ | 0      |
| 2   | b             | ✓               | × | ×  | ×   | ×    | ... | $\Lambda$ | 0      |
| 3   | ba            | ✓               | ✓ | ×  | ×   | ×    | ... | a         | 1      |
| 4   | bab           | ✓               | × | ✓  | ×   | ×    | ... | ab        | 2      |
| 5   | baba          | ✓               | ✓ | ×  | ✓   | ×    | ... | aba       | 3      |
| 6   | babac         | ✓               | × | ×  | ×   | ×    | ... | $\Lambda$ | 0      |

**Runtime:**  $O(m^3)$  time. Can be improved to  $O(m)$  (difficult)

# Summary

## Total:

- KMP main routine takes  $O(n)$  time
- *compute-failure-array* takes  $O(m^3)$  time, can be reduced to  $O(m)$ .

**Result:** KMP pattern matching has  $O(n + m)$  *worst-case* run-time

- This is  $O(n)$  if  $m \leq n$
- Auxiliary space is  $O(m)$
- Time and space bounds **independent** of  $|\Sigma|$
- Faster than brute-force even with only one query