

# CS 240 – Data Structures and Data Management

## Module 9: String Matching

A. Jamshidpey G. Kamath É. Schost

Based on lecture notes by many previous cs240 instructors

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

Spring 2020

References: Goodrich & Tamassia 9.1

# Outline

- 1 String Matching
  - Introduction
  - Rabin-Karp Algorithm
  - Boyer-Moore Algorithm
  - String Matching with Finite Automata
  - Knuth-Morris-Pratt algorithm
  - Suffix Trees
  - Conclusion

# Outline

- 1 String Matching
  - Introduction
  - Rabin-Karp Algorithm
  - Boyer-Moore Algorithm
  - String Matching with Finite Automata
  - Knuth-Morris-Pratt algorithm
  - Suffix Trees
  - Conclusion

# Pattern Matching Definition [1]

- Search for a string (pattern) in a large body of text
- $T[0..n - 1]$  – The **text** (or **haystack**) being searched within
- $P[0..m - 1]$  – The **pattern** (or **needle**) being searched for
- Strings over **alphabet**  $\Sigma$
- Return the first  $i$  such that

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

- This is the first **occurrence** of  $P$  in  $T$
- If  $P$  does not **occur** in  $T$ , return FAIL
- Applications:
  - ▶ Information Retrieval (text editors, search engines)
  - ▶ Bioinformatics
  - ▶ Data Mining

## Pattern Matching Definition [2]

Example:

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

Definitions:

- **Substring**  $T[i..j]$   $0 \leq i \leq j < n$ : a string of length  $j - i + 1$  which consists of characters  $T[i], \dots, T[j]$  in order
- A **prefix** of  $T$ :  
a substring  $T[0..i]$  of  $T$  for some  $0 \leq i < n$
- A **suffix** of  $T$ :  
a substring  $T[i..n - 1]$  of  $T$  for some  $0 \leq i \leq n - 1$

# General Idea of Algorithms

Pattern matching algorithms consist of **guesses** and **checks**:

- A **guess** or **shift** is a position  $i$  such that  $P$  might start at  $T[i]$ . Valid guesses (initially) are  $0 \leq i \leq n - m$ .
- A **check** of a guess is a single position  $j$  with  $0 \leq j < m$  where we compare  $T[i + j]$  to  $P[j]$ . We must perform  $m$  checks of a single **correct** guess, but may make (many) fewer checks of an **incorrect** guess.

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

# Brute-force Algorithm

**Idea:** Check every possible guess.

```
BruteforcePM( $T[0..n - 1]$ ,  $P[0..m - 1]$ )
```

$T$ : String of length  $n$  (text),  $P$ : String of length  $m$  (pattern)

```
1.  for  $i \leftarrow 0$  to  $n - m$  do  
2.      for  $j \leftarrow 0$  to  $m - 1$  do  
3.          if  $T[i + j] \neq P[j]$  then  
4.              break  
5.      if  $j = m$  then  
6.          return  $i$   
7.  return FAIL
```

# Brute-Force Example

- Example:  $T = \text{abbbababbab}$ ,  $P = \text{abba}$

	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		

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



# How to improve?

## More sophisticated algorithms

- Do extra **preprocessing** on the pattern  $P$ 
  - ▶ **Rabin-Karp**
  - ▶ **Boyer-Moore**
  - ▶ Deterministic finite automata (**DFA**), **KMP**
  - ▶ We **eliminate guesses** based on completed matches and mismatches.
- Do extra **preprocessing** on the text  $T$ 
  - ▶ **Suffix-trees**
  - ▶ We **create a data structure** to find matches easily.

# Outline

## 1 String Matching

- Introduction
- Rabin-Karp Algorithm
- Boyer-Moore Algorithm
- String Matching with Finite Automata
- Knuth-Morris-Pratt algorithm
- Suffix Trees
- Conclusion

# Rabin-Karp Fingerprint Algorithm – Idea

**Idea:** use hashing to eliminate guesses

- Compute hash function for each guess, compare with pattern hash
- If values are unequal, then the guess cannot be an occurrence
- Example:  $P = 5\ 9\ 2\ 6\ 5$ ,  $T = 3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5\ 3\ 5$ 
  - ▶ Use standard hash-function: flattening + modular (radix  $R = 10$ ):

$$h(x_0 \dots x_4) = (x_0x_1x_2x_3x_4)_{10} \bmod 97$$

- ▶  $h(P) = 59265 \bmod 97 = 95$ .

3	1	4	1	5	9	2	6	5	3	5
hash-value 84										
	hash-value 94									
		hash-value 76								
			hash-value 18							
			hash-value 95							

# Rabin-Karp Fingerprint Algorithm – First Attempt

*Rabin-Karp-Simple*( $T, P$ )

1.  $h_P \leftarrow h(P[0..m-1])$
2. **for**  $k \leftarrow 0$  to  $n - m$
3.      $h_T \leftarrow h(T[k..k+m-1])$
4.     **if**  $h_T = h_P$  **then**
5.         **if**  $T[k..k+m-1] = P$
6.             **return** “found at shift  $k$ ”
7.     **return** “not found”

- Never misses a match since  $h(k_1) \neq h(k_2)$  implies  $k_1 \neq k_2$
- $h(T[k..k+m-1])$  depends on  $m$  characters, so naive computation takes  $\Theta(m)$  time per shift
- Running time is  $\Theta(mn)$  for search miss (how can we improve this?)

# Rabin-Karp Fingerprint Algorithm – Fast Rehash

The initial hashes are called **fingerprints**.

Crucial insight: We can update these fingerprints in constant time.

- Use previous hash to compute next hash
- $O(1)$  time per hash, except first one

## Example:

- Pre-compute:  $10000 \bmod 97 = 9$
- Previous hash:  $41592 \bmod 97 = 76$
- Next hash:  $15926 \bmod 97 = ??$

## Observation:

$$\begin{aligned} 15926 \bmod 97 &= (41592 - (4 \cdot 10000)) \cdot 10 + 6 \bmod 97 \\ &= (76 - (4 \cdot 9)) \cdot 10 + 6 \bmod 97 \\ &= 406 \bmod 97 = 18 \end{aligned}$$

## Rabin-Karp Fingerprint Algorithm – Conclusion

*Rabin-Karp-Fast*( $T, P$ )

1.  $M \leftarrow$  suitable prime number
2.  $h_P \leftarrow h(P[0..m-1])$
3.  $h_T \leftarrow h(T[0..m-1])$
4.  $s \leftarrow 10^{m-1} \bmod M$
5. **for**  $k \leftarrow 0$  to  $n - m$
6.     **if**  $h_T = h_P$  **then**
7.         **if**  $T[k..k+m-1] = P$
8.             **return** “found at shift  $k$ ”
9.     **if**  $k < n - m$  **then**
10.          $h_T \leftarrow ((h_T - T[k] \cdot s) \cdot 10 + T[k+m]) \bmod M$
11. **return** “not found”

- Choose “table size”  $M$  at **random** to be huge prime
- Expected running time is  $O(m + n)$
- $\Theta(mn)$  worst-case, but this is (unbelievably) unlikely
- Extends to 2d patterns and other generalizations

# Outline

## 1 String Matching

- Introduction
- Rabin-Karp Algorithm
- **Boyer-Moore Algorithm**
- String Matching with Finite Automata
- Knuth-Morris-Pratt algorithm
- Suffix Trees
- Conclusion

# Boyer-Moore Algorithm

Brute-force search with two changes:

- **Reverse-order searching:** Compare  $P$  with a guess moving backwards
- **Bad character jumps:** When a mismatch occurs, then eliminate guesses where  $P$  does not agree with this char of  $T$
- In practice large parts of  $T$  will not be looked at.

```
BoyerMoore(T,P)
1.    $L \leftarrow$  last occurrence array computed from  $P$ 
2.    $k \leftarrow 0$  // current guess
3.   while  $k \leq n - m$ 
4.       for ( $j \leftarrow m - 1, j \geq 0, j --$ )
5.           if  $T[k + j] \neq P[j]$  break
6.           if  $j = -1$  return  $k$ 
7.           else
8.                $k \leftarrow k + \max\{1, j - L[T[k + j]]\}$ 
9.   return FAIL
```

$L$  will be explained below.



# Bad character heuristic

$P$ : p a t i k i

$T$ : R u m a k i i n H a w a i i !

			i	k	i										
							i								
													k	i	
															i

Shift to where 'a' fits

'n'  $\notin P \Rightarrow$  shift past 'n'

'i' does not eliminate shifts

$P$  not in  $T$

- Build the **last-occurrence function**  $L$  mapping  $\Sigma$  to integers
- $L(c)$  is defined as
  - ▶ the largest index  $i$  such that  $P[i] = c$  or
  - ▶  $-1$  if no such index exists

$c$	p	a	t	i	k	all others
$L(c)$	0	1	2	5	4	-1

- At first mismatch:

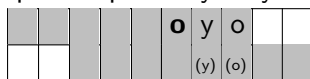
$k = 0, j = 3, T[k+j] = 'a', L[T[k+j]] = 1, \text{ so } k \leftarrow k + 3 - 1$

# Boyer-Moore algorithm conclusion

- On typical **English text** the algorithm probes approximately 25% of the characters in  $T$ . Fastest in practice.
- Worst-case run-time with only bad-character heuristic is  $\Theta(mn + |\Sigma|)$ .
- Worst-case run-time can be reduced to  $\Theta(n + m + |\Sigma|)$  with *good-suffix heuristic*:

$P$ : y o u r y o y o

$T$ : p u r p l e y o y o



Shift to where 'yo' fits

We will not give the details of this.

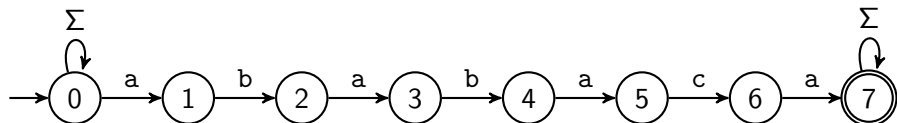
- In practice bad-character-heuristic alone is good enough.

# Outline

- 1 String Matching
  - Introduction
  - Rabin-Karp Algorithm
  - Boyer-Moore Algorithm
  - **String Matching with Finite Automata**
  - Knuth-Morris-Pratt algorithm
  - Suffix Trees
  - Conclusion

# String Matching with Finite Automata

**Example:** Automaton for the pattern  $P = ababaca$



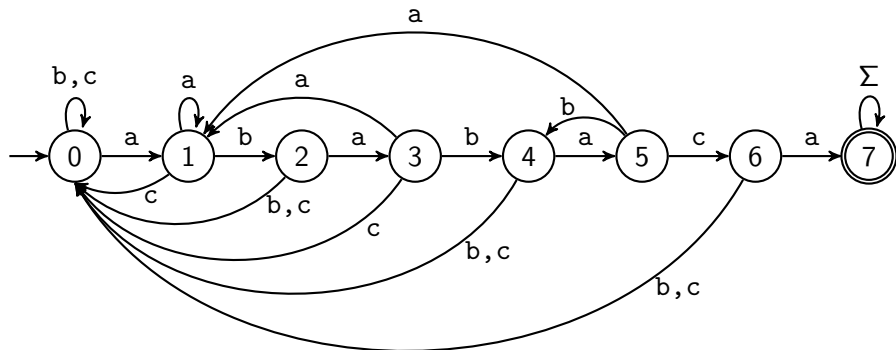
You should be familiar with:

- finite automaton, DFA, NFA, converting NFA to DFA
- transition function  $\delta$ , states  $Q$ , accepting states  $F$

- The above finite automation is an **NFA**
- State  $q$  expresses “we have seen  $P[0..q-1]$ ”
  - ▶ NFA accepts  $T$  if and only if  $T$  contains  $ababaca$
  - ▶ But evaluating NFAs is very slow.

# String matching with DFA

Can show: There exists an equivalent small DFA.



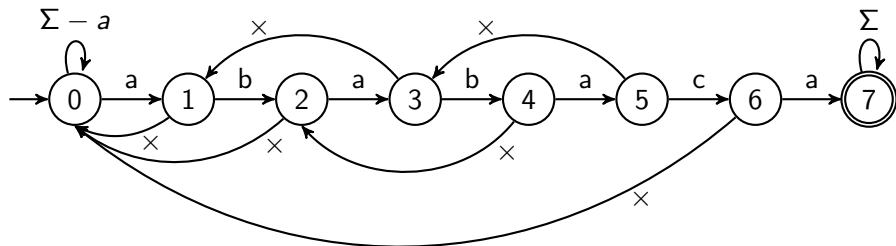
- Easy to test whether  $P$  is in  $T$ .
- But how do we find the arcs?
- We will not give the details of this since there is an even better automaton.

# Outline

## 1 String Matching

- Introduction
- Rabin-Karp Algorithm
- Boyer-Moore Algorithm
- String Matching with Finite Automata
- **Knuth-Morris-Pratt algorithm**
- Suffix Trees
- Conclusion

# Knuth-Morris-Pratt Motivation



- Use a new type of transition  $\times$  (“failure”):
  - ▶ Use this transition only if no other fits.
  - ▶ Does **not** consume a character.
  - ▶ With these rules, computations of the automaton are deterministic. (But it is formally not a valid DFA.)
- Can store **failure-function** in an array  $F[0..m-1]$ 
  - ▶ The failure arc from state  $j$  leads to  $F[j-1]$
- Given the failure-array, we can easily test whether  $P$  is in  $T$ :  
Automaton accepts  $T$  if and only if  $T$  contains ababaca

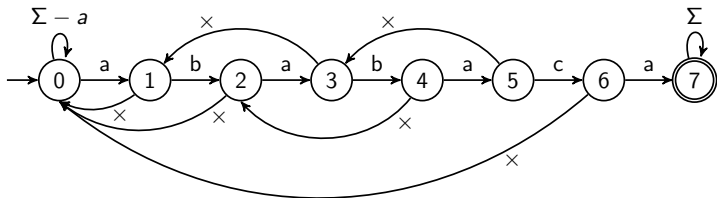
# Knuth-Morris-Pratt Algorithm

```
KMP(T, P), to return the first match
T: String of length n (text), P: String of length m (pattern)
1.  F ← failureArray(P)
2.  i ← 0 // current character of T to parse
3.  j ← 0 // current state that we are in
4.  while i < n do
5.      if P[j] = T[i] then
6.          if j = m - 1 then
7.              return i - m + 1 //match
8.          else
9.              i ← i + 1
10.             j ← j + 1
11.         else // i.e. P[j] ≠ T[i]
12.             if j > 0 then
13.                 j ← F[j - 1]
14.             else
15.                 i ← i + 1
16.         return FAIL // no match
```



# String matching with KMP – Example

Example:  $T = \text{ababababaca}$ ,  $P = \text{ababaca}$



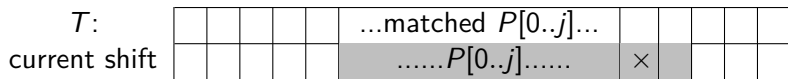
$T$ :	a	b	a	b	a	b	a	b	a	c	a	
$P$ :	a	b	a	b	a	×						to state 3
			(a)	(b)	(a)	b	a	×				to state 3
					(a)	(b)	(a)	b	a	c	a	

$q$ :	1	2	3	4	5	3,4	5	3,4	5	6	7
-------	---	---	---	---	---	-----	---	-----	---	---	---

(after reading this character)

# String matching with KMP – Failure-function

Assume we reach state  $j+1$  and now have mismatch.



- Can eliminate “shift by 1” if  $P[1..j] \neq P[0..j-1]$ .
- Can eliminate “shift by 2” if  $P[1..j]$  does not end with  $P[0..j-2]$ .
- Generally eliminate shift if that prefix of  $P$  is not a suffix of  $P[1..j]$ .
- So want longest prefix  $P[0..\ell-1]$  that is a suffix of  $P[1..j]$ .
- The  $\ell$  characters of this prefix are matched, so go to state  $\ell$ .

$F[j]$  = head of failure-arc from state  $j+1$   
= length of the longest prefix of  $P$  that is a suffix of  $P[1..j]$ .

## KMP Failure Array – Example

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

Consider  $P = \text{ababaca}$

$j$	$P[1..j]$	Prefixes of $P$	longest	$F[j]$
0	$\Lambda$	$\Lambda, a, ab, aba, abab, ababa, \dots$	$\Lambda$	0
1	b	$\Lambda, a, ab, aba, abab, ababa, \dots$	$\Lambda$	0
2	ba	$\Lambda, a, ab, aba, abab, ababa, \dots$	a	1
3	bab	$\Lambda, a, ab, aba, abab, ababa, \dots$	ab	2
4	baba	$\Lambda, a, ab, aba, abab, ababa, \dots$	aba	3
5	babac	$\Lambda, a, ab, aba, abab, ababa, \dots$	$\Lambda$	0
6	babaca	$\Lambda, a, ab, aba, abab, ababa, \dots$	a	1

This can clearly be computed in  $O(m^3)$  time, but we can do better!

# Outline

## 1 String Matching

- Introduction
- Rabin-Karp Algorithm
- Boyer-Moore Algorithm
- String Matching with Finite Automata
- Knuth-Morris-Pratt algorithm
- **Suffix Trees**
- Conclusion

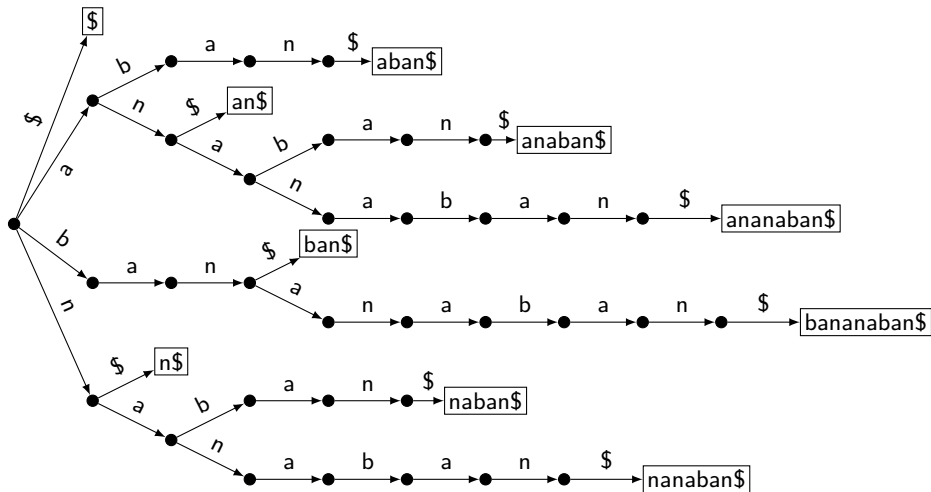
# Tries of Suffixes and Suffix Trees

- What if we want to search for **many patterns**  $P$  within the same **fixed text**  $T$ ?
- Idea: Preprocess the text  $T$  rather than the pattern  $P$
- Observation:  $P$  is a substring of  $T$  if and only if  $P$  is a prefix of some suffix of  $T$ .
- So want to store all suffixes of  $T$  in a trie.
- To save space:
  - ▶ Use a compressed trie.
  - ▶ Store suffixes implicitly via indices into  $T$ .
- This is called a **suffix tree**.

# Trie of suffixes: Example

$T = \text{bananaban}$  has suffixes

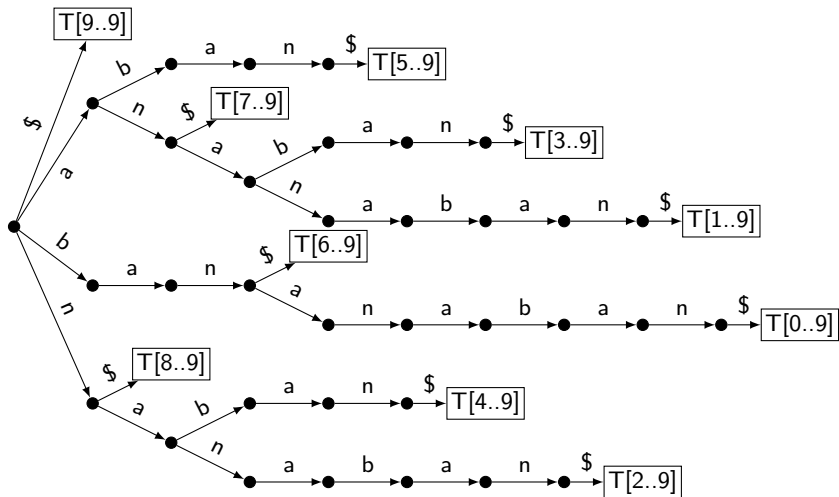
$\{\text{bananaban}, \text{ananaban}, \text{nanaban}, \text{anaban}, \text{naban}, \text{aban}, \text{ban}, \text{an}, \text{n}, \Lambda\}$



# Tries of suffixes

Store suffixes via indices:

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

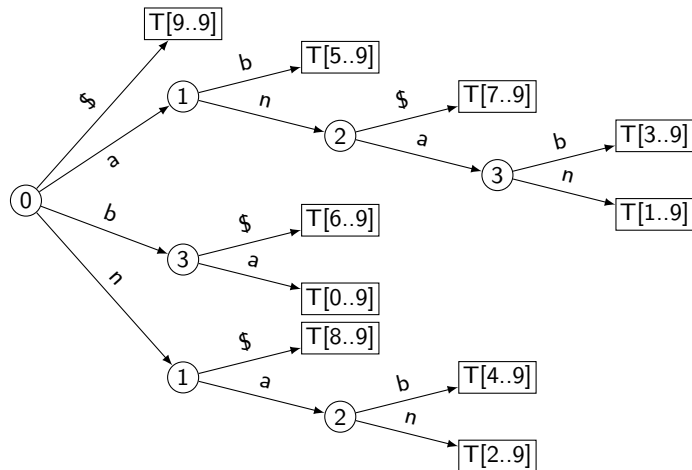


# Suffix tree

**Suffix tree:** Compressed trie of suffixes

$T =$ 

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





# Building Suffix Trees

- Text  $T$  has  $n$  characters and  $n + 1$  suffixes
- We can build the suffix tree by inserting each suffix of  $T$  into a compressed trie.  
This takes time  $\Theta(n^2)$ .
- There *is* a way to build a suffix tree of  $T$  in  $\Theta(n)$  time.  
This is quite complicated and beyond the scope of the course.

# Suffix Trees: String Matching

Assume we have a suffix tree of text  $T$ .

To search for pattern  $P$  of length  $m$ :

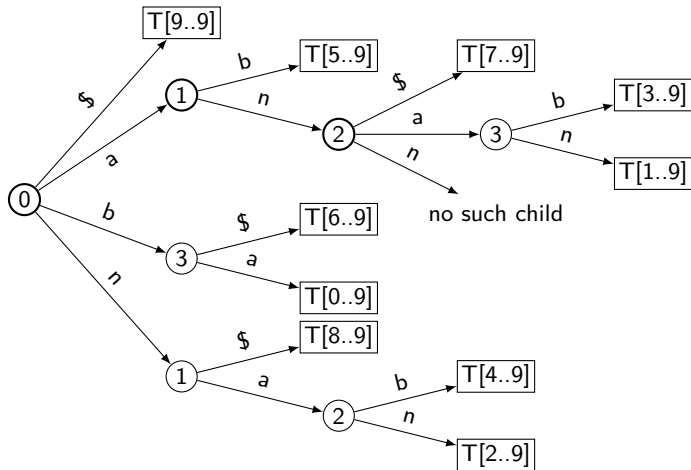
- We assume that  $P$  does not have the final \$.
- $P$  is the prefix of some suffix of  $T$ .
- In the *uncompressed* trie, searching for  $P$  would be easy:  $P$  exists in  $T$  if and only search for  $P$  reaches a node in the trie.
- In the suffix tree, search for  $P$  until one of the follow occurs:
  - ① If search fails due to “no such child” then  $P$  is not in  $T$
  - ② If we reach end of  $P$ , say at node  $v$ , then jump to leaf  $\ell$  in subtree of  $v$ . (We presume that suffix trees stores such shortcuts.)
  - ③ Else we reach a leaf  $\ell = v$  while characters of  $P$  left.
- Either way, left index at  $\ell$  gives the shift that we should check.
- This takes  $O(|P|)$  time.

# Pattern Matching in Suffix Tree: Example 1

$T =$ 

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

$P = \text{ann}$

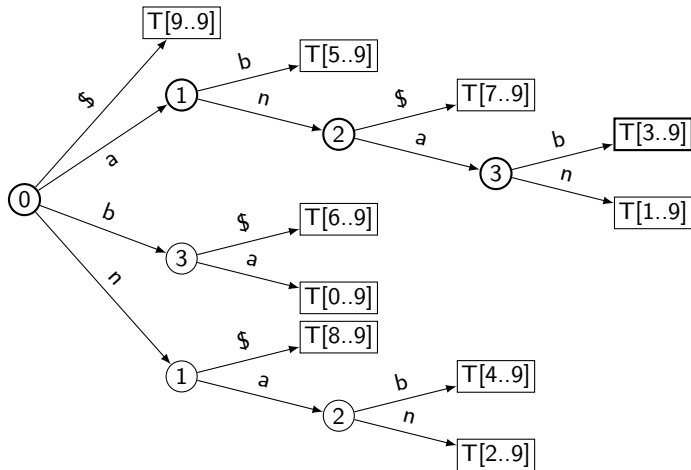


# Pattern Matching in Suffix Tree: Example 2

$T =$ 

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

$P = ana$

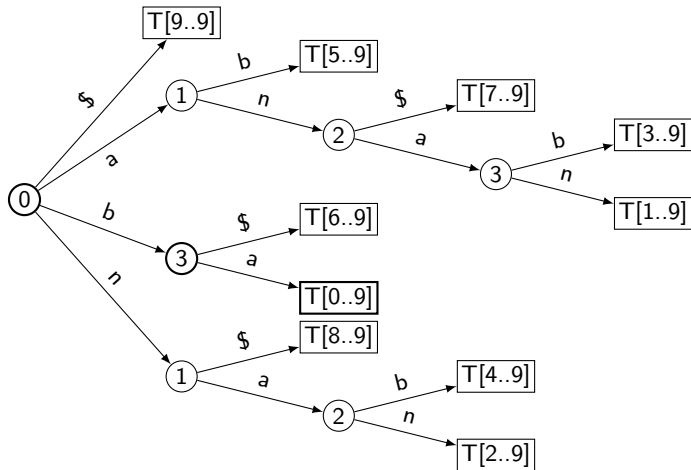


# Pattern Matching in Suffix Tree: Example 3

$T =$ 

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

$P = \text{briar}$



# Outline

## 1 String Matching

- Introduction
- Rabin-Karp Algorithm
- Boyer-Moore Algorithm
- String Matching with Finite Automata
- Knuth-Morris-Pratt algorithm
- Suffix Trees
- Conclusion

# String Matching Conclusion

	<b>Brute-Force</b>	<b>RK</b>	<b>BM</b>	<b>DFA</b>	<b>KMP</b>	<b>Suffix trees</b>
<b>Preproc.</b>	—	$O(m)$	$O(m +  \Sigma )$	$O(m \Sigma )$	$O(m)$	$O(n^2)$ ( $\rightarrow O(n)$ )
<b>Search time</b>	$O(nm)$	$O(n + m)$ (expected)	$O(n)$ (often better)	$O(n)$	$O(n)$	$O(m)$
<b>Extra space</b>	—	$O(1)$	$O(m +  \Sigma )$	$O(m \Sigma )$	$O(m)$	$O(n)$