Module 9: String Matching

T. Biedl  M. Petrick  O. Veksler
Based on lecture notes by many previous cs240 instructors

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

Winter 2019

References: Goodrich & Tamassia 23
Outline

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

1 String Matching
   • Introduction
     • Karp-Rabin 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

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

- 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:
- \( T = \) “Where is he?”
- \( P_1 = \) “he”
- \( P_2 = \) “who”

Definitions:
- **Substring** \( T[i..j] \) \( 0 \leq i \leq j < n \): a string of length \( j - i + 1 \) which consists of characters \( T[i], \ldots , T[j] \) 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 \( 0 \leq i \leq n - 1 \)
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.

```plaintext
BruteforcePM(T[0..n - 1], P[0..m - 1])
T: String of length n (text), P: String of length m (pattern)
1. for i ← 0 to n - m do
2. if strcmp(T[i..i+m-1], P) = 0
3. return “found at guess i”
4. return FAIL
```

Note: `strcmp` takes $\Theta(m)$ time.

```plaintext
strcmp(T[i..i+m-1], P[0..m-1])
1. for j ← 0 to m - 1 do
2. if T[i + j] is before P[j] in $\Sigma$ then return -1
3. if T[i + j] is after P[j] in $\Sigma$ then return 1
4. return 0
```
Brute-Force Example

- Example: \( T = \text{abbbabababbab}, \ P = \text{abba} \)

```
a b b b a a b a b b a b
```

- 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 $P$
  - Karp-Rabin
  - 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
   - Karp-Rabin Algorithm
   - Boyer-Moore Algorithm
   - String Matching with Finite Automata
   - Knuth-Morris-Pratt algorithm
   - Suffix Trees
   - Conclusion
Karp-Rabin 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 \( R = 10 \)):
    \[
    h(x_0 \ldots x_4) = (x_0x_1x_2x_3x_4)_{10} \mod 97
    \]
  - \( h(P) = 59265 \mod 97 = 95 \).

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>4</th>
<th>1</th>
<th>5</th>
<th>9</th>
<th>2</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>hash-value 84</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>hash-value 94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash-value 94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>hash-value 76</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash-value 18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>hash-value 95</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Karp-Rabin Fingerprint Algorithm – First Attempt

Karp-Rabin-Simple($T, P$)
1. $h_P \leftarrow h(P[0..m-1])$
2. for $i \leftarrow 0$ to $n-m$
3. $h_T \leftarrow h(T[i..i+m-1])$
4. if $h_T = h_P$
5. if strcmp($T[i..i+m-1], P$) = 0
6. return “found at guess $i$”
7. return FAIL

- Never misses a match: $h(T[i..i+m-1]) \neq h(P) \Rightarrow$ guess $i$ is not $P$
- $h(T[i..i+m-1])$ depends on $m$ characters, so naive computation takes $\Theta(m)$ time per guess
- Running time is $\Theta(mn)$ if $P$ not in $T$ (how can we improve this?)
Karp-Rabin 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: $10000 \mod 97 = 9$
- Previous hash: $41592 \mod 97 = 76$
- Next hash: $15926 \mod 97 = ??$

**Observe:** $15926 = (41592 - 4 \cdot 10000) \cdot 10 + 6$

$$15926 \mod 97 = \left(\left(\frac{41592 \mod 97}{76} - 4 \cdot \frac{10000 \mod 97}{9}\right) \cdot 10 + 6\right) \mod 97$$

$$= \left((76 - 4 \cdot 9) \cdot 10 + 6\right) \mod 97 = 18$$
Karp-Rabin Fingerprint Algorithm – Conclusion

Karp-Rabin-RollingHash(\(T, P\))

1. \(h_P \leftarrow h(P[0..m-1])\)
2. \(p \leftarrow\) suitable prime number
3. \(s \leftarrow 10^{m-1} \mod p\)
4. \(h_T \leftarrow h(T[0..m-1])\)
5. \(\text{for } i \leftarrow 0 \text{ to } n - m\)
6. \(\text{if } i > 0 \quad \text{// compute hash-value for next guess}\)
7. \(h_T \leftarrow ((h_T - T[i] \cdot s) \cdot 10 + T[i+m]) \mod p\)
8. \(\text{if } h_T = h_P\)
9. \(\text{if } strcmp(T[i..i+m-1], P) = 0\)
10. \(\text{return } \text{“found at guess } i\”\)
11. \(\text{return } \text{“FAIL”}\)

- Choose “table size” \(p\) at random to be huge prime
- Expected running time is \(O(m + n)\)
- \(\Theta(mn)\) worst-case, but this is (unbelievably) unlikely
Outline

1. String Matching
   - Introduction
   - Karp-Rabin 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. $i \leftarrow 0$ // current guess
3. while $i \leq n - m$
4. for $(j \leftarrow m - 1, j \geq 0, j --)$
5. if $T[i + j] \neq P[j]$ break
6. if $j = -1$ return “found at guess $i”
7. else $i \leftarrow i + \max\{1, j-L[T[i + j]]\}$
8. return FAIL

$L$ will be explained below.
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

<table>
<thead>
<tr>
<th>$c$</th>
<th>$p$</th>
<th>$a$</th>
<th>$t$</th>
<th>$i$</th>
<th>$k$</th>
<th>all others</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(c)$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>-1</td>
</tr>
</tbody>
</table>

At first mismatch:

$i = 0, j = 3, T[i+j] = \text{'}a\text{'}, L[\text{'}a\text{'}] = 1$, so $i \leftarrow i + 3 - 1$
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: youyoyo
  T: purpleyoyo
  ```

  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
   - Karp-Rabin 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 $P = \text{ababaca}$

![Finite Automaton Diagram]

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

String Matching
- Introduction
- Karp-Rabin 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

\[
\text{KMP}(T, P)
\]
1. \( F \leftarrow \text{failureArray}(P) \)
2. \( i \leftarrow 0 \) // current character of \( T \) to parse
3. \( j \leftarrow 0 \) // current state that we are in
4. \( \text{while } i < n \ \text{do} \)
5. \( \quad \text{if } P[j] = T[i] \)
6. \( \quad \quad \text{if } j = m - 1 \)
7. \( \quad \quad \quad \text{return } \text{"found at guess } i - m + 1\text{"} \)
8. \( \quad \quad \text{else} \)
9. \( \quad \quad \quad i \leftarrow i + 1 \)
10. \( \quad \quad \quad j \leftarrow j + 1 \)
11. \( \quad \text{else} \) // i.e. \( P[j] \neq T[i] \)
12. \( \quad \quad \text{if } j > 0 \)
13. \( \quad \quad \quad j \leftarrow F[j - 1] \)
14. \( \quad \text{else} \)
15. \( \quad \quad i \leftarrow i + 1 \)
16. \( \text{return } \text{FAIL} \)
String matching with KMP – Example

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

$q$: 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.

\[
\begin{array}{|c|c|c|c|}
\hline
T: & \text{current guess} & \ldots \text{matched } P[0..j] & \ldots \times \\
\hline
\end{array}
\]

- 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 guess 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 \( \ell \).

\[
F[j] = \text{head of failure-arc from state } j+1 \\
= \text{length of the longest prefix of } P \text{ that is a suffix of } P[1..j].
\]
KMP Failure Array – Example

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

Consider $P = \text{ababaca}$

<table>
<thead>
<tr>
<th>$j$</th>
<th>$P[1..j]$</th>
<th>Prefixes of $P$</th>
<th>longest</th>
<th>$F[j]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\Lambda$</td>
<td>$\Lambda, a, ab, aba, abab, ababa, \ldots$</td>
<td>$\Lambda$</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$b$</td>
<td>$\Lambda, a, ab, aba, abab, ababa, \ldots$</td>
<td>$\Lambda$</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>$ba$</td>
<td>$\Lambda, a, ab, aba, abab, ababa, \ldots$</td>
<td>$a$</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>$bab$</td>
<td>$\Lambda, a, ab, aba, abab, ababa, \ldots$</td>
<td>$ab$</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>$baba$</td>
<td>$\Lambda, a, ab, aba, abab, ababa, \ldots$</td>
<td>$aba$</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>$babac$</td>
<td>$\Lambda, a, ab, aba, abab, ababa, \ldots$</td>
<td>$\Lambda$</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>$babaca$</td>
<td>$\Lambda, a, ab, aba, abab, ababa, \ldots$</td>
<td>$a$</td>
<td>1</td>
</tr>
</tbody>
</table>

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

\[ \text{failureArray}(P) \]

\( P \): String of length \( m \) (pattern)

1. \( F[0] \leftarrow 0 \)
2. \( i \leftarrow 1 \)
3. \( j \leftarrow 0 \)
4. \( \text{while } i < m \text{ do} \)
5. \( \quad \text{if } P[i] = P[j] \)
6. \( \quad \quad j \leftarrow j + 1 \)
7. \( \quad \quad F[i] \leftarrow j \)
8. \( \quad i \leftarrow i + 1 \)
9. \( \quad \text{else if } j > 0 \)
10. \( \quad \quad j \leftarrow F[j - 1] \)
11. \( \quad \text{else} \)
12. \( \quad F[i] \leftarrow 0 \)
13. \( \quad i \leftarrow i + 1 \)

**Correctness-idea:** \( F[\cdot] \) is defined via pattern matching of \( P[1..i] \) in \( P \). So KMP uses itself! Already-built parts of \( F[\cdot] \) are used to expand it.
KMP failure function – fast computation

\[ P = \text{ababaca} \]

\[ \Sigma - a \]

Parse \( P[1..m-1] = \text{babaca} \) while adding failure-arcs:

<table>
<thead>
<tr>
<th>( i )</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>4</th>
<th>4</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P[i] )</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( P[j] )</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( j )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( F[i] )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
KMP – Runtime

failureArray

- Consider how $2i - j$ changes in each iteration of the while loop
  - $i$ and $j$ both increase by 1 $\Rightarrow$ $2i - j$ increases – OR –
  - $j$ decreases ($F[j - 1] < j$) $\Rightarrow$ $2i - j$ increases – OR –
    - $i$ increases $\Rightarrow$ $2i - j$ increases

- Initially $2i - j \geq 0$, at the end $2i - j \leq 2m$

- So no more than $2m$ iterations of the while loop.

- Running time: $\Theta(m)$

KMP main function

- failureArray can be computed in $\Theta(m)$ time

- Same analysis gives at most $2n$ iterations of the while loop since $2i - j \leq 2n$.

- Running time KMP altogether: $\Theta(n + m)$
Outline

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

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

Store suffixes via indices:

\[ T = \begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{b} & \text{a} & \text{n} & \text{a} & \text{n} & \text{a} & \text{b} & \text{a} & \text{n} & \$
\end{array} \]
Suffix tree

**Suffix tree**: Compressed trie of suffixes

\[ T = \text{bananaabanan}\]

- \( T[0..9] \) leads to \( \text{b} \) at node 0.
- \( T[1..9] \) leads to \( \text{a} \) at node 1.
- \( T[2..9] \) leads to \( \text{n} \) at node 2.
- \( T[3..9] \) leads to \( \text{a} \) at node 3.
- \( T[4..9] \) leads to \( \text{b} \) at node 4.
- \( T[5..9] \) leads to \( \text{n} \) at node 5.
- \( T[6..9] \) leads to \( \text{a} \) at node 6.
- \( T[7..9] \) leads to \$ at node 7.
- \( T[8..9] \) leads to \$ at node 8.
- \( T[9..9] \) leads to \$ at node 9.

Nodes 0, 1, 2, 3, and 7 are leaves, representing the suffixes.

Nodes 2 and 7 are internal nodes with multiple outgoing edges.
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

- In the **uncompressed** trie, searching for $P$ would be easy.
- In the **compressed** suffix tree, search as in a compressed trie. Stop the search once $P$ has run out of characters.

```
SuffixTreePM( T[0..n-1], P[0..m-1], T )
T: text, P: pattern, T: Suffix tree of T
1.   v ← T.root
2.   repeat
3.     w ← child of v corresponding to P[v.index]
4.     if there is no such child return “no match”
5.     if w is leaf or w.index ≥ m  // have gone beyond pattern P
6.         ℓ ← leaf in subtree of w
7.         if (ℓ.start+m−1 < n)
8.             if (strcmp(T[ℓ.start..ℓ.start+m−1], P) = 0)
9.                 return “match at guess ℓ.start”
10.             else return FAIL
11.     v ← w
```
Pattern Matching in Suffix Tree: Example 1

\[ T = \begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
b & a & n & a & n & a & b & a & n & $ \\
\end{array} \]

\[ P = \text{ann} \]

FAIL

\[
\begin{array}{c}
T[9..9] \\
\text{b} \\
\text{n} \\
\text{n} \\
\text{a} \\
\text{n} \\
\text{b} \\
\text{a} \\
\text{n} \\
\$ \\
\end{array}
\]

\[
\begin{array}{c}
T[5..9] \\
\text{b} \\
\text{n} \\
\text{a} \\
\text{n} \\
\text{b} \\
\text{a} \\
\text{n} \\
\$ \\
\end{array}
\]

\[
\begin{array}{c}
T[7..9] \\
\text{a} \\
\text{n} \\
\text{b} \\
\text{a} \\
\text{n} \\
\text{b} \\
\text{a} \\
\text{n} \\
\$ \\
\end{array}
\]

\[
\begin{array}{c}
T[3..9] \\
\text{b} \\
\text{n} \\
\text{a} \\
\text{n} \\
\text{b} \\
\text{a} \\
\text{n} \\
\text{n} \\
\end{array}
\]

\[
\begin{array}{c}
T[1..9] \\
\text{a} \\
\text{b} \\
\text{n} \\
\text{a} \\
\text{n} \\
\text{b} \\
\text{a} \\
\text{n} \\
\text{n} \\
\end{array}
\]

no such child

\[
\begin{array}{c}
T[6..9] \\
\text{b} \\
\text{n} \\
\text{a} \\
\text{n} \\
\text{b} \\
\text{a} \\
\text{a} \\
\text{n} \\
\$ \\
\end{array}
\]

\[
\begin{array}{c}
T[0..9] \\
\text{b} \\
\text{n} \\
\text{a} \\
\text{n} \\
\text{b} \\
\text{a} \\
\text{a} \\
\text{n} \\
\text{n} \\
\end{array}
\]

\[
\begin{array}{c}
T[8..9] \\
\text{b} \\
\text{n} \\
\text{a} \\
\text{n} \\
\text{b} \\
\text{a} \\
\text{a} \\
\text{n} \\
\text{n} \\
\end{array}
\]

\[
\begin{array}{c}
T[2..9] \\
\text{a} \\
\text{b} \\
\text{n} \\
\text{n} \\
\text{a} \\
\text{n} \\
\text{b} \\
\text{a} \\
\text{n} \\
\text{n} \\
\end{array}
\]

\[
\begin{array}{c}
T[4..9] \\
\text{a} \\
\text{b} \\
\text{n} \\
\text{n} \\
\text{a} \\
\text{n} \\
\text{b} \\
\text{a} \\
\text{n} \\
\text{n} \\
\end{array}
\]

Biedl, Petrick, Veksler  (SCS, UW)  

CS240 – Module 9  

Winter 2019  

31 / 34
Pattern Matching in Suffix Tree: Example 2

\[ T = \begin{array}{c|c|c|c|c|c|c|c|c|c|c|c} 
 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline 
 b & a & n & a & n & a & b & a & n & $ \\
\end{array} \]

\[ P = \text{ana} \]

“found at guess 3”
Pattern Matching in Suffix Tree: Example 3

$T = \begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{b} & \text{a} & \text{n} & \text{a} & \text{n} & \text{a} & \text{b} & \text{a} & \text{n} & \$
\end{array}$

$P = \text{briar}$

FAIL
Outline

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

<table>
<thead>
<tr>
<th></th>
<th>Brute-Force</th>
<th>KR</th>
<th>BM</th>
<th>DFA</th>
<th>KMP</th>
<th>Suffix trees</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preproc.</strong></td>
<td>—</td>
<td>O(m)</td>
<td>O(m +</td>
<td>Σ</td>
<td>)</td>
<td>O(m</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(→ O(n))</td>
</tr>
<tr>
<td><strong>Search time</strong></td>
<td>O(nm)</td>
<td>O(n + m)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(m)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(expected)</td>
<td>(often better)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Extra space</strong></td>
<td>—</td>
<td>O(1)</td>
<td>O(m +</td>
<td>Σ</td>
<td>)</td>
<td>O(m</td>
</tr>
</tbody>
</table>

- Our algorithms stopped once they have found one occurrence.
- Most of them can be adapted to find all occurrences within the same worst-case run-time.