Aho-Corasick Automata
String Data Structures

• Over the next few days, we're going to be exploring data structures specifically designed for string processing.

• These data structures and their variants are frequently used in practice.
  
  • In fact, the data structures we’ll talk about today were all invented by people implementing real systems!
Looking Forward

• Today: *Aho-Corasick Automata*
  • A fast data structure for string matching.
• Thursday/Tuesday: *Suffix Trees and Suffix Arrays*
  • Absurdly versatile string data structures.
String Searching
The String Searching Problem

• Consider the following problem:

  Given a string $T$ and $k$ nonempty strings $P_1, \ldots, P_k$, find all occurrences of $P_1, \ldots, P_k$ in $T$.

• $T$ is called the **text string** and $P_1, \ldots, P_k$ are called **pattern strings**.

• This problem was originally studied in the context of compiling indexes, but has found applications in computer security and computational genomics.
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
</tr>
</tbody>
</table>
Some Terminology

- Let \( m = |T| \), the length of the string to be searched.
- Let \( n = |P_1| + |P_2| + \ldots + |P_k| \) be the total length of all the pattern strings.
- Let \( L_{\text{max}} \) be the length of the longest pattern string.
- Assume that strings are drawn from an alphabet \( \Sigma \), where \( |\Sigma| \) is some constant.
- We'll use these terms when talking about the runtime of the algorithms and data structures we'll explore over the next couple of days.
How quickly can we solve the string searching problem?
Let's start with a naïve approach.
### Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>g</td>
<td>e</td>
</tr>
</tbody>
</table>

For each position in $T$:
- For each pattern string $P_i$:
  - Check if $P_i$ appears at that position.
For each position in $T$:
For each pattern string $P_i$:
Check if $P_i$ appears at that position.
For each position in $T$:
   - For each pattern string $P_i$:
     - Check if $P_i$ appears at that position.
For each position in $T$:
For each pattern string $P_i$:
Check if $P_i$ appears at that position.
Pattern Strings

For each position in $T$:
For each pattern string $P_i$:
Check if $P_i$ appears at that position.
For each position in $T$:
  For each pattern string $P_i$:
    Check if $P_i$ appears at that position.
For each position in $T$:
   For each pattern string $P_i$:
      Check if $P_i$ appears at that position.
### Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>g</td>
<td>e</td>
</tr>
</tbody>
</table>

For each position in $T$:
- For each pattern string $P_i$:
  - Check if $P_i$ appears at that position.
For each position in $T$:
For each pattern string $P_i$:
Check if $P_i$ appears at that position.
For each position in $T$:
For each pattern string $P_i$:
Check if $P_i$ appears at that position.
For each position in $T$: 
   For each pattern string $P_i$: 
      Check if $P_i$ appears at that position.
For each position in $T$:
For each pattern string $P_i$:
Check if $P_i$ appears at that position.
For each position in $T$:
   For each pattern string $P_i$:
      Check if $P_i$ appears at that position.
For each position in $T$:
For each pattern string $P_i$:
Check if $P_i$ appears at that position.
For each position in $T$:
For each pattern string $P_i$:
Check if $P_i$ appears at that position.
For each position in $T$
For each pattern string $P_i$:
Check if $P_i$ appears at that position.
Patterns Strings

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>g</td>
<td>e</td>
</tr>
</tbody>
</table>

For each position in $T$:
For each pattern string $P_i$:
Check if $P_i$ appears at that position.
For each position in $T$:
For each pattern string $P_i$:
Check if $P_i$ appears at that position.
Pattern Strings

For each position in $T$:
For each pattern string $P_i$:
Check if $P_i$ appears at that position.

$T$: abedggettab
For each position in $T$: 
For each pattern string $P_i$: 
Check if $P_i$ appears at that position.
For each position in $T$:
For each pattern string $P_i$:
Check if $P_i$ appears at that position.
Pattern Strings

For each position in $T$:
For each pattern string $P_i$:
Check if $P_i$ appears at that position.
## Pattern Strings

<table>
<thead>
<tr>
<th></th>
<th>ab</th>
<th>b</th>
<th>about</th>
<th>a</th>
<th>t</th>
<th>ate</th>
<th>be</th>
<th>bed</th>
<th>edge</th>
<th>get</th>
</tr>
</thead>
</table>

For each position in $T$:
- For each pattern string $P_i$:
  - Check if $P_i$ appears at that position.
For each position in $T$:
    For each pattern string $P_i$:
        Check if $P_i$ appears at that position.
For each position in $T$:
For each pattern string $P_i$:
Check if $P_i$ appears at that position.
For each position in $T$:
  For each pattern string $P_i$:
    Check if $P_i$ appears at that position.
For each position in $T$:
For each pattern string $P_i$:
Check if $P_i$ appears at that position.
For each position in $T$:
For each pattern string $P_i$:
Check if $P_i$ appears at that position.
Analyzing Our Approach

• As before, let $m$ be the length of the text and $n$ the total length of the pattern strings.

• For each character of the text string $T$, in the worst case, we scan over all $n$ total characters in the patterns.

• Time complexity: $O(mn)$.

• Is this a tight bound?
$\Theta(mn)$

**Pattern Strings**

- a b
- a a b
- a a a b
- a a a a b
- a a a a a b

- a a a a a a a a a a a a a
Can we do better?
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
</tr>
</tbody>
</table>

a b e d g e t
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
</tr>
</tbody>
</table>

Sequence: abedget
### Pattern Strings

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>o</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>t</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
<td>g</td>
</tr>
<tr>
<td>g</td>
<td>e</td>
<td>t</td>
</tr>
</tbody>
</table>

String: abedgedget
### Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>b</th>
<th>a</th>
<th>t</th>
<th>e</th>
<th>e</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>o</td>
<td>u</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>t</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>d</td>
<td>g</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>e</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

```
 abedgedget
```
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>g</td>
<td>e</td>
</tr>
</tbody>
</table>

Matching strings:

- a b
- about
- at
- ate
- be
- bed
- edge
- get

Matching pattern:

- abedget
### Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>about</td>
<td></td>
</tr>
<tr>
<td>at</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>e</td>
<td>t</td>
</tr>
<tr>
<td>bed</td>
<td></td>
</tr>
<tr>
<td>edge</td>
<td></td>
</tr>
<tr>
<td>get</td>
<td></td>
</tr>
</tbody>
</table>

- **Pattern Strings**

```
| ab | about | at | ate | bed | edge | get |
```

- **Benchmark Text**

```
| a | b | e | d | g | e | t |
```
Pattern Strings

- ab
- about
- at
- ate
- be
- bed
- edge
- get
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b</td>
<td></td>
</tr>
<tr>
<td>a b o u t</td>
<td></td>
</tr>
<tr>
<td>a t</td>
<td></td>
</tr>
<tr>
<td>a t e</td>
<td></td>
</tr>
<tr>
<td>b e</td>
<td></td>
</tr>
<tr>
<td>b e d</td>
<td></td>
</tr>
<tr>
<td>e d g e</td>
<td></td>
</tr>
<tr>
<td>g e t</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

```
| a | b | e | d | g | e | t |
```
Pattern Strings

- ab
- about
- at
- ate
- be
- bed
- edge
- get

[Diagram showing strings ab, bed, and edge being matched with abedget]
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>g</td>
<td>e</td>
</tr>
</tbody>
</table>

a b e d g e t
### Pattern Strings

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>t</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
<td>g</td>
</tr>
<tr>
<td>g</td>
<td>e</td>
<td>t</td>
</tr>
</tbody>
</table>

![Diagram](image-url)
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>g</td>
<td>e</td>
</tr>
</tbody>
</table>

Diagram:

```
| a | b | e | d | g | e | t |
```

Red arrow pointing to "edge".
Pattern Strings

- ab
- about
- at
- ate
- be
- bed
- edge
- get
Parallel Searching

- **Idea:** Rather than searching the pattern strings in *serial*, try searching them in *parallel*.
- Intuitively, this should cut down on a lot of the unnecessary rescanning that we're doing.
- **Challenge:** How exactly do we do this in practice?
**Pattern Strings**

- ab
- about
- at
- ate
- be
- bed
- edge
- get
This data structure is called a **trie**. It comes from the word *retrieval*. It is not pronounced like “retrieval.”
Representing Tries

- Each trie node needs to store pointers to its children.
- There are many different data structures we could use to store these pointers.
- For today, we'll assume we have an array of $|\Sigma|$ pointers, one per possible child.
- You'll explore variants on this strategy in the problem set.
Representing Tries

- Each trie node needs to store pointers to its children.
- There are many different data structures we could use to store these pointers.
- For today, we'll assume we have an array of $|\Sigma|$ pointers, one per possible child.
- You'll explore variants on this strategy in the problem set.
**Pattern Strings**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>e</td>
<td>g</td>
</tr>
<tr>
<td>e</td>
<td>t</td>
</tr>
</tbody>
</table>

- Pattern: `about eat ate bed edge get`
Pattern Strings

- about
- at
- ate
- bed
- edge
get
Pattern Strings

- about
- at
- ate
- bed
- edge
get
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
</tr>
</tbody>
</table>

Diagram with nodes and edges connected to form a tree structure.
Pattern Strings

- about
- ate
- bed
- edge
- get
Pattern Strings

- about
- at
- ate
- bed
- edge
get
Pattern Strings

- about
- at
- ate
- be
- bed
- edge
- get

The diagram shows a tree structure with nodes labeled 'a', 'b', 't', 'd', 'e', 'g', and 'u'. The highlighted path matches the pattern string 'abedget'.
Pattern Strings

- about
- at
- ate
- bed
- edge
- get

Diagram showing a tree structure with nodes labeled with letters and edges connecting them.
Pattern Strings:

- about
- ate
- bed
- edge
- get
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>o</th>
<th>u</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>o</td>
<td>u</td>
<td>t</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>t</td>
<td>e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>d</td>
<td>g</td>
<td>e</td>
<td>t</td>
</tr>
</tbody>
</table>
Pattern Strings

- about
- ate
- bed
- edge
- get

Diagram showing a graph with nodes and edges, indicating the structure of the pattern strings.
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>f</td>
<td>g</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
</tr>
</tbody>
</table>
Pattern Strings

- about
- at
- ate
- bed
- edge
- get
Analyzing our New Algorithm

• Let's suppose we've already constructed the trie. How much work is required to perform the match?

• For each character of $T$, we inspect as most as many characters as exist in the deepest branch of the trie.

• Time complexity: $O(mL_{\text{max}})$, where $L_{\text{max}}$ is the length of the longest pattern string. (Do you see why?)

• In the (reasonable) case where $L_{\text{max}}$ is much smaller than $n$, this is a huge win over before. If $L_{\text{max}}$ is “objectively” small, this is a pretty good good runtime.

• How much time does it take to build the trie?
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 

| a | t | e |
Building a Trie

• **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 

```
  a  t  e
```
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 

![Diagram showing a trie structure with nodes labeled 'a', 't', and 'e']
Claim: Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 

![Diagram of a Trie]

The diagram shows a trie with nodes for the strings "ate" and "ate".


Building a Trie

● **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 

![Trie Diagram]

- `ate`
- `an`
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 
Building a Trie

• **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$.
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 

![Trie Diagram]

1. ate
2. an
3. at
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$.
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 

![Trie diagram]

- **Trie diagram:**
  - Root node with edges labeled 'a', 't', 'e'.
  - 'a' leads to a node with 'n' edge.
  - 't' leads to a node with 'e' edge.
  - 'n' leads to a leaf node.
  - 'e' leads to a leaf node.

- **Strings:**
  - 'ate'
  - 'an'
  - 'at'
Building a Trie

**Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$.
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 

\[
\begin{array}{cccc}
  a & t & e \\
  a & n \\
  a & t \\
  b & e \\
\end{array}
\]
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 
**Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 

![Trie Diagram](image.png)
Building a Trie

• **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 
Building a Trie

- **Claim:** Given a set of strings $P_1, \ldots, P_k$ of total length $n$, it's possible to build a trie for those strings in time $\Theta(n)$. 
Our Strategies

• Following our foray into RMQ, we'll say that a solution to multi-string matching runs in time $\langle p, q \rangle$ if the preprocessing time is $p$ and the matching time is $q$.

• We now have two approaches:
  • No preprocessing: $\langle O(1), O(mn) \rangle$.
  • Trie searching: $\langle O(n), O(mL_{max}) \rangle$.

• *Can we do better?*
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>
Pattern Strings

| at | art | oars | soar |

Diagram:

- `a` → `r` → `t` → (sink)
- `o` → `a` → `r` → `s` → (sink)
- `s` → `o` → `a` → `r` → (sink)
Pattern Strings

```plaintext
- art
- soars
- soar
```

Diagram showing a transition diagram with states and transitions labeled with letters.
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>

Diagram with nodes labeled with 'a', 'r', 't', 'o', 'a', 'r', 's', 'o', 'a', 'r'. The diagram shows the pattern strings and their relationships. The diagram also includes a sequence of 's', 'o', 'a', 'r', 's' at the bottom, indicating the pattern being matched.
Pattern Strings

- at
- art
- soars
- soar

Diagram:

- Nodes labeled with characters: o, a, r, t, s.
- Edges connecting the nodes with arrows showing transitions.
- Start node connected to 'o' and 's', ending node connected to 't' and 's'.
Pattern Strings

<table>
<thead>
<tr>
<th>at</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>art</td>
<td>r</td>
</tr>
<tr>
<td>soar</td>
<td>s</td>
</tr>
</tbody>
</table>
Pattern Strings

- at
- art
- soars
- soar

Diagram:

- Initial state:
  - Input: s
  - Header: yellow circle
- Transition:
  - s -> o
  - o -> a
  - a -> r
  - r -> s
- Final state:
  - Output: t

- Additional state:
  - Input: t
  - Header: yellow circle
Pattern Strings

- at
- art
- soars
- soar

Diagram showing the transition of strings from left to right.
Pattern Strings

at
art
soars
soar

Diagram:

- Graph showing transitions between characters: 'at' leads to 't', 'art' leads to 't', 'soars' leads to 'soars', and 'soar' leads to 'soar'.

- Red arrow highlights a specific path through the graph.

- Yellow circle at the end of the path indicates the end state.
Pattern Strings:

- at
- art
- soars
- soar

Graph:

- From 's', move to 'o', then to 'a', then to 'r', then to 's'.
- From 'o', move to 'a', then to 'r', then to 's'.
- From 'a', move to 'r', then to 't', then to 's'.
- From 't', move to 's'.

The sequence 'soars' is highlighted in yellow.
This red link is called a suffix link. We'll talk about them more formally in a minute.
Pattern Strings

a t
a r t
o a r s
s o a r
Pattern Strings

<table>
<thead>
<tr>
<th>at</th>
<th>art</th>
</tr>
</thead>
<tbody>
<tr>
<td>oars</td>
<td>soar</td>
</tr>
</tbody>
</table>

Diagram showing a transition graph with states and transitions labeled with 'o', 'a', 'r', 's', 't'.
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>

Diagram:

- Initial state
- Transition paths:
  - a → r → t
  - o → a → r → s
  - s → o → a → r

Highlighted path:
- o → a → r → s
Pattern Strings

<table>
<thead>
<tr>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a r t</td>
</tr>
<tr>
<td>o a r s</td>
</tr>
<tr>
<td>s o a r</td>
</tr>
</tbody>
</table>

Diagram:

- Node a connects to nodes r and t.
- Node o connects to nodes a and r.
- Node s connects to nodes o, a, and r.
- The path from the start to the yellow node is marked with a red arrow.
Pattern Strings

| a | t | a | t | o | a | r | s | s | o | a | r |

Diagram:

- Nodes: o, a, r, t
- Edges: o → a → r → t
- Pattern: oart
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>

Diagram of a pattern string recognition process.
Pattern Strings

```
<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>
```
Pattern Strings

a t
a r t
o a r s
s o a r

Diagram: A graph with vertices labeled with strings such as "a", "t", "o", "s", and "r" connected by directed edges.
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>

---

Diagram showing a sequence of nodes with edges labeled 'a', 'o', 's', 't', 'r', and 'a', forming a path through the nodes.
Pattern Strings

- at
- art
- oars
- soar

Diagram:

- Input:
  - s
  - o
  - a
  - t

- Output:
  - t
  - a
  - r
  - t

- States:
  - Yellow (accepted)
  - Grey (rejected)
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>

Diagram: Node transitions for strings with arrows indicating the sequence.
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>

Diagram with nodes and edges for pattern strings.
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>
Pattern Strings

| at | art | soars | soar |

Diagram showing a tree structure with nodes labeled 'a', 'r', 't', 's', 'o', 'a', 'r', 's' connecting to form the strings 'soars' and 'soars'.
Pattern Strings

- at
- art
- soars
- soar

Diagram:

- States and transitions for a, r, o, a, r, s, t
- Red arrows indicate specific transitions
- Input: soarsoars
Pattern Strings

- at
- art
- soars
- soar

Diagram:

- Graph with nodes labeled with letters (a, o, r, s, t).
- Arrows indicate transitions between nodes.
- States at the end of the graph represent match or acceptance.
Pattern Strings

\[
\begin{array}{cccc}
  a & t \\
  a & r & t \\
  o & a & r & s \\
  s & o & a & r & s \\
\end{array}
\]
### Pattern Strings

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
<td>t</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

![Diagram](image-url)

- **Pattern Strings**: at, art, oars, soar

![Diagram](image-url)

- **Diagram Example**: Shows a sequence of nodes representing the relationships between the pattern strings.
Pattern Strings

```
  a  t
   |   |
  a  r  t
   |   |
s o a r s
```

---

Diagram: Transition automaton for the pattern strings.
Pattern Strings

at
art
oars
soar

Diagram with nodes connected by arrows, showing transitions between characters: t, a, r, s.

Output: soars soars soars
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>

Diagram showing a pattern recognition process with nodes and edges. The diagram includes transitions for strings such as soar and soars.
Pattern Strings

a  t
a  r  t
o  a  r  s
s  o  a  r  s
In general, suffix links might jump the red cursor forward more than one step. The number of steps taken is equal to the change of depth in the trie.
Pattern Strings

```
<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>
```

Diagram:

- Arrows indicate transitions from one state to another.
- The diagram shows a finite state machine with inputs and outputs.
- The pattern strings are used to match different paths through the machine.
Pattern Strings

at
art
soars
soar

---

s o a r s o a r s o a r s
Pattern Strings

\[
\begin{array}{ccc}
\text{a} & \text{t} \\
\text{a} & \text{r} & \text{t} \\
\text{o} & \text{a} & \text{r} & \text{s} \\
\text{s} & \text{o} & \text{a} & \text{r} & \text{s}
\end{array}
\]
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>

Diagram showing a pattern matching algorithm with states for each character in the pattern strings: "s o a r s o a r s".
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>

s o t a t t
Pattern Strings

```
<table>
<thead>
<tr>
<th>at</th>
</tr>
</thead>
<tbody>
<tr>
<td>art</td>
</tr>
<tr>
<td>oars</td>
</tr>
<tr>
<td>soar</td>
</tr>
</tbody>
</table>
```

Diagram with transitions labeled by characters.
Pattern Strings

```
  at
  art
  oars
  soar
```
Pattern Strings

```
<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>
```

Diagram:

- Nodes labeled with letters (a, o, s, t, r)
- Edges showing transitions between nodes
- Red arrows indicating specific transitions
- Initial state represented by an arrow pointing upward
- Final state represented by a circle

Sequence:

```
soatat
```
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>

Diagram:

- States and transitions are represented by circles and arrows.
- The pattern strings are shown on the left:
  - atm
  - art
  - oars
  - soar
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>

Diagram with nodes labeled with letters and connections indicated.
Pattern Strings

\[ \begin{array}{cccc}
  a & t \\
  a & r & t \\
  o & a & r & s \\
  s & o & a & r \\
\end{array} \]
Pattern Strings

<table>
<thead>
<tr>
<th>a</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>

Diagram:

- Start node
- Transitions:
  - a → r → t → (final node)
  - o → a → r → s → (final node)
  - s → o → a → r → s → (final node)

- Highlighted path: s → o → a → r → s
Suffix Links

- A **suffix link** (sometimes called a **failure link**) is a red edge from a trie node corresponding to string $\alpha$ to the trie node corresponding to a string $\omega$ such that $\omega$ is the longest proper suffix of $\alpha$ that is still in the trie.

- **Intuition**: When we hit a part of the string where we cannot continue to read characters, we fall back by following suffix links to try to preserve as much context as possible.

- Every node in the trie, except the root (which corresponds to the empty string $\varepsilon$), will have a suffix link associated with it.
Why Suffix Links Matter

- Suffix links can substantially improve the performance of our string search.
- At each step, we either
  - advance the black (end) pointer forward in the trie, or
  - advance the red (start) pointer forward.
- Each pointer can advance forward at most $O(m)$ times.
- This reduces the amount of time spent scanning characters from $O(mL_{max})$ down to $Θ(m)$.
- This is only useful if we can compute suffix links quickly... which we'll see how to do later.
A Problem with our Optimization
Pattern Strings

\[ \text{string} \]

\[ \text{tinting} \]

\[ \text{string} \]
Pattern Strings

i
in
tin
string
What Happened?

- Our heavily optimized string searcher no longer starts searching from each position in the string.
- As a result, we now might forget to output matches in certain cases.
- We need to figure out
  - when this happens, and
  - how to correct for it.
Pattern Strings

s t i n g
Pattern Strings

i
in
thin
string
Pattern Strings

Pattern String:

\[
i \rightarrow in \rightarrow tin \rightarrow sting
\]
We missed the pattern string i because it's a proper suffix of sti.
We missed both *tin* and *in* because each is a proper suffix of *stin*. 
How do we address this?
Pattern Strings

in
in
in
in
in

s t i n g
Pattern Strings

Input:

Pattern:

Matching:

Output:

String:
Pattern Strings

i
in
tin
string

string
Pattern Strings

i
in
tin
string
Pattern Strings

i
in
tin
sting
Pattern Strings

string
Pattern Strings

i
in
тин
sting

string

s
t
i
n
g
Pattern Strings

In

In

Stringing

Stringing

Stringing
The Problem

• The approach described previously will ensure that we don't miss any patterns, but it increases the complexity of the search.

• Specifically, we do $O(L_{\text{max}})$ additional work at each character following suffix links backwards.

• This brings our search cost back up to $O(mL_{\text{max}})$ again – and that's too slow.

• *Can we do better?*
This blue arrow is called an output link. Whenever we visit this gold node, we'll output the string represented by the node at the end of the blue arrow.
Pattern Strings

i
in
tin
string
Pattern Strings

i
in
stitning
By precomputing where we eventually need to end up, we can instantly read off any extra patterns to emit at this point. As you'll see, we can precompute these links really quickly!
Even nodes that themselves correspond to are patterns might need output links if other patterns also end at the corresponding string.
Notice that the blue edges here form a linked list. If we visit this node, we need to output everything in the chain, not just the “tin” node we're immediately pointing at.
Pattern Strings

sting
Pattern Strings

In

Pattern:

String: sting

Diagram:

Transition:

States:
Pattern Strings

s t i n g
Pattern Strings

\[
\begin{array}{c}
i \\
in \\
tin \\
sting \\
\end{array}
\]
The Final Matching Algorithm

- Start at the root node in the trie.
- For each character $c$ in the string:
  - While there is no edge labeled $c$:
    - If you're at the root, break out of this loop.
    - Otherwise, follow a suffix link.
  - If there is an edge labeled $c$, follow it.
  - If the current node corresponds to a pattern, output that pattern.
  - Output all words in the chain of output links originating at this node.
The Runtime Impact
Pattern Strings
Pattern Strings
Pattern Strings

```
<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>a</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
</tbody>
</table>
```
Pattern Strings
Pattern Strings
Pattern Strings
Pattern Strings
Pattern Strings

a a a a a a a a

a a a a

Pattern Strings
Pattern Strings
Pattern Strings
The Runtime

- In the worst case, we may have to spend a huge amount of time listing off all the matches in the string.
- This isn't the fault of the algorithm – any algorithm that matches strings this way would have to spend the time reporting matches.
- To account for this, let $z$ denote the number of matches reported by our algorithm.
- The runtime of the match phase is then $\Theta(m + z)$, with the $m$ term coming from the string scanning and the $z$ term coming from the matches.

- You sometimes hear algorithms whose runtime depends on how much output is generated referred to as output-sensitive algorithms.
Where We Are

- Given the matching automaton (which is called an *Aho-Corasick automaton* or an *AC automaton*), we can find all occurrences of the pattern strings in any text of length $m$ in time $\Theta(m + z)$.
- To see whether this is worthwhile, we need to see how quickly we can build the automaton.
Time-Out for Announcements!
Problem Sets

• Problem Set 0 was due today at 2:30PM.
  • You have two 24-hour late days you can use throughout the quarter. Nothing more than 48 hours late will be accepted, since we’ll be giving out solutions!

• Problem Set 1 goes out today. It’s due next Tuesday at 2:30PM.
  • Play around with the RMQ structures we explored last week!
  • See how well they work in practice!

• As always, stop by office hours or visit Piazza if you have any questions!
HackOverflow

- Stanford WiCS is hosting HackOverflow, a hackathon for programmers of all skill levels. It's coming up on Saturday, April 14 from 10AM – 10PM. Everyone is welcome!

- Highly recommended! If you've never been to a hackathon before, this is one of the best places to start.
Statistics for Social Good

- Stanford Statistics for Social Good is holding a spring quarter event on Thursday, April 12 from 4PM – 6PM in the BEAM conference room.
- Interesting? RSVP using this link.
Back to CS166!
Building the Aho-Corasick Automaton
Building the Automaton

- To construct the Aho-Corasick automaton, we need to
  - construct the trie,
  - construct suffix links, and
  - construct output links.
- We know we can build the trie in time $\Theta(n)$ using our logic from before.
- How quickly can we construct suffix and output links?
Constructing Suffix Links
Pattern Strings

- ta
- ate
- coats
- so
Pattern Strings

```
ta
te
coats
so
```
Pattern Strings

<table>
<thead>
<tr>
<th>t</th>
<th>a</th>
<th>e</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>o</td>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
An Initial Algorithm

• Here is a simple, brute-force approach for computing suffix links:
  • For each node in the trie:
    - Let $\alpha$ be the string that this particular node corresponds to.
    - For each proper suffix $\omega$ of $\alpha$:
      • Look up $\omega$ in the trie.
      • If the search ends up at some trie node, point the suffix link there and stop.
  • This approach is not very efficient – that doubly-nested loop is exactly the sort of thing we're trying to avoid.

• *Can we do better?*
$$t \rightarrow a \rightarrow t \rightarrow e \rightarrow o \rightarrow a \rightarrow t \rightarrow s$$
Pattern Strings

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>a</td>
<td>t</td>
<td>e</td>
<td>t</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
<td>e</td>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>c</td>
<td>o</td>
<td>a</td>
<td>t</td>
<td>s</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fast Suffix Link Construction
Constructing Suffix Links

- **Key insight**: Suppose we know the suffix link for $w$ points to some string $x$. We can say a lot about the suffix link for $wa$.

- **Case 1**: $xa$ exists.

![Diagram showing suffix links for $w$, $xa$, and $wa$.]
Constructing Suffix Links

- **Key insight**: Suppose we know the suffix link for $w$ points to some string $x$. We can say a lot about the suffix link for $wa$.

- **Case 2**: $xa$ does not exist.
Constructing Suffix Links

- **Key insight**: Suppose we know the suffix link for $w$ points to some string $x$. We can say a lot about the suffix link for $wa$.
- **Case 2**: $xa$ does not exist.
Constructing Suffix Links

- To construct the suffix link for a node \( wa \):
  - Follow \( w \)'s suffix link to node \( x \).
  - If node \( xa \) exists, \( wa \) has a suffix link to \( xa \).
  - Otherwise, follow \( x \)'s suffix link and repeat.
  - If you need to follow backwards from the root, then \( wa \)'s suffix link points to the root.

- **Observation 1**: Suffix links point from longer strings to shorter strings.

- **Observation 2**: If we precompute suffix links for nodes in ascending order of string length, all of the information needed for the above approach will be available at the time we need it.
Pattern Strings

ta
tate
toats
tos
Pattern Strings

<table>
<thead>
<tr>
<th>t</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>c</td>
<td>o</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
</tr>
</tbody>
</table>
Pattern Strings

- tate
- coats
- so
Pattern Strings

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>t</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>t</td>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>c</td>
<td>t</td>
<td>e</td>
<td>t</td>
</tr>
<tr>
<td>s</td>
<td>a</td>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>o</td>
<td>t</td>
<td>s</td>
<td>t</td>
</tr>
</tbody>
</table>
Pattern Strings

- ta
- ate
- coats
- so
Pattern Strings

\begin{align*}
\text{t} & \rightarrow \\
\text{a} & \rightarrow \\
\text{t} & \rightarrow \\
\text{e} & \rightarrow \\
\text{c} & \rightarrow \\
\text{o} & \rightarrow \\
\text{a} & \rightarrow \\
\text{t} & \rightarrow \\
\text{s} & \rightarrow \\
\end{align*}

- \text{t a e t c o a t s}
- \text{s o}
Pattern Strings

<table>
<thead>
<tr>
<th>t</th>
<th>a</th>
<th>t</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>o</td>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pattern Strings:

- ta
- ate
- coats
- so
Pattern Strings

<table>
<thead>
<tr>
<th>t</th>
<th>a</th>
<th>t</th>
<th>e</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>t</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>t</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
<td>a</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>o</td>
<td>o</td>
<td>a</td>
<td>t</td>
<td>t</td>
</tr>
</tbody>
</table>
Pattern Strings

<table>
<thead>
<tr>
<th>t</th>
<th>a</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>t</td>
<td>e</td>
</tr>
<tr>
<td>c</td>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
<td></td>
</tr>
</tbody>
</table>
Pattern Strings

<table>
<thead>
<tr>
<th>t</th>
<th>a</th>
<th>t</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>o</td>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pattern Strings

<table>
<thead>
<tr>
<th>t</th>
<th>a</th>
<th>t</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>o</td>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pattern Strings

<table>
<thead>
<tr>
<th>t</th>
<th>a</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>t</td>
<td>e</td>
</tr>
<tr>
<td>c</td>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>s</td>
<td>o</td>
<td></td>
</tr>
</tbody>
</table>

Diagram with states and transitions representing the pattern strings.
Pattern Strings

- t
- a
t
- e
- a
t
s
- o
- a
- t
- s
Pattern Strings

```
t a
e t e
s o
```
Constructing Suffix Links

• Do a breadth-first search of the trie, performing the following operations:
  • If the node is the root, it has no suffix link.
  • If the node is one hop away from the root, its suffix link points to the root.
  • Otherwise, the node corresponds to some string \( wa \).
  • Let \( x \) be the node pointed at by \( w \)'s suffix link. Then, do the following:
    - If the node \( xa \) exists, \( wa \)'s suffix link points to \( xa \).
    - Otherwise, if \( x \) is the root node, \( wa \)'s suffix link points to the root.
    - Otherwise, set \( x \) to the node pointed at by \( x \)'s suffix link and repeat.
Analyzing Efficiency

• How much time does it take to actually build all the suffix links?

• When filling in any individual suffix link, we might have to keep walking backwards in the trie following suffix links repeatedly while searching for a place to extend.

• Intuitively, it seems like it should be quadratic in the length of the longest string in the trie.

• Is that bound tight?
Analyzing Efficiency

- **Claim:** The previously-described algorithm for computing suffix links takes time $O(n)$.

- **Intuition:** Focus on any one word in the trie. As you add suffix links, keep track of the depth of the node pointed at by the current node's suffix link.
Construction Efficiency

- Focus on the time to fill in the suffix links for a single pattern of length $h$.
- The gold node (where the previous suffix link points) begins at the root. At each step, the gold node
  - takes some number of steps backward, then
  - takes at most one step forward.
- The gold node cannot take more steps backward than forward. Therefore, across the entire construction, the gold node takes at most $h$ steps backward.
- Total time required to construct suffix links for a pattern of length $h$: $O(h)$.
- Total time required to construct all suffix links: $O(n)$. 
Computing Output Links
Pattern Strings

Patterns:
- sting

Diagram:
- Nodes and edges representing transitions between characters in the string.
- The string "sting" is processed through the diagram, indicating a matching pattern.
Pattern Strings

- sting
Pattern Strings

i
in
tin
string
Pattern Strings

i
in
tin
string
The Idea

- Some trie nodes represent strings that have a pattern string as a proper suffix.
- Our goal is to introduce output links so that, when these nodes are visited, the automaton outputs all the suffixes that end there.
Output Links, Formally

• The output link at a node corresponding to a string $w$ points to
  • the node corresponding to the longest proper suffix of $w$ that is a pattern, or
  • null if no such suffix exists.

• By always pointing to the node corresponding to the $longest$ such word, we ensure that we chain together all the patterns using output links.
Pattern Strings

i

in

tin

string
We want the gold node to point to the first node reachable by suffix links that's also a pattern.

The blue node (at the end of the suffix link) isn't a pattern, but it knows where the first pattern is. We set the gold node's output link to equal the blue node's output link.
We have the gold node point to the blue node because the blue node corresponds to a word.
Filling In Output Links

- Initially, set every node's output link to be a null pointer.
- While doing the BFS to fill in suffix links, set the output link of the current node $v$ as follows:
  - Let $u$ be the node pointed at by $v$'s suffix link.
  - If $u$ corresponds to a pattern, set $v$'s output link to $u$ itself.
  - Otherwise, set $v$'s output link to $u$'s output link.
- Time complexity of building all output links: $O(n)$. 
The Net Complexity

- Our preprocessing time is
  - $\Theta(n)$ work to build the trie,
  - $O(n)$ work to fill in suffix links, and
  - $O(n)$ work to fill in output links.
- Total preprocessing time: $\Theta(n)$. 
The Final Totals

- We now have a multi-string search data structure with time complexity \(\langle O(n), O(m + z)\rangle\).
- In other words, this is exceptionally good in the case where there are a fixed set of patterns and a variable string to search.
Where We're Going

• A powerful data structure called the suffix tree lets us solve this problem in \( \langle O(m), O(n + z) \rangle \).

• In other words, it excels when there's a fixed string to search and a variable set of patterns.
More to Explore

- There are a number of other approaches to solving this problem, and there's often a large gap between theory and practice!

- The **Boyer-Moore** algorithm searches for a single pattern in a large text. It can actually run in *sublinear time* if the string searched for isn't present, but runs in quadratic case if a match exists.

- The **Commentz-Walter** algorithm generalizes Boyer-Moore for multiple strings and has similar time guarantees, but tends to be a little chaotic in practice.

- The **Knuth-Morris-Pratt** algorithm is a special case of the Aho-Corasick algorithm when there is just one pattern. (Aho-Corasick is actually a generalization of KMP!)
Next Time

- **Suffix Trees**
  - A highly versatile, flexible, powerful data structure for string processing.

- **Suffix Arrays**
  - Shrinking down trie space usage.

- **Applications of RMQ**
  - Getting some mileage out of Fischer-Heun.