Tries and Suffix Trees
String Data Structures

- Our next three lectures are all on the wonderful world of string data structures.
- Why are they worth studying?
  - *They’re practical*. These data structures were developed to meet practical needs in data processing. Lots of important data can be encoded as strings.
  - *They’re different*. The questions typically asked about strings involve properties of sequences, not individual elements, in a way that you don’t normally otherwise see.
  - *They’re algorithmically interesting*. The techniques that power these data structures involve some truly beautiful connections and observations.
Where We’re Going

• Today, we’ll cover *tries* and *suffix trees*, two powerful data structures for exposing shared structures in strings.

• On Thursday, we’ll see the *suffix array* and *LCP array*, which are a more space-efficient way of encoding suffix trees.

• Next Tuesday, we’ll see the *SA-IS algorithm*, which quickly builds suffix trees and suffix arrays, and is probably the most beautiful divide and conquer algorithm ever invented.
Part I: *Tries and Patricia Tries*
A Motivating Problem
How is this done so quickly?
The Autocomplete Problem

- We have a series of text strings $T_1, T_2, \ldots, T_k$ of total length $m$. ($|T_1| + \ldots + |T_k| = m$)
- We have a pattern string $P$ of length $n$. ($|P| = n$).
- **Goal:** Find all text strings that start with $P$.
- If we just do a single query, then we can solve this pretty easily.
  - Just scan over all the strings and see which ones start with $P$.
- **Question:** If we have a set of fixed text strings and varying patterns, can we speed this up?
A Naive Solution
We're spending a lot of time scanning shared prefixes. Is there a way to avoid this?
This data structure is called a trie. It comes from the word retrieval. It is not pronounced like “retrieval.”
Each edge is labeled with a character.

Some nodes are marked as representing words.

By convention, children are stored in sorted order.

A preorder traversal of the trie prints all words in sorted order.
Now, do a DFS to find all words rooted here.
We fell off the trie. There are no matches!
Tries

- **Recall:** The total length of our text strings is $m$, and the length of our pattern string is $n$.
- How long does it take to build our trie?
- **Claim:** Ignoring the size of the alphabet, the runtime is $O(m)$. 
Tries

- **Recall:** The total length of our text strings is $m$, and the length of our pattern string is $n$.
- How long does it take to check if the pattern is a prefix of any string?
- **Claim:** Ignoring the size of the alphabet, the runtime is $O(n)$. 
Tries

- **Recall**: The total length of our text strings is $m$, and the length of our pattern string is $n$.
- How long does it take to find all text strings that start with the pattern?
- That’s a trickier question.
Tries

- **Question**: In what format do we want our matches?

- **Option 1**: Just print out all the matches.
  - Search for the prefix as usual.
  - Do a DFS, recording the letters seen on each branch, to rebuild all the words.

- We can upper-bound runtime at $O(m + n)$, but it’s hard to say much more than that.
  - (We could upper-bound this expression at $O(m)$ if we’d like, but I like showing both costs here.)
**Question:** In what format do we want our matches?

**Option 2:** Assume each text string has some numeric ID, and we want all matching IDs.

Ideally, we’d like a time complexity of something like $O(n + z)$, where $z$ is the number of matches.

Our current DFS can’t achieve this; the lengths of the strings matter.

Can we do better?
ant
ante
anteater
antelope
antique
The $ symbol is called the **sentinel** or **end-marker**. It’s a special character that can only appear at the ends of words. (Think “null terminator,” *Theoryland* edition.)
By convention, the sentinel $\$$ precedes all other characters.

(It really is like a null terminator!)
Nodes now fall into one of two classes:

**Leaf nodes** correspond to words in the trie.

**Internal nodes** correspond to routing structure.
A node is a **silly node** if it is a non-root node that only has one child.

A **Patricia trie** is a trie where silly nodes are merged into their parents.
A node is a **silly node** if it is a non-root node that only has one child.

A **Patricia trie** is a trie where silly nodes are merged into their parents.
A node is a **silly node** if it is a non-root node that only has one child.

A **Patricia trie** is a trie where silly nodes are merged into their parents.
A node is a **silly node** if it is a non-root node that only has one child.

A **Patricia trie** is a trie where silly nodes are merged into their parents.

**Observation 1:** Every internal node in a Patricia trie (except possibly the root) has two or more children.
A node is a **silly node** if it is a non-root node that only has one child.

A **Patricia trie** is a trie where silly nodes are merged into their parents.

**Observation 2:** Leaves correspond to words; internal nodes are there for routing purposes.
Theorem: The number of nodes in a Patricia trie with \( k \) words is always \( O(k) \), regardless of what those words are.

Proof Sketch: There are \( k \) leaves, one per word. Remove all internal nodes, leaving a forest of \( k \) trees. Add the internal nodes back one at a time. Each addition (except possibly root) decreases the number of trees in the forest by at least one, since each (non-root) internal node has at least two children. This means there are at most \( k \) internal nodes, for a total of \( O(k) \) nodes. ■
Patricia Tries

- **Theorem:** The number of nodes in a Patricia trie with $k$ words is always $O(k)$, regardless of what those words are.

Proof Sketch: There are $k$ leaves, one per word in the trie. Remove all internal nodes, leaving a forest of $k$ trees. Add the internal nodes back one at a time. Each addition (except possibly root) decreases the number of trees in the forest by at least one, since each (non-root) internal node has at least two children. This means there are at most $k$ internal nodes, for a total of $O(k)$ nodes. ■
**Theorem:** The number of nodes in a Patricia trie with $k$ words is always $O(k)$, regardless of what those words are.

**Proof Sketch:** There are $k$ leaves, one per word. Remove all internal nodes, leaving a forest of $k$ trees. Add the internal nodes back one at a time. Each addition (except possibly root) decreases the number of trees in the forest by at least one, since each (non-root) internal node has at least two children. This means there are at most $k$ internal nodes, for a total of $O(k)$ nodes. ■

Think of this as a forest of $k$ different trees.
Patricia Tries

- **Theorem:** The number of nodes in a Patricia trie with \( k \) words is always \( O(k) \), regardless of what those words are.

Proof Sketch:

There are \( k \) leaves, one per word. Remove all internal nodes, leaving a forest of \( k \) trees.

Add the internal nodes back one at a time. Each addition (except possibly root) decreases the number of trees in the forest by at least one, since each (non-root) internal node has at least two children. This means there are at most \( k \) internal nodes, for a total of \( O(k) \) nodes. ■

Adding an internal node merges two or more trees together, decreasing the number of trees by at least one.
• **Theorem:** The number of nodes in a Patricia trie with \( k \) words is always \( O(k) \), regardless of what those words are.

Proof Sketch:

There are \( k \) leaves, one per word. Remove all internal nodes, leaving a forest of \( k \) trees. Add the internal nodes back one at a time. Each addition (except possibly root) decreases the number of trees in the forest by at least one, since each (non-root) internal node has at least two children. This means there are at most \( k \) internal nodes, for a total of \( O(k) \) nodes. ■

Adding the root might not decrease the number of trees, but there's only one root.
Patricia Tries

- **Theorem:** The number of nodes in a Patricia trie with $k$ words is always $O(k)$, regardless of what those words are.

- **Proof Sketch:** There are $k$ leaves, one per word. Remove all internal nodes, leaving a forest of $k$ trees. Add the internal nodes back one at a time. Each addition (except possibly root) decreases the number of trees in the forest by at least one, since each (non-root) internal node has at least two children. This means there are at most $k$ internal nodes, for a total of $O(k)$ nodes. ■
Patricia Tries

• **Claim:** If each leaf in a Patricia trie is annotated with the index of the word it comes from, all strings starting with a given prefix can be found in time $O(n + z)$, where $n$ is the length of that prefix and $z$ is the number of matches.

• **Question:** How is this possible?
Patricia Tries

- Use a two-phase search algorithm!
  
  **(Character-aware)** Read the prefix to search for, matching characters as you walk down the Patricia trie.
  
  - Time required: $O(n)$, since we have to read all the characters of the prefix.
  
- **(Character-blind)** If you didn’t walk off the trie, do a DFS below your current point to find all leaves, ignoring the strings on the edges.
  
  - Time required: $O(z)$. If there are $z$ matches, there are $z$ leaves to explore. As we saw earlier, in a Patricia trie, a subtree with $z$ leaves has $O(z)$ total nodes.
The Story So Far

- Adopting our notation from RMQ, a Patricia trie gives an \(\langle O(m), O(n + z)\rangle\) solution to prefix matching.

- Those runtimes hide the effect of the alphabet size; take some time to evaluate those tradeoff's!
Part II: *Suffix Trees*
Two Motivating Problems
The *United States Statutes at Large* contains all legislation ever passed in the United States.

Make it searchable.
Cancers often have repeated copies of the same gene.

Given a cancer genome (length \( \sim 3,000,000,000 \)), find the longest repeated DNA sequence.
Patricia tries are great tools for finding *prefixes*. These problems involve looking for *substrings*. Can we use what we’ve developed so far?
A Fundamental Theorem

• The *fundamental theorem of stringology* says that, given two strings $w$ and $x$, that

\[w\text{ is a substring of } x\]

*if and only if*

\[w\text{ is a prefix of a suffix of } x\]
A Fundamental Theorem

- The *fundamental theorem of stringology* says that, given two strings $w$ and $x$, that

  \[ w \text{ is a substring of } x \text{ if and only if } w \text{ is a prefix of a suffix of } x. \]
A Fundamental Theorem

- The *fundamental theorem of stringology* says that, given two strings $w$ and $x$, that

  \[
  w \text{ is a substring of } x \\
  \text{if and only if} \\
  w \text{ is a prefix of a suffix of } x
  \]
A Fundamental Theorem

- The *fundamental theorem of stringology* says that, given two strings $w$ and $x$, that

  $w$ is a substring of $x$

  *if and only if*

  $w$ is a prefix of a suffix of $x$
A Fundamental Theorem

• The *fundamental theorem of stringology* says that, given two strings \( w \) and \( x \), that

\[
\text{\( w \) is a substring of \( x \) if and only if}
\]

\( w \) is a prefix of a suffix of \( x \)
A Fundamental Theorem

- The *fundamental theorem of stringology* says that, given two strings \( w \) and \( x \), that

  \[
  w \text{ is a substring of } x \quad \text{if and only if} \quad w \text{ is a prefix of a suffix of } x
  \]

\[f l i b b e r t i g i b b e t\]
A Fundamental Theorem

• The *fundamental theorem of stringology* says that, given two strings $w$ and $x$, that

  \[
  \text{w is a substring of } x \\
  \text{if and only if} \\
  \text{w is a prefix of a suffix of } x
  \]

• To find all matches of $w$ in $x$, we just need to find all suffixes of $x$ that start with $w$. 
Suffix Trees

- A **suffix tree** for a string $T$ is a Patricia trie of all suffixes of $T$.
- Each leaf is labeled with the starting index of that suffix.
Claim: Once we have a suffix tree for a string $T$, we can find all matches of a pattern $P$ of length $n$ in time $O(n + z)$, where $z$ is the number of matches.

Idea: Use the standard Patricia trie search from before!
Substring Search

- **Algorithm:** Use the standard Patricia trie search!

- Look up the pattern in the suffix tree, then use a DFS to find all matches.

- Looking up the pattern takes time $O(n)$.

- Finding all matches takes time $O(z)$. 

```plaintext
nonsense$
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
```
Cancers often have repeated copies of the same gene.

Given a cancer genome (length $\sim 3,000,000,000$), find the longest repeated DNA sequence.
The Anatomy of a Suffix Tree

- Think back to Cartesian trees. We can describe them in two ways.
  - **Mechanically:** Hoist the minimum element up to the root, then recursively process the two subarrays.
  - **Operationally:** It’s a min-heap whose inorder traversal gives the original array.
- We now have a mechanical definition of a suffix tree. Can we get an operational one?
The Anatomy of a Suffix Tree

- The leaves of a suffix tree correspond to the suffixes of the text string $T$.
- **Question:** What do the *internal* nodes of the suffix tree correspond to?
The Anatomy of a Suffix Tree

- In this suffix tree, there are internal nodes for the substrings $e$, $n$, $nse$, and $se$.
- All these substrings appear at least twice in the original string!
- More generally: if there is an internal node for a substring $\alpha$, then $\alpha$ appears at least twice in the original text.
The Anatomy of a Suffix Tree

- **Question**: why is there an internal node for the substring n, but isn’t there an internal node for the substring ns?
- Every occurrence of ns can be extended by appending the same character (e)
- Not all occurrences of n can be extended by appending the same character.
The Anatomy of a Suffix Tree

- **branching word** in $T\$ is a string $\omega$ such that there are characters $a \neq b$ where $\omega a$ and $\omega b$ are substrings of $T\$.
- Edge case: the empty string is always considered branching.
- **Theorem**: The suffix tree for a string $T$ has an internal node for a string $\omega$ if and only if $\omega$ is a branching word in $T\$. 

The image shows a suffix tree with the string "nonsense$" and the corresponding index 012345678.
• Combining our previous points together, we can give a (partial) operational definition of a suffix tree:

The leaves of a suffix tree for $T$ correspond to suffixes of $T\$, and the internal nodes of a suffix tree for $T$ correspond to branching words of $T\$.

• We’ll make extensive use of this fact going forward.
Longest Repeated Substrings

• **Theorem:** The longest repeated substring of a string $T$ must be a branching word in $T$.

• **Proof idea:** If $\omega$ isn’t branching, it can’t be the longest repeated substring.
Longest Repeated Substrings

- **Theorem:** The longest repeated substring of a string $T$ must be a branching word in $T\$.
- **Proof idea:** If $\omega$ isn’t branching, it can’t be the longest repeated substring.

The substring `berti` isn’t repeated.

It therefore can’t be the longest repeated substring.
Longest Repeated Substrings

• **Theorem:** The longest repeated substring of a string $T$ must be a branching word in $T$.

• **Proof idea:** If $\omega$ isn’t branching, it can’t be the longest repeated substring.
Longest Repeated Substrings

- **Theorem**: The longest repeated substring of a string $T$ must be a branching word in $T$.
- **Proof idea**: If $\omega$ isn’t branching, it can’t be the longest repeated substring.

Every instance of $bb$ can be extended to $bbe$. It therefore can’t be the longest repeated substring.
Theorem: The longest repeated substring of $T$ is a branching word in $T$.

To find the longest repeated substring of a string $T$, we just need to find the internal node with the longest label!
Longest Repeated Substrings

- Given a suffix tree for a string $T$ of length $m$, there is an $O(m)$-time algorithm for finding the longest repeated substring of $m$.

- **Basic idea:** Run a DFS over the tree and find the internal node with the longest string on its path from the root.

- There are some subtle details required to get this to run in time $O(m)$. Think this over! See what you find.
More to Explore

• We’ve barely scratched the surface of suffix trees. They can be used for tons of other problems.

• A sampling:
  
  • **Generalized suffix trees**: Solves fast substring searching over multiple text strings, not just a single text string.
  
  • **Approximate string matching**: Given a text string $T$ and a pattern $P$, see the closest match to $P$ in $T$.
  
  • **Fast matrix multiplication**: The matrix multiplications needed in computing word embeddings can, amazingly, be optimized using suffix trees.

• This is a rich space to explore for a research project, if that’s something you’d like to do!
Next Time

• *Suffix Arrays*
  • A space-efficient alternative to suffix trees.

• *LCP Arrays*
  • Implicitly capturing suffix tree structure.