

Suffix Trees

- String Indexing/String Dictionaries
- Tries
- Suffix Trees

Inge Li Gørtz

Suffix Trees

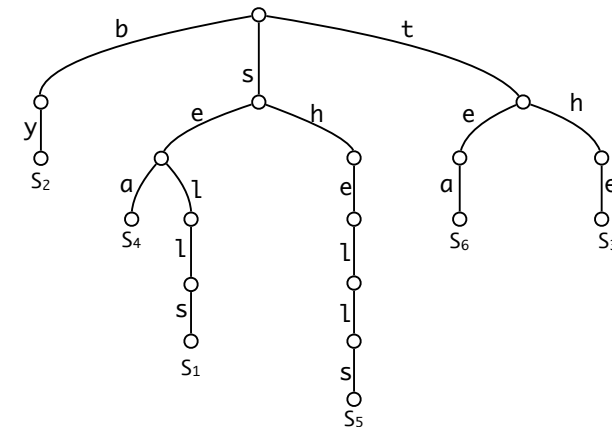
- String Dictionaries
- Tries
- Suffix Trees

String Indexing

- **String indexing problem.** Let S be a string of characters from alphabet Σ . Preprocess S into data structure to support:
 - $\text{search}(P)$: Return the starting positions of all occurrences of P in S .
- **Example.**
 - $S = \text{yabbadabbado}$
 - $\text{search}(\text{abba}) = \{1,6\}$
- **String dictionary problem.** Let $S = \{S_1, S_2, \dots, S_k\}$ be a set of strings of characters from alphabet Σ . Preprocess S into data structure to support:
 - $\text{search}(P)$: Return yes if $P = S_i$ for some S_i in S .
 - $\text{prefix-search}(P)$: Return all strings in S for which P is a prefix.

Tries

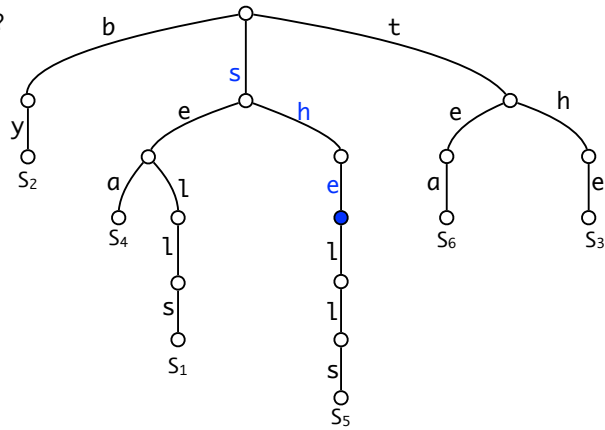
- Text retrieval



- Trie over the strings: sells, by, the, sea, shells, tea.

Tries

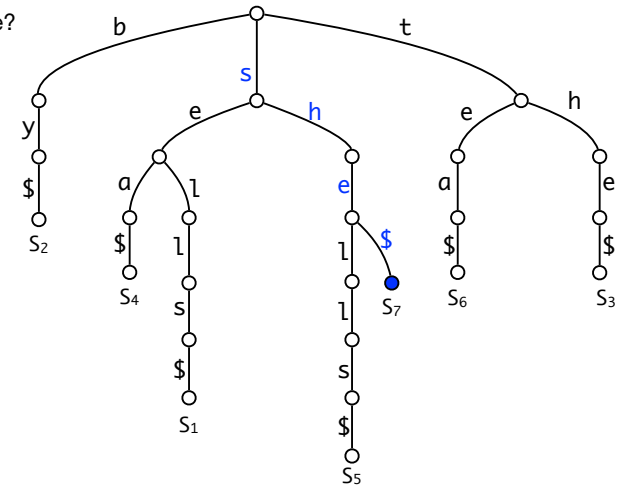
- Text retrieval
- Prefix-free?



- Trie over the strings: sells, by, the, sea, shells, tea, she.

Tries

- Text retrieval
- Prefix-free?



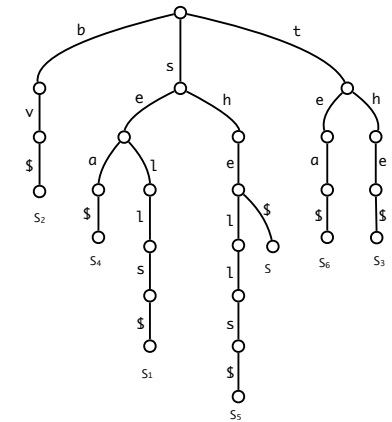
- Trie over the strings: sells\$, by\$, the\$, sea\$, shells\$, tea\$, she\$.

Tries

- Fredkin 1960]. Retrieval. Store a set of strings in a rooted tree such that:
 - Each edge is labeled by a character. Edges to children of a node are sorted from left-to-right alphabetically.
 - Each root-to-leaf path represents a string in the set. (obtained by concatenating the labels of edges on the path).
 - Common prefixes share same path maximally.
- Properties of the trie. A trie T storing a collection S of s strings of total length n from an alphabet of size d has the following properties:
 - How many children can a node have?
 - How many leaves does T have?
 - What is the height of T ?
 - What is the number of nodes in T ?

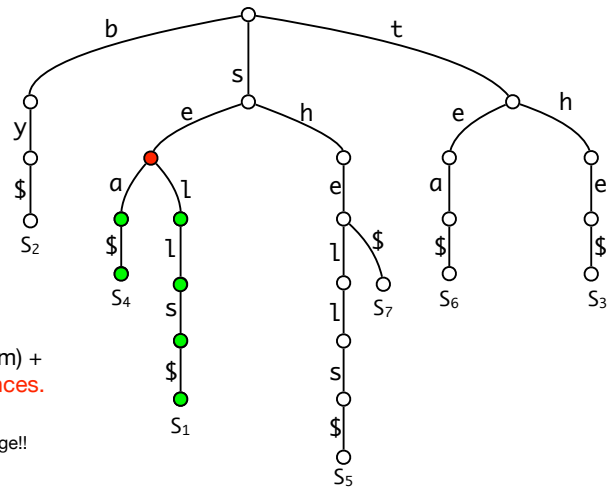
Trie

- Search time: $O(d)$ in each node $\Rightarrow O(dm)$.
 - $O(m)$ if d constant.
 - d not constant: use dictionary
 - Perfect hashing $O(1)$
 - Balanced BST: $O(\log d)$
- Time and space for a trie (for constant d):
 - $O(m)$ for searching for a string of length m .
 - $O(n)$ space.
 - Preprocessing: $O(n)$



Tries

- Prefix search: return all words in the trie starting with "se"



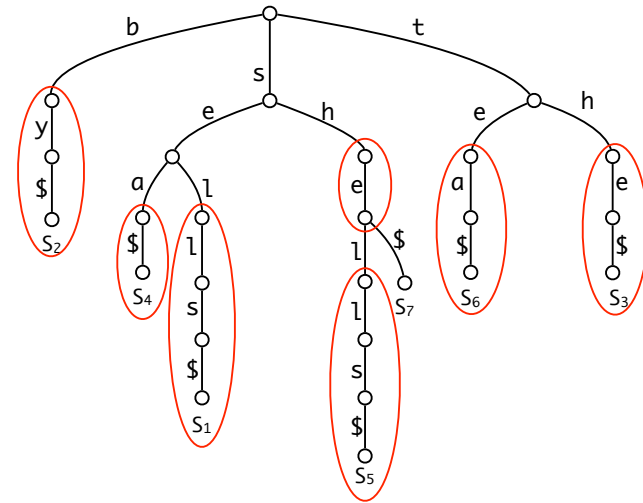
- Time for prefix search: $O(m)$ + time to report all occurrences.

Could be large!!

- Solution: compact tries.

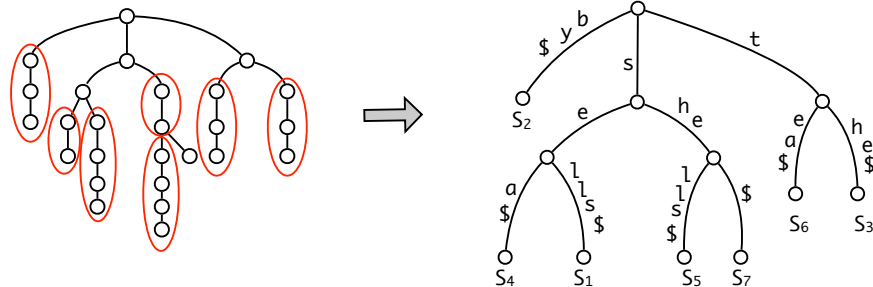
Tries

- Compact trie. Chains of nodes with a single child is merged into a single edge.



Tries

- Compact trie. Chains of nodes with a single child is merged into a single edge.



- Properties of the compact trie. A compact trie T storing a collection S of s strings of total length n from an alphabet of size d has the following properties:

- Every internal node of T has at least 2 and at most d children.
- T has s leaves
- The number of nodes in T is $\leq 2s$.

Trie

- Time and space for a compact trie (constant d).
 - $O(m)$ for searching for a string of length m .
 - $O(m + \text{occ})$ for prefix search, where $\text{occ} = \#\text{occurrences}$
 - $O(n)$ space.
 - Preprocessing: $O(n)$

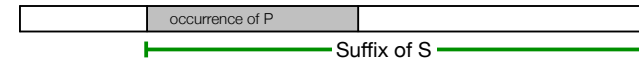
Suffix Trees

- String Dictionaries
- Tries
- Suffix Trees

Suffix tree

- **String indexing problem.** Given a string S of characters from an alphabet Σ . Preprocess S into a data structure to support
 - Search(P): Return starting position of all occurrences of P in S.

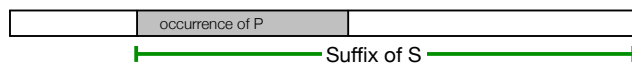
- Observation: An occurrence of P is a prefix of a suffix of S.



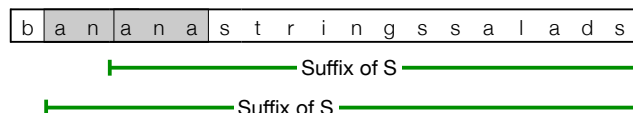
Suffix tree

- **String indexing problem.** Given a string S of characters from an alphabet Σ . Preprocess S into a data structure to support
 - Search(P): Return starting position of all occurrences of P in S.

- Observation: An occurrence of P is a prefix of a suffix of S.

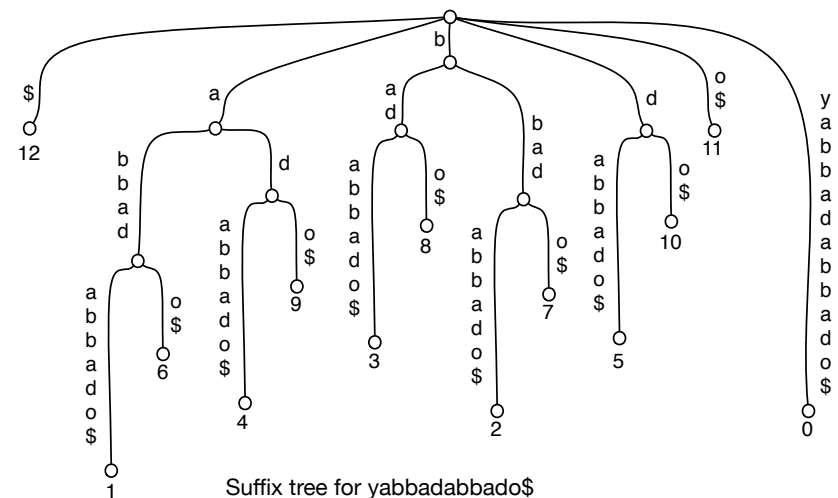


- Example: P = ana.



Suffix Trees

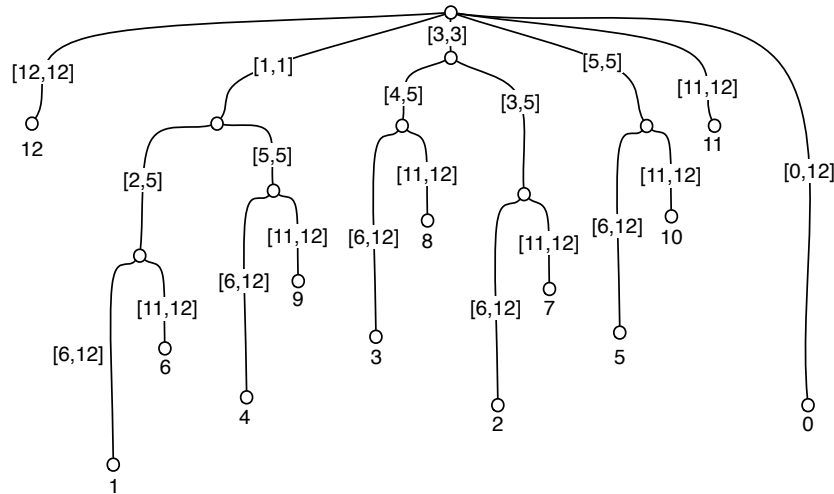
- **Suffix trees.** The compact trie of all suffixes of S.



Suffix Trees

- **Suffix trees.** The **compact trie** of all suffixes of S.
- Store S and store edge labels by reference to S.

0 1 2 3 4 5 6 7 8 9 10 11 12
y a b b a d a b b a d o \$

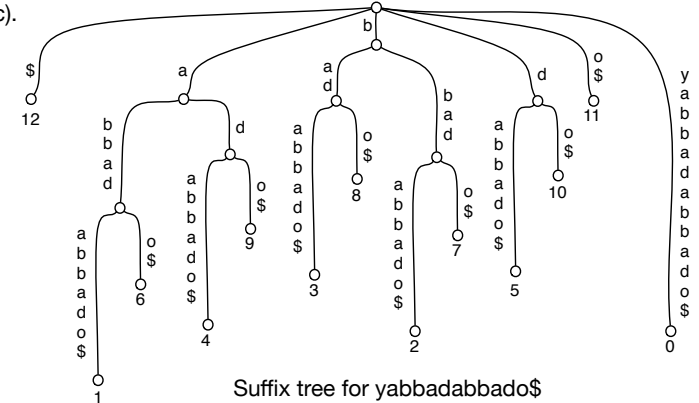


Suffix Trees

- **Theorem.** We can solve the string indexing problem in
 - $O(n)$ space and $\text{sort}(n, |\Sigma|)$ preprocessing time.
 - $O(m + \text{occ})$ time for queries.

Suffix Trees

- **Space.**
 - Number of edges + space for edge labels
 - $\Rightarrow O(n)$ space
- **Preprocessing.** $O(\text{sort}(n, |\Sigma|))$
- $\text{sort}(n, |\Sigma|)$ = time to sort n characters from an alphabet Σ .
- **Search(P):** $O(m + \text{occ})$.

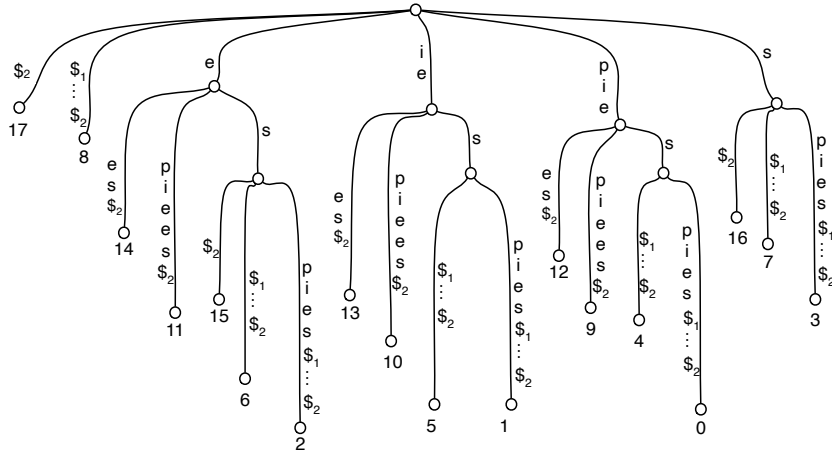


Suffix Trees

- **Applications.**
 - Approximate string matching problems
 - Compression schemes (Lempel-Ziv family, ...)
 - Repetitive string problems (palindromes, tandem repeats, ...)
 - Information retrieval problems (document retrieval, top-k retrieval, ...)
 - ...

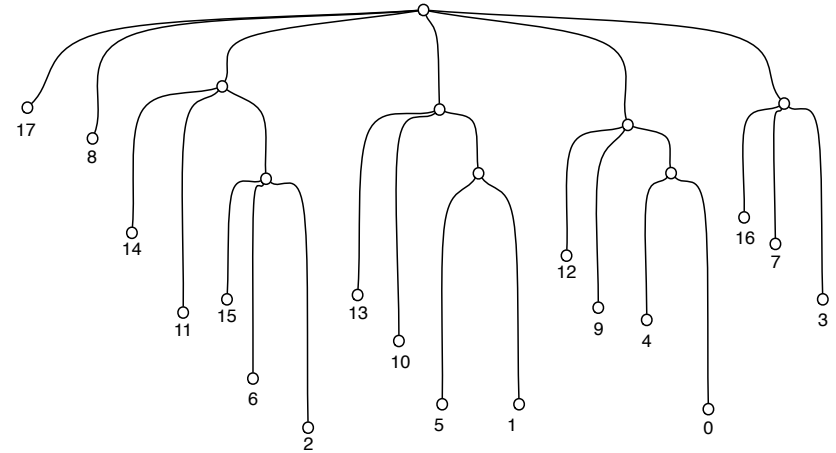
Longest common substring

- Find **longest common substring** of strings S_1 and S_2 .
- Construct the suffix tree over S_1S_2 .
- Example.** Find longest common substring of **piespies** and **piepies**:
 - Construct suffix tree of **piespies** $_1$ **piepies** $_2$.



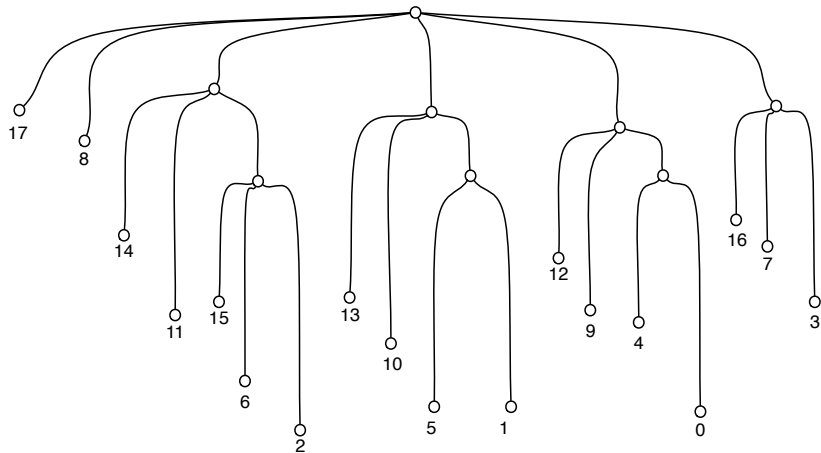
Longest common substring

- Suffix tree of **piespies** $_1$ **piepies** $_2$.



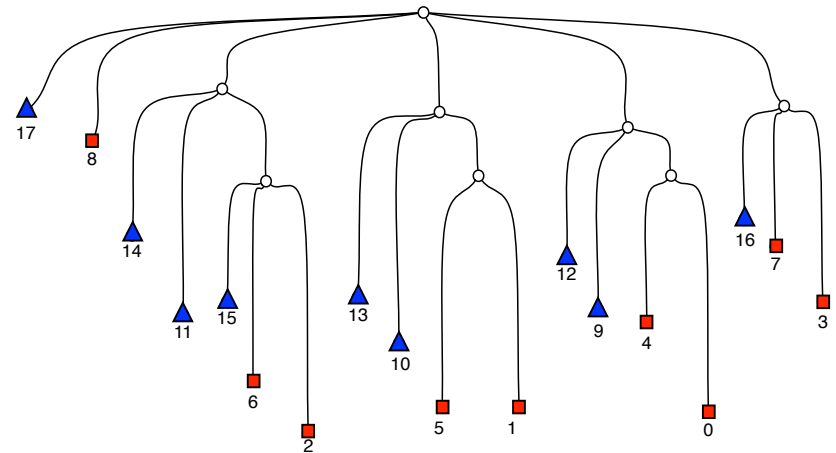
Longest common substring

- Suffix tree of **piespies** $_1$ **piepies** $_2$.
- Mark leaves: ■ = S_1 ▲ = S_2



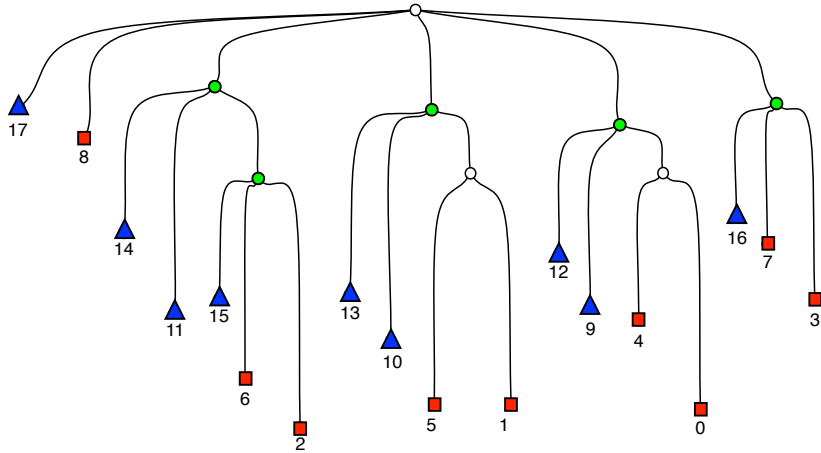
Longest common substring

- Suffix tree of **piespies** $_1$ **piepies** $_2$.
- Mark leaves: ■ = S_1 ▲ = S_2



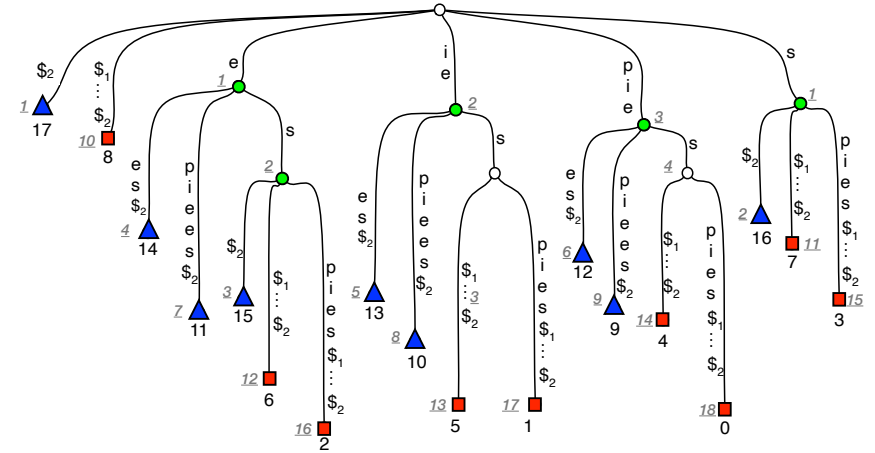
Longest common substring

- Suffix tree of $\text{piespies}\$_1\text{piepiees}\$_2$.
- Mark leaves: $\blacksquare = S_1$ $\blacktriangle = S_2$



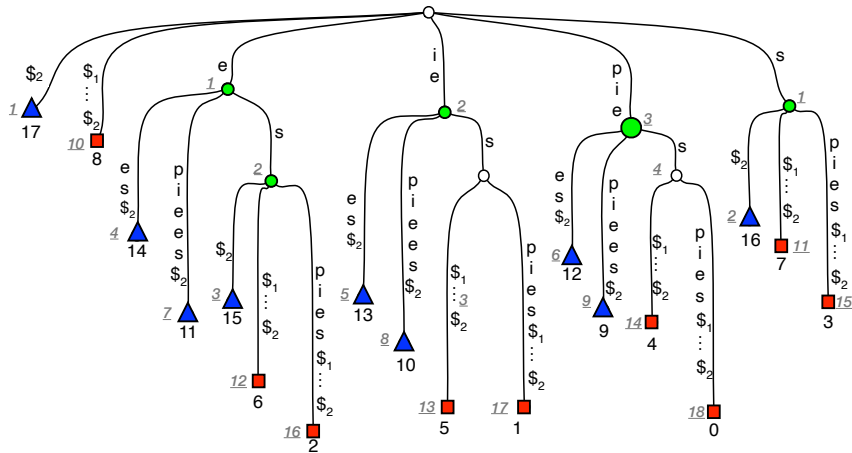
Longest common substring

- Suffix tree of $\text{piespies}\$_1\text{piepiees}\$_2$.
- Mark leaves: $\blacksquare = S_1$ $\blacktriangle = S_2$
- Add string depth.



Longest common substring

- Suffix tree of $\text{piespies}\$_1\text{piepiees}\$_2$.
- Mark leaves: $\blacksquare = S_1$ $\blacktriangle = S_2$
- Add string depth.



Longest common substring

- Using a suffix tree we can solve the longest common substring problem in linear time (for a constant size alphabets).