Advanced Data Structures for Sequence Analysis

Solon P. Pissis

Department of Informatics, King's College London, London, UK

solon.pissis@kcl.ac.uk

October 26, 2018

University of Central Florida, Orlando,FL

Let us have only a glimpse...

Preliminaries: Suffix Trees

Preliminaries: Suffix Trees

Let $T = T[0 \,..\, n-1]$ be our text and $T_i = T[i \,..\, n-1]$, $i \in [0, n]$.

Preliminaries: Suffix Trees

Let $T = T[0 \mathinner{.\,.} n-1]$ be our text and $T_i = T[i \mathinner{.\,.} n-1]$, $i \in [0, n]$.

For any subset $C \in [0, n]$, let $T_C = \{T_i | i \in C\}$.

Preliminaries: Suffix Trees

Let $T = T[0 \,..\, n-1]$ be our text and $T_i = T[i \,..\, n-1]$, $i \in [0, n]$.

For any subset $C \in [0, n]$, let $T_C = \{T_i | i \in C\}$.

Thus $T_{[0..n]}$ contains $n+1$ strings of total length $\Theta(n^2)$.

Preliminaries: Suffix Trees

Let $T = T[0 \mathinner{.\,.} n-1]$ be our text and $T_i = T[i \mathinner{.\,.} n-1]$, $i \in [0, n]$.

For any subset $C \in [0, n]$, let $T_C = \{T_i | i \in C\}$.

Thus $T_{[0..n]}$ contains $n+1$ strings of total length $\Theta(n^2)$.

Suffix tree is a **compact trie** for $T_{[0..n]}$: the set of all suffixes of T.

Preliminaries: Suffix Trees

Let $T = T[0 \mathinner{.\,.} n-1]$ be our text and $T_i = T[i \mathinner{.\,.} n-1]$, $i \in [0, n]$.

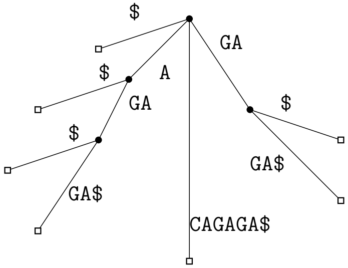
For any subset $C \in [0, n]$, let $T_C = \{T_i | i \in C\}$.

Thus $T_{[0 \mathinner{.\,.} n]}$ contains $n+1$ strings of total length $\Theta(n^2)$.

Suffix tree is a **compact trie** for $T_{[0 \mathinner{.\,.} n]}$: the set of all suffixes of T.

Example

Let $T =$ `CAGAGA$`.



Preliminaries: Suffix Trees

Preliminaries: Suffix Trees

We assume there is an extra **unique** letter \$ at the end of T:

Preliminaries: Suffix Trees

We assume there is an extra **unique** letter \$ at the end of T:

- No suffix is a prefix of another suffix: $T_{[0..n]}$ is **prefix free**.

Preliminaries: Suffix Trees

We assume there is an extra **unique** letter \$ at the end of T:

- No suffix is a prefix of another suffix: $T_{[0..n]}$ is **prefix free**.
- All nodes in the suffix tree representing a suffix are **leaves**.

Preliminaries: Suffix Trees

We assume there is an extra **unique** letter \$ at the end of T:

- No suffix is a prefix of another suffix: $T_{[0..n]}$ is **prefix free**.
- All nodes in the suffix tree representing a suffix are **leaves**.

This simplifies algorithms!

Preliminaries: Suffix Trees

We assume there is an extra **unique** letter \$ at the end of T:

- No suffix is a prefix of another suffix: $T_{[0..n]}$ is **prefix free**.
- All nodes in the suffix tree representing a suffix are **leaves**.

This simplifies algorithms!

We also decorate **leaves** with starting positions from T.

Preliminaries: Suffix Trees

We assume there is an extra **unique** letter \$ at the end of T:

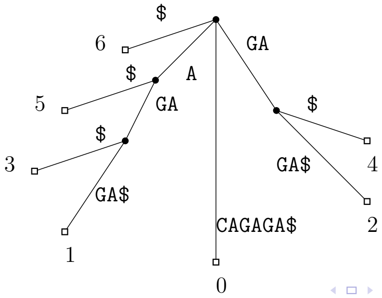
- No suffix is a prefix of another suffix: $T_{[0..n]}$ is **prefix free**.
- All nodes in the suffix tree representing a suffix are **leaves**.

This simplifies algorithms!

We also decorate **leaves** with starting positions from T.

Example

Let $T = \texttt{CAGAGA\$}$.



Preliminaries: Suffix Trees

Preliminaries: Suffix Trees

Why compact?

Preliminaries: Suffix Trees

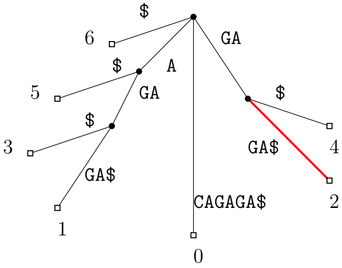
Why compact? How much space does it take?

Preliminaries: Suffix Trees

Why compact? How much space does it take?

Example

Let $T =$ CAGAGA$.

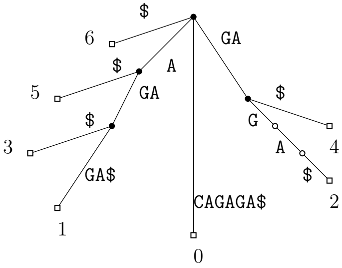


Preliminaries: Suffix Trees

Why compact? How much space does it take?

Example

Let $T =$ CAGAGA$.

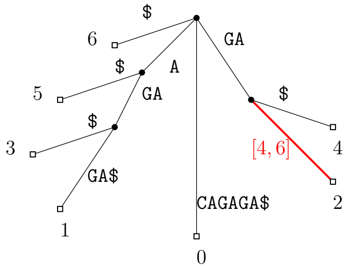


Preliminaries: Suffix Trees

Why compact? How much space does it take?

Example

Let $T = \texttt{CAGAGA\$}$.

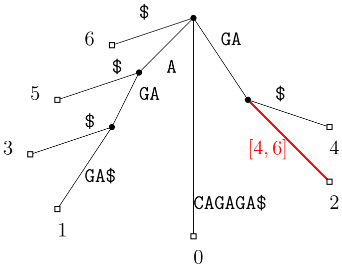


Preliminaries: Suffix Trees

Why compact? How much space does it take?

Example

Let $T =$ CAGAGA$.



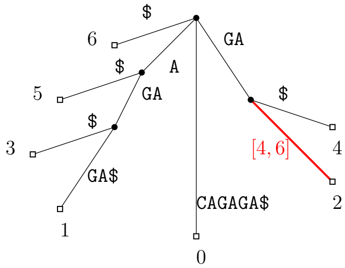
- Edge labels are substrings of T: represented by T intervals.

Preliminaries: Suffix Trees

Why compact? How much space does it take?

Example

Let $T =$ CAGAGA$.



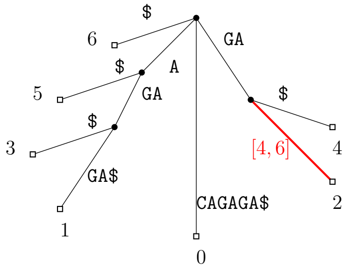
- Edge labels are substrings of T: represented by T intervals.
- Exactly $n+1$ leaves and at most n internal nodes.

Preliminaries: Suffix Trees

Why compact? How much space does it take?

Example

Let $T =$ CAGAGA$.



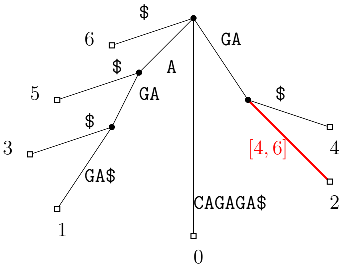
- Edge labels are substrings of T: represented by T intervals.
- Exactly $n + 1$ leaves and at most n internal nodes.
- At most $2n$ edges.

Preliminaries: Suffix Trees

Why compact? How much space does it take?

Example

Let $T =$ CAGAGA$.



- Edge labels are substrings of T: represented by T intervals.
- Exactly $n+1$ leaves and at most n internal nodes.
- At most $2n$ edges.

Space linear in n: $O(n)$.

Preliminaries: Suffix Trees

What about construction?

Preliminaries: Suffix Trees

What about construction?

Theorem (Farach, FOCS 1997)

Let T be a string of length n over a linearly-sortable alphabet. The suffix tree of T can be constructed in $O(n)$ time.

Preliminaries: Suffix Trees

What about construction?

Theorem (Farach, FOCS 1997)

Let T be a string of length n over a linearly-sortable alphabet. The suffix tree of T can be constructed in $O(n)$ time.

Linearly-sortable alphabet: $\Sigma = \{1, 2, \ldots, n^{O(1)}\}$.

Preliminaries: Suffix Trees

What about construction?

Theorem (Farach, FOCS 1997)

Let T be a string of length n over a linearly-sortable alphabet. The suffix tree of T can be constructed in $O(n)$ time.

Linearly-sortable alphabet: $\Sigma = \{1, 2, \ldots, n^{O(1)}\}$.

Farach's algorithm is in fact optimal for all alphabets!

Preliminaries: Suffix Trees

What about construction?

Theorem (Farach, FOCS 1997)

Let T be a string of length n over a linearly-sortable alphabet. The suffix tree of T can be constructed in $O(n)$ time.

Linearly-sortable alphabet: $\Sigma = \{1, 2, \ldots, n^{O(1)}\}$.

Farach's algorithm is in fact optimal for all alphabets!

In bioinformatics we usually have that $\Sigma = O(1)$.

Preliminaries: Suffix Trees

An important auxiliary tool are **suffix links**:

Preliminaries: Suffix Trees

An important auxiliary tool are **suffix links**:

- Let S_u denote the string represented by node u.

Preliminaries: Suffix Trees

An important auxiliary tool are **suffix links**:

- Let S_u denote the string represented by node u.
- $slink(u)$ is the node v such that S_v is the longest proper suffix of S_u, i.e., if $S_u = T[i\,..\,j]$ then $S_v = T[i+1\,..\,j]$.

Preliminaries: Suffix Trees

An important auxiliary tool are **suffix links**:

- Let S_u denote the string represented by node u.
- $slink(u)$ is the node v such that S_v is the longest proper suffix of S_u, i.e., if $S_u = T[i \,..\, j]$ then $S_v = T[i+1 \,..\, j]$.
- Constructible for all **internal nodes** in $O(n)$ time.

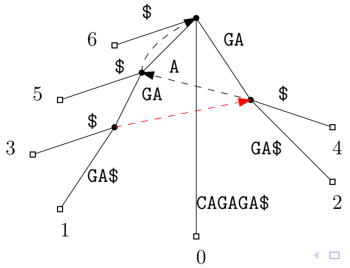
Preliminaries: Suffix Trees

An important auxiliary tool are **suffix links**:

- Let S_u denote the string represented by node u.
- $\mathit{slink}(u)$ is the node v such that S_v is the longest proper suffix of S_u, i.e., if $S_u = T[i..j]$ then $S_v = T[i+1..j]$.
- Constructible for all **internal nodes** in $O(n)$ time.

Example

Let $T = \texttt{CAGAGA\$}$. $S_u = \texttt{AGA}$ and $S_v = \texttt{GA}$.



Application 1: Exact string matching

PREPROCESS: text T

QUERY: a pattern P; return all **occ** starting positions of P in T

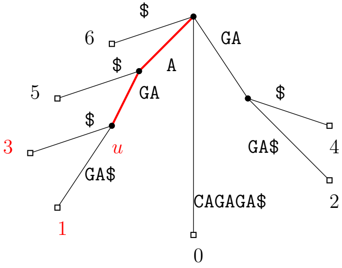
Application 1: Exact string matching

PREPROCESS: text T

QUERY: a pattern P; return all **occ** starting positions of P in T

Example

Let $T =$ CAGAGA\$ and $P =$ AGA.



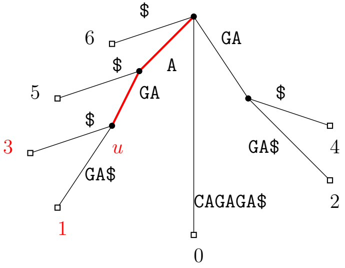
Application 1: Exact string matching

PREPROCESS: text T

QUERY: a pattern P; return all **occ** starting positions of P in T

Example

Let $T = \texttt{CAGAGA\$}$ and $P = \texttt{AGA}$.



Traverse the subtree rooted at u with $S_u = P$.

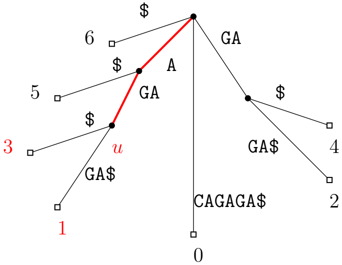
Application 1: Exact string matching

PREPROCESS: text T

QUERY: a pattern P; return all **occ** starting positions of P in T

Example

Let $T = \texttt{CAGAGA\$}$ and $P = \texttt{AGA}$.



Traverse the subtree rooted at u with $S_u = P$. Its size is $O(\text{occ})$.

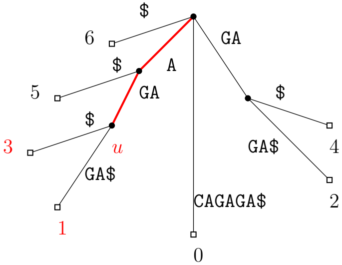
Application 1: Exact string matching

PREPROCESS: text T

QUERY: a pattern P; return all **occ** starting positions of P in T

Example

Let $T = \texttt{CAGAGA\$}$ and $P = \texttt{AGA}$.



Traverse the subtree rooted at u with $S_u = P$. Its size is $O(\text{occ})$.

Theorem

Exact string matching queries can be answered in $O(|P| + occ)$ time after $O(n)$ time preprocessing.

Application 2: Number of distinct substrings

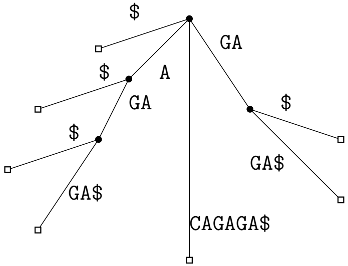
INPUT: text T
OUTPUT: the number of distinct substrings

Application 2: Number of distinct substrings

INPUT: text T
OUTPUT: the number of distinct substrings

Example

Let $T =$ CAGAGA$.

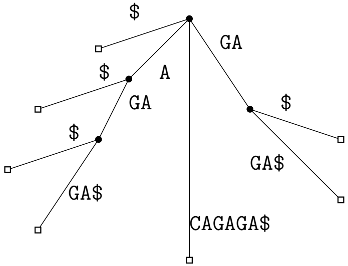


Application 2: Number of distinct substrings

INPUT: text T
OUTPUT: the number of distinct substrings

Example

Let $T =$ CAGAGA$.



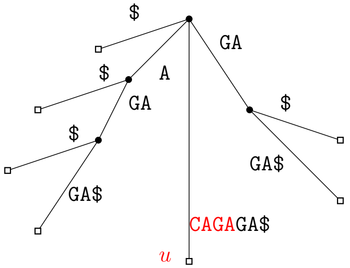
Every *locus* **(node, depth)** in the suffix tree represents a substring of the text and every substring is represented by some locus.

Application 2: Number of distinct substrings

INPUT: a text T
OUTPUT: the number of distinct substrings

Example

Let $T =$ CAGAGA\$. Locus $(u, 4)$ represents CAGA.

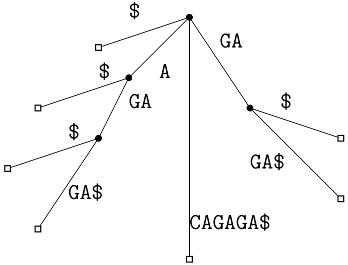


Every *locus* **(node, depth)** in the suffix tree represents a substring of the text and every substring is represented by some locus.

INPUT: text T
OUTPUT: the number of distinct substrings

Example

Let $T =$ CAGAGA\$.

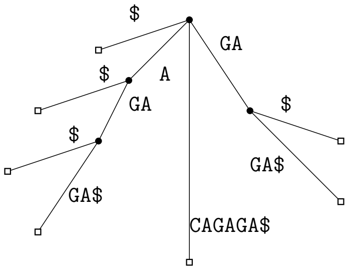


Application 2: Number of distinct substrings

INPUT: text T
OUTPUT: the number of distinct substrings

Example

Let $T = \texttt{CAGAGA\$}$.



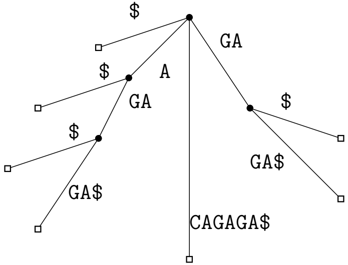
Count the number of distinct loci using a suffix tree traversal.

Application 2: Number of distinct substrings

INPUT: text T
OUTPUT: the number of distinct substrings

Example

Let $T =$ CAGAGA\$.

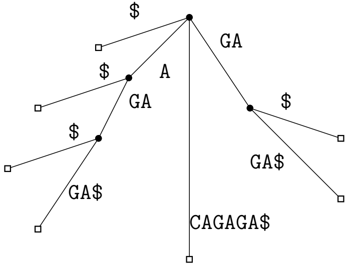


Application 2: Number of distinct substrings

INPUT: text T
OUTPUT: the number of distinct substrings

Example

Let $T = \texttt{CAGAGA\$}$.



Theorem

The number of distinct substrings can be computed in $O(n)$ time.

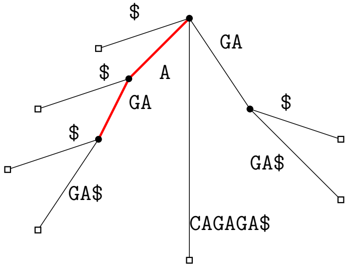
Application 3: Longest repeating substring

INPUT: text T

OUTPUT: a longest string occurring at least twice

Example

Let $T =$ CAGAGA\$. The answer is AGA.



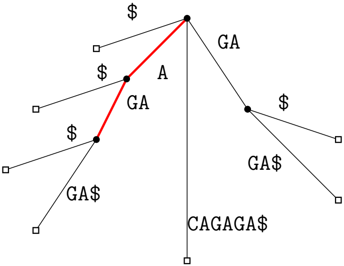
Find a deepest internal node using a traversal of the suffix tree.

Application 3: Longest repeating substring

INPUT: text T
OUTPUT: a longest string occurring at least twice

Example

Let $T =$ CAGAGA\$. The answer is AGA.



Find a deepest internal node using a traversal of the suffix tree.

Theorem

A longest repeating substring can be found in $O(n)$ time.

Application 4: Longest common substring

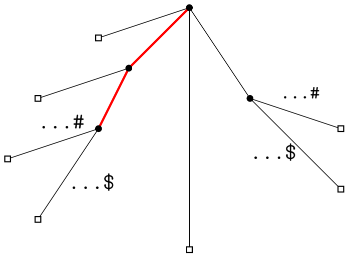
INPUT: text T and a text S
OUTPUT: a longest common substring

Application 4: Longest common substring

INPUT: text T and a text S
OUTPUT: a longest common substring

Example

Suffix tree of $T\#S\$$.

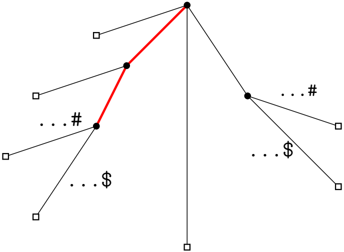


Application 4: Longest common substring

INPUT: text T and a text S
OUTPUT: a longest common substring

Example

Suffix tree of $T\#S\$$.



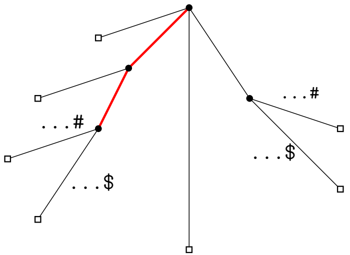
Find a deepest internal node containing both T- and S-leaves.

Application 4: Longest common substring

INPUT: text T and a text S
OUTPUT: a longest common substring

Example

Suffix tree of $T\#S\$$.



Find a deepest internal node containing both T- and S-leaves.

Theorem

A longest common substring can be found in $O(n + |S|)$ time.

Application 5: Matching statistics

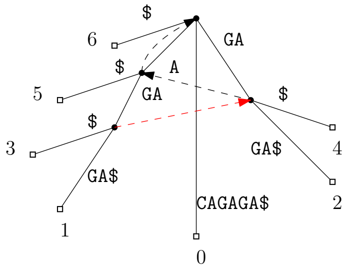
PREPROCESS: text T
QUERY: a text S; return the longest prefix of $S[i\,..]$ that is a substring of T, for all $i \in [0, |S|-1]$

Application 5: Matching statistics

PREPROCESS: text T
QUERY: a text S; return the longest prefix of $S[i\,..]$ that is a substring of T, for all $i \in [0, |S| - 1]$

Example

Let $T =$ CAGAGA\$.



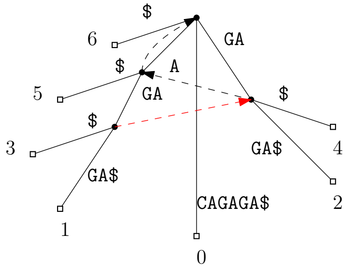
Scan S using the suffix tree of T.

Application 5: Matching statistics

PREPROCESS: text T
QUERY: a text S; return the longest prefix of $S[i\,..]$ that is a substring of T, for all $i \in [0, |S|-1]$

Example

Let $T =$ CAGAGA$.



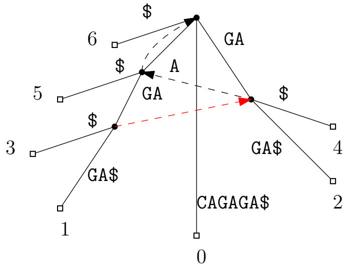
Spell $S[i\,..]$ as much as possible;

Application 5: Matching statistics

PREPROCESS: text T
QUERY: a text S; return the longest prefix of $S[i\,..]$ that is a substring of T, for all $i \in [0, |S|-1]$

Example

Let $T = \texttt{CAGAGA\$}$.



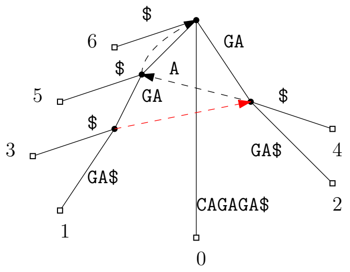
Spell $S[i\,..]$ as much as possible; say $S[i\,..\,j-1]$.

Application 5: Matching statistics

PREPROCESS: text T
QUERY: a text S; return the longest prefix of $S[i\,..]$ that is a substring of T, for all $i \in [0, |S|-1]$

Example

Let $T =$ CAGAGA\$.



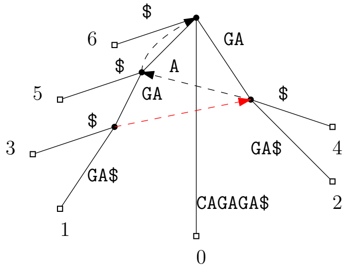
Mismatch at $S[i\,..\,j]$?

Application 5: Matching statistics

PREPROCESS: text T
QUERY: a text S; return the longest prefix of $S[i\,..]$ that is a substring of T, for all $i \in [0, |S|-1]$

Example

Let $T =$ CAGAGA\$.



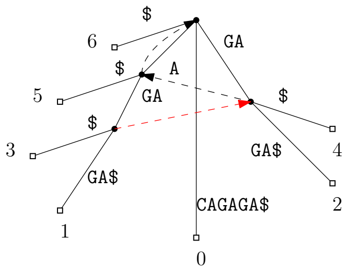
Mismatch at $S[i\,..\,j]$? Use suffix link as the failure transition!

Application 5: Matching statistics

PREPROCESS: text T
QUERY: a text S; return the longest prefix of $S[i\,..]$ that is a substring of T, for all $i \in [0, |S| - 1]$

Example

Let $T =$ CAGAGA\$.

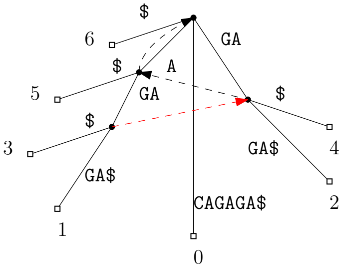


This takes us at node u: $S_u = S[i+1\,..\,j]$.

PREPROCESS: text T
QUERY: a text S; return the longest prefix of $S[i\,..]$ that is a substring of T, for all $i \in [0, |S|-1]$

Example

Let $T =$ CAGAGA$.



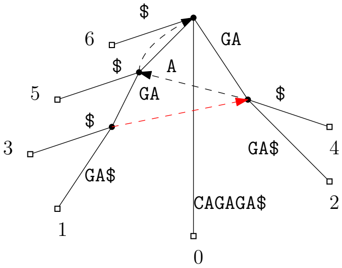
This takes us at node u: $S_u = S[i+1\,..\,j]$. Repeat from here!

Application 5: Matching statistics

PREPROCESS: text T
QUERY: a text S; return the longest prefix of $S[i\,..]$ that is a substring of T, for all $i \in [0, |S|-1]$

Example

Let $T =$ CAGAGA\$.



Theorem

Matching statistics of S with respect to T can be computed in $O(|S|)$ time after $O(n)$ time preprocessing.

Application 6: Longest common prefix

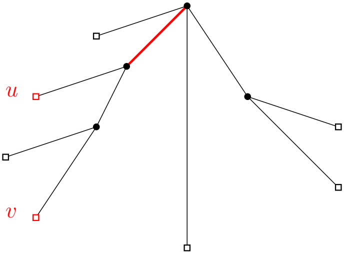
PREPROCESS: text T
QUERY: a pair (i, j); return the longest common prefix of $T[i\,..]$ and $T[j\,..]$

Application 6: Longest common prefix

PREPROCESS: text T

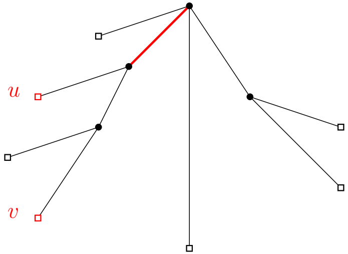
QUERY: a pair (i, j); return the longest common prefix of $T[i\,..]$ and $T[j\,..]$

The lowest common ancestor (LCA) of two nodes u and v is the deepest node that is an ancestor of both u and v.



Application 6: Longest common prefix

PREPROCESS: text T
QUERY: a pair (i, j); return the longest common prefix of $T[i\,..]$ and $T[j\,..]$
The lowest common ancestor (LCA) of two nodes u and v is the deepest node that is an ancestor of both u and v.



Theorem (Bender and Farach-Colton, LATIN 2000)

Any tree of size $O(N)$ can be preprocessed in $O(N)$ time so that the LCA of any two nodes can be computed in $O(1)$ time.

Application 6: Longest common prefix

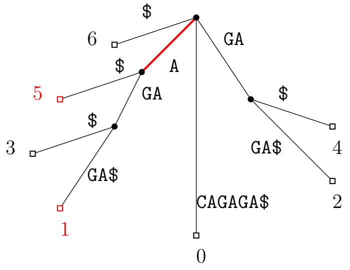
PREPROCESS: text T
QUERY: a pair (i, j); return the longest common prefix of $T[i\,..]$ and $T[j\,..]$

Application 6: Longest common prefix

PREPROCESS: text T
QUERY: a pair (i, j); return the longest common prefix of $T[i\,..]$ and $T[j\,..]$

Example

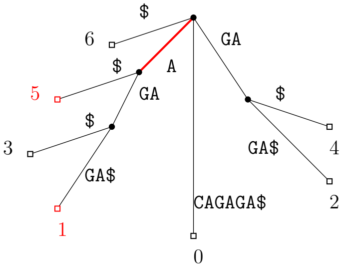
Let $T =$ CAGAGA\$. Let $(1, 5)$ be the query. The answer is A.



PREPROCESS: text T
QUERY: a pair (i, j); return the longest common prefix of $T[i\,..]$ and $T[j\,..]$

Example

Let $T =$ CAGAGA$. Let $(1, 5)$ be the query. The answer is A.



Theorem

Longest common prefix queries can be answered in $O(1)$ time after $O(n)$ time preprocessing.

Application 7: Longest palindromic substring

INPUT: text T
OUTPUT: a longest palindromic substring of T

Application 7: Longest palindromic substring

INPUT: text T

OUTPUT: a longest palindromic substring of T

Palindrome: $S = \texttt{ATTA} = S^R = \texttt{ATTA}$.

Application 7: Longest palindromic substring

INPUT: text T
OUTPUT: a longest palindromic substring of T
Palindrome: $S = \texttt{ATTA} = S^R = \texttt{ATTA}$.

- Construct the suffix tree of $T\#T^R\$$.

Application 7: Longest palindromic substring

INPUT: text T
OUTPUT: a longest palindromic substring of T
Palindrome: $S = \texttt{ATTA} = S^R = \texttt{ATTA}$.

- Construct the suffix tree of $T\#T^R\$$.
- Preprocess the suffix tree for LCA queries.

Application 7: Longest palindromic substring

INPUT: text T
OUTPUT: a longest palindromic substring of T
Palindrome: $S = \texttt{ATTA} = S^R = \texttt{ATTA}$.

- Construct the suffix tree of $T\#T^R\$$.
- Preprocess the suffix tree for LCA queries.
- Say we are interested in odd-length palindromes.

Application 7: Longest palindromic substring

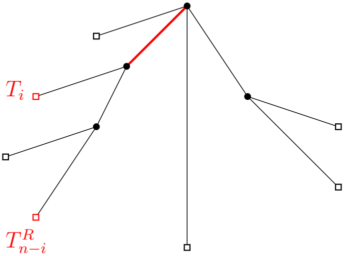
INPUT: text T
OUTPUT: a longest palindromic substring of T
Palindrome: $S = \texttt{ATTA} = S^R = \texttt{ATTA}$.

- Construct the suffix tree of $T\#T^R\$$.
- Preprocess the suffix tree for LCA queries.
- Say we are interested in odd-length palindromes.
- Answer LCA queries for T_i and T^R_{n-i}, for all i.

Application 7: Longest palindromic substring

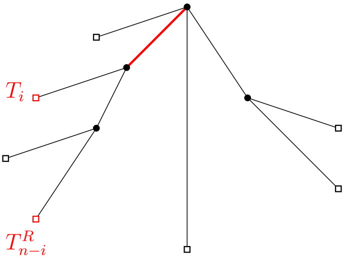
INPUT: text T
OUTPUT: a longest palindromic substring of T
Palindrome: $S =$ ATTA $= S^R =$ ATTA.

- Construct the suffix tree of $T\#T^R\$$.
- Preprocess the suffix tree for LCA queries.
- Say we are interested in odd-length palindromes.
- Answer LCA queries for T_i and T^R_{n-i}, for all i.



Application 7: Longest palindromic substring

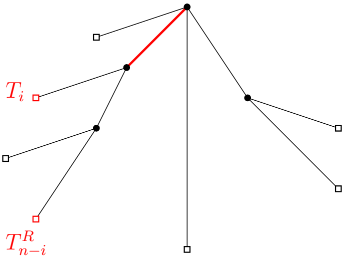
INPUT: text T
OUTPUT: a longest palindromic substring of T



Application 7: Longest palindromic substring

INPUT: text T

OUTPUT: a longest palindromic substring of T

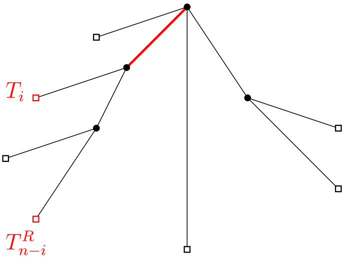


- A deepest LCA represents the longest odd-length palindrome.

Application 7: Longest palindromic substring

INPUT: text T

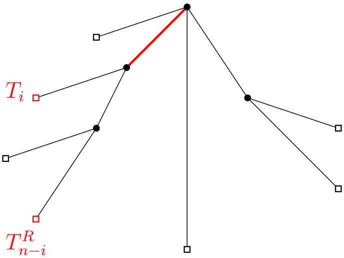
OUTPUT: a longest palindromic substring of T



- A deepest LCA represents the longest odd-length palindrome.
- Even-length palindromes are handled analogously.

Application 7: Longest palindromic substring

INPUT: text T
OUTPUT: a longest palindromic substring of T

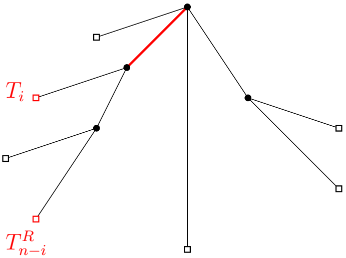


- A deepest LCA represents the longest odd-length palindrome.
- Even-length palindromes are handled analogously.
- Take the longer of the two as the answer.

Application 7: Longest palindromic substring

INPUT: text T

OUTPUT: a longest palindromic substring of T



- A deepest LCA represents the longest odd-length palindrome.
- Even-length palindromes are handled analogously.
- Take the longer of the two as the answer.

Theorem

A longest palindromic substring can be computed in $O(n)$ time.

Application 8: Approximate string matching

INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$

Application 8: Approximate string matching

INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$
Hamming distance d_H: $d_H(\texttt{GCTA}, \texttt{GCAA}) = 1$; $d_H(\texttt{GCTA}, \texttt{ACAA}) = 2$.

Application 8: Approximate string matching

INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$
Hamming distance d_H: $d_H(\mathtt{GCTA}, \mathtt{GCAA}) = 1$; $d_H(\mathtt{GCTA}, \mathtt{ACAA}) = 2$.

- Construct the suffix tree of $P\#T\$$.

Application 8: Approximate string matching

INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$
Hamming distance d_H: $d_H(\texttt{GCTA}, \texttt{GCAA}) = 1$; $d_H(\texttt{GCTA}, \texttt{ACAA}) = 2$.

- Construct the suffix tree of $P\#T\$$.
- Answer LCA query for T_i and P, for $i = 0$.

Application 8: Approximate string matching

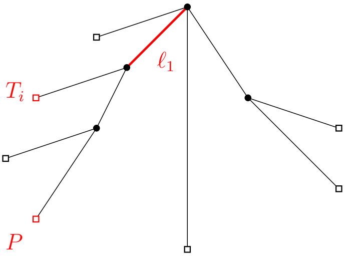
INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$
Hamming distance d_H: $d_H(\texttt{GCTA}, \texttt{GCAA}) = 1$; $d_H(\texttt{GCTA}, \texttt{ACAA}) = 2$.

- Construct the suffix tree of $P\#T\$$.
- Answer LCA query for T_i and P, for $i = 0$.
- Say this gives an LCP of length ℓ_1.

Application 8: Approximate string matching

INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$
Hamming distance d_H: $d_H(\mathtt{GCTA}, \mathtt{GCAA}) = 1$; $d_H(\mathtt{GCTA}, \mathtt{ACAA}) = 2$.

- Construct the suffix tree of $P\#T\$$.
- Answer LCA query for T_i and P, for $i = 0$.
- Say this gives an LCP of length ℓ_1.



Application 8: Approximate string matching

INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$

Application 8: Approximate string matching

INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$

- "Jump" over the mismatch $T[i + \ell_1] \neq P[\ell_1]$.

Application 8: Approximate string matching

INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$

- "Jump" over the mismatch $T[i + \ell_1] \neq P[\ell_1]$.
- Via answering the LCA query for $T_{i+\ell_1+1}$ and P_{ℓ_1+1}.

Application 8: Approximate string matching

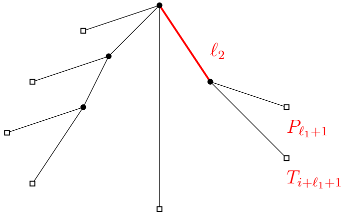
INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$

- "Jump" over the mismatch $T[i + \ell_1] \neq P[\ell_1]$.
- Via answering the LCA query for $T_{i+\ell_1+1}$ and P_{ℓ_1+1}.
- This gives an LCP of length ℓ_2.

Application 8: Approximate string matching

INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$

- "Jump" over the mismatch $T[i + \ell_1] \neq P[\ell_1]$.
- Via answering the LCA query for $T_{i+\ell_1+1}$ and P_{ℓ_1+1}.
- This gives an LCP of length ℓ_2.

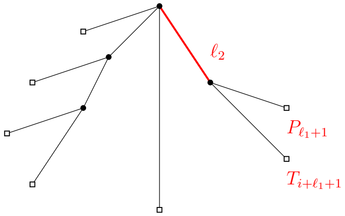


Application 8: Approximate string matching

INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$

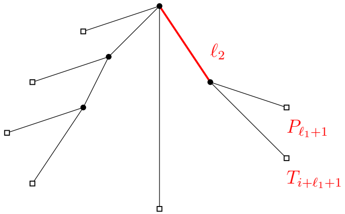
Application 8: Approximate string matching

INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$



Application 8: Approximate string matching

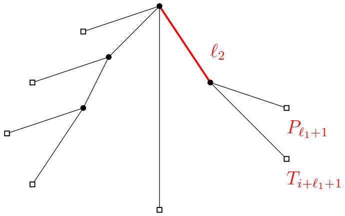
INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$



- Answer (at most) $k + 1$ queries per i.

Application 8: Approximate string matching

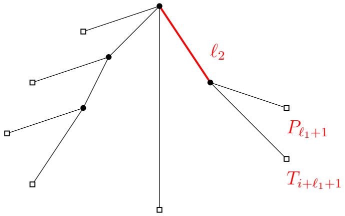
INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$



- Answer (at most) $k + 1$ queries per i.
- Report i if the total length $\ell_1 + 1 + \ell_2 + 1 + \cdots$ is at least $|P|$.

Application 8: Approximate string matching

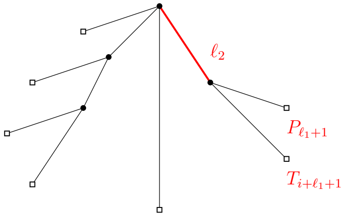
INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$



- Answer (at most) $k + 1$ queries per i.
- Report i if the total length $\ell_1 + 1 + \ell_2 + 1 + \cdots$ is at least $|P|$.
- Repeat for all $i \in [1, n]$.

Application 8: Approximate string matching

INPUT: text T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$



- Answer (at most) $k + 1$ queries per i.
- Report i if the total length $\ell_1 + 1 + \ell_2 + 1 + \cdots$ is at least $|P|$.
- Repeat for all $i \in [1, n]$.

Theorem (Landau and Vishkin, TCS 1986)

Approximate string matching can be solved in $O(kn)$ time.

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T
LZ factorization of T:

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T
LZ factorization of T:

- $T = F_0 \cdot F_1 \cdots F_k$;

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T
LZ factorization of T:

- $T = F_0 \cdot F_1 \cdots F_k$;
- Each F_i is the longest prefix of $F_i \cdots F_k$ with some occurrence to the left;

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T
LZ factorization of T:

- $T = F_0 \cdot F_1 \cdots F_k$;
- Each F_i is the longest prefix of $F_i \cdots F_k$ with some occurrence to the left;
- (or a single letter in case this prefix is empty.)

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T
LZ factorization of T:

- $T = F_0 \cdot F_1 \cdots F_k$;
- Each F_i is the longest prefix of $F_i \cdots F_k$ with some occurrence to the left;
- (or a single letter in case this prefix is empty.)

Example

Let $T = \texttt{abbaabbbaaabab}$.

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T
LZ factorization of T:

- $T = F_0 \cdot F_1 \cdots F_k$;
- Each F_i is the longest prefix of $F_i \cdots F_k$ with some occurrence to the left;
- (or a single letter in case this prefix is empty.)

Example

Let $T = \texttt{abbaabbbaaabab}$. The LZ factorization of T is
$\texttt{a} \cdot \texttt{b} \cdot \texttt{b} \cdot \texttt{a} \cdot \texttt{abb} \cdot \texttt{baa} \cdot \texttt{ab} \cdot \texttt{ab}$.

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T
LZ factorization of T:

- $T = F_0 \cdot F_1 \cdots F_k$;
- Each F_i is the longest prefix of $F_i \cdots F_k$ with some occurrence to the left;
- (or a single letter in case this prefix is empty.)

Example

Let $T = \texttt{abbaabbbaaabab}$. The LZ factorization of T is $\texttt{a} \cdot \texttt{b} \cdot \texttt{b} \cdot \texttt{a} \cdot \texttt{abb} \cdot \texttt{baa} \cdot \texttt{ab} \cdot \texttt{ab}$.

Why do we care?

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T
LZ factorization of T:

- $T = F_0 \cdot F_1 \cdots F_k$;
- Each F_i is the longest prefix of $F_i \cdots F_k$ with some occurrence to the left;
- (or a single letter in case this prefix is empty.)

Example

Let $T = \texttt{abbaabbbaaabab}$. The LZ factorization of T is $\texttt{a} \cdot \texttt{b} \cdot \texttt{b} \cdot \texttt{a} \cdot \texttt{abb} \cdot \texttt{baa} \cdot \texttt{ab} \cdot \texttt{ab}$.

Why do we care? LZ factorization is a basic and powerful technique for text compression (and string algorithms)!

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T

- Construct the suffix tree of T.

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T

- Construct the suffix tree of T.
- Decorate each internal node with the leftmost starting position the string it represents occurs.

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T

- Construct the suffix tree of T.
- Decorate each internal node with the leftmost starting position the string it represents occurs.
- How?

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T

- Construct the suffix tree of T.
- Decorate each internal node with the leftmost starting position the string it represents occurs.
- How? Use a depth-first traversal and propagate the starting positions upwards.

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T

- Construct the suffix tree of T.
- Decorate each internal node with the leftmost starting position the string it represents occurs.
- How? Use a depth-first traversal and propagate the starting positions upwards.
- Run the matching statistics algorithm for T with respect to T.

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T

- Construct the suffix tree of T.
- Decorate each internal node with the leftmost starting position the string it represents occurs.
- How? Use a depth-first traversal and propagate the starting positions upwards.
- Run the matching statistics algorithm for T with respect to T.
- For each longest match check the leftmost starting position.

Application 9: Lempel-Ziv factorization

INPUT: text T
OUTPUT: Lempel-Ziv factorization of T

- Construct the suffix tree of T.
- Decorate each internal node with the leftmost starting position the string it represents occurs.
- How? Use a depth-first traversal and propagate the starting positions upwards.
- Run the matching statistics algorithm for T with respect to T.
- For each longest match check the leftmost starting position.

Theorem

LZ factorization can be computed in $O(n)$ time.

Application 10: Shortest unique substring

INPUT: text T
OUTPUT: a shortest unique substring of T

Application 10: Shortest unique substring

INPUT: text T
OUTPUT: a shortest unique substring of T

- Construct the suffix tree of T.

Application 10: Shortest unique substring

INPUT: text T

OUTPUT: a shortest unique substring of T

- Construct the suffix tree of T.
- For each leaf node labeled i, for all $i \in [0, n]$, pick up the closest ancestor v using a depth-first traversal.

Application 10: Shortest unique substring

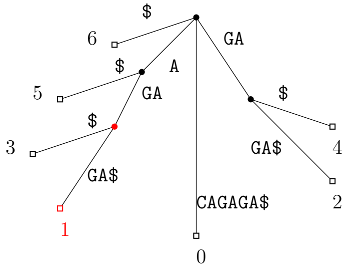
INPUT: text T

OUTPUT: a shortest unique substring of T

- Construct the suffix tree of T.
- For each leaf node labeled i, for all $i \in [0, n]$, pick up the closest ancestor v using a depth-first traversal.

Example

Let $T =$ CAGAGA$.



Application 10: Shortest unique substring

INPUT: text T
OUTPUT: a shortest unique substring of T

Application 10: Shortest unique substring

INPUT: text T
OUTPUT: a shortest unique substring of T

- The substring represented by v concatenated with the succeeding letter is the shortest unique substring starting at i.

Application 10: Shortest unique substring

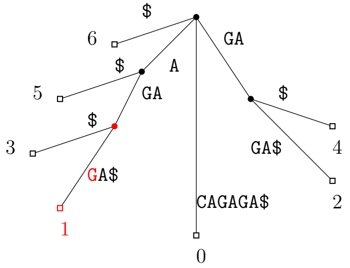
INPUT: text T

OUTPUT: a shortest unique substring of T

- The substring represented by v concatenated with the succeeding letter is the shortest unique substring starting at i.

Example

Let $T =$ CAGAGA\$. The shortest unique substring starting at 1 is AGAG.



Application 10: Shortest unique substring

INPUT: text T
OUTPUT: a shortest unique substring of T

Application 10: Shortest unique substring

INPUT: text T
OUTPUT: a shortest unique substring of T

- Take a shortest substring among all i.

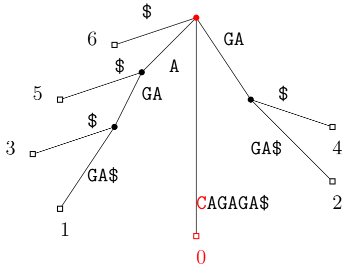
Application 10: Shortest unique substring

INPUT: text T
OUTPUT: a shortest unique substring of T

- Take a shortest substring among all i.

Example

Let $T =$ CAGAGA\$. The shortest unique substring is C.



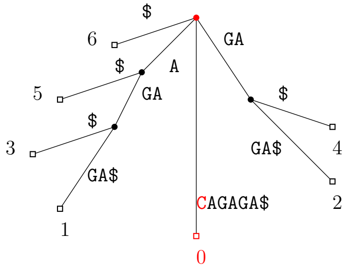
Application 10: Shortest unique substring

INPUT: text T
OUTPUT: a shortest unique substring of T

- Take a shortest substring among all i.

Example

Let $T =$ CAGAGA\$. The shortest unique substring is C.



Theorem

A shortest unique substring can be computed in $O(n)$ time.

Take-home message

Take-home message

- Suffix tree is a fundamental data structure for processing any type of sequential data.

Take-home message

- Suffix tree is a fundamental data structure for processing any type of sequential data.
- It provides fast implementations of many important string operations.

Take-home message

- Suffix tree is a fundamental data structure for processing any type of sequential data.
- It provides fast implementations of many important string operations.
- Practice?

Take-home message

- Suffix tree is a fundamental data structure for processing any type of sequential data.
- It provides fast implementations of many important string operations.
- Practice? Suffix arrays enhanced with some extra information.

Take-home message

- Suffix tree is a fundamental data structure for processing any type of sequential data.
- It provides fast implementations of many important string operations.
- Practice? Suffix arrays enhanced with some extra information.

Thanks!