Applied Databases

Lecture 15
Suffix Trees and Suffix Arrays

Sebastian Maneth

University of Edinburgh - March 13th, 2017
Horspool

Match **RIGHT-TO-LEFT**

\[ T = \begin{array}{cccccccccccccc}
  a & b & a & b & a & a & b & b & a & b & c & a & b & a & b & a & b & a & a & b & c \\
  a & b & a & b & c \\
\end{array} \]

\[ R(a) = 2 \]

\[ R(c) = 5 \]

\[ R(b) = 1 \]

**Horspool**

If mismatch and \( P[m] \) aligned to \( z \) in \( T \), shift pattern to the RIGHT by \( R(z) \).

**Question** → can you do Horspool on **Unicode** (e.g. **UTF-8**)??

variable length encoding
## UTF-8

Maps a unicode character into **1, 2, 3, or 4 bytes**.

<table>
<thead>
<tr>
<th>Unicode range</th>
<th>Byte sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+000000 → U+00007F</td>
<td>0BBBBB....</td>
</tr>
<tr>
<td>U+000080 → U+0007FF</td>
<td>110BBBBB 10BBBBBB</td>
</tr>
<tr>
<td>U+000800 → U+00FFFF</td>
<td>1110BBBB 10BBBBBB 10BBBBBB</td>
</tr>
<tr>
<td>U+010000 → U+10FFFF</td>
<td>11110BBB 10BBBBBB 10BBBBBB 10BBBBBB</td>
</tr>
</tbody>
</table>

Spare bits (□) are filled from right to left. Pad to the left with 0-bits.

E.g. U+00A9 in **UTF-8** is 11000010 10101001
U+2260 in **UTF-8** is 11100010 10001001 10100000

**Question** → can you do Horspool on **Unicode** (e.g. UTF-8)??

→ try to solve it yourself
→ possibly consult Patent US8819045

**variable length encoding**
Outline

1. Suffix Tree
2. Suffix Tree Construction
3. Applications of Suffix Trees
4. Suffix Array
1. Suffix Tree

T = abaababa

Suffixes
1 abaababa
2 baababa
3 aababa
4 ababa
5 baba
6 aba
7 ba
8 a

New Idea
→ collapse paths of white nodes!
1. Suffix Tree

Suffixes
1. abaababa
2. baababa
3. aababa
4. ababa
5. baba
6. aba
7. ba
8. a

New Idea
→ collapse paths of white nodes!
1. Suffix Tree

$T = \text{abaababa}$
1. Suffix Tree

$T = \text{abaababa}$
Suffix Tree

\[12345678\]
\( T = abaababa \)

Suffix Tree of \( T \)
**Suffix Tree**

$T = \text{abaababa}$

- How many nodes (at most) in the suffix tree of $T$?
Suffix Tree

$T = \text{abaababa}\$

$\rightarrow$ add end marker “$”

$\rightarrow$ one-to-one correspondence of leaves to suffixes

$\rightarrow$ a tree with $n+1$ leaves (and no nodes with only one child) has $\leq 2n+1$ nodes!

Lemma
Size of suffix tree for “$T$” is linear in $n=|T|$, i.e., in $O(n)$. 
Suffix Tree

$T = abaababa$

→ add end marker “$”

→ one-to-one correspondence of leaves to suffixes

→ a tree with $n+1$ leaves (and no nodes with only one child) has $\le 2n+1$ nodes!

Lemma
Size of suffix tree for “$T$” is linear in $n=|T|$, i.e., in $O(n)$.

→ search time still $O(|P|)$, as for suffix trie!

→ perfect data structure for our task!
2. Suffix Tree Construction

Good news: 
**Suffix tree can be constructed in linear time!**

But, rather complex construction algorithms

→ **Weiner 1973**  [Knuth: “Algorithm of the year 1973”]
2. Suffix Tree Construction

Good news: 
Suffix tree can be constructed in linear time!

Complex construction algorithms


→ McCreight 1976 Simplification of Weiner’s algorithm
2. Suffix Tree Construction

Good news: Suffix tree can be constructed in linear time!

Complex construction algorithms

→ McCreight 1976 Simplification of Weiner’s algorithm
→ Ukkonen 1995 first online algorithm!
  → T may come from a stream
  → build suffix tree for TT’ from suffix tree for T
  → huge breakthrough!!
2. Suffix Tree Construction

Good news:
Suffix tree *can be constructed in linear time!*

Complex construction algorithms

→ Weiner 1973
→ McCreight 1976 Linear time only for *constant-size alphabets*!
   Otherwise, $O(n \log n)$
→ Ukkonen 1995
2. Suffix Tree Construction

Good news: *Suffix tree can be constructed in linear time!*

Complex construction algorithms

→ Weiner 1973
→ McCreight 1976  Linear time only for *constant-size alphabets*! Otherwise, $O(n \log n)$
→ Ukkonen 1995
→ Farach 1997  Linear time for *any integer alphabet*, drawn from a polynomial range

→ again a big breakthrough
2. Suffix Tree Construction

Good news: Suffix tree can be constructed in linear time!

Complex construction algorithms

→ Weiner 1973
→ McCreight 1976
→ Ukkonen 1995
→ Farach 1997
→ Kurtz 1999

Practical algorithm
13–15n Bytes space requirement.

(→ e.g. McCreight: 28n Bytes)
2. Suffix Tree Construction

Good news: 
**Suffix tree can be constructed in linear time!**

Complex construction algorithms

→ Weiner 1973

→ McCreight 1976

→ Ukkonen 1995

→ Farach 1997
Suffix Link

**Definition**

If $x = ay$ is the string corresponding to a node $u$ in the ST then the *suffix link* $\text{suf}[u]$ is the node $v$ corresponding to $y$ in ST.
Suffix Link

Definition

If \( x = ay \) is the string corresponding to a node \( u \) in the ST then the suffix link \( \text{suf}[u] \) is the node \( v \) corresponding to \( y \) in ST.

Where is the suffix link of node “2”? 

\[ T = \text{abaababa} \]

\[ 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \]
Suffix Link

Definition

If $x = ay$ is the string corresponding to a node $u$ in the ST then the suffix link $\text{suf}[u]$ is the node $v$ corresponding to $y$ in ST.

Where is the suffix link of node “2”?

- essential node
- non-essential node
Suffix Link

Definition

If \( x = ay \) is the string corresponding to a node \( u \) in the ST then the **suffix link** \( \text{suf}[u] \) is the node \( v \) corresponding to \( y \) in ST.

Using suffix links, we can *on-line* build the Suffix-TRIE of \( T \) in time \( O(|\text{Suffix-TRIE}(T)|) \).

- essential node
- non-essential node
Using suffix links, we can *on-line* build the Suffix-TRIE of $T$ in time $O(|\text{Suffix-TRIE}(T)|)$.

$T = \text{abaabb}$

*Online construction*

$v = \text{lowest leaf in tree}$
$b = T[\text{current}]$

From $v$, follow ($k$ times) suffix links (to $u$) until $\text{child}(u, b)$ is defined.

Create $b$-sons for $v$, $\text{suffix}[v]$, $\text{suffix}^2[v]$, ..., $\text{suffix}^{k-1}[v]$

If there is no such $u$, create $b$-sons for all of them, up to $k$
Using suffix links, we can *on-line* build the Suffix-TRIE of $T$ in time $O(\mid \text{Suffix-TRIE}(T)\mid)$.

$T = \text{abaabb}$

**Online construction**

$v = \text{lowest leaf in tree}$
$b = T[\text{current}]$

From $v$, follow ($k$ times) suffix links (to $u$) until $\text{child}(u, b)$ is defined.

Create $b$-sons for $v$, $\text{suf}[v]$, $\text{suf}^2[v]$, ..., $\text{suf}^{k-1}[v]$

If there is no such $u$, create $b$-sons for all of them, up to $k$

New suffix links:
Using suffix links, we can **on-line** build the Suffix-TRIE of $T$ in time $O(|\text{Suffix-TRIE}(T)|)$.

$T = \text{abaabb}$

**Online construction**

$v = \text{lowest leaf in tree}$
$b = T[\text{current}]$

From $v$, follow ($k$ times) suffix links (to $u$) until child($u$, $b$) is defined.
Create $b$-sons for $v$, $\text{suf}[v]$, $\text{suf}^2[v]$, ..., $\text{suf}^{k-1}[v]$
If there is no such $u$, create $b$-sons for all of them, up to $k$

New suffix links:
Using suffix links, we can on-line build the Suffix-TRIE of T in time $O(|\text{Suffix-TRIE}(T)|)$.

T = abaabb

Online construction

v = lowest leaf in tree
b = T[current]
From v, follow (k times) suffix links (to u) until child(u, b) is defined.
Create b-sons for v, su[v], su^2[v], ..., su^{k-1}[v]
If there is no such u, create b-sons for all of them, up to k

New suffix links:
Using suffix links, we can **on-line** build the Suffix-TRIE of $T$ in time $O(|\text{Suffix-TRIE}(T)|)$.

$T = \text{abaabb}$

**Online construction**

$v = \text{lowest leaf in tree}$

$b = T[\text{current}]$

From $v$, follow ($k$ times) suffix links (to $u$) until $\text{child}(u, b)$ is defined.

Create $b$-sons for $v$, $\text{suffix}[v]$, $\text{suffix}^2[v]$, ..., $\text{suffix}^{k-1}[v]$

If there is no such $u$, create $b$-sons for all of them, up to $k$.

New suffix links:
Using suffix links, we can on-line build the Suffix-TRIE of $T$ in time $O(|\text{Suffix-TRIE}(T)|)$.

$T = \text{abaabb}$

**Online construction**

$v = \text{lowest leaf in tree}$
$b = T[\text{current}]$

From $v$, follow ($k$ times) suffix links (to $u$) until $\text{child}(u, b)$ is defined.
Create $b$-sons for $v$, $\text{suf}[v]$, $\text{suf}^2[v]$, ..., $\text{suf}^{k-1}[v]$.
If there is no such $u$, create $b$-sons for all of them, up to $k$.

New suffix links:
Using suffix links, we can on-line build the Suffix-TRIE of $T$ in time $O(|\text{Suffix-TRIE}(T)|)$.

$T = \text{abaabb}$

Online construction

$v =$ lowest leaf in tree

$b =$ $T[\text{current}]$

From $v$, follow (k times) suffix links (to $u$) until child($u$, $b$) is defined.
Create $b$-sons for $v$, $\text{suf}[v]$, $\text{suf}^2[v]$, ..., $\text{suf}^{k-1}[v]$
If there is no such $u$, create $b$-sons for all of them, up to $k$

New suffix links:
Using suffix links, we can *on-line* build the Suffix-Trie of $T$ in time $O(|\text{Suffix-Trie}(T)|)$.

$T = \text{abaabb}$

**Online construction**

$v = \text{lowest leaf in tree}$
$b = T[\text{current}]$

From $v$, follow ($k$ times) suffix links (to $u$) until $\text{child}(u, b)$ is defined.
Create $b$-sons for $v$, $\text{suffix}[v]$, $\text{suffix}^2[v]$, ..., $\text{suffix}^{k-1}[v]$
If there is no such $u$, create $b$-sons for all of them, up to $k$

New suffix links:
Using suffix links, we can **on-line** build the Suffix-Trie of T in time $O(|\text{Suffix-Trie}(T)|)$.

$T = \text{abaabb}$

**Online construction**

$v = \text{lowest leaf in tree}$

$b = T[\text{current}]$

From $v$, follow (k times) suffix links (to $u$) until child($u$, $b$) is defined.

Create $b$-sons for $v$, $\text{suf}[v]$, $\text{suf}^2[v]$, ..., $\text{suf}^{k-1}[v]$.

If there is no such $u$, create $b$-sons for all of them, up to $k$.

New suffix links:
Using suffix links, we can \textit{on-line} build the Suffix-TRIE of $T$ in time $O(|\text{Suffix-TRIE}(T)|)$.

$T = \text{abaabb}$

\textbf{Online construction}

Ukkonen's on-line construction of suffix trees works in a similar way.

It maintains \textcolor{red}{collapsed edges} at all times.
$T = \text{mississippi}$
3. Applications of Suffix Trees

Generalized Suffix tree for a SET $S$ of strings:

$S = \{ S_1, S_2, S_3, \ldots, S_k \}$

$T = S_1 \#_1 S_2 \#_2 S_3 \#_3 \ldots S_k \#_k$

Where $\#_1, \#_2, \ldots, \#_k$ are fresh new symbols.
(b) Longest Common Substring of two Strings

\[ S_1 = \text{superiorcalifornialives} \]
\[ S_2 = \text{sealiver} \]

\[ \text{LCS}(S_1, S_2) = \text{alive} \]

\[ \rightarrow \text{Build generalized suffix tree of } \{ S_1, S_2 \} \]
\[ \rightarrow \text{Mark internal nodes with "1" or "2" if subtree contains } (1,\_ \text{) pair or } (2,\_ \text{) pair.} \]

\[ \text{LCS}(S_1, S_2) = \]
\[ \text{maximal string depth of any node marked "1,2"} \]

\[ \rightarrow \text{Can be determined by a simple tree traversal} \]
(b) Longest Common Substring of two Strings

\[ S_1 = \text{fornialives} \]
\[ S_2 = \text{sealiver} \]

\[ \text{LCS}(S_1, S_2) = \text{alive} \]
(b) Longest Common Substring of two Strings

Theorem
The *longest common substring* of two strings can be found in *linear time*, using a generalized suffix tree.

[Karp, Miller, Rosenberg 1972] solved the problem in \( O((m+n)\log(m+n)) \) time where \( m = |S_1| \) and \( n = |S_2| \).

In 1970 Donald Knuth conjectured that it is *impossible* to solve the problem in linear time!

→ Linear time solution by [Weiner, 1973]

First linear time suffix tree construction algorithm
(c) Matching Statistics

\[ ms(k) = \text{length L of longest substring } T[k...k+L] \text{ that matches a substring in } P. \]

\[ p(k) = \text{start position in } P \text{ of a substring of length } ms(k) \text{ matching } T[k...k+ms(k)] \]

\[ T = abc\text{x}abcd\text{ex} \ldots \]

\[ P = yabcwzq\text{abcdw} \]

**Computation of ms and p**

**Build suffix tree of** \( P \) *(including suffix links).*

At node \( v \) corresponding to \( ms(i) \),
compute \( ms(i+1) \) as follows:

1. If \( v \) is internal, follow its suffix link.
2. If \( v \) is leaf, walk to parent (label \( \gamma \))

Current node is prefix of \( T[i+1...n] \).
Proceed downwards to longest match
(as in ordinary search)

→ Allows to find \( LCS(S_1,S_2) \) using only

*one* suffix tree (of the shorter string).
(d) Compression

→ E.g., infinite-window Lempel-Ziv like compression

a b a abaa aba baba ab b → a b a (1,4) (1,3) (9,4) (1,2) b

M. C. Escher (1948)
(d) Compression

LZ-variant with infinite window

abaabaaabababaabb

a b a abaa aba baba ab b

longest string that has appeared before coded as: (position, length)

a b a (1,4) (1,3) (9,4) (1,2) b

→ Build suffix tree of text T
→ Annotate internal nodes by smallest position number in their subtree

→ To find pair (x, y) at a position p in T, match T[x...] against suffix tree as long as minimal pos number is smaller than x.
3. Applications of Suffix Trees

Suffix trees have *many* more applications
e.g. in computational biology see [Gusfield book].

→ Substring problem for a database of patterns
→ DNA contamination problem
→ Find complemented palindroms in DNA (e.g. AGCTCGCGAGCT)
→ Find all maximal repeats / maximal pairs
→ ...
7 First Applications of Suffix Trees

7.1 APL1: Exact string matching
7.2 APL2: Suffix trees and the exact set matching problem
7.3 APL3: The substring problem for a database of patterns
7.4 APL4: Longest common substring of two strings
7.5 APL5: Recognizing DNA contamination
7.6 APL6: Common substrings of more than two strings
7.7 APL7: Building a smaller directed graph for exact matching
7.8 APL8: A reverse role for suffix trees, and major space reduction
7.9 APL9: Space-efficient longest common substring algorithm
7.10 APL10: All-pairs suffix-prefix matching
7.11 Introduction to repetitive structures in molecular strings
7.12 APL11: Finding all maximal repetitive structures in linear time
7.13 APL12: Circular string linearization
7.14 APL13: Suffix arrays – more space reduction
7.15 APL14: Suffix trees in genome-scale projects
7.16 APL15: A Boyer–Moore approach to exact set matching
7.17 APL16: Ziv–Lempel data compression
7.18 APL17: Minimum length encoding of DNA
7.19 Additional applications
7.20 Exercises
Space Consumption of Suffix Trees
Questions

→ is the size of this tree really in $O(n)$?
→ in terms of #nodes/edges: OK

→ how about the sizes of labels ??
Questions

→ is the size of this tree really in $O(n)$?
→ in terms of #nodes/edges: OK

→ how about the sizes of labels ??
Yes, but:

→ $\log(n)$ is small, e.g., 64 bits

→ can be considered constant!

each requires $\log(n)$ bits!?
Lesson to learn:

→ log(n) in terms of a **run-time factor**, can be *fatal*

→ in terms of a **space-factor**, it is *fine*!

how long will a text be?

2^64???

4 / 8 Bytes each is enough!
For any text with fewer characters than the number of atoms in the universe, the label size for the suffix tree is a constant of \( x \) bits. Each is enough!
label size is not an issue

but, size of edge-pointers?

imagine each edge requires a 32-bit pointer!!
Actual Space of Suffix Trees

Space for edge-pointers is problematic:

→ actual space of suffix tree, ca. $20|T|$

→ on commodity hardware, texts of more than 1GB are not doable
→ how to avoid the huge space needed for edges?
4. Suffix Array

Definition
Given text $T$ of length $n$. For $i=1\ldots n$, $SA[k]=i$ if suffix $T[i\ldots n]$ is at position $k$ in the lexicographic order $T$’s suffixes.

$T = \text{mississippi}\$  
$\text{Order } \$ < i < m < p < s$

$SA(T) = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]$
Suffix Array Construction
4. Suffix Array

→ read leaves from left-to-right!

\[ SA(T) = [12, 11, 8, 5, 2, 10, 9, 7, 4, 6, 3] \]
4. Suffix Array

→ read leaves from left-to-right!

\[ \text{SA}(T) = [12, 11, 8, 5, 2, 10, 9, 7, 4, 6, 3] \]

**Theorem**
The suffix array of \( T \) can be constructed in time \( O(|T|) \).
Search

Theorem
Using binary search on $SA(T)$, all occurrences of $P$ in $T$ can be located in $O(|P| \times \log|T|)$ time.

$T = \text{mississippi}$

$SA(T) = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]$

Search for $P = \text{iissippi}$ all occurrence's consecutive in $SA$!

Binary search for start-index:
$L = 1$, $R = |T| = n$
Repeat
$M = \lceil (L+R-1)/2 \rceil$
If $P \leq_{\text{lex}} T[M \ldots M+|P|]$ then $R := M$ else $L := M$
Until $M$ does not change.
Search

1234567890
$T = \text{mississippi}$

$SA(T) = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]$

Binary search for start-index:
L=1, R=|T|=n
Repeat
    M = [(L+R-1)/2]
    If $P \leq_{\text{lex}} T[M...M+|P|]$ then R:=M else L:=M
Until M does not change.
Search

1234567890
T = mississippi$

\text{SA}(T) = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]

\begin{align*}
\text{L} & \rightarrow \text{R} \\
\text{issi} & \leq \text{miss?} \\
\text{yes} & \\
\text{issi} & \leq \text{ippi?}
\end{align*}

Binary search for start-index:
L=1, R=|T|=n
Repeat
\begin{align*}
M &= \lfloor (\text{L}+\text{R}-1)/2 \rfloor \\
\text{If } P \leq_{\text{lex}} T[M...M+|P|] & \text{ then } \text{R:=M else } \text{L:=M}
\end{align*}
Until M does not change.

12 $ \\
11 i$ \\
8 ippi$ \\
5 issippi$ \\
2 ississippi$ \\
1 mississippi$ \\
10 pi$ \\
9 ppi$ \\
7 sippi$ \\
4 sississippi$ \\
6 ssippi$ \\
3 ssissippi$
Search

\[ T = \text{mississippi}\$

\[ \text{SA}(T) = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3] \]

\[ \text{issi} \leq \text{issi} \]

Binary search for start-index:
\( L = 1, R = |T| = n \)
Repeat
\[ M = \lceil (L+R-1)/2 \rceil \]
If \( P \leq_{\text{lex}} T[M...M+|P|] \) then \( R := M \) else \( L := M \)
Until \( M \) does not change.
Search

1234567890
T = mississippi$

SA(T) = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]

issi ≤ issi

Binary search for start-index:
L = 1, R = |T| = n
Repeat
  M = ⌊(L+R-1)/2⌋
  If P ≤_{lex} T[M...M+|P|] then R := M else L := M
Until M does not change.
Search

1234567890
\( T = \text{mississippi}\$

\( SA(T) = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3] \)

issi \( \leq \) issi

M does not change.
\( \Rightarrow \) Return R

Binary search for start-index:
L=1, R=\(|T|\)=n
Repeat
\[ M = \lceil (L+R-1)/2 \rceil \]
If \( P \leq_{\text{lex}} T[M...M+|P|] \) then \( R:=M \) else \( L:=M \)
Until M does not change.
Search

1234567890
T = mississippi$

SA(T) = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]

Binary search for end-index:

L=1, R=|T|=n

Repeat

\[ M = \lfloor (L+R-1)/2 \rfloor \]

If \( P <_{\text{lex}} T[M...M+|P|] \) then \( R := M \) else \( L := M \)

Until \( M \) does not change.
Search

\[ T = \text{mississippi}\$
\]
\[ \text{SA}(T) = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3] \]

Binary search for end-index:
L=1, R=|T|=n
Repeat
\[ M = \lfloor (L+R-1)/2 \rfloor \]
If \( P \text{ lex} T[M...M+|P|] \) then R:=M else L:=M
Until M does not change.
Search

T = mississippi$

SA(T) = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]

not( issi < issi )
→ L := M

Binary search for end-index:
L = 1, R = |T| = n
Repeat
  M = [(L + R - 1)/2]
  If P $\leq_{\text{lex}}$ T[M...M+|P|] then R := M else L := M
Until M does not change.
Search

Start-index

\[ T = \text{mississippi}\]

\[ SA(T) = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3] \]

M does not change.
\[ \Rightarrow \text{Return } L \]

Binary search for end-index:
\[ L=1, \ R=|T|=n \]
Repeat
\[ M = \lfloor (L+R-1)/2 \rfloor \]
If \[ P <_{\text{lex}} T[M...M+|P|] \] then \[ R := M \] else \[ L := M \]
Until M does not change.
Binary search for end-index:
L = 1, R = |T| = n
Repeat
   M = \left\lceil \frac{(L+R-1)}{2} \right\rceil
   If P <_{\text{lex}} T[M...M+|P|] then R := M else L := M
Until M does not change.
Search

**Theorem**
Using binary search on $SA(T)$, all occurrences of $P$ in $T$ can be located in $O(|P| \cdot \log|T|)$ time.

**Note**
This is a pessimistic bound!
*We almost never* need $O(|P|)$ time for one lexicographic comparison!

On random strings, this should run in $O(|P| + \log|T|)$ time.
Search

Theorem
Using binary search on $SA(T)$, all occurrences of $P$ in $T$ can be located in $O(|P| * \log|T|)$ time.

Note
This is a pessimistic bound! We almost never need $O(|P|)$ time for one lexicographic comparison!

On random strings, this should run in $O(|P| + \log|T|)$ time.

→ $O(|P| + \log|T|)$ in practise, using a simple trick

→ $O(|P| + \log|T|)$ guaranteed, using LCP-array

$LCP(k,j) = \text{longest common prefix of } T[SA[k]...] \text{ and } T[SA[j]...]$
Suffix Arrays

→ much more space efficient than Suffix Tree
→ used in practise (suffix tree more used in theory)

→ Suffix Array Construction, without Suffix Trees?

[ Linear Work Suffix Array Construction,
  J. Kärkkäinen, Sanders, Burkhardt,
  *Journal of the ACM*, 2006 ]

→ See also (linked from course web page)

[ A taxonomy of suffix array construction algorithms,
  S. J. Puglisi, W. F. Smyth, A. Turpin,
  *ACM Computing Surveys* 39, 2007 ]
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst Case</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prefix-Doubling</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MM [Manber and Myers 1993]</td>
<td>$O(n \log n)$</td>
<td>30</td>
<td>$8n$</td>
</tr>
<tr>
<td>LS [Larsson and Sadakane 1999]</td>
<td>$O(n \log n)$</td>
<td>3</td>
<td>$8n$</td>
</tr>
<tr>
<td><strong>Recursive</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KA [Ko and Aluru 2003]</td>
<td>$O(n)$</td>
<td>2.5</td>
<td>7–10$n$</td>
</tr>
<tr>
<td>KS [Kärkkäinen and Sanders 2003]</td>
<td>$O(n)$</td>
<td>4.7</td>
<td>10–13$n$</td>
</tr>
<tr>
<td>KSPP [Kim et al. 2003]</td>
<td>$O(n)$</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>HSS [Hon et al. 2003]</td>
<td>$O(n)$</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>KJP [Kim et al. 2004]</td>
<td>$O(n \log \log n)$</td>
<td>3.5</td>
<td>13–16$n$</td>
</tr>
<tr>
<td>N [Na 2005]</td>
<td>$O(n)$</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>Induced Copying</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT [Itoh and Tanaka 1999]</td>
<td>$O(n^2 \log n)$</td>
<td>6.5</td>
<td>$5n$</td>
</tr>
<tr>
<td>S [Seward 2000]</td>
<td>$O(n^2 \log n)$</td>
<td>3.5</td>
<td>$5n$</td>
</tr>
<tr>
<td>BK [Burkhardt and Kärkkäinen 2003]</td>
<td>$O(n \log n)$</td>
<td>3.5</td>
<td>5–6$n$</td>
</tr>
<tr>
<td>MF [Manzini and Ferragina 2004]</td>
<td>$O(n^2 \log n)$</td>
<td>1.7</td>
<td>$5n$</td>
</tr>
<tr>
<td>SS [Schürmann and Stoye 2005]</td>
<td>$O(n^2)$</td>
<td>1.8</td>
<td>9–10$n$</td>
</tr>
<tr>
<td>BB [Baron and Bresler 2005]</td>
<td>$O(n \sqrt{\log n})$</td>
<td>2.1</td>
<td>18$n$</td>
</tr>
<tr>
<td>M [Maniscalco and Puglisi 2007]</td>
<td>$O(n^2 \log n)$</td>
<td>1.3</td>
<td>5–6$n$</td>
</tr>
<tr>
<td>MP [Maniscalco and Puglisi 2006]</td>
<td>$O(n^2 \log n)$</td>
<td>1</td>
<td>5–6$n$</td>
</tr>
<tr>
<td><strong>Hybrid</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT+KA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BK+IT+KA</td>
<td>$O(n^2 \log n)$</td>
<td>4.8</td>
<td>$5n$</td>
</tr>
<tr>
<td>BK+S</td>
<td>$O(n \log n)$</td>
<td>2.3</td>
<td>5–6$n$</td>
</tr>
<tr>
<td><strong>Suffix Tree</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K [Kurtz 1999]</td>
<td>$O(n \log \sigma)$</td>
<td>6.3</td>
<td>13–15$n$</td>
</tr>
</tbody>
</table>

Time is relative to MP, the fastest in our experiments. Memory is given in bytes including space required for the suffix array and input string and is the average space required in our experiments. Algorithms HSS and N are included, even though to our knowledge they have not been implemented. The time for algorithm MM is estimated from experiments in Larsson and Sadakane [1999].
END
Lecture 15