Suffix arrays

String Algorithms
The suffix array

**Remember:** suffix \( j \) of \( x = x[1..n] \) is \( x[j..n] \)

**Suffix array** \( SA \) of \( x \) is an array of \( n \) integers such that:

\[
SA[i] = j \text{ implies that suffix } j \text{ of } x \text{ has rank } i \text{ in the lexicographical ordering of all suffixes of } x
\]

... assumes an ordered alphabet ...

... consumes \( \text{lint} \cdot n \) bytes, usually \( 4n \) bytes ...
An example

Mississippi

1: mississippi
2: ississippi
3: ssissippi
4: siissippi
5: issippi
6: ssippi
7: sippi
8: ippi
9: ppi
10: pi
11: i
An example

Mississippi

1: mississippi
2: ississippi
3: ssissippi
4: sissippi
5: issippi
6: ssippi
7: sippi
8: ippi
9: ppi
10: pi
11: i
An example

Mississippi

1: mississippi
2: ississippi
3: ssissippi
4: sississippi
5: issippi
6: ssippi
7: sippi
8: ippi
9: ppi
10: pi
11: i
An example

Mississippi

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

SA = [11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]
An example

\[ S = \text{Mississippi}$ \]

1 2 3 4 5 6 7 8 9 10 11 12

\[ SA = \{11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3\} \]
An example

The leaves in a sub-tree correspond to an interval in $SA$, e.g. $T(v) = SA[8..11]$

$S = Mississippis$

$SA = [11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]$
Constructing suffix arrays

Straightforward
Sort suffixes using e.g. radix sort, takes time $O(n^2)$

Mississippi

1: mississippi
2: ississippi
3: ssissippi
4: sissippi
5: issippi
6: ssippi
7: sippi
8: ippi
9: ppi
10: pi
11: i

Monday, May 6, 13
Constructing suffix arrays

Using the suffix tree
Make a depth-first traversal of $T$, where edges are chosen in lexicographical order (and the sentinel $\$ is the smallest letter)
Constructing suffix arrays

Using the suffix tree
Make a depth-first traversal of $T$, where edges are chosen in lexicographical order (and the sentinel $\$\$ is the smallest letter)

$SA = [11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]$
Constructing suffix arrays

Using the suffix tree
Make a depth-first traversal of $T$, where edges are chosen in lexicographical order (and the sentinel $\$\$ is the smallest letter)

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

Time: $O(n \cdot \alpha)$ ...
but suffix tree must be available
Using suffix arrays

Exact matching
Given string $x$ and pattern $u[1..m]$, report where $u$ occurs in $x$

If $u$ occurs in $x$ at position $i$, then $u$ is a prefix of suffix $i$ of $x$

In the suffix array we have access to the suffixes in sorted order, use binary search to find a suffix which has $u$ as a prefix ...
An example

Searching for ssi in mississippi

L 11: i
  8: ippi
  5: issippi
  2: ississppi
  1: mississippi

M 10: pi
  9: ppi
  7: sippi
  4: sissippi
  6: ssippi

R  3: ssississippi
An example

Searching for **ssi** in **mississippi**

11: i
8:ippi
5:issippi
2:ississippi
1:mississippi

L 10: pi
9:ppi
7:sippi

M 4:sissippi
6:ssippi

R 3:ssissippi

ssi > sis
An example

Searching for ssi in mississippi

11: i
  8:ippi
  5:issippi
  2:ississippi
  1:mississippi
10: pi
  9: ppi
  7: sippi
L  4: sissippi
M  6: ssippi
R  3: ssissippi

ssi = ssi
An example

Searching for \textit{ssi} in \textit{mississippi}

11: \textit{i}
8: \textit{ippi}
5: \textit{ississippi}
2: \textit{ississippi}
1: \textit{mississippi}
10: \textit{pi}
9: \textit{ppi}
7: \textit{sippi}

\textcolor{blue}{L} 4: \textit{sississippi}
\textcolor{red}{M} 6: \textit{ssissippi}
\textcolor{red}{R} 3: \textit{ssissippi}

\textcolor{blue}{ssi} = \textcolor{blue}{ssi}

\textbf{Note}: that all $k$ occurrences of $ssi$ are indexed by neighboring positions in $SA$ ...
An example

Searching for **ssi** in **mississippi**

11: i
  8: ippi
  5: issippi
  2: ississippi
  1: mississippi
10: pi
  9: ppi
  7: sippi

Time: $O(m \log n)$
      $O(m(\log n + k))$

**ssi = ssi**

Note: that all $k$ occurrences of **ssi** are indexed by neighboring positions in SA ...
The naive algorithm

\[ j = 0; \quad L = 1; \quad R = n \]
repeat
\[ M = \lceil (R+L)/2 \rceil \]
if \( u = x[SA[M] .. SA[M]+m-1] \) then
\[ j = SA[M] \]
elseif \( u > x[SA[M] .. SA[M]+m-1] \) then
\[ L = M \]
else
\[ R = M \]
until \( L = M \) or \( j \neq 0 \)

Time: \( O(m(\log n + k)) \)
The naive algorithm

\[ j = 0; \quad L = 1; \quad R = n \]

\textbf{repeat}

\[ M = \lceil (R+L)/2 \rceil \]

\textbf{if} \( u = x[SA[M] \ldots SA[M]+m-1] \) \textbf{then}

\[ j = SA[M] \]

\textbf{elseif} \( u > x[SA[M] \ldots SA[M]+m-1] \) \textbf{then}

\[ L = M \]

\textbf{else}

\[ R = M \]

\textbf{until} \( L = M \) \textbf{or} \( j \neq 0 \)

\textbf{Time:} \( O(m(\log n + k)) \)

Can we do better? In the worst case? In practice?
Practical speed-up

Observation

If suffixes $SA[L]$ and $SA[R]$ share a prefix, then suffixes $SA[K]$, for $K=L, L+1, \ldots, R$, share the same prefix ...

| 7: sippi |
| 4: sissippi |
| 6: ssippi |
| 3: ssissippi |

Suffixes 4, 6 and 3, share the prefix s ...
Practical speed-up

Observation
If suffixes \( SA[L] \) and \( SA[R] \) share a prefix, then suffixes \( SA[K] \), for \( K=L, L+1, ..., R \), share the same prefix ...

7: sippi
L 4: sississippi
M 6: ssippi
R 3: ssissippi

Suffixes 4, 6 and 3, share the prefix s ...

Trick
\[
P = \min\{\text{lcp}(u, SA[L]), \text{lcp}(u, SA[R])\},
\]
then
\[
u[1..P] = x[SA[K]..SA[K]+P-1]
\]
for all \( K=L, L+1, ..., R \),
i.e. don’t inspect this part of the pattern
Why care?

... searching for $u[1..m]$ in $x[1..n]$ takes time $O(m)$ using a suffix tree but $O(m \log n)$ using a suffix array, why care about suffix arrays?
Why care?

... searching for \( u[1..m] \) in \( x[1..n] \) takes time \( O(m) \) using a suffix tree but \( O(m \log n) \) using a suffix array, why care about suffix arrays?

A suffix array consumes only \( 4n \) bytes, much less than a suffix tree ...
Why care?

... searching for $u[1..m]$ in $x[1..n]$ takes time $O(m)$ using a suffix tree but $O(m\log n)$ using a suffix array, why care about suffix arrays?

A suffix array consumes only $4n$ bytes in practice ...
Why care?

... searching for $u[1..m]$ in $x[1..n]$ takes time $O(m)$ using a suffix tree but $O(m\log n)$ using a suffix array, why care about suffix arrays?

\[ \log n \text{ isn’t much in practice} \]

A suffix array consumes only $4n$ bytes in practice ....

The real reasons

Suffix trees can systematically be replaced with suffix arrays \textit{without} loss of time [AKO 2004] ... 

The suffix array can be constructed in time $O(n)$, \textit{without} using the suffix tree [KS 2003] ...
Enhanced suffix arrays

The suffix array plus additional tables of $n$ integers ...

\begin{itemize}
  \item \textbf{Mississippi}
  \item 11: \textit{i} \hspace{1cm} 0
  \item 8: \textit{ippi} \hspace{1cm} 1
  \item 5: \textit{issippi} \hspace{1cm} 1
  \item 2: \textit{ississppi} \hspace{1cm} 4
  \item 1: \textit{mississippi} \hspace{1cm} 0
  \item 10: \textit{pi} \hspace{1cm} 0
  \item 9: \textit{ppi} \hspace{1cm} 1
  \item 7: \textit{sippi} \hspace{1cm} 0
  \item 4: \textit{sississippi} \hspace{1cm} 2
  \item 6: \textit{sippi} \hspace{1cm} 1
  \item 3: \textit{ssississippi} \hspace{1cm} 3
\end{itemize}

\textit{lcp}(SA[i], SA[i-1])

... makes it possible to “simulate” top-down and bottom-up traversals of suffix trees and the concept of suffix-links ...
Farach’s idea for suffix tree construction

**Step 1**

Construct the suffix tree of the suffixes starting at odd positions.

... done recursively by reducing the problem to a string of half size ...

**Step 2**

Construct the suffix tree of the remaining positions.

... done using the suffix tree constructed in step 1 ...

**Step 3**

Merge the two suffix trees into one
Linear time construction

Kärkkäinen and Sanders’s idea for suffix array construction

**Step 1**
Construct the suffix array of the suffixes starting at positions $i \mod 3 \neq 0$.

... done recursively by reducing the problem to a string of $2/3$ size ...

**Step 2**
Construct the suffix array of the remaining suffixes.

... done using the suffix array constructed in step 1 ...

**Step 3**
Merge the two suffix arrays into one
Step 1 - Compute $SA^{12}$

**Suffixes $i \mod 3 \neq 0$**

1: ississippi
2: ssissippi
4: issippi
5: ssippi
7: ippi
8: ppi
10: i

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
</table>
Step 1 - Compute $SA^{12}$

Suffixes $i \mod 3 \neq 0$

1: ississippi
2: ssissippi
4: issippi
5: ssippi
7: ippi
8: ppi
10: i$$

Sort by prefix in time $O(n)$

10: i$$
7: ipp
1: iss
4: iss
8: ppi
2: ssi
5: ssi

sentinels

mississippi

0 1 2 3 4 5 6 7 8 9 10
Step 1 - Compute $SA_{12}$

Suffixes $i \mod 3 \neq 0$

1: ississippi
2: ssissippi
4: issippi
5: ssippi
7: ippi
8: ppi
10: i$$

Sort by prefix in time $O(n)$

10: i$$ 1
7: ipp 2
4: iss 3
1: iss 3
8: ppi 4
2: ssi 5
5: ssi 5

If no suffix is assigned the same lex-name, we are done
Step 1 - Compute $SA^{12}$

Suffixes $i \mod 3 \neq 0$

1: ississippi
2: ssissippi
4: issippi
5: ssippi
7:ippi
8:ppi
10: i$$

Sort by prefix in time $O(n)$

10: i$$ 1
7: ipp 2
1: iss 3
4: iss 3
8: ppi 4
2: ssi 5
5: ssi 5

If no suffix is assigned the same lex-name, we are done, otherwise:

$u = \text{“lex-names for } i \mod 3 = 1\text{”} \# \text{“lex-names for } i \mod 3 = 2\text{”} = 3 3 2 1 \# 5 5 4$

where $\#$ is a special character not occurring anywhere else. The suffix array $SA(u)$ of $u$ implies $SA^{12}$ as there is a 1-1 mapping from suffixes of $u$ and suffixes of $s[i..n]$ where $i \mod 3 \neq 0$
Step 1 - Compute $SA^{12}$

<table>
<thead>
<tr>
<th>Suffixes $i \mod 3 \neq 0$</th>
<th>Sort by prefix in time $O(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ississippi</td>
<td>10: i$$ 1</td>
</tr>
<tr>
<td>2: ssissippi</td>
<td>7: ipp 2</td>
</tr>
<tr>
<td>4: issipii</td>
<td>1: iss 3</td>
</tr>
<tr>
<td>5: ssipii</td>
<td>4: iss 3</td>
</tr>
<tr>
<td>7:ippi</td>
<td>8: ppi 4</td>
</tr>
<tr>
<td>8: ppi</td>
<td>2: ssi 5</td>
</tr>
<tr>
<td>10: i$$</td>
<td></td>
</tr>
</tbody>
</table>

**Time:** $T(n) = O(n) + T(\lceil 2n/3 \rceil) = O(n)$

If no suffix is assigned the same lex-name, we are done, otherwise:

$u = \text{“lex-names for } i \mod 3 = 1\text{”} \# \text{“lex-names for } i \mod 3 = 2\text{”} = 3 3 2 1 \# 5 5 4$

where # is a special character not occurring anywhere else. The suffix array $SA(u)$ of $u$ implies $SA^{12}$ as there is a 1-1 mapping from suffixes of $u$ and suffixes of $s[i..n]$ where $i \mod 3 \neq 0$
Step 1 - Why ...

\[ u = 3 \ 3 \ 2 \ 1 \ # \ 5 \ 5 \ 4 \]

iss iss ipp i$$ ssi ssi ppi

Lexicographical names

10: i$$ 1
7: ipp 2
1: iss 3
4: iss 3
8: ppi 4
2: ssi 5
5: ssi 5
Step 1 - Why ... 

\[ u = 3 \ 3 \ 2 \ 1 \ # \ 5 \ 5 \ 4 \]

\( \text{iss iss ipp i} $$\)

\( \text{ssi ssi ppi} \)

**Suffixes of** \( u \)

0 : 3321#554
1 : 321#554
2 : 21#554
3 : 1#554
4 : 554
5 : 54
6 : 4
Step 1 - Why ...

\[ u = 3 \ 3 \ 2 \ 1 \ # \ 5 \ 5 \ 4 \]

iss iss ipp i$$
ssi ssi ppi

Suffixes of \( u \)

0: 3321#554  ississippi$$#ssissippi
1: 321#554  issippi$$#ssissippi
2: 21#554  ippi$$#ssissippi
3: 1#554  i$$#ssissippi
4: 554  ssissippi
5: 54  ssippi
6: 4  ppi

Lex-names expanded to corresponding prefixes of length 3
Step 1 - Why ...

\[ u = 3 3 2 1 \# 5 5 4 \]

**Lex-names expanded to corresponding prefixes of length 3**

\[ i < j \iff xxx < yyy, \text{i.e.} 1 < 3 \text{ because } i\$\$ < i\$ss \]

**Suffixes of** \( u \)

\[
\begin{align*}
0: & \quad 3321\#554 \quad \text{ississippi}\$\$#\text{ssissippi} \\
1: & \quad 321\#554 \quad \text{issippi}\$\$#\text{ssissippi} \\
2: & \quad 21\#554 \quad \text{ippi}\$\$#\text{ssissippi} \\
3: & \quad 1\#554 \quad \text{i}\$\$#\text{ssissippi} \\
4: & \quad 554 \quad \text{ssissippi} \\
5: & \quad 54 \quad \text{ssippi} \\
6: & \quad 4 \quad \text{ppi}
\end{align*}
\]
Step 1 - Why ...

\[ u = 3 \ 3 \ 2 \ 1 \ # \ 5 \ 5 \ 4 \]

Suffixes of \( u \)

0: 3321#554 ississippi$$#ssissippi
1: 321#554 issippi$$#ssissippi
2: 21#554 ippi$$#ssissippi
3: 1#554 i$$#ssissippi
4: 554 ssissippi
5: 54 ssippi
6: 4 ppi

Since the special character # is unique and occurs in unique positions, we can sort these strings by sorting the prefixes (of \( u \)) up to #
Step 1 - Why ...

\[
u = \begin{array}{cccccc}
1 & 3 & 3 & 2 & 1 & \# \\
2 & 3 & 2 & 1 & 5 & 5 & 4
\end{array}
\]

\[
\text{iss iss iss} \quad \text{issippi} \\
\text{isti ssi ssi} \quad \text{ippi ppi}
\]

**Suffixes of** \( u \)

\[
\begin{array}{lcl}
0 & : & 3321#554 \\
1 & : & 321#554 \\
2 & : & 21#554 \\
3 & : & 1#554 \\
4 & : & 554 \\
5 & : & 54 \\
6 & : & 4
\end{array}
\]

\[
\begin{array}{l}
\text{ississippi}\#\text{ssissippi} \\
\text{issippi}\#\text{ssissippi} \\
\text{ippi}\#\text{ssissippi} \\
\text{i}\#\text{ssissippi} \\
\text{ssissippi} \\
\text{ssippi} \\
\text{ppi}
\end{array}
\]

Since the sentinel $ is unique and occurs in unique positions, we can sort these strings by sorting the prefixes (of \( u \)) up to the sentinel $ ...
Step 1 - Why ...

Since the sentinel $ is unique and occurs in unique positions, we can sort these strings by sorting the prefixes (of $u$) up to the sentinel $$. ...
Step 1 - Why ...

Since the sentinel $ is unique and occurs in unique positions, we can sort these strings by sorting the prefixes (of $u$) up to the sentinel $ ...
Step 2 - Compute $SA^0$

Suffixes $i \mod 3 = 0$

0: missississippi
3: sissippi
6: sippi
9: pi
Step 2 - Compute $SA^0$

Suffixes $i \mod 3 = 0$

0: mississippi  m $s[1..]$
3: sissippi    s $s[4..]$
6: sippi       s $s[7..]$
9: pi          p $s[10..]$

Idea: every suffix $i \mod 3 = 0$ can be written $s[i] s[i+1..]$ where $i+1 \mod 3 \neq 0$, i.e. the ordering of $s[i+1..]$ is known c.f. $SA^{12} ...$
Step 2 - Compute $SA^0$

Suffixes $i \mod 3 = 0$

0: mississippi $m \ s[1..]$  
3: sissippi $s \ s[4..]$  
6: sippi $s \ s[7..]$  
9: pi $p \ s[10..]$  

Sort by first letter and use (inverse) $SA^{12}$ to solve ties:

0: $m \ s[1..]$  
9: $p \ s[10..]$  
6: $s \ s[7..]$  
3: $s \ s[4..]$  

Idea: every suffix $i \mod 3 = 0$ can be written $s[i..]s[i+1..]$ where $i+1 \mod 3 \neq 0$, i.e. the ordering of $s[i+1..]$ is known c.f. $SA^{12}$...
Step 2 - Compute $SA^0$

Suffixes $i \mod 3 = 0$

0: mississippi  m s[1..]
3: sissippi    s s[4..]
6: sippi       s s[7..]
9: pi          p s[10..]

Can be sorted in time $O(n)$ using radix-sort where the ordering of the suffixes $s[1..]$, $s[4..]$ etc. can be determine in constant time using $SA^{12}$

Sort by first letter and use (inverse) $SA^{12}$ to solve ties:

0: m s[1..]
9: p s[10..]
6: s s[7..]
3: s s[4..]

Written as $S[0..] S[1..] ..$ where $S[-1..]$ is know c.f. $SA^{12} ..$
Step 3 - Merging

**Idea:** every suffix $j$ can be written $s[j] \ s[j{+}1..]$ and $s[j] \ s[j{+}1] \ s[j{+}2..]$, where $j{+}1 \ mod \ 3 \neq 0$ and/or $j{+}2 \ mod \ 3 \neq 0$, and the ordering of suffixes $s[i..]$ for $i \ mod \ 3 \neq 0$ is know c.f. $SA^{12} \ldots$

---

**Example**

Determine order of suffix $s[3..]$ and $s[7..]$

- $s[3..] = s \ s[4..]$
- $s[7..] = s \ s[8..]$

$s[4..] < s[8..]$, i.e. $s[3..] < s[7..]$
Step 3 - Merging

Let \( i \) be an element in \( SA^0 \) and \( j \) be an element in \( SA^{12} \). We can always determine the ordering of \( s[i..] \) and \( s[j..] \) in time \( O(1) \).

**Case 1:** If \( j \mod 3 = 1 \) then consider \( s[i..] = s[i]s[i+1..] \) and \( s[j..] = s[j]s[j+1..] \). Since \( i \mod 3 = 0 \) and \( j \mod 3 = 1 \), then \( (i+1) \mod 3 = 1 \) and \( (j+1) \mod 3 = 2 \), i.e. the ordering of \( s[i+1..] \) and \( s[j+1..] \) can be determined from \( SA^{12} \).

**Case 2:** If \( j \mod 3 = 2 \) then consider \( s[i..] = s[i] s[i+1]s[i+2..] \) and \( s[j..] = s[j] s[j+1] s[j+2..] \). Since \( i \mod 3 = 0 \) and \( j \mod 3 = 2 \), then \( (i+2) \mod 3 = 2 \) and \( (j+2) \mod 3 = 1 \), i.e. the ordering of \( s[i+2..] \) and \( s[j+2..] \) can be determined from \( SA^{12} \).

In both cases the ordering of \( s[i..] \) and \( s[j..] \) can be determined in time \( O(1) \) by inspecting a constant number of symbols and maybe \( SA^{12} \).
Step 3 - Merging

**SA:**

$m s[1..]$  

0 9 6 3  

10 7 4 1 8 5 2  

m i s s i s s i p p i

0 1 2 3 4 5 6 7 8 9 10
Step 3 - Merging

\[ \text{m} s[1..] \quad \text{i} s[8..] \]

\[
\begin{array}{cccc}
0 & 9 & 6 & 3 \\
\end{array}
\quad \begin{array}{cccccc}
10 & 7 & 4 & 1 & 8 & 5 & 2 \\
\end{array}
\]

\text{SA: 10,}

\[
\begin{array}{cccccccccccc}
\text{m} & \text{i} & \text{s} & \text{i} & \text{s} & \text{i} & \text{s} & \text{i} & \text{p} & \text{i} \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10
\end{array}
\]
Step 3 - Merging

$m[1..]$

\[
\begin{array}{cccc}
0 & 9 & 6 & 3 \\
\end{array}
\]

$i[5..]$

\[
\begin{array}{cccccc}
10 & 7 & 4 & 1 & 8 & 5 & 2 \\
\end{array}
\]

SA: 10, 7,

misssissippi

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
\]
Step 3 - Merging

\[ m_s[1..] \]

\[
\begin{array}{cccc}
0 & 9 & 6 & 3 \\
\end{array}
\]

\[ i_s[2..] \]

\[
\begin{array}{cccccc}
10 & 7 & 4 & 1 & 8 & 5 & 2 \\
\end{array}
\]

SA: 10, 7, 4,

\[
\begin{array}{cccccccccccc}
m& i & s& s& s& s& s& s& i& p& p& i \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
\]

Monday, May 6, 13
Step 3 - Merging

SA: 10, 7, 4, 1,
Step 3 - Merging

SA: 10, 7, 4, 1, 0,

mississippi

0 1 2 3 4 5 6 7 8 9 10
Step 3 - Merging

SA: 10, 7, 4, 1, 0, 9,

mississippi
Step 3 - Merging

SA: 10, 7, 4, 1, 0, 9, 8,

mississippi
Step 3 - Merging

\[ \text{SA: 10, 7, 4, 1, 0, 9, 8, 6,} \]

\[ m i s s i s s i p p i \]

Monday, May 6, 13
Step 3 - Merging

SA: 10, 7, 4, 1, 0, 9, 8, 6, 3,

mississippi
Step 3 - Merging

SA: 10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2

m i s s i s s i p p i

0 1 2 3 4 5 6 7 8 9 10
Step 3 - Merging

**Time:** $O(n)$

SA: 10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2

*mississippi*

0 1 2 3 4 5 6 7 8 9 10
The “Skew” algorithm

**Step 1**
Construct the suffix array of the suffixes starting at positions $i \mod 3 \neq 0$.
... done recursively by reducing the problem to a string of 2/3 size ...

**Step 2**
Construct the suffix array of the remaining suffixes.
... done using the suffix array constructed in step 1 ...

**Step 3**
Merge the two suffix arrays into one
The “Skew” algorithm

... each step takes time $O(n)$, i.e **total time**: $O(n)$ ...

**Step 1**
Construct the suffix array of the suffixes starting at positions $i \mod 3 \neq 0$.

... done recursively by reducing the problem to a string of $2/3$ size ...

**Step 2**
Construct the suffix array of the remaining suffixes.

... done using the suffix array constructed in step 1 ...

**Step 3**
Merge the two suffix arrays into one
Source code

```c
A Source Code

The following C++ code contains a complete linear time implementation of suffix array construction.
This code strives for conciseness rather than for speed. It has only 50 lines not counting comments,
empty lines, and lines with a bracket only. A driver program can be found at http://www.sandia.
mpg.de/sanders/programs/suffix/.

inline bool leq(int a1, int a2, int b1, int b2) // lexicographic order
{return (a1 < b1 || a1 == b1 && a2 <= b2);} // for pairs
inline bool leq(int a1, int a2, int a3, int b1, int b2, int b3)
{return (a1 < b1 || a1 == b1 && leq(a2, a3, b2, b3));} // and triples

// stably sort a[0..n-1] to b[0..n-1] with keys in 0..X from r
static void radixPass(int* a, int* b, int* r, int n, int X)
{ // count occurrences
int c = new int[X + 1]; // counter array
for (int i = 0; i < X; i++) c[i] = 0; // reset counters
for (int i = 0; i < n; i++) c[r[a[i]]]++; // count occurrences
for (int i = 0, sum = 0; i < X; i++) // exclusive prefix sums
{ int t = c[i]; c[i] = sum = sum + t;
for (int i = 0; i < n; i++) b[c[r[a[i]]]]++ - a[i]; // sort
}

// find the suffix array SA of a[0..n-1] in (1..n)^n
// require suffixArray(int* a, int* SA, int n, int X)
void suffixArray(int* a, int* SA, int n, int X)
{ int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=3*n2;
int a12 = new int[n02+3]; a12[0]=a12[n02+1]=a12[n02+2]=0;
int a12= new int[n02+3]; a12[0]=a12[n02+1]=a12[n02+2]=0;
int a0 = new int[n02];
int* a03 = new int[n03];

// generate positions of mod 1 and mod 2 suffixes
// the +(n0-n1) adds a dummy mod 1 suffix if n3 == 1
for (int i=0; i<n0-n1; i++) if ((i3 = i) a12[j+1] = i;

// lab radix sort the mod 1 and mod 2 triples
radixPass(a12, a12, a12, n02, X);
radixPass(a12, a12, a12, n02, X);
radixPass(a12, a12, a12, n02, X);

// find lexicographic names of triples
int name = 0, c0 = -1, c1 = -1, c2 = -1;
for (int i = 0; i < n02; i++)
{ if (a[SA12[i]] = c0 || a[SA12[i]+1] = c1 || a[SA12[i]+2] = c2)
{name++; c0 = a[SA12[i]]; c1 = a[SA12[i]+1]; c2 = a[SA12[i]+2];}
if (SA12[i] % 3 == 1) (a[SA12[i]/3] = name); // left half
else (a[SA12[i]/3] + n0 = name); // right half
}

// recurse if names are not yet unique
if (name < n02) {
suffixArray(a12, a12, n02, name);
// store unique names in a12 using the suffix array
for (int i = 0; i < n02; i++) a12[SA12[i]] = i + 1;
else // generate the suffix array of a12 directly
for (int i = 0; i < n02; i++) SA12[SA12[i] - 1] = i;

// stably sort the mod 0 suffixes from a12 by their first character
for (int i = 0; i < n02; i++) if (SA12[i] < n0) a0[j++] = 3*SA12[i];
radixPass(a0, a0, s, n0, X);

// merge sorted a0 suffixes and sorted a12 suffixes
for (int p = 0, t = a0-n1, k = 0; k < n; k++)
{ if (t = a0) // check for current offset 12 suffix
int k = a0[p]; // pos of current offset 0 suffix
if (SA12[s] < n0) // different compare for mod 1 and mod 2 suffixes
leq[a[i], a12[SA12[s] + n0], s[j], a12[j]/3] = a12[SA12[i] + n0 + 1];
leq[a[i], a12[SA12[i] + n0 + 1], s[j], a12[j]/3 + n0];

// suffix from a12 is smaller
SA[k] = i; t++;
if (t = a0) // done ---- only a0 suffixes left
for (k++; p < n0; p++) SA[k] = a0[p];
else // suffix from a12 is smaller
SA[k] = j; p++;
if (p = n0) // done ---- only a12 suffixes left
for (k++; t < n0; t++, k++) SA[k] = et (j);
}
}
delte [] a12; delete [] SA12; delete [] a0; delete [] s0;
```
Things to remember

Think in terms of suffix trees, but use suffix arrays ...