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:

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

... assumes an ordered alphabet ...

... consumes \( \text{lint}1 \cdot n \) bytes, usually \( 4n \) bytes ...
<table>
<thead>
<tr>
<th>Mississippi</th>
<th>1: mississippi</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2: ississippi</td>
</tr>
<tr>
<td></td>
<td>3: ssissippi</td>
</tr>
<tr>
<td></td>
<td>4: sissippi</td>
</tr>
<tr>
<td></td>
<td>5: issippi</td>
</tr>
<tr>
<td></td>
<td>6: ssippi</td>
</tr>
<tr>
<td></td>
<td>7: sippi</td>
</tr>
<tr>
<td></td>
<td>8:ippi</td>
</tr>
<tr>
<td></td>
<td>9: ppi</td>
</tr>
<tr>
<td></td>
<td>10: pi</td>
</tr>
<tr>
<td></td>
<td>11: i</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Mississippi</th>
<th>1: mississippi</th>
<th>11: i</th>
</tr>
</thead>
<tbody>
<tr>
<td>2: ississippi</td>
<td>8: ippi</td>
<td></td>
</tr>
<tr>
<td>3: ssissippi</td>
<td>5: ississippi</td>
<td></td>
</tr>
<tr>
<td>4: sissippi</td>
<td>2: ississippi</td>
<td></td>
</tr>
<tr>
<td>5: ississippi</td>
<td>1: mississippi</td>
<td></td>
</tr>
<tr>
<td>6: ssippi</td>
<td>10: pi</td>
<td></td>
</tr>
<tr>
<td>7: sippi</td>
<td>9: ppi</td>
<td></td>
</tr>
<tr>
<td>8: ippi</td>
<td>7: sippi</td>
<td></td>
</tr>
<tr>
<td>9: ppi</td>
<td>4: sissippi</td>
<td></td>
</tr>
<tr>
<td>10: pi</td>
<td>6: ssippi</td>
<td></td>
</tr>
<tr>
<td>11: i</td>
<td>3: ssissippi</td>
<td></td>
</tr>
</tbody>
</table>
An example

Mississippi

1: mississippi
2: ississippi
3: ssissippi
4: sissippi
5: ississippi
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

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

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

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

$S = \text{Mississippi}$

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

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

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>mississippi</td>
<td>1</td>
</tr>
<tr>
<td>ississippi</td>
<td>2</td>
</tr>
<tr>
<td>ssissippi</td>
<td>3</td>
</tr>
<tr>
<td>sissippi</td>
<td>4</td>
</tr>
<tr>
<td>issippi</td>
<td>5</td>
</tr>
<tr>
<td>ssippi</td>
<td>6</td>
</tr>
<tr>
<td>sippi</td>
<td>7</td>
</tr>
<tr>
<td>ippi</td>
<td>8</td>
</tr>
<tr>
<td>ppi</td>
<td>9</td>
</tr>
<tr>
<td>pi</td>
<td>10</td>
</tr>
<tr>
<td>i</td>
<td>11</td>
</tr>
</tbody>
</table>
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: missississippi

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

R  3: ssississippi

ssi > pi
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: sississippi
  6: ssippi

R  3: ssissippi
An example

Searching for $ssi$ in $mississippi$

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

$ssi = ssi$
An example

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

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

\begin{tabular}{ll}
L & 4: \texttt{sissippi} \\
M & 6: \texttt{ssippi} \\
R & 3: \texttt{ssississippi} \\
\end{tabular}

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

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

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

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

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

\textbf{Note:} that all $k$ occurrences of \textit{ssi} are indexed by neighboring positions in \textit{SA} ...
The naive algorithm

\[ j = 0; \ L = 1; \ R = n \]
\begin{algorithm}
\begin{repeat}
\quad \ M = \lfloor (R+L)/2 \rfloor \\
\quad \text{if } u = x[SA[M] .. SA[M]+m-1] \text{ then} \\
\quad \quad \ j = SA[M] \\
\quad \text{elseif } u > x[SA[M] .. SA[M]+m-1] \text{ then} \\
\quad \quad \quad \ L = M \\
\quad \text{else} \\
\quad \quad \ R = M \\
\end{repeat}
\end{algorithm}

\textbf{Time: } \mathcal{O}(m(\log n + k))
The naive algorithm

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

repeat

\[
M = \left\lfloor \frac{(R+L)}{2} \right\rfloor 
\]

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

\[
j = SA[M]
\]

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

\[
L = M 
\]

else

\[
R = M 
\]

until \( L = M \) or \( j \neq 0 \)

**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 $K=L, L+1,...,R$, share the same prefix ...

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 $K=L, L+1, \ldots, R$, share the same prefix ...

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

$$u[1..P] = x[SA[K] \ldots SA[K]+P-1]$$

for all $K=L, L+1, \ldots, R$,
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, much less than a suffix tree ...

$log n$ isn’t much 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?

**The real reasons**

Suffix trees can systematically be replaced with suffix arrays *without* loss of time [AKO 2004] ...

The suffix array can be constructed in time \( O(n) \), *without* using the suffix tree [KS 2003] ...

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

\( \log n \) isn’t much in practice ...
**Enhanced suffix arrays**

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

<table>
<thead>
<tr>
<th>Mississippi</th>
<th>lcp(SA[i], SA[i-1])</th>
</tr>
</thead>
<tbody>
<tr>
<td>11: i</td>
<td>0</td>
</tr>
<tr>
<td>8:ippi</td>
<td>1</td>
</tr>
<tr>
<td>5:issippi</td>
<td>1</td>
</tr>
<tr>
<td>2:ississippi</td>
<td>4</td>
</tr>
<tr>
<td>1:mississippi</td>
<td>0</td>
</tr>
<tr>
<td>10:pi</td>
<td>0</td>
</tr>
<tr>
<td>9:ppi</td>
<td>1</td>
</tr>
<tr>
<td>7:sippi</td>
<td>0</td>
</tr>
<tr>
<td>4:sississippi</td>
<td>2</td>
</tr>
<tr>
<td>6:sippi</td>
<td>1</td>
</tr>
<tr>
<td>3:ssissippi</td>
<td>3</td>
</tr>
</tbody>
</table>

... makes it possible to “simulate” top-down and bottom-up traversals of suffix trees and the concept of suffix-links...
Linear time construction

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
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 $i \mod 3 \neq 0$

<table>
<thead>
<tr>
<th></th>
<th>Suffixes</th>
<th></th>
<th>Suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ississippi</td>
<td>10</td>
<td>i$$</td>
</tr>
<tr>
<td>2</td>
<td>ssissippi</td>
<td>7</td>
<td>ipp</td>
</tr>
<tr>
<td>4</td>
<td>issippi</td>
<td>1</td>
<td>iss</td>
</tr>
<tr>
<td>5</td>
<td>ssippi</td>
<td>4</td>
<td>iss</td>
</tr>
<tr>
<td>7</td>
<td>ippi</td>
<td>8</td>
<td>ppi</td>
</tr>
<tr>
<td>8</td>
<td>ppi</td>
<td>2</td>
<td>ssi</td>
</tr>
<tr>
<td>10</td>
<td>i$$</td>
<td>5</td>
<td>ssi</td>
</tr>
</tbody>
</table>

Sort by prefix in time $O(n)$

If no suffix is assigned the same lex-name, we are done.
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$$ 1</td>
</tr>
<tr>
<td>2: ssissippi</td>
<td>7: ipp 2</td>
</tr>
<tr>
<td>4: issippi</td>
<td>1: iss 3</td>
</tr>
<tr>
<td>5: ssippi</td>
<td>4: iss 3</td>
</tr>
<tr>
<td>7: ippi</td>
<td>8: ppi 4</td>
</tr>
<tr>
<td>8: ppi</td>
<td>2: ssi 5</td>
</tr>
<tr>
<td>10: i$$</td>
<td>5: ssi 5</td>
</tr>
</tbody>
</table>

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: issippi</td>
<td>1: iss 3</td>
</tr>
<tr>
<td>5: ssippi</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>
</tbody>
</table>

Time: $T(n) = O(n) + T([2n/3]) = O(n)$

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 = \begin{array}{cccccccc}
3 & 3 & 2 & 1 & \# & 5 & 5 & 4 \\
iss & iss & ipp & i$$ & ssi & ssi & ppi
\end{array} \]

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 ...

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

**Suffixes of** \( u \)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3321#554</td>
</tr>
<tr>
<td>1</td>
<td>321#554</td>
</tr>
<tr>
<td>2</td>
<td>21#554</td>
</tr>
<tr>
<td>3</td>
<td>1#554</td>
</tr>
<tr>
<td>4</td>
<td>554</td>
</tr>
<tr>
<td>5</td>
<td>54</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>
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

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

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

**Suffixes of u**

\[
\begin{align*}
0 & : 3321\#554 \quad \text{ississippi} & & \text{ississippi} \\
1 & : 321\#554 \quad \text{ississippi} & & \text{ississippi} \\
2 & : 21\#554 \quad \text{ippi} & & \text{ippi} \\
3 & : 1\#554 \quad \text{i} & & \text{i} \\
4 & : 554 \quad \text{ssissippi} & & \text{ssissippi} \\
5 & : 54 \quad \text{SSIP} & & \text{SSIP} \\
6 & : 4 \quad \text{SSIP} & & \text{SSIP} \\
\end{align*}
\]

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

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

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

Suffixes of \( u \)

- 0: 3321#554 ississippi $$# sissippi
- 1: 321#554 sissippi $$# sissippi
- 2: 21#554ippi $$# sissippi
- 3: 1#554 i $$# sissippi
- 4: 554 sissippi
- 5: 54 sippi
- 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 #.
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 $$. 

Suffix array

$u = 3 \ 3 \ 2 \ 1 \ \# \ 5 \ 5 \ 4$

iss iss i$$ ssi ssi ppi

Suffixes of $u$

$0: 3321#554$ ississippi$$#ssissippi 1
$1: 321#554$ issippi$$#ssissippi 4
$2: 21#554$ ippi$$#ssissippi 7
$3: 1#554$ i$$#ssissippi 10
$4: 554$ ssissippi 2
$5: 54$ ssippi 5
$6: 4$ ppi 8
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: mississippi
3: sissippi
6: sippi
9: pi
Step 2 - Compute $SA^0$

Suffixes $i \mod 3 = 0$

0: mississippi  $ms[1..]$
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 \( ms[1..] \)
- $3$: sissippi \( ss[4..] \)
- $6$: sippi \( ss[7..] \)
- $9$: pi \( ps[10..] \)

The order of suffix 6 and 3 is by the position of \( ss[7..] \) and \( ss[4..] \) in $SA^{12}$.

Sort by first letter and use (inverse) $SA^{12}$ to solve ties:
- $0$: m \( ms[1..] \)
- $9$: p \( ps[10..] \)
- $6$: s \( ss[7..] \)
- $3$: s \( ss[4..] \)

The written $ss[4..]$ is fixed where $ss[i+1..]$ is known c.f. $SA^{12}$...

\[10 \ 7 \ 4 \ 1 \ 8 \ 5 \ 2\]

mississippi
0 1 2 3 4 5 6 7 8 9 10
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..]

The order of suffix 6 and 3 is by the position of $s[7..]$ and $s[4..]$ in $SA^{12}$.

Time: $O(n)$

The table below shows the order of symbols in $SA^{12}$:

<table>
<thead>
<tr>
<th>10</th>
<th>7</th>
<th>4</th>
<th>1</th>
<th>8</th>
<th>5</th>
<th>2</th>
</tr>
</thead>
</table>

mississippi

0 1 2 3 4 5 6 7 8 9 10
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 known c.f. \( SA^{12} \) ...

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

Example

Determine order of suffix \( s[1..] \) and \( s[4..] \)

\[
s[1..] = i s[2..]
\]

\[
s[4..] = i s[5..]
\]

\[
s[5..] < s[2..], \text{ i.e. } s[4..] < s[1..]
\]
Step 3 - Merging

SA:

mississippi

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

SA: 10,

mississippi

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

**SA:** 10, 7,

\begin{align*}
ms[1..] & \quad ms[5..] \\
0 & 9 & 6 & 3 & 10 & 7 & 4 & 1 & 8 & 5 & 2
\end{align*}

\textit{mississippi}

\begin{align*}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10
\end{align*}
Step 3 - Merging

m s[1..]

0 9 6 3

i s[2..]

10 7 4 1 8 5 2

SA: 10, 7, 4,

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

\[ m_s[1..] \]

\[ p_{s[9..]} \]

\[ p_{p_{s[10..]}} \]

\[ \text{SA: 10, 7, 4, 1,} \]

\[ \text{mississippi} \]
Step 3 - Merging

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

\[ \text{mississippi} \]

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

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

\[\text{mississippi}\]

\[\text{0 1 2 3 4 5 6 7 8 9 10}\]
Step 3 - Merging

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

SA: 10, 7, 4, 1, 0, 9, 8,
Step 3 - Merging

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

mississippi
Step 3 - Merging

Time: $O(n)$

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

The following C++ file 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.mpi-sb.mpg.de/~sanders/programs/suffixarray/

```
inline bool leq(int x1, int x2, int h1, int h2) // lexicographic order
{ return (h1 < h2) || (h1 == h2 && x1 < x2); } // for pairs
inline bool leq(int x1, int x2, int x3, int h1, int h2, int h3)
{ return (h1 < h2) || (h1 == h2 && leq(x2, x3, h1, h2, h3)); } // and triples

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

// find the suffix array S[0..n-1] in (1..0)n
// require S[0..n-1] = e[0..2n]+o, n=2n
void suffixArray(int* e, int n, int x, int k)
{ int n0=(n+1)/3; n1=(n+1)/3; n2=n/3, n02=0;
  int* s2 = new int[n+3];
  int* s1 = new int[n+3];
  int* s0 = new int[n0];
  int* S0 = new int[n0];

  // generate positions of mod 1 and mod 2 suffixes
  // for the "0 either" add a dummy mod 1 suffix if n/3 == 1
  for (int i=0, j=0; i < n0*(x+1); i++) S0[i] = (i/3 == 0) ? s1[i+1] : i;
  // let radix sort the mod 1 and mod 2 triples
  radixPass(s1, S1, n0, x, K);
  radixPass(s2, S2, n02, x2, K);
  radixPass(s0, S0, n0, x0, K);

  // find lexicographic names of triples
  int name = 0, c0 = -1, c1 = -1, c2 = -1;
  for (int i = 0; i < n; i++)
  { if (S0[i] < 0) i++;
    if (S0[i] < 0) i++;
    name++; c0 = S0[i]; c1 = S0[i+1]; c2 = S0[i+2];
  } else
  { s1[S1[0]+1] = name; // right half
    if (S1[0] + 1) i++;
  }

  // recurses if names are not yet unique
  if (name < n2)
  { suffixArray(s1, S1, n0, name);
    // store unique names in S12 using the suffix array
    for (int i = 0; i < n0; i++) S12[i] = 0;
    // generate suffix array of S12 directly
    for (int i = 0; i < n02; i++) S13[0][i] = 0;
    // stably sort the n0 suffixes from S12 by their first character
    for (int i = 0, j=0; i < n0; i++)
    if (S12[i] < 0) S12[i] = S12[i+1] - S13[0][i] - 3*S13[1][i] -
    // merge sorted S09 suffixes and sorted S12 suffixes
    for (int p=0, t=n02; p < n2; p++)
    if (S12[i] < 0) S12[i] = S12[i+1] - 3*S13[1][i] -
    // define -st () S12[t] < n0 S12[t] + 3 + i : (S12[t] - n0) + i
    int i = 0; // pos of current offset 0 suffix
    int j = S0[p]; // pos of current offset 0 suffix
    if (S12[t] < n0) // different compares for mod 1 and mod 2 suffixes
    { leq(s1, s12(S12[t]+n0), c1, s12(S12[t]+n0)+12);
      leq(s1, s12(S12[t]+n0), c1, s12(S12[t]+n0)+12);
    } else // suffix from S12 is smaller
    { S13[0][i] = 0; S13[1][i] = 0;
      if (t == n0) // done --- only S09 suffixes left
      for (t++)
      { if (p < n0) // done --- only S12 suffixes left
        for (t++, k++) S13 = S10[p];
      } else // suffix from S12 is smallest
      { S13[0][i] = 0; S13[1][i] = 0;
        if (p == n2) // done --- only S12 suffixes left
        for (t++, k++) S13 = st();
      }
    } delete[] s12; delete[] S1; delete[] S0; delete[] s0;
  }
```
Things to remember

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