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 \(|\text{int|}\cdot n \) bytes, usually \( 4n \) bytes ...
An example

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

Mississippi

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

\[ SA = [11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3] \]
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. $T(v) = SA[8..11]$
Constructing suffix arrays

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

Mississippi

1: mississippi  
2: ississippi  
3: ssissippi  
4: sississippi  
5: ississippi  
6: ssippi  
7: sippi  
8: ippi  
9: ppi  
10: pi  
11: i
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: $O(n \cdot \alpha) \ldots$
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: ssissippi

ssi > pi
An example

Searching for ssi in mississippi

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

ssi > sis
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: ssiippi
R  3: ssissippi

ssi = ssi
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: ssiippi
R  3: ssissippi

Note: that all k occurrences of ssi are indexed by neighboring positions in SA ...
An example

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

11: i
10: pi
 9: ppi
 7: sippi
 L 4: sississippi
 M 6: ssippi
 R 3: ssissippi

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

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

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

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

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

\[
j = 0; \quad L = 1; \quad R = n
\]
repeat
\[
M = \lfloor (R+L)/2 \rfloor
\]
if \( u = x[SA[M] \ldots SA[M]+m-1] \) then
\[
j = SA[M]
\]
ext elif \( u > x[SA[M] \ldots SA[M]+m-1] \) then
\[
L = M
\]
else
\[
R = M
\]
until \( L = M \) 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
L 4: sissippi
M 6: ssippi
R 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 $K=L, L+1, \ldots, R$, share the same prefix ...

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

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

A suffix array consumes only $4n$ bytes, much less than a suffix tree ...

log $n$ isn’t much 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 ...

<table>
<thead>
<tr>
<th>Mississippi</th>
<th>11: i</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8: ippi</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>5: issippi</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2: ississppi</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>1: mississippi</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>10: pi</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9: ppi</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>7: sippi</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4: sissippi</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6: sippi</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3: ssississippi</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

$lcp(SA[i], SA[i-1])$ makes it possible to “simulate” top-down and bottom-up traversals of suffix trees and the concept of suffix-links ...
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

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>mi</td>
<td>ss</td>
<td>sis</td>
<td>si</td>
<td>sippi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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**
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.

m i s s i s s i p p i

m i s s i s s i p p i

0 1 2 3 4 5 6 7 8 9 10

lexicographical naming
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:

$u = "\text{lex-names for } i \mod 3 =1"$ # "lex-names for $i \mod 3 =2" = 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 $u$ to the suffixes $s[i..n]$ where $i \mod 3 \neq 0$. 

**Lexicographical naming**
Step 1 - Compute $SA_{12}$

Suffixes $i \mod 3 \neq 0$

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

Sort by prefix in time $O(n)$

<table>
<thead>
<tr>
<th>$i$</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i$$ 1</td>
</tr>
<tr>
<td>7</td>
<td>ipp 2</td>
</tr>
<tr>
<td>1</td>
<td>iss 3</td>
</tr>
<tr>
<td>4</td>
<td>iss 3</td>
</tr>
<tr>
<td>8</td>
<td>ppi 4</td>
</tr>
<tr>
<td>2</td>
<td>ssi 5</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 = \begin{array}{ccccccc} 3 & 3 & 2 & 1 & \# & 5 & 5 & 4 \\ iss & iss & ipp & i$$ & ssi & ssi & ppi \end{array} \]

**Suffixes of** \( u \)**

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

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

**Suffixes of u**

0: 3321#554  ississippi$$#ssissippi
1: 321#554  issipii$$#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 ...

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

<table>
<thead>
<tr>
<th>Suffixes of u</th>
<th>Lex-names expanded to corresponding prefixes of length 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: 3321#554</td>
<td>ississippi$$#ssissippi</td>
</tr>
<tr>
<td>1: 321#554</td>
<td>issippi$$#ssissippi</td>
</tr>
<tr>
<td>2: 21#554</td>
<td>ippi$$#ssissippi</td>
</tr>
<tr>
<td>3: 1#554</td>
<td>i$$#ssissippi</td>
</tr>
<tr>
<td>4: 554</td>
<td>ssissippi</td>
</tr>
<tr>
<td>5: 54</td>
<td>ssippi</td>
</tr>
<tr>
<td>6: 4</td>
<td>ppi</td>
</tr>
</tbody>
</table>

$$i < j \iff xxx < yyy, \text{i.e } 1 < 3 \text{ because } i$$ $$< iss$$
Step 1 - Why ...

$u = \begin{bmatrix} 3 & 3 & 2 & 1 & \# & 5 & 5 & 4 \end{bmatrix}$

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

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

Suffixes $i \mod 3 = 0$

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

Suffixes $i \mod 3 = 0$

<table>
<thead>
<tr>
<th>0</th>
<th>mississippi</th>
<th>$m$ $s[1..]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>sissippi</td>
<td>$s$ $s[4..]$</td>
</tr>
<tr>
<td>6</td>
<td>sippi</td>
<td>$s$ $s[7..]$</td>
</tr>
<tr>
<td>9</td>
<td>pi</td>
<td>$p$ $s[10..]$</td>
</tr>
</tbody>
</table>

**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 $i \mod 3 = 0$**

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>mississippi</td>
<td>0</td>
</tr>
<tr>
<td>sissippi</td>
<td>3</td>
</tr>
<tr>
<td>sippi</td>
<td>6</td>
</tr>
<tr>
<td>pi</td>
<td>9</td>
</tr>
</tbody>
</table>

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

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

```
| 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 $s[10..]$ |

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 $s[0..]s[4..]..$ 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 known c.f. $SA^{12}$ ...

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:

mississippi

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

SA: 10,

```
ms[1..]  
0 9 6 3

is[8..]  
10 7 4 1 8 5 2
```

```plaintext
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

$m_s[1..]$  $
\begin{array}{cccc}
0 & 9 & 6 & 3 \\
\end{array}$

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

$SA: 10, 7,$

$m\ s\ i\ s\ s\ i\ s\ s\ i\ s\ s\ i\ s\ s\ i\ s\ s\ i\ s\ s\ i\ s\ s\ i\ s\ s\ i\ s\ s\ i\ s\ s\ i\ s\ s\ i\ s\ s\ i\ s\ s\ i\ s\ s\ i\ s\ s\ i\ s\ s\ i\ s\ s\ i\ s\ s$
Step 3 - Merging

SA: 10, 7, 4,

mississippi
Step 3 - Merging

\[
\text{SA: } 10, 7, 4, 1, \\
\text{mississippi}
\]
Step 3 - Merging

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

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

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

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**

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

**mississippi**

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

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

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

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

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

mississippi
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
A Source Code

The following C++ file contains a complete time implementation of suffix array construction. This code strives for conciseness rather than for speed. It has only 90 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/suffix/.  

```cpp
// find lexicographic names of triples
int name = 0, c0 = -1, c1 = -1, c2 = -1;
for (int i = 0; i < n0; i++) {
  if (s[SAIL[i]]) { c0 = s[SAIL[i] - 1]; c1 = s[SAIL[i] - 2]; c2 = s[SAIL[i] - 3];
    if (c1 % 3 == 1) { s12[SAIL[i] / 3] = name; } // left half
    else { s12[SAIL[i] / 3 + n0] = name; } // right half
  }
}

//  // recursive if names are not yet unique
if (name < n0) {
  suffixArray(s12, SAIL, n0, name);
  // store unique names in s12 using the suffix array
  for (int i = 0; i < n0; i++) s12[SAIL[i]] = i + 1;
  else { // generate the suffix array of s12 directly
    for (int i = 0; i < n0; i++) SAIL[x[SAIL[i] - 1]] = i;
  }

  // stably sort the mod 0 suffixes from SAIL by their first character
  for (int i = 0, j = 0; i < n0; i++) if (SAIL[i] < no) s12[i]++ = s12[SAIL[i] / 3] + 3 * SAIL[i];
  radixPass(s12, SAIL, s, no, 0, K);
}

// merge sorted S00 suffixes and sorted S12 suffixes
for (int p = 0, t = no - mi, k = 0; k < n; k++) {
  SAIL[mi] = i;
  if (s12[i] < n0) 
    for (int i = 0, j = 0; i < n; i++) if (SAIL[i] != 0) s12[i]++ = i;
  // lsb radix sort the mod 1 and mod 2 triples
  radixPass(s12, SAIL, s, no, 2, K);
  radixPass(SAIL, s12, s, no, 2, K);
  radixPass(s12, SAIL, s, no, 2, K);
}
```
Things to remember

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