Applications of suffix trees and suffix arrays
Generalized suffix tree

• Given a set of strings $S = \{S_1, \ldots, S_z\}$, we can build a generalized suffix tree for these strings.

• To associate each suffix with a unique string in $S$, a distinct symbol is appended to each string $s$ in $S$.

• Concatenate the resulting words and build a suffix tree for it.
  • $S_1 =$ basa; $S_2 =$ abas and $S_3 =$ sa$
  • basa@abas#sa$
Generalized suffix tree

Problem
• The resulting tree has some synthetic suffix that span more than one original word

Solution
• The label of each path from the root to an internal node corresponds to a substring of the original string.
• we can replace the second index of the leaf edges with the end of the corresponding string
Let $s_1=abab$ and $s_2=aab$ here is a generalized suffix tree for $s_1$ and $s_2$
So what can we do with it?
Matching a pattern against a database of strings

• **Input**: a pattern $P$ and a set of strings $S$

• **Output**: Find every occurrence of the pattern $P$ in the database $S$
Application: longest common substring problem (LCS)

• **Input**: two strings s and s’
• **Output**: the longest substring that occurs both in s and s’

Example: s=casamento and s’=mensagem
LCS(s,s’)=men
Key Observation: every node that has as descendants both a leaf from string $s$ and a leaf from string $s'$ represents a common substring and vice versa.

Application: longest common substring problem (LCS)
Algorithm

1. Build a generalized suffix tree for $s$ and $s'$

2. Do a DFS to mark the nodes that have descendants from both strings; compute the string depths

3. Find the marked node with largest “string depth”

Application: longest common substring problem (LCS)
Application: longest common substring problem (LCS)

Time analysis

1. Build a generalized suffix tree for s and s’ \( O(m+n) \)

2. Do a DFS to mark the nodes that have descendants from both strings; compute the string depths \( O(m+n) \)

3. Find the marked node with largest “string depth” \( O(m+n) \)
The Lowest Common Ancestor

- In a rooted tree $T$, a node $u$ is an ancestor of a node $v$ if $u$ is on the unique path from the root to $v$.

- In a rooted tree $T$, the Lowest Common Ancestor (LCA) of two nodes $u$ and $v$ is the deepest node in $T$ that is the ancestor of both $u$ and $v$. 
• Node 3 is the LCA of nodes 4 and 6.
• Node 1 is the LCA of node 2 and 5.
Theorem. Given a rooted tree $T$ with $n$ nodes, there is a data structure of $O(n)$ space that supports LCA queries in constant time. Furthermore, this data structure can be built in $O(n)$ time.

Later in the course
Lowest common ancestors

A lot more can be gained from the suffix tree if we preprocess it so that we can answer LCA queries on it.
Why?

The LCA of two leaves represents the longest common prefix (LCP) of these 2 suffixes.
Finding Maximal Palindromes

Definition. A palindrome is a string that coincides with its reverse

- Palindromes: cabaabac, ctcfgfctc
- Palindromes and other repeated structures play an important role in computational biology
Finding Maximal Palindromes

**Definition.** A substring $s'$ is a maximal palindrome of $s$ if: (i) $s'$ is a substring of $s$; (ii) $s'$ is a palindrome and (iii) $s'$ is maximal, that is, if we extend $s'$ by one character in each direction we do not have a palindrome anymore.

- *aba* and *abaaba* are maximal palindromes of *cabaabada*. 

- *aba* and *abaaba* are maximal palindromes of *cabaabada*.
Finding Maximal Palindromes

- **Input**: a string $S$
- **Output**: all maximal palindromes of $S$
Maximal palindromes algorithm

**Key observation** The maximal palindrome with center between \( i-1 \) and \( i \) is the LCP of the suffix at position \( i \) of \( s \) and the suffix at position \( m-i+1 \) of \( s^r \)

Example

\( s = cbaaba \) and \( s^r = abaabc \)

\( s = cbaaba \) and \( s^r = abaabc \)
Maximal palindromes algorithm

1. Prepare a generalized suffix tree for $s$ and $s^r$

2. Preprocess a structure for solving LCA's

3. For every $i$ find the LCA of suffix $i$ of $s$ and suffix $m-i+1$ of $s^r$
   
   • $O(n)$ time to identify all maximal palindromes
Maximal palindromes algorithm

Let $s = cbaaba$ then $s^r = abaabc#$
Exact Matching with wild cards

• A wild card is a character * that matches every character
• **Input**: two strings P and T, with at most k wild cards distributed along them
• **Output**: all occurrences of P in T

Example
P=a*va*cf    T= ndaf*aacfp
Exact Matching with wild cards

• A wild card is a character * that matches every character

• **Input**: two strings P and T, with at most k wild cards distributed along them

• **Output**: all occurrences of P in T

Example

P=a*va*cf  T= ndaf*aacfp
Exact Matching with wild cards

Algorithm High-Level

1. Build a generalized suffix tree for both strings. The symbol * is considered as an extra char.

2. preprocess the longest common ancestors and calculate the string depths for the resulting tree.

3. For each position i of the text T verify whether P and T[i,…,i+|P|-1] matches through LCA queries
Exact Matching with wild cards

Step 3 (Detailed)

3.1 $j \leftarrow 1$ and $k \leftarrow i$

3.2 $len \leftarrow$ Longest Common Prefix between $P[j]$ and $T[k]$

3.3 If $j + len = n+1$ then $P$ occur at $T$ starting at $i$

3.4 Else Check if a wild card occurs in position $j+len$ of $P$ or position $k+len$ of $T$.

   If positive $j \leftarrow j+len+1$ and $k \leftarrow k+len+1$

   goto 3.2

   Else $P$ does not match $T[i,\ldots,i+n-1]$
Exact Matching with wild cards

Time Analysis

- Step 1 takes $O(|P| + |T|)$ time
- Step 2 spends $O(|T|k)$
  - The verification for each $i$ executes up to $k$ LCA queries and each of them costs constant time
The k-mismatch problem

- **Input**: two strings P and T and an integer k
- **Output**: all occurrences of P in T allowing at most k mismatches

Example

P=bend and T=abentbananaend, k=2
3 matches: bent, bana, aend
The \( k \)-mismatch problem

Algorithm High-Level

1. Build a generalized suffix tree for both strings.
2. Preprocess the longest common ancestors and calculate the string depths for the resulting tree.
3. For each position \( i \) of the text \( T \) verify whether \( P \) and \( T[i,...,i+|P|-1] \) matches through LCA queries
The k-mismatch problem

Step 2: Detailed

2.1 Compute the length len of the LCA between T[i,…,i+|P|-1] and P.

2.2 If len=|P| then there is a match starting at i. Otherwise,

- If k>0 try to match recursively P[1+len+1,...,|P|] and T[i+len+1,... i+|P|-1] allowing at most k-1 matches
- If k=0 they do not match
The k-mismatch problem

Step 3: Detailed

1. \( j \leftarrow 1; \ i' \leftarrow i \) and \( \text{count} \leftarrow 0 \)
2. \( \text{len} \leftarrow \text{LCP} \) between \((P,j)\) and \((T,i')\)
3. If \( j+\text{len}=n+1 \) then a k-mismatch occurs in \( T \)
4. If \( \text{count} \leq k \)
   \( \quad \text{count}++; \ j \leftarrow j+\text{len}+1; \ i' \leftarrow i'+\text{len}+1 \)
   \( \text{Goto 2} \)
   **Else** a k-mismatch does not occur
The k-mismatch problem

Time Analysis

- Step 1 takes $O(|P|+|T|)$ time
- Step 2 spends $O(|T|k)$
  - The verification for each $i$ executes up to $k$ LCA queries and each of them costs constant time
Drawbacks

• Suffix trees consume a lot of space
  – It consumes $O(n)$ space but the constant is quite big

• Notice that if we indeed want to traverse an edge in $O(1)$ time then we need an array of pointers of size $|\Sigma|$ in each node
**Suffix array**

**Definition.** A suffix array for a text $T$ is an array $SA$ that specifies the lexicographical order of the suffixes of $T$.

<table>
<thead>
<tr>
<th>Index</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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>$$</td>
</tr>
<tr>
<td>Suffix Array</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>
**Suffix array**

**Note** $\$ is also appended to the end of the string and interpreted as a character lexicographically smaller than all others

<table>
<thead>
<tr>
<th>Index</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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>$</td>
</tr>
<tr>
<td>Suffix Array</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>
How do we build it?

- Build a suffix tree
- Traverse the tree in a DFS, lexicographically picking edges outgoing from each node
- The suffix array order is given by the order in which the leaves are visited
- Extra $O(n)$ time from the suffix tree
How do we build it?

Example
How do we search for a pattern?

• If P occurs in T then all its occurrences are consecutive in the suffix array.

• Do a binary search on the suffix array

• Takes $O(m \log n)$ time
Example

Let $S = \text{mississippi}$

Let $P = \text{issa}$
Example

Let $S = \text{mississippi}$

Let $P = \text{issa}$
How do we accelerate the search?

**Definition** The LCP between i and j is the length of the longest common prefix between the i-th and the j-th smallest suffixes (SA[i] and SA[j]).

<table>
<thead>
<tr>
<th>Index</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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>$</td>
</tr>
<tr>
<td>Suffix Array</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

- $LCP(3,4) = 4$ ; $LCP(2,6) = 1$
How do we accelerate the search?

**Definition** The LCP between \(i\) and \(j\) is the length of the longest common prefix between the \(i\)-th and the \(j\)-th smallest suffixes (\(SA[i]\) and \(SA[j]\))

- LCP’s are used to accelerate the search for the pattern \(P\) in the suffix array
  - \(LCP(L,P)\) and \(LCP(R,P)\) are maintained during the search
  - \(LCP(L,M)\) and \(LCP(M,R)\) are precalculated
How do we accelerate the search?

\[ \ell = \text{LCP}(P,L); \ r = \text{LCP}(P,R) \]

Case 1) \( \ell = r \)

then start comparing \( M \) with \( P \) at position \( \ell + 1 \). If a mismatch occurs at position \( j \) of \( P \):

- \( P[j] \) is lexicog. smaller:
  \[ R \leftarrow M; \ r \leftarrow j-1 \]
- \( P[j] \) is lexicog. larger:
  \[ L \leftarrow M; \ \ell \leftarrow j-1 \]
How do we accelerate the search?

Case 2: $\ell > r$ (w.l.o.g)

**Case 2.1** $\text{LCP}(L, M) < \ell$

we go left; $R \leftarrow M$

**Case 2.2** $\text{LCP}(L, M) > \ell$

we go right; $L \leftarrow M$

**Case 2.3** $\text{LCP}(L, M) = \ell$

we start comparing $P$ and $M$

at position $\ell + 1$ of $P$
How do we accelerate the search?

Assume $\ell > r$ (w.l.o.g)

**Case 2.3** $\text{LCP}(L,M) = \ell$

- we start comparing $P$ and $M$ at position $\ell + 1$ of $P$
- If there is no mismatch we report a match
- If there is a mismatch at position $j$ of $P$ we either go left or right and update $L/ \ell$ or $M/r$ accordingly
Analysis of the acceleration

Theorem The search with LCP’s takes \( O(\log n + m) \) time

Proof. If we do more than a single comparison in an iteration then \( \max(\ell, r) \) grows by 1 for each comparison \( \Rightarrow \)

\[ O(\log n + m) \] time
Computing the LCP’s

Key Observations

- We just need to precalculate the LCP’s for the extremes (L,R) of the binary search $\Rightarrow O(n)$ pairs

```
1,1 1,2 1,4 1,8
1,2
1,2 1,4 2,3 2,4 4,5 4,6 4,8
4,6
4,6 5,6 6,7 6,8
6,7
6,7 7,8
7,8
```
Computing the LCP’s

Key Observations

• $LCP(i, i+1)$ is the string depth of the Lowest Common Ancestor of the $i$-th leaf and $(i+1)$-th leaf of the suffix tree
  
  • It can be calculated through a DFS over the suffix tree
Computing the LCP’s

**Lemma** The LCP(i,j) is the smallest value of LCP(k,k+1) where k ranges from i to j-1

**Proof.**

Let $k^*$ be the index of the minimum LCP(k,k+1) where k ranges from i to j.

Let $\beta$ be the longest common prefix between $SA[k^*]$ and $SA[k^*+1]$. It follows that $\beta$ is common to $SA[i],SA[i+1],...,SA[j]$
Computing the LCP’s

We can compute the LCP’s of the nodes in a bottom-up fashion

- LCP of a node = min between LCP of its children
Efficient Construction of Suffix Arrays

• Many Available algorithms (Survey by Puglisi, Smyth and Turpin 2006)
  – Comparative table in the next page
## Efficient Construction of Suffix Arrays

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst Case</th>
<th>Speed</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prefix-Doubling</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MM [MM93]</td>
<td>$O(n \log n)$</td>
<td>16</td>
<td>$8n$</td>
</tr>
<tr>
<td>LS [LS99]</td>
<td>$O(n \log n)$</td>
<td>1.7</td>
<td>$8n$</td>
</tr>
<tr>
<td><strong>Recursive</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KA [KA03]</td>
<td>$O(n)$</td>
<td>2.2</td>
<td>13-14n</td>
</tr>
<tr>
<td>KS [KS03]</td>
<td>$O(n)$</td>
<td>2.8</td>
<td>10-13n</td>
</tr>
<tr>
<td>KSPP [KSPP03]</td>
<td>$O(n)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSS [HSS03]</td>
<td>$O(n)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KJP [KJP04]</td>
<td>$O(n \log \log n)$</td>
<td>2.1</td>
<td>13-16n</td>
</tr>
<tr>
<td><strong>Induced Copying</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT [IT99]</td>
<td>$O(n^2 \log n)$</td>
<td>4</td>
<td>$5n$</td>
</tr>
<tr>
<td>S [S00]</td>
<td>$O(n^2 \log n)$</td>
<td>2.1</td>
<td>$5n$</td>
</tr>
<tr>
<td>BK [BK03]</td>
<td>$O(n \log n)$</td>
<td>2.1</td>
<td>5-6n</td>
</tr>
<tr>
<td>MF [MF04]</td>
<td>$O(n^2 \log n)$</td>
<td>1</td>
<td>$5n$</td>
</tr>
<tr>
<td>SS [SS05]</td>
<td>$O(n^2)$</td>
<td>1</td>
<td>9-10n</td>
</tr>
<tr>
<td>M [M05]</td>
<td>$O(n^2 \log n)$</td>
<td>1</td>
<td>5-7n</td>
</tr>
<tr>
<td><strong>Suffix Tree</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K [K99]</td>
<td>$O(n \log \sigma)$</td>
<td>4</td>
<td>15-20n</td>
</tr>
</tbody>
</table>
Efficient Construction of Suffix Arrays

- h-order of the suffixes ($SA_h$)
  The suffixes are ordered according to its h first symbols

<table>
<thead>
<tr>
<th>Index</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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>$</td>
</tr>
<tr>
<td>Suffix Array (h=1)</td>
<td>12</td>
<td>(1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>11)</td>
<td>(2</td>
<td>9)</td>
<td>5</td>
<td>7</td>
<td>(3</td>
<td>10)</td>
</tr>
</tbody>
</table>
**Efficient Construction of Suffix Arrays**

- **h-order of the suffixes** \((SA_h)\)
  - The suffixes are ordered according to its \(h\) first symbols

<table>
<thead>
<tr>
<th>Index</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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T)</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>$</td>
</tr>
<tr>
<td>Suffix Array ((h=1))</td>
<td>12</td>
<td>11</td>
<td>(1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>(2</td>
<td>9</td>
<td>5</td>
<td>7</td>
<td>(3</td>
</tr>
<tr>
<td>Suffix Array ((h=3))</td>
<td>12</td>
<td>11</td>
<td>(1</td>
<td>8</td>
<td>4</td>
<td>6</td>
<td>(2</td>
<td>9</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>
Efficient Construction of Suffix Arrays

- Prefix Doubling Algorithms (Manber and Myers 1990)

**Key observation:** If $SA_h$ is scanned left to right (in $h$-order), then, for $j=1 \ldots n$, the suffixes

$$SA_h[j] - h > 0$$

are scanned in $2h$-order within their respective $h$-groups in $SA_h$
Efficient Construction of Suffix Arrays

**Key observation:** If $SA_h$ is scanned left to right (in $h$-order), then, for $j=1\ldots n$, the suffixes $SA_h[j] - h$ are scanned in $2h$-order within their respective $h$-groups in $SA_h$. 
Efficient Construction of Suffix Arrays

If $SA_h[j]$ and $SA_h[j']$ belong to the same $h$-group then the 2$h$-order of $SA_h[j]$ is not smaller than that of $SA_h[j']$.
Efficient Construction of Suffix Arrays

- $ISA_h(i)$: indicates the location of the beginning of the $h$-group where the $i$-th suffix lies
- It is useful to obtain $SA_{2h}$ form $SA_h$

<table>
<thead>
<tr>
<th>Index</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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>$$</td>
</tr>
<tr>
<td>Suffix Array (h=1)</td>
<td>12</td>
<td>(1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>11)</td>
<td>(2</td>
<td>9)</td>
<td>5</td>
<td>7</td>
<td>(3</td>
<td>10)</td>
</tr>
<tr>
<td>Inverse Suffix Array (h=1)</td>
<td>2</td>
<td>7</td>
<td>11</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>10</td>
<td>2</td>
<td>7</td>
<td>11</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Efficient Construction of Suffix Arrays

<table>
<thead>
<tr>
<th>Index</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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>$</td>
</tr>
<tr>
<td>Suffix Array (h=1)</td>
<td>12</td>
<td>(1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>11)</td>
<td>(2</td>
<td>9)</td>
<td>5</td>
<td>7</td>
<td>(3</td>
<td>10)</td>
</tr>
<tr>
<td>Inverse Suffix Array (h=1)</td>
<td>2</td>
<td>7</td>
<td>11</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>10</td>
<td>2</td>
<td>7</td>
<td>11</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Head</td>
<td>1</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>7</td>
<td>-1</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>-1</td>
</tr>
</tbody>
</table>

- The array **Head** keeps track of the first available position in each h-group
Efficient Construction of Suffix Arrays

Execute a BucketSort to obtain $SA_1$
Obtain $ISA_1$ and Head from $SA_1$

While some $h$-group is not a singleton
  1. Construct $SA_{2h}$ from $SA_h$ and $ISA_h$ and Head using the key observation
  2. Obtain $ISA_{2h}$ from $SA_{2h}$
  3. Update $2h$-groups
  4. $h \leftarrow 2h$

End While
Efficient Construction of Suffix Arrays

Observations

• The external loop executes at most $\log n$ iterations

• Inside one iteration only the suffixes that belong to the same $h$-group have their positions switched
Efficient Construction of Suffix Arrays

• Obtaining $SA_{2h}$ from $SA_h$

\begin{align*}
\text{For } j=1\ldots n \\
i &\leftarrow SA_h[j] - h \\
\text{If } i>0 \text{ then} \\
q &\leftarrow \text{Head}[ISA_h[i]] \\
SA_{2h}[q] &\leftarrow i \\
\text{Head}[ISA_h[i]] &\leftarrow +1 \\
\text{End}
\end{align*}
Efficient Construction of
Suffix Arrays

- MM algorithm (after first iteration)

<table>
<thead>
<tr>
<th>Index</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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>$</td>
</tr>
<tr>
<td>Suffix Array (h=1)</td>
<td>12</td>
<td>(1</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>(2</td>
<td>9</td>
<td>5</td>
<td>7</td>
<td>(3</td>
<td>10)</td>
</tr>
<tr>
<td>Inverse Suffix Array (h=1)</td>
<td>2</td>
<td>7</td>
<td>11</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>10</td>
<td>2</td>
<td>7</td>
<td>11</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Head</td>
<td>1</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>7</td>
<td>-1</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>-1</td>
</tr>
</tbody>
</table>
Efficient Construction of Suffix Arrays

- MM algorithm (after second iteration)

<table>
<thead>
<tr>
<th>Index</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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>$</td>
</tr>
<tr>
<td>Suffix Array (h=2)</td>
<td>12</td>
<td>11</td>
<td>(1)</td>
<td>8</td>
<td>4</td>
<td>6</td>
<td>(2)</td>
<td>9</td>
<td>5</td>
<td>7</td>
<td>(3)</td>
<td>10</td>
</tr>
<tr>
<td>Inverse Suffix Array (h=2)</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>5</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Head</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>-1</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>-1</td>
</tr>
</tbody>
</table>
Efficient Construction of Suffix Arrays

- MM algorithm (after third iteration)

<table>
<thead>
<tr>
<th>Index</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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>a</td>
<td>$</td>
</tr>
<tr>
<td>Suffix Array (h=4)</td>
<td>12</td>
<td>11</td>
<td>(1</td>
<td>8)</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>Inverse Suffix Array (h=4)</td>
<td>3</td>
<td>8</td>
<td>12</td>
<td>5</td>
<td>9</td>
<td>6</td>
<td>10</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Head</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
Efficient Construction of Suffix Arrays

Time Complexity

- At most log n iterations
- Each loop runs in O(n) time
- Overall complexity O(n log n)