Text Indexing: Lecture 7
Simon Gog – gog@kit.edu
Problem 2
Given a \([0, n-1] \times [0, n-1]\) grid \(G\) and a set \(P\) of \(n\) points \((i, S[i])\) with weight \(w[i]\) for \(0 \leq i < n\). For a pair of points \((x_0, y_0)\) and \((x_1, y_1)\) with \(x_0 \leq x_1\) and \(y_0 \leq y_1\) we define the top-\(k\) range query:

- A top-\(k\)-range report query asks for the \(k\) points \((x, y) \in P\) such that \(x \in [x_0, x_1]\) and \(y \in [y_0, y_1]\) with maximum weight sorted in decreasing order of weight.

Results [5]
Top-\(k\) range queries can be answered in \(O(\log^2 n + k \cdot \log n)\) time using a index of size \(3n \log n + o(n \log n) + |weights|\) bits, where \(|weights|\) is the space required to store the weights associated with the \(n\) points.
Outline of solution

- Use the range count algorithm to get the $O(\log n)$ WT nodes $C$ which cover the $y$ range
- For each $x$ range in $C$ use the RMQ structure to navigate to the heaviest point ($O(\log n)$ time per range, i.e $O(\log^2 n)$ total)
- Insert the heaviest points into a max priority queue $Q$
- Remove maximum point from $Q$, report it and split its corresponding $x$ range in $C$. Navigate to the heaviest points in the two new ranges $Q$. Insert
- Repeat last step until $Q$ is empty or $k$ points were reported

Total time: $O(\log^2 n + k \cdot \log n)$
Top-\(k\) Range Report Queries

\[ y \text{ range} \]

- \(x\) ranges
- \(\text{RMQ on associated weights}\)
Top-$k$ Range Report Queries

$x$ ranges

RMQ on associated weights

$y$ range
Top-$k$ Range Report Queries

$x$ ranges

$RMQ$ on associated weights

$y$ range
The suffix tree (ST) extends the functionality of suffix array construction in three phases (each takes linear time):

- Suffix array construction (Timo Bingmann will present this at the 21st of January 2016)
- LCP array construction
- Tree topology construction

As the pointer-based representation takes too much space in most application we will present a more space-efficient version: The compressed suffix tree (CST).
Versatile index structure: E.g. we can solve longest common substring queries for $k$ strings efficiently.

Exercise: Find longest palindromic substring of a string.
Most representations consist of three parts:
- Suffix Array (leaves of suffix tree)
- LCP Array (longest common prefix lengths/depth of inner nodes)
- Tree Topology
Definition

Let \( lcp(U, V) \) denote the longest common prefix between two strings \( U \) and \( V \). For a text \( T \) of size \( n \) the longest common prefix (LCP) array of size \( n + 1 \) is defined as follows. \( LCP[i] = |lcp(SA[i], SA[i - 1])| \) for \( i \geq 1 \) and \( LCP[0] = 0 \) and \( LCP[n] = -1 \).

([4] introduced this array as \( Hgt \) array)
### LCP Array – Example

<table>
<thead>
<tr>
<th>$i$</th>
<th>SA</th>
<th>$LCP$</th>
<th>$T[SA[i],n-1]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>0</td>
<td>$$</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
<td>i$</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>1</td>
<td>ippi$</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>issippi$</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>ississippi$</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>mississippi$</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>0</td>
<td>pi$</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>1</td>
<td>ppi$</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>0</td>
<td>sippi$</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>2</td>
<td>sissippi$</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>1</td>
<td>sissippi$</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>3</td>
<td>ssissippi$</td>
</tr>
</tbody>
</table>

- **Time complexity of naive computation** (for each $i > 0$, compare suffix $SA[i]$ and $SA[i-1]$): $O(n^2)$.
- **Comparison in SA-order.**
Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
m & i & s & s & i & s & s & i & p & p & i & \$
\end{array}
\]

\[
i = 0 \\
SA[\text{ISA}[i] - 1] = 1 \\
\]

0
Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.

\[
\begin{align*}
0 & \quad 1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 & \quad 6 & \quad 7 & \quad 8 & \quad 9 & \quad 10 & \quad 11 \\
\text{m} & \quad i & \quad s & \quad s & \quad i & \quad s & \quad s & \quad i & \quad p & \quad p & \quad i & \quad \$
\end{align*}
\]

\[
i = 1
\]

\[
\text{SA}[\text{ISA}[i] - 1] = 4
\]

0 \quad 4
Idea of [3]: Processing in text-order.

\[
i = 2
\]

\[
SA[ISA[i] - 1] = 5
\]
Idea of [3]: Processing in text-order.

$$i = 3$$

$$SA[ISA[i] - 1] = 6$$
Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.

\[ i = 4 \]

\[ \text{SA[ISA}[i] - 1] = 7 \]
Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.

In the text "mississippi", we have:

- i = 5

Steps:

1. Initialize SA and ISA arrays.
2. Calculate LCP values.
3. The text ordering is maintained.

Example:

```
0 1 2 3 4 5 6 7 8 9 10 11
m i s s i s s i p p i $
```


```
0 4 3 2 1 1
```
Idea of [3]: Processing in text-order.

\[ i = 6 \]

\[ \text{SA}[	ext{ISA}[i] - 1] = 8 \]
Idea of [3]: Processing in text-order.

\[ i = 7 \]

\[ \text{SA}[\text{ISA}[i] - 1] = 10 \]
Idea of [3]: Processing in text-order.

\[
\begin{array}{ccccccccccl}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\text{m} & \text{i} & \text{s} & \text{s} & \text{i} & \text{s} & \text{s} & \text{i} & \text{p} & \text{p} & \text{p} & \$
\end{array}
\]

\[
i = 8
\]

\[
\text{SA}[\text{ISA}[i] - 1] = 9
\]

0 4 3 2 1 1 0 1 1
Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.

\[ \text{SA}[\text{ISA}[i] - 1] = 0 \]

Index: \( i = 9 \)

\[ \begin{array}{cccccccccccc}
  & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\text{mississippi} & m & i & s & s & i & s & s & i & p & p & i & $ \\
\end{array} \]
Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.

<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>
<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>
<td>11</td>
</tr>
</tbody>
</table>

mississippi $pipping$

$SA[ISA[i] - 1] = 11$

$i = 10$

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 3 | 2 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
Idea of [3]: Processing in text-order.

Let's consider the text: *mississippi*

<table>
<thead>
<tr>
<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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
</tr>
</tbody>
</table>

$SA[ISA[i] - 1] = 5$

For $i = 11$:


<table>
<thead>
<tr>
<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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Lemma ([3])
For $0 < i \leq n$, we have $LCP[ISA[i]] \geq LCP[ISA[i - 1]] - 1$. 
Linear Time Calculation of LCP Array

00 \(LCP[0] \leftarrow 0\)
01 \(LCP[n] \leftarrow -1\)
02 \(\text{for } i \leftarrow 0 \text{ to } n - 1 \text{ do}\)
03 \(ISA[SA[i]] \leftarrow i\)
04 \(\ell \leftarrow 0\)
05 \(\text{for } i \leftarrow 0 \text{ to } n - 1 \text{ do}\)
06 \(j \leftarrow SA[(ISA[i] - 1) \mod n]\)
07 \(\text{while } T[i + \ell] = T[j + \ell] \text{ do}\)
08 \(\ell \leftarrow \ell + 1\)
09 \(LCP[ISA[i]] \leftarrow \ell\)
10 \(\ell \leftarrow \max(0, \ell - 1)\)

Exercise

How much memory is required during the algorithms execution?
Linear Time Calculation of LCP Array

Engineered version of [2]:

```java
for i ← 0 to n − 1 do
    \Phi[SA[i]] ← SA[(i − 1) mod n]
end for

ℓ ← 0

for i ← 0 to n − 1 do
    j ← \Phi[i]
    while T[i + ℓ] = T[j + ℓ] do
        ℓ ← ℓ + 1
    end while
    PLCP[i] ← ℓ
    ℓ ← max(0, ℓ − 1)
end for

for i ← 0 to n − 1 do
    LCP[i] ← PLCP[SA[i]]
end for

LCP[n] ← −1
```

(Explain why this algorithm is faster in practice)
[6] proposed to use a CSA and the permuted LCP array (LCP values in text-order) PLCP. We know

- $PLCP[i + 1] \geq PLCP[i] - 1$
- $PCLP[i] \leq n - 1 - i$ (for $0 \leq i < n$)

\[
\begin{align*}
    i & = 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \\
    PCLP[i] & = 0 \quad 4 \quad 3 \quad 2 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0
\end{align*}
\]
[6] proposed to use a CSA and the permuted LCP array (LCP values in text-order) PLCP. We know

- \( PLCP[i + 1] \geq PLCP[i] - 1 \)
- \( PCLP[i] \leq n - 1 - i \) (for \( 0 \leq i < n \))

\[
\begin{align*}
  i &= \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \\
  PCLP[i] &= \quad 0 \quad 4 \quad 3 \quad 2 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \\
  PCLP[i] + i &= \quad 0 \quad 5 \quad 5 \quad 5 \quad 5 \quad 6 \quad 6 \quad 8 \quad 9 \quad 9 \quad 10 \quad 11
\end{align*}
\]
Space-Efficient LCP Representation

[6] proposed to use a CSA and the permuted LCP array (LCP values in text-order) PLCP. We know

- $PLCP[i + 1] \geq PLCP[i] - 1$
- $PCLP[i] \leq n - 1 - i$ (for $0 \leq i < n$)

\[
i = 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11
\]
\[
PCLP[i] = \quad 0 \quad 4 \quad 3 \quad 2 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0
\]
\[
PCLP[i] + i = \quad 0 \quad 5 \quad 5 \quad 5 \quad 5 \quad 6 \quad 6 \quad 8 \quad 9 \quad 9 \quad 10 \quad 11
\]

- Encode gaps of $PLCP[i] + i$ with unary code (results in bitvector $H$ of length $2n$)
- In this example: $H = 10000011110110010110101$
- What additional structure is required to calculated $LCP[i]$?
Space-Efficient LCP Representation

With a $o(n)$-space select structure (arguments starting from 1) and a CSA we get:

```plaintext
00  access_lcp(i)
01  x ← SA[i]
02  return select(x + 1, 1, H) + 1 − 2x
```

Summary:
- Time complexity depends on CSA access
- Space: $2n + o(n)$ bits (for bitvector $H + select$) + $|CSA|$ 

Note: The LCP between arbitrary suffixes can be calculated in constant time using a RMQ structure.
The LCP-Interval Tree

Definition of an LCP-interval ([1])

An interval \([i, j]\), where \(0 \leq i \leq n - 1\) is called LCP-interval of LCP value \(\ell\) (denoted by \(\ell - [i, j]\)) if

- \(LCP[i] < \ell\) or \(i = 0\)
- \(LCP[k] \geq \ell\) for all \(k \in [i + 1, j]\)
- \(LCP[k] = \ell\) for at least one \(k \in [i + 1, j]\)
- \(LCP[j + 1] < \ell\)

Every index \(k\) with \(i < k \leq j\) and \(LCP[k] = \ell\) is called \(\ell\)-index. There are at most \(\sigma - 1\) \(\ell\)-indices in an LCP-interval.

Note: Each LCP-interval corresponds to a node in the suffix tree.
The LCP-Interval Tree – Example

\begin{tabular}{|l|l|l|l|}
\hline
\textbf{$i$} & \textbf{SA} & \textbf{LCP} & \textbf{$T[SA[i], n-1]$} \\
\hline
0 & 11 & 0 & $\$ \\
1 & 10 & 0 & i$ \\
2 & 7 & 1 & ippi$ \\
3 & 4 & 1 & issippi$ \\
4 & 1 & 4 & ississippi$ \\
5 & 0 & 0 & mississippi$ \\
6 & 9 & 0 & pi$ \\
7 & 8 & 1 & ppi$ \\
8 & 6 & 0 & sippi$ \\
9 & 3 & 2 & sississippi$ \\
10 & 5 & 1 & ssippi$ \\
11 & 2 & 3 & ssissippi$ \\
\hline
\end{tabular}

- Singleton intervals $\ell - [i, i]$ are omitted
Bibliography


