Recap: Pattern Matching with the LCP-Array (1/3)

- remember how many characters of the pattern and suffix match
- identify how long the prefix of the old and next suffix is
- do so using the LCP-array and
- range minimum queries detailed introduction in Advanced Data Structures

Definition: Range Minimum Queries

Given an array $A[1..m]$, a **range minimum query** for a range $\ell \leq r \in [1, n)$ returns

$$RMQ_A(\ell, r) = \arg \min \{ A[k] : k \in [\ell, r] \}$$

- $lcp(i, j) = \max \{ k : T[i..i+k) \}$
- $lcp(i, j) = T[j..j+k) = LCP[RMQ_{LCP}(i+1, j)]$
- RMQs can be answered in $O(1)$ time and
- require $O(n)$ space
Recap: Pattern Matching with the LCP-Array (2/3)

- during binary search matched
- \( \lambda \) characters with left border \( \ell \) and
- \( \rho \) characters with right border \( r \)
- w.l.o.g. let \( \lambda \geq \rho \)

- middle position \( i \)
- decide if continue in \([\ell, i]\) or \([i, r]\)

- let \( \xi = \text{lcp}(SA[\ell], SA[i]) \) \( \in O(1) \) time with RMQs
Recap: Pattern Matching with the LCP-Array (3/3)

- Let $\xi = \text{lcp}(SA[\ell], SA[i])$

$\xi > \lambda$
- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison

$\xi = \lambda$
- Compare as before

$\xi < \lambda$
- $\xi \geq \rho$ and $P[\xi + 1] < T[SA[i] + \xi]$
- $r = i$ and $\rho = \xi$ without character comparison
Definition: Longest Common Extensions

Given a text $T$ of size $n$ over an alphabet of size $\sigma$, construct data structure that answers for $i, j \in [1, n]$

$$\text{lce}_T(i, j) = \max\{\ell \geq 0 : T[i, i+\ell] = T[j, j+\ell]\}$$

also denoted as $lcp(i, j)$ in this lecture
**Old Problem, New Name**

**Definition: Longest Common Extensions**

Given a text $T$ of size $n$ over an alphabet of size $\sigma$, construct data structure that answers for $i, j \in [1, n]$

$$\text{lce}_T(i, j) = \max\{\ell \geq 0 : T[i, i + \ell] = T[j, j + \ell]\}$$

also denoted as $lcp(i, j)$ in this lecture.

---

**Example**

<table>
<thead>
<tr>
<th>$T$</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>$0$</th>
<th>$1$</th>
<th>$2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>C</td>
<td>D</td>
<td>B</td>
<td>C</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>D</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

$$\text{lce}_T(1, 14) = 0 1 2 3 4 5$$
Old Problem, New Name

Definition: Longest Common Extensions

Given a text $T$ of size $n$ over an alphabet of size $\sigma$, construct data structure that answers for $i, j \in [1, n]$

$$lce_T(i, j) = \max \{ \ell \geq 0 : T[i, i + \ell) = T[j, j + \ell) \}$$

also denoted as $lcp(i, j)$ in this lecture

Applications

- (sparse) suffix sorting
- approximate pattern matching
- …

![Diagram showing text $T$ and its longest common extensions]

$$lce_T(1, 14) = 0 1 2 3 4 5$$
Sophisticated Black Box (BB)

- based on ISA, LCP, and RMQ

Black Box

- $O(1)$ query time, $\approx 9n$ bytes additional space
Sophisticated Black Box (BB)
- based on ISA, LCP, and RMQ
- \(O(1)\) query time, \(\approx 9n\) bytes additional space

Ultra Naive Scan (UNS)
- compare character by character
- \(O(n)\) query time, no additional space
Sophisticated Black Box (BB)
- based on ISA, LCP, and RMQ
- $O(1)$ query time, $\approx 9n$ bytes additional space

Ultra Naive Scan (UNS)
- compare character by character
- $O(n)$ query time, no additional space

Practical Algorithms for Longest Common Extensions [IT09]
Space vs. Query Time graph showing a better trade-off for BB compared to UNS.
Monte Carlo and Las Vegas Algorithms

- setting: randomized algorithms

Monte Carlo Algorithm
- returns wrong result with small probability
- deterministic running time

Las Vegas Algorithm
- returns correct result only expected running time

Some Monte Carlo algorithms can be turned into Las Vegas algorithms depending on correctness check. All Monte Carlo algorithms presented today can be turned into Las Vegas algorithms.
Monte Carlo and Las Vegas Algorithms

- setting: randomized algorithms

**Monte Carlo Algorithm**
- returns wrong result with small probability
- deterministic running time

**Las Vegas Algorithm**
- returns correct result
- only expected running time
Monte Carlo and Las Vegas Algorithms

- setting: randomized algorithms

Monte Carlo Algorithm
- returns wrong result with small probability
- deterministic running time

Las Vegas Algorithm
- returns correct result
- only expected running time

- some Monte Carlo algorithms can be turned into Las Vegas algorithms
- depends on correctness check
- all Monte Carlo algorithms presented today can be turned into Las Vegas algorithms
Randomized String Matching

- compute fingerprints of strings
- application not limited to LCEs
Randomized String Matching

- compute fingerprints of strings
- application not limited to LCEs

**Definition: Karp-Rabin Fingerprint [KR87]**

Given a text $T$ of length $n$ over an alphabet of size $\sigma$ and a random prime number $q \in \Theta(n^c)$, the Karp-Rabin fingerprint of $T[i..j]$ is

$$\mathcal{F}(i, j) = \left( \sum_{k=i}^{j} T[k] \cdot \sigma^{j-k} \right) \mod q$$

**Example:**

- $(x + y) \mod z = z \mod z + y \mod z \mod z$
Randomized String Matching

- compute hashes of strings
- application not limited to LCEs

Definition: Karp-Rabin Fingerprint [KR87]

Given a text $T$ of length $n$ over an alphabet of size $\sigma$ and a random prime number $q \in \Theta(n^c)$, the Karp-Rabin fingerprint of $T[i..j]$ is

$$\Phi(i, j) = \left(\sum_{k=i}^{j} T[k] \cdot \sigma^{j-k}\right) \mod q$$

if $T[i..i + \ell] = T[j..j + \ell]$, then

$$\Phi(i, i + \ell) = \Phi(j, j + \ell)$$

if $T[i..i + \ell] \neq T[j..j + \ell]$, then

$$\text{Prob}(\Phi(i, i + \ell) = \Phi(j, j + \ell)) \in O\left(\frac{\ell \log \sigma}{n^c}\right)$$

- prime has to be chosen uniformly at random
- how to turn it into Las Vegas algorithm?
Randomized String Matching

- compute fingerprints of strings
- application not limited to LCEs

**Definition: Karp-Rabin Fingerprint [KR87]**

Given a text $T$ of length $n$ over an alphabet of size $\sigma$ and a random prime number $q \in \Theta(n^c)$, the Karp-Rabin fingerprint of $T[i..j]$ is

$$\mathcal{F}(i, j) = \left( \sum_{k=i}^{j} T[k] \cdot \sigma^{j-k} \right) \mod q$$

**(x + y) \mod z = z \mod z + y \mod z \mod z**

- if $T[i..i + \ell] = T[j..j + \ell]$, then
  $$\mathcal{F}(i, i + \ell) = \mathcal{F}(j, j + \ell)$$

- if $T[i..i + \ell] \neq T[j..j + \ell]$, then
  $$\text{Prob}(\mathcal{F}(i, i + \ell) = \mathcal{F}(j, j + \ell)) \in O\left( \frac{\ell \log \sigma}{n^c} \right)$$

- prime has to be chosen uniformly at random
- how to turn it into Las Vegas algorithm?

**example on the board**
given a text $T$ over an alphabet of size $\sigma$
Overwriting the Text with Fingerprints (1/2) [Pre18]

- given a text $T$ over an alphabet of size $\sigma$
- let $w$ be size of a computer word e.g., 64 bit
Overwriting the Text with Fingerprints (1/2) [Pre18]

- given a text $T$ over an alphabet of size $\sigma$
- let $w$ be size of a computer word \( \uparrow \) e.g., 64 bit
- choose $\tau \in \Theta(w / \lg \sigma) \uparrow 8$ for byte alphabet
given a text $T$ over an alphabet of size $\sigma$

let $w$ be size of a computer word e.g., 64 bit

choose $\tau \in \Theta(w/\lg \sigma)$ 8 for byte alphabet

choose random prime $q \in \left[\frac{1}{2}\sigma^\tau, \sigma^\tau\right)$
Overwriting the Text with Fingerprints (1/2) [Pre18]

- given a text $T$ over an alphabet of size $\sigma$
- let $w$ be size of a computer word \( \circ \) e.g., 64 bit
- choose $\tau \in \Theta(w/\lg \sigma)$ \( \circ \) 8 for byte alphabet
- choose random prime $q \in \left[ \frac{1}{2} \sigma^\tau, \sigma^\tau \right)$
- group the text into size-$\tau$ blocks: $B[1..n/\tau]$ with

\[
B[i] = T[(i - 1)\tau + 1..i\tau]
\]
given a text \( T \) over an alphabet of size \( \sigma \)

let \( w \) be size of a computer word \( \dagger \) e.g., 64 bit

choose \( \tau \in \Theta(w / \lg \sigma) \) \( \dagger \) 8 for byte alphabet

choose random prime \( q \in [\frac{1}{2} \sigma^\tau, \sigma^\tau) \)

group the text into size-\( \tau \) blocks: \( B[1..n/\tau] \) with

\[
B[i] = T[(i - 1)\tau + 1..i\tau]
\]

compute \( P'[i] = \mathcal{H}(i, \tau i) \) for \( i \in [1, n/\tau] \)
Overwriting the Text with Fingerprints (1/2) [Pre18]

- given a text $T$ over an alphabet of size $\sigma$
- let $w$ be size of a computer word e.g., 64 bit
- choose $\tau \in \Theta(w / \lg \sigma)$ 8 for byte alphabet
- choose random prime $q \in [\frac{1}{2}\sigma^\tau, \sigma^\tau)$
- group the text into size-$\tau$ blocks: $B[1..n/\tau]$ with
  \[ B[i] = T[(i - 1)\tau + 1..i\tau] \]
- compute $P'[i] = \mathcal{H}(i, \tau i)$ for $i \in [1, n/\tau]$
- $P'[i]$ can fits in $B[i]$
given a text $T$ over an alphabet of size $\sigma$

let $w$ be size of a computer word \( \in \) e.g., 64 bit

choose $\tau \in \Theta(w/\lg \sigma)$ \( \in \) 8 for byte alphabet

choose random prime $q \in [\frac{1}{2} \sigma^\tau, \sigma^\tau)$

group the text into size-$\tau$ blocks: $B[1..n/\tau]$ with

\[
B[i] = T[(i - 1)\tau + 1..i\tau]
\]

compute $P'[i] = \bigotimes(i, \tau i)$ for $i \in [1, n/\tau]$

$P'[i]$ can fits in $B[i]$

- overwrite text with fingerprints (in-place)

all parts of text are restorable

how?
choose random prime $q \in \left[ \frac{1}{2} \sigma^\tau, \sigma^\tau \right)$

$B[i] = T[(i - 1)\tau + 1..i\tau]$

- overwrite text with fingerprints (in-place)
- choose random prime \( q \in \left[ \frac{1}{2} \sigma^r, \sigma^r \right) \)
- \( B[i] = T[(i - 1)r + 1..ir] \)
- \( \lfloor B[i]/q \rfloor \in \{0, 1\} \)

- overwrite text with fingerprints (in-place)
choose random prime $q \in \left[\frac{1}{2} \sigma^\tau, \sigma^\tau\right)$

$B[i] = T[(i-1)\cdot \tau + 1..i\cdot \tau]$

$\lfloor B[i]/q \rfloor \in \{0, 1\}$

$D[i] = \lfloor B[i]/q \rfloor$ bit vector of size $n/\tau$

- overwrite text with fingerprints (in-place)
Overwriting the Text with Fingerprints (2/2)

- choose random prime $q \in \left[\frac{1}{2} \sigma^\tau, \sigma^\tau\right)$
- $B[i] = T[(i - 1)\tau + 1..i\tau]$
- $\lfloor B[i]/q \rfloor \in \{0, 1\}$
- $D[i] = \lfloor B[i]/q \rfloor$ \begin{align*} \text{ bit vector of size } n/\tau \end{align*}
- $P'[i] = \mathbb{B}(i, \tau i)$ and together with $D$:
  \begin{align*} \lfloor B[i]/q \rfloor & \in \{0, 1\} \\ D[i] & \in \{0, 1\} \end{align*}

\[ B[i] = (P'[i] - \sigma^\tau \cdot P'[i - 1] \mod q) + D[i] \cdot q \]

- overwrite text with fingerprints (in-place)
choose random prime $q \in \left[\frac{1}{2} \sigma^\tau, \sigma^\tau\right)$

- $B[i] = T[(i - 1)\tau + 1..i\tau]$
- $\lfloor B[i]/q \rfloor \in \{0, 1\}$
- $D[i] = \lfloor B[i]/q \rfloor$ \hspace{1em} bit vector of size $n/\tau$
- $P'[i] = \text{\#}(i, \tau i)$ and together with $D$:
  
  $$B[i] = (P'[i] - \sigma^\tau \cdot P'[i - 1] \mod q) + D[i] \cdot q$$

- this gives us access to the text(!)
choose random prime $q \in \left[\frac{1}{2} \sigma^\tau, \sigma^\tau\right)$

$B[i] = T[(i - 1)\tau + 1..i\tau]$

$\lfloor B[i]/q \rfloor \in \{0, 1\}$

$D[i] = \lfloor B[i]/q \rfloor$  bit vector of size $n/\tau$

$P'[i] = \overline{(i, \tau i)}$ and together with $D$:

$B[i] = (P'[i] - \sigma^\tau \cdot P'[i - 1] \mod q) + D[i] \cdot q$

this gives us access to the text(!)

$q$ can be chosen such that MSB of $P'[i]$ is zero w.h.p., then

$D$ can be stored in the MSBs

overwrite text with fingerprints (in-place)
choose random prime $q \in \left[\frac{1}{2} \sigma^\tau, \sigma^\tau\right)$

$B[i] = T[(i-1)^\tau + 1..i^\tau]$

$\lfloor B[i]/q \rfloor \in \{0, 1\}$

$D[i] = \lfloor B[i]/q \rfloor \oplus$ bit vector of size $n/\tau$

$P'[i] = \oplus (i, \tau i)$ and together with $D$:

$B[i] = (P'[i] - \sigma^\tau \cdot P'[i-1] \mod q) + D[i] \cdot q$

this gives us access to the text(!)

$q$ can be chosen such that MSB of $P'[i]$ is zero
w.h.p., then

$D$ can be stored in the MSBs

- overwrite text with fingerprints (in-place)

- enough to answer LCE queries
choose random prime $q \in \left[ \frac{1}{2} \tau, \tau \right)$

$B[i] = T[(i - 1) \tau + 1..i \tau]$

$\lfloor B[i]/q \rfloor \in \{0, 1\}$

$D[i] = \lfloor B[i]/q \rfloor$ \text{ bit vector of size $n/\tau$}

$P'[i] = (i, \tau i)$ and together with $D$:

$B[i] = (P'[i] - \sigma^\tau \cdot P'[i - 1] \mod q) + D[i] \cdot q$

this gives us access to the text(!)

$q$ can be chosen such that MSB of $P'[i]$ is zero w.h.p., then

$D$ can be stored in the MSBs

---

overwrite text with fingerprints (in-place)

enough to answer LCE queries

how?
LCEs with Fingerprints

- compute LCE of $i$ and $j$
- exponential search until $\not= (i, i + 2^k) \neq (j, j + 2^k)$
- binary search to find correct block $m$
- recompute $B[m]$ and find mismatching character

overwrite text with fingerprints (in-place)
LCEs with Fingerprints

- compute LCE of $i$ and $j$
- exponential search until $f(i, i + 2^k) \neq f(j, j + 2^k)$
- binary search to find correct block $m$
- recompute $B[m]$ and find mismatching character

- requires $O(\lg \ell)$ time for LCEs of size $\ell$
Definition: Simplified $\tau$-Synchronizing Sets [KK19]

Given a text $T$ of length $n$ and $0 < \tau \leq n/2$, a simplified $\tau$-synchronizing set $S$ of $T$ is

$$S = \{i \in [1, n - 2\tau + 1] : \min\{\min(j, j + \tau - 1) : j \in [i, i + \tau]\} \in \{(i, i + \tau - 1), (i + \tau, i + 2\tau - 1)\}\}$$
Definition: Simplified $\tau$-Synchronizing Sets [KK19]

Given a text $T$ of length $n$ and $0 < \tau \leq n/2$, a simplified $\tau$-synchronizing set $S$ of $T$ is

$$S = \{ i \in [1, n - 2\tau + 1] : \min \{ (j, j + \tau - 1) : j \in [i, i + \tau] \} \in \{(i, i + \tau - 1), (i + \tau, i + 2\tau - 1)\} \}$$
String Synchronizing Sets (Simplified, 1/2)

Definition: Simplified $\tau$-Synchronizing Sets [KK19]

Given a text $T$ of length $n$ and $0 < \tau \leq n/2$, a simplified $\tau$-synchronizing set $S$ of $T$ is

$$S = \{i \in [1, n - 2\tau + 1] : \min\{\max(j, j + \tau - 1) : j \in [i, i + \tau]\} \in \{(i, i + \tau - 1), (i + \tau, i + 2\tau - 1)\}\}$$
String Synchronizing Sets (Simplified, 1/2)

Definition: Simplified $\tau$-Synchronizing Sets [KK19]

Given a text $T$ of length $n$ and $0 < \tau \leq n/2$, a simplified $\tau$-synchronizing set $S$ of $T$ is

$$S = \{i \in [1, n - 2\tau + 1] : \min\{(j, j + \tau - 1) : j \in [i, i + \tau]\} \in \{(i, i + \tau - 1), (i + \tau, i + 2\tau - 1)\}\}$$
String Synchronizing Sets (Simplified, 1/2)

Definition: Simplified $\tau$-Synchronizing Sets [KK19]

Given a text $T$ of length $n$ and $0 < \tau \leq n/2$, a simplified $\tau$-synchronizing set $S$ of $T$ is

$$S = \{i \in [1, n - 2\tau + 1]: \min\{min(j, j + \tau - 1): j \in [i, i + \tau]\} \in \{(i, i + \tau - 1), (i + \tau, i + 2\tau - 1)\}\}$$
Definition: Simplified $\tau$-Synchronizing Sets [KK19]

Given a text $T$ of length $n$ and $0 < \tau \leq n/2$, a simplified $\tau$-synchronizing set $S$ of $T$ is

$$S = \{ i \in [1, n - 2\tau + 1] : \min\{ (j, j + \tau - 1) : j \in [i, i + \tau] \} \in \{(i, i + \tau - 1), (i + \tau, i + 2\tau - 1)\} \}$$
String Synchronizing Sets (Simplified, 1/2)

**Definition: Simplified $\tau$-Synchronizing Sets [KK19]**

Given a text $T$ of length $n$ and $0 < \tau \leq n/2$, a simplified $\tau$-synchronizing set $S$ of $T$ is

$$S = \{ i \in [1, n-2\tau + 1] : \min \{ (j, j+\tau-1) : j \in [i, i+\tau] \} \in \{ (i, i+\tau-1), (i+\tau, i+2\tau-1) \} \}$$
Definition: Simplified $\tau$-Synchronizing Sets [KK19]

Given a text $T$ of length $n$ and $0 < \tau \leq n/2$, a simplified $\tau$-synchronizing set $S$ of $T$ is

$$S = \{i \in [1, n - 2\tau + 1]: \min\{\text{synchronizing}(j, j + \tau - 1): j \in [i, i + \tau]\} \in \{\text{synchronizing}(i, i + \tau - 1), \text{synchronizing}(i + \tau, i + 2\tau - 1)\}\}$$
Definition: Simplified $\tau$-Synchronizing Sets [KK19]

Given a text $T$ of length $n$ and $0 < \tau \leq n/2$, a simplified $\tau$-synchronizing set $S$ of $T$ is

$$S = \{ i \in [1, n-2\tau+1] : \min\{(j, j+\tau-1) : j \in [i, i+\tau]\} \in \{(i, i+\tau-1), (i+\tau, i+2\tau-1)\} \}$$
String Synchronizing Sets (Simplified, 2/2)

- $|S| = \Theta(n/\tau)$ in practice (on most data sets)
- more complex definition required to obtain this size

Consistency & (Simplified) Density Property

- for all $i, j \in [1, n - 2\tau + 1]$ we have

$$T[i, i+2\tau-1] = T[j, j+2\tau-1] \Rightarrow i \in S \Leftrightarrow j \in S$$

- for any $\tau$ consecutive positions there is at least one position in $S$
# Answering LCE Queries with String Synchronizing Sets (1/2)

## Text $T'$ for Positions in $S$

| $s_1$ | $s_2$ | $s_3$ | $s_{|S| - 3}$ | $s_{|S| - 2}$ | $s_{|S| - 1}$ |
|-------|-------|-------|----------------|----------------|----------------|
| ✓     | ✓     | ✓     | ✓              | ✓              | ✓              |

...
Answering LCE Queries with String Synchronizing Sets (1/2)

Text $T'$ for Positions in $S$

$T' \left[ \begin{array} {cccccc} s_1 & s_2 & s_3 & s_{|S| - 3} & s_{|S| - 2} & s_{|S| - 1} \\ \checkmark & \checkmark & \checkmark & \cdots & \checkmark & \checkmark \end{array} \right]$

$T'\left[1\right]$ 3$\tau$

$T'\left[2\right]$ 3$\tau$

$T'\left[3\right]$ 3$\tau$

$\cdots$

$T'\left[|S| - 3\right]$ 3$\tau$

$T'\left[|S| - 2\right]$ 3$\tau$

$T'\left[|S| - 1\right]$ 3$\tau$
in practice, we sort the substrings
- characters of $T'$ are the ranks of substrings
- build BB LCE for $T'$ w.r.t. length in $T$

**Answering Queries**
- compare naively for $3\tau$ characters
- if equal find successors of $i$ and $j$ in $S$
- compute LCE of successors in $T'$

---

Answering LCE Queries with String Synchronizing Sets (2/2)

![Diagram of string $T$ with LCE values for suffixes $s_1$, $s_2$, $s_3$, $s_{|S|-3}$, $s_{|S|-2}$, and $s_{|S|-1}$]
in practice, we sort the substrings
- characters of $T'$ are the ranks of substrings
- build BB LCE for $T'$ w.r.t. length in $T$

**Answering Queries**
- compare naively for $3\tau$ characters
- if equal find successors of $i$ and $j$ in $S$
- compute LCE of successors in $T'$
in practice, we sort the substrings
characters of $T'$ are the ranks of substrings
build BB LCE for $T'$ w.r.t. length in $T$

Answering Queries
- compare naively for $3\tau$ characters
- if equal find successors of $i$ and $j$ in $S$
- compute LCE of successors in $T'$
in practice, we sort the substrings
characters of $T'$ are the ranks of substrings
build BB LCE for $T'$ w.r.t. length in $T$

Answering Queries
- compare naively for $3\tau$ characters
- if equal find successors of $i$ and $j$ in $S$
- compute LCE of successors in $T'$
in practice, we sort the substrings
characters of $T'$ are the ranks of substrings
build BB LCE for $T'$ w.r.t. length in $T$

Answering Queries
- compare naively for $3\tau$ characters
- if equal find successors of $i$ and $j$ in $S$
- compute LCE of successors in $T'$

in this example: \( \text{lce}_T(i, j) = s_1 - i + \text{lce}_{T'}(1, |S| - 2) \)
Answering LCE Queries with String Synchronizing Sets (2/2)

- in practice, we sort the substrings
- characters of $T'$ are the ranks of substrings
- build BB LCE for $T'$ w.r.t. length in $T$

Answering Queries

- compare naively for $3\tau$ characters
- if equal find successors of $i$ and $j$ in $S$
- compute LCE of successors in $T'$

In this example: $\text{lce}_T(i, j) = s_1 - i + \text{lce}_{T'}(1, |S| - 2)$

In practice: we have a very fast static successor data structure
Practical Evaluation [Din+20]

Throughput (queries/s) vs. LCEs in $[2^k, 2^{k+1})$ for dna, english.1024MB, and cere.

Legend:
- our-rk
- SSS$_{512}$
- SSS$^{pl}_{512}$
- naive
- prezza-rk
- ultra_naive
- sada
- sct3
Practical Evaluation [Din+20]

DNA

English.1024MB

Cere

LCEs in $[2^k, 2^{k+1})$

# throughput [queries/s]#

- our-rk
- sss512
- sss^pl
- ultra_naive
- sada
- sct3

Institute of Theoretical Informatics, Algorithm Engineering
Practical Evaluation [Din+20]

DNA

English.1024MB

Cere

 throughput [queries/s]

LCEs in \([2^k, 2^{k+1})\]

- our-rk
- sss\(_{512}\)
- sss\(_{512}\) pl
- naive
- prezza-rk
- ultra_naive
- sada
- sct3
Practical Evaluation [Din+20]

![Graphs showing throughput vs. LCEs in DNA, English.1024MB, and Cere](Image)

- **DNA**: throughput [queries/s] for LCEs in $[2^k, 2^{k+1})$.
- **English.1024MB**: throughput [queries/s] for LCEs in $[2^k, 2^{k+1})$.
- **Cere**: throughput [queries/s] for LCEs in $[2^k, 2^{k+1})$.

Legend:
- **our-rk**
- **SSS512**
- **ssp512**
- **naive**
- **prezza-rk**
- **ultra_naive**
- **sada**
- **sct3**

Throughput is measured as queries per second [queries/s].
Warning

This is just a very succinct overview.
Please refer to the lecture slides for more details.
Tries & Suffix Trees

Trie Representations
- different trie representations
- space-time trade-off

Suffix Tree (Compact Trie)
Suffix Array

Given a text $T$ of length $n$, the suffix array (SA) is a permutation of $[1..n]$, such that for $i \leq j \in [1..n]$

$$T[SA[i]..n] \leq T[SA[j]..n]$$

**SAIS**

- linear time suffix array construction
- induced copying and recursion
  - classification
  - sorting special suffixes
  - inducing other suffixes

**SA Construction in EM**

- Prefix Doubling
- DC3
### LCP-Array & LCE-Queries

<table>
<thead>
<tr>
<th></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>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>$</td>
</tr>
<tr>
<td>$SA$</td>
<td>13</td>
<td>12</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>3</td>
<td>11</td>
<td>2</td>
<td>10</td>
<td>7</td>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>$LCP$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

- speed up pattern matching in suffix array
- suffix tree construction
- compression

### Longest Common Extensions

- lcp-value between any suffix
- scan or RMQ
- Rabin-Karp fingerprints
- string synchronizing sets
Compression

Entropy

Given a text $T$ of length $n$ over an alphabet $\Sigma = [1, \sigma]$ and its histogram $Hist$, then

$$H_k = \frac{1}{n} \sum_{S \in \Sigma^k} |T_S| \cdot H_0(T_S)$$

Huffman Codes

- variable length codes
- more frequent characters get shorter codes
- canonical Huffman-codes
- Shannon-Fano codes can be worse, but
- are still optimal

LZ77

$T = \text{abababbbaba}$

- $f_1 = a$
- $f_2 = b$
- $f_3 = abab$
- $f_4 = bbb$
- $f_5 = aba$
- $f_6 = $

LZ78

$T = \text{abababbbaba}$

- $f_1 = a$
- $f_2 = b$
- $f_3 = ab$
- $f_4 = abb$
- $f_5 = bb$
- $f_6 = aba$
- $f_7 = $
Burrows-Wheeler Transform

Given a text $T$ of length $n$ and its suffix array $SA$, for $i \in [1, n]$ the Burrows-Wheeler transform is

$$BWT[i] = \begin{cases} T[SA[i] - 1] & SA[i] > 1 \\ $ & SA[i] = 1 \end{cases}$$

<table>
<thead>
<tr>
<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>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a $</td>
</tr>
<tr>
<td>$SA$</td>
<td>13</td>
<td>12</td>
<td>1</td>
<td>9</td>
<td>6</td>
<td>3</td>
<td>11</td>
<td>2</td>
<td>10</td>
<td>7</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>$BWT$</td>
<td>a</td>
<td>b</td>
<td>$c$</td>
<td>c</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>b</td>
</tr>
</tbody>
</table>

LF-Mapping

Given a $BWT$, its $C$-array, and its $rank$-Function, then

$$LF(i) = C[BWT[i]] + rank_{BWT[i]}(i)$$

- transform back to text
- used in backwards search

Compression using BWT

- move-to-front
- run-length compression
Wavelet Tree

- generalize rank and select to alphabets of size $\geq 2$

Compression
- build over text compressed with canonical Huffman codes

Bit Vectors
- rank and select queries on bit vectors in $O(1)$ time and $o(n)$ space
### FM-Index & r-Index

#### Function \( \text{BackwardsSearch}(P[1..n], C, rank) \):

1. \( s = 1, e = n \)
2. \( \text{for } i = m, \ldots, 1 \text{ do} \)
3. \( s = C[P[i]] + rank_{P[i]}(s - 1) + 1 \)
4. \( e = C[P[i]] + rank_{P[i]}(e) \)
5. \( \text{if } s > e \text{ then} \)
6. \( \quad \text{return } \emptyset \)
7. \( \text{return } [s, e] \)

#### FM-Index

- use (compressed wavelet tree for rank)
- compress bit vectors further

#### r-Index

- store lots of arrays
- containing information for each run
- size proportional to number of runs
- queries become harder

#### Move Data Structure

- make use of “same” intervals in BWT and first row
- constant time mapping on balanced input/output intervals
- balancing with blowup \( \leq 2 \) achievable
Compressed Indices

**Block Tree**
- answer rank and select queries
- size proportional to number of LZ-factors

**Number of Runs and LZ-Factors**
Let $T$ be a text of length $n$, then

$$r(T) \in O(z(T) \log^2 n)$$
Document Retrieval

Document Listing

- optimal with document array and chain array

$P = TA$

![Diagram showing a graph with nodes labeled with characters and indices, representing a longest common extension data structure.](image_url)
1 The old night keeper keeps the keep in the town
2 In the big old house in the big old gown
3 The house in the town had the big old keep
4 Where the old night keeper never did sleep
5 The night keeper keeps the keep in the night
6 And keeps in the dark and sleeps in the light

<table>
<thead>
<tr>
<th>term</th>
<th>$t_i$</th>
<th>$L(t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>1</td>
<td>[6]</td>
</tr>
<tr>
<td>big</td>
<td>2</td>
<td>[2, 3]</td>
</tr>
<tr>
<td>dark</td>
<td>1</td>
<td>[6]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>had</td>
<td>1</td>
<td>[3]</td>
</tr>
<tr>
<td>house</td>
<td>2</td>
<td>[2, 3]</td>
</tr>
<tr>
<td>in</td>
<td>5</td>
<td>[1, 2, 3, 5, 6]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Encodings**
- unary/ternary encoding
- Fibonacci encoding
- Elias-$\delta/\gamma$ encoding
- Golomb encoding

**List Intersecting**
- binary/exponential search
- two levels
Longest Common Extensions

Sophisticated Black Box (BB)
- based on ISA, LCP, and RMQ
- $O(1)$ query time, $\approx 9n$ bytes additional space

Ultra Naive Scan (UNS)
- compare character by character
- $O(n)$ query time, no additional space

Definition: Simplified $\tau$-Synchronizing Sets

Given a text $T$ of length $n$ and $0 < \tau \leq n/2$, a simplified $\tau$-synchronizing set $S$ of $T$ is

$$S = \{i \in [1, n - 2\tau + 1]: \min\{(j, j + \tau - 1): j \in [i, i + \tau]\} \in \{(i, i + \tau - 1), (i + \tau, i + 2\tau - 1)\}\}$$
Conclusion and Outlook

This Lecture
- longest common extension queries
- Karp-Rabin fingerprints
- string synchronizing sets
- big recap and Q&A

That's all! We are (mostly) done.
Conclusion and Outlook

This Lecture
- longest common extension queries
- Karp-Rabin fingerprints
- string synchronizing sets
- big recap and Q&A

Next Week
- project presentation

That's all! We are (mostly) done.