Text Indexing

Lecture 11: Longest Common Extensions

Florian Kurpicz

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**

- $\mathit{lcp}(i,j) = \max\{k \colon T[i..i+k)$
- $\mathit{lcp}(i,j) = T[j..j+k)\} = \mathit{LCP}[\mathit{RMQ}_{\mathit{LCP}}(i+1,j)]$
- RMQs can be answered in $O(1)$ time and
- require $O(n)$ space

Definition: Range Minimum Queries

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

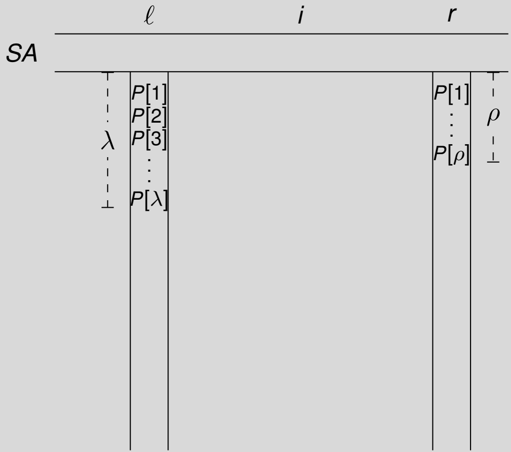
$$\mathit{RMQ}_A(\ell,r) = \arg\min\{A[k] \colon k \in [\ell,r]\}$$

Recap: Pattern Matching with the LCP-Array (2/3)

- during binary search matched
- λ characters with left border ℓ and
- ρ 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 $\xi = lcp(SA[\ell], SA[i])$ ℹ $O(1)$ time with RMQs



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

- let $\xi = lcp(SA[\ell], SA[i])$

$\xi > \lambda$

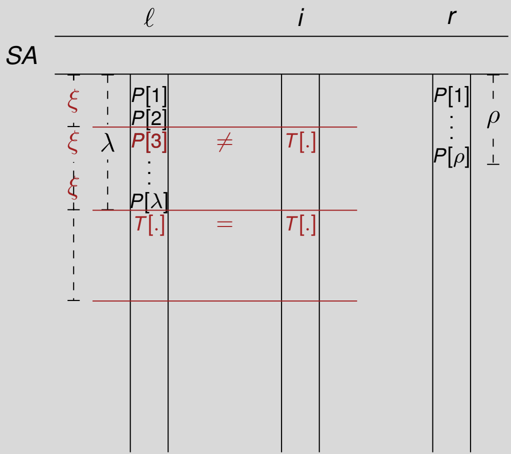
- $P[\lambda + 1] > T[SA[\ell] + \lambda] = T[SA[i] + \lambda]$
- $\ell = i$ without character comparison

$\xi = \lambda$

- compare as before

$\xi < \lambda$

- $\xi \geq \rho$ and $P[\xi + 1] < T[SA[i] + \xi]$
- $r = i$ and $\rho = \xi$ without character comparison



Old Problem, New Name

Definition: Longest Common Extensions

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

$$\mathrm{lce}_T(i,j) = \max\{\ell \geq 0 \colon T[i, i+\ell) = T[j, j+\ell)\}$$

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

Old Problem, New Name

Definition: Longest Common Extensions

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

$$\mathrm{lce}_T(i,j) = \max\{\ell \geq 0\colon\ T[i, i+\ell) = T[j, j+\ell)\}$$

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

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	A	B	C	D	A	B	C	C	D	B	C	C	B	A	B	C	D	A	D	A

$$\mathrm{lce}_T(1, 14) = 0\ 1\ 2\ 3\ 4\ 5$$

Old Problem, New Name

Definition: Longest Common Extensions

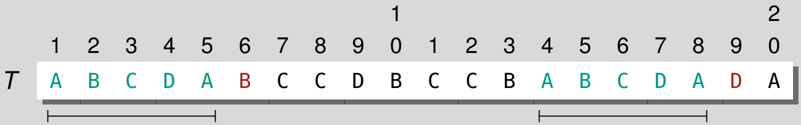
Given a text T of size n over an alphabet of size σ, construct data structure that answers for $i, j \in [1, n]$

$$\mathrm{lce}_T(i,j) = \max\{\ell \geq 0 \colon T[i, i+\ell) = T[j, j+\ell)\}$$

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

Applications

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



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	A	B	C	D	A	B	C	C	D	B	C	C	B	A	B	C	D	A	D	A

$$\mathrm{lce}_T(1, 14) = 0\ 1\ 2\ 3\ 4\ 5$$

Practical Algorithms for Longest Common Extensions [IT09]

Sophisticated Black Box (BB)

- based on ISA, LCP, and RMQ

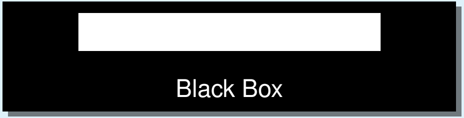
Black Box

- $O(1)$ query time, $\approx 9n$ bytes additional space

Practical Algorithms for Longest Common Extensions [IT09]

Sophisticated Black Box (BB)

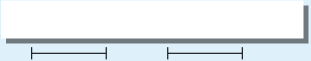
- based on ISA, LCP, and RMQ



- $O(1)$ query time, $\approx 9n$ bytes additional space

Ultra Naive Scan (UNS)

- compare character by character

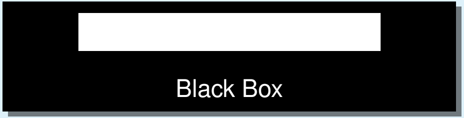


- $O(n)$ query time, no additional space

Practical Algorithms for Longest Common Extensions [IT09]

Sophisticated Black Box (BB)

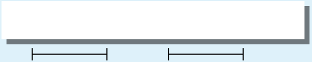
- based on ISA, LCP, and RMQ



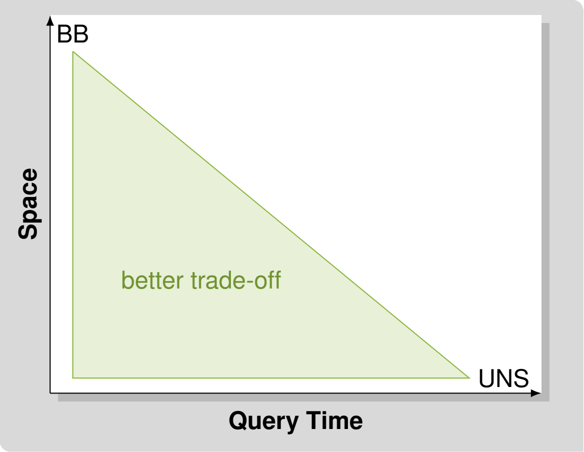
- $O(1)$ query time, $\approx 9n$ bytes additional space

Ultra Naive Scan (UNS)

- compare character by character



- $O(n)$ query time, no additional space



Monte Carlo and Las Vegas Algorithms

- setting: randomized algorithms

Monte Carlo Algorithm

- returns wrong result with small probability
- deterministic 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

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 🫆s of strings
- application not limited to LCEs

Randomized String Matching

- compute 🫆s of strings
- application not limited to LCEs

Definition: Karp-Rabin Fingerprint [KR87]

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

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

ℹ $(x+y) \bmod z = x \bmod z + y \bmod z \pmod z$

Randomized String Matching

- compute 🫆s of strings
- application not limited to LCEs

Definition: Karp-Rabin Fingerprint [KR87]

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

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

ℹ $(x+y) \bmod z = x \bmod z + y \bmod z \pmod{z}$

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

$$\text{🫆}(i,i+\ell) = \text{🫆}(j,j+\ell)$$

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

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

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

Randomized String Matching

- compute 🫆s of strings
- application not limited to LCEs

Definition: Karp-Rabin Fingerprint [KR87]

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

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

ℹ $(x+y) \bmod z = x \bmod z + y \bmod z \pmod z$

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

$$\text{🫆}(i,i+\ell) = \text{🫆}(j,j+\ell)$$

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

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

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

- example on the board

Overwriting the Text with Fingerprints (1/2) [Pre18]

- given a text T over an alphabet of size σ

Overwriting the Text with Fingerprints (1/2) [Pre18]

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

Overwriting the Text with Fingerprints (1/2) [Pre18]

- given a text T over an alphabet of size σ
- 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 $q \in [\frac{1}{2}\sigma^\tau, \sigma^\tau)$

Overwriting the Text with Fingerprints (1/2) [Pre18]

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

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

Overwriting the Text with Fingerprints (1/2) [Pre18]

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

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

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

Overwriting the Text with Fingerprints (1/2) [Pre18]

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

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

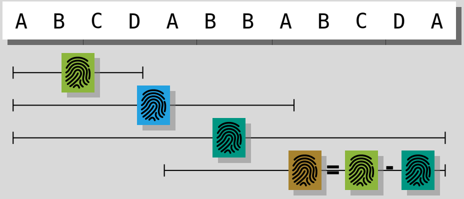
- compute $P'[i] = \text{🫆}(i, \tau i)$ for $i \in [1, n/\tau]$
- $P'[i]$ can fits in $B[i]$

Overwriting the Text with Fingerprints (1/2) [Pre18]

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

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

- compute $P'[i] = 🫆(i, \tau i)$ for $i \in [1, n/\tau]$
- $P'[i]$ can fits in $B[i]$

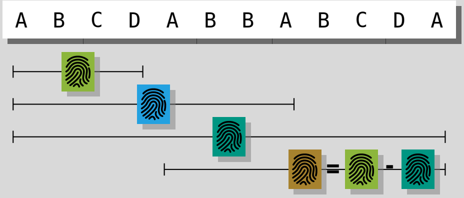


- overwrite text with fingerprints (in-place)

- all parts of text are restorable
- how?

Overwriting the Text with Fingerprints (2/2)

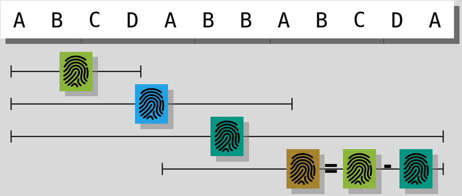
- choose random prime $q \in [\frac{1}{2}\sigma^\tau, \sigma^\tau)$
- $B[i] = T[(i-1)\tau + 1..i\tau]$



- overwrite text with fingerprints (in-place)

Overwriting the Text with Fingerprints (2/2)

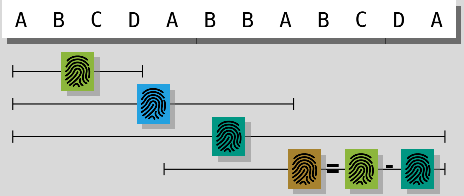
- choose random prime $q \in [\frac{1}{2}\sigma^{\tau}, \sigma^{\tau})$
- $B[i] = T[(i-1)\tau + 1..i\tau]$
- $\lfloor B[i]/q \rfloor \in \{0, 1\}$



- overwrite text with fingerprints (in-place)

Overwriting the Text with Fingerprints (2/2)

- choose random prime $q \in [\frac{1}{2}\sigma^\tau, \sigma^\tau)$
- $B[i] = T[(i-1)\tau + 1..i\tau]$
- $\lfloor B[i]/q \rfloor \in \{0, 1\}$
- $D[i] = \lfloor B[i]/q \rfloor$ ℹ bit vector of size n/τ

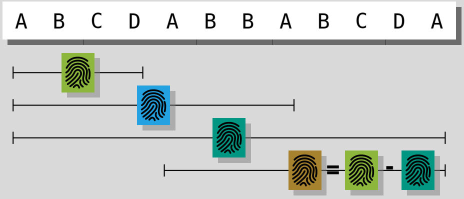


- overwrite text with fingerprints (in-place)

Overwriting the Text with Fingerprints (2/2)

- choose random prime $q \in [\frac{1}{2}\sigma^\tau, \sigma^\tau)$
- $B[i] = T[(i-1)\tau + 1..i\tau]$
- $\lfloor B[i]/q \rfloor \in \{0, 1\}$
- $D[i] = \lfloor B[i]/q \rfloor$ ℹ bit vector of size n/τ
- $P'[i] =$ 🫆$(i, \tau i)$ and together with D:

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



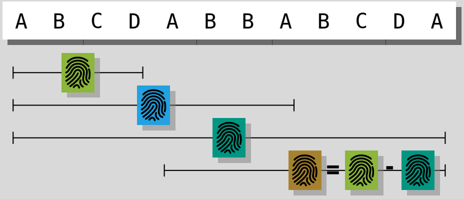
- overwrite text with fingerprints (in-place)

Overwriting the Text with Fingerprints (2/2)

- choose random prime $q \in [\frac{1}{2}\sigma^\tau, \sigma^\tau)$
- $B[i] = T[(i-1)\tau + 1..i\tau]$
- $\lfloor B[i]/q \rfloor \in \{0, 1\}$
- $D[i] = \lfloor B[i]/q \rfloor$ ℹ bit vector of size n/τ
- $P'[i] =$ 🫆$(i, \tau i)$ and together with D:

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

- this gives us access to the text(!)



- overwrite text with fingerprints (in-place)

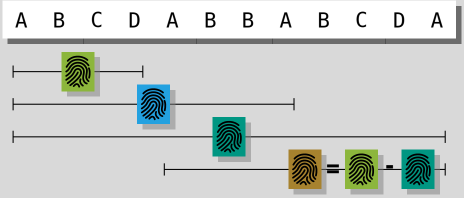
Overwriting the Text with Fingerprints (2/2)

- choose random prime $q \in [\frac{1}{2}\sigma^\tau, \sigma^\tau)$
- $B[i] = T[(i-1)\tau + 1..i\tau]$
- $\lfloor B[i]/q \rfloor \in \{0, 1\}$
- $D[i] = \lfloor B[i]/q \rfloor$ ℹ bit vector of size n/τ
- $P'[i] =$ 🫆$(i, \tau i)$ and together with D:

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

- 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)

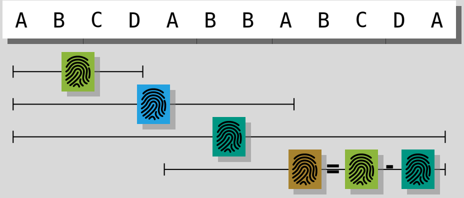
Overwriting the Text with Fingerprints (2/2)

- choose random prime $q \in [\frac{1}{2}\sigma^\tau, \sigma^\tau)$
- $B[i] = T[(i-1)\tau + 1..i\tau]$
- $\lfloor B[i]/q \rfloor \in \{0, 1\}$
- $D[i] = \lfloor B[i]/q \rfloor$ ⓘ bit vector of size n/τ
- $P'[i] =$ 🫆$(i, \tau i)$ and together with D:

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

- 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

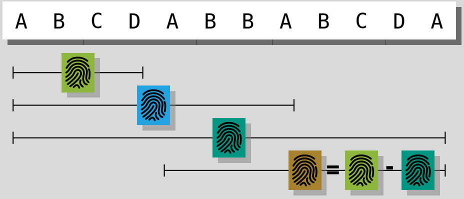
Overwriting the Text with Fingerprints (2/2)

- choose random prime $q \in [\frac{1}{2}\sigma^\tau, \sigma^\tau)$
- $B[i] = T[(i-1)\tau + 1..i\tau]$
- $\lfloor B[i]/q \rfloor \in \{0, 1\}$
- $D[i] = \lfloor B[i]/q \rfloor$ ℹ bit vector of size n/τ
- $P'[i] =$ 🫆$(i, \tau i)$ and together with D:

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

- 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



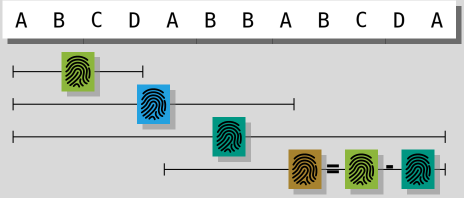
- overwrite text with fingerprints (in-place)

- enough to answer LCE queries
- how?

Answering LCE Queries with Fingerprints

LCEs with Fingerprints

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



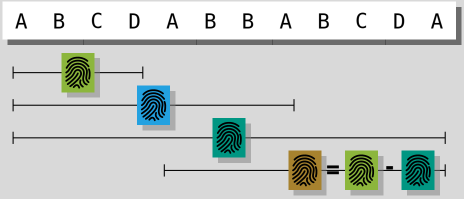
- overwrite text with fingerprints (in-place)

Answering LCE Queries with Fingerprints

LCEs with Fingerprints

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

- requires $O(\lg \ell)$ time for LCEs of size ℓ



- overwrite text with fingerprints (in-place)

String Synchronizing Sets (Simplified, 1/2)

Definition: Simplified τ-Synchronizing Sets [KK19]

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

$$S = \{i \in [1, n - 2\tau + 1] \colon min\{\text{🫆}(j, j + \tau - 1) \colon j \in [i, i + \tau]\} \in \{\text{🫆}(i, i + \tau - 1), \text{🫆}(i + \tau, i + 2\tau - 1)\}\}$$

String Synchronizing Sets (Simplified, 1/2)

Definition: Simplified τ-Synchronizing Sets [KK19]

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

$$S = \{i \in [1, n-2\tau+1] \colon \mathit{min}\{\text{🫆}(j, j+\tau-1) \colon j \in [i, i+\tau]\} \in \{\text{🫆}(i, i+\tau-1), \text{🫆}(i+\tau, i+2\tau-1)\}\}$$

String Synchronizing Sets (Simplified, 1/2)

Definition: Simplified τ-Synchronizing Sets [KK19]

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

$$S = \{i \in [1, n - 2\tau + 1] \colon \mathit{min}\{\text{🫆}(j, j + \tau - 1) \colon j \in [i, i + \tau]\} \in \{\text{🫆}(i, i + \tau - 1), \text{🫆}(i + \tau, i + 2\tau - 1)\}\}$$

String Synchronizing Sets (Simplified, 1/2)

Definition: Simplified τ-Synchronizing Sets [KK19]

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

$$S = \{i \in [1, n-2\tau+1] \colon min\{\text{🫆}(j, j+\tau-1) \colon j \in [i, i+\tau]\} \in \{\text{🫆}(i, i+\tau-1), \text{🫆}(i+\tau, i+2\tau-1)\}\}$$

String Synchronizing Sets (Simplified, 1/2)

Definition: Simplified τ-Synchronizing Sets [KK19]

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

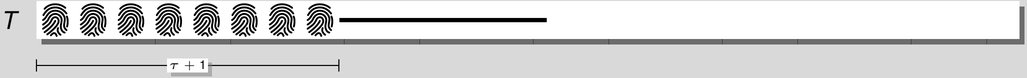
$$S = \{i \in [1, n-2\tau+1] \colon \mathit{min}\{🫆(j, j+\tau-1) \colon j \in [i, i+\tau]\} \in \{🫆(i, i+\tau-1), 🫆(i+\tau, i+2\tau-1)\}\}$$

String Synchronizing Sets (Simplified, 1/2)

Definition: Simplified τ-Synchronizing Sets [KK19]

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

$$S = \{i \in [1, n - 2\tau + 1] \colon min\{\text{🫆}(j, j + \tau - 1) \colon j \in [i, i + \tau]\} \in \{\text{🫆}(i, i + \tau - 1), \text{🫆}(i + \tau, i + 2\tau - 1)\}\}$$



String Synchronizing Sets (Simplified, 1/2)

Definition: Simplified τ-Synchronizing Sets [KK19]

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

$$S = \{i \in [1, n - 2\tau + 1] \colon min\{\text{🫆}(j, j + \tau - 1) \colon j \in [i, i + \tau]\} \in \{\text{🫆}(i, i + \tau - 1), \text{🫆}(i + \tau, i + 2\tau - 1)\}\}$$

String Synchronizing Sets (Simplified, 1/2)

Definition: Simplified τ-Synchronizing Sets [KK19]

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

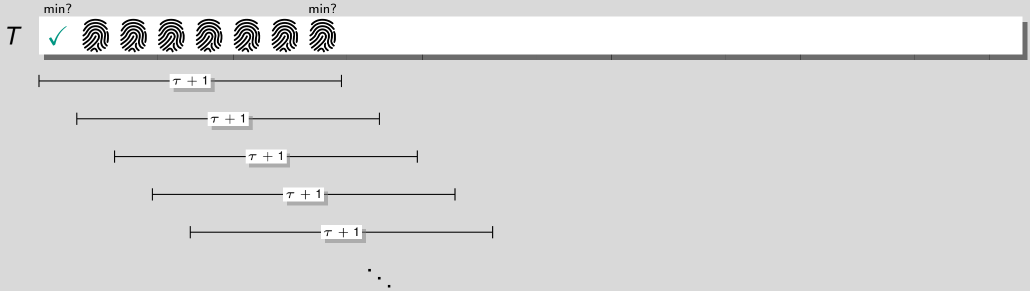
$$S = \{i \in [1, n - 2\tau + 1] \colon min\{\text{🫆}(j, j + \tau - 1) \colon j \in [i, i + \tau]\} \in \{\text{🫆}(i, i + \tau - 1), \text{🫆}(i + \tau, i + 2\tau - 1)\}\}$$

String Synchronizing Sets (Simplified, 1/2)

Definition: Simplified τ-Synchronizing Sets [KK19]

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

$$S = \{i \in [1, n - 2\tau + 1] \colon \mathit{min}\{\text{🫆}(j, j + \tau - 1) \colon j \in [i, i + \tau]\} \in \{\text{🫆}(i, i + \tau - 1), \text{🫆}(i + \tau, i + 2\tau - 1)\}\}$$



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

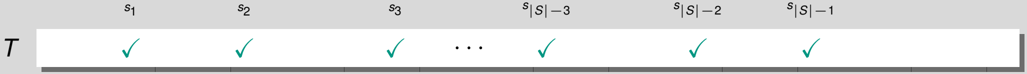
- for all $i, j \in [1, n-2\tau+1]$ we have

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

- for any τ consecutive positions there is at least one position in S

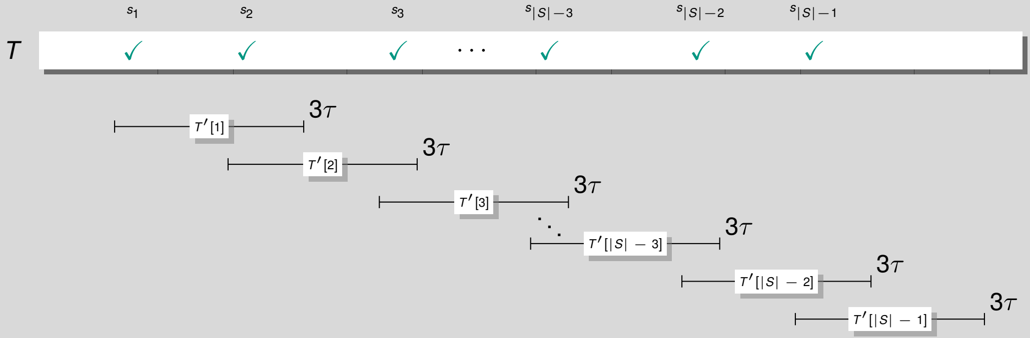
Answering LCE Queries with String Synchronizing Sets (1/2)

Text T' for Positions in S



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

Text T' for Positions in S

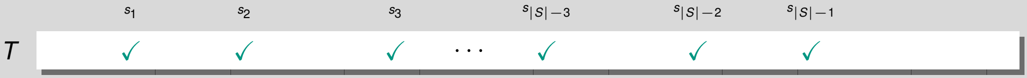


Answering LCE Queries with String Synchronizing Sets (2/2)

- 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 3τ characters
- if equal find successors of i and j in S
- compute LCE of successors in T'

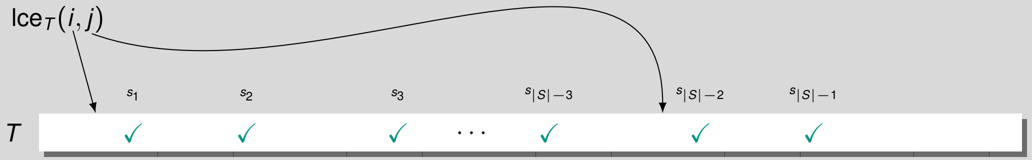


Answering LCE Queries with String Synchronizing Sets (2/2)

- 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 3τ characters
- if equal find successors of i and j in S
- compute LCE of successors in T'

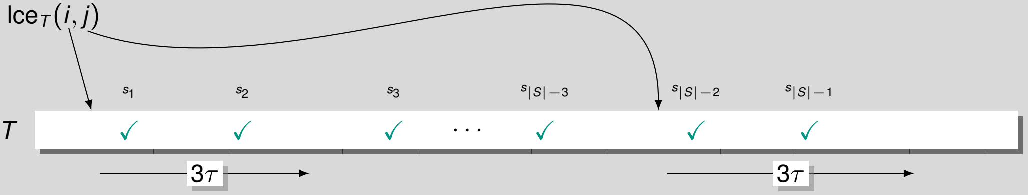


Answering LCE Queries with String Synchronizing Sets (2/2)

- 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 3τ characters
- if equal find successors of i and j in S
- compute LCE of successors in T'

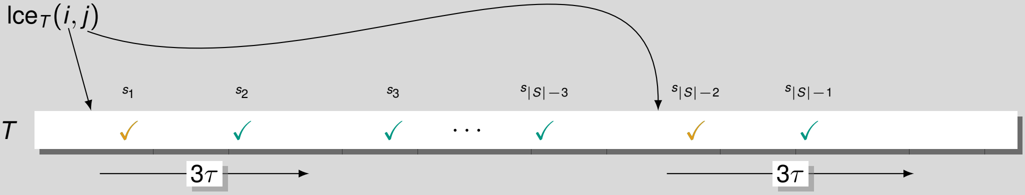


Answering LCE Queries with String Synchronizing Sets (2/2)

- 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 3τ characters
- if equal find successors of i and j in S
- compute LCE of successors in T'

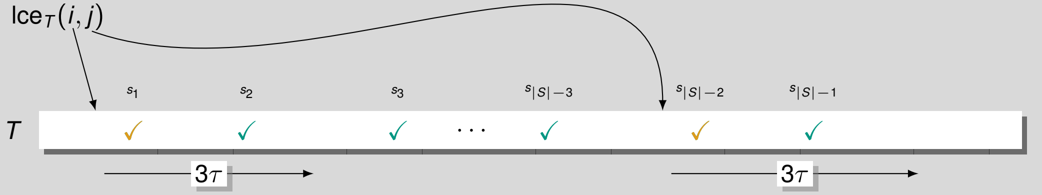


Answering LCE Queries with String Synchronizing Sets (2/2)

- 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 3τ characters
- if equal find successors of i and j in S
- compute LCE of successors in T'



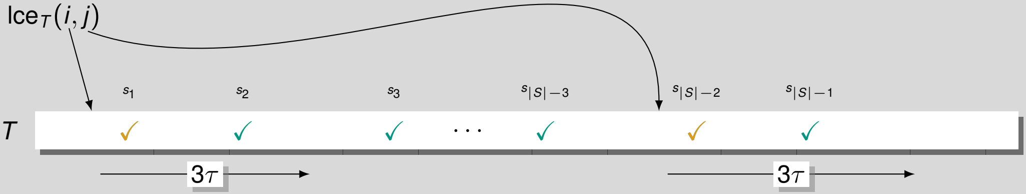
- in this example: $\mathrm{lce}_T(i,j) = s_1 - i + \mathrm{lce}_{T'}(1, |S| - 2)$

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

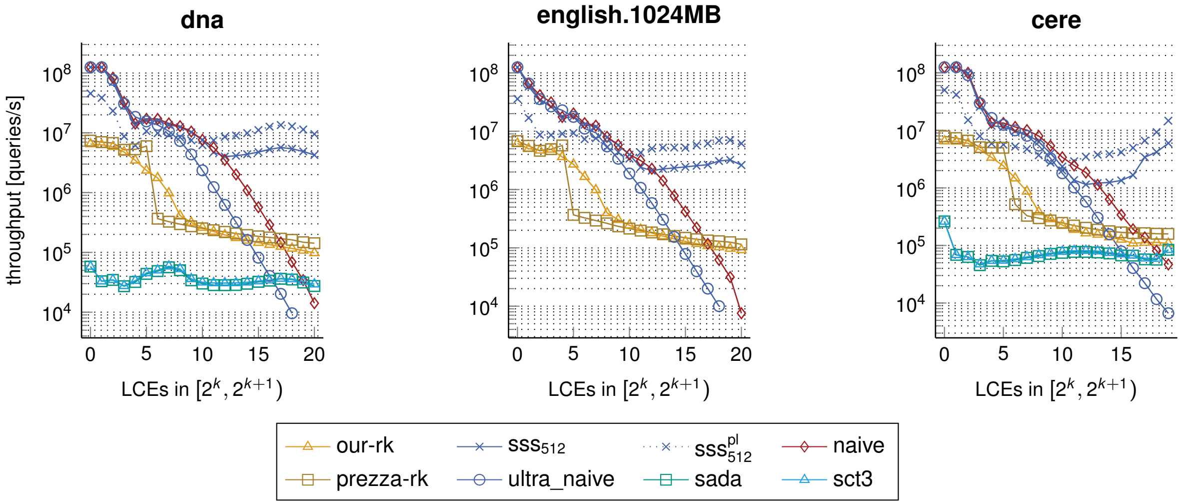
Answering Queries

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

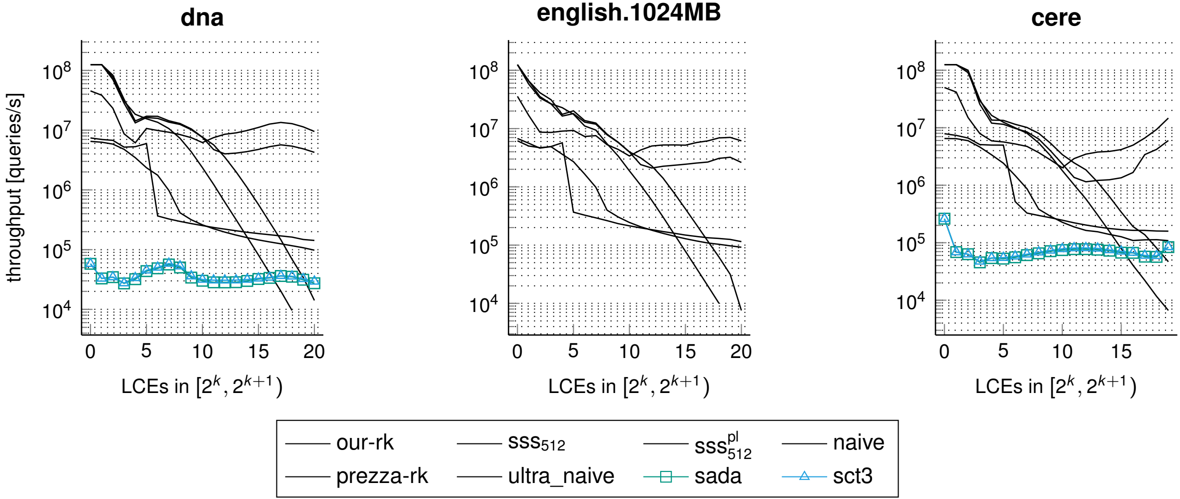


- in this example: $\mathrm{lce}_T(i,j) = s_1 - i + \mathrm{lce}_{T'}(1, |S| - 2)$
- in practice: we have a very fast static successor data structure

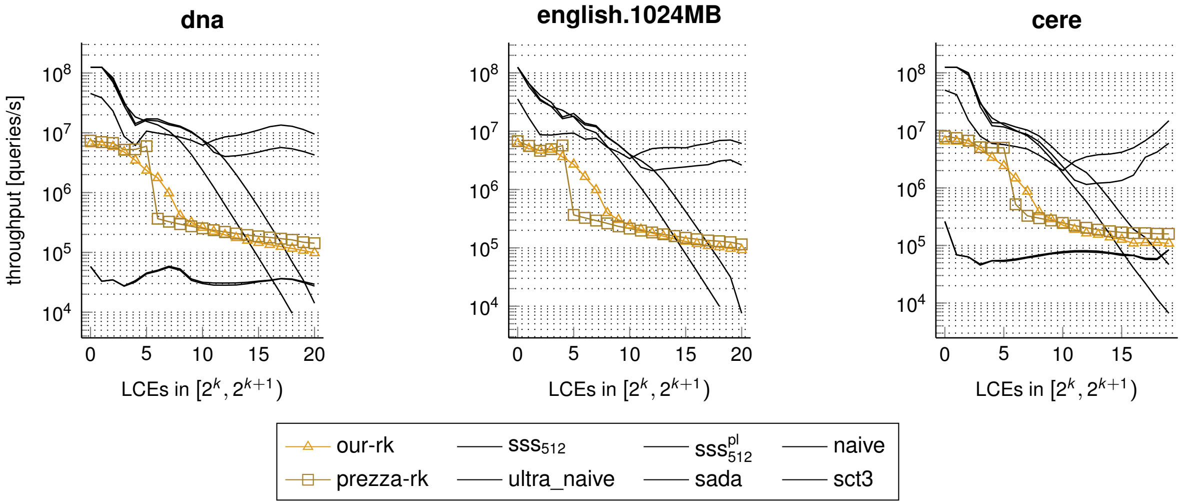
Practical Evaluation [Din+20]



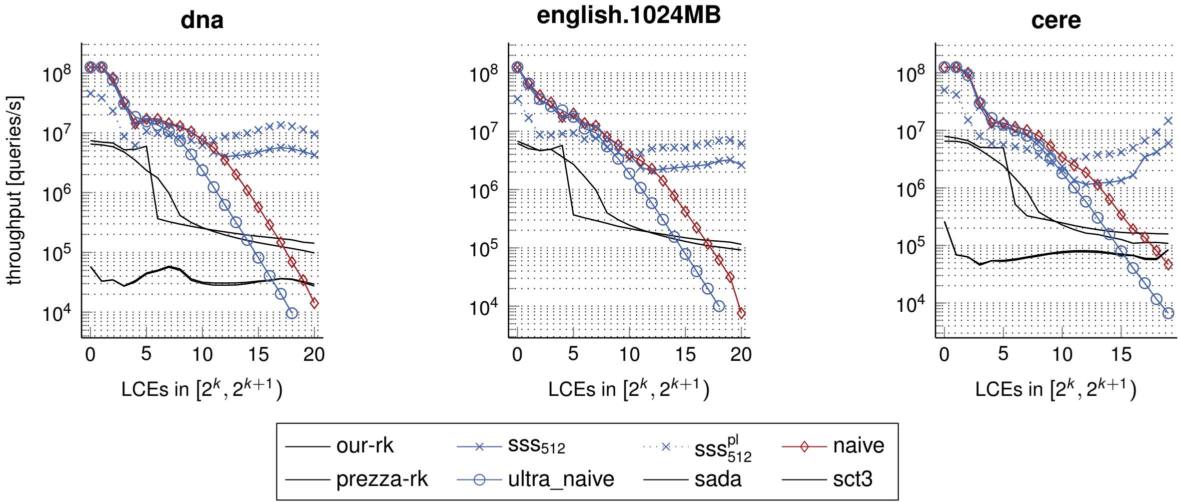
Practical Evaluation [Din+20]



Practical Evaluation [Din+20]



Practical Evaluation [Din+20]



Evaluation

https://onlineumfrage.kit.edu/evasys/online.php?p=HHRXC

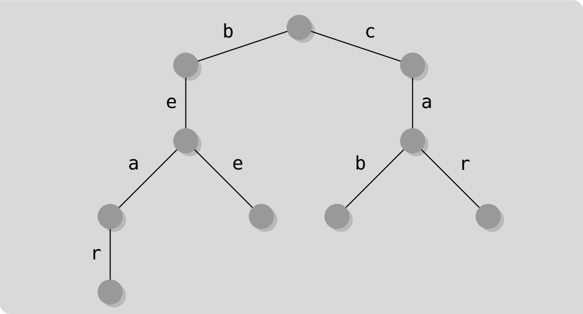
Warning

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

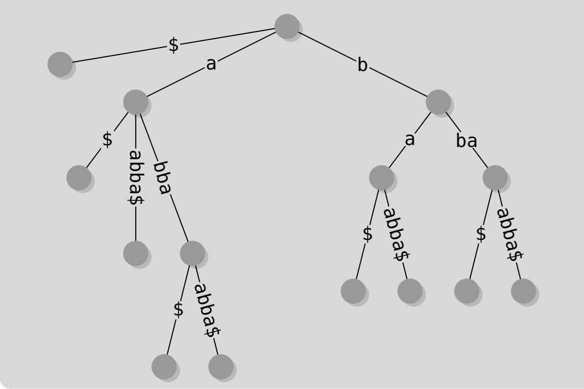
Tries & Suffix Trees

Trie Representations

- different trie representations
- space-time trade-off



Suffix Tree (Compact Trie)



Suffix Array

Suffix Array

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

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

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3

SAIS

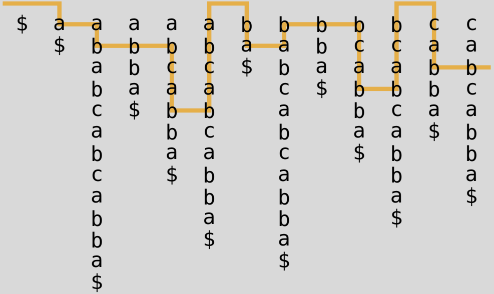
- linear time suffix array construction
- induced copying and recursion
  - classification
  - sorting special suffixes
  - inducing other suffixes

SA Construction in EM

- Prefix Doubling
- DC3

LCP-Array & LCE-Queries

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3



- 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

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

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

$T =$ `ababab bbbaba$`

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

LZ78

$T =$ `abababbbbaba$`

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

Burrows-Wheeler Transform

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

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

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
BWT	a	b	$	c	c	b	b	a	a	a	a	b	b

LF-Mapping

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

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

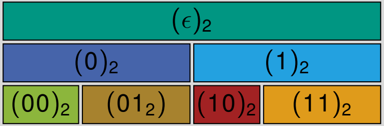
- transform back to text
- used in backwards search

Compression using BWT

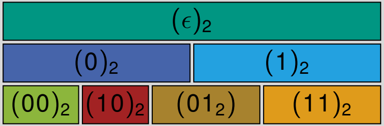
- move-to-front
- run-length compression

Wavelet Tree

Wavelet Tree



Wavelet Matrix



- generalize rank and select to alphabets of size > 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 BackwardsSearch($P[1..n]$, $C$, $rank$):
    $s = 1, e = n$
    for $i = m, \ldots, 1$ do
        $s = C[P[i]] + rank_{P[i]}(s - 1) + 1$
        $e = C[P[i]] + rank_{P[i]}(e)$
        if $s > e$ then
            return $\emptyset$
    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

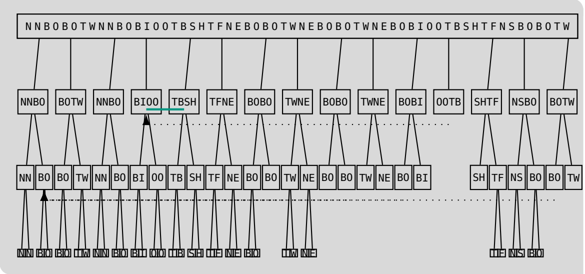
Move Data Structure

- make use of “same” intervals in BWT and first row
- constant time mapping on balanced input/output intervals
- balancing with blowup ≤ 2 achievable

Compressed Indices

Block Tree

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



Number of Runs and LZ-Factors

T be a text of length n, then

$$r(T) \in O(z(T)\lg^2 n)$$

Next Lecture!

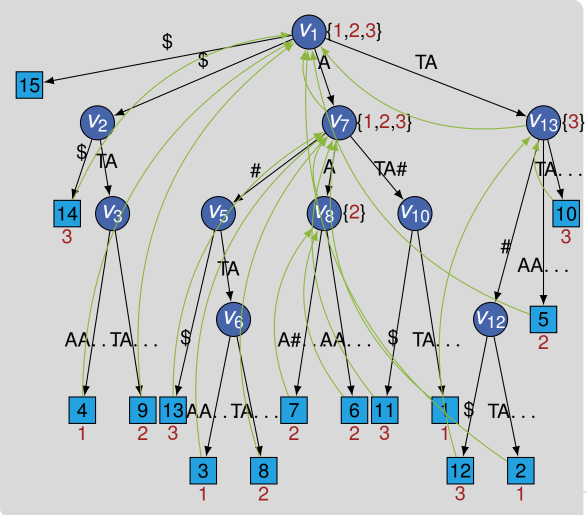
Document Retrieval

Document Listing

- optimal with document array and chain array

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	A	T	A	#	T	A	A	A	#	T	A	T	A	#	$
SA	15	14	4	9	13	3	8	7	6	11	1	12	2	5	10
DA	0	3	1	2	3	1	2	2	2	3	1	3	1	2	3
CA	0	0	0	0	2	3	4	7	8	5	6	10	11	9	12

- $P = TA$



Inverted Index

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

term t	f_t	$L(t)$
and	1	$[6]$
big	2	$[2, 3]$
dark	1	$[6]$
...	...	...
had	1	$[3]$
house	2	$[2, 3]$
in	5	$[1, 2, 3, 5, 6]$
...	...	...

Encodings

- unary/ternary encoding
- Fibonacci encoding
- Elias-δ/γ encoding
- Golomb encoding

List Interseciong

- binary/exponential search
- two levels

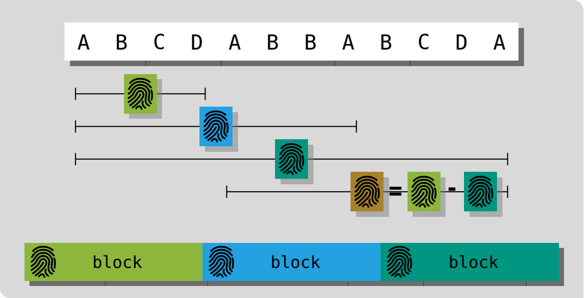
Longest Common Extensions

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 τ-Synchronizing Sets

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

$$S = \{i \in [1, n - 2\tau + 1] \colon min\{\text{🫆}(j, j + \tau - 1) \colon j \in [i, i + \tau]\} \in \{\text{🫆}(i, i + \tau - 1), \text{🫆}(i + \tau, i + 2\tau - 1)\}\}$$

Conclusion and Outlook

This Lecture

- longest common extension queries
- Karp-Rabin fingerprints
- string synchronizing sets

- big recap and Q&A

Thats 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

Thats all! We are (mostly) done.