# Construction of Sparse Suffix Trees and LCE Indexes in Optimal Time and Space

Dmitry Kosolobov    Nikita Sivukhin

Ural Federal University, Ekaterinburg, Russia

# Content

# Content

- Problems: small-space LCE indexes and sparse suffix trees

# Content

- Problems: small-space LCE indexes and sparse suffix trees
- New and known results: deterministic and randomized

# Content

- Problems: small-space LCE indexes and sparse suffix trees
- New and known results: deterministic and randomized
- Known reduction to a locally consistent parsing

# Content

- Problems: small-space LCE indexes and sparse suffix trees
- New and known results: deterministic and randomized
- Known reduction to a locally consistent parsing
- Deterministic locally consistent parsing in small space

# Sparse suffix tree and LCE index

# Sparse suffix tree and LCE index

Given a read-only length-$n$ string $s$ over the alphabet $\{0, \ldots, n^{O(1)}\}$

# Sparse suffix tree and LCE index

Given a read-only length-$n$ string $s$ over the alphabet $\{0, \ldots, n^{O(1)}\}$

- Sparse suffix tree (SST) is a compacted trie containing only specified $b$ suffixes $s[i_1..n], s[i_2..n], \ldots, s[i_b..n]$

# Sparse suffix tree and LCE index

Given a read-only length-$n$ string $s$ over the alphabet $\{0, \ldots, n^{O(1)}\}$

- Sparse suffix tree (SST) is a compacted trie containing only specified $b$ suffixes $s[i_1..n], s[i_2..n], \ldots, s[i_b..n]$
- Longest common extension (LCE) index supports the queries $LCE(p, q) = \max\{\ell \colon s[p \mathinner{..} p+\ell) = s[q \mathinner{..} q+\ell)\}$

# Sparse suffix tree and LCE index

Given a read-only length-$n$ string $s$ over the alphabet $\{0, \dots, n^{O(1)}\}$

- Sparse suffix tree (SST) is a compacted trie containing only specified $b$ suffixes $s[i_1..n], s[i_2..n], \dots, s[i_b..n]$
- Longest common extension (LCE) index supports the queries $LCE(p, q) = \max\{\ell\colon s[p\,..\,p+\ell) = s[q\,..\,q+\ell)\}$

SST takes $O(b)$ space (in machine words) on top of $s$ itself

# Sparse suffix tree and LCE index

Given a read-only length-$n$ string $s$ over the alphabet $\{0, \ldots, n^{O(1)}\}$

- Sparse suffix tree (SST) is a compacted trie containing only specified $b$ suffixes $s[i_1..n], s[i_2..n], \ldots, s[i_b..n]$
- Longest common extension (LCE) index supports the queries $LCE(p, q) = \max\{\ell\colon s[p \mathinner{..} p + \ell) = s[q \mathinner{..} q + \ell)\}$

SST takes $O(b)$ space (in machine words) on top of $s$ itself
LCE index with $O(b)$ space has $O(\frac{n}{b})$-time queries [Bille et al. 15]
(optimal trade-off, at least for $b \geq \Omega(n/\log n)$ [Kosolobov 17])

# Sparse suffix tree and LCE index

Given a read-only length-$n$ string $s$ over the alphabet $\{0, \ldots, n^{O(1)}\}$

- Sparse suffix tree (SST) is a compacted trie containing only specified $b$ suffixes $s[i_1..n], s[i_2..n], \ldots, s[i_b..n]$
- Longest common extension (LCE) index supports the queries $LCE(p, q) = \max\{\ell \colon s[p\,..\,p+\ell) = s[q\,..\,q+\ell)\}$

SST takes $O(b)$ space (in machine words) on top of $s$ itself
LCE index with $O(b)$ space has $O(\frac{n}{b})$-time queries [Bille et al. 15]
(optimal trade-off, at least for $b \geq \Omega(n/\log n)$ [Kosolobov 17])

Construct LCE index and SST in $O(b)$ space and $O(n)$ time?

# Results

| SST construction algorithm | $O(b)$ space build |
|---|---|

# Results

| SST construction algorithm | $O(b)$ space build |
|---|---|
| [Gawrychowsky, Kociumaka 17] | $O(n)^*$ |

*randomized Monte-Carlo

# Results

| SST construction algorithm | $O(b)$ space build |
|---|---|
| [Gawrychowsky, Kociumaka 17] | $O(n)^*$ |
| [Birenzwige et al. 20] | $O(n)^{**}$ |

*randomized Monte-Carlo **randomized Las Vegas

# Results

| SST construction algorithm | $O(b)$ space build |
|---|---|
| [Gawrychowsky, Kociumaka 17] | $O(n)^*$ |
| [Birenzwige et al. 20] | $O(n)^{**}$ |
| [Birenzwige et al. 20] | $O(n \log \frac{n}{b})$ |

*randomized Monte-Carlo **randomized Las Vegas

# Results

| SST construction algorithm | $O(b)$ space build |
|---:|:---:|
| [Gawrychowsky, Kociumaka 17] | $O(n)^*$ |
| [Birenzwige et al. 20] | $O(n)^{**}$ |
| [Birenzwige et al. 20] | $O(n \log \frac{n}{b})$ |
| **ours** | $\boldsymbol{O(n \log_b n)}$ |

*randomized Monte-Carlo **randomized Las Vegas

# Results

| SST construction algorithm | $O(b)$ space build |
|---:|:---:|
| [Gawrychowsky, Kociumaka 17] | $O(n)^*$ |
| [Birenzwige et al. 20] | $O(n)^{**}$ |
| [Birenzwige et al. 20] | $O(n \log \frac{n}{b})$ |
| **ours** | $\boldsymbol{O(n \log_b n)}$ |

*randomized Monte-Carlo **randomized Las Vegas

Ours: $O(n)$-time deterministic construction for $b > n^{\epsilon}$ with constant $\epsilon$

# Results

| SST construction algorithm | $O(b)$ space build |
|---|---|
| [Gawrychowsky, Kociumaka 17] | $O(n)^*$ |
| [Birenzwige et al. 20] | $O(n)^{**}$ |
| [Birenzwige et al. 20] | $O(n \log \frac{n}{b})$ |
| **ours** | $\boldsymbol{O(n \log_b n)}$ |

| LCE construction algorithm | $O(b)$ space build | Query time |
|---|---|---|

*randomized Monte-Carlo **randomized Las Vegas

Ours: $O(n)$-time deterministic construction for $b > n^\epsilon$ with constant $\epsilon$

# Results

| SST construction algorithm | $O(b)$ space build |
|---|---|
| [Gawrychowsky, Kociumaka 17] | $O(n)^*$ |
| [Birenzwige et al. 20] | $O(n)^{**}$ |
| [Birenzwige et al. 20] | $O(n \log \frac{n}{b})$ |
| **ours** | $\boldsymbol{O(n \log_b n)}$ |

| LCE construction algorithm | $O(b)$ space build | Query time |
|---|---|---|
| [Gawrychowsky, Kociumaka 17] | $O(n)^*$ | $O(\frac{n}{b})$ |
| [Birenzwige et al. 20] | $O(n)^{**}$ | $O(\frac{n}{b})$ |

*randomized Monte-Carlo **randomized Las Vegas

Ours: $O(n)$-time deterministic construction for $b > n^{\epsilon}$ with constant $\epsilon$

# Results

| SST construction algorithm | $O(b)$ space build |
|---|---|
| [Gawrychowsky, Kociumaka 17] | $O(n)^*$ |
| [Birenzwige et al. 20] | $O(n)^{**}$ |
| [Birenzwige et al. 20] | $O(n \log \frac{n}{b})$ |
| **ours** | $\boldsymbol{O(n \log_b n)}$ |

| LCE construction algorithm | $O(b)$ space build | Query time |
|---|---|---|
| [Gawrychowsky, Kociumaka 17] | $O(n)^*$ | $O(\frac{n}{b})$ |
| [Birenzwige et al. 20] | $O(n)^{**}$ | $O(\frac{n}{b})$ |
| [Tanimura et al. 16] | $O(n \cdot \frac{n}{b})$ | $O(\frac{n}{b} \log \frac{n}{b})$ |

*randomized Monte-Carlo **randomized Las Vegas

Ours: $O(n)$-time deterministic construction for $b > n^\epsilon$ with constant $\epsilon$

# Results

| SST construction algorithm | $O(b)$ space build |
|---|---|
| [Gawrychowsky, Kociumaka 17] | $O(n)^*$ |
| [Birenzwige et al. 20] | $O(n)^{**}$ |
| [Birenzwige et al. 20] | $O(n \log \frac{n}{b})$ |
| **ours** | $\boldsymbol{O(n \log_b n)}$ |

| LCE construction algorithm | $O(b)$ space build | Query time |
|---|---|---|
| [Gawrychowsky, Kociumaka 17] | $O(n)^*$ | $O(\frac{n}{b})$ |
| [Birenzwige et al. 20] | $O(n)^{**}$ | $O(\frac{n}{b})$ |
| [Tanimura et al. 16] | $O(n \cdot \frac{n}{b})$ | $O(\frac{n}{b} \log \frac{n}{b})$ |
| [Birenzwige et al. 20] | $O(n \log \frac{n}{b})$ | $O(\frac{n}{b}\sqrt{\log^* n})$ |

*randomized Monte-Carlo **randomized Las Vegas

Ours: $O(n)$-time deterministic construction for $b > n^{\epsilon}$ with constant $\epsilon$

# Results

| SST construction algorithm | $O(b)$ space build |
|---|---|
| [Gawrychowsky, Kociumaka 17] | $O(n)^*$ |
| [Birenzwige et al. 20] | $O(n)^{**}$ |
| [Birenzwige et al. 20] | $O(n \log \frac{n}{b})$ |
| **ours** | $\boldsymbol{O(n \log_b n)}$ |

| LCE construction algorithm | $O(b)$ space build | Query time |
|---|---|---|
| [Gawrychowsky, Kociumaka 17] | $O(n)^*$ | $O(\frac{n}{b})$ |
| [Birenzwige et al. 20] | $O(n)^{**}$ | $O(\frac{n}{b})$ |
| [Tanimura et al. 16] | $O(n \cdot \frac{n}{b})$ | $O(\frac{n}{b} \log \frac{n}{b})$ |
| [Birenzwige et al. 20] | $O(n \log \frac{n}{b})$ | $O(\frac{n}{b}\sqrt{\log^* n})$ |
| **ours** | $\boldsymbol{O(n \log_b n)}$ | $\boldsymbol{O(\frac{n}{b})}$ |

*randomized Monte-Carlo **randomized Las Vegas

Ours: $O(n)$-time deterministic construction for $b > n^{\epsilon}$ with constant $\epsilon$

# Partitioning sets imply LCE index + SST

# Partitioning sets imply LCE index + SST

For $1 \le \tau \le n$, a $\tau$-partitioning set of $s[1..n]$ is a subset of positions $\{1, \ldots, n\}$ with certain properties

# Partitioning sets imply LCE index + SST

For $1 \leq \tau \leq n$, a $\tau$-partitioning set of $s[1..n]$ is a subset of positions $\{1, \ldots, n\}$ with certain properties

[Birenzwige et al. 20]

Given a $\tau$-partitioning set of size $O(b)$ with $\tau = \frac{n}{b}$, one can construct in $O(n)$ time an LCE index with $O(\frac{n}{b})$-time queries and an SST on any $b$ suffixes using $O(b)$ space on top of the input

# Partitioning sets imply LCE index + SST

For $1 \le \tau \le n$, a $\tau$-partitioning set of $s[1..n]$ is a subset of positions $\{1, \ldots, n\}$ with certain properties

[Birenzwige et al. 20]

Given a $\tau$-partitioning set of size $O(b)$ with $\tau = \frac{n}{b}$, one can construct in $O(n)$ time an LCE index with $O(\frac{n}{b})$-time queries and an SST on any $b$ suffixes using $O(b)$ space on top of the input

Main result

For $\tau = \frac{n}{b}$, a $\tau$-partitioning set of size $O(b)$ can be constructed in $O(n \log_b n)$ time using $O(b)$ space on top of the string $s[1..n]$

# Partitioning sets

# Partitioning sets

A set $S \subseteq [1..n]$ is $\tau$-partitioning for a string $s[1..n]$ if:

# Partitioning sets

A set $S \subseteq [1..n]$ is $\tau$-partitioning for a string $s[1..n]$ if:

(a) if $s[i-\tau..i+\tau] = s[j-\tau..j+\tau]$, then $i \in S \Leftrightarrow j \in S$ (locally consistent)

# Partitioning sets

A set $S \subseteq [1..n]$ is $\tau$-partitioning for a string $s[1..n]$ if:

(a) if $s[i-\tau..i+\tau] = s[j-\tau..j+\tau]$, then $i \in S \Leftrightarrow j \in S$ (locally consistent)

# Partitioning sets

A set $S \subseteq [1..n]$ is $\tau$-partitioning for a string $s[1..n]$ if:

(a) if $s[i-\tau..i+\tau] = s[j-\tau..j+\tau]$, then $i \in S \Leftrightarrow j \in S$ (locally consistent)

# Partitioning sets

A set $S \subseteq [1..n]$ is $\tau$-partitioning for a string $s[1..n]$ if:

(a) if $s[i{-}\tau..i{+}\tau] = s[j{-}\tau..j{+}\tau]$, then $i \in S \Leftrightarrow j \in S$ (locally consistent)

$$\cdots\overbrace{\cdots\underset{i}{\bullet}\cdots}\cdots\overbrace{\cdots\underset{j}{\bullet}\cdots}\cdots$$

(b) if $s[i..i{+}\ell] = s[j..j{+}\ell]$ for $i, j \in S$, then for each $d \in [0..\ell{-}\tau)$, $i + d \in S$ iff $j + d \in S$ (forward synchronized)

# Partitioning sets

A set $S \subseteq [1..n]$ is $\tau$-partitioning for a string $s[1..n]$ if:

(a) if $s[i{-}\tau..i{+}\tau] = s[j{-}\tau..j{+}\tau]$, then $i \in S \Leftrightarrow j \in S$ (locally consistent)


(b) if $s[i..i{+}\ell] = s[j..j{+}\ell]$ for $i, j \in S$, then for each $d \in [0..\ell{-}\tau)$, $i + d \in S$ iff $j + d \in S$ (forward synchronized)

# Partitioning sets

A set $S \subseteq [1..n]$ is $\tau$-partitioning for a string $s[1..n]$ if:

(a) if $s[i{-}\tau..i{+}\tau] = s[j{-}\tau..j{+}\tau]$, then $i \in S \Leftrightarrow j \in S$ (locally consistent)


(b) if $s[i..i{+}\ell] = s[j..j{+}\ell]$ for $i, j \in S$, then for each $d \in [0..\ell{-}\tau)$, $i + d \in S$ iff $j + d \in S$ (forward synchronized)

# Partitioning sets

A set $S \subseteq [1..n]$ is $\tau$-partitioning for a string $s[1..n]$ if:

(a) if $s[i-\tau..i+\tau] = s[j-\tau..j+\tau]$, then $i \in S \Leftrightarrow j \in S$ (locally consistent)


(b) if $s[i..i+\ell] = s[j..j+\ell]$ for $i, j \in S$, then for each $d \in [0..\ell-\tau)$, $i + d \in S$ iff $j + d \in S$ (forward synchronized)


(c) if $i, j \in S \cup \{1, n\}$ with $j-i > \tau$ and $(i..j) \cap S = \emptyset$, then the period of $s[i..j]$ is at most $\tau/4$ (dense)

# Partitioning sets

A set $S \subseteq [1..n]$ is $\tau$-partitioning for a string $s[1..n]$ if:

(a) if $s[i{-}\tau..i{+}\tau] = s[j{-}\tau..j{+}\tau]$, then $i \in S \Leftrightarrow j \in S$ (locally consistent)

$$\cdots\overbrace{\cdots\underset{i}{\bullet}\cdots}\cdots\overbrace{\cdots\underset{j}{\bullet}\cdots}\cdots$$

(b) if $s[i..i{+}\ell] = s[j..j{+}\ell]$ for $i, j \in S$, then for each $d \in [0..\ell{-}\tau)$, $i + d \in S$ iff $j + d \in S$ (forward synchronized)

$$\cdots\overbrace{\underset{i}{\bullet}\cdot\cdot\bullet\cdot\bullet\bullet\cdot\cdot\bullet\cdot\cdot\bullet}\cdots\overbrace{\underset{j}{\bullet}\cdot\cdot\bullet\cdot\bullet\bullet\cdot\cdot\bullet\cdot\cdot\cdot}\cdots$$

(c) if $i, j \in S \cup \{1, n\}$ with $j{-}i > \tau$ and $(i..j) \cap S = \emptyset$, then the period of $s[i..j]$ is at most $\tau/4$ (dense)

$$\cdots\bullet\cdot\bullet\bullet\cdot\bullet\cdot\bullet\bullet\cdot\bullet\overbrace{\bullet\cdot}\overbrace{\cdot\cdot}\overbrace{\cdot\cdot}\overbrace{\cdot\cdot}\overbrace{\cdot\cdot}\overbrace{\cdot\cdot}\overbrace{\cdot\cdot}\bullet\cdot\bullet\bullet\bullet\cdot\cdot\bullet\cdot\bullet\cdot\bullet\bullet\cdot\cdot\bullet$$

# Partitioning sets

A set $S \subseteq [1..n]$ is $\tau$-partitioning for a string $s[1..n]$ if:

(a) if $s[i{-}\tau..i{+}\tau] = s[j{-}\tau..j{+}\tau]$, then $i \in S \Leftrightarrow j \in S$ (locally consistent)


(b) if $s[i..i{+}\ell] = s[j..j{+}\ell]$ for $i, j \in S$, then for each $d \in [0..\ell{-}\tau)$, $i + d \in S$ iff $j + d \in S$ (forward synchronized)


(c) if $i, j \in S \cup \{1, n\}$ with $j{-}i > \tau$ and $(i..j) \cap S = \emptyset$, then the period of $s[i..j]$ is at most $\tau/4$ (dense)

Note: often we have $|S| \geq \Omega(\frac{n}{\tau}) = \Omega(b)$ due to (c)

# Partitioning sets

A set $S \subseteq [1..n]$ is $\tau$-partitioning for a string $s[1..n]$ if:

(a) if $s[i{-}\tau..i{+}\tau] = s[j{-}\tau..j{+}\tau]$, then $i \in S \Leftrightarrow j \in S$ (locally consistent)


(b) if $s[i..i{+}\ell] = s[j..j{+}\ell]$ for $i, j \in S$, then for each $d \in [0..\ell{-}\tau)$, $i + d \in S$ iff $j + d \in S$ (forward synchronized)


(c) if $i, j \in S \cup \{1, n\}$ with $j{-}i > \tau$ and $(i..j) \cap S = \emptyset$, then the period of $s[i..j]$ is at most $\tau/4$ (dense)

Note: often we have $|S| \geq \Omega(\frac{n}{\tau}) = \Omega(b)$ due to (c)

Related: synchronizing sets, minimizers, locally consistent parsing...

# Construction scheme for the $\tau$-partitioning set

# Construction scheme for the $\tau$-partitioning set

1. Produce a $\tau$-partitioning set of size $O(\frac{n}{\tau}\log^* n) = O(b\log^* n)$ using a variant of deterministic coin tossing [Cole, Vishkin 86], [Mehlhorn et al. 97]

# Construction scheme for the $\tau$-partitioning set

1. Produce a $\tau$-partitioning set of size $O(\frac{n}{\tau}\log^* n) = O(b \log^* n)$ using a variant of deterministic coin tossing [Cole, Vishkin 86], [Mehlhorn et al. 97]
2. Store it in $O(\frac{n}{\tau})$ space in a packed form (losing some info)

# Construction scheme for the $\tau$-partitioning set

1. Produce a $\tau$-partitioning set of size $O(\frac{n}{\tau} \log^* n) = O(b \log^* n)$ using a variant of deterministic coin tossing [Cole, Vishkin 86], [Mehlhorn et al. 97]
2. Store it in $O(\frac{n}{\tau})$ space in a packed form (losing some info)
3. Sparsify it to size $O(\frac{n}{\tau})$ using recompression [Jeż 15]

# Construction scheme for the $\tau$-partitioning set

1. Produce a $\tau$-partitioning set of size $O(\frac{n}{\tau} \log^* n) = O(b \log^* n)$ using a variant of deterministic coin tossing [Cole, Vishkin 86], [Mehlhorn et al. 97]
2. Store it in $O(\frac{n}{\tau})$ space in a packed form (losing some info)
3. Sparsify it to size $O(\frac{n}{\tau})$ using recompression [Jeż 15]
4. Run 1 again retaining only positions remaining after the sparsification

# Construction scheme for the $\tau$-partitioning set

1. Produce a $\tau$-partitioning set of size $O(\frac{n}{\tau}\log^* n) = O(b\log^* n)$ using a variant of deterministic coin tossing [Cole, Vishkin 86], [Mehlhorn et al. 97]
2. Store it in $O(\frac{n}{\tau})$ space in a packed form (losing some info)
3. Sparsify it to size $O(\frac{n}{\tau})$ using recompression [Jeż 15]
4. Run 1 again retaining only positions remaining after the sparsification

The devil is in details

# Deterministic coin tossing

# Deterministic coin tossing

Iteratively, for $k = 0, \dots, \log \frac{\tau}{\log^* n}$, given a $(2^k \log^* n)$-partitioning set $S_k$ of size $O(\frac{n}{2^k})$, sparsify it to make a $(2^{k+1} \log^* n)$-partitioning set $S_{k+1} \subseteq S_k$ as follows (initial $S_0 = \{1, \dots, n\}$):

# Deterministic coin tossing

Iteratively, for $k = 0, \dots, \log \frac{\tau}{\log^* n}$, given a $(2^k \log^* n)$-partitioning set $S_k$ of size $O(\frac{n}{2^k})$, sparsify it to make a $(2^{k+1} \log^* n)$-partitioning set $S_{k+1} \subseteq S_k$ as follows (initial $S_0 = \{1, \dots, n\}$):

- Let $S_k = \{j_1 < \dots < j_{|S_k|}\}$. For $j_h \in S_k$, assign BIG number $v_h$ whose bit representation is the bit string $s[j_h \mathinner{..} j_h{+}2^{k+1}]$, interpreting letters $s[j_h], s[j_h{+}1], \dots$ as $O(\log n)$-bit sequences

# Deterministic coin tossing

Iteratively, for $k = 0, \ldots, \log \frac{\tau}{\log^* n}$, given a $(2^k \log^* n)$-partitioning set $S_k$ of size $O(\frac{n}{2^k})$, sparsify it to make a $(2^{k+1} \log^* n)$-partitioning set $S_{k+1} \subseteq S_k$ as follows (initial $S_0 = \{1, \ldots, n\}$):

- Let $S_k = \{j_1 < \cdots < j_{|S_k|}\}$. For $j_h \in S_k$, assign BIG number $v_h$ whose bit representation is the bit string $s[j_h \mathinner{..} j_h{+}2^{k+1}]$, interpreting letters $s[j_h], s[j_h{+}1], \ldots$ as $O(\log n)$-bit sequences
- Given $j_h \in S_k$:

# Deterministic coin tossing

Iteratively, for $k = 0, \dots, \log \frac{\tau}{\log^* n}$, given a $(2^k \log^* n)$-partitioning set $S_k$ of size $O(\frac{n}{2^k})$, sparsify it to make a $(2^{k+1} \log^* n)$-partitioning set $S_{k+1} \subseteq S_k$ as follows (initial $S_0 = \{1, \dots, n\}$):

- Let $S_k = \{j_1 < \cdots < j_{|S_k|}\}$. For $j_h \in S_k$, assign BIG number $v_h$ whose bit representation is the bit string $s[j_h \mathinner{..} j_h{+}2^{k+1}]$, interpreting letters $s[j_h], s[j_h{+}1], \dots$ as $O(\log n)$-bit sequences
- Given $j_h \in S_k$:
  - if $j_h - j_{h-1} > 2^k$ or $v_{h-1} = v_h$ or $v_{h-1} = \infty$, assign $v_h = \infty$

# Deterministic coin tossing

Iteratively, for $k = 0, \dots, \log \frac{\tau}{\log^* n}$, given a $(2^k \log^* n)$-partitioning set $S_k$ of size $O(\frac{n}{2^k})$, sparsify it to make a $(2^{k+1} \log^* n)$-partitioning set $S_{k+1} \subseteq S_k$ as follows (initial $S_0 = \{1, \dots, n\}$):

- Let $S_k = \{j_1 < \dots < j_{|S_k|}\}$. For $j_h \in S_k$, assign BIG number $v_h$ whose bit representation is the bit string $s[j_h \mathinner{..} j_h{+}2^{k+1}]$, interpreting letters $s[j_h], s[j_h{+}1], \dots$ as $O(\log n)$-bit sequences
- Given $j_h \in S_k$:
  - if $j_h - j_{h-1} > 2^k$ or $v_{h-1} = v_h$ or $v_{h-1} = \infty$, assign $v_h = \infty$
  - otherwise, assign $v_h = \text{vbit}(v_{h-1}, v_h)$ where vbit is the Vishkin–Cole magic reducing the bit length logarithmically

# Deterministic coin tossing

Iteratively, for $k = 0, \dots, \log \frac{\tau}{\log^* n}$, given a $(2^k \log^* n)$-partitioning set $S_k$ of size $O(\frac{n}{2^k})$, sparsify it to make a $(2^{k+1} \log^* n)$-partitioning set $S_{k+1} \subseteq S_k$ as follows (initial $S_0 = \{1, \dots, n\}$):

- Let $S_k = \{j_1 < \dots < j_{|S_k|}\}$. For $j_h \in S_k$, assign BIG number $v_h$ whose bit representation is the bit string $s[j_h \mathinner{.\,.} j_h{+}2^{k+1}]$, interpreting letters $s[j_h], s[j_h{+}1], \dots$ as $O(\log n)$-bit sequences
- Given $j_h \in S_k$:
  - if $j_h - j_{h-1} > 2^k$ or $v_{h-1} = v_h$ or $v_{h-1} = \infty$, assign $v_h = \infty$
  - otherwise, assign $v_h = \mathrm{vbit}(v_{h-1}, v_h)$ where vbit is the Vishkin–Cole magic reducing the bit length logarithmically

# Deterministic coin tossing

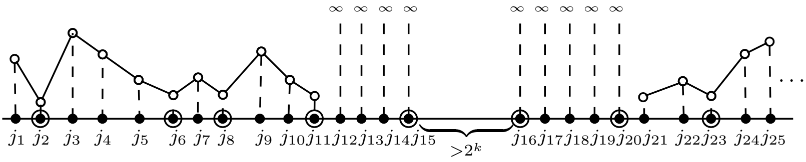
Iteratively, for $k = 0, \dots, \log \frac{\tau}{\log^* n}$, given a $(2^k \log^* n)$-partitioning set $S_k$ of size $O(\frac{n}{2^k})$, sparsify it to make a $(2^{k+1} \log^* n)$-partitioning set $S_{k+1} \subseteq S_k$ as follows (initial $S_0 = \{1, \dots, n\}$):

- Let $S_k = \{j_1 < \cdots < j_{|S_k|}\}$. For $j_h \in S_k$, assign BIG number $v_h$ whose bit representation is the bit string $s[j_h \mathinner{..} j_h{+}2^{k+1}]$, interpreting letters $s[j_h], s[j_h{+}1], \dots$ as $O(\log n)$-bit sequences
- Given $j_h \in S_k$:
  - if $j_h - j_{h-1} > 2^k$ or $v_{h-1} = v_h$ or $v_{h-1} = \infty$, assign $v_h = \infty$
  - otherwise, assign $v_h = \text{vbit}(v_{h-1}, v_h)$ where vbit is the Vishkin–Cole magic reducing the bit length logarithmically
- Do $O(\log^* n)$ reductions until $v_h$ is $O(1)$ or $\infty$ for all $h$

# Deterministic coin tossing

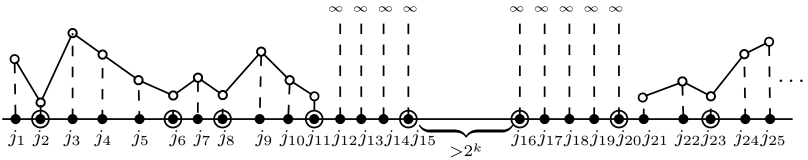
# Deterministic coin tossing

Put into $S_{k+1}$ all $j_h$ such that $j_h - j_{h-1} > 2^k$ or $j_{h+1} - j_h > 2^k$ or $\infty > v_{h-1} > v_h < v_{h+1}$ (local minima $v_h$)



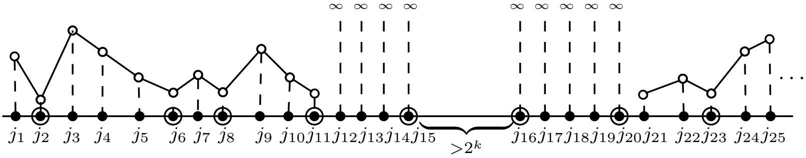
# Deterministic coin tossing

Put into $S_{k+1}$ all $j_h$ such that $j_h - j_{h-1} > 2^k$ or $j_{h+1} - j_h > 2^k$ or $\infty > v_{h-1} > v_h < v_{h+1}$ (local minima $v_h$)




(Exact conditions are more complicated...)

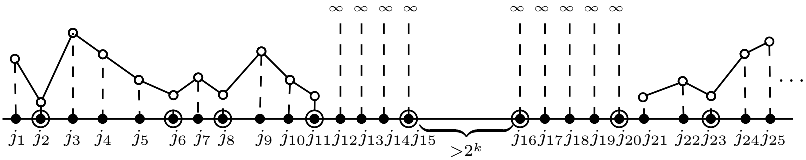
Put into $S_{k+1}$ all $j_h$ such that $j_h - j_{h-1} > 2^k$ or $j_{h+1} - j_h > 2^k$ or $\infty > v_{h-1} > v_h < v_{h+1}$ (local minima $v_h$)




(Exact conditions are more complicated...)

# Deterministic coin tossing

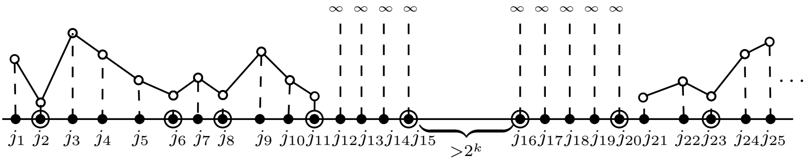
Put into $S_{k+1}$ all $j_h$ such that $j_h - j_{h-1} > 2^k$ or $j_{h+1} - j_h > 2^k$ or $\infty > v_{h-1} > v_h < v_{h+1}$ (local minima $v_h$)




(Exact conditions are more complicated...)
How to store the sets $S_k$?

# Deterministic coin tossing

Put into $S_{k+1}$ all $j_h$ such that $j_h - j_{h-1} > 2^k$ or $j_{h+1} - j_h > 2^k$ or $\infty > v_{h-1} > v_h < v_{h+1}$ (local minima $v_h$)

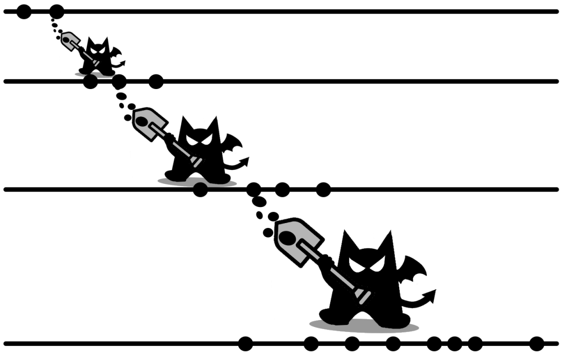



(Exact conditions are more complicated...)
How to store the sets $S_k$? Do we have to?

# Deterministic coin tossing

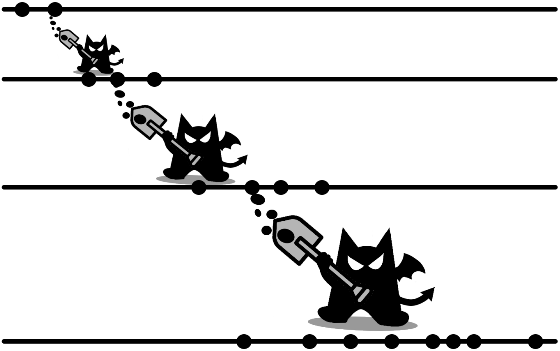
# Deterministic coin tossing

The decision to put $j \in S_k$ into $S_{k+1}$ is "local". We process $S_k$ left-to-right and feed the result to the same procedure processing $S_{k+1}$ left-to-right. The "cascade" of procedures feeding each other:



# Deterministic coin tossing

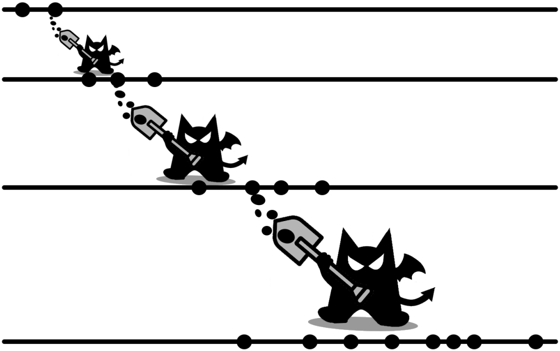
The decision to put $j \in S_k$ into $S_{k+1}$ is "local". We process $S_k$ left-to-right and feed the result to the same procedure processing $S_{k+1}$ left-to-right. The "cascade" of procedures feeding each other:



On the way, we build LCE indexes and SSTs for Vishkin–Cole magic

## Deterministic coin tossing

The decision to put $j \in S_k$ into $S_{k+1}$ is "local". We process $S_k$ left-to-right and feed the result to the same procedure processing $S_{k+1}$ left-to-right. The "cascade" of procedures feeding each other:



On the way, we build LCE indexes and SSTs for Vishkin–Cole magic

The last level receives a $\tau$-partitioning set of size $O(\frac{n}{\tau} \log^* n)$

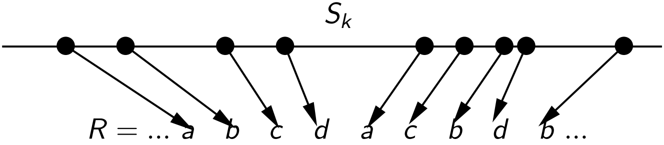
# To recompression

# To recompression

The resulting set $S$ of size $O(\frac{n}{\tau}\log^* n)$ cannot be stored. Instead we make a string $R$ of length $|S|$ over a small alphabet which can be stored in $O(\frac{n}{\tau})$ machine words, such that any two letters of $R$ corresponding to positions of $S$ at a distance $\le \frac{\tau}{2}$ are distinct.

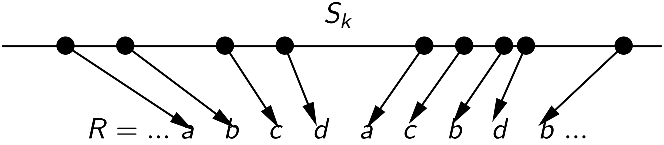
# To recompression

The resulting set $S$ of size $O(\frac{n}{\tau} \log^* n)$ cannot be stored. Instead we make a string $R$ of length $|S|$ over a small alphabet which can be stored in $O(\frac{n}{\tau})$ machine words, such that any two letters of $R$ corresponding to positions of $S$ at a distance $\le \frac{\tau}{2}$ are distinct.



# To recompression

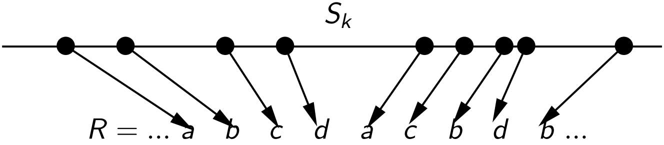
The resulting set $S$ of size $O(\frac{n}{\tau}\log^* n)$ cannot be stored. Instead we make a string $R$ of length $|S|$ over a small alphabet which can be stored in $O(\frac{n}{\tau})$ machine words, such that any two letters of $R$ corresponding to positions of $S$ at a distance $\leq\frac{\tau}{2}$ are distinct.




How the letters of $R$ are constructed?

# To recompression

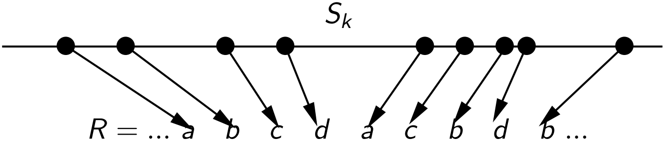
The resulting set $S$ of size $O(\frac{n}{\tau}\log^* n)$ cannot be stored. Instead we make a string $R$ of length $|S|$ over a small alphabet which can be stored in $O(\frac{n}{\tau})$ machine words, such that any two letters of $R$ corresponding to positions of $S$ at a distance $\leq\frac{\tau}{2}$ are distinct.




How the letters of $R$ are constructed?
Simple: interwined magic cascade of Vishkin–Cole magic!

# To recompression

The resulting set $S$ of size $O(\frac{n}{\tau}\log^* n)$ cannot be stored. Instead we make a string $R$ of length $|S|$ over a small alphabet which can be stored in $O(\frac{n}{\tau})$ machine words, such that any two letters of $R$ corresponding to positions of $S$ at a distance $\leq \frac{\tau}{2}$ are distinct.




How the letters of $R$ are constructed?
Simple: interwined magic cascade of Vishkin–Cole magic!
We store separately the approximate info about distances between positions of $S$ sufficient to determine if $|j - j'| < \frac{\tau}{2}$ for $j, j' \in S$

# Recompression

# Recompression

- Collect statistics of occurrences of pairs $R[i], R[i+1]$

# Recompression

- Collect statistics of occurrences of pairs $R[i], R[i+1]$
- Mark alphabet letters with 0 and 1 so that the number of pairs $R[i], R[i+1]$ marked $0, 1$, respectively, is at least $\frac{1}{4}|R|$

# Recompression

- Collect statistics of occurrences of pairs $R[i], R[i+1]$
- Mark alphabet letters with 0 and 1 so that the number of pairs $R[i], R[i+1]$ marked $0, 1$, respectively, is at least $\frac{1}{4}|R|$
- For each such pair, remove $R[i]$ from $R$ if the distance from the positions of $S$ corresponding to $R[i]$ and $R[i+1]$ is $\leq \frac{\tau}{2}$

# Recompression

- Collect statistics of occurrences of pairs $R[i], R[i{+}1]$
- Mark alphabet letters with 0 and 1 so that the number of pairs $R[i], R[i{+}1]$ marked $0, 1$, respectively, is at least $\frac{1}{4}|R|$
- For each such pair, remove $R[i]$ from $R$ if the distance from the positions of $S$ corresponding to $R[i]$ and $R[i{+}1]$ is $\leq\frac{\tau}{2}$
- Do this until $|R| \leq O(\frac{n}{\tau})$

# Recompression

- Collect statistics of occurrences of pairs $R[i], R[i{+}1]$
- Mark alphabet letters with 0 and 1 so that the number of pairs $R[i], R[i{+}1]$ marked $0, 1$, respectively, is at least $\frac{1}{4}|R|$
- For each such pair, remove $R[i]$ from $R$ if the distance from the positions of $S$ corresponding to $R[i]$ and $R[i{+}1]$ is $\leq \frac{\tau}{2}$
- Do this until $|R| \leq O(\frac{n}{\tau})$

a b c d a c b d b a c d b a

# Recompression

- Collect statistics of occurrences of pairs $R[i], R[i{+}1]$
- Mark alphabet letters with 0 and 1 so that the number of pairs $R[i], R[i{+}1]$ marked $0, 1$, respectively, is at least $\frac{1}{4}|R|$
- For each such pair, remove $R[i]$ from $R$ if the distance from the positions of $S$ corresponding to $R[i]$ and $R[i{+}1]$ is $\le \frac{\tau}{2}$
- Do this until $|R| \le O(\frac{n}{\tau})$

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| a | b | c | d | a | c | b | d | b | a | c | d | b | a |

# Recompression

- Collect statistics of occurrences of pairs $R[i], R[i+1]$
- Mark alphabet letters with 0 and 1 so that the number of pairs $R[i], R[i+1]$ marked $0, 1$, respectively, is at least $\frac{1}{4}|R|$
- For each such pair, remove $R[i]$ from $R$ if the distance from the positions of $S$ corresponding to $R[i]$ and $R[i+1]$ is $\leq \frac{\tau}{2}$
- Do this until $|R| \leq O(\frac{n}{\tau})$

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| a | b | c | d | a | c | b | d | b | a | c | d | b | a |

# Recompression

- Collect statistics of occurrences of pairs $R[i], R[i{+}1]$
- Mark alphabet letters with 0 and 1 so that the number of pairs $R[i], R[i{+}1]$ marked $0, 1$, respectively, is at least $\frac{1}{4}|R|$
- For each such pair, remove $R[i]$ from $R$ if the distance from the positions of $S$ corresponding to $R[i]$ and $R[i{+}1]$ is $\leq \frac{\tau}{2}$
- Do this until $|R| \leq O(\frac{n}{\tau})$

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| a | b | c | d | a | c | b | d | b | a | c | d | b | a |

a b c d a c b d b a c d b a

# Recompression

- Collect statistics of occurrences of pairs $R[i], R[i+1]$
- Mark alphabet letters with 0 and 1 so that the number of pairs $R[i], R[i+1]$ marked $0, 1$, respectively, is at least $\frac{1}{4}|R|$
- For each such pair, remove $R[i]$ from $R$ if the distance from the positions of $S$ corresponding to $R[i]$ and $R[i+1]$ is $\leq \frac{\tau}{2}$
- Do this until $|R| \leq O(\frac{n}{\tau})$

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| a | b | c | d | a | c | b | d | b | a | c | d | b | a |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | | 1 | 0 | | 0 | 1 | 1 | 0 | | 1 | 0 | 0 |
| a | b | c | d | a | c | b | d | b | a | c | d | b | a |

# Recompression

- Collect statistics of occurrences of pairs $R[i], R[i{+}1]$
- Mark alphabet letters with 0 and 1 so that the number of pairs $R[i], R[i{+}1]$ marked $0, 1$, respectively, is at least $\frac{1}{4}|R|$
- For each such pair, remove $R[i]$ from $R$ if the distance from the positions of $S$ corresponding to $R[i]$ and $R[i{+}1]$ is $\leq \frac{\tau}{2}$
- Do this until $|R| \leq O(\frac{n}{\tau})$

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| a | b | c | d | a | c | b | d | b | a | c | d | b | a |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | | 1 | 0 | | 0 | 1 | 1 | 0 | | 1 | 0 | 0 |
| a | b | c | d | a | c | b | d | b | a | c | d | b | a |

# Recompression

- Collect statistics of occurrences of pairs $R[i], R[i{+}1]$
- Mark alphabet letters with 0 and 1 so that the number of pairs $R[i], R[i{+}1]$ marked $0, 1$, respectively, is at least $\frac{1}{4}|R|$
- For each such pair, remove $R[i]$ from $R$ if the distance from the positions of $S$ corresponding to $R[i]$ and $R[i{+}1]$ is $\leq \frac{\tau}{2}$
- Do this until $|R| \leq O(\frac{n}{\tau})$

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| a | b | c | d | a | c | b | d | b | a | c | d | b | a |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | | 1 | 0 | | 0 | 1 | 1 | 0 | | 1 | 0 | 0 |
| a | b | c | d | a | c | b | d | b | a | c | d | b | a |

Generate the set $S$ using Vishkin–Cole again, retaining only those positions that correspond to remaining letters of $R$

# Thank you for your attention!