Recap: Suffix Array and LCP-Array

**Definition: Suffix Array [GBS92; MM93]**

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

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

**Definition: Longest Common Prefix Array**

Given a text $T$ of length $n$ and its SA, the **LCP-array** is defined as

$$LCP[i] = \begin{cases} 
0 & i = 1 \\
\max\{\ell : T[SA[i]..SA[i] + \ell) = T[SA[i - 1]..SA[i - 1] + \ell) \} & i \neq 1
\end{cases}$$
Timeline Sequential Suffix Sorting

- based on [Bah+19; Bin18; Kur20; PST07]
- darker grey: linear running time
- brown: available implementation

Prefix Doubling
- [MM] original

Induced Copying
- [LS] qsufsort
- [Sew] 1/2 copy
- [MF] deep-shallow
- [SS] bpr
- [MP] cache aware
- [Bai] GSACA
- [LLH] O(1) space

Recursion
- [IT] A/B copy
- [IT] L/S split
- [Man] chains
- [NA] succinct
- [AN] SFE-coding
- [NS] DSufSort
- [SS] bpr
- [MF] deep-shallow
- [IT] A/B copy
- [F] O(n) tree
- [BK] diffcover
- [KA] L/S split
- [Na] succinct
- [AN] SFE-coding
- [MK] fixed Σ
- [SS] bpr
- [MP] cache aware
- [Na] succinct
- [AN] SFE-coding
- [MB] cache aware
- [SS] bpr
- [MF] deep-shallow
- [IT] A/B copy
- [F] O(n) tree
- [BK] diffcover
- [KA] L/S split
- [Na] succinct
- [AN] SFE-coding
- [MB] cache aware
- [SS] bpr
- [MF] deep-shallow
- [IT] A/B copy
- [F] O(n) tree
- [BK] diffcover
- [KA] L/S split
- [Na] succinct
- [AN] SFE-coding
- [MB] cache aware
- [SS] bpr
- [MF] deep-shallow
- [IT] A/B copy
- [F] O(n) tree
- [BK] diffcover
- [KA] L/S split
- [Na] succinct
- [AN] SFE-coding
- [MB] cache aware
- [SS] bpr
- [MF] deep-shallow
- [IT] A/B copy
- [F] O(n) tree
- [BK] diffcover
- [KA] L/S split
- [Na] succinct
- [AN] SFE-coding
- [MB] cache aware
- [SS] bpr
- [MF] deep-shallow
- [IT] A/B copy
- [F] O(n) tree
- [BK] diffcover
- [KA] L/S split
- [Na] succinct
- [AN] SFE-coding
- [MB] cache aware
- [SS] bpr
Timeline Sequential Suffix Sorting
- based on [Bah+19; Bin18; Kur20; PST07]
- darker grey: linear running time
- brown: available implementation

Special Mentions
- DC3 first $O(n)$ algorithm
- $O(n)$ running time and $O(1)$ space for integer alphabets possible
Timeline Sequential Suffix Sorting

- based on [Bah+19; Bin18; Kur20; PST07]
- darker grey: linear running time
- brown: available implementation

Special Mentions

- DC3 first $O(n)$ algorithm
- $O(n)$ running time and $O(1)$ space for integer alphabets possible
- until 2021: DivSufSort fastest in practice with $O(n \lg n)$ running time
Timeline Sequential Suffix Sorting

- based on [Bah+19; Bin18; Kur20; PST07]
- darker grey: linear running time
- brown: available implementation

Special Mentions

- DC3 first $O(n)$ algorithm
- $O(n)$ running time and $O(1)$ space for integer alphabets possible
- until 2021: DivSufSort fastest in practice with $O(n \lg n)$ running time
- since 2021: libSAIS fastest in practice with $O(n)$ running time
External and Distributed Memory

**External Memory**
- internal memory of size $M$ words
- external memory of unlimited size
- transfer of blocks of size $B$ words

- scanning $N$ elements: $\Theta\left(\frac{N}{B}\right)$
- sorting $N$ elements: $\Theta\left(\frac{N}{B} \log_{\frac{B}{M}} \frac{N}{B}\right)$

**semi-external memory**
### External and Distributed Memory

#### External Memory
- Internal memory of size $M$ words
- External memory of unlimited size
- Transfer of blocks of size $B$ words

- Scanning $N$ elements: $\Theta\left(\frac{N}{B}\right)$
- Sorting $N$ elements: $\Theta\left(\frac{N}{B} \log \frac{M}{B} \frac{N}{B}\right)$

- Semi-external memory

#### Distributed Memory
- $p$ PEs with internal memory
- Communication between PEs over network

- Bulk-synchronous parallel model [Val90]
- Supersteps: local work, communication, synchronization
Challenges for Suffix Array Construction

Distributed Memory
- suffixes span over whole input ⊙ no locality
- comparing suffixes requires text access ⊙ random access

External Memory

main memory \rightarrow B \rightarrow external memory

PE 1 \rightarrow \text{memory} \rightarrow \text{memory} \rightarrow \text{memory} \rightarrow \text{memory} \rightarrow PE 2 \rightarrow \text{memory} \rightarrow \text{memory} \rightarrow \text{memory} \rightarrow \text{memory} \rightarrow PE 3 \rightarrow \text{memory} \rightarrow \text{memory} \rightarrow \text{memory} \rightarrow \text{memory} \rightarrow \cdots \rightarrow PE p \rightarrow \text{memory} \rightarrow \text{memory} \rightarrow \text{memory} \rightarrow \text{memory}
Challenges for Suffix Array Construction

**Distributed Memory**
- suffixes span over whole input \( \# \) no locality
- comparing suffixes requires text access
  \( \# \) random access
- random access expensive in both models
- whole suffix not available locally in distributed memory

**External Memory**
- PE 1
- PE 2
- PE 3
- ... PE \( p \)
Challenges for Suffix Array Construction

**Distributed Memory**
- suffixes span over whole input 払い no locality
- comparing suffixes requires text access 払い random access

- random access expensive in both models
- whole suffix not available locally in distributed memory

- express suffix array construction algorithm using
  - scanning
  - sorting
  - merging
Prefix-Doubling

Induced-Copying

Recursion

2003

[KSB] DC3

2012

[B] DC3/7/13

2014

[AKA] cloudSACA

2015

[FA] PSAC

Distributed Memory

PE 1

PE 2

PE 3

... PE p

Speicher

Speicher

Speicher

Speicher
 Prefix-Doubling | Induced-Copying | Recursion

- 2003
- 2012
- 2014
- 2015
- 2018

[AKA] cloudSACA
[FA] PSAC
[BGK] Doubling

[KSB] DC3
[B] DC3/7/13
[BGK] DC3/7/13

Distributed Memory

PE 1 → Speicher
PE 2 → Speicher
PE 3 → Speicher
... → Speicher
PE p → Speicher
**h-Order, h-Groups, and h-Ranks**

**Definition: h-Order**

- **h-Order:**
  \[ T[i..n] \leq_h T[j..n] \iff T[i..i+h] \leq T[j..j+h] \]

- **SA_h** is the suffix array of all suffixes ordered by 
  **h-order** not unambiguously
**h-Order, h-Groups, and h-Ranks**

**Definition: h-Order**
- **h-Order:**
  \[ T[i..n] \leq_h T[j..n] \iff T[i..i+h] \leq T[j..j+h] \]
- \( SA_h \) is the suffix array of all suffixes ordered by \( h \)-order; not unambiguously

**Definition: h-Ranks and h-Groups**
- all suffixes that are equal w.r.t. an \( h \)-order are in an **h-group**
- **h-rank**: number of lexicographically smaller \( h \)-groups plus one
**h-Order, h-Groups, and h-Ranks**

**Definition: h-Order**
- **h-Order**:
  \[ T[i..n] \leq_h T[j..n] \iff T[i..i+h] \leq T[j..j+h] \]
- \( SA_h \) is the suffix array of all suffixes ordered by \( h \)-order not unambiguously

**Definition: h-Ranks and h-Groups**
- All suffixes that are equal w.r.t. an \( h \)-order are in an \( h \)-group
- **h-rank**: number of lexicographically smaller \( h \)-groups plus one

\[
\begin{align*}
mississippi & $ \\
ississippi & $ \\
ssissippi & $ \\
ssissippi & $ \\
ississippi & $ \\
ississippi & $ \\
issippi & $ \\
issippi & $ \\
issippi & $ \\
issippi & $ \\
issippi & $ \\
i & $ \\
$ \\
\end{align*}
\]
**h-Order, h-Groups, and h-Ranks**

**Definition: h-Order**
- **h-Order**:
  \[ T[i..n] \leq_h T[j..n] \iff T[i..i+h] \leq T[j..j+h] \]
- **SA_h** is the suffix array of all suffixes ordered by h-order not unambiguously

**Definition: h-Ranks and h-Groups**
- all suffixes that are equal w.r.t. an h-order are in an **h-group**
- **h-rank**: number of lexicographically smaller h-groups plus one
**h-Order, h-Groups, and h-Ranks**

**Definition: h-Order**

- **h-Order:**
  \[ T[i..n] \leq_h T[j..n] \iff T[i..i+h] \leq T[j..j+h] \]

- **SA_h** is the suffix array of all suffixes ordered by h-order not unambiguously

**Definition: h-Ranks and h-Groups**

- All suffixes that are equal w.r.t. an h-order are in an **h-group**

- **h-rank:** number of lexicographically smaller h-groups plus one

---

**Example:**

\[
\begin{align*}
&\text{mississippi} \\
&\text{isissippi} \\
&\text{ssissippi} \\
&\text{sissippi} \\
&\text{isip} \\
&\text{spi} \\
&\text{p} \\
&\text{i} \\
&\$
\end{align*}
\]
Prefix-Doubling: The Idea

- 1-rank is the first character
Prefix-Doubling: The Idea

- 1-rank is the first character
- 2-rank can be computed from first 2 characters
Prefix-Doubling: The Idea

- 1-rank is the first character
- 2-rank can be computed from first 2 characters
- 3-rank can be computed from first 3 characters
Prefix-Doubling: The Idea

- 1-rank is the first character
- 2-rank can be computed from first 2 characters
- 3-rank can be computed from first 3 characters
- 4-rank can be computed from first 4 characters
Prefix-Doubling: The Idea

- 1-rank is the first character
- 2-rank can be computed from first 2 characters
- 3-rank can be computed from first 3 characters
- 4-rank can be computed from first 4 characters
- 4-rank can be computed from two 2-ranks
Prefix-Doubling: The Idea

- 1-rank is the first character
- 2-rank can be computed from first 2 characters
- 3-rank can be computed from first 3 characters
- 4-rank can be computed from first 4 characters
- 4-rank can be computed from two 2-ranks

- compute $2^{k+1}$-ranks using $2^k$-ranks
Prefix-Doubling: Example

1. initial rank is $T[i] \odot 1$-rank
2. for $k = 0$ to $\lceil \lg n \rceil$
3. new $2^{k+1}$-ranks based on 
   \[ ISA_{2^k}[i] \& ISA_{2^k}[i + 2^k] \]
4. if all ranks are unique, break
5. compute SA from ISA
Prefix-Doubling: Example

1. initial rank is $T[i] \circ 1$-rank
2. for $k = 0$ to $\lceil \log n \rceil$
3. new $2^{k+1}$-ranks based on $ISA_{2^k}[i] \& ISA_{2^k}[i + 2^k]$
4. if all ranks are unique, break
5. compute $SA$ from $ISA$
Prefix-Doubling: Example

1. initial rank is $T[i]$ 1-rank
2. for $k = 0$ to $\lceil \lg n \rceil$
3. new $2^{k+1}$-ranks based on $ISA_{2^k}[i] \& ISA_{2^k}[i + 2^k]$
4. if all ranks are unique, break
5. compute $SA$ from $ISA$

```
2   m i s s i s s i p p i $
1   i s s i s s i p p i$
4   s s i s s i p p i$
4   s i s s i p p i$
1   i s s s i p p i$
4   s s i p p i$
4   s i p p i$
1   i p p i$
3   p p i$
3   p i$
1   i$
0   $
```
Prefix-Doubling: Example

1. initial rank is $T[i]$ 1-rank
2. for $k = 0$ to $\lceil \lg n \rceil$
3. new $2^{k+1}$-ranks based on $ISA_{2^k}[i] \& ISA_{2^k}[i + 2^k]$
4. if all ranks are unique, break
5. compute $SA$ from $ISA$
Prefix-Doubling: Example

1. initial rank is $T[i] \oplus 1$-rank
2. for $k = 0$ to $\lceil \lg n \rceil$
3. new $2^{k+1}$-ranks based on $ISA_{2^k}[i] \& ISA_{2^k}[i + 2^k]$
4. if all ranks are unique, break
5. compute $SA$ from $ISA$
Prefix-Doubling: Example

1. initial rank is $T[i]$ 1-rank
2. for $k = 0$ to $\lceil \lg n \rceil$
3. new $2^{k+1}$-ranks based on
   $$ISA_{2^k}[i] \& ISA_{2^k}[i + 2^k]$$
4. if all ranks are unique, break
5. compute $SA$ from $ISA$
Prefix-Doubling: Example

1. initial rank is $T[i] \overset{1}{\overset{1}{\circ}} 1$-rank
2. for $k = 0$ to $\lceil \lg n \rceil$
3. new $2^{k+1}$-ranks based on
   \[ ISA_{2^k}[i] \text{ and } ISA_{2^k}[i + 2^k] \]
4. if all ranks are unique, break
5. compute $SA$ from $ISA$
Prefix-Doubling: Example

1. initial rank is $T[i]$ 1-rank
2. for $k = 0$ to $\lceil \lg n \rceil$
3. new $2^{k+1}$-ranks based on
   \[ ISA_{2^k}[i] \& ISA_{2^k}[i + 2^k] \]
4. if all ranks are unique, break
5. compute $SA$ from $ISA$
Prefix-Doubling: Example

1. initial rank is \( T[i] \) \( 1 \)-rank
2. for \( k = 0 \) to \( \lceil \lg n \rceil \)
3. new \( 2^{k+1} \)-ranks based on
   \[ ISA_{2^k}[i] \& ISA_{2^k}[i + 2^k] \]
4. if all ranks are unique, break
5. compute SA from ISA
Prefix-Doubling: Example

1. initial rank is $T[i] \odot 1$-rank
2. for $k = 0$ to $\lceil \lg n \rceil$
3. new $2^{k+1}$-ranks based on
   
   $ISA_{2^k}[i] \land ISA_{2^k}[i + 2^k]$

4. if all ranks are unique, break
5. compute $SA$ from $ISA$
Prefix-Doubling: Example

1. initial rank is \( T[i] \) 1-rank
2. for \( k = 0 \) to \( \lceil \lg n \rceil \)
3. new \( 2^{k+1} \)-ranks based on \( ISA_{2^k}[i] \) & \( ISA_{2^k}[i + 2^k] \)
4. if all ranks are unique, break
5. compute \( SA \) from \( ISA \)
Prefix-Doubling: Example

1. initial rank is \( T[i] \) 1-rank
2. for \( k = 0 \) to \( \lceil \lg n \rceil \)
3. new \( 2^{k+1} \)-ranks based on 
   \[ ISA_{2^k}[i] \) & \( ISA_{2^k}[i + 2^k] \)
4. if all ranks are unique, break
5. compute \( SA \) from \( ISA \)
Prefix-Doubling: Example

1. initial rank is $T[i]$ 1-rank
2. for $k = 0$ to $\lceil \lg n \rceil$
3. new $2^{k+1}$-ranks based on $ISA_{2^k}[i] \& ISA_{2^k}[i + 2^k]$
4. if all ranks are unique, break
5. compute $SA$ from $ISA$
Prefix-Doubling: Example

1. initial rank is $T[i] \overset{1}{\circ} 1$-rank
2. for $k = 0$ to $\lceil \lg n \rceil$
3. new $2^{k+1}$-ranks based on $ISA_{2^k}[i] \& ISA_{2^k}[i + 2^k]$
4. if all ranks are unique, break
5. compute $SA$ from $ISA$
Prefix-Doubling: Example

1. initial rank is $T[i] \oplus 1$-rank
2. for $k = 0$ to $\lceil \lg n \rceil$
3. new $2^{k+1}$-ranks based on $ISA_{2^k}[i] \& ISA_{2^k}[i + 2^k]$
4. if all ranks are unique, break
5. compute SA from ISA
Prefix-Doubling: Example

1. initial rank is $T[i]$ 1-rank
2. for $k = 0$ to $\lceil \lg n \rceil$
3. new $2^{k+1}$-ranks based on $ISA_{2^k}[i] \& ISA_{2^k}[i + 2^k]$
4. if all ranks are unique, break
5. compute SA from ISA
Prefix-Doubling: Example

1. initial rank is $T[i]$ 1-rank
2. for $k = 0$ to $\lceil \lg n \rceil$
3. new $2^{k+1}$-ranks based on $ISA_{2^k}[i] \& ISA_{2^k}[i + 2^k]$
4. if all ranks are unique, break
5. compute SA from ISA

Simple Algorithm

Prefix-Doubling: Practical Approaches

Use $ISA_h$ [FA15]

- use $ISA_{2^k}$ to compute rank tuples
- for position $i$ use rank $ISA_{2^k}[i + 2^k]$
- if $i + 2^k > n$, second rank is 0
- example on the board 📚
**Prefix-Doubling: Practical Approaches**

<table>
<thead>
<tr>
<th>Use $ISA_h$ [FA15]</th>
<th>Sort by Text Positions [Dem+08; FK19]</th>
</tr>
</thead>
<tbody>
<tr>
<td>- use $ISA_{2^k}$ to compute rank tuples</td>
<td></td>
</tr>
<tr>
<td>- for position $i$ use rank $ISA_{2^k}[i + 2^k]$</td>
<td></td>
</tr>
<tr>
<td>- if $i + 2^k &gt; n$, second rank is 0</td>
<td></td>
</tr>
<tr>
<td>- example on the board 📚</td>
<td></td>
</tr>
<tr>
<td>- especially good if access to $ISA_h$ is expensive</td>
<td></td>
</tr>
<tr>
<td>- sort tuples (Textposition $i$, Rang $r$)</td>
<td></td>
</tr>
<tr>
<td>- using $(i, r) \leq (j, r')$ iff</td>
<td></td>
</tr>
<tr>
<td>$$(i \mod 2^k, \lfloor i/2^k \rfloor) &lt; (j \mod 2^k, \lfloor j/2^k \rfloor)$$</td>
<td></td>
</tr>
<tr>
<td>- example on the board 📚</td>
<td></td>
</tr>
</tbody>
</table>
Prefix-Doubling: Running Time

- running time: $O(n \lg n)$
- memory requirements: $8n(\pm n)$ words \footnote{for texts \leq 4 \text{GiB}}
- worst-case input: $T = a^{n-1}$
Prefix-Doubling: Running Time

- running time: $O(n \log n)$
- memory requirements: $8n(+n)$ words for texts $\leq 4\ \text{GiB}$
- worst-case input: $T = a^{n-1}$

Generalization

- more than doubling is possible
- compute $\alpha^{k+1}$-ranks using $\alpha\ \alpha^k$-ranks
- can save I/Os in EM $\alpha = 4$ requires 30 \% less I/Os than $\alpha = 2$ [Dem+08]
Prefix Doubling: Experimental Results [Kur20]

**Commoncrawl throughput (MiB/s)**
- 512 MiB per PE
- 1024 MiB per PE
- 1536 MiB per PE

**Commoncrawl construction space (B/n)**
- PEs (20 threads)

Legend:
- pDivSufSort
- pPreDoubling
- psac

Prefix Doubling: Experimental Results [Kur20]
Recap: SAIS

The Idea: Inducing

Given a text $T$ of length $n$ and two positions $i, j \in [1..n]$ with $T[i] = T[j]$, then

$$T[i..n] < T[j..n] \iff T[i + 1..n] < T[j + 1..n]$$
The Idea: Inducing

Given a text $T$ of length $n$ and two positions $i, j \in [1..n]$ with $T[i] = T[j]$, then

$$T[i..n] < T[j..n] \iff T[i + 1..n] < T[j + 1..n]$$

Recap: SAIS
Recap: SAIS

The Idea: Inducing

Given a text $T$ of length $n$ and two positions $i, j \in [1..n]$ with $T[i] = T[j]$, then

$$T[i..n] < T[j..n] \iff T[i+1..n] < T[j+1..n]$$

The Algorithm: SAIS

- using inducing for everything
- described in [NZC11]
Recap: SAIS

The Idea: Inducing
Given a text $T$ of length $n$ and two positions $i, j \in [1..n]$ with $T[i] = T[j]$, then

$T[i..n] < T[j..n] \iff T[i+1..n] < T[j+1..n]$ 

The Algorithm: SAIS
- using inducing for everything
- described in [NZC11]

Suffix Array Construction in 3 Phases
- classification
- sort special substrings/suffixes recursively
- induce all non-sorted suffixes
Recap: SAIS

The Idea: Inducing

Given a text $T$ of length $n$ and two positions $i, j \in [1..n]$ with $T[i] = T[j]$, then

\[ T[i..n] < T[j..n] \iff T[i+1..n] < T[j+1..n] \]

The Algorithm: SAIS

- using inducing for everything
- described in [NZC11]

Suffix Array Construction in 3 Phases

- classification
- sort special substrings/suffixes recursively
- induce all non-sorted suffixes

- classification helps identifying special suffixes
- everything in linear time
SAIS in External Memory [BFO16; Kär+17]

**Classification**
- simple scan of the text
- works well in external memory
- separate text during classification
- blockwise preinducing
- heavily relies on external memory priority queue

**Sort Special Substrings**
- recursion
- works well in external memory if rest works well

**Inducing**
- keep buffer for each $\alpha$-interval of suffix array
- scan text and induce characters by writing them in buffer
Jack of all Trades: DC3

- first direct linear time suffix array construction algorithm: DC3
- suffix tree construction algorithm with similar idea [Far97]
- based on Difference Cover
Definition: Difference Cover

The set $D \subseteq [0, \nu)$ is a difference cover modulo $\nu$, if

$$\{(i - j) \mod \nu : i, j \in D\} = [0, \nu)$$

- $\{0, 1\}$ is difference cover modulo 3
- $\{0, 1, 3\}$ is difference cover modulo 7
- $\{0, 1, 3, 9\}$ is difference cover modulo 13
Definition: Difference Cover

The set \( D \subseteq [0, \nu) \) is a \textit{difference cover} modulo \( \nu \), if

\[
\{(i - j) \mod \nu : i, j \in D\} = [0, \nu)
\]

- \{0, 1\} is difference cover modulo 3
- \{0, 1, 3\} is difference cover modulo 7
- \{0, 1, 3, 9\} is difference cover modulo 13

- \( 0 \equiv 0 - 0 \pmod{3} \)
- \( 1 \equiv 1 - 0 \pmod{3} \)
- \( 2 \equiv 0 - 1 \pmod{3} \)
Definition: Difference Cover

The set $D \subseteq [0, \nu)$ is a difference cover modulo $\nu$, if

$$\{(i-j) \mod \nu : i, j \in D\} = [0, \nu)$$

- $\{0, 1\}$ is difference cover modulo 3
- $\{0, 1, 3\}$ is difference cover modulo 7
- $\{0, 1, 3, 9\}$ is difference cover modulo 13

- $0 \equiv 0 - 0 \pmod{3}$
- $1 \equiv 1 - 0 \pmod{3}$
- $2 \equiv 0 - 1 \pmod{3}$
- $0 \equiv 0 - 0 \pmod{7}$
- $1 \equiv 1 - 0 \pmod{7}$
- $2 \equiv 3 - 1 \pmod{7}$
- $3 \equiv 3 - 0 \pmod{7}$
- $4 \equiv 0 - 3 \pmod{7}$
- $5 \equiv 1 - 3 \pmod{7}$
- $6 \equiv 0 - 1 \pmod{7}$
1. Sample Suffixes

- for $i \in \{0, 1, 2\}$ let be
  
  \[ B_k = \{ i \in [0, n) : i \mod 3 = k \} \]

- $C = B_0 \cdot B_1$

  \[ \{0, 1\} \text{ is difference cover modulo } 3 \]
1. Sample Suffixes

- for $i \in \{0, 1, 2\}$ let be

$$B_k = \{i \in [0, n): i \mod 3 = k\}$$

- $C = B_0 \cdot B_1$
  - $\{0, 1\}$ is difference cover modulo 3
1. Sample Suffixes

- for $i \in \{0, 1, 2\}$ let be
  \[ B_k = \{ i \in [0, n) : i \mod 3 = k \} \]
- $C = B_0 \cdot B_1$
  \[ \{0, 1\} \text{ is difference cover modulo 3} \]
1. Sample Suffixes

- for $i \in \{0, 1, 2\}$ let be
  
  $B_k = \{i \in [0, n) : i \mod 3 = k\}$

- $C = B_0 \cdot B_1$
  
  $\{0, 1\}$ is difference cover modulo 3

- $C = \{0, 3, 6, 9, 1, 4, 7, 10\}$
2. Sort Sampled Suffixes

- for \( k = 0, 1 \) let be
  
  \[
  R_k = [T[k] T[k+1] T[k+2]] [T[k+3] T[k+4] T[k+5]] \ldots [T[\text{max } B_k] T[\text{max } B_k + 1] T[\text{max } B_k + 2]]
  \]

- \( R = R_0 \cdot R_1 \)
- sort \( R \) with Radix Sort in \( O(n) \) time
- all characters unique: ranks of sampled suffixes are known
- otherwise: recursively execute algorithm on \( R \)
2. Sort Sampled Suffixes

- for $k = 0, 1$ let be

$$R_k = [T[k] T[k + 1] T[k + 2]] [T[k + 3] T[k + 4] T[k + 5]] \ldots [T[\text{max } B_k] T[\text{max } B_k + 1] T[\text{max } B_k + 2]]$$

- $R = R_0 \cdot R_1$

- sort $R$ with Radix Sort in $O(n)$ time

- all characters unique: ranks of sampled suffixes are known

- otherwise: recursively execute algorithm on $R$
2. Sort Sampled Suffixes

- for $k = 0, 1$ let be

\[
\]

- $R = R_0 \cdot R_1$
- sort $R$ with Radix Sort in $O(n)$ time
- all characters unique: ranks of sampled suffixes are known
- otherwise: recursively execute algorithm on $R$
Suffix Array Construction with DC3 (3/6)

Recursion: Step 1

\[
C = \{0, 3, 6, 1, 4, 7\}
\]
Suffix Array Construction with DC3 (3/6)

Recursion: Step 1

\[
C = \{0, 3, 6, 1, 4, 7\}
\]
Suffix Array Construction with DC3 (3/6)

Recursion: Step 1

```
C = {0, 3, 6, 1, 4, 7}
```
Suffix Array Construction with DC3 (3/6)

Recursion: Step 1

\[ C = \{0, 3, 6, 1, 4, 7\} \]
### Suffix Array Construction with DC3 (3/6)

**Recursion: Step 1**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- $C = \{0, 3, 6, 1, 4, 7\}$

**Recursion: Step 2**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>[365]</td>
<td>[422]</td>
<td>[100]</td>
<td>[654]</td>
<td>[221]</td>
<td>[000]</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
3. Sort Non-Sampled Suffixes

- let \( i, j \in B_2 \), then
  \[
  S_i \leq S_j \iff (T[i], Rang(S_{i+1})) \leq (T[j], Rang(S_{j+1}))
  \]

- ranks of next two suffixes is known
- sort tuples (in \( B_2 \)) using Radix Sort
- \( O(n) \) time
3. Sort Non-Sampled Suffixes

- let \(i, j \in B_2\), then
  \[S_i \leq S_j \iff (T[i], \text{Rang}(S_{i+1})) \leq (T[j], \text{Rang}(S_{j+1}))\]

- ranks of next two suffixes is known
- sort tuples (in \(B_2\)) using Radix Sort
- \(O(n)\) time

<table>
<thead>
<tr>
<th>ranks</th>
<th>3</th>
<th>5</th>
<th>4</th>
<th>2</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
3. Sort Non-Sampled Suffixes

- let \( i, j \in B_2 \), then
  \[
  S_i \leq S_j \iff (T[i], Rang(S_{i+1})) \leq (T[j], Rang(S_{j+1}))
  \]

- ranks of next two suffixes is known
- sort tuples (in \( B_2 \)) using Radix Sort
- \( O(n) \) time
3. Sort Non-Sampled Suffixes

- let $i, j \in B_2$, then
  \[ S_i \leq S_j \iff (T[i], \text{Rang}(S_{i+1})) \leq (T[j], \text{Rang}(S_{j+1})) \]

- ranks of next two suffixes is known
- sort tuples (in $B_2$) using Radix Sort
- $O(n)$ time

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>ranks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

$(2, 1) \leq (5, 4)$

$S_2 \leq S_5$
4. Merge Suffixes

- let $i \in C$ and $j \in B_2$, then
  - if $i \in B_0$, then
    
    
    
    $S_i \leq S_j \iff (T[i], \text{Rang}(S_{i+1})) \leq (T[j], \text{Rang}(S_{j+1}))$

  - if $i \in B_1$, then
    
    
    
    $S_i \leq S_j \iff (T[i], T[i+1], \text{Rang}(S_{i+2})) \leq (T[j], T[j+1], \text{Rang}(S_{j+2}))$
4. Merge Suffixes

- let $i \in C$ and $j \in B_2$, then
  - if $i \in B_0$, then
    $$S_i \leq S_j \iff \langle T[i], \text{Rang}(S_{i+1}) \rangle \leq \langle T[j], \text{Rang}(S_{j+1}) \rangle$$
  - if $i \in B_1$, then
    $$S_i \leq S_j \iff \langle T[i], T[i+1], \text{Rang}(S_{i+2}) \rangle \leq \langle T[j], T[j+1], \text{Rang}(S_{j+2}) \rangle$$
4. Merge Suffixes

- let \( i \in C \) and \( j \in B_2 \), then
  - if \( i \in B_0 \), then
    \[
    S_i \leq S_j \iff (T[i], \text{Rang}(S_{i+1})) \leq (T[j], \text{Rang}(S_{j+1}))
    \]
  - if \( i \in B_1 \), then
    \[
    S_i \leq S_j \iff (T[i], T[i+1], \text{Rang}(S_{i+2})) \leq (T[j], T[j+1], \text{Rang}(S_{j+2}))
    \]
4. Merge Suffixes

- let $i \in C$ and $j \in B_2$, then
  - if $i \in B_0$, then
    $$S_i \leq S_j \iff (T[i], Rang(S_{i+1})) \leq (T[j], Rang(S_{j+1}))$$
  - if $i \in B_1$, then
    $$S_i \leq S_j \iff (T[i], T[i+1], Rang(S_{i+2})) \leq (T[j], T[j+1], Rang(S_{j+2}))$$
4. Merge Suffixes

- let $i \in C$ and $j \in B_2$, then
  - if $i \in B_0$, then
    $$S_i \leq S_j \iff (T[i], \text{Rang}(S_{i+1})) \leq (T[j], \text{Rang}(S_{j+1}))$$
  - if $i \in B_1$, then
    $$S_i \leq S_j \iff (T[i], T[i+1], \text{Rang}(S_{i+2})) \leq (T[j], T[j+1], \text{Rang}(S_{j+2}))$$

<table>
<thead>
<tr>
<th>ranks</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>5</td>
<td></td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- $(2, 1) \leq (5, 4)$
- $(0, 0, 0) \leq (2, 0, 0)$
4. Merge Suffixes

- let $i \in C$ and $j \in B_2$, then
  - if $i \in B_0$, then
    $$S_i \leq S_j \iff (T[i], \text{Rang}(S_{i+1})) \leq (T[j], \text{Rang}(S_{j+1}))$$
  - if $i \in B_1$, then
    $$S_i \leq S_j \iff (T[i], T[i+1], \text{Rang}(S_{i+2})) \leq (T[j], T[j+1], \text{Rang}(S_{j+2}))$$
4. Merge Suffixes

- let $i \in C$ and $j \in B_2$, then
  - if $i \in B_0$, then
    \[ S_i \leq S_j \iff (T[i], \text{Rang}(S_{i+1})) \leq (T[j], \text{Rang}(S_{j+1})) \]
  - if $i \in B_1$, then
    \[ S_i \leq S_j \iff (T[i], T[i+1], \text{Rang}(S_{i+2})) \leq (T[j], T[j+1], \text{Rang}(S_{j+2})) \]

\[
\begin{align*}
0 & \leq 1 & \leq 2 & \leq 3 & \leq 4 & \leq 5 & \leq 6 & \leq 7 \\
3 & \leq 6 & \leq 5 & \leq 4 & \leq 2 & \leq 2 & \leq 1 & \leq 0 \\
\end{align*}
\]

- \((2, 1) \leq (5, 4)\)
  - \(S_2 \leq S_5\)

- \((0, 0, 0) \leq (2, 0, 0)\)
- \((1, 0) \leq (2, 1)\)
- \((2, 1, 0) \leq (2, 2, 1)\)
- \(\ldots\)
4. Merge Suffixes

- let \( i \in C \) and \( j \in B_2 \), then
  - if \( i \in B_0 \), then
    \( S_i \leq S_j \iff (T[i], \text{Rang}(S_{i+1})) \leq (T[j], \text{Rang}(S_{j+1})) \)
  - if \( i \in B_1 \), then
    \( S_i \leq S_j \iff (T[i], T[i+1], \text{Rang}(S_{i+2})) \leq (T[j], T[j+1], \text{Rang}(S_{j+2})) \)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

ranks: 3 5 ↓ 4 2 ↓ 1 0

- \( (2, 1) \leq (5, 4) \)
- \( (0, 0, 0) \leq (2, 0, 0) \)
- \( (1, 0) \leq (2, 1) \)
- \( (2, 1, 0) \leq (2, 2, 1) \)
- \( \ldots \)
- ranks: 4 7 6 5 3 2 1 0
## Suffix Array Construction with DC3 (6/6)

### Finish Recursion

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[mis]</td>
<td>[sis]</td>
<td>[sip]</td>
<td>[pi$]</td>
<td>[iss]</td>
<td>[iss]</td>
<td>[ipp]</td>
<td>[i$$]</td>
</tr>
<tr>
<td>ranks</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
## Suffix Array Construction with DC3 (6/6)

<table>
<thead>
<tr>
<th>Finish Recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>[mis]</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

### Ranks

<table>
<thead>
<tr>
<th>Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>mississippiippipi$</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
### Suffix Array Construction with DC3 (6/6)

**Finish Recursion**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$mis$</td>
<td>$sis$</td>
<td>$sip$</td>
<td>$pi$</td>
<td>$iss$</td>
<td>$iss$</td>
<td>$ipp$</td>
<td>$i$$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**ranks**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$mis$</td>
<td>$iss$</td>
<td>$iss$</td>
<td>$s$</td>
<td>$ipp$</td>
<td>$i$$</td>
<td>ranks</td>
<td>4</td>
<td>3</td>
<td>$\downarrow$</td>
<td>7</td>
<td>2</td>
<td>$\downarrow$</td>
</tr>
</tbody>
</table>

- rest can be used as exercise

solution: 11 10 7 4 1 0 9 8 6 3 5 2
DC3: Running Times

- everything but recursion obviously in $O(n)$ time
- only sorting tuples of size $\leq 3$
- Radix Sort in $O(n)$ time
DC3: Running Times

- everything but recursion obviously in $O(n)$ time
- only sorting tuples of size $\leq 3$
- Radix Sort in $O(n)$ time

- recursion on texts of size $\lceil 2n/3 \rceil$
- $T(n) = T(2n/3) + O(n) = O(n)$
DC3: Running Times

- everything but recursion obviously in $O(n)$ time
- only sorting tuples of size $\leq 3$
- Radix Sort in $O(n)$ time

- recursion on texts of size $\lceil 2n/3 \rceil$
- $T(n) = T(2n/3) + O(n) = O(n)$

Generalization

- works with every difference cover
- sorting somewhat more complicated
- running time: $O(\nu n)$
DC3: Running Times

- everything but recursion obviously in $O(n)$ time
- only sorting tuples of size $\leq 3$
- Radix Sort in $O(n)$ time

- recursion on texts of size $\lceil 2n/3 \rceil$
- $T(n) = T(2n/3) + O(n) = O(n)$

Generalization

- works with every difference cover
- sorting somewhat more complicated
- running time: $O(\nu n)$

In Other Models of Computation

- external memory: $O\left(\frac{n}{DB} \log \frac{n}{B}\right)$ using $D$ disks
- BSP: $O\left(\frac{n \log n}{P} + L \log^2 P + g \frac{n \log n}{P \log(n/P)}\right)$ using $P$ PEs
- EREW-PRAM: $O(\log^2 n)$ time and $O(n \log n)$ work

DC3: Running Times

- everything but recursion obviously in $O(n)$ time
- only sorting tuples of size $\leq 3$
- Radix Sort in $O(n)$ time

- recursion on texts of size $\lceil 2n/3 \rceil$
- $T(n) = T(2n/3) + O(n) = O(n)$

Generalization

- works with every difference cover
- sorting somewhat more complicated
- running time: $O(\nu n)$

In Other Models of Computation

- external memory: $O\left(\frac{n}{DB} \log \frac{n}{B}\right)$ using $D$ disks
- BSP: $O\left(\frac{n \log n}{P} + L \log^2 P + g \frac{n \log n}{P \log(n/P)}\right)$ using $P$ PEs
- EREW-PRAM: $O(\log^2 n)$ time and $O(n \log n)$ work
Prefix Doubling: Experimental Results [Kur20]

Commoncrawl throughput (MiB/s)

- 512 MiB per PE
- 1024 MiB per PE
- 1536 MiB per PE

Commoncrawl construction space (B/n)

- PEs (20 threads)

Legend:
- pDivSufSort
- pPreDoubling
- psac
- pDC3
- pDC7
- pDC13

Prefix Doubling: Experimental Results [Kur20]
Conclusion and Outlook

This Lecture
- distributed and external memory suffix sorting
- more suffix sorting techniques

Linear Time Construction

- ST
- SA
- WT
- LZ
- LCP
- BWT
- FM-Index
- r-Index
Conclusion and Outlook

This Lecture
- distributed and external memory suffix sorting
- more suffix sorting techniques

Next Lecture
- move data structure (???)

Linear Time Construction

- ST
- SA
- WT
- LZ
- LCP
- BWT
- FM-Index
- r-Index
Evaluation

https://onlineumfrage.kit.edu/evasys/online.php?p=K2FFL
Bibliography I


Bibliography II


Bibliography III


