Text Indexing

Lecture 09: Suffix Array Construction in Distributed and External Memory

Florian Kurpicz
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
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

Prefix Doubling
- 1990: [MM] original
- 1999: [LS] qsufsort

Induced Copying
- 2000: 1/2 copy
- 2002: A/B copy
- 2003: [BWT] BWT
- 2004: [IT] A/B copy
- 2005: [IT] A/B copy

Recursion
- 2006: [MF] deep-shallow
- 2007: [KA] L/S split
- 2008: [Na] succinct
- 2009: [MF] cache aware
- 2011: [NS] O(n lg |Σ|)
- 2016: [NS] O(n lg |Σ|)
- 2017: [Got] O(1) space
- 2021: [Gre] libSAIS

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

Prefix Doubling
- 1990: [MM] original
- 1999: [LS] qsufsort

Induced Copying
- 2000: 1/2 copy
- 2002: A/B copy
- 2003: [BWT] BWT
- 2004: [IT] A/B copy
- 2005: [IT] A/B copy

Recursion
- 2006: [MF] deep-shallow
- 2007: [KA] L/S split
- 2008: [Na] succinct
- 2009: [MF] cache aware
- 2011: [NS] O(n lg |Σ|)
- 2016: [NS] O(n lg |Σ|)
- 2017: [Got] O(1) space
- 2021: [Gre] libSAIS

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 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

External and Distributed 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: $\Theta\left(\frac{N}{B} \log \frac{M}{B} \frac{N}{B}\right)$

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 — B — external memory

PE 1  PE 2  PE 3  ...  PE p

Scanning, Sorting, Merging

Karlsruhe Institute of Technology
Challenges for Suffix Array Construction

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

External Memory
- random access expensive in both models
- whole suffix not available locally in distributed memory
Challenges for Suffix Array Construction

**Distributed Memory**
- Suffixes span over whole input \(\text{no locality}\)
- Comparing suffixes requires text access \(\text{random access}\)
- Random access expensive in both models
- Whole suffix not available locally in distributed memory

**Express suffix array construction algorithm using**
- Scanning
- Sorting
- Merging

**External Memory**

```
PE 1    PE 2    PE 3    ...    PE p
```

```
[...]
```
Prefix-Doubling | Induced-Copying | Recursion

2003 | [KSB] DC3

2012 | [B] DC3 7/13

2014 | cloudSACA

2015 | PSAC

Distributed Memory

PE 1 | PE 2 | PE 3 | \ldots | PE p

Speicher | Speicher | Speicher | Speicher | Speicher

Institute for Theoretical Informatics, Algorithm Engineering
### Distributed Memory

<table>
<thead>
<tr>
<th>PE 1</th>
<th>PE 2</th>
<th>PE 3</th>
<th>...</th>
<th>PE p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speicher</td>
<td>Speicher</td>
<td>Speicher</td>
<td>...</td>
<td>Speicher</td>
</tr>
</tbody>
</table>

Prefix-Doubling: 2003
Induced-Copying: 2012
Recursion: 2014

Prefix-Doubling: [AKA] cloudSACA
Induced-Copying: [FA] PSAC [BGK]
Recursion: [BGK] Doubling

DC3: 2003
DC3/7/13: 2015
[6/25] 2023-01-09 Florian Kurpicz | Text Indexing | 09 Suffix Array in Distributed and External Memory
Institute for Theoretical Informatics, Algorithm Engineering
Prefix-Doubling

Induced-Copying

Recursion

Distributed Memory

PE 1

PE 2

PE 3

…

PE p
**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 und 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
**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

---

<table>
<thead>
<tr>
<th>$m$</th>
<th>$i$</th>
<th>$s$</th>
<th>$s$</th>
<th>$i$</th>
<th>$s$</th>
<th>$i$</th>
<th>$p$</th>
<th>$p$</th>
<th>$i$</th>
<th>$$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>$s$</td>
<td>$s$</td>
<td>$i$</td>
<td>$s$</td>
<td>$i$</td>
<td>$p$</td>
<td>$p$</td>
<td>$i$</td>
<td>$$$</td>
<td></td>
</tr>
<tr>
<td>$s$</td>
<td>$i$</td>
<td>$s$</td>
<td>$i$</td>
<td>$p$</td>
<td>$p$</td>
<td>$i$</td>
<td>$$$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i$</td>
<td>$s$</td>
<td>$i$</td>
<td>$p$</td>
<td>$p$</td>
<td>$i$</td>
<td>$$$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Definition: h-Order**

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

---

**Definition: h-Ranks und 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
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 und $h$-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
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 $T[i] \circ 1$-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 $T[i] \odot 1$-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

|m| i| s| s| i| s| s| i| p| p| i| $|
|i| s| s| i| s| s| i| p| p| i| $|
|s| i| s| s| i| s| s| i| p| p| i| $|
|s| i| s| s| i| s| s| i| p| p| i| $|
|i| s| s| i| p| i| $|
|s| i| p| p| i| $|
|i| p| p| i| $|
p| p| i| $|
i| $|
$|
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 $T[i]$ 1-rank
2. for $k = 0$ to $\lceil \log_2 n \rceil$
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 $T[i] \circlearrowleft 1$-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 $ISA_{2^k}[i] \& ISA_{2^k}[i + 2^k]$
4. if all ranks are unique, break
5. compute $SA$ from $ISA$
1. initial rank is $T[i] \overset{1}{\text{-rank}}$

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

1. initial rank is $T[i] \triangleright 1$-rank
2. for $k = 0$ to $\left\lceil \log n \right\rceil$
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 $T[i] \in [1, \ldots, \ell]$
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 $T[i] \odot 1$-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 $T[i] \{1\}$-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 $ISA_{2^k}[i] \& ISA_{2^k}[i + 2^k]$
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 $ISA$

---

Simple Algorithm

- N. Jesper Larsson and Kunihiko Sadakane.
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

**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

**Sort by Text Positions [Dem+08; FK19]**
- especially good if access to ISA \_h is expensive
- sort tuples (Textposition \( i \), Rang \( r \))
- using \((i, r) \leq (j, r')\) iff
  \[(i \mod 2^k, \lfloor i/2^k \rfloor) < (j \mod 2^k, \lfloor j/2^k \rfloor)\]
- example on the board
Prefix-Doubling: Running Time

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

- running time: $O(n \lg n)$
- 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 $\alpha^{k}$-ranks
- can save I/Os in EM $\alpha = 4$ requires 30% less I/Os than $\alpha = 2$ [Dem+08]
Prefix Doubling: Experimental Results [Kur20]
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

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

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

```
<table>
<thead>
<tr>
<th>a</th>
<th>α</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>β</td>
</tr>
</tbody>
</table>
```
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

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

- separate text during classification
- blockwise preinducing
- heavily relies on external memory priority queue
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
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
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

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

- for $i \in \{0, 1, 2\}$ let be
  \[ B_i = \{i \in [0, n) : i \mod 3 = k\} \]
- $C = B_0 \cdot B_1$

$\{0, 1\}$ is difference cover modulo 3
1. Sample Suffixes

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

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

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
</tr>
</tbody>
</table>

$mississippi$
1. Sample Suffixes

- for \( i \in \{0, 1, 2\} \) let be
  \[
  B_i = \{i \in [0, n) : i \mod 3 = k\}
  \]
- \( C = B_0 \cdot B_1 \)
  \( \{0, 1\} \) is difference cover modulo 3

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

- for $k = 0, 1$ let be

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

- $R = R_0 \cdot R_1$

- 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 $R$

$$\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\text{mis} & \text{sis} & \text{sip} & \text{pi$} & \text{iss} & \text{iss} & \text{ipp} & \text{i$}$ \\
3 & 6 & 5 & 4 & 2 & 2 & 1 & 0 \\
\end{array}$$
2. Sort Sampled Suffixes

- for \( k = 0, 1 \) let be

\[
R_k = \left[ T[k] T[k + 1] T[k + 2] \right] \left[ T[k + 3] T[k + 4] T[k + 5] \right] \ldots \left[ T[\max B_k] T[\max B_k + 1] T[\max B_k + 2] \right]
\]

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

<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>mis</td>
<td>sis</td>
<td>sip</td>
<td>pi$</td>
<td>iss</td>
<td>iss</td>
<td>ipp</td>
<td>i$</td>
</tr>
</tbody>
</table>

| 3 | 6 | 5 | 4 | 2 | 2 | 1 | 0 |
## 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>
### 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>
Suffix Array Construction with DC3 (3/6)

Recursion: Step 1

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

\[\begin{array}{cccccccc}
3 & 6 & 5 & 4 & 2 & 2 & 1 & 0
\end{array}\]
Recursion: Step 1

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

Recursion: Step 1

\[ C = \{0, 3, 6, 1, 4, 7\} \]

Recursion: Step 2

\[ C = \{0, 3, 6, 1, 4, 7\} \]
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
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>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
```
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
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
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>6</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

ranks: 4 7 6 5 3 2 1 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
    $$S_i \leq S_j \iff (T[i], T[i+1], \text{Rang}(S_{i+2})) \leq (T[j], T[j+1], \text{Rang}(S_{j+2}))$$
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
    \[ S_i \leq S_j \iff (T[i], T[i+1], Rang(S_{i+2})) \leq (T[j], T[j+1], Rang(S_{j+2})) \]

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

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

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
3 & 6 & 5 & 4 & 2 & 2 & 1 & 0 \\
\end{array}
\]

ranks: \([3, 5, 4, 2, 2, 1, 0]\)

- \( (2, 1) \leq (5, 4) \)
- \( (0, 0, 0) \leq (2, 0, 0) \)
- \( (1, 0) \leq (2, 1) \)
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
    \[
    S_i \leq S_j \iff (T[i], T[i+1], \text{Rang}(S_{i+2})) \leq (T[j], T[j+1], \text{Rang}(S_{j+2}))
    \]
## Suffix Array Construction with DC3 (6/6)

### Finish Recursion

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

The ranks are as follows: 11 10 7 4 1 0 9 8 6 3 5 2.
## Suffix Array Construction with DC3 (6/6)

### Finish Recursion

<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>[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>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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>mis</td>
<td>sis</td>
<td>iss</td>
<td>sip</td>
<td>pi$</td>
<td>iss</td>
<td>iss</td>
<td>ipp</td>
<td>i$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Ranks

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>↓</td>
<td>7</td>
<td>2</td>
<td>↓</td>
<td>6</td>
<td>1</td>
<td>↓</td>
<td>5</td>
<td>0</td>
<td>↓</td>
</tr>
</tbody>
</table>
### Suffix Array Construction with DC3 (6/6)

#### Finish Recursion

<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>[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>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>

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

Common crawl throughput (MiB/s)

512 MiB per PE

1024 MiB per PE

1536 MiB per PE

Common crawl construction space (B/n)

PEs (20 threads)

Prefix Doubling: Experimental Results [Kur20]

pDivSufSort

pPreDoubling

psac

pDC3

pDC7

pDC13
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
- inverted indices

Linear Time Construction

- ST
- SA
- WT
- LZ
- LCP
- BWT
- FM-Index
- r-Index
Bibliography I


Bibliography II


Bibliography III

[KSB06] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. “Linear work suffix array construction”. In: 

Germany, 2020. DOI: 10.17877/DE290R-21114.


[MM93] Udi Manber and Eugene W. Myers. “Suffix Arrays: A New Method for On-Line String Searches”. In: 

Construction”. In: IEEE Trans. Computers 60.10 (2011), pages 1471–1484. DOI: 
10.1109/TC.2010.188.