Definition: Suffix Array \[\text{GBS92; MM93}\]
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
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: $\Theta\left(\frac{N}{B} \log \frac{M}{B} \frac{N}{B}\right)$

Distributed Memory

- $p$ PEs with internal memory
- communication between PEs over network

bulk-synchronous parallel model \cite{Val90}

supersteps: local work, communication, synchronization
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
- PE 1
- PE 2
- PE 3
- ...
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
- ...
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
- express suffix array construction algorithm using
  - scanning
  - sorting
  - merging

External Memory
- main memory
- external memory
- PE 1, PE 2, PE 3, ..., PE p
- suffixes span over whole input
- comparing suffixes requires text 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

[AKA] cloudSACA

[FA] PSAC

[Doubling] [BGK]

[Doubling] [FK]

DivSufSort

[KSB] DC3

DC3/7/13

[PSAC]

Distributed Memory

PE 1
Speicher

PE 2
Speicher

PE 3
Speicher

... 

PE p
Speicher
Prefix-Doubling

Induced-Copying

Recursion

Distributed Memory

PE 1

PE 2

PE 3

... 

PE p
**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-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

**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
**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

\( \text{not unambiguously} \)

**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
**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:**

Suppose we have the following suffixes ordered by h-order:

- `mississippi$`
- `ississippi$`
- `ssississippi$`
- `ississippi$`
- `ssissippi$`
- `ississippi$`
- `issippi$`
- `ppi$`
- `pi$`
- `$`

The suffix array \( SA_h \) would be ordered as follows:

1. `mississippi$
2. `ississippi$
3. `ssissippi$
4. `ississippi$
5. `issippi$
6. `ppi$
7. `pi$
8. `$`

Where each number represents the index of the suffix in the array.
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 \( T[i] \) 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

\textbf{Simple Algorithm}

N. Jesper Larsson and Kunihiko Sadakane.
DOI: 10.1016/j.tcs.2007.07.017
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

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 $\lceil \log n \rceil$
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

Prefix-Doubling: Example

Prefix-Doubling: Example

Prefix-Doubling: Example

Prefix-Doubling: Example
Prefix-Doubling: Example

1. initial rank is $T[i]$ 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 $T[i] \odot 1$-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 $ISA$

| 2 | 1 | $n$ | $s$ | $s$ | $i$ | $s$ | $i$ | $p$ | $p$ | $i$ | $|$ |
|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | 4 | $3$ | $s$ | $i$ | $s$ | $i$ | $p$ | $p$ | $i$ | $|$ |
| 4 | 4 | $8$ | $s$ | $i$ | $s$ | $i$ | $p$ | $p$ | $i$ | $|$ |
| 4 | 1 | $7$ | $s$ | $s$ | $i$ | $p$ | $p$ | $i$ | $|$ |
| 1 | 4 | $3$ | $s$ | $i$ | $p$ | $p$ | $i$ | $|$ |
| 4 | 4 | $8$ | $i$ | $p$ | $p$ | $i$ | $|$ |
| 4 | 1 | $7$ | $p$ | $p$ | $i$ | $|$ |
| 1 | 3 | $2$ | $p$ | $i$ | $|$ |
| 3 | 3 | $6$ | $i$ | $|$ |
| 3 | 1 | $5$ | $|$ |
| 1 | 0 | $1$ | $|$ |
| 0 | 0 | $0$ | $|$ |
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 $ISA$

\begin{align*}
2 & 1 14 ss\text{s} s s i s i p p i s \\
1 & 4 1 3 ss s i s i p p i s \\
4 & 4 8 i s s i p p i s \\
4 & 1 7 s i s i p p i s \\
1 & 4 1 3 s i p p i s \\
4 & 4 8 i p p i s \\
4 & 1 7 p p i s \\
1 & 3 1 2 p i s \\
3 & 3 6 i s \\
3 & 1 5 s \\
1 & 0 1 i s \\
0 & 0 0 s
\end{align*}
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 $\lceil \log n \rceil$
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 $T[i] \oplus 1$-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 $T[i] \downarrow$ 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

```
4 8 s s i s s i p p i $
3 7 s i s i s i p p i $
8 3 10 i s s i p p i $
7 8 s i s i p p i $
3 7 s i p p i $
8 2 p p i $
7 6 p p i $
2 5 p i $
6 1 6 i $
5 0 5 $
1 0 1 $
0 0 0 $
```
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 $T[i] \in 1$-rank
2. for $k = 0$ to $\lceil \lg n \rceil$
3. new $2^{k+1}$-ranks based on
   \begin{align*}
   ISA_{2^k}[i] \& ISA_{2^k}[i + 2^k]
   \end{align*}
4. if all ranks are unique, break
5. compute SA from ISA
Prefix-Doubling: Example

1. initial rank is $T[i] \in 1$-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>- especially good if access to $ISA_h$ is expensive</td>
</tr>
<tr>
<td>- for position $i$ use rank $ISA_{2^k}[i + 2^k]$</td>
<td>- sort tuples $(\text{Textposition } i, \text{Rang } r)$</td>
</tr>
<tr>
<td>- if $i + 2^k &gt; n$, second rank is 0</td>
<td>- using $(i, r) \leq (j, r')$ iff</td>
</tr>
<tr>
<td>- example on the board</td>
<td>$(i \mod 2^k, \lfloor i/2^k \rfloor) &lt; (j \mod 2^k, \lfloor j/2^k \rfloor)$</td>
</tr>
<tr>
<td></td>
<td>- example on the board</td>
</tr>
</tbody>
</table>
Prefix-Doubling: Running Time

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

- running time: \( O(n \log n) \)
- memory requirements: \( 8n(n+n) \) words \( \uparrow \) for texts \( \leq 4 \) 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 \( \uparrow \alpha = 4 \) requires 30% less I/Os than \( \alpha = 2 \) \([\text{Dem}+08]\)
Prefix Doubling: Experimental Results [Kur20]

Throughput (MB/s) vs. PEs (20 threads):
- 512 MiB per PE
- 1024 MiB per PE
- 1536 MiB per PE

Construction space (B/n) vs. PEs (20 threads):
- 63.1
- 39.8
- 31.6
- 25.1

Graphs show the performance of the algorithms under different memory constraints and processor element (PE) counts.
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

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

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

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

![Illustration of alphabets α and β]

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

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

Suffix Array Construction in 3 Phases

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

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

- 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
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 **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
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

\[ C = \{0, 3, 6, 9, 1, 4, 7, 10\} \]
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

$$C = \{0, 3, 6, 9, 1, 4, 7, 10\}$$
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 $R$
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 \( R \)

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

<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>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

\begin{align*}
C &= \{0, 3, 6, 1, 4, 7\} \\
\begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
3 & 6 & 5 & 4 & 2 & 2 & 1 & 0
\end{bmatrix}
\end{align*}
Suffix Array Construction with DC3 (3/6)

Recursion: Step 1

Recursion: Step 1
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>

- $C = \{0, 3, 6, 1, 4, 7\}$
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>
</tbody>
</table>

| 3 | 4 | 1 | 5 | 2 | 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

ranks:

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
3 & 6 & 5 & 4 & 2 & 2 & 1 & 0 \\
\end{array}
\]
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
Suffix Array Construction with DC3 (5/6)

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
    $S_i \leq S_j \iff (T[i], T[i+1], \text{Rang}(S_{i+2})) \leq (T[j], T[j+1], \text{Rang}(S_{j+2}))$

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

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

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

ranks

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

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

\[
\begin{array}{cccccccc}
S_2 & S_5 \\
\end{array}
\]
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
    \[ S_i \leq S_j \iff (T[i], T[i+1], \text{Rang}(S_{i+2})) \leq (T[j], T[j+1], \text{Rang}(S_{j+2})) \]

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

- \( (2,1) \leq (5,4) \)
- \( (0,0,0) \leq (2,0,0) \)
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
    $S_i \leq S_j \iff (T[i], T[i+1], \text{Rang}(S_{i+2})) \leq (T[j], T[j+1], \text{Rang}(S_{j+2}))$

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

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

- \((0, 0, 0) \leq (2, 0, 0)\)
- \((1, 0) \leq (2, 1)\)
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
    
    $S_i \leq S_j \iff (T[i], T[i+1], \text{Rang}(S_{i+2})) \leq (T[j], T[j+1], \text{Rang}(S_{j+2}))$

<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

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

- $(0, 0, 0) \leq (2, 0, 0)$
- $(1, 0) \leq (2, 1)$
- $(2, 1, 0) \leq (2, 2, 1)$
- ...
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
    \[ S_i \leq S_j \iff (T[i], T[i+1], \text{Rang}(S_{i+2})) \leq (T[j], T[j+1], \text{Rang}(S_{j+2})) \]
### 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>
## 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>rnk</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>

### Problem Solution

The suffix array for the given string is:

- Suffixes: `mis sissippi $`
- Ranks: 4 3 7 2 6 1 5 0 0 0 0

The suffix array is constructed as follows:

- **mis** → 4
- **sis** → 7
- **sip** → 6
- **pi$** → 5
- **iss** → 3
- **iss** → 2
- **ipp** → 1
- **i$$** → 0

The ranks are calculated based on the lexicographical order of the suffixes.
### 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>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></td>
<td>mis</td>
<td>sis</td>
<td>sis</td>
<td>$sip$</td>
<td>$pi$</td>
<td>$i$$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ranks</td>
<td>4</td>
<td>3</td>
<td>$\perp$</td>
<td>7</td>
<td>$\perp$</td>
<td>2</td>
<td>$\perp$</td>
<td>6</td>
<td>$\perp$</td>
<td>1</td>
<td>$\perp$</td>
<td>5</td>
</tr>
</tbody>
</table>

- rest can be used as exercise

Lösung: 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 $[2n/3]$
- $T(n) = T(2n/3) + O(n) = O(n)$

Generalization

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

In Other Models of Computation

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

512 MiB per PE

1024 MiB per PE

1536 MiB per PE

PeDivSufSort

pPreDoubling

psac

pDC3

pDC7

pDC13

Commoncrawl throughput (MiB/s)

Commoncrawl construction space (B/n)

PEs (20 threads)

PEs (20 threads)

PEs (20 threads)
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 III


