Optimal In-Place Suffix Sorting

Zhize Li, Jian Li, Hongwei Huo

IIIS, Tsinghua University

https://zhizeli.github.io/

October 10th, SPIRE 2018

- Problem Definition
- Related Work
- Our Results
- Our Algorithm
- Conclusion

Problem Definition

Suffix array is a fundamental data structure introduced by Manber and Myers as a **space-saving** alternative to suffix trees in SODA'90.

Problem Definition

Suffix array is a fundamental data structure introduced by Manber and Myers as a **space-saving** alternative to suffix trees in SODA'90.

- Definition:
  - Given a string T[0..n-1], each T[i] belongs to an integer alphabet Σ
  - *Suffix*: suf(i) is a substring T[i…n-1] (from index i to the end of T)
  - *Suffix array* SA contains the indices of all sorted suffixes

- Example:
  - If **T="130"** (integer alphabet), then all suffixes are **{130, 30, 0}**
  - suf(2)<suf(0)<suf(1), i.e. 0<130<30 (in lexicographical order**)**
  - **SA=[2,0,1]** suf(SA[i]) < suf(SA[j]) for all i<j

Problem Definition

Suffix array is a fundamental data structure introduced by Manber and Myers as a **space-saving** alternative to suffix trees in SODA’90.

- Definition:
  - Given a string T[0..n-1], each T[i] belongs to an integer alphabet Σ
  - *Suffix*: suf(i) is a substring T[i…n-1] (from index i to the end of T)
  - *Suffix array* SA contains the indices of all sorted suffixes

- Example:
  - If **T="130"** (integer alphabet), then all suffixes are **{130, 30, 0}**
  - suf(2)<suf(0)<suf(1), i.e. 0<130<30 (in lexicographical order)
  - **SA=[2,0,1]**   suf(SA[i]) < suf(SA[j]) for all i<j

- Problem:

  Construct the suffix array SA for a given string T[0…n-1]

- Problem Definition
- Related Work
- Our Results
- Our Algorithm
- Conclusion

Related Work

- Manber and Myers [MM90] constructed the SA using a doubling technique.
  - beginning characters -> first two characters-> first four characters….

    Time: O(nlogn)   Space: O(n) workspace

Workspace denotes the **total space** used by an algorithm except for the **input string T** and the **suffix array SA**.

Related Work

- Manber and Myers [MM90] constructed the SA using a doubling technique.
  - beginning characters -> first two characters-> first four characters….

    Time: $O(n\log n)$ Space: $O(n)$ workspace

- In 2003, the first linear time algorithms were obtained by [KSPP03,KS03,KA03] using the divide-and-conquer technique.

  Time: $T(n)=T(cn)+O(n)=O(n)$ Space: $O(n)$ workspace

Related Work

- Manber and Myers [MM90] constructed the SA using a doubling technique.
  - beginning characters -> first two characters-> first four characters….

    Time: $O(n\log n)$ Space: $O(n)$ workspace

- In 2003, the first linear time algorithms were obtained by [KSPP03,KS03,KA03] using the divide-and-conquer technique.

  Time: $T(n)=T(cn)+O(n)=O(n)$ Space: $O(n)$ workspace

- Bottleneck: space

Related Work

- Open problem [Franceschini and Muthukrishnan, ICALP'07]:
  - Design in-place (O(1) workspace) and o(nlogn) time algorithms for integer alphabets Σ with $|\Sigma| \le n$.

Related Work

- Open problem [Franceschini and Muthukrishnan, ICALP'07]:
  - Design in-place (O(1) workspace) and o(nlogn) time algorithms for integer alphabets Σ with $|\Sigma| \leq n$.
- Ultimate goal: design *in-place algorithms*, and maintain the optimal O(n) time complexity.

Related Work

- Open problem [Franceschini and Muthukrishnan, ICALP'07]:
  - Design in-place (O(1) workspace) and o(nlogn) time algorithms for integer alphabets Σ with $|\Sigma| \leq n$.

- Ultimate goal: design *in-place algorithms*, and maintain the optimal O(n) time complexity.

- Previous best result [Nong, TOIS'13]:

  Time: O(n)   Space: $|\Sigma|$ + O(1) workspace

Related Work

- Open problem [Franceschini and Muthukrishnan, ICALP'07]:
  - Design in-place (O(1) workspace) and o(nlogn) time algorithms for integer alphabets Σ with $|\Sigma| \leq n$.

- Ultimate goal: design *in-place algorithms*, and maintain the optimal O(n) time complexity.

- Previous best result [Nong, TOIS'13]:

  Time: O(n)   Space: $|\Sigma|$ + O(1) workspace

Theorem. *Our optimal in-place algorithm takes O(n) time to compute the suffix array even if the string T is read-only and* $|\Sigma| = O(n)$.

- Problem Definition
- Related Work
- Our Results
- Our Algorithm
- Conclusion

Our Results

Table 1: Time and workspace of suffix sorting algorithms for integer alphabets Σ

Time	Workspace (words)	Algorithms
$O(n^2)$	$O(n)$	[SS07]
$O(n\log^2 n)$	$O(n)$	[Sad98]
$O(n\sqrt{\lvert\Sigma\rvert\log(n/\lvert\Sigma\rvert)})$	$O(n)$	[BB05]
$O(n\log n)$	$O(n)$	[MM90, LS07]
$O(n\log\log n)$	$O(n)$	[KJP04]
$O(n)$	$O(n)$	[KSPP03, KS03, KA03]
$O(n\log\log\lvert\Sigma\rvert)$	$O(n\log\lvert\Sigma\rvert/\log n)$	[HSS03]
$O(vn)$	$O(n/\sqrt{v})\ \ v\in[1,\sqrt{n}]$	[KSB06]
$O(n)$	$n+n/\log n+O(1)$	[NZC09]
$O(n^2\log n)$	$cn+O(1)\ \ c<1$	[MF02, MP06]
$O(n^2\log n)$	$\lvert\Sigma\rvert+O(1)$	[IT99]
$O(n\log\lvert\Sigma\rvert)$	$\lvert\Sigma\rvert+O(1)$	[NZ07]
$O(n)$	$\lvert\Sigma\rvert+O(1)$	[Nong13]
$O(n)$	$O(1)$	This paper

- Problem Definition
- Related Work
- Our Results
- Our Algorithm
- Conclusion

Our Algorithm

- Notations:
  - A suf(i) (T[i..n-1]) is ***L-suffix*** if suf(i)>suf(i+1)
  - Type of character T[i] is the same as suf(i)
  - ***LMS-suffix*** (leftmost S-suffix) if suf(i) is S-type and suf(i-1) is L-type

- Example: T[0..7]="31221120"

Index	0	1	2	3	4	5	6	7
T	3	1	2	2	1	1	2	0
Type	L	S	L	L	S	S	L	S
LMS		$*$			$*$			$*$

Our Algorithm

- **Framework:**
  - 1. Sort all LMS-characters of T (counting sort)
  - 2. Induced sort all LMS-substrings from sorted LMS-characters (same as 5)
  - 3. Construct the reduced subproblem T_1 from sorted LMS-substrings (simple)

Index	0	1	2	3	4	5	6	7
T	3	1	2	2	1	1	2	0
Type	L	S	L	L	S	S	L	S
LMS		$*$			$*$			$*$

LMS-substrings are $\{1221, 1120, 0\}$.

SA	Index(LMS)	$E\ E\ E\ E\ E\ 7\ 1\ 4$
2.	LMS-substring	$E\ E\ E\ E\ E\ 7\ 4\ 1$
3.	T_1 LMS-substring	$2\ 1\ 0\ E\ E\ 7\ 4\ 1$

Our Algorithm

- **Framework:**
  - 1. Sort all LMS-characters of T (counting sort)
  - 2. Induced sort all LMS-substrings from sorted LMS-characters (same as 5)
  - 3. Construct the reduced subproblem T_1 from sorted LMS-substrings (simple)

Index	0	1	2	3	4	5	6	7
T	3	1	2	2	1	1	2	0
Type	L	S	L	L	S	S	L	S
LMS		*			*			*

LMS-substrings are $\{1221, 1120, 0\}$.

3. $\boxed{T_1 \quad \text{LMS-substring}}$ $\boxed{2\ 1\ 0\ E\ E\ 7\ 4\ 1}$

Index	1	4	7
LMS-Substring	1221	1120	0
Rank (T_1)	2	1	0

T_1 ("210") shortens T("12211120") by using **one character** to **replace a substring** of T

Our Algorithm

- **Framework:**
  - 1. Sort all LMS-characters of T (counting sort)
  - 2. Induced sort all LMS-substrings from sorted LMS-characters (same as 5)
  - 3. Construct the reduced subproblem T_1 from sorted LMS-substrings (simple)

Index	0	1	2	3	4	5	6	7
T	3	1	2	2	1	1	2	0
Type	L	S	L	L	S	S	L	S
LMS		$*$			$*$			$*$

LMS-substrings are $\{1221, 1120, 0\}$.

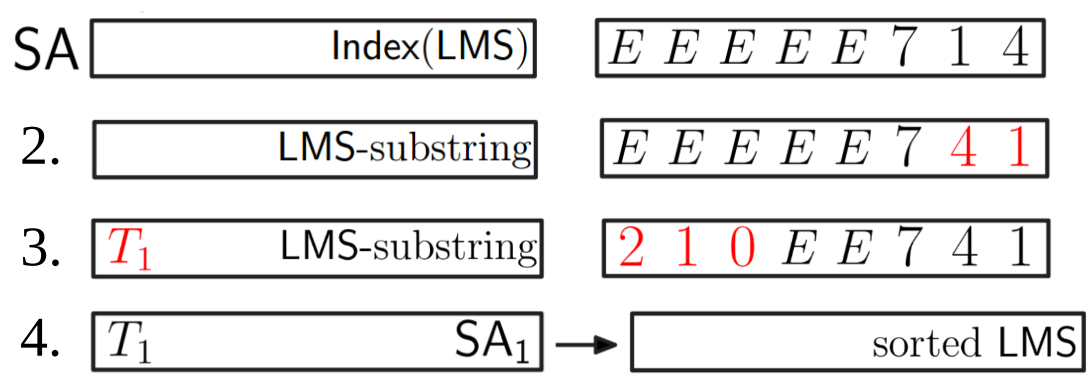
SA	Index(LMS)	$E\ E\ E\ E\ E\ 7\ 1\ 4$
2.	LMS-substring	$E\ E\ E\ E\ E\ 7\ 4\ 1$
3.	T_1 LMS-substring	$2\ 1\ 0\ E\ E\ 7\ 4\ 1$

Our Algorithm

- **Framework:**
  - 1. Sort all LMS-characters of T (counting sort)
  - 2. Induced sort all LMS-substrings from sorted LMS-characters (same as 5)
  - 3. Construct the reduced subproblem T_1 from sorted LMS-substrings (simple)
  - 4. Sort LMS-suffixes of T by solving T_1 recursively (simple)

Index	0	1	2	3	4	5	6	7
T	3	1	2	2	1	1	2	0
Type	L	S	L	L	S	S	L	S
LMS		$*$			$*$			$*$

LMS-substrings are $\{1221, 1120, 0\}$.



Our Algorithm

- **Framework:**
  - 1. Sort all LMS-characters of T (counting sort)
  - 2. Induced sort all LMS-substrings from sorted LMS-characters (same as 5)
  - 3. Construct the reduced subproblem T_1 from sorted LMS-substrings (simple)
  - 4. Sort LMS-suffixes of T by solving T_1 recursively (simple)

3. T_1 LMS-substring $2\ 1\ 0\ E\ E\ 7\ 4\ 1$

4. T_1 SA_1 $\rightarrow$ sorted LMS

Index	0	1	2	3	4	5	6	7
T	3	1	2	2	1	1	2	0
Type	L	S	L	L	S	S	L	S
LMS		*			*			*

LMS-substrings are $\{1221, 1120, 0\}$.

LMS-Suffixes are $\{1221120, 1120, 0\}$.

Index	1	4	7
LMS-Substring	1221	1120	0
Rank (T_1)	2	1	0

T_1 ("210") shortens T("12211120") by using **one character** to **replace a substring** of T

A suffix of T_1 corresponds to an LMS-suffix of T

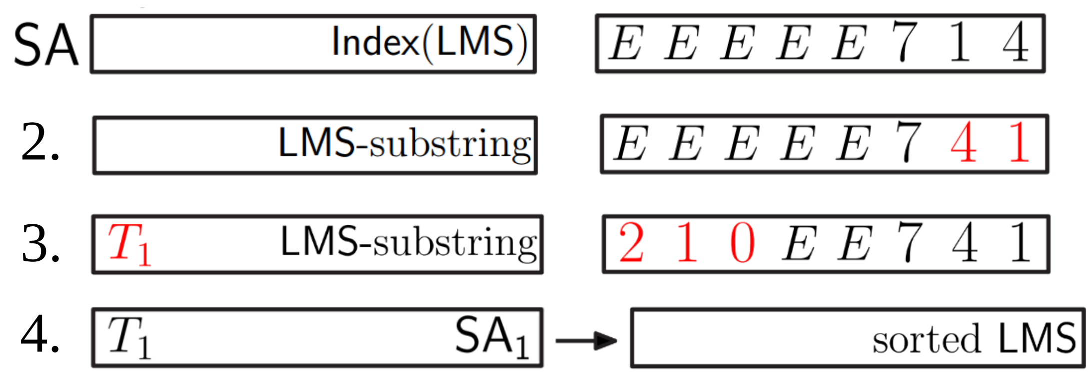
Our Algorithm

- **Framework:**
  - 1. Sort all LMS-characters of T (counting sort)
  - 2. Induced sort all LMS-substrings from sorted LMS-characters (same as 5)
  - 3. Construct the reduced subproblem T_1 from sorted LMS-substrings (simple)
  - 4. Sort LMS-suffixes of T by solving T_1 recursively (simple)

Index	0	1	2	3	4	5	6	7
T	3	1	2	2	1	1	2	0
Type	L	S	L	L	S	S	L	S
LMS		$*$			$*$			$*$

LMS-substrings are $\{1221, 1120, 0\}$.

LMS-Suffixes are $\{1221120, 1120, 0\}$.



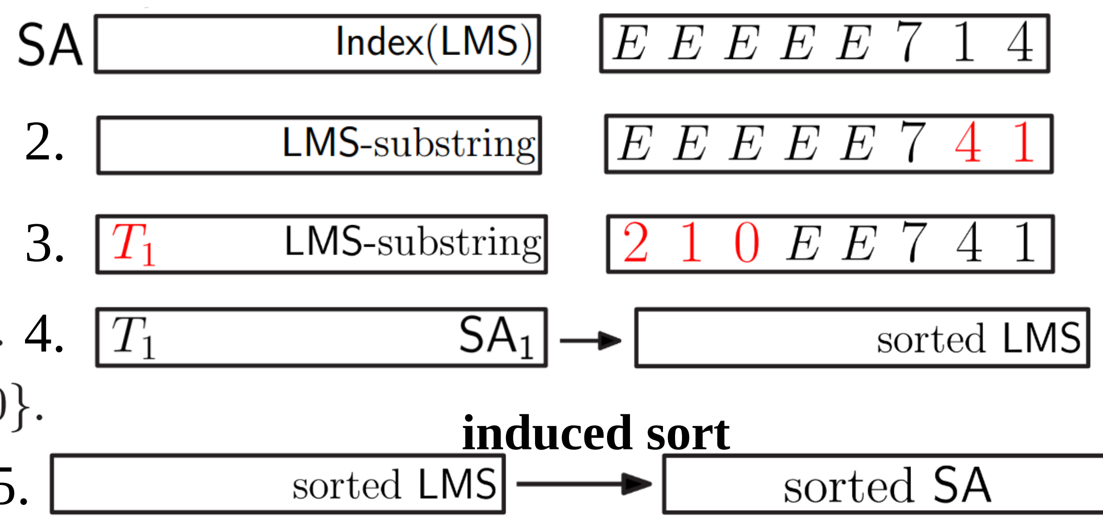
Our Algorithm

- **Framework:**
  - 1. Sort all LMS-characters of T (counting sort)
  - 2. Induced sort all LMS-substrings from sorted LMS-characters (same as 5)
  - 3. Construct the reduced subproblem T_1 from sorted LMS-substrings (simple)
  - 4. Sort LMS-suffixes of T by solving T_1 recursively (simple)
  - 5. **Induced sort all suffixes from the sorted LMS-suffixes** (technical part)

Index	0	1	2	3	4	5	6	7
T	3	1	2	2	1	1	2	0
Type	L	S	L	L	S	S	L	S
LMS		*			*			*

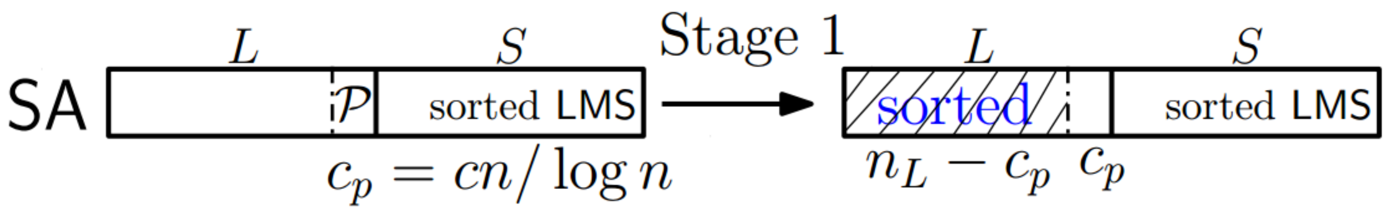
LMS-substrings are $\{1221, 1120, 0\}$.

LMS-Suffixes are $\{1221120, 1120, 0\}$.



Our Algorithm

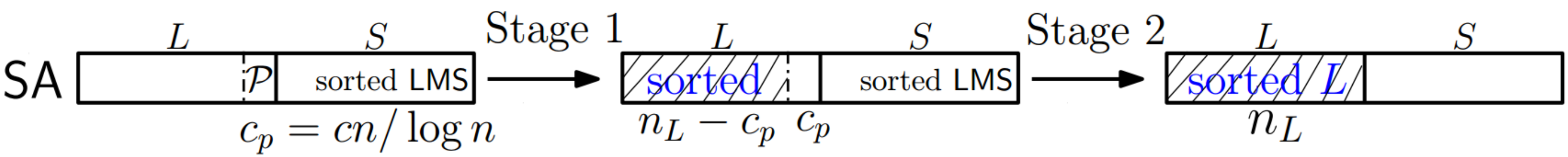
- **Induced sorting all suffixes from the sorted LMS-suffixes**
  - 1. First induced sort **all L-suffixes** from the sorted LMS-suffixes
    - Divide into two stages



Stage 1: Construct *pointer data structure* $\mathcal{P}$ & combine *interior counter trick* to induced sort the first $n_L - c_p$ L-suffixes.

Our Algorithm

- **Induced sorting all suffixes from the sorted LMS-suffixes**
  - 1. First induced sort **all L-suffixes** from the sorted LMS-suffixes
    - Divide into two stages



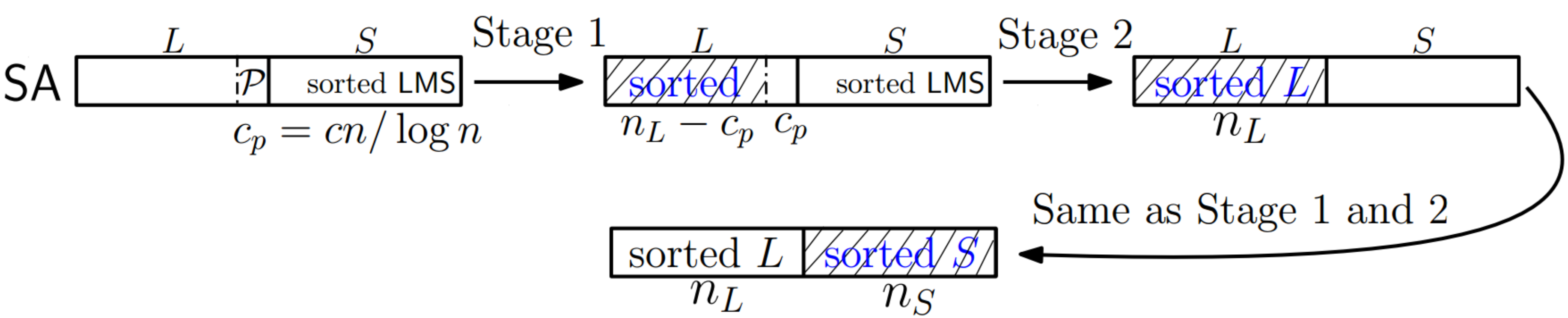
Stage 1: Construct *pointer data structure $\mathcal{P}$* & combine *interior counter trick* to induced sort the first $n_L - c_p$ L-suffixes.

Stage 2: Use binary search to extend the interior counter trick to induced sort the last c_p L-suffixes without $\mathcal{P}$.

Key: without $\mathcal{P}$, Stage 2 also maintains **linear time** since c_p is small enough (i.e., $c_p \log n = cn$).

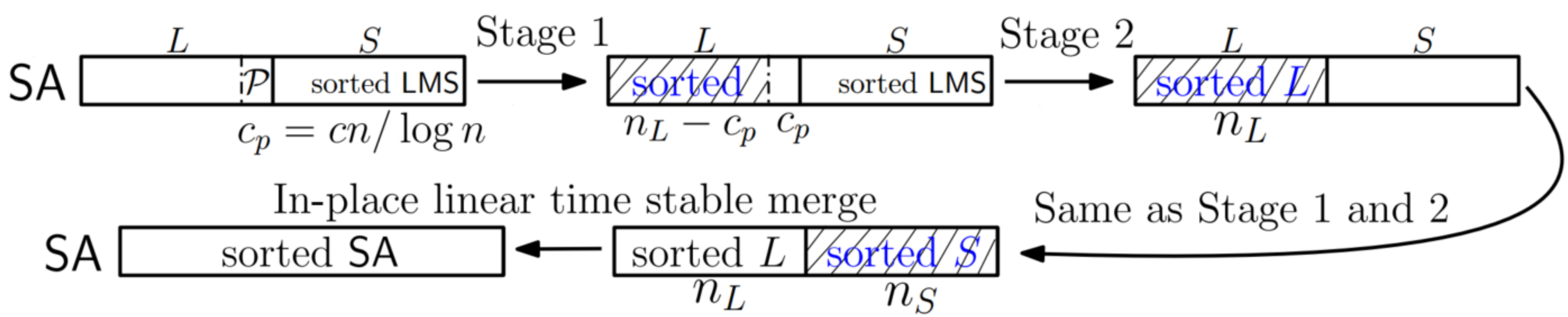
Our Algorithm

- **Induced sorting all suffixes from the sorted LMS-suffixes**
  - 1. First induced sort **all L-suffixes** from the sorted LMS-suffixes
    - Divide into two stages
  - 2. Then induced sort **all S-suffixes** from the sorted L-suffixes (same as 1)



Our Algorithm

- **Induced sorting all suffixes from the sorted LMS-suffixes**
  - 1. First induced sort **all L-suffixes** from the sorted LMS-suffixes
    - Divide into two stages
  - 2. Then induced sort **all S-suffixes** from the sorted L-suffixes (same as 1)
  - 3. **Merge the sorted L- and S-suffixes** to get the final suffix array



- Problem Definition
- Related Work
- Our Results
- Our Algorithm
- Conclusion

Conclusion

- We propose the *first* in-place suffix sorting algorithm which is *optimal both in time and space*.

  Time: $O(n)$ Space: $O(1)$ workspace (in-place)

- Our algorithm solves the open problem posed by Franceschini and Muthukrishnan in ICALP 2007.
  - Desired time and space in their open problem:

    Time: $o(n\log n)$ Space: $O(1)$ workspace (in-place)

Thanks!

Zhize Li (IIIS,Tsinghua University)
https://zhizeli.github.io/

Our Results

We also give a **simpler** in-place algorithm for the general alphabet (**only comparisons between characters are allowed**).

Table 2: Time and workspace of suffix sorting algorithms for general alphabets

Time	Workspace(words)	Algorithms
$O(n \log n)$	$O(n)$	[MM90, LS07]
$O(vn + n \log n)$	$O(v + n/\sqrt{v})$ $v \in [2, n]$	[BK03]
$O(vn + n \log n)$	$O(n/\sqrt{v})$ $v \in [1, \sqrt{n}]$	[KSB06]
$O(n \log n)$	$O(1)$	[FM07]
$O(n \log n)$	$O(1)$	This paper

Clearly, $\Omega(n\log n)$ is a lower bound, as it generalizes comparison-based sorting.