# A Taxonomy of Suffix Array Construction Algorithms

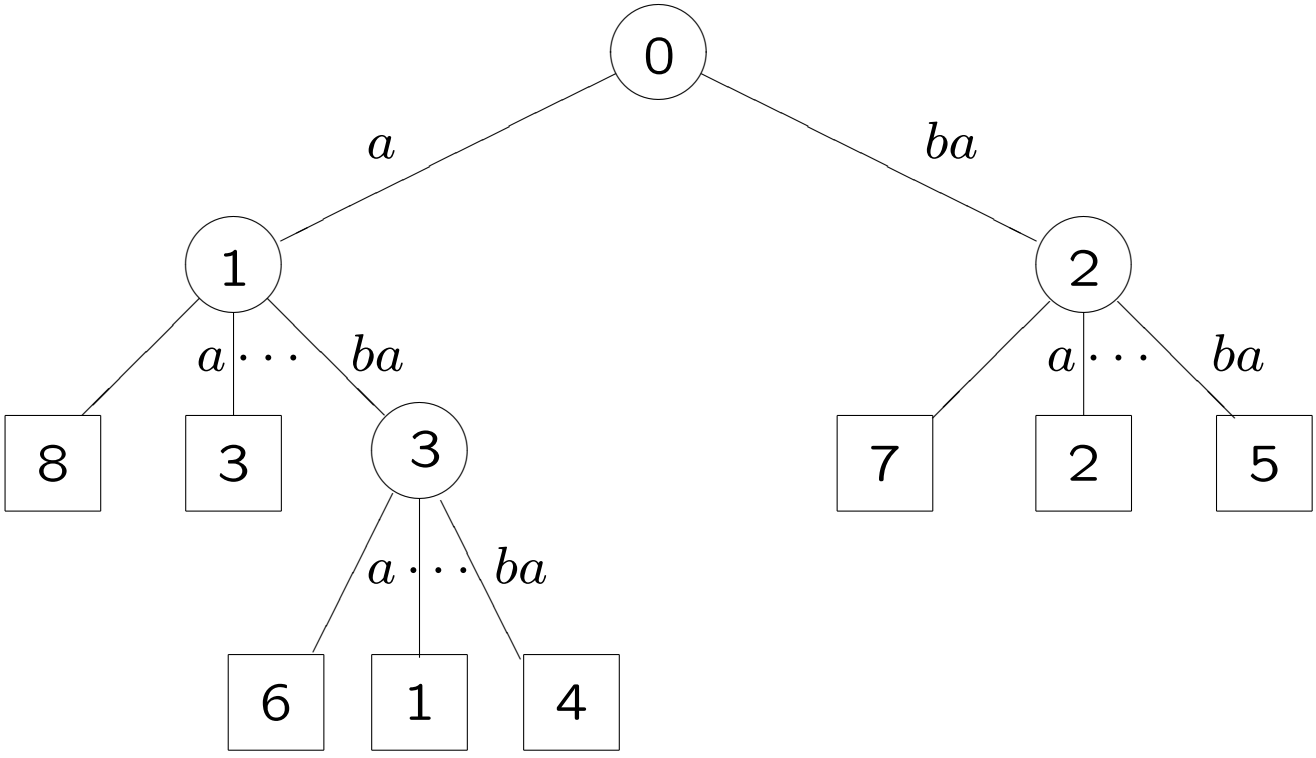
*Simon Puglisi*[*], *Bill Smyth*[†*]
*& Andrew Turpin*[‡]

In 1990 Manber & Myers proposed suffix arrays as a space-saving alternative to suffix trees and described the first algorithms for suffix array construction and use. Since that time there have been many new suffix array construction algorithms. This talk gives simple high-level descriptions of these algorithms that highlight both distinctive features and commonalities, while avoiding as much as possible the details of implementation. We also provide comparisons of the algorithms' worst-case time complexity and use of additional space, together with results of recent experimental test runs on many of their implementations.

[*] Department of Computing, Curtin University
[†] Algorithms Research Group, Department of Computing & Software, McMaster University
[‡] School of Computer Science & Information Technology, RMIT University

# What is a Suffix Array?

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{x} =$ | $a$ | $b$ | $a$ | $a$ | $b$ | $a$ | $b$ | $a$ |
| $\mathsf{SA}_{\boldsymbol{x}} =$ | 8 | 3 | 6 | 1 | 4 | 7 | 2 | 5 |
| $\mathsf{lcp}_{\boldsymbol{x}} =$ | $-$ | 1 | 1 | 3 | 3 | 0 | 2 | 2 |



## Why is an SA Important?

- does everything an ST can do with identical asymptotic time complexity:
  - pattern-matching in time proportional to pattern length;
  - find all repetitions/repeats in time linear in string length $n$;
- requires only $4n$ bytes of storage (ST perhaps 12–20$n$).

## Brief History of STs & SAs

| | | |
|---|---|---|
| 1968 | Morrison | Patricia (compacted) trie |
| 1973 | Weiner | invents ST & first STCA |
| 1976 | McCreight | faster STCA |
| 1981 | Crochemore | STCA in disguise |
| 1985 | Apostolico | "myriad virtues" of STs |
| 1990 | Manber & Myers | invent SA & first SACA |
| 1992 | Ukkonen | on-line STCA |
| 1997 | Farach | linear-time STCA |
| 2003 | KA/KS/KSPP | three linear-time SACAs! |
| 2005 | many researchers | 14 SACAs + variants |
| 2006 | Maniscalco/Puglisi | blow them away! |

## Vocabulary

| | |
|---|---|
| **suffix** $i$: | $\boldsymbol{x}[i..n]$ |
| **SACA**: | suffix array construction algorithm |
| **indexed** | letters $\boldsymbol{x}[i]$ in $\boldsymbol{x}$ can be radix-sorted |
| **alphabet**: | in $\Theta(n)$ time |
| **inverse** SA: | $\text{ISA}[i] = j \Leftrightarrow \text{SA}[j] = i$ |

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{x} =$ | $a$ | $b$ | $e$ | $a$ | $c$ | $a$ | $d$ | $a$ | $b$ | $e$ | $a$ | \$ |
| $\text{SA}_{\boldsymbol{x}} =$ | 12 | 11 | 8 | 1 | 4 | 6 | 9 | 2 | 5 | 7 | 10 | 3 |
| $\text{ISA}_{\boldsymbol{x}} =$ | 4 | 8 | 12 | 5 | 9 | 6 | 10 | 3 | 7 | 11 | 2 | 1 |

| | |
|---|---|
| **rank**: | $\text{ISA}[i] = j \Leftrightarrow$ suffix $i$ has rank $j$ in lexorder |
| $h$-**sort**: | ordering of suffixes $i$ based on prefixes $\boldsymbol{x}[i..i+h-1]$ or $\boldsymbol{x}[i..n]$, whichever is shorter |

Hence $h$-**order**, $h$-**ordering**, $h$-**rank**, $h$-**group**, $h$-**equal**.

Vocabulary (continued)

Then an **approximate** $\mathrm{SA}_h/\mathrm{ISA}_h$ can be formed:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{x} =$ | $a$ | $b$ | $e$ | $a$ | $c$ | $a$ | $d$ | $a$ | $b$ | $e$ | $a$ | \$ |
| $\mathrm{SA}_1 =$ | 12 | (1 | 4 | 6 | 8 | 11) | (2 | 9) | 5 | 7 | (3 | 10) |
| $\mathrm{ISA}_1 =$ | 2 | 7 | 11 | 2 | 9 | 2 | 10 | 2 | 7 | 11 | 2 | 1 |
| or | 6 | 8 | 12 | 6 | 9 | 6 | 10 | 6 | 8 | 12 | 6 | 1 |
| or | 2 | 3 | 6 | 2 | 4 | 2 | 5 | 2 | 3 | 6 | 2 | 1 |

The $h$-rank in $\mathrm{ISA}_h$ may be expressed in various ways:

- the leftmost position $j$ in the $h$-group. called the **head** of the $h$-group;
- the rightmost position $j$ in the $h$-group, called the **tail** of the $h$-group;
- the ordinal left-to-right counter of the $h$-group.

Three Genera of SACAs

**(1) Prefix-Doubling** First a fast 1-sort yields $\mathsf{SA}_1/\mathsf{ISA}_1$. Then for every $h = 1, 2, \ldots$, $\mathsf{SA}_{2h}/\mathsf{ISA}_{2h}$ are computed in $\Theta(n)$ time from $\mathsf{SA}_h/\mathsf{ISA}_h$ The time required is therefore $O(n \log n)$.

**(2) Recursive** Form strings $\boldsymbol{x'}$ and $\boldsymbol{y}$ from $\boldsymbol{x}$, then show that if $\mathsf{SA}_{\boldsymbol{x'}}$ is computed, therefore $\mathsf{SA}_{\boldsymbol{y}}$ and finally $\mathsf{SA}_{\boldsymbol{x}}$ can be computed in $O(n)$ time. Hence the problem of computing $\mathsf{SA}_{\boldsymbol{x'}}$ recursively replaces the computation of $\mathsf{SA}_{\boldsymbol{x}}$. Since $|\boldsymbol{x'}|$ is always chosen so as to be less than $2|\boldsymbol{x}|/3$, the overall time requirement of these algorithms is $\Theta(n)$.

Three Genera of SACAs (continued)

**(3) Induced Copying** The idea is the same as for the recursive algorithms — a complete sort of a selected subset of suffixes can be used to "induce" a complete sort of other subsets of suffixes. But the approach is nonrecursive: an efficient suffix sorting technique is invoked for the selected subset of suffixes. These methods are very efficient in practice, but worst-case complexity may be as high as $O(n^2 \log n)$.

The goal is to design SACAs that

- have minimal asymptotic complexity $\Theta(n)$;
- are fast "in practice";
- are **lightweight** — that is, use a small amount of working storage in addition to the $5n$ bytes required by $\boldsymbol{x}$ and $\mathrm{SA}_{\boldsymbol{x}}$.

None of the 14 SACAs satisfy all these criteria!

## Performance Summary of SACAs

Speed is relative to MF; memory is in bytes, including space for SA and $\boldsymbol{x}$.

| Algorithm | Worst Case | Speed | Memory |
|---|---|---|---|
| Prefix-Doubling | | | |
| MM | $O(n \log n)$ | 16 | $8n$ |
| LS | $O(n \log n)$ | 1.7 | $8n$ |
| Recursive | | | |
| KA | $O(n)$ | 2.2 | 13-14$n$ |
| KS | $O(n)$ | 2.8 | 10-13$n$ |
| KSPP | $O(n)$ | - | – |
| HSS | $O(n)$ | - | – |
| KJP | $O(n \log \log n)$ | 2.1 | 13-16$n$ |
| Induced Copying | | | |
| IT | $O(n^2 \log n)$ | 4 | $5n$ |
| S | $O(n^2 \log n)$ | 2.1 | $5n$ |
| BK | $O(n \log n)$ | 2.1 | 5-6$n$ |
| MF | $O(n^2 \log n)$ | 1 | $5n$ |
| SS | $O(n^2)$ | 1 | 9-10$n$ |
| M | $O(n^2 \log n)$ | 1 | 5-7$n$ |
| Suffix Tree | | | |
| K | $O(n \log \sigma)$ | 4 | 15-20$n$ |

## Prefix-Doubling Algorithms: $O(n \log n)$

These algorithms **refine** an $h$-order $\mathrm{SA}_h$ into a $2h$-order $\mathrm{SA}_{2h}$. After a bucket sort computes $\mathrm{SA}_1$, the algorithms employ

**Lemma 1 (Karp** *et al.* **1972)** *Given* $\mathrm{SA}_h$ *and* $\mathrm{ISA}_h$, $h > 0$, *where* $i = \mathrm{SA}_h[j]$ *is the* $j^{th}$ *suffix in* $h$*-order and* $h$-$\mathrm{rank}[i] = \mathrm{ISA}_h[i]$, *a sort of the integer pairs*

$$\Big(\mathrm{ISA}_h[i], \mathrm{ISA}_h[i+h]\Big),$$

$i+h \leq n$, *computes a* $2h$*-order of the suffixes* $i$. *(Suffixes* $i > n-h$ *are already fully ordered.)*

MM does an implicit $2h$-sort by performing a left-to-right scan of $\mathrm{SA}_h$ that induces the $2h$-rank of $\mathrm{SA}_h[j]-h$. MM uses the *head* of each $h$-group as $h$-rank.

LS explicitly sorts each $h$-group using the ternary-split quicksort [Bentley & McIlroy 1993]. LS uses the *tail* of each $h$-group as $h$-rank.

## Prototype of a Prefix-Doubling Algorithm (MM)

$h \leftarrow 1$
initialize $SA_1$, $\mathrm{ISA}_1$
**while** some $h$-group not a singleton
  **for** $j \leftarrow 1$ **to** $n$ **do**
    $i \leftarrow \mathrm{SA}_h[j] - h$
    **if** $i > 0$ **then**
      $q \leftarrow \mathrm{head}\big[h\text{-group}[i]\big]$
      $\mathrm{SA}_{2h}[q] \leftarrow i$
      $\mathrm{head}\big[h\text{-group}[i]\big] \leftarrow q+1$
  compute $\mathrm{ISA}_{2h}$ — *update $2h$-groups*
  $h \leftarrow 2h$

## Recursive Algorithms: $\Theta(n)$

These are based on the ST construction algorithm of Farach [1997]: separate the suffixes of $\boldsymbol{x}$ into two or more classes based on a **type**, then **split** them into disjoint strings (subsequences) $\boldsymbol{x}^{(1)}$ and $\boldsymbol{y}^{(1)}$, chosen so that, if $\mathrm{SA}_{\boldsymbol{x}^{(1)}}$ is (recursively) computed, then in linear time

- $\mathrm{SA}_{\boldsymbol{x}^{(1)}}$ can be used to **induce** construction of $\mathrm{SA}_{\boldsymbol{y}^{(1)}}$, and furthermore

- $\mathrm{SA}\boldsymbol{x} = \mathrm{SA}_{\boldsymbol{x}^{(0)}}$ can then also be computed by a **merge** of $\mathrm{SA}_{\boldsymbol{x}^{(1)}}$ and $\mathrm{SA}_{\boldsymbol{y}^{(1)}}$.

Thus the computation of $\mathrm{SA}_{\boldsymbol{x}^{(0)}}$ (in general, $\mathrm{SA}_{\boldsymbol{x}^{(i)}}$) is reduced to the computation of $\mathrm{SA}_{\boldsymbol{x}^{(1)}}$ (in general, $\mathrm{SA}_{\boldsymbol{x}^{(i+1)}}$).

Recursive Algorithms (continued)

To achieve linear time:

1. At each recursive step, ensure that

$$|\boldsymbol{x}^{(i+1)}| \Big/ |\boldsymbol{x}^{(i)}| \leq f < 1,$$

   so that the sum of the lengths processed by all recursive steps is at most $|\boldsymbol{x}|/\Big(1-f\Big)$.

2. Devise a linear-time approximate suffix-sorting procedure, **semisort** say, that for some sufficiently short string $\boldsymbol{x}^{(i+1)}$ will yield a complete sort of its suffixes and thus terminate the recursion, allowing the suffixes of $\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(i-1)}, \ldots, \boldsymbol{x}^{(0)}$ all to be sorted in turn.

## The Generalized Recursive Algorithm

**procedure** construct($x$; SA)
split($x$; $x'$, $y$)
semisort($x'$; ISA$'$)
**if** ISA$'$ contains duplicate ranks **then**
  construct(ISA$'$; $\text{SA}_{x'} = \text{SA}'$)
**else**
  invert($\text{ISA}_{x'} = \text{ISA}'$; $\text{SA}_{x'}$)
induce($\text{SA}_{x'}$, $\text{ISA}_{x'}$; $\text{SA}_y$)
merge($\text{SA}_{x'}$, $\text{SA}_y$; $\text{SA}_x$)

## Induced Copying Algorithms: $O(n^2 \log n)$

These algorithms are based on an idea originally put forward by Burrows & Wheeler [1994]: use a complete sort of a selected subset of suffixes to **induce** a fast sort of the remaining suffixes. Perhaps mainly because they avoid recursion, these algorithms are both fast and lightweight:

**IT** Itoh & Tanaka select suffixes $i$ of "type B" — those satisfying $\boldsymbol{x}[i] \leq \boldsymbol{x}[i+1]$ — for complete sorting, thus inducing a sort of the remaining suffixes.

**S** Seward sorts certain 1-groups, using the results to induce sorts of corresponding 2-groups, an approach that also forms the basis of Algorithms MF & SS.

Induced Copying Algorithms (continued)

**BK** Burkhardt & Kärkkäinen use a small integer $h$ to form a "sample" $S$ of suffixes that is then $h$-sorted; using a technique reminiscent of the recursive algorithms, the resulting $h$-ranks are then used to induce a complete sort of all the suffixes.

**M** The as-yet-unpublished algorithm of Maniscalco computes $\mathrm{ISA}_{\boldsymbol{x}}$ using an iterative technique that, beginning with 1-groups, uses $h$-groups to induce the formation of $(h+1)$-groups.

## Prototype of an Induced Copying Algorithm (IT)

initialize $\mathsf{SA} \leftarrow \mathsf{SA}_1$
— $\mathsf{head}[1..\sigma]$ *and* $\mathsf{tail}[1..\sigma]$ *mark 1-group boundaries*
— $\mathsf{part}[1..\sigma]$ *marks* $A/B$ *partition of each 1-group*
**for** $h \leftarrow 1$ **to** $\sigma$ **do**
  suffixsort $\Big(\mathsf{SA}\Big[\mathsf{part}[h]\Big], \mathsf{SA}\Big[\mathsf{part}[h]{+}1\Big], \ldots,$
    $\mathsf{SA}\Big[\mathsf{tail}[h]\Big]\Big)$
**for** $j \leftarrow 1$ **to** $n$ **do**
  $i \leftarrow \mathsf{SA}[j]$
  **if** $\mathsf{type}[i{-}1] = A$ **then**
    $\lambda \leftarrow \boldsymbol{x}[i{-}1]$
    $\mathsf{SA}\Big[\mathsf{head}[\lambda]\Big] \leftarrow i{-}1$
    $\mathsf{head}[\lambda] \leftarrow \mathsf{head}[\lambda]{+}1$

# Experimental Results

Future Directions

The profusion of SACAs discovered in a relatively short period of time is a tribute to the ingenuity of stringologists!

But there still remain interesting questions:

(1) Is it possible to devise an algorithm (probably nonrecursive) that satisfies all three criteria (fast, lightweight, linear)?

(2) The calculation of repetitions and repeats in strings does not require an SA/ST that satisfies lexorder. Can faster algorithms be found for these **valid** suffix arrays?

(3) The number $\rho(n)$ of **maximal periodicities** in any string of length $n$ probably satisfies $\rho(n) < n$ and almost certainly decreases sharply as alphabet size increases. Can algorithms be found to compute maximal periodicities that are more direct and that avoid the calculation of suffix arrays altogether?