A Taxonomy of Suffix Array Construction Algorithms*

Simon J. Puglisi[1], W. F. Smyth[1,2], and Andrew Turpin[3]

[1] Department of Computing, Curtin University, GPO Box U1987
Perth WA 6845, Australia
e-mail: puglissj@computing.edu.au

[2] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton ON L8S 4K1, Canada
e-mail: smyth@mcmaster.ca
www.cas.mcmaster.ca/cas/research/groups.shtml

[3] School of Computer Science & Information Technology
RMIT University, GPO Box 2476V
Melbourne V 3001, Australia
e-mail: aht@cs.rmit.edu.au

Abstract. 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, and especially in the last few years, suffix array construction algorithms have proliferated in bewildering abundance. This survey paper attempts to provide simple high-level descriptions of these numerous algorithms that highlight both their distinctive features and their commonalities, while avoiding as much as possible the complexities of implementation details. 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.

1 Introduction

Suffix arrays were introduced in 1990 by Manber & Myers [MM90, MM93], along with algorithms for their construction and use as a space-saving alternative to suffix trees. In the intervening fifteen years there have certainly been hundreds of research articles published on the construction and use of suffix trees and their variants. Over that period, it has been shown that

- practical space-efficient suffix array construction algorithms (SACAs) exist that require worst-case time linear in string length [KA03, KS03];
- SACAs exist that are even faster in practice, though with supralinear worst-case construction time requirements [LS99, BK03, MF04, M05];

*Supported in part by grants from the Natural Sciences & Engineering Research Council of Canada and the Australian Research Council.

- any problem whose solution can be computed using suffix trees is solvable with the same asymptotic complexity using suffix arrays [AKO04].

Thus suffix arrays have become the data structure of choice for many, if not all, of the string processing problems to which suffix tree methodology is applicable.

In this survey paper we do not attempt to cover the entire suffix array literature. Our more modest goal is to provide an overview of SACAs, in particular those modeled on the efficient use of main memory — we exclude the substantial literature (for example, [CF02]) that discusses strategies based on the use of secondary storage. Further, we deal with the construction of compressed ("succinct") suffix arrays only insofar as they relate to standard SACAs. For example, algorithms such as those of Grossi et al. and references therein [GGV04] are not covered.

Section 2 provides an overview of the SACAs known to us, organized into a "taxonomy" based primarily on the methodology used. As with all classification schemes, there is room for argument: there are many cross-connections between algorithms that occur in disjoint subtrees of the taxonomy, just as there may be between species in a biological taxonomy. Our aim is to provide as comprehensive and, at the same time, as accessible a description of SACAs as we can.

Also in Section 2 we present the vocabulary to be used for the structured description of each of the algorithms that will be given in Section 3. Then in Section 4, we report on the results of experimental results on many of the algorithms described and so draw conclusions about their relative speed and space-efficiency.

2 Overview

We consider throughout a finite nonempty ***string*** $\boldsymbol{x} = \boldsymbol{x}[1..n]$ of ***length*** $n \geq 1$, defined on an ***indexed*** alphabet Σ; that is,

- the letters $\lambda_j, j = 1, 2, \ldots, \sigma$ of $|\Sigma|$ are ordered: $\lambda_1 < \lambda_2 < \cdots < \lambda_\sigma$;
- an array $A[\lambda_1..\lambda_\sigma]$ can be defined in which, for every $j \in 1..\sigma$, $A[\lambda_j]$ is accessible in constant time;
- $\lambda_\sigma - \lambda_1 \in O(n)$.

Essentially, we assume that Σ can be treated as a sequence of integers whose range is not too large. Typically, the λ_j may be represented by ASCII codes 0..255 (English alphabet) or binary integers 00..11 (DNA) or simply bits, as the case may be. We shall generally assume that a letter can be stored in a byte and that n can be stored in one computer word (four bytes).

The use of terminology not defined here follows [S03].

We are interested in computing the ***suffix array*** of $\boldsymbol{x}$, which we write $\mathrm{SA}_{\boldsymbol{x}}$ or just SA; that is, an array SA$[1..n]$ in which SA$[j] = i$ iff $\boldsymbol{x}[i..n]$ is the j^{th} suffix of $\boldsymbol{x}$ in (ascending) lexicographical order (***lexorder***). For simplicity we will frequently refer to $\boldsymbol{x}[i..n]$ simply as "suffix i"; also, it will often be convenient for processing to incorporate into $\boldsymbol{x}$ at position n an ending sentinel \$ assumed to be less than any λ_j.

Then, for example, on alphabet $\Sigma = \{\$, a, b, c, d, e\}$:

	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	\$
SA =	12	11	8	1	4	6	9	2	5	7	10	3

Thus SA tells us that $\boldsymbol{x}[12..12] =$ \$ is the least suffix, $\boldsymbol{x}[11..12] = a$\$ the second least, and so on (alphabetical ordering of the letters assumed). Note that SA is always a permutation of $1..n$.

Often used in conjunction with $\mathrm{SA}_{\boldsymbol{x}}$ is the ***lcp array*** lcp$[1..n]$: for every $j \in 2..n$, lcp$[j]$ is just the ***longest common prefix*** of suffixes SA$[j-1]$ and SA$[j]$. In our example:

	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	\$
SA =	12	11	8	1	4	6	9	2	5	7	10	3
lcp =	$-$	0	1	4	1	1	0	3	0	0	0	2

Thus the longest common prefix of suffixes 11 and 8 is 1, that of suffixes 8 and 1 is 4. Since lcp can be computed in linear time from $\mathrm{SA}_{\boldsymbol{x}}$ [KLAAP01, M04], also as a byproduct of some of the SACAs discussed below, we do not consider its construction further in this paper. However, the ***average lcp*** — that is, the average $\overline{\mathrm{lcp}}$ of the $n-1$ integers in the lcp array — is as we shall see a useful indicator of the relative efficiency of certain SACAs, notably Algorithm S.

We remark that both SA and lcp can be computed in linear time by a preorder traversal of a suffix tree.

Many of the SACAs also make use of the ***inverse suffix array***, written $\mathrm{ISA}_{\boldsymbol{x}}$ or ISA: an array ISA$[1..n]$ in which

$$\mathrm{ISA}[i] = j \iff \mathrm{SA}[j] = i.$$

ISA$[i] = j$ therefore says that suffix i has ***rank*** j in lexorder. Continuing our example:

	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	\$
ISA =	4	8	12	5	9	6	10	3	7	11	2	1

Thus ISA tells us that suffix 1 has rank 4 in lexorder, suffix 2 rank 8, and so on. Note that ISA is also a permutation of $1..n$, and so SA and ISA are computable, one from the other, in $\Theta(n)$ time:

for $j \leftarrow 1$ **to** n **do**
  $\mathrm{SA}\big[\mathrm{ISA}[j]\big] \leftarrow j$

As shown in Figure 1, this computation can if required also be done in place.

Many of the algorithms we shall be describing depend upon a partial sort of some or all of the suffixes of $\boldsymbol{x}$, partial because it is based on an ordering of the prefixes of these suffixes that are of length $h \geq 1$. We refer to this partial ordering as an ***h-ordering*** of suffixes into ***h-order***, and to the process itself as an ***h-sort***. If two or more suffixes are equal under h-order, we say that they have the same ***h-rank*** and therefore fall into the same ***h-group***; they are accordingly said to be ***h-equal***. Usually an h-sort is ***stable***, so that any previous ordering of the suffixes is retained within each h-group.

```
for j ← 1 to n do
  i ← SA[j]
 — Negative entries already processed
  if i > 0 then
    j0, j' ← j
    repeat
      temp ← SA[i]; SA[i] ← −j'
      j' ← i; i ← temp
    until i = j0
    SA[i] ← −j'
  else
    SA[j] ← −i
```

Figure 1: Algorithm for computing ISA from SA in place

The results of an h-sort are often stored in an approximate suffix array, written SA_h, and/or an approximate inverse suffix array, written ISA_h. Here is the result of a 1-sort on all the suffixes of our example string:

	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 parentheses in SA_1 enclose 1-groups not yet reduced to a single entry, thus not yet in final sorted order. Note that SA_h retains the property of being a permutation of $1..n$, while ISA_h may not. Depending on the requirements of the particular algorithm, ISA_h may as shown express the h-rank of each h-group in various ways:

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

Compare the result of a 3-sort:

	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	\$
$\mathrm{SA}_3 = 12$	11	(1	8)	4	6	(2	9)	5	7	10	3	
$\mathrm{ISA}_3 = 3$	7	12	5	9	6	10	3	7	11	2	1	
or 4	8	12	5	9	6	10	4	8	11	2	1	
or 3	6	10	4	7	5	8	3	6	9	2	1	

Observe that an $(h{+}1)$-sort is a ***refinement*** of an h-sort: all members of an $(h{+}1)$-group belong to a single h-group.

We now have available a vocabulary sufficient to characterize the main species of SACA as follows.

(1) Prefix-Doubling

First a fast 1-sort is performed (since Σ is indexed, bucket sort can be used); this yields SA_1/ISA_1. Then for every $h = 1, 2, \ldots$, $\mathrm{SA}_{2h}/ISA_{2h}$ are computed in $\Theta(n)$ time from SA_h/ISA_h until every $2h$-group is a singleton. The time required is therefore $O(n \log n)$. There are two algorithms in this class: MM [MM90, MM93] and LS [S98, LS99].

(2) Recursive

Form strings $\boldsymbol{x'}$ and $\boldsymbol{y}$ from $\boldsymbol{x}$, then show that if $\mathrm{SA}_{\boldsymbol{x'}}$ is computed, therefore $\mathrm{SA}_{\boldsymbol{y}}$ and finally $\mathrm{SA}_{\boldsymbol{x}}$ can be computed in $O(n)$ time. Hence the problem of computing $\mathrm{SA}_{\boldsymbol{x'}}$ recursively replaces the computation of $\mathrm{SA}_{\boldsymbol{x}}$. Since $|\boldsymbol{x'}|$ is always chosen so as to be less than $2|\boldsymbol{x}|/3$, the overall time requirement of these algorithms is $\Theta(n)$. There are three main algorithms in this class: KA [KA03], KS [KS03] and KJP [KJP04].

(3) Induced Copying

The key insight here 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. The approach however is nonrecursive: an efficient suffix sorting technique (for example, [BM93, MBM93, M97, BS97, SZ04]) is invoked for the selected subset of suffixes. The general idea seems to have been first proposed by Burrows & Wheeler [BW94], but it has been implemented in quite different ways [IT99, S00, MF04, SS05, BK03, M05]. In general, these methods are very efficient in practice, but may have worst-case asymptotic complexity 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" (that is, on collections of large real-world data sets such as [H04]);
- are ***lightweight*** — that is, use a small amount of working storage in addition to the $5n$ bytes required by $\boldsymbol{x}$ and $\mathrm{SA}_{\boldsymbol{x}}$.

To date none of the SACAs that has been proposed achieves all of these objectives.

Figure 2 presents our taxonomy of the fourteen species of SACA that have been recognized so far; Table 1 summarizes their time and space requirements.

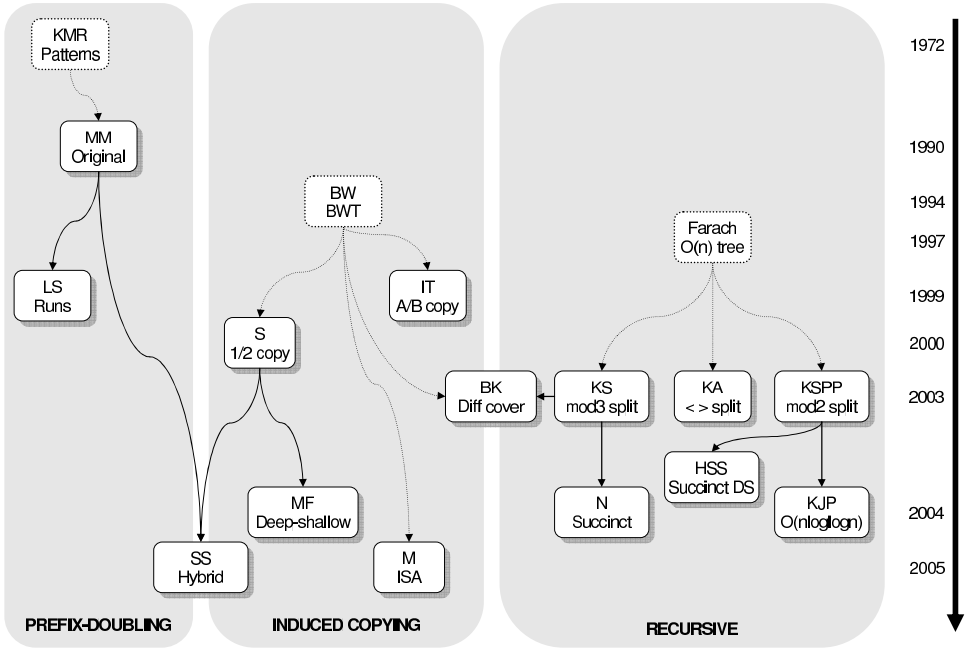

Figure 2: Taxonomy of suffix array construction algorithms

Table 1: Performance summary of the construction algorithms. Speed is relative to MF, the fastest in our experiments, and Memory is given in the number of bytes required including space required for the suffix array and input.

Algorithm	Worst Case	Speed	Memory
Prefix-Doubling			
MM [MM93]	$O(n \log n)$	16	$8n$
LS [LS99]	$O(n \log n)$	1.7	$8n$
Recursive			
KA [KA03]	$O(n)$	2.2	13-14n
KS [KS03]	$O(n)$	2.8	10-13n
KSPP [KSPP03]	$O(n)$	–	–
HSS [HSS03]	$O(n)$	–	–
KJP [KJP04]	$O(n \log\log n)$	2.1	13-16n
Induced Copying			
IT [IT99]	$O(n^2 \log n)$	4	$5n$
S [S00]	$O(n^2 \log n)$	2.1	$5n$
BK [BK03]	$O(n \log n)$	2.1	5-6n
MF [MF04]	$O(n^2 \log n)$	1	$5n$
SS [SS05]	$O(n^2)$	1	9-10n
M [M05]	$O(n^2 \log n)$	1	5-7n
Suffix Tree			
K [K99]	$O(n \log \sigma)$	4	15-20n

3 The Algorithms

3.1 Prefix-Doubling Algorithms [KMR72]

Here we consider algorithms that, given an h-order SA_h of the suffixes of $\boldsymbol{x}$, $h \geq 1$, compute a $2h$-order in $O(n)$ time. Thus prefix-doubling algorithms require at most $\log_2 n$ steps to complete the suffix sort and execute in $O(n \log n)$ time in the worst case.

Normally prefix-doubling algorithms initialize SA_1 for $h = 1$ using a linear-time bucket sort. The main idea [KMR72] is as follows:

Observation 1. *Suppose that* SA_h *and* ISA_h *have been computed for some* $h > 0$, *where* $i = \mathrm{SA}_h[j]$ *is the* j^{th} *suffix in* h*-order and* h-$\mathrm{rank}[i] = \mathrm{ISA}_h[i]$. *Then a sort using the integer pairs*

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

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

The two main prefix-doubling algorithms differ primarily in their application of this observation:

- Algorithm MM does an implicit $2h$-sort by performing a left-to-right scan of SA_h that induces the $2h$-rank of $\mathrm{SA}_h[j]-h$, $j = 1, 2, \ldots, n$;
- Algorithm LS explicitly sorts each h-group using the ternary-split quicksort (TSQS) of Bentley & McIlroy [BM93].

Manber & Myers [MM90, MM93]

Algorithm MM employs Observation 1 as follows:

> If SA_h is scanned left to right (thus in h-order of the suffixes), $j = 1, 2, \ldots, n$, then the suffixes
>
> $$i-h = \mathrm{SA}_h[j]-h > 0$$
>
> are necessarily scanned in $2h$-order within their respective h-groups in SA_h.

After the bucket sort that forms SA_1, MM computes ISA_1 by specifying as the h-rank of each suffix i in SA_1 the leftmost position in SA_1 (the head) of its group:

	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

To form SA_2, we consider positive values of $i-1 = \mathrm{SA}_1[j]-h$ for $j = 1, 2, \ldots, n$:

- for $j = 1, 7, 8, 9, 10$, identify in 2-order the suffixes $11, (1, 8), 4, 6$ beginning with a;

- for $j = 11, 12$, identify in 2-order the 2-equal suffixes $(2, 9)$ beginning with b;
- for $j = 3, 6$, identify in 2-order the 2-equal suffixes $(3, 10)$ beginning with e.

Of course groups that are singletons in SA_1 remain singletons in SA_2, and so, after relabeling the groups, we get

$$\begin{array}{rcccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ \mathrm{SA}_2 = & 12 & 11 & (1 & 8) & 4 & 6 & (2 & 9) & 5 & 7 & (3 & 10) \\ \mathrm{ISA}_2 = & 3 & 7 & 11 & 5 & 9 & 6 & 10 & 3 & 7 & 11 & 2 & 1 \end{array}$$

To form SA_4, we consider positive values of $i-2 = \mathrm{SA}_2[j]-h$ for $j = 1, 2, \ldots, n$:

- for $j = 11, 12$, we identify in 4-order the 4-equal suffixes $(1, 8)$ beginning with ab;
- for $j = 2, 5$, we identify in 4-order the 4-distinct suffixes $9, 2$ beginning with be;
- for $j = 1, 9$, we identify in 4-order the 4-distinct suffixes $10, 3$ beginning with ea.

Hence:

$$\begin{array}{rcccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ \mathrm{SA}_4 = & 12 & 11 & (1 & 8) & 4 & 6 & 9 & 2 & 5 & 7 & 10 & 3 \\ \mathrm{ISA}_4 = & 3 & 8 & 12 & 5 & 9 & 6 & 10 & 3 & 7 & 11 & 2 & 1 \end{array}$$

The final $\mathrm{SA} = \mathrm{SA}_8$ and $\mathrm{ISA} = \mathrm{ISA}_8$ are achieved after one further doubling that separates the *abea*'s $(1, 8)$ into $8, 1$.

Algorithm MM is complicated by the requirement to keep track of the head of each h-group, but can nevertheless be implemented using as few as $4n$ bytes of storage, in addition to that required for $\boldsymbol{x}$ and SA. It can be represented conceptually as shown in Figure 3.

A time- and space-efficient implementation of MM is available at [M97].

```
h ← 1
initialize SA_1, ISA_1
while some h-group not a singleton
  for j ← 1 to n do
    i ← SA_h[j]−h
    if i > 0 then
      q ← head[h-group[i]]
      SA_2h[q] ← i
      head[h-group[i]] ← q+1
  compute ISA_2h  — update 2h-groups
  h ← 2h
```

Figure 3: Algorithm MM

Larsson & Sadakane [S98, LS99]

After using TSQS to form SA_1, Algorithm LS computes ISA_1 using the *rightmost* (rather than, as in Algorithm MM, the leftmost) position of each group in SA_1 to identify h-rank$[i]$.

	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	\$
$\mathrm{SA}_1 =$	12	(1	4	6	8	11)	(2	9)	5	7	(3	10)
$\mathrm{ISA}_1 =$	6	8	12	6	9	6	10	6	8	12	6	1

In addition to identifying h-groups in SA_h that are not singletons, LS also identifies ***runs*** of consecutive positions that are singletons (fully sorted). For this purpose an array $\mathrm{L} = \mathrm{L}[1..n]$ is maintained, in which $\mathrm{L}[j] = \ell$ (respectively, $-\ell$) if and only if j is the leftmost position in SA_h of an h-group (respectively, run) of length ℓ:

	1	2	3	4	5	6	7	8	9	10	11	12
$\mathrm{L} =$	-1	5					2		-2		2	

Left-to-right processing of L thus allows runs to be skipped and non-singleton h-groups to be identified, in time proportional to the total number of runs and h-groups. TSQS is again used to sort the suffixes i in each of the identified h-groups according to keys $\mathrm{ISA}_h[i+h]$, thus yielding, by Observation 1, a collection of subgroups and subruns in $2h$-order. A straightforward update of L and ISA then yields stage $2h$:

	1	2	3	4	5	6	7	8	9	10	11	12
$\mathrm{SA}_2 =$	12	11	(1	8)	4	6	(2	9)	5	7	(3	10)
$\mathrm{ISA}_2 =$	4	8	12	5	9	6	10	4	8	12	2	1
$\mathrm{L} =$	-2		2		-2		2		-2		2	

A further doubling yields

	1	2	3	4	5	6	7	8	9	10	11	12
$\mathrm{SA}_4 =$	12	11	(1	8)	4	6	9	2	5	7	10	3
$\mathrm{ISA}_4 =$	4	8	12	5	9	6	10	4	7	11	2	1
$\mathrm{L} =$	-2		2		-8							

and then the final results SA_8 and ISA_8 are achieved as for Algorithm MM, with $\mathrm{L}[1] = -12$.

Observe that, like MM, LS maintains $\mathrm{ISA}_{2h}[i] = \mathrm{ISA}_h[i]$ for every suffix i that is a singleton in its h-group. However, unlike MM, LS avoids having to process every position in SA_h (see the **for** loop in Figure 3) by virtue of its use of the array L — in fact, once for some h, i is identified as a singleton, $\mathrm{SA}_h[i]$ is never accessed again.

We now remark that in fact L can be eliminated! L is not required to determine non-singleton h-groups because for every suffix i in such a group, $\mathrm{ISA}_h[i]$ is by definition the rightmost position in the group. Thus, in particular, at the leftmost position j of the h-group, where $i = \mathrm{SA}_h[j]$, we can compute the length ℓ of the group from $\ell = \mathrm{ISA}_h[i] - j + 1$. Of course L also keeps track of runs of fully sorted suffixes in SA_h,

but, as just remarked, positions in SA_h corresponding to such runs are thereafter unused — it turns out that they can be recycled to perform the run-tracking role. This implementation requires that SA_h be reconstructed from ISA_h in order to provide the final output, a straightforward procedure (see Section 2).

Algorithm LS thus requires $4n$ additional bytes of storage (the integer array ISA), just like MM. As shown in [LS99], LS executes in $O(n \log n)$ time, again the same as MM; however, in practice its running time is usually several times faster.

3.2 Recursive Algorithms [F97]

In this section we consider a family of algorithms that were all discovered in 2003 or later, that are recursive in nature, and that generally execute in worst-case time linear in string length. All are based on an idea first put forward by Farach [F97] for linear-time suffix *tree* construction of strings on an indexed alphabet: they depend on an initial assignment of ***type*** to each suffix (position) in $\boldsymbol{x}$ that separates the suffixes into two or more classes. Thus the recursion in all cases is based on a ***split*** of the given string $\boldsymbol{x} = \boldsymbol{x}^{(0)}$ into disjoint (or almost disjoint) components (subsequences) that are transformed into strings we call $\boldsymbol{x}^{(1)}$ 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
- $\mathrm{SA}_{\boldsymbol{x}^{(0)}}$ 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, $\mathrm{SA}_{\boldsymbol{x}^{(i+1)}}$). To make this strategy efficient and effective, two requirements need to be met.

1. At each recursive step, ensure that

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

   thus the sum of the lengths of the strings processed by all recursive steps is

   $$|\boldsymbol{x}|(1+f+f^2+\ldots) < |\boldsymbol{x}|/\big(1-f\big).$$

   In fact, over all the algorithms proposed so far, $f \leq 2/3$, so that the sum of the lengths is guaranteed to be less than $3|\boldsymbol{x}|$ — for most of them $\leq 2|\boldsymbol{x}|$.

2. Devise an 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. Ensure moreover that the time required for *semisort* is linear in the length of the string being processed.

Clearly suffix-sorting algorithms satisfying the above description will compute $\mathrm{SA}_{\boldsymbol{x}}$ (or equivalently $\mathrm{ISA}_{\boldsymbol{x}}$) of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ in $\Theta(n)$ time. The structure of such algorithms is shown in Figure 4.

All of the algorithms discussed in this subsection compute $\boldsymbol{x'}$ (that is, $\boldsymbol{x}^{(1)}$) and $\boldsymbol{y}$ (that is, $\boldsymbol{y}^{(1)}$) from $\boldsymbol{x}$ (that is, $\boldsymbol{x}^{(0)}$) in similar ways: the alphabet of the split strings

```
procedure construct(x; SA)
split(x; x', y)
semisort(x'; ISA')
if ISA' contains duplicate ranks then
   construct(ISA'; SA_x = SA')
else
   invert(ISA_x = ISA'; SA_x')
induce(SA_x', ISA_x'; SA_y)
merge(SA_x', SA_y; SA_x)
```

Figure 4: General algorithm for recursive SA construction

is in fact the set of suffixes (positions) $1..n$ in $\boldsymbol{x}$, so that $\boldsymbol{x'}$ and $\boldsymbol{y}$ together form a permutation of $1..n$.

Attention then focuses on computing the ranks of the suffixes (positions) i of $\boldsymbol{x}$ that occur in $\boldsymbol{x'}$: we call this sequence (string) of ranks $\mathrm{ISA}_{\boldsymbol{x'}}$, where for $j = 1, 2, \ldots, |\boldsymbol{x'}|$, $\mathrm{ISA}_{\boldsymbol{x'}}[i]$ gives the rank of suffix $i = \boldsymbol{x'}[j]$ of $\boldsymbol{x}$.

Procedure *semisort* computes an approximation ISA' of $\mathrm{ISA}_{\boldsymbol{x'}}$ that ultimately, at some level of recursion, becomes exact — and so we may write $\mathrm{ISA}_{\boldsymbol{x'}} = \mathrm{ISA}'$, then *invert* $\mathrm{ISA}_{\boldsymbol{x'}}$ to form $\mathrm{SA}_{\boldsymbol{x'}}$.

If however ISA' is not exact, then it is used as the input string for a recursive call of the *construct* procedure, thus yielding the suffix array, SA' say, of ISA' — the key observation made here, common to all the recursive algorithms, is that since SA' is the suffix array for the (approximate) ranks of the suffixes identified by $\boldsymbol{x'}$, it is therefore the suffix array for those suffixes themselves. We may accordingly write $\mathrm{SA}_{\boldsymbol{x'}} = \mathrm{SA}'$.

In our discussion below of these algorithms, we focus on the nature of *split* and *semisort* and their consequences for the *induce* and *merge* procedures.

Ko & Aluru [KA03]

Algorithm KA's *split* procedure assigns suffixes $i < n$ in left-to-right order to a sequence $\mathcal{S}$ (respectively, $\mathcal{L}$) iff $\boldsymbol{x}[i..n] <$ (respectively, $>$) $\boldsymbol{x}[i+1..n]$. Suffix n (\$) is assigned to both $\mathcal{S}$ and $\mathcal{L}$. Since $\boldsymbol{x}[i] = \boldsymbol{x}[i{+}1]$ implies that suffixes i and $i{+}1$ belong to the same sequence, it follows that the KA *split* requires time linear in $\boldsymbol{x}$.

Then $\boldsymbol{x'}$ is formed from the sequence of suffixes of smaller cardinality, $\boldsymbol{y}$ from the sequence of larger cardinality. Hence for KA, $|\boldsymbol{x'}| \leq |\boldsymbol{x}|/2$.

For example,

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	b	a	d	d	a	d	d	a	c	c	a	\$
type $=$	L	S	L	L	S	L	L	S	L	L	L	S/L

yields $|\mathcal{S}| = 4$, $|\mathcal{L}| = 9$, $\boldsymbol{x'} = 2\,5\,8\,12$, $\boldsymbol{y} = 1\,3\,4\,6\,7\,9\,10\,11\,12$.

For every $j \in 1..|\boldsymbol{x'}|$, KA's *semisort* procedure forms $i = \boldsymbol{x'}[j]$, $i_1 = \boldsymbol{x'}[j+1]$ ($i_1 = \boldsymbol{x'}[j]$ if $j = |\boldsymbol{x'}|$), and then performs a radix sort on the resulting substrings $\boldsymbol{x}[i..i_1]$, a calculation that requires $\Theta(n)$ time. The result of this sort is a ranking ISA'

of the substrings $\boldsymbol{x}[i..i_1]$, hence an approximate ranking of the suffixes (positions) $i = \boldsymbol{x'}[j]$. In our example, *semisort* yields

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	b	a	d	d	a	d	d	a	c	c	a	\$
$\boldsymbol{x'} =$		2			5			8				12
$\mathrm{ISA}' =$		3			3			2				1

If after *semisort* the entries (ranks) in ISA' are distinct, then a complete ordering of the suffixes of $\boldsymbol{x'}$ has been computed ($\mathrm{ISA}' = \mathrm{ISA}_{\boldsymbol{x'}}$); if not, then as indicated in Figure 4, the *construct* procedure is recursively called on ISA'. In our example, one recursive call suffices for a complete ordering $(12, 8, 5, 2)$ of the suffixes of $\boldsymbol{x'}$, yielding $\mathrm{ISA}_{\boldsymbol{x'}} = 4321$.

At this point KA deviates from the pattern of Figure 4 in two ways: it combines the *induce* and *merge* procedures into a single KA-merge (see Figure 5), and it computes $\mathrm{SA}_{\boldsymbol{x}}$ directly without reference to $\mathrm{ISA}_{\boldsymbol{x}}$ [1].

```
initialize SA ← SA_1, head[1..α], tail[1..α]
for i ← |x'| downto 1 do
   λ ← x[x'[i]]
   SA[tail[λ]] ← x'[i]
   tail[λ] ← tail[λ]−1
for j ← 1 to n do
   i ← SA[j]
   if type[i−1] = L then
      λ ← x[i−1]
      SA[head[λ]] ← i−1
      head[λ] ← head[λ]+1
```

Figure 5: Algorithm KA-merge

First SA_1 is computed, yielding 1-groups for which the leftmost and rightmost positions are specified in arrays $\mathrm{head}[1..\alpha]$ and $\mathrm{tail}[1..\alpha]$, respectively. Since in each 1-group all the S-suffixes are lexicographically greater than all the L-suffixes, and since the S-suffixes have been sorted, KA-*merge* can place all the S-suffixes in their final positions in SA — each time this is done, the tail for the current group is decremented by one. (In this description, we assume that $|\mathcal{S}| \leq |\mathcal{L}|$; obvious adjustments yields a corresponding approach for the case $|\mathcal{L}| < |\mathcal{S}|$.)

The SA at this stage is shown below, with "$-$" denoting an empty position:

	1	2	3	4	5	6	7	8	9	10	11	12
$\mathrm{SA} =$	12	$(-$	8	5	2)	$(-)$	$(-$	$-)$	$(-$	$-$	$-$	$-)$
$\mathrm{type} =$	S	L	S	S	S	L	L	L	L	L	L	L

To sort the L-suffixes, we scan SA left to right. For each suffix position $i = \mathrm{SA}[j]$ that we encounter in the scan, if $i-1$ is an L-suffix still awaiting sorting (not yet placed in the SA), we place $i-1$ at the head of its group in SA and increment the

[1] In [KA03] it is claimed that the ISA must be built in unison with the SA for this procedure to work, but we have found that this is actually unnecessary.

head of the group by one. Suffix $i-1$ is now sorted and will not be moved again. The correctness of this procedure depends on the fact that when the scan of SA reaches position j, SA[j] is already in its final position. In our example, placements begin when $j = 1$, so that $i = \text{SA}[1] = 12$. Since suffix $i-1 = 11$ is type L, it is placed at the front of the a group (of which it happens to be the only member):

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \text{SA} = & 12 & (11 & 8 & 5 & 2) & (-) & (- & -) & (- & - & - & -) \\ \text{type} = & S & L & S & S & S & L & L & L & L & L & L & L \end{array}$$

Next the scan reaches $j = 2$, $i = \text{SA}[2] = 11$, and we place $i-1 = 10$ at the front of the c group at SA[7] and increment the group head.

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \text{SA} = & 12 & (11 & 8 & 5 & 2) & (-) & (10 & -) & (- & - & - & -) \\ \text{type} = & S & L & S & S & S & L & L & L & L & L & L & L \end{array}$$

The scan continues until finally

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \text{SA} = & 12 & 11 & 8 & 5 & 2 & 1 & 10 & 9 & 7 & 4 & 6 & 3 \end{array}$$

Algorithm KA can be implemented to use only $4n$ bytes plus $1.25n$ bits in addition to the storage required for $\boldsymbol{x}$ and SA.

Kärkkäinen & Sanders [KS03]

The *split* procedure of Algorithm KS first separates the suffixes i of $\boldsymbol{x}$ into sequences $\mathcal{S}_1$ (every third suffix in $\boldsymbol{x}$: $i \equiv 1 \bmod 3$) and $\mathcal{S}_{02}$ (the remaining suffixes: $i \not\equiv 1 \bmod 3$). Thus in this algorithm three types $0, 1, 2$ are identified: $\boldsymbol{x'}$ is formed from $\mathcal{S}_{02}$ by

$$\boldsymbol{x'} = (i \equiv 2 \bmod 3)\ (i \equiv 0 \bmod 3),$$

while $\boldsymbol{y}$ is formed directly from $\mathcal{S}_1$. For our example string

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \boldsymbol{x} = & b & a & d & d & a & d & d & a & c & c & a & \$ \end{array}$$

we find $\boldsymbol{x'} = (2\,5\,8\,11)(3\,6\,9\,12)$, $\boldsymbol{y} = 1\,4\,7\,10$. Note that $|\boldsymbol{x'}| \le \lfloor 2|\boldsymbol{x}|/3 \rfloor$.

Construction of ISA$'$ using *semisort* begins with a linear-time 3-sort of suffixes $i \in \mathcal{S}_{02}$ based on triples $t_i = \boldsymbol{x}[i..i+2]$. Thus a 3-order of these suffixes is established for which a 3-rank r_i can be computed, as illustrated by our example:

$$\begin{array}{lcccccccc} i & 2 & 3 & 5 & 6 & 8 & 9 & 11 & 12 \\ t_i & add & dda & add & dda & acc & cca & a\$- & \$-- \\ r_i & 4 & 6 & 4 & 6 & 3 & 5 & 2 & 1 \end{array}$$

These ranks enable ISA$'$ to be formed for $\boldsymbol{x'}$:

$$\begin{array}{rcccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} \\ \text{ISA}' = & (4 & 4 & 3 & 2) & (6 & 6 & 5 & 1) \end{array}$$

As with Algorithm KA, one recursive call on $\boldsymbol{x'} = 44326651$ suffices to complete the ordering, yielding $\mathrm{ISA}_{\boldsymbol{x'}} = 54328761$ — this gives the ordinal ranks in $\boldsymbol{x}$ of the suffixes $\boldsymbol{x'} = 2\,5\,8\,11\,3\,6\,9\,12$.

The *induce* procedure sorts the suffixes specified by $\boldsymbol{y}$ based on the ordering $\mathrm{ISA}_{\boldsymbol{x'}}$. First $\mathrm{SA}_{\boldsymbol{x'}} = 12\,11\,8\,5\,2\,9\,6\,3$ is formed by linear-time processing of $\mathrm{ISA}_{\boldsymbol{x'}}$. Then a left-to-right scan of $\mathrm{SA}_{\boldsymbol{x'}}$ allows us to identify suffixes $i \equiv 2 \bmod 3$ in increasing order of rank and thus to select letters $\boldsymbol{x}[i{-}1]$, $i{-}1 \equiv 1 \bmod 3$, in the same order. A stable bucket sort of these letters will then provide the suffixes of $\boldsymbol{y}$ in increasing lexorder. In our example $\mathrm{SA}_{\boldsymbol{x'}}[2..5] = 11\,8\,5\,2$, and so we consider $\boldsymbol{x}[10] = c$, $\boldsymbol{x}[7] = \boldsymbol{x}[4] = d$, $\boldsymbol{x}[1] = b$. A stable sort yields *bcdd* corresponding to $\mathrm{SA}_{\boldsymbol{y}} = 1\,10\,7\,4$.

Thus we may suppose that $\mathrm{SA}_{\boldsymbol{x'}}$ and $\mathrm{SA}_{\boldsymbol{y}}$ are both in sorted order of suffix. The KS *merge* procedure may then be thought of as a straightforward merge of these two strings into the output array $\mathrm{SA}_{\boldsymbol{x}}$, where at each step we need to decide in constant time whether suffix i_{02} of $\mathrm{SA}_{\boldsymbol{x'}}$ is greater or less than suffix i_1 of $\mathrm{SA}_{\boldsymbol{y}}$. Observing that $i_1{+}1 \equiv 2 \bmod 3$ and $i_1{+}2 \equiv 0 \bmod 3$, we identify two cases:

- if $i_{02} \equiv 2 \bmod 3$, $i_{02}+1 \equiv 0 \bmod 3$, and so it suffices to compare the pairs $\big(\boldsymbol{x}[i_{02}], \mathrm{rank}(i_{02}{+}1)\big)$ and $\big(\boldsymbol{x}[i_1], \mathrm{rank}(i_1{+}1)\big)$;
- if $i_{02} \equiv 0 \bmod 3$, $i_{02}+2 \equiv 2 \bmod 3$, and so it suffices to compare the triples $\big(\boldsymbol{x}[i_{02}..i_{02}{+}1], \mathrm{rank}(i_{02}{+}2)\big)$ and $\big(\boldsymbol{x}[i_1..i_1{+}1], \mathrm{rank}(i_1{+}2)\big)$.

We now observe that each of the ranks required by these comparisons is available in constant time from $\mathrm{ISA}_{\boldsymbol{x'}}$! For if $i \equiv 2 \bmod 3$, then

$$\mathrm{rank}(i) = \mathrm{ISA}_{\boldsymbol{x'}}\big[\lfloor (i{+}1)/3 \rfloor\big],$$

while if $i \equiv 0 \bmod 3$, then

$$\mathrm{rank}(i) = \mathrm{ISA}_{\boldsymbol{x'}}\big[\lfloor (n{+}1)/3 \rfloor + \lfloor i/3 \rfloor\big].$$

Thus the merge of the two lists requires $\Theta(n)$ time.

Excluding $\boldsymbol{x}$ and SA, Algorithm KS can be implemented in $6n$ bytes of working storage. A recent variant of KS [N05] permits construction of a succinct suffix array in $O(n)$ time using only $O(n \log\sigma \log^q_\sigma n)$ bits of working memory, where $q = \log_2 3$.

Kim, Jo & Park [KSPP03, HSS03, KJP04]

The KJP *split* procedure adopts the same approach as Farach's suffix tree construction algorithm [F97]: it forms $\boldsymbol{x'}$, the string of odd suffixes (positions) $i \equiv 1 \bmod 2$ in $\boldsymbol{x}$, and the corresponding string $\boldsymbol{y}$ of even positions. $\mathrm{ISA}_{\boldsymbol{x'}}$ is then formed by a recursive sort of the suffixes identified by $\boldsymbol{x'}$. Algorithm KJP is not quite linear in its operation, running in $O(n\mathrm{loglog}n)$ worst-case time.

For KJP we modify our example slightly to make it more illustrative:

$$\begin{array}{rc} & \begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \end{array} \\ \boldsymbol{x} = & \begin{array}{cccccccccccc} b & a & d & d & d & d & a & c & c & a & \$ \end{array} \end{array}$$

yielding $\boldsymbol{x'} = 1\,3\,5\,7\,9\,11$, $\boldsymbol{y} = 2\,4\,6\,8\,10$.

The KJP *semisort* 2-sorts prefixes $p_i = \boldsymbol{x}[i..i{+}1]$ of each odd suffix i and assigns to each an ordinal rank r_i:

i	11	7	1	9	3	5
p_i	\$−	ac	ba	ca	dd	dd
r_i	1	2	3	4	5	5

As in the other recursive algorithms, a new string ISA' is formed from these ranks; in our example,

$$\begin{array}{rc} & \scriptstyle 1\ 2\ 3\ 4\ 5\ 6 \\ \mathrm{ISA}' = & 3\ 5\ 5\ 2\ 4\ 1 \end{array}$$

As with the other recursive algorithms, one recursive call suffices to find $\mathrm{ISA}_{\boldsymbol{x'}} = 365241$ corresponding to $\boldsymbol{x'} = 1\,3\,5\,7\,9\,11$. At this point KJP computes the inverse array $\mathrm{SA}_{\boldsymbol{x'}} = 11\,7\,1\,9\,5\,3$. The KJP *induce* procedure can now compute $\mathrm{SA}_{\boldsymbol{y}}$, the sorted list of even suffixes, in a straightforward manner: first set $\mathrm{SA}_{\boldsymbol{y}}[i] \leftarrow \mathrm{SA}_{\boldsymbol{x'}}[i]-1$, and then sort $\mathrm{SA}_{\boldsymbol{y}}$ stably, using $\boldsymbol{x}\big[\mathrm{SA}_{\boldsymbol{y}}[i]\big]$ as the sort key for suffix $\mathrm{SA}_{\boldsymbol{y}}[i]$:

$$\begin{array}{rc} & \scriptstyle 1\ \ 2\ 3\ 4\ 5\ 6 \\ \mathrm{SA}_{\boldsymbol{x'}} = & 11\ 7\ 1\ 9\ 5\ 3 \\ \mathrm{SA}_{\boldsymbol{y}} = & 10\ 2\ 8\ 6\ 4 \end{array}$$

The KJP *merge* is more complex. In order to merge $\mathrm{SA}_{\boldsymbol{x'}}$ and $\mathrm{SA}_{\boldsymbol{y}}$ efficiently, we need to compute an array $C\big[1..\lceil n/2\rceil\big]$, in which $C[i]$ gives the number of suffixes in $\mathrm{SA}_{\boldsymbol{x'}}$ that lie between $\mathrm{SA}_{\boldsymbol{y}}[i]$ and $\mathrm{SA}_{\boldsymbol{y}}[i-1]$ in the final SA (with special attention to end conditions $i = 1$ and $i = |\boldsymbol{y}|+1$). In [KJP04] it is explained how C can be computed in $\log|\boldsymbol{x'}|$ time using a suffix array search (pattern-matching) algorithm described in [SKPP03]. We omit the details, however, for our example we would find

$$\begin{array}{rc} & \scriptstyle 1\ 2\ 3\ 4\ 5\ 6 \\ \mathrm{C} = & 0\ 1\ 1\ 0\ 1\ 1 \end{array}$$

With C in hand, merging is just a matter of using each $C[i]$ to count how many consecutive $\mathrm{SA}_{\boldsymbol{x'}}$ entries to insert between consecutive $\mathrm{SA}_{\boldsymbol{y}}$ entries.

There are two other algorithms which, like KJP, perform an odd/even split of the suffixes. Algorithm KSPP [KSPP03] was the first of these, and although its worst-case execution time is $\Theta(n)$, it is generally considered to be of only theoretical interest, mainly due to high memory requirements. On the other hand, Algorithm HSS [HSS03] uses "succinct data structures" [M99] effectively to construct a (succinct) suffix array in $O(n \log\log\sigma)$ time with only $\Theta(n\log\sigma)$ bits of working memory. (Compare the variant [N05] of Algorithm KS mentioned above.) It is not clear how practical these lightweight approaches are, since their succinctness may well adversely affect speed.

3.3 Induced Copying Algorithms [BW94]

The algorithms in this class are arguably the most diverse of the three main divisions of SACAs discussed in this paper. They are united by the idea that a (usually) complete sort of a selected subset of suffixes can be used to ***induce*** a fast sort of the remaining suffixes. This induced sort is similar to the *induce* procedures employed in the recursive SACAs; the difference is that some sort of iteration is used in place of the recursion. This replacement (of recursion by iteration) probably largely explains why several of the induced copying algorithms are faster in practice than

any of the recursive algorithms (as we shall discover in Section 4), eventhough none of these algorithms is linear in the worst case. In fact, their worst-case asymptotic complexity is generally $O(n^2 \log n)$. In terms of space requirements, these algorithms are lightweight: for many of them, use of additional working storage amounts to less than n bytes.

We begin with brief outlines of the induced copying algorithms:

- Itoh & Tanaka [IT99] 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.
- Seward [S00] on the other hand sorts certain 1-groups, using the results to induce sorts of corresponding 2-groups, an approach that also forms the basis of Algorithms MF [MF04] and SS [SS05].
- A third approach, due to Burkhardt & Kärkkäinen, uses 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.
- Finally, the as-yet-unpublished algorithm of Maniscalco [M05] 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.

Itoh & Tanaka [IT99]

Algorithm IT classifies each suffix i of $\boldsymbol{x}$ as being type A if $x[i] > x[i+1]$ or type B if $x[i] \leq x[i+1]$ (compare types L and S of Algorithm KA). The key observation of Itoh and Tanaka is that once all the groups of type B suffixes are sorted, the order of the type A suffixes is easy to derive. For example:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	b	a	d	d	a	d	d	a	c	c	a	\$
type $=$	A	B	B	A	B	B	A	B	B	A	A	B

To form the full SA, we begin by computing the 1-group boundaries, noting the beginning and end of each 1-group with arrays head$[1..\sigma]$ and tail$[1..\sigma]$ (recall $\sigma = |\Sigma|$). Each 1-group is further partitioned into two portions, so that in the first portion there is room for the type A suffixes, and in the second for the type B suffixes. For each group the position of the A/B partition is recorded. Observe that within a 1-group, type A suffixes should always come before type B suffixes. The SA at this stage is shown below, with "–" denoting an empty position:

	1	2	3	4	5	6	7	8	9	10	11	12
SA $=$	12	(–	2	5	8)	(–)	(–	9)	(–	–	3	6)
type $=$	B	A	B	B	B	A	A	B	A	A	A	A

Algorithm IT now sorts the B suffixes using a fast string sorting algorithm. In [IT99] multikey quicksort (MKQS) [BS97] is proposed, but any other fast sort, such as burst sort [SZ04] or the elaborate approach introduced in Algorithm MF (see below), could be used:

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \text{SA} = & 12 & (- & 8 & 5 & 2) & (-) & (- & 9) & (- & - & 6 & 3) \\ \text{type} = & B & A & B & B & B & A & A & B & A & A & A & A \end{array}$$

To sort the A-suffixes, and complete the SA, we scan SA left to right, $j = 1, 2, \ldots, n$. For each suffix position $i = \text{SA}[j]$ that we encounter in the scan, if $i-1$ is an A-suffix still awaiting sorting (that is, it has not yet been placed in the SA), then we place $i-1$ at the head of its group in SA and increment the head of the group by one. Suffix $i-1$ is now sorted and will not be moved again. Like Algorithm KA, the correctness of this procedure depends on $\text{SA}[j]$ already being in its final position when the scan of SA reaches position j. In our example, placements begin when $j = 1$, $i = \text{SA}[1] = 12$. Suffix $i-1 = 11$ is type A, so we place 11 at the front of the a group (of which it happens to be the only unsorted member), and it is now sorted:

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \text{SA} = & 12 & 11 & 8 & 5 & 2 & (-) & (- & 9) & (- & - & 6 & 3) \\ \text{type} = & B & A & B & B & B & A & A & B & A & A & A & A \end{array}$$

Next the scan reaches $j = 2$, $i = \text{SA}[2] = 11$, and so we place $i-1 = 10$ at the front of its c group at SA[7] and increment the group head, completing that group:

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \text{SA} = & 12 & 11 & 8 & 5 & 2 & (-) & 10 & 9 & (- & - & 6 & 3) \\ \text{type} = & B & A & B & B & B & A & A & B & A & A & A & A \end{array}$$

The scan continues, eventually arriving at the final SA :

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \text{SA} = & 12 & 11 & 8 & 5 & 2 & 1 & 10 & 9 & 7 & 4 & 6 & 3 \end{array}$$

Figure 6 gives an algorithm capturing these ideas. The attentive reader will note the similarity between it and Algorithm KA (Subsection 3.2). In fact, the set of B-suffixes used in Algorithm IT is a superset of the S-suffixes treated in Algorithm KA.

```
initialize SA ← SA_1
  — head[1..σ] and tail[1..σ] mark 1-group boundaries
  — part[1..σ] marks A/B partition of each 1-group
for h ← 1 to σ do
  suffixsort(SA[part[h]], SA[part[h]+1], ..., SA[tail[h]])
for j ← 1 to n do
  i ← SA[j]
  if type[i−1] = A then
    λ ← x[i−1]
    SA[head[λ]] ← i−1
    head[λ] ← head[λ]+1
```

Figure 6: Algorithm IT

Clearly IT executes in time linear in n except for the up to σ suffix sorts of the possibly $\Theta(n)$ B-suffixes in each 1-group; these sorts may require $O(n^2 \log n)$ time in pathological cases. In practice, however, IT is quite fast. It is also lightweight: with careful implementation (for example, both head and tail arrays do not need to be stored, and *suffixsort* can be executed in place), IT requires less than n bytes of additional working storage when n is large (megabytes or more) with respect to σ.

Seward [S00]

Algorithm S begins with a linear-time 2-sort of the suffixes of $\boldsymbol{x}$, thus forming SA_2 in which the boundaries of each 2-group are identified by the head array — also used to mark boundaries between the 1-groups. Therefore in this case head = head$[1..\sigma, 1..\sigma]$, allowing access to every boundary head$[\lambda, \mu]$ for every $\lambda, \mu \in \Sigma$. For our example the result of the 2-sort could be represented as follows:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	b	a	d	d	a	d	d	a	c	c	a	\$
$\mathrm{SA}_2 =$	12	(11	8	[2	5])	1	(10	9)	([4	7]	[3	6])

where () encloses non-singleton 1-groups, [ ] encloses non-singleton 2-groups.

Now consider a 1-group G_λ corresponding to a common single-letter prefix λ. Suppose that the suffixes of G_λ are fully sorted, yielding a sequence G^*_λ in ascending lexorder. Imagine now that G^*_λ is traversed in lexorder: for every suffix $i > 1$, the suffix $i-1$ can be placed in its final position in $\mathrm{SA}_{\boldsymbol{x}}$ at the head of the 2-group for $\boldsymbol{x}[i-1]\lambda$ — provided head$\big[\boldsymbol{x}[i-1], \lambda\big]$ is incremented by one after the suffix is placed there, thus allowing for correct placement of any other suffixes in the same 2-group. The lexorder of G^*_λ ensures that the suffixes $i-1$ also occur in lexorder within each 2-group.

This is essentially the strategy of Algorithm S: it uses an efficient string sort [BM93] to sort completely the unsorted suffixes in a 1-group that currently contains a minimum number of unsorted suffixes, then uses the sorted suffixes i to ***induce*** a sort of suffixes $i-1$. Thus all suffixes can be completely sorted at the cost of a complete sort of only half of them.

The process can be made still more efficient by observing that when G_λ is sorted, the suffixes with prefix λ^2 can be omitted, provided the 2-group corresponding to λ^2 is traversed *after* the traversal of G^*_λ. To see this, suppose there exists a suffix $\lambda^k \mu \boldsymbol{v}$ in G_λ, $k \geq 2, \mu \neq \lambda$. Then the suffix $\lambda\mu\boldsymbol{v}$ will have been sorted into G^*_λ and already processed to place suffix $\boldsymbol{x}[i..n] = \lambda^2\mu\boldsymbol{v}$ at head$[\lambda, \lambda]$. Thus when $\lambda^2\mu\boldsymbol{v}$ is itself processed, suffix $\boldsymbol{x}[i-1]\lambda^2\mu\boldsymbol{v}$ will be placed at head$\big[\boldsymbol{x}[i-1], \lambda\big]$ — this will again be (the now incremented) head$[\lambda, \lambda]$ if $k \geq 3$ ($\boldsymbol{x}[i-1] = \lambda$).

We can apply Algorithm S to our example string:

Iteration 1 The 1-group corresponding to $\lambda =$ \$ contains only the singleton unsorted suffix $i = 12$. Thus the sort is trivial: 12 is already in its final position in SA, and suffix $i-1 = 11$ is put in final position at head$[a, \$] = 2$.

Iteration 2 The minimum 1-group corresponding to b contains only suffix $i = 1$, which is therefore in final position. Since $i-1 = 0$, there is no further action.

Iteration 3 The minimum 1-group corresponds to $\lambda = c$; it again has only one entry to be sorted, since one of the 2-groups represented is cc. Thus suffix $i = 10$ is in final position at head$[c, a] = 7$, and determines the final position of suffix $i-1 = 9$ at head$[c, c] = 8$. Then finally for $i = 9$, the final position of suffix $i-1 = 8$ is fixed at head$[a, c] = 3$.

Iteration 4 The 1-group for $\lambda = a$ now contains only the two unsorted suffixes 2 and 5, since 11 and 8 have been put in final position by previous iterations. The sort yields SA[4] = 5, SA[5] = 2, so that the completely sorted 1-group becomes SA[2..5] = 11 8 5 2. For $i = 11$, suffix $i-1 = 10$ is already in final position; for $i = 8$, suffix $i-1 = 7$ is placed in final position at head$[d, a] = 9$; then, for $i = 5$, after head$[d, a]$ is incremented, suffix $i-1 = 4$ is placed in final position at head$[d, a] = 10$; for $i = 2$, $i-1 = 1$ is already in final position.

Iteration 5 The final group corresponds to $\lambda = d$; by now its only unsorted suffixes, 3 and 6, belong to the 2-group dd and so do not require sorting. As a result of Iteration 4, SA[9..10] = 7 4. Thus, for $i = 7$, suffix $i-1 = 6$ is placed at head$[d, d] = 11$, while for $i = 4$, the final suffix $i-1 = 3$ is placed at head$[d, d] = 12$.

For this example, only one simple sort (of suffixes 2 and 5 in Iteration 4) needs to be performed in order to compute $\mathrm{SA}_{\boldsymbol{x}}$!

Algorithm S shares the $O(n^2 \log n)$ worst case time of other induced copying algorithms, but is nevertheless very fast in practice. However, its running time sometimes seems to degrade significantly when the average lcp, $\overline{\mathrm{lcp}}$, is large, for reasons that are not quite clear. This problem is addressed by a variant, Algorithm MF, discussed next. Like IT, Algorithm S can run using less than n bytes of working storage.

Manzini & Ferragina [MF04]

Algorithm MF is a variant of Algorithm S that replaces TSQS [BM93], used to sort the 2-groups within a selected 1-group, by a more elaborate and sophisticated approach to suffix-sorting. This approach is two-tiered, depending initially on a user-specified integer lcp^*, the longest lcp of a group of suffixes that will be sorted using a standard method. (Typically, for large files, lcp^* will be chosen in the range 500..5000.) Thus, if a 2-group of suffixes is to be sorted, then MKQS [BS97] (rather than TSQS) will be employed until the recursion of MKQS reaches depth lcp^*: if the sort is not complete, this defines a set $I_m = \{i_1, i_2, ..., i_m\}$, $m \geq 2$, of suffixes such that

$$\mathrm{lcp}(i_1, i_2, \ldots, i_m) \geq \mathrm{lcp}^*.$$

At this point, the methodology used to complete the sort of these m suffixes is chosen depending on whether m is "large" or "small".

If m is small, then a sorting method called ***blind sort*** [FG99] is invoked that uses at most 36m bytes of working storage. Therefore, if blind sort is used only for $m \leq n/Q$, its space overhead will be at most $(36/Q)n$ bytes; by choosing $Q \geq 1000$, say — and thus giving special treatment to cases where "not too many" suffixes share

a "long" lcp — it can be ensured that for small m, the space used is a very small fraction of the $5n$ bytes required for $\boldsymbol{x}$ and $\mathrm{SA}_{\boldsymbol{x}}$.

Blind sort of I_m depends on the construction of a ***blind trie*** data structure [FG99]: essentially the strings

$$\boldsymbol{x}[i_j{+}\mathrm{lcp}^*..n],\ j = 1, 2, \ldots, m$$

are inserted one-by-one into an initially empty blind trie; then, as explained in [FG99], a left-to-right traversal of the trie obtains the suffixes in lexorder, as required.

If m is large ($> n/Q$), Algorithm MF reverts to the use of a slightly modified TSQS, as in Algorithm S; however, whenever at some recursive level of execution of TSQS a new set of suffixes I'_m is identified for which $m \leq n/Q$, then blind sort is again invoked to complete the sort of I'_m.

Following the initial MKQS sort to depth lcp^*, the dual strategy (blind sort/TSQS) described so far to complete the sort is actually only one of two strategies employed by Algorithm MF. Before resorting to the dual strategy, MF tries to make use of ***generalized induced copying***, as we now explain.

Suppose that for $i_1 \in I_m$ and for some least $\ell \in 1..\mathrm{lcp}^*{-}1$,

$$\boldsymbol{x}[i_1{+}\ell..i_1{+}\ell{+}1] = \lambda\mu,$$

where $[\lambda, \mu]$ identifies a 2-group that as a result of previous processing has already been fully sorted. Since the m suffixes in I_m share a common prefix, it follows that *every* suffix in I_m occurs in the same 2-group $[\lambda, \mu]$. Since moreover the m suffixes in I_m are identical up to position ℓ, it follows that the order of the suffixes in I_m is determined by their order in $[\lambda, \mu]$. Thus if such a 2-group exists, it can be used to "induce" the correct ordering of the suffixes in I_m, as follows:

(1) Bucket-sort the entries $i_j \in I_m$ in ascending order of *position* (not suffix), so membership in I_m can be determined using binary search (step (3)).

(2) Scan the 2-group $[\lambda, \mu]$ to identify a match for suffix $i_1{+}\ell$, say at some position q.

(3) Scan the suffixes (positions) listed to the left and to the right of q in 2-group $[\lambda, \mu]$; for each suffix i, use binary search to determine whether or not $i{-}\ell$ occurs in (the now-sorted) I_m. If it does occur, then mark the suffix i in $[\lambda, \mu]$.

(4) When m suffixes have been marked, scan the 2-group $[\lambda, \mu]$ from left to right: for each marked suffix i, copy $i{-}\ell$ left-to-right into I_m.

Step (2) of this procedure can be time-consuming, since it may involve a $\Theta(n)$-time match of two suffixes; in [MF04] an efficient implementation of step (2) is described that uses only a very small amount of extra space.

Of course if no such ℓ, hence no such 2-group, exists, then this method cannot be used: the dual strategy described above must be used instead.

In practice Algorithm MF runs faster than any of Algorithms KS, IT or S; in common with other induced copying algorithms, it uses less than n bytes of additional working storage but can require as much as $O(n^2 \log n)$ time in the worst case.

Schürmann & Stoye [SS05]

Algorithm SS could arguably be classified as a prefix-doubling algorithm. Certainly it is a hybrid: it first applies a prefix-doubling technique to sort individual h-groups, then uses Seward's induced copying approach to extend the sort to other groups of suffixes.

For SS, the integer h is actually a user-specified parameter, chosen to satisfy $h < \log_\sigma n$. First a radix sort is performed to compute SA_h, then the corresponding ISA_h, in which the h-rank of each h-group is formed from the tail of the h-group in SA_h (the same system used in Algorithm LS). Thus, for example, using $h = 2$, the result of the first phase of processing would be just the same as after the second iteration of LS:

	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	\$
$\mathrm{SA}_2 =$	12	11	(1	8)	4	6	(2	9)	5	7	(3	10)
$\mathrm{ISA}_2 =$	4	8	12	5	9	6	10	4	8	12	2	1

In its second phase, SS considers h-groups in SA_h that are not singletons. Let H be one such h-group. The observation is made that since every suffix i in H has the same prefix of length h, therefore the order of each i in H is determined by the rank of suffix $i+h$; that is, by $\mathrm{ISA}_h[i+h]$. A sort of all the non-singleton h-groups in SA_h thus leads to the construction of SA_{2h} and ISA_{2h}:

	1	2	3	4	5	6	7	8	9	10	11	12
$\mathrm{SA}_4 =$	12	11	(1	8)	4	6	9	2	5	7	10	3
$\mathrm{ISA}_4 =$	3	8	12	5	9	6	10	3	7	11	2	1

Observe that as a result of the prefix-doubling, the h-groups (2 9) and (3 10) have become completely sorted.

To entries in h-groups that become completely sorted by prefix-doubling, SS applies Algorithm S: if suffix i is in fixed position in SA, then the final position of suffix $i-1$ can also be determined. Thus, in our example, the sort of the h-group (2 9) that yields $2h$-order $9, 2$ induces a corresponding sorted order $8, 1$ for the $2h$-group (1 8), completing the sort.

Algorithm SS iterates this second phase – prefix-doubling followed by induced copying – until all entries in SA are singletons. Note that after the first iteration, the induced copying will as a rule refine the h-groups so that they break down into $(h+k)$-groups for various values of $k \geq 0$; thus, after the first iteration, the prefix-doubling is approximate.

Algorithm SS has worst-case time complexity $O(n^2)$ and appears to be very fast in practice, competitive with Algorithm MF. However, it is not quite lightweight, requiring somewhat more than $4n$ bytes of additional working storage.

Burkhardt & Kärkkäinen [BK03]

In a similar way to the recursive algorithms of Section 3.2, Algorithm BK computes $\mathrm{SA}_{\boldsymbol{x}}$ by first ordering a sample of the suffixes $\mathcal{S}$. The relative ranks of the suffixes in

$\mathcal{S}$ are then used to accelerate a basic string sorting algorithm, such as MKQS [BS97], applied to all the suffixes.

Central to BK is a mathematical construct called a difference cover, which defines the suffixes in $\mathcal{S}$. A difference cover D_h is a set of integers in the range $0..h-1$ such that for all $i \in 0..h-1$, there exist $j, k \in D_h$ such that $i \equiv k - j (\mathrm{mod}\ h)$. For a chosen D_h, $\mathcal{S}$ contains the suffixes of $\boldsymbol{x}$ beginning at positions i such that $i \bmod h \in D_h$.

For example $D_7 = \{1, 2, 4\}$ is a difference cover modulo 7. If we were to sample according to D_7 then for the string

$$\begin{array}{rcccccccccccccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ \boldsymbol{x} = & b & a & d & d & a & d & d & b & a & d & d & a & d & d & b & a & d & d & \$ \end{array}$$

we would obtain $\mathcal{S} = \{1, 2, 4, 8, 9, 11, 15, 16, 18, 22, 23, 25\}$. Observe for every $i \in \mathcal{S}$ that $i \mod 7$ is in D_7.

In practice, only covers D_h with $|D_h| \in \Theta(\sqrt{h})$ are suitable. However, for the chosen D_h a function $\delta(i, j)$ must also be precomputed. For any integers i, j, $\delta(i, j)$ is the smallest integer $k \in 0..h-1$ such that $(i + k) \bmod h$ and $(j + k) \bmod h$ are both in D_h. A lookup table allows constant time evaluation of $\delta(i, j)$ — we omit the details here.

Algorithm BK consists of two main phases. The goal of the first phase is to compute a data structure $\mathrm{ISA}_{\boldsymbol{x}'}$ allowing the lexicographical rank of $i \in \mathcal{S}$, relative to the other members of $\mathcal{S}$, to be computed in constant time. To this end, BK first h-sorts $\mathcal{S}$ using MKQS (or alternative) and then assigns each suffix its h-rank in the resulting h-ordering. For our example the h-ranks are:

$i \in \mathcal{S}$	1	2	4	8	9	11	15	16	18
h-rank	3	6	4	3	6	4	2	5	1

These ranks are then used to construct a new string $\boldsymbol{x}'$ (compare to $\boldsymbol{x}'$ for Algorithm KS) as follows

$$\begin{array}{rc} i \in \mathcal{S} & 1\ \ 8\ \ 15\ \ 2\ \ 9\ \ 16\ \ 4\ \ 11\ \ 18 \\ \boldsymbol{x'} = & (3\ 3\ 2)\ (6\ 6\ 5)\ (4\ 4\ 1) \end{array}$$

The structure of $\boldsymbol{x}'$ is deceptively simple. The h-ranks, r_i, appear in $|D_h|$ groups in $\boldsymbol{x}'$ (indicated above with ()) according to i modulo h. Then, within each group, ranks r_i are sorted in ascending order according to i. Because of this structure in $\boldsymbol{x'}$, its inverse suffix array, $\mathrm{ISA}_{\boldsymbol{x}'}$, can be used to obtain the rank of any $i \in \mathcal{S}$ in constant time. To compute ISA', BK makes use of Algorithm LS as an auxiliary routine (recall that LS computes both the ISA and the SA). Although LS is probably the best choice, any SACA suitable for bounded integer alphabets can be used.

With $\mathrm{ISA}_{\boldsymbol{x}'}$ computed, construction of $\mathrm{SA}_{\boldsymbol{x}}$ can begin in earnest. All suffixes are h-ordered using a string sorting algorithm, such as MKQS, to arrive at SA_h. The sorting of non-singleton h-groups which remain is then completed with a comparison based sorting algorithm using $\mathrm{ISA}_{\boldsymbol{x}'}[i + \delta(i, j)]$ and $\mathrm{ISA}_{\boldsymbol{x}'}[j + \delta(i, j)]$ as keys when comparing suffixes i and j.

In [BK03] it is shown that by choosing $h = \log_2 n$ an overall worst case running time of $O(n \log n)$ is achieved. Another attractive feature of BK is its small working space – less then $6n$ bytes – made possible by the small size of $\mathcal{S}$ relative to $\boldsymbol{x}$ and by use of inplace string sorting.

Finally, we remark that the ideas of Algorithm BK can be used to ensure any of the induced copying algorithms described in this section execute in $O(n \log n)$ worst case time.

Maniscalco [M05]

Algorithm M differs from the other algorithms in this section in that it directly computes $\mathrm{ISA}_{\boldsymbol{x}}$ and then transforms it into $\mathrm{SA}_{\boldsymbol{x}}$ inplace. At the time of writing, Algorithm M is published as C++ code on the Internet [M05], the details of which are examined in [P05].

At the heart of Algorithm M is an efficient bucket sorting regime. Most of the work is done in what is eventually $\mathrm{ISA}_{\boldsymbol{x}}$, with extra space required for a few stacks. The bucket sorting begins by linking together suffixes that are 2-equal, to form *chains* of suffixes. For example, the string

	0	1	2	3	4	5	6	7
$\boldsymbol{x}$	a	a	a	b	a	b	a	a	\$

would result in the creation of the following chains

7	6,1,0	4,2	5,3
a\$	aa	ab	ba

We define an h-chain in the same way as an h-group – that is, suffixes i and j are in the same h-chain iff they are h-equal. Thus, the chains above are all 2-chains, and the chain for a\$ is a singleton.

The space allocated for the ISA provides a way to efficiently manage chains. Instead of storing the chains explicitly as above, Algorithm M computes the equivalent array

	0	1	2	3	4	5	6	7
$\boldsymbol{x}$	a	a	a	b	a	b	a	a	\$
ISA	$\perp$	0	$\perp$	$\perp$	2	3	1	$\perp$	

in which $\mathrm{ISA}[i]$ is the largest $j < i$ such that $x[j..j+1] = x[i..i+1]$ or $\perp$ if no such j exists. In our example, the chain of all the suffixes prefixed with aa contains suffixes 6, 1 and 0 and so we have $\mathrm{ISA}[6] = 1$, $\mathrm{ISA}[1] = 0$ and $\mathrm{ISA}[0] = \perp$, marking the end of the chain. Observe that chains are singly linked, and are only traversable right-to-left. We keep track of h-chains to be processed by storing a stack of integer pairs (s, h), where s is the start of the chain (its rightmost index), and h is the length of the common prefix. Chains always appear on the stack in ascending lexicographical order, according to $x[s..s+h-1]$. Thus for our example, initially $(7, 2)$ for chain a\$ is atop the stack, and $(5, 2)$ for chain ba at the bottom.

Chains are popped from the stack and progressively refined by looking at further pairs of characters. So long as we process the chains in lexicographical order, when we pop a singleton chain, the suffix contained has been differentiated from all others and can be assigned the next lexicographic rank. Elements in the ISA which are ranks are differentiated from elements in chains by setting the sign bit, that is, if $\mathrm{ISA}[i] < 0$, then the rank for suffix i is $-\mathrm{ISA}[i]$. The evolution of the ISA of our example string subsequent sorting rounds proceed as follows.

```
formInitialChains()
repeat
  (h, ℓ) ← chainStack.pop()
  if ISA[h] = ⊥ then
    ISA[h] ← nextRank()
  else
    while h ≠ ⊥ do
      sym ← getSymbol(h + ℓ)
      updateSubChain(sym,h)
      h ← ISA[h]
    sortAndPushSubChains()
until chainstack is empty
```

Figure 7: Bucket sorting of Algorithm M

	0	1	2	3	4	5	6	7	
$\boldsymbol{x}$	a	a	a	b	a	b	a	a	\$
ISA	⊥	0	⊥	⊥	2	3	1	⊥	*Initial chains* $(7,2)_{a\$}(6,2)_{aa}(4,2)_{ab}(5,2)_{ba}$
ISA	⊥	0	⊥	⊥	1	2	1	-1	*Pop* $(7,2)_{a\$}$ *and assign rank*
ISA	⊥	⊥	⊥	⊥	1	2	⊥		*Split chain* $(6,2)_{aa}$ *into* $(6,4)_{aa\$}(0,4)_{aaab}(1,4)_{aaba}$
ISA	-3	-4	⊥	⊥	1	2	-2		*Pop* $(6,4)_{aa\$}(0,4)_{aaab}(1,4)_{aaba}$*, assign ranks*
ISA			⊥	⊥	⊥	2			*Split chain* $(4,2)_{ab}$ *into* $(4,4)_{abaa}(2,4)_{abab}$
ISA			-6	⊥	-5	2			*Pop* $(4,4)_{abaa}(2,4)_{abab}$*, assign ranks*
ISA				⊥		⊥			*Split chain* $(5,2)_{ba}$ *into* $(5,4)_{baa\$}(3,4)_{baba}$
ISA				-8		-7			*Pop* $(5,4)_{baa\$}(3,4)_{baba}$*, assign ranks*
ISA_x	3	4	6	8	5	7	2	1	*Completed Inverse Suffix Array*

When the value in a column becomes negative, the suffix has been assigned its (negated) rank and is effectively sorted. We reiterate here that when a chain is split, the resulting subchains must be placed on the stack in lexicographical order for the subsequent assignment of ranks to singletons to be correct. This is illustrated in the example above when the chain for aa is split, and the next chain processed is the singleton chain for $aa\$$. An algorithm embodying these ideas is listed in Figure 7.

Algorithm M adds two powerful heuristics to the string sorting algorithm described in Figure 7. We discuss only the first (and more important) of these heuristics here and refer the reader to [M05, P05] for details of the second.

The processing of chains in lexicographical order allows for the possibility to use previously assigned ranks as sort keys for some of the suffixes in a chain. To elucidate this idea we first need to make some observations about the way chains are processed.

When processing an h-chain, suffixes can be classified into three types: suffix i is of type X if the rank for suffix $i + h - 1$ is known, and is of type Y if the rank for suffix $i + h$ is known. If i is not of type X or type Y, then it is of type Z. Any suffix can be classified to its type in constant time by virtue of the fact we are building the ISA (we inspect $\text{ISA}[i + h - 1]$ or $\text{ISA}[i + h]$ and a checked sign bit indicates a rank). Now consider the following observation: lexicographically, type X suffixes are smaller than type Y suffixes, which in turn are smaller than type Z suffixes.

To use this observation, when we refine a chain, we place only type Z suffixes into subchains and place type X and type Y suffixes to one side. Now, the order of the m suffixes of type X suffixes can be determined via a comparison based sort, using for suffix i the rank of suffix $i+h-1$ as the sort key. Once sorted, the type X suffixes can be assigned the next m ranks by virtue of the fact that chains are being processed in lexicographical order. Type Y suffixes are treated similarly, using the rank of $j+h$ as the sort key for suffix j. Maniscalco refers to the sorting of suffixes in this way as *induction sorting* [2].

Loosely speaking, as the number of assigned ranks increases, the probability that a suffix can be sorted using the rank of another also increases. In fact, every chain of suffixes with prefix $\alpha_1\alpha_2$ such that $\alpha_2 < \alpha_1$ will be sorted entirely in this way. Clearly, induction sorting will lead to a significant reduction in work for many texts.

One could consider the induction sorting of Algorithm M an extension of the ideas in Algorithm IT. As noted above, suffixes in a 2-chain with common prefix $\alpha_1\alpha_2$ and $\alpha_1 > \alpha_2$ are sorted entirely by induction (like the type A suffixes of Algorithm IT. However the lexicographical processing of suffixes in Algorithm M means this property can be applied to suffixes at deeper levels of sorting (when $h > 2$).

The complexity of Algorithm M is likely to be $O(n^2 \log n)$ in the worst case, though on average it is usually as fast as Algorithm MF. By carefully using the space in the ISA, and converting it to the SA inplace, it also achieves a small memory footprint — rarely requiring more than n bytes of additional working space.

4 Experimental Results

To gauge the performance of the SACAs in practice we measured their runtimes and peak memory usage for a selection of files from the Canterbury corpus[3] and from the corpus compiled by Manzini[4] and Ferragina [MF04]. Details of all files tested are given in Table 2.

We implemented Algorithm IT as described in [IT99] and Algorithm KS with heuristics described in [PST05]. The implementation of Algorithm KA tested was that of [LP04]. Implementations of all other algorithms were obtained either online or by request to respective authors. For completeness we also tested a tuned suffix tree implementation [K99]. Algorithm MF was run with default parameters and Algorithm SS with parameter h=7 for genomic data and h=3 otherwise, as per testing in [SS05]. Algorithm BK used parameter h=32, as per [BK03].

All tests were conducted on a 2.8 GHz Intel Pentium 4 processor with 2Gb main memory. The operating system was RedHat Linux Fedora Core 1 (Yarrow) running kernel 2.4.23. The compiler was g++ (gcc version 3.3.2) executed with the -O3 option. Running times, shown in Table 3, are the average of four runs and do not include time spent reading input files. Times were recorded with the standard unix `time` function. Memory usage, shown in Table 4, was recorded with the `memusage` command available with most Linux distributions.

Results are summarized in Figure 8. Algorithm MF is the fastest algorithm on

[2] In fact, we can sort the type X and Y suffixes in the same sort call by using as a key for a type X suffix i the rank of $i+h-1$ and for a type Y suffix the *negated* rank of $i+h$.

[3] `http://www.cosc.canterbury.ac.nz/corpus/`

[4] `http://www.mfn.unipmn.it/~manzini/lightweight/corpus/`

Table 2: Description of the data set used for testing. LCP refers to the Longest Common Prefix amongst all suffixes in the string.

String	Mean LCP	Max LCP	Size (bytes)	σ	Description
E.coli	17	2,815	4,638,690	4	*Escherichia coli* genome
chr22.dna	1,979	199,999	34,553,758	4	Human chromosome 22
bible	14	551	4,047,392	63	King James bible
world192	23	559	2,473,400	94	CIA world fact book
sprot34	89	7,373	109,617,186	66	SwissProt database
rfc	93	3,445	116,421,901	120	Concatenated IETF RFC files
howto	267	70,720	39,422,105	197	Linux Howto files
reuters	282	26,597	114,711,151	93	Reuters news in XML format
jdk13c	679	37,334	69,728,899	113	JDK 1.3 documentation
etext99	1,108	286,352	105,277,340	146	Texts from Gutenberg project

average, narrowly shading algorithms M and SS. These three algorithms (MF,M,SS) outperform the next fastest algorithm, LS, by roughly a factor of 2. Note that for file jdk13c it is the suffix tree which is fastest — leaving room for at least some improvement in the SACAs.

When testing algorithm M, we observed that the final step of transforming the ISA into the SA constituted 20-30% of the overall runtime. For some applications though (most notably the BWT [BW94]), this transformation is not required, making M significantly faster than MF – see experiments in [P05].

The speed of MF and M is particularly impressive given their small working memory $5.01n$ and $5.49n$ bytes on average respectively. The lightweight nature, of these algorithms separates them from SS which requires slightly more than $9n$ bytes on average. We also remark that while Algorithm BK is not amongst the fastest algorithms tested the ideas in it are important because they could be used to guarantee acceptable worst case behavior of algorithms MF and M, without adversely affecting the speed or space usage of those algorithms.

Times in Table 3 for Algorithm SS versus Algorithm MF seem to run contrary to results published in [SS05], however our experiment is different. In [SS05], files were bounded to at most 50,000,000 characters, making many test files shorter than their original form. We suspect the full length files are harder for Algorithm SS to sort.

The large variation in performance of Algorithm KS can be attributed to the occasional ineffectiveness of heuristics described in [PST05]. Of interest also is the general poor performance of the recursive algorithms KS, KA and KJP. These algorithms have superior asymptotic behaviour, but for many files run several times slower than the other algorithms and often consume more memory than the suffix tree (KJP in particular). Memory profiling reveals that the recursive algorithms suffer form very poor cache behaviour, which largely nullifies their asymptotic advantage. These results leave open the question: is there a practically fast $\Theta(n)$ time suffix array construction algorithm which is also lightweight?

Table 3: CPU time (seconds) on test data. Minimum is shown in bold for each string.

	E.coli	chr22	bible	world	sprot	rfc	howto	reuters	jdk13c	etext
M	**2**	20	2	**1**	90	89	25	**99**	60	**75**
SS	**2**	25	2	**1**	99	93	22	133	64	92
MF	**2**	**16**	2	**1**	**74**	**65**	**18**	147	82	76
IT	**2**	416	**1**	**1**	125	108	38	278	286	331
S	3	29	2	**1**	126	110	37	258	217	290
BK	4	40	3	2	200	171	43	280	152	141
LS	4	35	3	2	144	154	40	183	105	146
KA	6	47	5	3	183	179	63	185	98	202
KS	5	57	4	2	306	288	55	377	204	219
KJP	4	31	4	3	183	189	61	192	102	179
Tree	6	51	5	3	183	193	80	141	**52**	226

Table 4: Peak Memory Usage (Mbs)

	E.coli	chr22	bible	world	sprot	rfc	howto	reuters	jdk13c	etext
M	32	205	29	13	547	599	197	572	357	542
SS	40	297	36	24	942	1,006	368	988	604	915
MF	22	165	19	12	524	557	188	548	333	503
IT	22	165	19	12	523	555	188	547	332	502
S	22	165	19	12	523	555	188	547	332	502
BK	26	194	23	14	614	652	221	643	391	590
LS	35	264	31	19	836	888	301	875	532	803
KA	58	429	50	31	1,359	1,443	526	1,422	864	1,406
KS	43	334	37	23	1,279	1,230	389	1,434	870	1,071
KJP	58	427	58	36	1,574	1,673	571	1,645	1,000	1,509
Tree	74	541	54	32	1,421	1,554	526	1,444	931	1,405

References

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz & Enno Ohlebusch, **Replacing suffix trees with enhanced suffix arrays**, *J. Discrete Algs. 2* (2004) 53–86.

[BK03] Stefan Burkhardt & Juha Kärkkäinen, **Fast lightweight suffix array construction and checking**, *Proc. 14th Ann. Symp. Combinatorial Pattern Matching*, LNCS 2676, Springer-Verlag (2003) 55-69.

[BM93] Jon L. Bentley & M. Douglas McIlroy, **Engineering a sort function**, *Software — Practice & Experience 23–11* (1993) 1249–1265.

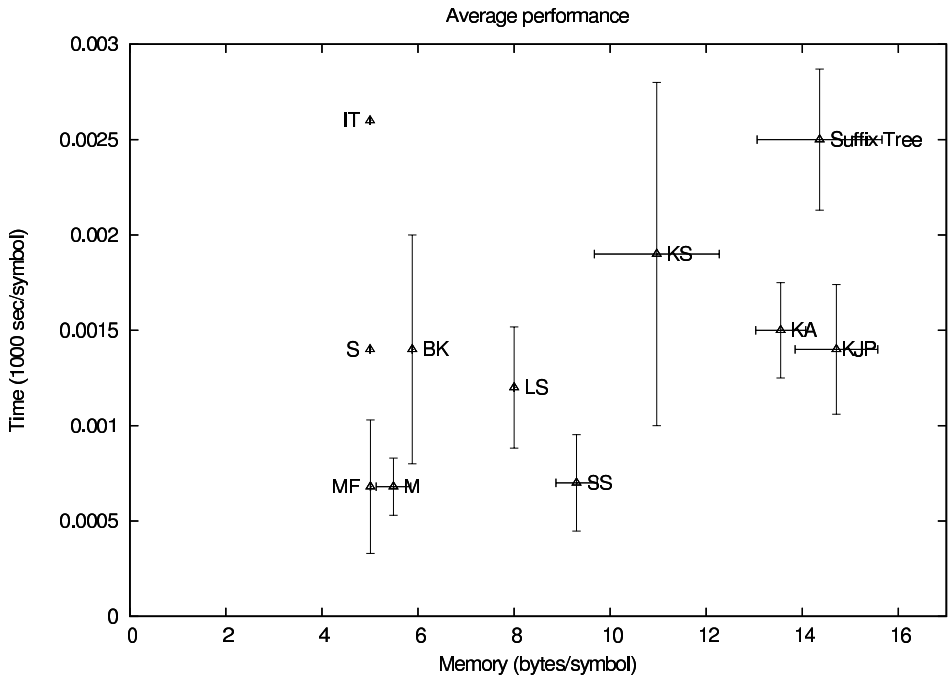

Figure 8: Resource requirements of the algorithms averaged over the test corpus. Error bars are one standard deviation. Abscissa error bars for algorithms MF, S, IT, BK and LS are insignificantly small. Ordinate error bars for algorithms S and IT are not shown to improve presentation (sd 0.009 and 0.0036 respectively).

[BS97] Jon L. Bentley & Robert Sedgewick, **Fast algorithms for sorting and searching strings**, Proc. ACM-SIAM Symp. Discrete Algs. (1997) 360–369.

[BW94] Michael Burrows & David J. Wheeler, *A Block-Sorting Lossless Data Compression Algorithm*, Research Report 124, Digital Equipment Corporation (1994) 18 pp.

[CF02] Andreas Crauser & Paolo Ferragina, **A theoretical and experimental study on the construction of suffix arrays in external memory**, *Algorithmica 32–1* (2002) 1–35.

[F97] Martin Farach, **Optimal suffix tree construction with large alphabets**, *Proc. 38th IEEE Symp. Found. Comp. Sci.* (1997) 137–143.

[FG99] Paolo Ferragina & Roberto Grossi, **The string B-tree: a new data structure for string search in external memory and its applications**, *J. Assoc. Comput. Mach. 46–2* (1999) 236–280.

[GGV04] Roberto Grossi, Ankur Gupta & Jeffrey Scott Vitter, **When indexing equals compression: experiments with compressing suffix arrays and applications**, *Proc. 15th ACM-SIAM Symp. Discrete Algs.* (2004) 636–645.

[H04] Michael Hart, *Project Gutenberg*: `http://www.gutenberg.net`

[HSS03] Wing-Kai Hon, Kunihiko Sadakane & Wing-Kin Sung, **Breaking a time-and-space barrier in constructing full-text indices**, *Proc. 44th IEEE Symp. Found. Comp. Sci.* (2003) 251–260.

[IT99] Hideo Itoh & Hozumi Tanaka, **An efficient method for in memory construction of suffix arrays**, *Proc. IEEE Symp. String Process. & Inform. Retrieval* (1999) 81–88.

[K99] Stefan Kurtz, Reducing the space requirement of suffix trees, *Software — Practice & Experience 29–13* (1999) 1149–1171.

[KA03] Pang Ko & Srinivas Aluru, **Space Efficient Linear Time Construction of Suffix Arrays**, *Proc. 14th Ann. Symp. Combinatorial Pattern Matching*, LNCS 2676, Springer-Verlag (2003) 200–210.

[KJP04] Dong Kyue Kim, Junha Jo & Heejin Park, **A fast algorithm for constructing suffix arrays for fixed-size alphabets**, *Proc. Workshop on Experimental Algorithms*, LNCS 3059, Springer-Verlag (2004) 301–314.

[KLAAP01] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa & Kunsoo Park, **Linear-time longest-common-prefix computation in suffix arrays and its applications**, *Proc. 12th Ann. Symp. Combinatorial Pattern Matching*, LNCS 2089, Springer-Verlag (2001) 181-192.

[KMR72] Richard M. Karp, Raymond E. Miller & Arnold L. Rosenberg, **Rapid identification of repeated patterns in strings, trees and arrays**, *Fourth Annual ACM Symp. Theory of Comput.* (1972) 125–136.

[KS03] Juha Kärkkäinen & Peter Sanders, **Simple Linear Work Suffix Array Construction**, *Proc. 30th Internat. Colloq. Automata, Languages & Programming*, LNCS 2719, Springer-Verlag (2003) 943–955.

[KSPP03] Dong Kyue Kim, Jeong Seop Sim, Heejin Park & Kunsoo Park, **Linear-time Construction of Suffix Arrays**, *Proc. 14th Ann. Symp. Combinatorial Pattern Matching*, LNCS 2676, Springer-Verlag (2003) 186–199.

[LP04] Sunglim Lee & Kunsoo Park, Efficient implementations of suffix array construction algorithms, *Proc. 15th Australasian Workshop on Combinatorial Algs.* , Seok-Hee Hong (ed.) (2004) 64–72.

[LS99] N. Jesper Larsson & Kunihiko Sadakane, *Faster Suffix Sorting*, Technical Report LU-CS-TR:99–214, Lund University (1999) 20 pp.

[M97] M. Douglas McIlroy, *ssort.c*:
`http://cm.bell-labs.com/cm/cs/who/doug/source.html`

[M99] J. Ian Munro, **Succinct data structures**, *Proc. Workshop on Data Structures* (1999) 3–7.

[M04] Giovanni Manzini, **Two space saving tricks for linear time LCP array computation**, *Proc. Scandinavian Workshop on Algorithm Theory* (2004) 372–383.

[M05] Michael Maniscalco, *MSufSort*: `http://www.michael-maniscalco.com/`

[MBM93] Peter M. McIlroy, Keith Bostic & M. Douglas McIlroy, **Engineering radix sort**, *Computing Systems 6–1* (1993) 5–27.

[MF04] Giovanni Manzini & Paolo Ferragina, **Engineering a lightweight suffix array construction algorithm**, *Algorithmica 40* (2004) 33–50.

[MM90] Udi Manber & Gene W. Myers, **Suffix Arrays: A new method for on-line string searches**, *Proc. First ACM-SIAM Symp. Discrete Algs.* (1990) 319–327.

[MM93] Udi Manber & Gene W. Myers, **Suffix Arrays: A new method for on-line string searches**, *SIAM J. Comput. 22* (1993) 935–948.

[N05] Jeong Chae Na, **Linear-time construction of compressed suffix arrays using $O(n \log n)$-bit working space for large alphabets**, *Proc. 16th Ann. Symp. Combinatorial Pattern Matching*, LNCS 3537, Springer-Verlag (2005) 57–67.

[P05] Simon J. Puglisi, *Exposition and analysis of a suffix sorting algorithm*, Technical Report CAS-05-02-WS, Department of Computing and Software, McMaster University (2005) 19 pp.

[PST05] Simon J. Puglisi, Bill F. Smyth & Andrew Turpin, **The performance of linear time suffix sorting algorithms**, *Proc. Data Compression Conf.* (2005) 358–367.

[S98] Kunihiko Sadakane, **A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation**, *Proc. Data Compression Conf.* (1998) 129–138.

[S00] Julian Seward, **On the performance of BWT sorting algorithms**, *Proc. Data Compression Conf.* (2000) 173–182.

[S03] Bill F. Smyth, *Computing Patterns in Strings*, Pearson Addison-Wesley (2003) 423 pp.

[SKPP03] Jeong Seop Sim, Dong Kyue Kim, Heejin Park & Kunsoo Park, **Linear-time search in suffix arrays**, *Proc. 14th Australasian Workshop on Combinatorial Algs.* (2003) 139–146.

[SZ04] Ranjan Sinha & Justin Zobel, **Cache-conscious sorting of large sets of strings with dynamic tries** *ACM J. Experimental Algs. 9* (2004) 1–31.

[SS05] Klaus-Bernd Schürmann & Jens Stoye, **An incomplex algorithm for fast suffix array construction**, *Proc. 7th Workshop on Algorithm Engineering & Experiments*