A Taxonomy of Suffix Array Construction Algorithms

SIMON J. PUGLISI
Curtin University of Technology
W. F. SMYTH
McMaster University and Curtin University of Technology
and
ANDREW H. TURPIN
RMIT University

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. New hybrid algorithms are also described. We 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.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms**]: Nonnumerical Algorithms and Problems; E.1 [**Data Structures**]: Arrays; H.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Suffix array, Suffix tree, Suffix sorting, Burrows-Wheeler transform

1. INTRODUCTION

Suffix arrays were introduced in 1990 [Manber and Myers 1990; 1993], 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

Authors' addresses: Simon J. Puglisi, Department of Computing, Curtin University, GPO Box U1987, Perth, WA 6845, Australia; email: `sjp@cs.curtin.edu.au`. W. F. Smyth, Algorithms Research Group, Department of Computing & Software, McMaster University, Hamilton, ON L8S 4K1, Canada; email: `smyth@mcmaster.ca`; url: `www.cas.mcmaster.ca/cas/research/groups.shtml`. Andrew Turpin, School of Computer Science & Information Technology, RMIT University, GPO Box 2476V, Melbourne V 3001, Australia; email: `aht@cs.rmit.edu.au`.
Corresponding author: Simon J. Puglisi `sjp@cs.curtin.edu.au`

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 [Ko and Aluru 2003; Kärkkäinen and Sanders 2003];

—SACAs exist that are even faster in practice, though with supralinear worst-case construction time requirements [Larsson and Sadakane 1999; Burkhardt and Kärkkäinen 2003; Manzini and Ferragina 2004; Maniscalco 2005];

—any problem whose solution can be computed using suffix trees is solvable with the same asymptotic complexity using suffix arrays [Abouelhoda et al. 2004].

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, [Crauser and Ferragina 2002]) 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 and techniques such as those discussed in [Grossi and Vitter 2005; Mäkinen and Navarro 2006] and references therein 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, including new hybrid 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 [Smyth 2003]; 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, an assumption almost universal for strings processed in a computer and thus a requirement for most efficient string processing algorithms. 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).

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 $\mathrm{SA}[1..n]$ in which $\mathrm{SA}[j] = i$ 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 $\Sigma = \{\$, a, b, c, d, e\}$:

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

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*** $\mathrm{lcp} = \mathrm{lcp}[1..n]$: for every $j \in 2..n$, $\mathrm{lcp}[j]$ is just the length of the ***longest common prefix*** of suffixes $\mathrm{SA}[j–1]$ and $\mathrm{SA}[j]$. In our example:

$$\begin{array}{rcccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ \boldsymbol{x} = & a & b & e & a & c & a & d & a & b & e & a & \$ \\ \mathrm{SA} = & 12 & 11 & 8 & 1 & 4 & 6 & 9 & 2 & 5 & 7 & 10 & 3 \\ \mathrm{lcp} = & - & 0 & 1 & 4 & 1 & 1 & 0 & 3 & 0 & 0 & 0 & 2 \end{array}$$

Thus the longest common prefix of suffixes 11 and 8 is a of length 1, while that of suffixes 8 and 1 is $abca$ of length 4. Since lcp can be computed in linear time from $\mathrm{SA}_{\boldsymbol{x}}$ [Kasai et al. 2001; Manzini 2004], 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 $\mathrm{ISA}[1..n]$ in which

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

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

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

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 ← 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
```

Fig. 1. Algorithm for computing ISA from SA in place

$$\begin{array}{l}\textbf{for } j \leftarrow 1 \textbf{ to } n \textbf{ do}\\ \quad \mathrm{SA}\big[\mathrm{ISA}[j]\big] \leftarrow j\end{array}$$

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.

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. Since there are at most $\log_2 n$ iterations, the time required is therefore $O(n \log n)$. There are two algorithms in this class: MM [Manber and Myers 1990; 1993] and LS [Sadakane 1998; Larsson and Sadakane 1999].

(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 [Ko and Aluru 2003], KS [Kärkkäinen and Sanders 2003] and KJP [Kim et al. 2004].

(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 string sorting technique (for example, [Bentley and McIlroy 1993; McIlroy et al. 1993; McIlroy 1997; Bentley and Sedgewick 1997; Sinha and Zobel 2004]) is invoked for the selected subset of suffixes. The general idea seems to have been first proposed in [Burrows and Wheeler 1994], but it has been implemented in quite different ways [Itoh and Tanaka 1999; Seward 2000; Manzini and Ferragina 2004; Schürmann and Stoye 2005; Burkhardt and Kärkkäinen 2003; Maniscalco 2005]. 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 [Hart 1997]);

—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 fifteen main species of SACA that have been recognized so far. Table I specifies a worst-case asymptotic time complexity

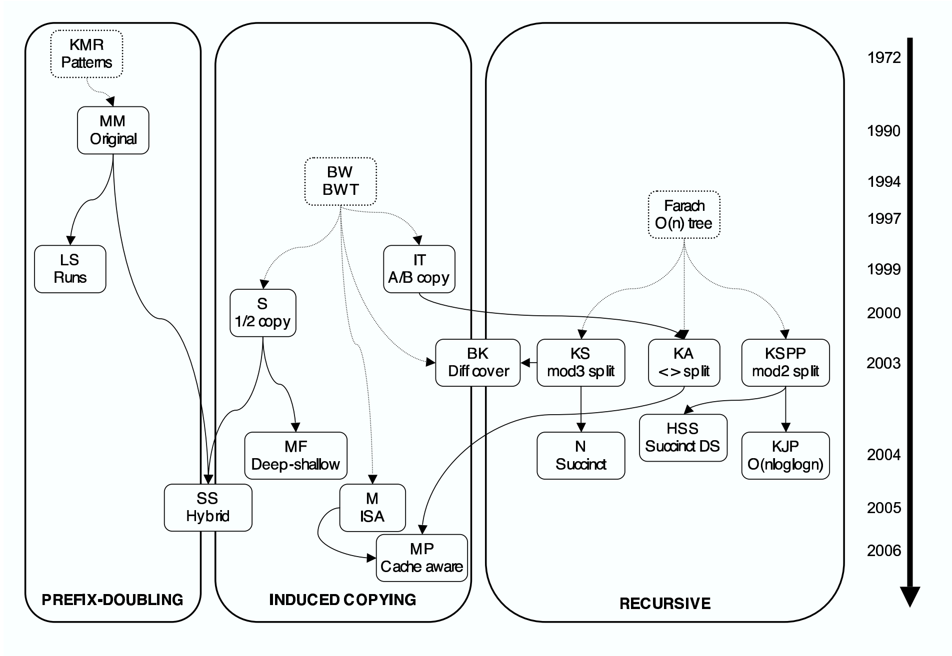

Fig. 2. Taxonomy of suffix array construction algorithms

for each of these algorithms, together with a few others that have been proposed in the literature; the table also gives average time and space requirements over a range of cases tested in our experiments (Section 4). The worst-case complexities given are those claimed by the designers of the algorithms, though it is possible for some of those that include the term $n^2 \log n$ that a somewhat lower worst-case bound holds. This could be a subject of future research. It should be remarked that in practice the behaviour of all the algorithms is linear; in other words, over strings of homogeneous data (that is, all English-language or all DNA), the execution time of each algorithm increases linearly in string length.

Table I. Performance summary of the construction algorithms. Time is relative to MP, the fastest in our experiments. Memory is given in bytes including space required for the suffix array and input string and is the average space required in our experiments. Algorithms HSS and N are included, even though to our knowledge they have not been implemented. The time for algorithm MM is estimated from experiments in [Larsson and Sadakane 1999].

Algorithm		Worst Case	Time	Memory
Prefix-Doubling				
	MM [Manber and Myers 1993]	$O(n \log n)$	30	$8n$
	LS [Larsson and Sadakane 1999]	$O(n \log n)$	3	$8n$
Recursive				
	KA [Ko and Aluru 2003]	$O(n)$	2.5	$7 - 10n$
	KS [Kärkkäinen and Sanders 2003]	$O(n)$	4.7	10-13n
	KSPP [Kim et al. 2003]	$O(n)$	–	–
	HSS [Hon et al. 2003]	$O(n)$	–	–
	KJP [Kim et al. 2004]	$O(n \log \log n)$	3.5	13-16n
	N [Na 2005]	$O(n)$	–	–
Induced Copying				
	IT [Itoh and Tanaka 1999]	$O(n^2 \log n)$	6.5	$5n$
	S [Seward 2000]	$O(n^2 \log n)$	3.5	$5n$
	BK [Burkhardt and Kärkkäinen 2003]	$O(n \log n)$	3.5	5-6n
	MF [Manzini and Ferragina 2004]	$O(n^2 \log n)$	1.7	$5n$
	SS [Schürmann and Stoye 2005]	$O(n^2)$	1.8	9-10n
	BB [Baron and Bresler 2005]	$O(n\sqrt{\log n})$	2.1	$18n$
	M [Maniscalco and Puglisi 2006a]	$O(n^2 \log n)$	1.3	5-6n
	MP [Maniscalco and Puglisi 2006b]	$O(n^2 \log n)$	1	5-6n
Hybrid				
	IT+KA	$O(n^2 \log n)$	4.8	$5n$
	BK+IT+KA	$O(n \log n)$	2.3	5-6n
	BK+S	$O(n \log n)$	2.8	5-6n
Suffix Tree				
	K [Kurtz 1999]	$O(n \log \sigma)$	6.3	13-15n

3. THE ALGORITHMS

3.1 Prefix-Doubling Algorithms [Karp et al. 1972]

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 [Karp et al. 1972] 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-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

```
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
```

Fig. 3. Algorithm MM

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 and McIlroy 1993].

MM [Manber and Myers 1990; 1993].

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

	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

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:

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

The final SA = SA_8 and ISA = 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. In fact, if the contents of $\boldsymbol{x}$ are not required after SA is computed, n bytes can be saved by initially storing one character as an integer in ISA and then overwriting each character with its corresponding 1-group value, available after a counting sort. If this is acceptable, then MM requires $8n$ bytes total. The algorithm can be represented conceptually as shown in Figure 3. A time and space-efficient implementation of MM is available at [McIlroy 1997].

LS [Sadakane 1998; Larsson and Sadakane 1999].

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$[i]$.

$$\begin{array}{rcccccccccccc}
 & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\
\boldsymbol{x} = & a & b & e & a & c & a & d & a & b & e & a & \$ \\
\mathrm{SA}_1 = & 12 & (1 & 4 & 6 & 8 & 11) & (2 & 9) & 5 & 7 & (3 & 10) \\
\mathrm{ISA}_1 = & 6 & 8 & 12 & 6 & 9 & 6 & 10 & 6 & 8 & 12 & 6 & 1
\end{array}$$

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:

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

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 $2h$:

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

A further doubling yields

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

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.

In fact, as observed in [Larsson and Sadakane 1999], L can be eliminated entirely. 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), and like MM can save n bytes during processing by overwriting the input string. As shown in [Larsson and Sadakane 1999], LS executes in $O(n \log n)$ time, again the same as MM; however, in practice its running time is usually much (10 times or so) faster.

3.2 Recursive Algorithms [Farach 1997]

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 in [Farach 1997] 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:

```
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)
```

Fig. 4. General algorithm for recursive SA construction. The convention procedure_name(input; output) is used (input and output parameters are separated by a semicolon).

(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 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 ranking the positions i' of $\boldsymbol{x'}$, a string of length $n' \leq \lfloor fn \rfloor$. This ranking is based on computing the ranks of the corresponding suffixes

$$\boldsymbol{x}\big[\boldsymbol{x'}[i']..n\big] \tag{1}$$

of $\boldsymbol{x}$, taking into account only those suffixes that have been assigned to $\boldsymbol{x'}$. We call this string of ranks $\mathrm{ISA}_{\boldsymbol{x'}} = \mathrm{ISA}_{\boldsymbol{x'}}[1..n']$.

Since computation of $\mathrm{ISA}_{\boldsymbol{x'}}$ may require more than $\Theta(n')$ time, we therefore invoke a procedure *semisort* that in $\Theta(n')$ time computes an *approximation* $\mathrm{ISA}' = \mathrm{ISA}'[1..n']$ of $\mathrm{ISA}_{\boldsymbol{x'}}$ — that is, a partial ranking (h-ranking) of the suffixes (1) that breaks ties among them only up to some common prefix of a predetermined constant length h. At some level of recursion, the approximation ISA' will be exact (its entries will be distinct) — 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.

KA [Ko and Aluru 2003; 2005].

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, $|\boldsymbol{x'}| \le |\boldsymbol{x}|/2$.

For example,

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

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

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

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].

First SA_1 is computed, yielding 1-groups for which the leftmost and rightmost positions are specified in arrays head$[1..\alpha]$ and 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

[1] In [Ko and Aluru 2003] 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.

```
initialize SA ← SA_1, head[1..σ], tail[1..σ]
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
```

Fig. 5. Algorithm KA-merge

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 yield a corresponding approach for the case $|\mathcal{L}| < |\mathcal{S}|$.)

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

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

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 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, $\mathrm{SA}[j]$ is already in its final position. In our example, placements begin when $j = 1$, so that $i = \mathrm{SA}[1] = 12$. 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):

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

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

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

The scan continues until finally

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

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.

KS [Kärkkäinen and Sanders 2003; Kärkkäinen et al. 2006].

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

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

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

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:

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

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

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

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 $i_{02} \equiv 2 \bmod 3$, then $i_{02}{+}1 \equiv 0 \bmod 3$, 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 $i_{02} \equiv 0 \bmod 3$, then $i_{02}{+}2 \equiv 2 \bmod 3$, 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 [Na 2005] permits construction of a succinct suffix array in $O(n)$ time using only $O(n \log\sigma \log^q n)$ bits of working memory, where $q = \log_2 3$.

KJP [Kim et al. 2003; Hon et al. 2003; Kim et al. 2004; Kim et al. 2005]. The KJP *split* procedure adopts the same approach as Farach's suffix tree construction algorithm [Farach 1997]: 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:

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

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:

$$\begin{array}{ccccccc} i & \scriptstyle 11 & \scriptstyle 7 & \scriptstyle 1 & \scriptstyle 9 & \scriptstyle 3 & \scriptstyle 5 \\ p_i & \$- & ac & ba & ca & dd & dd \\ r_i & 1 & 2 & 3 & 4 & 5 & 5 \end{array}$$

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

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

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 $\mathrm{SA}_{\boldsymbol{y}}[i]$:

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

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 [Kim et al. 2004] it is explained how

C can be computed in $\log|\boldsymbol{x'}|$ time using a suffix array search (pattern-matching) algorithm described in [Sim et al. 2003]. We omit the details; however, for our example we would find

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

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 [Kim et al. 2003] 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 [Hon et al. 2003] uses "succinct data structures" [Munro 1999] 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 [Na 2005] 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 [Burrows and Wheeler 1994]

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), even though 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 usually 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 discussed in this section:

—Itoh & Tanaka [Itoh and Tanaka 1999] 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 [Seward 2000] 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 [Manzini and Ferragina 2004] and SS [Schürmann and Stoye 2005].

—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.

—The algorithm of Maniscalco [Maniscalco and Puglisi 2006a] 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.

—Finally, the new algorithm of Maniscalco and Puglisi [Maniscalco and Puglisi 2006b] extends the IT/KA splitting idea to select a small sample of suffixes which are sorted in a cache-friendly way and then used to obtain the full SA.

Before describing the algorithms in more detail, we mention one other approach [Baron and Bresler 2005] that is also related to the Burrows-Wheeler transform. The authors describe a method in which suffixes are inserted in descending order of position in $\boldsymbol{x}$ into a suffix list maintained in lexorder. Three implementations are described, of which the fastest has time complexity $O(n\sqrt{\log n})$ and, according to our experiments, a speed in practice somewhat faster than that of Algorithm LS (Subsection 3.1). However, due to the need to use pointers, the space requirement of the fastest version is several times that of the lightweight algorithms (averaging $17.6n$ in our experiments), and is thus not suited to long strings.

IT [Itoh and Tanaka 1999].
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:

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

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:

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

Algorithm IT now sorts the B suffixes using a fast string sorting algorithm. In [Itoh and Tanaka 1999] multikey quicksort (MKQS) [Bentley and Sedgewick 1997] is proposed, but any other fast sort, such as burst sort [Sinha and Zobel 2004] or the elaborate approach introduced in Algorithm MF (see below), could be used:

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

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 $i-1$

```
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
```

Fig. 6. Algorithm IT

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 SA[j] already being in its final position when the scan of SA reaches position j. In our example, placements begin when $j = 1$, $i = \mathrm{SA}[1] = 12$. 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:

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

Next the scan reaches $j = 2$, $i = \mathrm{SA}[2] = 11$, 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:

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

The scan continues, eventually arriving at the final SA :

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

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.

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.

S [Seward 2000].

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:

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \boldsymbol{x} = & b & a & d & d & a & d & d & a & c & c & a & \$ \\ \mathrm{SA}_2 = & \multicolumn{12}{l}{12\ \big(11\ 8\ [2\ 5]\big)\ 1\ (10\ 9)\ \big([4\ 7]\ [3\ 6]\big)} \end{array}$$

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 [Bentley and McIlroy 1993] 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 that the 2-group λ^2 is the *last* 2-group of G_λ to be traversed. 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] = 11 852. 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] = 74. 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 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, $\overline{\text{lcp}}$, 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.

MF [Manzini and Ferragina 2004].

Algorithm MF is a variant of Algorithm S that replaces TSQS [Bentley and McIlroy 1993], 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 [Bentley and Sedgewick 1997] (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

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

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*** [Ferragina and Grossi 1999] 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 SA$\boldsymbol{x}$.

Blind sort of I_m depends on the construction of a ***blind trie*** data structure [Ferragina and Grossi 1999]: essentially the strings

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

are inserted one-by-one into an initially empty blind trie; then, as explained in [Ferragina and Grossi 1999], 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 $\ell \in 1..\text{lcp}^*-1$,

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

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 [Manzini and Ferragina 2004] 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.

SS [Schürmann and Stoye 2005].

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:

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

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}:

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

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.

BK [Burkhardt and Kärkkäinen 2003; Kärkkäinen et al. 2006].

In a way similar 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 [Bentley and Sedgewick 1997], applied to all the suffixes. Of the recursive algorithms, Algorithm KS is particularly related to Algorithm BK — a relationship elucidated in a recent paper [Kärkkäinen et al. 2006].

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

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

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 $(i + k)$ mod h and $(j + k)$ mod 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

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

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 $\mathrm{ISA}_{\boldsymbol{x}'}$, 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 the 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. We note that the tactic of storing ranks of some suffixes to later limit the sort depth of others is also adopted in the algorithm of [Khmelev 2003], albeit in a quite different way, and with worse performance (both asymptotically and in practice).

In [Burkhardt and Kärkkäinen 2003] 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 the 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.

M [Maniscalco 2005; Maniscalco and Puglisi 2006a].

Algorithm M differs from the other algorithms in this section in that it directly computes $\mathrm{ISA}_{\boldsymbol{x}}$ and then transforms it into $\mathrm{SA}_{\boldsymbol{x}}$ in place.

At the heart of Algorithm M is an efficient bucket sorting routine. Most of the work is done in what is eventually $\mathrm{ISA}_{\boldsymbol{x}}$, with extra space required for a few stacks.

The bucket sort begins by linking together suffixes that are 2-equal, to form *chains* of suffixes. For example, the string

	1	2	3	4	5	6	7	8	
$\boldsymbol{x}$	a	a	a	b	a	b	a	a	\$

would result in the creation of the following chains

8	7,2,1	5,3	6,4
$a\$$	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

	1	2	3	4	5	6	7	8	
$\boldsymbol{x}$	a	a	a	b	a	b	a	a	\$
ISA	$\perp$	1	$\perp$	$\perp$	3	4	2	$\perp$	

in which 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 7, 2 and 1 and so we have ISA$[7] = 2$, ISA$[2] = 1$ and ISA$[1] = \perp$, 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 $(8, 2)$ for chain $a\$$ is atop the stack, and $(6, 2)$ 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 ISA$[i] < 0$, then the rank for suffix i is $-$ISA$[i]$. The evolution of the ISA for our example string proceeds as follows in subsequent sorting rounds:

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

```
formInitialChains()
repeat
  (s, h) ← chainStack.pop()
  if ISA[s] = ⊥ then
    ISA[s] ← nextRank()
  else
    while s ≠ ⊥ do
      sym ← getSymbol(s + h)
      updateSubChain(sym,s)
      s ← ISA[s]
    sortAndPushSubChains(h+1)
until chainstack is empty
```

Fig. 7. Bucket sorting of Algorithm M

When the value in a column becomes negative, the suffix has been assigned its (negated) rank and is effectively sorted. We reiterate 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 shown 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 [Maniscalco and Puglisi 2006a] for details of the second.

The processing of chains in lexicographical order allows for the possibility of using 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 by type in constant time by virtue of the fact that we are building the ISA (we inspect ISA$[i + h - 1]$ or 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 according to their $h + 1$st symbol and place type X and type Y suffixes to one side. Now, the order of the m suffixes of type X 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. In [Maniscalco and Puglisi 2006a] this technique is referred to as *induction sorting* [2].

[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

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 to be 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 $h > 2$).

In practice Algorithm M is very fast. By carefully using the space in ISA, and converting it to SA in place, it also achieves a small memory footprint — rarely requiring more than n bytes of additional working space.

MP [Maniscalco and Puglisi 2006b].

Algorithm MP takes the sampling method of KA and IT a step further, introducing a way of splitting suffixes that is more separative and that thus allows the order of more suffixes to be induced cheaply. Recall the way algorithm KA divides suffixes into sets $\mathcal{S}$ and $\mathcal{L}$, by classifying each suffix as either type S or type L, as shown in the example string below:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\boldsymbol{x}$	e	d	a	b	d	c	c	d	e	e	d	a	b	\$
type	L	L	S	S	L	S	S	S	L	L	L	S	L	$-$

Algorithm KA sorts the set containing fewer suffixes, which in this example would be $\mathcal{S}$. In [Maniscalco and Puglisi 2006b] it is shown that only a subset of $\mathcal{S}$ need in fact be sorted (a fact contemporaneously discovered by Mori [Mori 2006]). This subset, denoted $\mathcal{T}$, is defined as follows:

$$\mathcal{T} = \{i \colon i \in \mathcal{S} \text{ and } i{+}1 \in \mathcal{L}\}.$$

That is, only the rightmost suffix in every run of S suffixes is included in $\mathcal{T}$ — thus for the example string $\mathcal{T} = \{4, 8, 12\}$, of size two less than $\mathcal{S}$. From the order of the suffixes in $\mathcal{T}$, the order of those in $\mathcal{S}$ can be inferred; from this point on, the algorithm is identical to algorithm KA.

We will shortly explain how algorithm MP sorts the suffixes in $\mathcal{T}$, but for now assume they have been sorted, and have been placed in their final positions in SA, as shown below:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA	$(-)$	12	$(-)$	$(-)$	4	$(-)$	$(-)$	$(-$	$-)$	$(-)$	8	$(-$	$-)$	$(-)$
group	\$	ab		b\$	bd	cc	cd	da		dc	de	ed		ee

The parentheses (), as usual, indicate *group* boundaries, in this case 2-group boundaries, which are easily computed with a counting sort. To sort the type $\mathcal{S}$ suffixes, we scan SA in its current state from right to left. For each suffix (that is, non-empty location) SA[i] that we encounter in the scan, if SA[i] $- 1$ is of type $\mathcal{S}$ we place

X suffix i the rank of $i + h - 1$ and for a type Y suffix the *negated* rank of $i + h$.

$\mathrm{SA}[i]-1$ at the current end of the group for $x[\mathrm{SA}[i]-1]x[\mathrm{SA}[i]]$ and decrement the end of that group.

When the scan is complete, all the members of $\mathcal{S}$ are in their final place in SA, and a method identical to that described for algorithm KA is used to complete the construction of SA.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Description
SA	(−)	12	(−)	(−)	4	(−)	(−)	(−	−)	(−)	8	(−	−)	(−)	Initial
SA	(−)	12	(−)	(−)	4	(−)	7	(−	−)	(−)	8	(−	−)	(−)	SA[11]=8, $7 \in \mathcal{S}$, place 7
SA	(−)	12	(−)	(−)	4	6	7	(−	−)	(−)	8	(−	−)	(−)	SA[7]=7, $6 \in \mathcal{S}$, place 6
SA	(−)	12	(−)	(−)	4	6	7	(−	−)	(−)	8	(−	−)	(−)	SA[6]=6, $5 \notin \mathcal{S}$, no action
SA	(−)	12	3	(−)	4	6	7	(−	−)	(−)	8	(−	−)	(−)	SA[5]=4, $3 \in \mathcal{S}$, place 3
SA	(−)	12	3	(−)	4	6	7	(−	−)	(−)	8	(−	−)	(−)	SA[3]=3, $2 \notin \mathcal{S}$, no action
SA	(−)	12	3	(−)	4	6	7	(−	−)	(−)	8	(−	−)	(−)	SA[2]=12, $11 \notin \mathcal{S}$, no action
group	\$	ab		b\$	bd	cc	cd	da		dc	de	ed		ee	

Clearly, in general, $|\mathcal{T}| \leq |\mathcal{S}|$ (consider the string $a^n\$$ to see the case where $|\mathcal{T}| = |\mathcal{S}|$). However, in [Maniscalco and Puglisi 2006a] measurements are given indicating that in practice there is a marked difference between the schemes, with $\mathcal{T}$ rarely containing more than 30% of suffixes.

Algorithm MP collects the sample suffix pointers in an array $\mathrm{SP}[1..|\mathcal{T}|]$ and sorts them with a variation of MKQS. Before sorting begins, however, the input string $x[1..n]$ is mapped onto an array of integers $\mathrm{ISA}'[1..n]$ by setting $\mathrm{ISA}'[i]$ to be the tail of the 1-group boundary for symbol $x[i]$. An important property of the transformation is that the lexorder of the suffixes of ISA' is the same as for x, so sorting the suffixes of ISA' is equivalent to sorting those of x. This recoding allows for the first of Algorithm MP's heuristics[3].

For each symbol $\alpha \in \Sigma$ a counter next_rank$[\alpha]$ is maintained, initially set to the head of the 1-group for α. When the final position of a sample suffix i prefixed with α becomes known relative to the other sample suffixes in SP, $\mathrm{ISA}'[i]$ is set to next_rank$[\alpha]$ and next_rank$[\alpha]$ is incremented. Because of the initial transform and the maintenance of next_rank counters, MKQS continues to sort suffixes correctly in the face of the changing ISA' values. The only other requirement for this to work correctly is that when suffix $\mathrm{SP}[i]$ is placed in its final position, suffixes in $\mathrm{SP}[0..i-1]$ are already sorted — this is a natural consequence of MKQS.

Modifying ISA' on-the-fly in this way will ultimately lead to faster sorting as the number of unique symbols (sort keys) in ISA' increases. The idea is essentially the *induction sorting* technique utilised by algorithm M (see above); the major difference here is that the sorted rank information is stored so it is immediately available when it is needed and does not have to be retrieved from another part of memory — the MKQS procedure carries on regardless, and the use of rank information is "seamlessly" integrated. This ultimately means algorithm MP will incur far fewer CPU-cache misses than algorithm M.

It is also shown in [Maniscalco and Puglisi 2006b] how MKQS can be modified

[3] Actually, the transformation is more complex, but this simplified version will do for our purposes here.

to reveal ***repetitions*** (adjacent repeating substrings) in the input and how, once revealed, they can sort them efficiently without the need for extraneous symbol comparisons. This allows MP to avoid the catastrophic slowdowns incurred by many induced copying algorithms on highly repetitive inputs.

Because ISA$'$ and SA can be implemented to use the same memory space and because the maximum sample size is $n/2$, algorithm MP is able to cap its memory usage at $6n$ bytes, and in practice uses little more than $5n$ bytes. It is also the fastest algorithm we tested.

3.4 Hybrid Algorithms

In this section we briefly describe three hybrid SACAs not previously described in the literature or implemented. While there are likely many ways in which the ideas of various SACAs can be combined, our aim here is to deal with the obvious cases and close some open questions in the literature. The hybrids are as follows:

(1) IT+KA. It was observed earlier that algorithms IT and KA divide suffixes in a similar way — in particular, IT chooses a subset of the suffixes chosen by KA. The hybrid algorithm selects suffixes as KA does: it labels each suffix L or S and chooses the set containing fewer members. These suffixes are then sorted with a string sorting algorithm into their final place in the L/S partitioned 1-groups (as in algorithm IT). The final phase is then a pass over the SA to move the remaining suffixes into place. Such an algorithm will undoubtedly be faster than IT as more suffixes have their order induced automatically. A disadvantage is that the type of a suffix can no longer be determined in constant time, unless n extra bits of working space are used.

(2) BK+S. This hybrid was mentioned by [Burkhardt and Kärkkäinen 2003] as a possible way to improve the average running time of algorithm BK. The combination of the ideas is simple: we select a sample of suffixes as in BK and sort them. Then sorting of the 1-groups in algorithm S using MKQS stops at depth h, where h is the modulus of the difference cover. To complete the sort the ranks of suffixes in the sample are used as sort keys. The pointer-copying of algorithm S operates as usual.

(3) BK+IT+KA. We can obtain a hybrid algorithm similar to BK+S by combining BK with algorithm IT, or perhaps better with the IT+KA hybrid described above.

Tests of these hybrids are included in the experiments described in the next section.

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[4] and from the corpus compiled by Manzini[5] and Ferragina [Manzini and Ferragina 2004]. Details of all files tested are given in Table II. The table also provides several statistics for

[4] `http://www.cosc.canterbury.ac.nz/corpus/`
[5] `http://www.mfn.unipmn.it/~manzini/lightweight/corpus/`

Table II. 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
eco	17	2,815	4,638,690	4	*Escherichia coli* genome
chr	1,979	199,999	34,553,758	5	Human chromosome 22
bibl	14	551	4,047,392	63	King James bible
worl	23	559	2,473,400	94	CIA world fact book
prot	89	7,373	109,617,186	66	SwissProt database
rfc	93	3,445	116,421,901	120	Concatenated IETF RFC files
how	267	70,720	39,422,105	197	Linux Howto files
reut	282	26,597	114,711,151	93	Reuters news in XML format
jdk	679	37,334	69,728,899	113	JDK 1.3 documentation
etext	1,108	286,352	105,277,340	146	Texts from Gutenberg project

each file; the Mean and Maximum LCP values provide a rough guide to suffix sorting difficulty [Sadakane 1998], and give the average and maximum number of character comparisons respectively, required by a string sorting algorithm to separate two suffixes.

We implemented Algorithm IT as described in [Itoh and Tanaka 1999] and Algorithm KS with heuristics described in [Puglisi et al. 2005]. We also implemented the hybrid algorithms IT+KA, BK+IT+KA and BK+S, and had them use MKQS for string sorting. Two implementations of Algorithm KA were tested: one by Lee and Park, and the other due to Ko [Lee and Park 2004; Ko 2006]. 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 [Kurtz 1999]. We did not include algorithm MM in experiments because results in [Larsson and Sadakane 1999] show it to be many times slower than other algorithms such as LS. We are confident that all tested implementations are of high quality.

Algorithm MF was run with default parameters and Algorithm SS with parameter $h = 7$ for genomic data (files eco and chr) and $h = 3$ otherwise, as used for testing in [Schürmann and Stoye 2005]. Algorithm BK and the hybrids BK+S and BK+IT+KA used parameter $h = 32$, as in [Burkhardt and Kärkkäinen 2003].

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 III, 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 IV, was recorded with the `memusage` command available with most Linux distributions.

Results are summarized in Figure 8. Algorithm MP is the fastest (or equal fastest) algorithm on all files, and shades algorithm M by about 33% on average. These two algorithms (MP, M) have a clear advantage over the next fastest algorithms, MF and SS, which are approximately 70% and 80% slower on average respectively, than MP. On the shorter files (eco,bib,wor), the times of MP are equalled by several algorithms. This result fits with the observations of several authors that on short files, which tend to have low average LCP, simple SACAs that do not deviate tremendously from their underlying string sorting algorithms

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

	eco	chr	bib	wor	prot	rfc	how	reut	jdk	etex
MP	**2**	**15**	**1**	**1**	**49**	**48**	**14**	**57**	**30**	**50**
M	**2**	18	2	**1**	59	61	18	73	44	58
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
IT+KA	**2**	205	**1**	**1**	119	86	31	281	274	335
S	3	29	2	**1**	126	110	37	258	217	290
BK	4	40	3	2	200	171	43	280	152	141
BK+IT+KA	3	27	2	**1**	129	103	27	176	97	116
BK+S	3	29	3	2	164	126	31	240	132	116
LS	4	35	3	2	144	154	40	183	105	146
BB	3	26	3	2	117	113	43	84	40	131
KA(Ko)	5	43	4	2	118	109	39	121	57	138
KA(Lee)	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

give acceptable behaviour.

The speed of MP, M and MF for the larger inputs is particularly impressive given their small working memory: $5.13n$, $5.49n$ and $5.01n$ bytes on average respectively. The lightweight nature of these algorithms separates them from SS which requires slightly more than $9n$ bytes on average. Times in Table III for Algorithm SS versus Algorithm MF seem to run contrary to results published in [Schürmann and Stoye 2005]; however, our experiment is different. In [Schürmann and Stoye 2005], 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.

It is important to note an advantage of MF over MP is the stability of its memory use. In the worst case MF can garauntee only $5.01n$ bytes will be used, whereas MP, in the worst case, may require $6n$ bytes. Thus, when memory is especially tight, MF may be the algorithm of choice. We also remark that while Algorithm BK is not amongst the fastest algorithms tested, its central concept (the difference cover) provides a simple strategy to provide $O(n \log n)$ worst case behavior to the faster algorithms without heavily impacting on their speed or space usage.

All three hybrid algorithms improve the average speed of their component algorithms. Algorithm IT+KA removes the wild variation of IT's runtime by reducing the amount of suffixes sorted with MKQS to less than $n/2$ for all files. This is at the cost of n bits of space required to store the type of each suffix. As one would expect, algorithms BK+S and BK+IT+KA improve on the runtime of BK for every file by reducing the need for string sorting. In constrast, BK brings stability to S and IT+KA: variation in runtime is diminished, but a slight slowdown is incurred on some files. Such slowdowns occur when the time taken to sort the sample required by BK outweighs the time saved by limiting the depth of the string sort. We observed the time taken for this phase was around 20-25% of overall runtime — a significant improvement here would make the BK hybrids more competetive with the leaders.

The large variation in performance of Algorithm KS can be attributed to the

Table IV. Peak Memory Usage (Mbs)

	eco	chr	bib	wor	prot	rfc	how	reut	jdk	etex
MP	23	167	20	12	542	560	194	571	340	572
M	25	182	21	14	555	580	197	564	342	519
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
IT+KA	23	169	19	12	536	569	193	561	340	514
S	22	165	19	12	523	555	188	547	332	502
BK	26	194	23	14	614	652	221	643	391	590
BK+IT+KA	27	198	23	14	627	666	226	657	399	603
BK+S	26	194	23	14	614	652	221	643	391	590
LS	35	264	31	19	836	888	301	875	532	803
BB	78	580	68	42	1,840	1,954	662	1,925	1,170	1,767
KA(Lee)	58	429	50	31	1,359	1,443	526	1,422	864	1,406
KA(Ko)	47	332	32	19	805	752	282	847	495	832
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

occasional ineffectiveness of heuristics described in [Puglisi et al. 2005]. 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 from very poor cache behaviour, which largely nullifies their asymptotic advantage.

5. CONCLUDING REMARKS

Over the last few years, suffix arrays — algorithms for their construction and use — have constituted, along with the closely-related topic of string sorting, an intense area of research within computer science. While this paper was in review two new (as yet unpublished) algorithms appeared on the Internet that are nearly as fast as Algorithm MP and use a similar amount of space [Malyshev 2006; Mori 2006].

The algorithms surveyed in this paper are a testament to the ingenuity of many researchers who have collectively made the suffix array the data structure of choice for a wide range of applications — replacing the suffix tree, whose "myriad virtues" were already well-recognized 20 years ago [Apostolico 1985].

Impressive as the progress has been, ingenious as the methods have been, there still remains the challenge to devise a SACA that is lightweight, linear in the worst case, and fast in practice.

We hope that by a timely exposition of existing SACAs to a wider audience, we will contribute to further progress in a fascinating and important area of research.

ACKNOWLEDGMENTS

The authors would like to thank Dror Baron, Dong Kyue Kim, Stefan Kurtz, Sunglim Lee and Kunsoo Park for responding to requests for source code, and to all authors who made code accessible via their web pages. They also thank three anonymous referees for constructive comments that have improved this paper. A

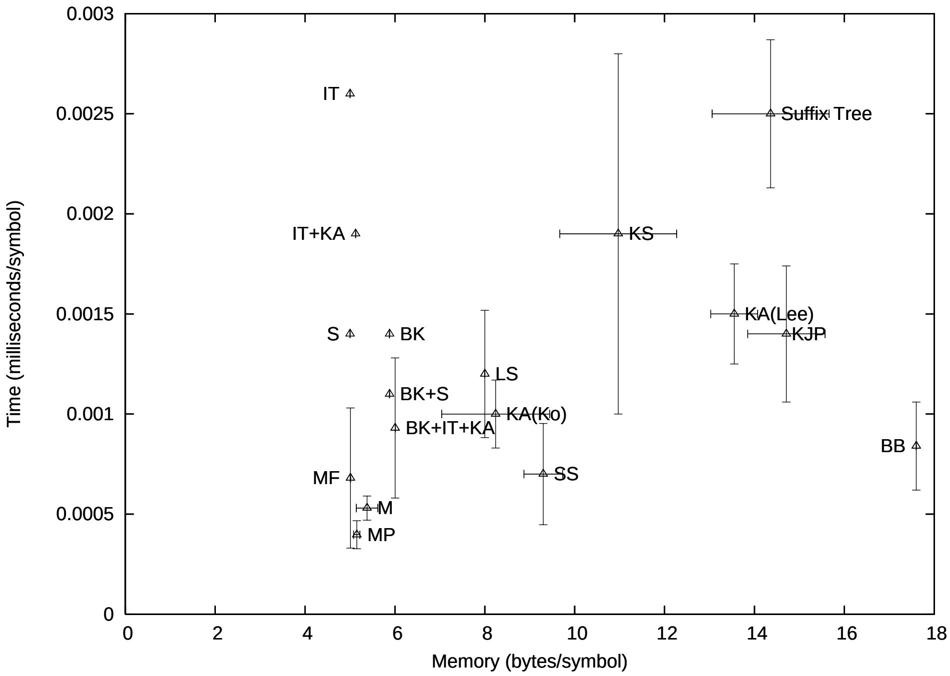

Fig. 8. Resource requirements of the algorithms averaged over the test corpus. Error bars are one standard deviation. Abscissa error bars for algorithms MF, S, IT, IT+KA, BK, BK+S, BK+IT+KA, BB and LS are insignificantly small. Ordinate error bars for algorithms S, IT, IT+KA, BK and BK+S are not shown to improve presentation (standard deviations 0.009, 0.0036, 0.0019, 0.0006 and 0.00053 respectively).

preliminary version of this paper appeared as "A taxonomy of suffix array construction algorithms", S. J. Puglisi, W. F. Smyth, A. Turpin, *Proceedings of the Prague Stringology Conference*, Prague, Czech Republic, Jan Holub, Ed., August, 2005. This work is supported in part by grants from the Natural Sciences & Engineering Research Council of Canada and the Australian Research Council.

REFERENCES

Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. 2004. Replacing suffix trees with suffix arrays. *Journal of Discrete Algorithms 2,* 1, 53–86.

Apostolico, A. 1985. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*. NATO ASI Series F12. Springer-Verlag, Berlin, 85–96.

Baron, D. and Bresler, Y. 2005. Antisequential suffix sorting for bwt-based data compression. *IEEE Transactions on Computers 54,* 4 (April), 385–397.

Bentley, J. L. and McIlroy, M. D. 1993. Engineering a sort function. *Software, Practice and Experience 23,* 11, 1249–1265.

Bentley, J. L. and Sedgewick, R. 1997. Fast algorithms for sorting and searching strings. In

Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms. New Orleans, Louisiana, 360–369.

Burkhardt, S. and Kärkkäinen, J. 2003. Fast lightweight suffix array construction and checking. In *Proceedings of the 14th Annual Symposium CPM 2003*, R. Baeza-Yates, E. Chávez, and M. Crochemore, Eds. Lecture Notes in Computer Science, vol. 2676. Springer-Verlag, Berlin, 55–69.

Burrows, M. and Wheeler, D. J. 1994. A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation, Palo Alto, California.

Crauser, A. and Ferragina, P. 2002. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica 32*, 1–35.

Farach, M. 1997. Optimal suffix tree construction for large alphabets. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, 137–143.

Ferragina, P. and Grossi, R. 1999. The string b-tree: a new data structure for search in external memory and its applications. *Journal of the ACM 46,* 2, 236–280.

Grossi, R. and Vitter, J. S. 2005. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing 35,* 2, 378–407.

Hart, M. 1997. Project Gutenberg. http://www.gutenberg.net.

Hon, W., Sadakane, K., and Sung, W. 2003. Breaking a time-and-space barrier in constructing full-text indices. In *Proceedings of the 44th IEEE Symposium on Foundations of Computer Science (FOCS'03)*. IEEE Computer Society Press, 251–260.

Itoh, H. and Tanaka, H. 1999. An efficient method for in memory construction of suffix arrays. In *Proceedings of the sixth Symposium on String Processing and Information Retrieval*. IEEE Computer Society, Cancun, Mexico, 81–88.

Kärkkäinen, J. and Sanders, P. 2003. Simple linear work suffix array construction. In *Proceedings of the 30th International Colloquium Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 2971. Springer-Verlag, Berlin, 943–955.

Kärkkäinen, J., Sanders, P., and Burkhardt, S. 2006. Linear work suffix array construction. *Journal of the ACM*. to appear.

Karp, R. M., Miller, R. E., and Rosenberg, A. L. 1972. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the fourth annual ACM symposium on Theory of computing*. Denver, Colorado, United States, 125–136.

Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K. 2001. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Annual Symposium CPM 2001*. Lecture Notes in Computer Science, vol. 2089. Springer-Verlag, Berlin, 181–192.

Khmelev, D. V. 2003. Program suffsort version 0.1.6. http://www.math.toronto.edu/dkhmelev/PROGS/tacu/suffsort-eng.html.

Kim, D. K., Jo, J., and Park, H. 2004. A fast algorithm for constructing suffix arrays for fixed-size alphabets. In *Proceedings of the 3rd Workshop on Experimental and Efficient Algorithms (WEA 2004)*, C. C. Ribeiro and S. L. Martins, Eds. Springer-Verlag, Berlin, 301–314.

Kim, D. K., Sim, J. S., Park, H., and Park, K. 2003. Linear-time construction of suffix arrays. In *Proceedings of the 14th Annual Symposium Combinatorial Pattern Matching*, R. Baeza-Yates, E. Chávez, and M. Crochemore, Eds. Lecture Notes in Computer Science, vol. 2676. Springer-Verlag, Berlin, 186–199.

Kim, D. K., Sim, J. S., Park, H., and Park, K. 2005. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms 3*, 126–142.

Ko, P. 2006. Linear time suffix array. http://www.public.iastate.edu/~kopang/progRelease/homepage.html.

Ko, P. and Aluru, S. 2003. Space efficient linear time construction of suffix arrays. In *Proceedings of the 14th Annual Symposium CPM 2003*, R. Baeza-Yates, E. Chávez, and M. Crochemore, Eds. Lecture Notes in Computer Science, vol. 2676. Springer-Verlag, Berlin, 200–210.

Ko, P. and Aluru, S. 2005. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms 3*, 143–156.

KURTZ, S. 1999. Reducing the space requirement of suffix trees. *Software, Practice and Experience 29,* 13, 1149–1171.

LARSSON, J. N. AND SADAKANE, K. 1999. Faster suffix sorting. Tech. Rep. LU-CS-TR:99-214 [LUNFD6/(NFCS-3140)/1-20/(1999)], Department of Computer Science, Lund University, Sweden.

LEE, S. AND PARK, K. 2004. Efficient implementations of suffix array construction algorithms. In *AWOCA 2004: Proceedings of the Fifteenth Australasian Workshop on Combinatorial Algorithms*, S. Hong, Ed. 64–72.

MÄKINEN, V. AND NAVARRO, G. 2006. Compressed full text indexes. Tech. Rep. TR/DCC-2006-6, Department of Computer Science, University of Chile. April.

MALYSHEV, D. 2006. DARK the universal archiver based on BWT-DC scheme. http://darchiver.narod.ru/.

MANBER, U. AND MYERS, G. W. 1990. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first ACM-SIAM Symposium on Discrete Algorithms*. 319–327.

MANBER, U. AND MYERS, G. W. 1993. Suffix arrays: a new method for on-line string searches. *SIAM Journal of Computing 22,* 5, 935–948.

MANISCALCO, M. A. 2005. MSufSort. http://www.michael-maniscalco.com/msufsort.htm.

MANISCALCO, M. A. AND PUGLISI, S. J. 2006a. An efficient, versatile approach to suffix sorting. *ACM Journal of Experimental Algorithmics*. to appear.

MANISCALCO, M. A. AND PUGLISI, S. J. 2006b. Faster lightweight suffix array construction. In *Proceedings of 17th Australasian Workshop on Combinatorial Algorithms*, J. Ryan and Dafik, Eds. 16–29.

MANZINI, G. 2004. Two space saving tricks for linear time LCP computation. In *Proceedings of 9th Scandinavian Workshop on Algorithm Theory (SWAT '04)*, T. Hagerup and J. Katajainen, Eds. Lecture Notes in Computer Science, vol. 3111. Springer-Verlag, Berlin, 372–383.

MANZINI, G. AND FERRAGINA, P. 2004. Engineering a lightweight suffix array construction algorithm. *Algorithmica 40*, 33–50.

MCILROY, M. D. 1997. ssort.c. http://cm.bell-labs.com/cm/cs/who/doug/source.html.

MCILROY, P. M., BOSTIC, K., AND MCILROY, M. D. 1993. Engineering radix sort. *Computing Systems 6,* 1, 5–27.

MORI, Y. 2006. DivSufSort. http://www.homepage3.nifty.com/wpage/software/libdivsufsort.html.

MUNRO, J. I. 1999. Succinct data structures. In *Proceedings of Workshop on Data Structures*.

NA, J. C. 2005. Linear-time construction of compressed suffix arrays using O(nlogn)-bit working space for large alphabets. In *Proceedings of the 16th Annual Symposium Combinatorial Pattern Matching*, A. Apostolico, M. Crochemore, and K. Park, Eds. Lecture Notes in Computer Science, vol. 3537. Springer-Verlag, Berlin, 57–67.

PUGLISI, S. J., SMYTH, W. F., AND TURPIN, A. H. 2005. The performance of linear time suffix sorting algorithms. In *Proceedings of the IEEE Data Compression Conference*, M. Cohn and J. Storer, Eds. IEEE Computer Society Press, Los Alamitos, CA, 358–368.

SADAKANE, K. 1998. A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation. In *DCC: Data Compression Conference*. IEEE Computer Society Press, Los Alamitos, CA, 129–138.

SCHÜRMANN, K. AND STOYE, J. 2005. An incomplex algorithm for fast suffix array construction. In *Proceedings of The Seventh Workshop on Algorithm Engineering and Experiments (ALENEX05)*. SIAM, 77–85.

SEWARD, J. 2000. On the performance of BWT sorting algroithms. In *DCC: Data Compression Conference*. IEEE Computer Society Press, Los Alamitos, CA, 173–182.

SIM, J. S., KIM, D. K., PARK, H., AND PARK, K. 2003. Linear-time search in suffix arrays. In *Proceedings of the 14th Australasian Workshop on Combinatorial Algorithms*, M. Miller and K. Park, Eds. Seoul, Korea, 139–146.

SINHA, R. AND ZOBEL, J. 2004. Cache-conscious sorting of large sets of strings with dynamic tries. *ACM Journal of Experimental Algorithmics 9*.

SMYTH, B. 2003. *Computing Patterns in Strings*. Pearson Addison-Wesley, Essex, England.

Received Month Year; revised Month Year; accepted Month Year