Technical Report Number CAS-05-02-WS (May, 2005)
Dept of Computing and Software, McMaster University
Hamilton, Ontario, Canada

# Exposition and Analysis of a Suffix Sorting Algorithm

SIMON J. PUGLISI*

## 1 Introduction

This paper focuses on the suffix sorting algorithm of Maniscalco [10], which at the time of writing is available only as C++ source code on the Internet. We will refer to the program as *MSufSort*.

MSufSort computes the Inverse Suffix Array (ISA) of an input string, which is equivalent to computing the Suffix Array (converting one to the other is discussed in section 8). Recall that for $i \in [0..n-1]$, ISA$[i]$ gives the lexicographic rank of the suffix $x[i..n-1]$ amongst all the other suffixes of the string $x[0..n-1]$. Experiments summarized in [10] suggest that MSufSort outperforms the fastest known suffix sorting programs, while using little extra space aside from the $4n$ bytes to hold the suffix array and the $n$ bytes for the input string (in the terms of [11] it would be *lightweight*). It is also purported to perform well on *periodic* strings, which are known to be catastrophic worst cases for some algorithms.

This paper addresses the need for a more formal examination of what appears to be a very robust suffix sorter. We examine and describe the inner workings of the algorithm, and try to explain why MSufSort performs well by analyzing its asymptotic behavior. As published in [10], the MSufSort source code crosses several classes and files and is not easy absorb in a single sitting. The code presented in this paper constitutes a complete rewrite of the original as just a few C functions, and is included not as an optimization, but rather to aid explanation of the approach.

After introducing some notation in Section 2, the basic algorithm is described in Section 3 before two powerful heuristics are introduced in Sections 4 and 5. Sections 6 and 7 consider time and space usage. Section 8 discusses ISA to SA transformation. In Section 9 we extensively test MSufSort and compare its performance to that of other leading suffix sorters. Possible areas for future work are outlined in Sections 10 and 11, and brief conclusions are offered in Section 12.

It is assumed the reader is familiar with the concept of suffix sorting and its applications, particularly the Burrows-Wheeler Transformation (BWT).

## 2 Assumptions and Notation

This section establishes some notation and assumptions used throughout. We consider a string $x[0..n] = x[0]x[1]...x[n]$ of $n+1$ symbols. The first $n$ symbols of $x$ are drawn from the alphabet, $A=\{\alpha_1, \alpha_2, ...\alpha_{|A|}\}$ of size $\alpha$, and comprise the actual string. We define $x[n] = \$$ to be a unique symbol, not in $A$, and lexicographically less than all symbols in $A$. Let $S_i = x[i]x[i+1]...x[n]$ (or sometimes just suffix $i$, or the $i$th suffix) denote the suffix beginning

*Dept of Computing, Curtin University of Technology, Perth, Australia (puglissj@computing.edu.au)

at position $i$ in $x$. To store the suffix $S_i$, we store only the starting position $i$. Substrings of $x$ having length 2 will be referred to as *bigrams*.

The *suffix array*, $SA$, is an array of integers in the range 0 to $n$, where each integer corresponds to a suffix of the string $x$. The integers specify the ascending lexicographic ordering of the suffixes, that is, $S_{SA[i-1]}$ lexicographically precedes $S_{SA[i]}$ for all $i \in [0, n-1]$. We will also be interested in the inverse permutation of the suffix array, the *inverse suffix array*, denoted ISA. For all $i \in [0, n]$, ISA$[i]$ gives the *rank* of $S_i$, that is, ISA$[i] = k$ if and only if SA$[k] = i$.

For reasons which will become clear, we will assume that $n < 2^{32} - 2$ and that $A$ is a constant, indexed alphabet, with $\alpha < 255$[1].

## 3 Sorting strings with buckets

At the heart of MSufSort is an efficient bucket sorting regime. Most of the work is done in an array of $n$ integers (eventually the ISA), with extra space required for a few stacks. The bucket sorting begins by linking together all the suffixes having the same first *two* characters to form *chains* of suffixes[2]. For example, the string

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{x}[i]$ | $a$ | $a$ | $a$ | $b$ | $a$ | $b$ | $a$ | $a$ | \$ |

would result in the creation of the following chains

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

We define an $\ell$-*chain*, $L_\ell$, as a set of suffixes such that for all $S_i$ and $S_j \in L$, $x[i..i+\ell-1] = x[j..j+\ell-1]$. In other words, $S_i$ and $S_j$ are in the same $\ell$-chain iff they share a length $\ell$ common prefix. If an $\ell$-chain contains only one suffix we say it is a *singleton*. 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, MSufSort computes the equivalent array

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{x}[i]$ | $a$ | $a$ | $a$ | $b$ | $a$ | $b$ | $a$ | $a$ | \$ |
| ISA$[i]$ | $\perp$ | 0 | $\perp$ | $\perp$ | 2 | 3 | 1 | $\perp$ | |

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[3]. In our example, the chain of all the suffixes prefixed with $aa$ contains $S_6$, $S_1$ and $S_0$ and so we have ISA$[6] = 1$, ISA$[1] = 0$ and ISA$[0] = \perp$, marking the end of the chain. Observe that chains are singly linked, and are only traversable right-to-left. We keep track of $\ell$-chains to be processed by storing a stack of integer pairs $(h, \ell)$ where $h$ is the head of the chain (its rightmost index), and $\ell$ is the length of the common prefix shared by the suffixes in the chain. Chains always appear on the stack in ascending lexicographical order, according

[1] This will suit many practical situations.

[2] So we have at most $\alpha^2$ chains initially. Of course, we could make chains based on inspection of just the first character of each suffix. The combining of two characters into one is a practical choice, speeding up sorting, but requiring the alphabet size to be less than 256.

[3] In practice we reserve $2^{32} - 1$ to represent $\perp$.

to $x[h..h+\ell-1]$. Thus for our example, initially $(7,2)$ for chain $a\$$ is atop the stack, and $(5,2)$ for chain $ba$ at the bottom.

Chains are popped from the stack and progressively refined by looking at further pairs of characters. So long as we take care to process the chains in lexicographical order, when we pop a singleton chain, the suffix contained has been differentiated from the rest and can be assigned the next lexicographic rank. We refer to a chain containing only one suffix as a *singleton*. This means ranks are assigned from 1 through to n, strictly in that order. Elements in the ISA which are ranks are differentiated from elements in lists by setting the sign bit, that is, if $\text{ISA}[i] < 0$, then the rank for $S_i$ is $\text{ISA}[i]$. The evolution of the ISA of our example string over subsequent sorting rounds is illustrated below.

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

When the value in a column becomes negative, the suffix has been assigned its (negated) rank and is effectively sorted. We reiterate here that when a chain is split, the resulting subchains must be placed on the stack in lexicographical order for the correct assignment of ranks to singletons. This is illustrated in the example above when the chain for $aa$ is split, and the next chain processed is the singleton chain for $aa\$$. A high level algorithm embodying these ideas is listed in Figure 1 and a full implementation in C code is shown in Figure 10.

Results in [15] would suggest that direct comparison bucket sorting alone would not make a competitive suffix sorter. In coming sections we will see how MSufSort exploits properties inherent to this basic approach to accelerate sorting.

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

Figure 1: The bucket sorting central to MSufSort.

## 4 Exploiting previously ranked suffixes

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

When processing a chain with common prefix length $\ell$ we can classify suffixes into three types: Suffix $S_i$ is of type $A$ if the rank for suffix $S_{i+\ell-1}$ is known, and is of type $B$ if the rank for suffix $S_{i+\ell}$ is known. If $S_i$ is not of type $A$ or type $B$, then it is of type $C$. We can classify a suffix to its type in constant time by virtue of the fact we are building the ISA (we inspect ISA$[i+\ell-1]$ or ISA$[i+\ell]$ and a checked sign bit indicates a rank). Now consider the following observation:

**Observation 1.** *Lexicographically, type A suffixes are smaller than type B suffixes, which in turn are smaller than type C suffixes.*

To use this observation, when we refine a chain, we place only type $C$ suffixes into subchains and place type $A$ and type $B$ suffixes to one side. Now, the order of the $m$ type $A$ suffixes can be determined via a comparison based sort, using for $S_i$ the rank of $S_{i+\ell-1}$ as the sort key. Once sorted, the type $A$ suffixes can be assigned the next $m$ ranks by virtue of the fact that chains are being processed in lexicographical order. The type $B$ suffixes are treated similarly, using the rank of $S_{j+\ell}$ as the sort key for $S_j$. [4] For the comparison sort, MSufSort employs a fast implementation of the approach described in [12]. Maniscalco refers to sorting suffixes this way as *induction sorting*.

Loosely speaking, as the number of assigned ranks increases, the probability that a suffix can be sorted by induction also increases. In fact, every chain of suffixes with prefix $\alpha_1\alpha_2$ such that $\alpha_2 < \alpha_1$ will be sorted entirely by induction. Clearly, induction sorting will lead to a significant reduction in work for many texts. We analyze the extent of this reduction in Section 6.

Induction sorting shares something with both the *two-stage* algorithm of Itoh and Tanaka [6] and *cache* algorithm of Seward [16]. We briefly review these two algorithms here and refer the reader to [6, 16, 11] for further details. Note that both *two-stage* and *cache* construct the *SA* directly, rather than the *ISA* first as MSufSort does.

Algorithm *cache* maintains an array $C[0..n-1]$ of integers, initially all 0. When $S_i$ is known to be in its final place, $k$, in *SA* we know that $S_i$ is ranked $k$ amongst all suffixes and place the most significant 16 bits (or alternatively 8 bits) of $k$ in $C[i]$. Later, if we are comparing suffixes $S_j$ and $S_k$ having $x[j] = x[k]$, then we compare $C[j+1]$ and $C[k+1]$, a difference giving us the correct order of $S_j$ and $S_k$. If $C[j+1] = C[k+1]$ then we inspect $x[j+1] = x[k+1]$ and if they are equal then we compare $C[j+2] = C[k+2]$ and so on. The possibility of equal $C$ values arises of course because we are only caching the most significant portion of each rank. This truncation of ranks means only an extra $2n$ bytes is required to store $C$ (or $1n$ if only 8 bits are used).

MSufSort can be thought of as not only a very space efficient version of the *cache* scheme, but also as one that allows use of *full* rank information, not just 16 bits. These improvements are possible primarily because MSufSort manipulates the *ISA* rather than the *SA*. Another key difference is the lexicographic order in which ranks are assigned and become available for use in induction sorting.

[4] In fact, we can sort the type $A$ and $B$ suffixes in the same sort call by using as a key for a type $A$ suffix $S_i$ the rank of $S_{i+\ell-1}$ and for a type $B$ suffix the *negated* rank of $S_{i+\ell}$.

```
formInitialChains()
repeat
  (h, ℓ) ← chainStack.pop()
  if ISA[h] = ⊥ then
    ISA[h] ← nextRank()
  else
    while not ISA[h] ≠ ⊥ do
      if isRanked(h + ℓ − 1) then
        noteSuffix(h,-ISA[h + ℓ − 1])
      elsif isRanked(h + ℓ) then
        noteSuffix(h,ISA[h + ℓ])
      else
        sym ← getSymbol(h + ℓ)
        updateSubChain(sym,h)
      h ← ISA[h]
    pushSubChains()
    rankNotedSuffixes()
until chainStack is empty
```

Figure 2: The way *induction sorting* integrates with bucket sorting in MSufSort.

Algorithm *two-stage* uses a counting sort to logically partition the $SA$ space into $\alpha$ buckets, with the $i$th bucket containing the suffixes having first character $\alpha_i$. Each bucket is further partitioned, the first portion containing type $X$ suffixes, and the second containing type $Y$ suffixes. Type $X$ suffixes have prefix $\alpha_1\alpha_2$ such that $\alpha_1 > \alpha_2$. Type $Y$ have $\alpha_1 \leq \alpha_2$. Note that within a bucket type $X$ suffixes always come before type $Y$ suffixes. The key observation of Itoh and Tanaka is that once all the groups of type $Y$ suffixes are sorted, the order of the type $X$ suffixes is implied. More precisely, with the type $Y$ suffixes sorted (with a direct comparison algorithm such as [2]), algorithm *two-stage* makes a single pass over $SA$ and for each suffix $S_i$ encountered, if $S_{i-1}$ is of type $X$, then it should be moved the current front of its bucket in $SA$ and the bucket front is incremented. In this way the type $X$ suffixes at least are sorted in *O(n)* time.

In a way, the induction sorting of MSufSort can be considered an extension of the *two-stage* algorithm. 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 $X$ suffixes of *two-stage*. However the lexicographical processing of suffixes in MSufSort means induction sorting can be applied to suffixes in $\ell$-chains with $\ell > 2$.

## 5 Detecting and processing repetitions

We now describe how the right-to-left chains of suffixes formed during bucket sorting enable us to detect *repetitions* in the input string and sort the suffixes involved in them efficiently.

**Definition 1.** *A* **repetition** *(or* tandem array *or* periodicity*) in a string is a substring of the form $u^r$, where $u$ is any non-empty substring and $r \geq 2$. We call $u$ the generator of the repetition, $p = |u|$ the period of the repetition, and $r$ the exponent. A repetition located at position $i$ in a string is succinctly represented by the tuple $u^r = (i, p, r)$. A* square *is a repetition with $r = 2$.*

For example, the string $x[0..17] = abcaaaabcdabcdabcdab$ contains the repetitions: $a^4 = (3, 1, 4)$; $abcd^3 = (6, 4, 3)$; $bcda^3 = (7, 4, 3)$; $cdab^2 = (8, 4, 2)$ and $dabc^2(9, 4, 2)$.

Highly *periodic* strings (containing many repetitions) are well known difficult cases for suffix sorting algorithms as the average common prefix between suffixes in such strings is very high. Detecting repetitions in strings is a well studied problem in stringology (see [17]). Such strings are known to be particularly catastrophic for direct comparison suffix sorting algorithms such as [15, 11] as illustrated by experiments in [7, 14, 13].

When processing an $\ell$-chain, if we encounter adjacent positions in the list $i$ and $j$, $j < i$, such that $i - j = \ell$ then there exists a square, beginning at $j$, $x[j..\ell - 1]^2$. A sequence of such $\ell$ spaced elements in the list constitutes a repetition[5]. Checking for repetitions in this way integrates naturally with the processing of suffix chains.

Consider the string

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{x}[i]$ | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ \$ |

After forming the initial chains we have

| 6,4,2,0 | 5,3,1 | 7 |
|---|---|---|
| $ab$ | $ba$ | $b$\$ |

Each of these chains has $\ell = 2$. We process the chains $ab$ first and detect the repetition $(i, p, r) = (0, 2, 4) = ab^4$ as each suffix in the list is spaced $\ell$ positions apart. Once we have detected the repetition $(i, p, r)$ we can sort the suffixes $i, i + p, i + 2p.., i + p(r - 1)$ relative to each other by making use of the following observation.

**Observation 2.** *If $i + p(r - 1)$ can be sorted by induction, then $S_i < S_{i+p} < ... < S_{i+p(r-1)}$ otherwise $S_i > S_{i+p} > ... > S_{i+p(r-1)}$.*

The correctness of this observation is due to the processing of the $\ell$-chains in lexicographically ascending order. In the coming discussion we will refer to $k = i+p(r-1)$ as the *terminating position* of the repetition in question, and to all the other positions $i, i+p, i+2p.., i+p(r-2)$ as *non-terminating* positions.

When more than one repetition is detected in an $\ell$-chain, things become trickier to deal with. In general we will have a set of repetitions $\mathcal{M} = \{...\}$ all with the same generator (because they have been detected in the same $\ell$-chain).

Suffixes in $\mathcal{M}$ are processed differently depending on whether the terminating position can be sorted by induction or not. $\mathcal{M}$ is divided into a set $\mathcal{I}$ containing repetitions with terminating position sortable by induction, and set $\mathcal{R}$ containing the others.

To sort $\mathcal{I}$, the terminating positions of repetitions in $\mathcal{I}$ are sorted along with other suffixes from the $\ell$-chain sortable by induction and ranks are assigned as normal. Once the order of the terminating positions is known, the order of the non-terminating positions in $\mathcal{I}$ can be determined. The algorithm in 3 outlines the process. Basically, for two repetitions $(i_1, \ell, r_1)$ and $(i_2, \ell, r_2)$ in $\mathcal{I}$, if $S_{i_1+p(r_1-1)} < S_{i_2+p(r_2-1)}$ then $S_{i_1+\ell(r_1-2)} < S_{i_2+\ell(r_2-2)}$ and so on for each non-terminating position in each repetition.

The repetitions in $\mathcal{R}$ must be dealt with more delicately. $\mathcal{R}$ contains repetitions for which terminating positions cannot be sorted by induction. To sort the non-terminating positions $\mathcal{R}$ we need to know the order of the terminating-positions, relative to the other suffixes in the

[5]This will detect repetitions with even period. It is easy to see how this idea could be extended to detect odd periods, by testing also if $i - j = \ell - 1$

```
Sort suffixes sortable by induction,
including terminating positions
for each suffix i just sorted do
  assign i ← nextRank()
  if i − ℓ is marked as a repetition then
    add i − ℓ to the end of P
while there are items in P do
  take r off the front of P
  assign r the next rank
  if r − ℓ is marked as a repetition
    add r − ℓ to the end of P
```

Figure 3: Ranking suffixes in repetitions sortable by induction.

$\ell$-chain not sortable by induction. We link together all the non-terminating positions from right to left to form a chain $\mathcal{C}$. Terminating positions are put in subchains as normal, along with other suffixes from the $\ell$-chain not sortable by induction. Now, we push chain $\mathcal{C}$ onto the chainstack *before* the other subchains. We continue to process chains from the chainstack, but now when we would normally assign a rank to a suffix, we instead add that suffix to the end of a list $\mathcal{Q}$. Later when we reach $\mathcal{C}$ on the stack, we can use the order of the suffixes in $\mathcal{Q}$ to order the suffixes in $\mathcal{C}$. The algorithm in Figure 4 outlines the processing of $\mathcal{C}$.

```
first ← last ← ⊥
for each q in Q do
  if q − ℓ is a terminating position then
    add q − ℓ to the end of P
while there are items in P do
  take r off the front of P
  assign r the next rank
  if r − ℓ is in C
    add r − ℓ to the end of P
for each q in Q do
  assign q the next rank
```

Figure 4: Ranking suffixes in repetitions not sortable by induction.

Via a very clever memory arrangement, MSufSort uses only constant extra space to implement the above schemes for dealing with repetitions. The clincher is implementing lists $\mathcal{Q}$, $\mathcal{P}$ and $\mathcal{C}$ with the ISA.

If $p \geq 2$ then the detection of $(i, p, r)$ implies the presence of another $p - 1$ repetitions in other $\ell$-chains. The sorting of $(i, p, r)$ means these other repetitions will be sorted by induction before they are detected as repetitions.

## 6 Time Complexity

The two heuristics described in previous sections help make MSufSort efficient, but they also make analysis of worst case behavior difficult. Periodic strings, which would postpone the use of induction sorting heuristic, are handled by the repetition heuristic. And strings with large period, which would postpone the repetitions heuristic, would necessarily allow for induction sorting of some suffixes because of our assumption that $\alpha < 256$. This section contains several

remarks about the *potential* complexity of MSufSort, but stops short of attempting to prove anything formally.

The performance of the algorithm is very heavily dependant on the alphabet being small and constant. For example, each time we split a chain we incur a sorting cost of $O(\alpha^2)$ time (if radix sort is used) to ensure subchains go onto the stack in lexicographic order. If $\alpha$ were not small relative to $n$ this procedure would swamp the algorithm. From time to time $\alpha$ is included in our analysis as a reminder of this assumption.

For a lower bound on complexity, $\Omega(n\log n)$ seems reasonable, because it is the cost of sorting $m$ groups of items of total length $n$. This would occur if every suffix was able to be sorted with the induction sorting heuristic.

The upper bound then will be determined by the time spent comparing pairs of characters before either induction sorting or the repetitions heuristic can be employed. Without the heuristics the chain refinement will take $O(n^2)$ time. One would expect the repetitions heuristic in particular to reduce this somewhat, perhaps even to $O(n)$, but we are unable to prove an improved bound as yet. Note though that a way to guarantee runtime in $O(n\log n)$ without the repetitions heuristic would be to augment MSufSort with the ideas of [7]. This would not affect the lightweight nature of MSufSort.

It is unclear whether radix sorting techniques would improve on the bounds claimed above. Sort keys are in the range $1..n$, and so each call to induction sort would require $O(n)$ time. Therefore, to prove that use of radix sort allows $O(n)$ overall performance, one must prove induction sort is called a constant number of times — it is not overly clear that this is possible. Further, using radix sort would increase memory requirements, with $4n$ bytes extra required for a frequency array in a straightforward implementation. With some difficulty one might get this requirement to less than $n$ bytes by exploiting the fact that sort keys are distinct[6], but this would likely hurt runtime.

## 7 Space Usage

MSufSort requires $n$ bytes to hold the input string, which is required throughout. The chains are handled in the $4n$ bytes which eventually contains the ISA. The heads of chains to be processed are maintained on a stack bound by $n/\alpha^2$, each item on which is 8 bytes (one integer for $h$, and one for $\ell$). Since we assume $n < 2^{32} - 2$ and $\alpha^2 = 2^{16}$, the maximum space required for the chain head stack is less than $2^{16} \times 8$ bytes for a total of 512Kb.

We now turn our attention to the stack holding suffixes waiting sorting by induction, the *induction stack*. Recall that each item on this stack requires 8 bytes - one integer for the suffix and one for its sort key. Generally, when the alphabet is small there are relatively few chains and so each chain will contain many suffixes. When the bigram list (ie. $\ell = 2$) for $\alpha_1\alpha_2$ is processed, if $\alpha_1 > \alpha_2$ then all $m$ suffixes in the list will be sorted by induction and so the induction stack will require $8m$ bytes. It is not hard to imagine often having $m = n/\alpha^2$ (or more), pushing the space for the induction stack out to $8n/\alpha^2$ bytes[7]. When $\alpha$ is very small and $n$ large, this will be a substantial amount of space. We can improve the situation however with the oft used trick of encoding multiple characters in a byte (see [9, 8, 14]). Specifically, each character in the input requires $b = \log_2\alpha$ bits, so we can store $c = \lfloor \frac{8}{b} \rfloor$ characters in a byte. Define a function $code_c(i)$, which produces a character, $d$, such that $code_c(i) < code_c(j)$

[6] This may be possible using ideas of Clark and Munro [4].

[7] When $\alpha = 4$ we would expect such bigrams about 40% of the time. And for the string $x[0..n-1] = (A[\alpha]A[\alpha-1]s(i))^n/3$ where $s(i) = A[i \bmod (\alpha-2)$ the chain for $A[\alpha]A[\alpha-1]$ will contain $n/3$ items.

if and only if $x[i..i+c]$ is lexicographically less than $x[j..j+c]$. We replace the character at each position in the input with $code_c(i)$. This recoding usually results in more lists and fewer items per list, and so helps keeps the induction stack size respectable. We found this technique reduced the memory usage of MSufSort in practice, particularly on genomic data (see Section 9).

In the worst case then, MSufSort may require a little more than $9n$ bytes, though in practice we have never found it to use more than $6n$, and usually closer to $5n$.

## 8 ISA to SA Transformation

MSufSort lends itself well to computing the Burrows-Wheeler Transformation of the input text: when a suffix $i$ receives its rank, the character in the corresponding final column of the BWT matrix is the $i$th character in the transformed text. When the transformation completes, the ISA has been computed as a byproduct. If one requires the SA rather than the ISA, the transformation can be done in place (constant extra memory required) and in linear time using the code in Figure 5.

```
for i ← 0 to n do
  if ISA[i] > 0
    start ← suff ← i
    rank ← ISA[i]-1
    repeat
      tmp ← ISA[rank]
      ISA[rank] ← -suff
      suff ← rank
      rank ← tmp
    until rank = start
    ISA[rank] ← -suff
```

Figure 5: Permute ISA into the SA in-place.

The idea is to cyclically displace ranks in the ISA, moving suffixes into their place in the SA. Say suffix $s$ with rank $q = \text{ISA}[s]$ is the next suffix we need to move into its place in the SA. We displace the rank at $\text{ISA}[q]$ to a temporary variable $r$, and set $\text{ISA}[q] = -s$, with the negated sign bit indicating the element is in place. Now, $r$ will be the position of suffix $q$ in the SA. We continue following the cycle in this way until the displaced rank, $r$ equals the index where we started, $s$. We then move to the right, until we find another element out of place, at which point we cycle again. When we reach the end of the array the ISA has been transformed to the SA.

The problem with this algorithm is its cache insensitivity: the next place in ISA accessed is essentially random, giving us little help from cache, with correspondingly slow runtime. If space is not an issue, we could allocate more memory for the SA move each suffix into place with a left to right pass over the ISA. We test these variations in Section 9.

## 9 Experiments

To assess practical performance we measured the runtimes and peak memory usage of MSufSort and other leading suffix sorting algorithms on a variety of inputs. We tested three variants of MSufSort: Algorithm MS, which computes the ISA and is suitable for computing the BWT; Algorithm MSI, which computes the ISA and transforms it to the SA in-place; and

Algorithm MSX, which uses $4n$ bytes of extra memory for the ISA to SA transformation[8]. Note that we only show runtimes for MSI and MSX because memory usage is the same for MSI as MS and MSX always requires exactly 4n bytes extra - average memory per symbol is shown for both variants in Figure 6.

As well as MSufSort we tested the $O(n \log n)$ algorithm of Larsson and Sadakane [8] (Algorithm LS); the so-called *deep-shallow* suffix sorter of Manzini and Ferragina [11] (Algorithm MF); and the recent *bucket pointer refinement* algorithm of Schürmann and Stoye [14]. All algorithms have source code available online. Algorithm MF was run with default parameters and Algorithm SS with parameter $d$=7 for genomic data and $d$=3 otherwise, as per testing in [14].

Files from the Canterbury corpus[9] and from the corpus compiled by Manzini[10] and Ferragina [11] listed in Table 1 were used for testing. To simulate difficult cases, we generated several files with large LCP values following the approach of Burkhardt and Kärkkäinen [7]. These synthetic files are repetitions of large random strings. We also tested on a large *Fibonacci* string.

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 2, are the average of four runs and do not include time spent reading input files. Times were recorded with the standard unix `time` function. Memory usage, shown in Table 4, was recorded with the `memusage` command available with most Linux distributions.

The times shown in Table 2 for Algorithm SS versus Algorithm MF seem to run contrary to results published in [14], however the experiment is actually different. In [14], files were truncated to be 50,000,000 characters at most, making many files from [11] shorter than they were originally, and we suspect this might have made them easier for Algorithm SS to sort.

## 10 Permuting the Alphabet

The lexicographic operation of MSufSort (that is, ranks are assigned strictly in order) prompted us to investigate the effects of alphabet permutations on the performance of the algorithm. One would expect a permutation to have an effect on the way the induction sorting heuristic is applied. The little prior work on alphabet permutations, in a BWT context at least has focused on compression rather than speed. In [3] it was demonstrated that specific alphabet permutations for different classes of text led to small improvements in compression (about 1%).

In Table 5 runtimes of MSufSort on texts with various alphabet permutations are given. The four permutations tested were the reverse mapping of the alphabet, a reordering based on ascending character frequency, a reordering based on descending character frequency, and a hand crafted permutation from [3] known to give good compression results for English text. The ordering of the alphabet was found to affect runtimes by between 8% and 10% in all cases, with the reversed alphabet in particular giving consistent improvements on the plain text. Improvements were greater for files containing mostly English text (howto, reuters and jdk13c).

[8] In both MSI and MSX the ISA transform is the final step. We think that having the $4n$ bytes available as working space throughout the algorithm could be exploited to improve runtimes, but this has not been explored.

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

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

Table 1: 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) | $\alpha$ | Description |
|---|---|---|---|---|---|
| E.coli | 17 | 2,815 | 4,638,690 | 4 | *Escherichia coli* genome |
| chr22.dna | 1,979 | 199,999 | 34,553,758 | 4 | Human chromosome 22 |
| bible | 14 | 551 | 4,047,392 | 63 | King James bible |
| world192 | 23 | 559 | 2,473,400 | 94 | CIA world fact book |
| sprot34 | 89 | 7,373 | 109,617,186 | 66 | SwissProt database |
| rfc | 93 | 3,445 | 116,421,901 | 120 | Concatenated IETF RFC files |
| howto | 267 | 70,720 | 39,422,105 | 197 | Linux Howto files |
| reuters | 282 | 26,597 | 114,711,151 | 93 | Reuters news in XML format |
| jdk13c | 679 | 37,334 | 69,728,899 | 113 | JDK 1.3 documentation |
| etext99 | 1,108 | 286,352 | 105,277,340 | 146 | Texts from Gutenberg project |
| w3c2 | 42,300 | 990,053 | 104,201,579 | 256 | Html files from W3C site |
| random | 4 | 9 | 20,000,000 | 26 | Bernoulli string |
| period 500K | 9,506,251 | 19,500,000 | 20,000,000 | 26 | Repetition of random string |
| period 1K | 9,999,001 | 19,999,000 | 20,000,000 | 26 | Repetition of random string |
| period 19 | 9,999,981 | 19,999,980 | 20,000,000 | 26 | Repetition of random string |
| period 20 | 9,999,981 | 19,999,980 | 20,000,000 | 26 | Repetition of random string |

To our knowledge, these are the first results on the affect alphabet encoding has on suffix sorting time. The variations in runtimes surely reflect the average depth at which induction sorting is being applied. We plan to further examine this curious property of MSufSort in future work. It will also be interesting to investigate how the permutations of [3], which enable better BWT compression, affect sort speed.

Of course the efficacy of permuting the alphabet very much depends on why one is suffix sorting in the first place. For BWT based compression it would only require sending the permutation to the decompressor, probably a small task. If the suffix array was being built for pattern matching, one would need to put query strings through an identical permutation before commencing search, again, probably a small task. Alphabet permutation may even accelerate some patter matching algorithms, in particular the approach of [1]. If specifically the suffix array of the original text was required, then reconstructing it from the one for the permuted text would be more difficult, possibly eclipsing any initial saving. We note though that the reconstruction can still be achieved in linear time [5].

## 11 Further Reducing Space Requirements

This section briefly outlines how to reduce space requirements for MSufSort when computing the BWT text is the desired result. The idea is based on the following observation:

**Observation 3.** *When there is a sequence of adjacent ranks in the* ISA*, only the leftmost one will ever be used for induction sorting.*

This suggests that if only the BWT text is of interest, space is being wasted in the ISA storing ranks which will never again be useful. The numbers in Table 6 show the peak of

Table 2: CPU Time for Real-world Data (seconds)

| | E.coli | chr22 | bible | world | sprot | rfc | howto | reuters | jdk13c | etext | w3c2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MS | 2 | 14 | 1 | 1 | 65 | 65 | 19 | 77 | 47 | 54 | 62 |
| MSI | 2 | 20 | 2 | 1 | 90 | 89 | 25 | 99 | 60 | 75 | 84 |
| MSX | 2 | 18 | 2 | 1 | 76 | 75 | 22 | 88 | 53 | 66 | 70 |
| SS | 2 | 25 | 2 | 1 | 99 | 93 | 22 | 133 | 64 | 92 | 84 |
| MF | 2 | 16 | 2 | 1 | 74 | 65 | 18 | 147 | 82 | 76 | 118 |
| LS | 4 | 35 | 3 | 2 | 144 | 154 | 40 | 183 | 105 | 146 | 192 |

Table 3: CPU Time for Synthetic Data (seconds)

| | period 500K | period 1K | period 20 | period 19 | random |
|---|---|---|---|---|---|
| MS | 14 | 15 | 3 | 3 | 8 |
| SS | 10 | 12 | 25 | 26 | 8 |
| MF | 573 | 834 | – | – | 8 |
| LS | 70 | 55 | 35 | 33 | 12 |

non-redundant ranks in the ISA for some of the files in the corpus, and shows only about 25% of ranks in the ISA are of useful for induction at any given time.

Eliminating use of the ISA has two main ramifications, namely:

1. Storing the rank for suffix $i$ at ISA[$i$] meant constant time access to ranks. Fast access to useful ranks should be preserved.

2. Chains were stored efficiently in the ISA, but now will need to be stored elsewhere.

To deal with the first issue, a hash table $T[0..t]$ could be used. The rank for suffix $i$ being hashed to position $i \bmod t$ in the table. If the numbers in Table 6 are indicative of the expected case, we could set $t$, the size of the $T$, to $n/4$ and expect only a few collisions. The hash table would require just $n$ bytes, with collisions stored in a separate list of (suffix,rank) pairs.

With the ISA no long available to store chains, they could be computed one at a time, and implemented as actual linked lists (ie. nodes of value, pointer pairs). Each element in a chain will now require $8n$ bytes, but because we only have one chain at a time of size approximately $n/\alpha^2$ elements, space requirement will remain small relative to $n$. Chain refinement will work the same way, and when a 2-chain has been fully processed, the next 2-chain is formed by traversing the input string again, at a slight cost to runtime.

The above scheme will not reduce memory usage in all cases, but will likely be very effective in the expected case, at the cost of some speed.

## 12 Conclusion

MSufSort takes a very different path to lightweight suffix sorting. For some applications, most notably when only the BWT text is required, it is clearly superior to alternative approaches, and uses similar space.

Table 4: Peak Memory Usage (Mbs)

| | E.coli | chr22 | bible | world | sprot | rfc | howto | reuters | jdk13c | etext | w3c2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MS | 32 | 205 | 29 | 13 | 547 | 599 | 197 | 572 | 357 | 542 | 525 |
| SS | 40 | 297 | 36 | 24 | 942 | 1006 | 368 | 988 | 604 | 915 | 958 |
| MF | 22 | 165 | 19 | 12 | 524 | 557 | 188 | 548 | 333 | 503 | 498 |
| LS | 35 | 264 | 31 | 19 | 836 | 888 | 301 | 875 | 532 | 803 | 795 |

Table 5: CPU Time in seconds for MSufSort on a subset of texts from the corpus. A code specifies the type of permutation as follows: *none* = no permutation, the original text; *rev* = alphabet reversed; *asc* = alphabet sorted to ascending order of frequency and recoded; *dsc* = alphabet sorted to descending order of frequency and recoded

| File | *none* | *rev* | *asc* | *dsc* |
|---|---|---|---|---|
| chr22.dna | 14 | 15 | 15 | 15 |
| sprot34.dat | 65 | 56 | 63 | 65 |
| howto | 19 | 17 | 22 | 22 |
| reuters | 77 | 71 | 75 | 73 |
| jdk13c | 48 | 42 | 43 | 43 |

There are several aspects of MSufSort which warrant further research, most notably the effects of alphabet permutations on sort time and the possibility of reducing memory consumption to below $5n$ bytes for some applications. Also of interest is the conversion of the ISA to the SA, a conceptually simple problem currently without a satisfactory solution.

## References


[1] M. I. Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Optimal exact string matching based on suffix arrays. In *SPIRE 2002*, number 2452 in Lecture Notes in Computer Science, pages 31–43. Springer-Verlag, Berlin, 2002.

[2] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 360–369, New Orleans, Louisiana, 1997.

[3] Brenton Chapin and Stephen R. Tate. Higher compression from the Burrows-Wheeler Transform by modified string sorting. In *Proceedings of the Data Compression Conference, DCC 98*, page 532. IEEE Computer Society Press, 1998.

[4] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 383–391, United States, 1996. ACM.

[5] Frantisek Franek and W. F. Smyth. Reconstructing a suffix array, 2005. Manuscript. Submitted for publication.

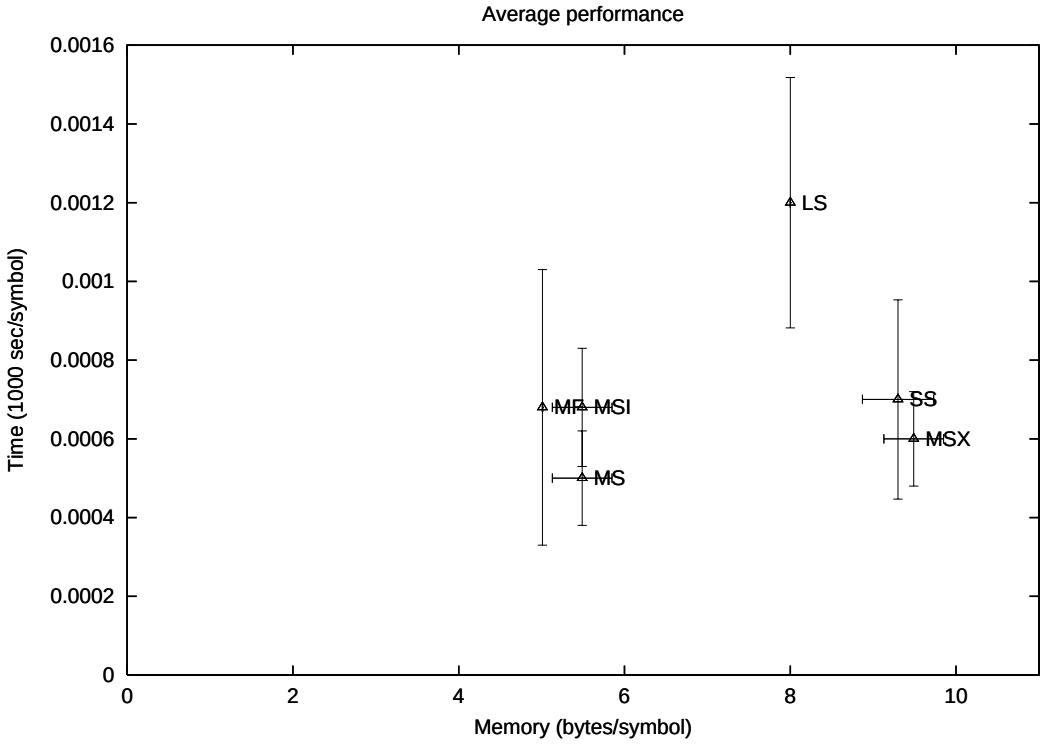


Figure 6: Resource requirements of the four algorithms averaged over the real-world test corpus. Error bars are one standard deviation.

Table 6: Peak percentage of values stored in the ISA which can be used for induction sorting

| E.coli | chr22 | bible | world | sprot | rfc | howto | reuters | jdk13c | etext | w3c2 |
|---|---|---|---|---|---|---|---|---|---|---|
| 24% | 22% | 26% | 20% | 20% | 18% | 21% | 21% | 17% | 25% | 17% |


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

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

[8] J. N. Larsson and K. Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214 [LUNFD6/(NFCS-3140)/1-20/(1999)], Department of Computer Science, Lund University, Sweden, 1999.

[9] U. Manber and G. W. Myers. Suffix arrays: a new model for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993.

[10] Michael Maniscalco. MSufSort, 2005. http://www.michael-maniscalco.com/msufsort.html.

[11] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40:33–50, 2004.

[12] David R. Musser. Introspective sorting and selection algorithms. *Software Practice and Experience*, 27(8):983–993, 1997.

[13] Simon J. Puglisi, Andrew H. Turpin, and William F. Smyth. The performance of linear time suffix sorting algorithms. In *Proceedings of the IEEE Data Compression Conference*, pages 358–368. IEEE Computer Society Press, March 2005.

[14] Klaus-Bernd Schürmann and Jens Stoye. An incomplex algorithm for fast suffix array construction. In *Proceedings of The Seventh Workshop on Algorithm Engineering and Experiments (ALENEX05)*. SIAM, January 2005. to appear.

[15] J. Seward. On the performance of BWT sorting algroithms. In *DCC: Data Compression Conference*, pages 173–182. IEEE Computer Society, 2000.

[16] W. F. Smyth. Repetitive perhaps, but certainly not boring. *Theoretical Computer Science*, 249(2):343–355, 2000.

[17] W. F. Smyth. *Computing Patterns in Strings*. Addison-Wesley-Pearson Education Limited, Essex, England, 2003.

```
int rank,*ISA;                          /*maintains chains and becomes the ISA*/
unsigned char* X;                       /*a pointer to the input string*/
stack<int> chainStack;

#define GET_SYM(i)     (X[(i)]<<8 | X[(i+1)])
#define IS_RANKED(i)   (ISA[i] < 0)
#define IS_END(i)      (ISA[i] == 0)
#define EMPTY          0xffffffff

void msufsort(unsigned char *x, int *isa, int n)
{
   X = x; ISA = isa;                    /*set global variables*/
   rank = 1;                            /*maintains next rank to assign*/
   formInitialChains();
   do{
      int h = chainStack.pop();         /*head of the next chain to process*/
      int l = chainStack.pop();         /*prefix length chained suffixes share*/
      if (ISA[h] == 0)                  /*if chain is a singleton it is sorted*/
         ISA[h] = -(rank++);            /*so assign a rank to its suffix*/
      else{                             /*refine the suffixes in the chain*/
         while (!IS_CHAIN_END(h)){      /*while not at the end of the chain*/
            if (IS_RANKED(h+l-1)){      /*Type A. we will later sort h*/
               noteSuffix(h,-ISA[h+l-1]); /*using rank of (h+l-1)*/
            } else if (IS_RANKED(h+l)){ /*Type B. we will later sort h*/
               noteSuffix(h,ISA[h+l]);  /*using rank of (h+l)*/
            } else{                     /*Type C.*/
               sym = GET_SYMBOL(h+l);   /*h will be put into a subchain*/
               updateSubChain(sym,h);   /*with other suffixes having same sym*/
            }
            h = ISA[h];                 /*move to next suffix in chain*/
         }
         pushSubChains();               /*put any new subchains on chainstack*/
         rankNotedSuffixes();           /*rank Type A and Type B suffixes*/
      }
   while(!chainstack.empty());
}
```

Figure 7: Function *msufsort*. Parameter $x$ points to the input string on $n$ characters and *isa* should point to an array of $n$ integers. Function *formInitialChains* creates chains of suffixes sharing a prefix of length 2 and pushes the heads in lexicographical order on *chainStack*. Functions *noteSuffix* and *rankNotedSuffixes* together implement the induction sorting heuristic and are shown in Figure 9. Functions *updateSubChain* and pushSubChains refine the current chain and are shown in 8. *ISA*, $X$ and *rank* are global variables in the program.

```
unsigned first[0..65536],last[0..65536];  /*used to hold chain heads and tails*/

static void updateSubChain(int suf, int sym)
{
   if(first[sym] == EMPTY){
      symbols[s++] = sym;
      first[sym] = last[sym] = suf;
   }else{
      ISA[last[sym]] = suf;
      last[sym] = suf;
   }
}

static void pushSubChains()
{
   if(s > 0){
      sort(symbols,s);
      for(i=s-1;i>=0;--i){
         int sym = symbols[i];
         chainstack.push(l+2);
         chainstack.push(first[sym]);
         ISA[last[sym]] = END;
         first[sym] = last[sym] = EMPTY;
      }
      s = 0;
   }
}

static void formInitialLists()
{
   for(int i=0; i<n; i++){          /*link suffixes with the same length 2 prefix*/
      unsigned sym = GET_SYMBOL(i); /*combine two chars*/
      updateSubChain(i,sym);
   }
   pushSubChains();
}
```

Figure 8: Functions *updateSubChain* and *pushSubChains*. Function *updateSubChain* places suffix $h$ in a chain according to parameter *sym*. If no chain yet exists for *sym* then one is created and $h$ is placed in it, arrays *first* and *last* maintains the first and last suffixes respectively for each *sym*. Function *pushSubChains* sorts subchains create with previous calls to *updateSubChain* and places them on *chainStack* for further refinement, zeroing out *first* and *last* in the process. Variables *ISA*, *first*, *last*, *symbols*, *s* and *chainStack* are global to these functions.

```
typedef struct indrec{
   int suff;
   int key;
}

stack<indrec> indStack;

static void noteSuffix(int suf, int key)
{
   indrec sortrec(suf, key);
   indstack.push(sortrec);
}

static void rankNotedSuffixes()
{
   if(indstack.size() > 1)
      sort(indstack,indstack.size());
   for(i=0;i < indstack.size();i++)
      ISA[indstack[i].suf] = -(rank++);

}
```

Figure 9: Functions *noteSuffix* and *rankNotedSuffix*. These implement the induction sorting heuristic described in section 4. Function *noteSuffix* records suffix position *suf*, previously deemed to be sortable by induction, on *indstack* along *key*. Function *rankNotedSuffixes*, called later, sorts any noted suffixes and assigns their *rank* in the *ISA*. *ISA*, *rank* and *indstack* are global to these functions.

```
void bucketSort(unsigned char *x, int *isa)
{
   X = x; ISA = isa;                      /*set global variables*/
   rank = 1;                              /*global, maintains the next rank to assign*/

   formInitialChains();
   do{
      int h = chainStack.pop();            /*head of the next chain to process*/
      int l = chainStack.pop();            /*prefix length chained suffixes share*/
      if (ISA[h] == 0)                     /*if chain is a singleton it is sorted*/
         ISA[h] = -(rank++);               /*so assign a rank to its suffix*/
      else{                                /*refine the suffixes in the chain*/
         while (!IS_CHAIN_END(h)){         /*while not at the end of the chain*/
               sym = GET_SYMBOL(h+l);      /*h will be put into a subchain*/
               updateSubChain(sym,h);      /*with other suffixes having same sym*/
            }
            h = ISA[h];                    /*move to next suffix in chain*/
         }
         pushSubChains();                  /*put any new subchains on chainstack*/
         rankNotedSuffixes();              /*rank Type A and Type B suffixes*/
      }
   while(!chainstack.empty());
}
```

Figure 10: Function *bucketSort*, a direct comparison suffix sorter. The function first links together suffix beginning with the same first two characters, and subsequently refines the lists until the lexicographic rank for all the suffixes is known. $X$, $ISA$, *rank* and *listStack* are global variables in the program.