The following observation is worth mentioning. If in phase 2, the sequence of suffixes $[S_i \mid 1 \leq i \leq n, i \bmod 3 = 1]$ would also be sorted recursively by the skew algorithm, then this would yield the recurrence

$$T(n) = T(n/3) + T(2n/3) + \Theta(n)$$

According to the master theorem, this recurrence has the solution $T(n) = \Theta(n \log n)$. In other words, the resulting algorithm would not be linear!

## 4.1.2 Induced sorting

In this section, we will explain another interesting linear-time SACA devised by Nong et al. [244]. It shares two key features with the skew algorithm: it is a recursive algorithm and it uses the method of induced sorting. We speak of "induced sorting" whenever a complete sort of a selected subset of suffixes can be used to "induce" a complete sort of other subsets of suffixes. In the skew algorithm, for example, a complete sort of the suffixes $\{S_i \mid 1 \leq i \leq n, i \bmod 3 \neq 1\}$ can be used to induce a complete sort of the suffixes $\{S_i \mid 1 \leq i \leq n, i \bmod 3 = 1\}$.

The induced sorting algorithm[2] of Nong et al. [244] heavily depends on the work of Ko and Aluru [184] (the reader can find the description of several precursor algorithms in [262]). Ko and Aluru classified suffixes into S-type and L-type suffixes and showed that a complete sort of the S-type suffixes can be used to induce a complete sort of the L-type suffixes (or vice versa). The contribution of Nong et al. is the insight that it is enough to sort the usually small set of LMS-substrings and to use it to induce the order of all suffixes. Moreover, the lexicographic order of the LMS-substrings is determined by the same principle, using recursion if needed.

In the rest of this section, let $S$ be a string of length $n+1$ that is terminated by the sentinel $\$$. The suffixes of $S$ are classified into two types: Suffix $S_i$ is S-type if $S_i < S_{i+1}$ and it is L-type if $S_i > S_{i+1}$. The last suffix, the sentinel $\$$, is S-type. We use a bit array $T[1..n+1]$ to store the types of the suffixes: $T[i] = 0$ means that suffix $S_i$ is L-type and $T[i] = 1$ means that it is S-type. For better readability, we write $T[i] =$ L instead of $T[i] = 0$ $T[i] =$ S instead of $T[i] = 1$.

**Lemma 4.1.7** *All suffixes can be classified in $O(n)$ time by a right-to-left scan of $S$.*

*Proof* It is readily verified that $S_i$ is S-type if (a) $S[i] < S[i+1]$ or (b) $S[i] = S[i+1]$ and $S_{i+1}$ is S-type. Analogously, $S_i$ is L-type if (a) $S[i] > S[i+1]$ or

[2]Nong et al. speak of "almost pure induced-sorting" but we will use the shorter term "induced sorting."

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S | i | m | i | m | m | m | i | s | i | s | m | i | s | i | s | s | i | i | p | i | $ |
| type | S | L | S | L | L | L | S | L | S | L | L | S | L | S | L | L | S | S | L | L | S |
| LMS | | | * | | | | * | | * | | | * | | * | | | * | | | | * |

Figure 4.6: The type classification of the suffixes of the string $S$ proceeds from right to left. It starts with the S-type suffix $S_{21} = \$$. Suffixes $S_{19}$ and $S_{20}$ are L-type because $p > i > \$$, whereas suffix $S_{18}$ is S-type because $i < p$. Both suffixes $S_{17}$ and $S_{18}$ start with the same character $i$, so $S_{17}$ is S-type because $S_{18}$ is S-type. The same argument applies to the suffixes $S_{15}$ and $S_{16}$: $S_{15}$ is L-type because $S_{16}$ is L-type. The LMS-positions of $S$ are marked with an asterisk. For example, $3$ is an LMS-position because suffix $S_3$ is S-type and the preceding suffix $S_2$ is L-type.

(b) $S[i] = S[i+1]$ and $S_{i+1}$ is L-type. Therefore, starting with the S-type sentinel $\$$, in a right-to-left scan of $S$, we can determine the types of all of its suffixes. □

Figure 4.6 shows an example of type classification.

**Lemma 4.1.8** *An S-type suffix is lexicographically greater than any L-type suffix starting with the same first character.*

*Proof* For an indirect proof, suppose that there is an S-type suffix $S_i$ and an L-type suffix $S_j$ so that $S_i[1] = S[i] = a = S[j] = S_j[1]$ and $S_i < S_j$. We can write $S_i = aubv$ and $S_j = aucw$, where $b \neq c$ are characters and $u$, $v$, and $w$ are (possibly empty) strings.

1. Suppose that $u$ consists solely of $a$'s. Because $S_i$ is S-type, it follows that $a \leq b$. Similarly, since $S_j$ is L-type, it follows that $a \geq c$. The combination of these two facts yields $c \leq b$. However, $S_i < S_j$ implies $b < c$, a contradiction.

2. Otherwise, $u$ contains a character other than $a$. Let $d$ be the leftmost character in $u$ that is different from $a$. Because $S_i$ is S-type, it follows that $a < d$. Similarly, since $S_j$ is L-type, it follows that $d > a$. This contradiction proves the lemma.

□

**Corollary 4.1.9** *In the suffix array of $S$, among all suffixes that begin with the same character, the L-type suffixes appear before the S-type suffixes.*

*Proof* Direct consequence of Lemma 4.1.8 □

We employ an array $C$ of size $\sigma$ to divide the suffix array of $S$ into buckets (without loss of generality, we assume that all characters from the ordered alphabet $\Sigma$ appear in the string $S$). For every $c \in \Sigma$, we define $C[c] = \sum_{b\in\Sigma,b<c} cnt[b]$, where $cnt[b]$ is the number of occurrences of character $b$ in $S$. In other words, if we consider all characters in $\Sigma$ that are smaller than $c$, then $C[c]$ is the overall number of their occurrences in $S$. The $c$-interval $[i..j]$ can be determined by $i = C[c]+1$ and $j = C[c+1]$ (where $c+1$ is the character that follows $c$ in the alphabet $\Sigma$). In the following, we call the $c$-interval the $c$-bucket. Every $c$-bucket $[i..j]$ can further be divided into the interval $[i..k]$ containing all L-type suffixes starting with character $c$ and the interval $[k+1..j]$ containing all S-type suffixes starting with character $c$. The interval $[i..k]$ is called the L-type region and $[k+1..j]$ is called the S-type region of the $c$-bucket ($k = i-1$ or $k = j$ is possible, i.e., the S-type region or the L-type region of the bucket may be empty).

**Definition 4.1.10** A position $i$, $1 < i \leq n+1$, in $S$ with $T[i-1] = \text{L}$ and $T[i] = \text{S}$ (i.e., suffix $S_{i-1}$ is L-type and suffix $S_i$ is S-type) is called *LMS-position* (leftmost S-type position); see Figure 4.6.

Now we are in a position to formulate the induced sorting algorithm.

Phase 0: Compute the type array $T$ by a right-to-left scan of $S$.

Phase I: Compute all LMS-positions in $S$ and sort the corresponding suffixes in ascending lexicographic order (we will elaborate on this phase later).

Phase II:

1. Scan the sorted sequence of LMS-positions from right to left. For each position encountered in the scan,[3] move it to the current end of its bucket in $A$ (initially, the end of a $c$-bucket is the index $C[c+1]$), and shift the current end of the bucket by one position to the left.

2. Scan the array $A$ from left to right. For each entry $A[i]$ encountered in the scan, if $S_{A[i]-1}$ is an L-type suffix, move its start position $A[i]-1$ in $S$ to the current front of its bucket in $A$ (initially, the front of a $c$-bucket is the index $C[c]+1$), and shift the current front of the bucket by one position to the right.

3. Scan the array $A$ from right to left. For each entry $A[i]$ encountered in the scan, if $S_{A[i]-1}$ is an S-type suffix, move its start position $A[i]-1$ in

[3]Each undefined entry $\perp$ is ignored because it does not correspond to a suffix.

$S$ to the current end of its bucket in $A$ (initially, the end of a $c$-bucket is the index $C[c+1]$), and shift the current end of the bucket by one position to the left.

An illustration of phase II can be found in Figure 4.7. It should be stressed that in step 3, the suffixes starting at LMS-positions (these are of S-type) are already in the array $A$. This does no harm because each of these suffixes will be overwritten before it is reached in the right-to-left scan of $A$; see Lemma 4.1.12.

**Lemma 4.1.11** *Step 2 of phase II correctly sorts all L-type suffixes of $S$.*

*Proof* First, we show that every L-type suffix is placed at its correct position. We proceed by induction on the number $q$ of placed L-type suffixes. Clearly, $A[1] = n+1$ because \$ appears at position $n+1$ in $S$, and \$ is the lexicographically smallest character. Furthermore, $S_{A[i]-1} = S_n = c\$$ is an L-type suffix because $S[n] = c > \$ = S[n+1]$. Consequently, position $n$ is moved to the front of the $c$-bucket. This is certainly correct because the suffix $S_n$ is the lexicographically smallest suffix starting with character $c$. As an inductive hypothesis, suppose that $q$ L-type suffixes have been placed correctly. We have to show that the $(q+1)$-th L-type suffix will be placed correctly. Suppose that in the left-to-right scan of the array $A$ we are at index $i > 1$ with $A[i] \neq \perp$, and let $A[i] = j+1$ for some $j \geq 1$. That is, $S_{j+1}$ is either an S-type suffix starting at an LMS-position or an L-type suffix that has already been placed, and suffix $S_j$ is the $(q+1)$-th L-type suffix that has to be placed. Let $c = S[j]$. For a proof by contradiction, suppose that when we move the start position $j$ to the current front of the $c$-bucket in $A$, there is already an L-type suffix $S_k$ in the $c$-bucket that is lexicographically greater than $S_j$. So in the $c$-bucket, $k$ is left to $j$. This means that there must be an index $i' < i$ so that $A[i'] = k+1$. In other words, $k+1$ precedes $j+1$ in the array $A$. Because both $S_j$ and $S_k$ are in the $c$-bucket, we have $S_j = cS_{j+1}$ and $S_k = cS_{k+1}$. In conjunction with $S_j < S_k$, this has $S_{j+1} < S_{k+1}$ as a consequence. Note that $S_{j+1}$ is either an S-type suffix starting at an LMS-position or an L-type suffix, and the same is true for $S_{k+1}$. According to the inductive hypothesis, S-type suffixes starting at LMS-positions and the first $q$ placed L-type suffixes are in the correct order. Thus, $j+1$ must precede $k+1$ in the array $A$. This, however, contradicts our previous conclusion that $k+1$ precedes $j+1$ in $A$. We conclude that the $(q+1)$-th L-type suffix $S_j$ is placed correctly.

Second, we prove that every L-type suffix will actually be placed during the scan. For an inductive proof, assume that the $q$ lexicographically smallest L-type suffixes have been placed. Let $S_j$ be the $(q+1)$-th lexicographically smallest L-type suffix. We have to show that $S_j$ will be placed during the scan. Clearly, $S_{j+1} < S_j$ because $S_j$ is L-type (thus, if $S_{j+1}$ appears in the array $A$, then it must appear left to $S_j$). Moreover, $S_{j+1}$ is

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S | i | m | i | m | m | m | i | s | i | s | m | i | s | i | s | s | i | i | p | i | $ |
| type | S | L | S | L | L | L | S | L | S | L | L | S | L | S | L | L | S | S | L | L | S |
| | | | * | | | | * | | * | | | * | | * | | | * | | | | * |

| | $ | i | | | | | | | | | m | | | | | p | s | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LMS | 21 | | | | 17 | 3 | 7 | 12 | 9 | 14 | | | | | | | | | | | |
| L-type suffixes | 21 | 20 | | | 17 | 3 | 7 | 12 | 9 | 14 | 2 | 6 | 11 | 5 | 4 | 19 | 16 | 8 | 13 | 10 | 15 |
| S-type suffixes | 21 | 20 | 17 | 1 | 17 | 3 | 7 | 12 | 9 | 14 | 2 | 6 | 11 | 5 | 4 | 19 | 16 | 8 | 13 | 10 | 15 |
| | | | | | 3 | 18 | 7 | 12 | 9 | 14 | | | | | | | | | | | |

Figure 4.7: The upper part shows the type classification of the suffixes of string $S$. The lower part illustrates phase II of the induced sorting algorithm. In phase I, all suffixes that start at LMS-positions have been sorted. In step 1 of phase II, the sorted sequence delivered by phase I is scanned from right to left and the suffixes are placed—also from right to left—into their buckets. After step 1, all suffixes that start at LMS-positions appear in ascending lexicographic order in the S-type regions of their buckets as shown in row "LMS." In step 2 of phase II, the array is scanned from left to right and L-type suffixes are placed—also from left to right—into their buckets. After step 2, all L-type suffixes appear in ascending lexicographic order in the L-type regions of their buckets as shown in row "L-type suffices." In step 3 of phase II, the array is scanned again from right to left and the S-type suffixes are placed—again from right to left—into their buckets. Step 3 overwrites S-type suffixes that start at LMS-positions before they are reached in the scan. By contrast, all L-type suffixes are unaffected since the L-type and S-type regions of the buckets are disjoint. After step 3, all S-type suffixes appear in ascending lexicographic order in the S-type regions of their buckets as shown in row "S-type suffices." It follows as a consequence that all suffixes are sorted lexicographically after step 3.

either a suffix starting at an LMS-position or an L-type suffix. By the induction hypothesis, $S_{j+1}$ has been placed. Consequently, when the scan reaches $S_{j+1}$, the L-type suffix $S_j$ is placed. □

**Lemma 4.1.12** *Step 3 of phase II correctly sorts all S-type suffixes of $S$.*

*Proof* The proof is very similar to the proof of the preceding lemma. First, we show that every S-type suffix is placed at its correct position. We proceed by induction on the number $q$ of placed S-type suffixes. For the base case, note that all suffixes starting with the largest character $c_\sigma$ must be L-type. That is, the S-type region of the $c_\sigma$-bucket is empty, and hence $A[n+1]$ corresponds to an L-type suffix. In the right-to-left scan of $A$, let $i$ be the first (rightmost) index so that $S_{A[i]-1} = cS_{A[i]}$ is an S-type suffix (clearly, $c < c_\sigma$ and it is not difficult to show that $i$ belongs to the $c_\sigma$-bucket). The algorithm places the index $A[i] - 1$ at the very end of the $c$-bucket. This is correct because $S_{A[i]-1} = cS_{A[i]}$ is the lexicographically largest suffix that starts with the character $c$. As an inductive hypothesis, suppose that $q$ S-type suffixes have been placed correctly. We have to show that the $(q+1)$-th S-type suffix will be placed correctly. Suppose that in the right-to-left scan of the array $A$ we are at index $i$, and let $A[i] = j + 1$ for some $j \geq 1$. That is, $S_{j+1}$ is either an L-type suffix or an S-type suffix that has already been placed, and suffix $S_j$ is the $(q+1)$-th S-type suffix that has to be placed. Let $c = S[j]$. For a proof by contradiction, suppose that when we move the start position $j$ to the current end of the $c$-bucket in $A$, there is already an S-type suffix $S_k$ in the $c$-bucket that is lexicographically smaller than $S_j$. So in the $c$-bucket, $k$ is right to $j$. This means that there must be an index $i' > i$ so that $A[i'] = k + 1$. In other words, in the right-to-left scan of $A$, $k + 1$ is encountered before $j + 1$. Because both $S_j$ and $S_k$ are in the $c$-bucket, we have $S_j = cS_{j+1}$ and $S_k = cS_{k+1}$. Moreover, $S_k < S_j$ has $S_{k+1} < S_{j+1}$ as a consequence. If $S_{j+1}$ ($S_{k+1}$) is an L-type suffix, then it is at its correct position by Lemma 4.1.11. Otherwise, if $S_{j+1}$ ($S_{k+1}$) is an S-type suffix, then it is at its correct position by the inductive hypothesis. Therefore, in the right-to-left scan of $A$, $j + 1$ must be encountered before $k + 1$, a contradiction to our assumption. We conclude that the $(q+1)$-th S-type suffix $S_j$ is placed correctly.

Second, we prove that every S-type suffix will actually be placed during the scan. For an inductive proof, assume that the $q$ lexicographically largest S-type suffixes have been placed, and let $S_j$ be the $(q+1)$-th lexicographically largest S-type suffix. We have to show that $S_j$ will be placed during the scan. Clearly, $S_j < S_{j+1}$ because $S_j$ is S-type (thus, if $S_{j+1}$ appears in the array $A$, then it must appear right to $S_j$). If $S_{j+1}$ is an L-type suffix, then it was placed correctly in step 2. Otherwise, if $S_{j+1}$ is an S-type suffix, then it has also been placed by the induction hypothesis. In both cases, $S_{j+1}$ was encountered before and $S_j$ is placed. □

The following notions are used in the formulation of phase I of the induced sorting algorithm.

**Definition 4.1.13** A substring of $S$ that starts at an LMS-position and ends at the next LMS-position is called an *LMS-substring*. By definition, the sentinel is also an LMS-substring. Any suffix of an LMS-substring is called an *LMS-suffix*.

Now let us elaborate on phase I, which is illustrated in Figure 4.8. Note that steps 2 and 3 are verbatim the same as in phase II.

Phase I:

1. Scan the array $T$ from left to right and place each LMS-position into its bucket and into an array $P$ (so if there are $m$ LMS-positions, then $P$ has size $m$). To be precise, for each position $j$ encountered in the scan, if $j$ is an LMS-position, move it to the current end of its bucket in $A$ (initially, the end of a $c$-bucket is the index $C[c+1]$), and shift the current end of the bucket by one position to the left. Furthermore, move $j$ to the current front of array $P$ (initially, the front of $P$ is the index $1$), and shift the current front of $P$ by one position to the right.

   In this step, each LMS-position $j$ represents the last character $S[j]$ of the LMS-substring that ends at $j$ (and not suffix $S_j$ as in phase II).

2. Scan the array $A$ from left to right. For each entry $A[i]$ encountered in the scan, if $S_{A[i]-1}$ is an L-type suffix, move its start position $A[i]-1$ in $S$ to the current front of its bucket in $A$ (initially, the front of a $c$-bucket is the index $C[c]+1$), and shift the current front of the bucket by one position to the right.

   In this step, a position $j$ with $T[j] = \text{L}$ represents the LMS-suffix that starts at position $j$ and ends at the next LMS-position (and not the L-type suffix $S_j$ as in phase II).

3. Scan the array $A$ from right to left. For each entry $A[i]$ encountered in the scan, if $S_{A[i]-1}$ is an S-type suffix, move its start position $A[i]-1$ in $S$ to the current end of its bucket in $A$ (initially, the end of a $c$-bucket is the index $C[c+1]$), and shift the current end of the bucket by one position to the left.

   In this step, a position $j$ with $T[j] = \text{S}$ represents the LMS-suffix that starts at position $j$ and ends at the next LMS-position (and not the S-type suffix $S_j$ as in phase II). Thus, if $j$ is an LMS-position, then it represents the LMS-substring that starts at position $j$ and ends at the next LMS-position.

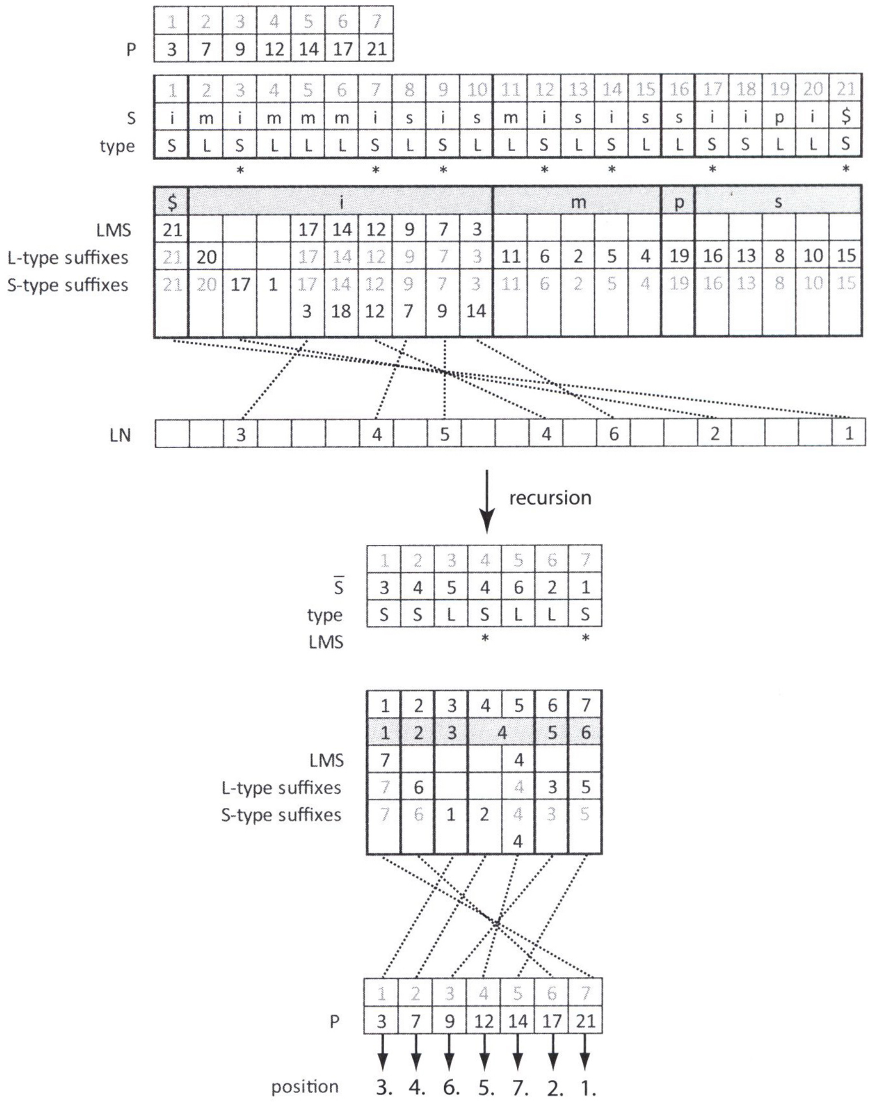

Figure 4.8: Phase I of the induced sorting algorithm.

4. Initialize an array $\mathsf{LN}[1..n+1]$ that will contain the new lexicographic names of the LMS-substrings, and initialize the counter $\bar{i}$ for new lexicographic names to 1. Because the sentinel is the lexicographically smallest LMS-substring, it gets the smallest new lexicographic name, i.e., $\mathsf{LN}[n+1] = 1$. Furthermore, initialize $prev = n+1$. Now, starting at index $i = 2$, scan the array $A$ from left to right. Whenever an entry $A[i] = j$ is encountered so that $j$ is the start position of an LMS-substring do the following:
   - Compare the LMS-substring starting at position $prev$ in $S$ with the LMS-substring starting at position $j$ in $S$ (character by character).
   - If they are different, increment $\bar{i}$ by one.
   - Set $\mathsf{LN}[j] = \bar{i}$ and $prev = j$.
5. Scan the array $P[1..m]$ from left to right and for each $k$ with $1 \le k \le m$ set $\overline{S}[k] = \mathsf{LN}[P[k]]$.
6. If $\bar{i} = m$, then the LMS-substrings are pairwise different. Compute the suffix array $\overline{\mathsf{SA}}$ of the string $\overline{S}$ directly by $\overline{\mathsf{SA}}[\overline{S}[k]] = k$.
7. Otherwise, *recursively* compute the suffix array $\overline{\mathsf{SA}}$ of the string $\overline{S}$.
8. The ascending lexicographic order of all the suffixes of $S$ that start at an LMS-position is $P[\overline{\mathsf{SA}}[1]], P[\overline{\mathsf{SA}}[2]], \ldots, P[\overline{\mathsf{SA}}[m]]$.

**Lemma 4.1.14** *After steps 1-3 of phase I, LMS-substrings appear in lexicographic order in $A$.*

*Proof* As already mentioned, in phase I an LMS-position $j$ corresponds to the last character $S[j]$ of the LMS-substring ending at position $j$. By contrast, in phase II an LMS-position $j$ corresponds to the suffix $S_j$. After step 1 of phase I, each LMS-position is in the S-type region of its bucket. That is, the length 1 suffixes (last characters) of all LMS-substrings are in the correct order in $A$. Now the proofs of Lemmata 4.1.11 and 4.1.12 apply with a grain of salt, and we conclude that after steps 1-3 all LMS-suffixes of length greater than 1 are in the correct order in $A$.[4] Of course, the sentinel (more precisely, its position $n+1$) is still the first entry of $A$. Therefore, all LMS-substrings appear in lexicographic order in $A$ (note that lexicographically adjacent LMS-substrings may be identical). □

By Lemma 4.1.14, LMS-substrings (represented by their start positions) appear in lexicographic order in the array $A$. Thus, there are indices

[4]In step 1, an LMS-position $j$ represents the length 1 LMS-suffix $S[j]$. In step 3, however, $j$ represents the LMS-substring starting at position $j$. So length 1 LMS-suffixes are no longer represented.

$1 \le i_1 < \cdots < i_m \le n+1$ and positions $j_1, \ldots, j_m$ so that $A[i_1] = j_1, \ldots, A[i_m] = j_m$ (hence $j_1, \ldots, j_m$ is a permutation of $P[1], \ldots, P[m]$). Step 4 renames all LMS-substrings according to their lexicographic order in the array $A$, where identical LMS-substrings get the same new name. To be precise, in step 4 we compare the current LMS-substring, say $S[j_k..j_{k+1}]$, with the previous LMS-substring $S[j_{k-1}..j_k]$. Suppose that the previous LMS-substring $S[j_{k-1}..j_k]$ has got the new lexicographic name $\bar{i}$. Now, if $S[j_k..j_{k+1}] = S[j_{k-1}..j_k]$, then $S[j_k..j_{k+1}]$ gets the same lexicographic name $\bar{i}$. Otherwise, $S[j_k..j_{k+1}]$ gets the new lexicographic name $\bar{i}+1$. In both cases, the lexicographic name of $S[j_k..j_{k+1}]$ is stored at position $j_k$ in array $\mathsf{LN}$; see Figure 4.8. The new string $\overline{S}$ is obtained by setting $\overline{S}[k] = \mathsf{LN}[P[k]]$ in step 5. Now, there are two possibilities (step 6 or step 7). Either all LMS-substrings are pairwise different (which is equivalent to $\bar{i} = m$ after step 4) or there are at least two identical LMS-substrings. In the former case (step 6), the inverse suffix array $\overline{\mathsf{ISA}}$ of the string $\overline{S}$ coincides with $\overline{S}$ (viewed as an array). In the latter case (step 7), we recursively apply the whole induced sorting algorithm to the string $\overline{S}$ to get its suffix array $\overline{\mathsf{SA}}$; see Figure 4.8. So after step 6 or step 7, we know the lexicographic order of all suffixes of $\overline{S}$. Lemma 4.1.15 proves that step 8 yields the lexicographic order of all the suffixes of $S$ that start at an LMS-position (by means of the suffix array $\overline{\mathsf{SA}}$ and the $P$ array).

**Lemma 4.1.15** *We have $\overline{S}_i < \overline{S}_j$ if and only if $S_{P[i]} < S_{P[j]}$.*

*Proof* Let $u_i$ be the string obtained by replacing every lexicographic name in $\overline{S}_i$ with the LMS-substring that is represented by this name. Furthermore, let $v_i$ be the string obtained from $S_{P[i]}$ by doubling every character at an LMS-position. It is readily verified that $u_i = v_i$. Moreover, $u_i < u_j$ (where $u_j$ is defined analogously) if and only if $S_{P[i]} < S_{P[j]}$. Hence the lemma follows. □

All in all, the induced sorting algorithm correctly computes the suffix array. It remains to analyze its worst-case time-complexity. It is not difficult to see that each step in phases I and II takes at most $O(n)$ time. By definition 4.1.10, position 1 is not an LMS-position and there must be at least one position in between two consecutive LMS-positions. Hence $|\overline{S}| = m \le \lfloor \frac{n+1}{2} \rfloor$. It follows from the master theorem that the whole induced sorting algorithm has a worst-case time complexity of $O(n)$.

Implementation details of the induced sorting algorithm:

1. It is not difficult to see that if two LMS-substrings are identical, then so are their type sequences. Consequently, in step 4 of phase I, if one compares characters and types of LMS-substrings simultaneously, then the comparison can be stopped when there is a character

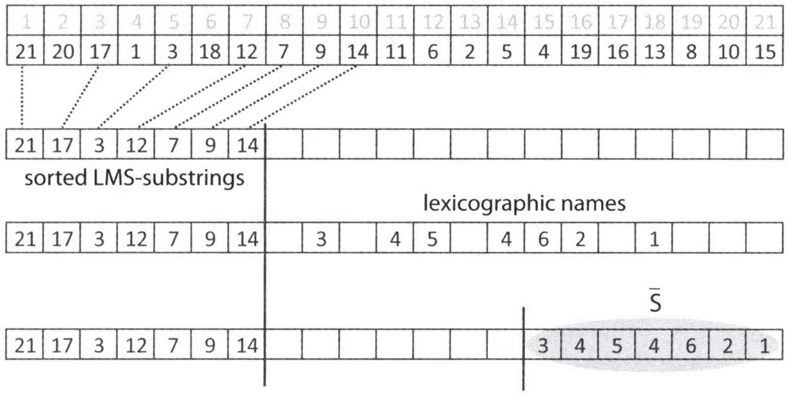

Figure 4.9: Apart from the string $S$, one array suffices in an implementation of the induced sorting algorithm.

mismatch or a type mismatch (this is done in the original algorithm by Nong et al. [244]).

2. Since $m \leq \lfloor \frac{n+1}{2} \rfloor$, the sorted LMS-substrings can be stored in the left half of the array $A$ and the array LN (later the string $\overline{S}$ and finally the array $\overline{\mathsf{SA}}$) can be stored in the right half of $A$; see Figure 4.9. As a matter of fact, it is possible to remove the type array $T$ in the initial call of the induced sorting algorithm and to integrate the type arrays in recursive calls into the array $A$. Moreover, the pointers to the (current) front/end of the buckets are solely required in the initial call but not in recursive calls. In summary, the induced sorting algorithm can be implemented in such a way that it keeps only the string $S$ (the input), the array $A$ (the output), and the bucket pointers of the initial call (for a constant-size alphabet, these pointers take only constant space) in main memory; see [243] for details.

**Exercise 4.1.16** Suppose that the word boundaries in a natural language text have already been determined by a parser. These word boundaries divide the text into tokens. As an example, consider the English sentence "This is a text." and the corresponding sequence of tokens "This,is,a,text". In this exercise, we assume that a text $T$ is given as the concatenation of the tokens, in which tokens are separated by a special separator symbol $\#$. We assume that $T$ consists of $k$ tokens and $n$ characters (including all occurrences of the separator symbol). In our example, $T = This\#is\#a\#text$ consists of $4$ tokens and $14$ characters. Let $P$ be the set of positions at which a token starts, i.e., $P = \{1\} \cup \{i \mid 2 \leq i \leq n \text{ and } T[i-1] = \#\}$. The set of all suffixes of $T$ starting with a full token is defined by $\{T_p \mid p \in P\}$, and the *word suffix array* WSA is a permutation of $P$ specifying the lexicographic order of the $k$ suffixes from $\{T_p \mid p \in P\}$, i.e., it satisfies $T_{\mathsf{WSA}[1]} < T_{\mathsf{WSA}[2]} < \cdots < T_{\mathsf{WSA}[k]}$. Devise an $O(n)$ time and $O(k)$ space algorithm to construct the word suffix array of $T$.

**Exercise 4.1.17** A *cyclic string* of length $n$ is a string $S$ in which the character at position $n$ is considered to precede the character at position 1. The *cyclic string linearization problem* is the following: Choose a position to cut $S$ so that the resulting linear string is the lexicographically smallest of all the $n$ possible linear strings created by cutting $S$. Give an algorithm that solves this problem in $O(n)$ time.

## 4.2 The LCP-array

Throughout this book, $\mathsf{lcp}(u, v)$ denotes the longest common prefix between two strings $u$ and $v$, whereas $\mathsf{lcs}(u, v)$ denotes the longest common suffix of $u$ and $v$.

The suffix array is often augmented with the so-called LCP-array (or LCP-table), containing the lengths of the longest common prefixes between consecutive suffixes in SA. The formal definition reads as follows.

**Definition 4.2.1** The LCP*-array* is an array of size $n + 1$ with boundary elements $\mathsf{LCP}[1] = -1$ and $\mathsf{LCP}[n + 1] = -1$, and for all $i$ with $2 \leq i \leq n$ we have $\mathsf{LCP}[i] = |\mathsf{lcp}(S_{\mathsf{SA}[i-1]}, S_{\mathsf{SA}[i]})|$.

Figure 4.10 shows the LCP-array of the string $S = ctaataatg$. The LCP-array first appeared in the seminal paper of Manber and Myers [214] on suffix arrays (where it was called $Hgt$ array).

A suffix array enhanced with the corresponding LCP-array will henceforth be called an *enhanced suffix array*. More generally, the generic name *enhanced suffix array* (and the acronym ESA) stands for data structures consisting of the suffix array enhanced with additional arrays.

### 4.2.1 Linear-time construction

It is possible to modify some SACAs so that they compute the LCP-array as a by-product of the suffix array construction. This has been shown in [175] for the skew algorithm presented in Section 4.1.1 and in [107] for the induced sorting algorithm presented in Section 4.1.2, but other SACAs could probably also be modified to produce the LCP-array.

Another approach is to construct the LCP-array from an already constructed suffix array. To date, several different LCP-array construction algorithms (LACAs) of this kind are known [33, 126, 174, 177, 216, 264], but to review all of them goes beyond the scope of this book. In this