Algorithmica (2011) 60: 806–818
DOI 10.1007/s00453-009-9369-1

Linear Time Algorithms for Generalizations of the Longest Common Substring Problem

Michael Arnold · Enno Ohlebusch

Received: 8 August 2008 / Accepted: 15 October 2009 / Published online: 27 October 2009
© Springer Science+Business Media, LLC 2009

Abstract In its simplest form, the *longest common substring problem* is to find a longest substring common to two or multiple strings. Using (generalized) suffix trees, this problem can be solved in linear time and space. A first generalization is the *k-common substring problem*: Given m strings of total length n, for all k with $2 \leq k \leq m$ simultaneously find a longest substring common to at least k of the strings. It is known that the k-common substring problem can also be solved in $O(n)$ time (Hui in Proc. 3rd Annual Symposium on Combinatorial Pattern Matching, volume 644 of Lecture Notes in Computer Science, pp. 230–243, Springer, Berlin, 1992). A further generalization is the *k-common repeated substring problem*: Given m strings $T^{(1)}, T^{(2)}, \ldots, T^{(m)}$ of total length n and m positive integers $x_1, \ldots, x_m$, for all k with $1 \leq k \leq m$ simultaneously find a longest string ω for which there are at least k strings $T^{(i_1)}, T^{(i_2)}, \ldots, T^{(i_k)}$ ($1 \leq i_1 < i_2 < \cdots < i_k \leq m$) such that ω occurs at least x_{i_j} times in $T^{(i_j)}$ for each j with $1 \leq j \leq k$. (For $x_1 = \cdots = x_m = 1$, we have the k-common substring problem.) In this paper, we present the first $O(n)$ time algorithm for the k-common repeated substring problem. Our solution is based on a new linear time algorithm for the k-common substring problem.

Keywords Suffix arrays · Longest common substring · Longest common repeat · String mining · Repeat analysis

M. Arnold
Capgemini sd&m AG, Löffelstraße 46, 70597 Stuttgart, Germany
e-mail: michael.arnold@capgemini-sdm.com

E. Ohlebusch (✉)
Institute of Theoretical Computer Science, University of Ulm, 89069 Ulm, Germany
e-mail: enno.ohlebusch@uni-ulm.de

1 Introduction

The longest common substring problem is a classic problem in string analysis. In 1970 the famous computer scientist Donald E. Knuth conjectured that a linear time algorithm for this problem would be impossible [2–4]. To the retrieval of his honor, it should be stressed that the first linear time construction algorithm of suffix trees [5] became known years after he made the conjecture. In fact, given that knowledge of suffix trees, a solution to the longest common substring problem is almost trivial; see e.g. [4, Sect. 7.4].

Given m strings $T^{(1)}, T^{(2)}, \ldots, T^{(m)}$ of total length n, a simple $O(m \cdot n)$ time solution to the k-common substring problem can be obtained by a bottom-up traversal of the generalized suffix tree $\mathcal{T}$ of $T^{(1)}, T^{(2)}, \ldots, T^{(m)}$; see e.g. [4, Sect. 7.7]. Hui [1] showed that one can even get rid of the m-factor. More precisely, he gave an $O(n)$ time solution to the problem based on constant time *lca* (longest common ancestor) computations in $\mathcal{T}$; see e.g. [4, Sect. 9.7] for details.

In this paper, we study the k-common repeated substring problem. This problem can also be solved in $O(m \cdot n)$ time by a bottom-up traversal of the generalized suffix tree of $T^{(1)}, T^{(2)}, \ldots, T^{(m)}$, but our goal is to solve it in optimal $O(n)$ time. (Note that Hui's technique [1] cannot be applied here.) Partial results in this direction were obtained by Lee et al. [6] and Lee and Pinzon [7]. In [6], a linear time algorithm is presented that solves the k-common repeated substring problem for the special case $x_1 = \cdots = x_m = 2$. In other words, the number of times a repeated substring should appear is fixed to 2 in each of the strings and cannot be set individually. Lee [7] mentions two drawbacks of this algorithm: It is not easy to implement and it is not memory-efficient. In [7] a linear time algorithm is presented that solves the following special case of the k-common repeated substring problem: Given m strings $T^{(1)}, T^{(2)}, \ldots, T^{(m)}$ of total length n and m positive integers $x_1, \ldots, x_m$, for a *fixed* k with $1 \leq k \leq m$ find a longest string ω for which there are at least k strings $T^{(i_1)}, T^{(i_2)}, \ldots, T^{(i_k)}$ $(1 \leq i_1 < i_2 < \cdots < i_k \leq m)$ such that ω occurs at least x_{i_j} times in $T^{(i_j)}$ for each j with $1 \leq j \leq k$. Consequently, if we would use this algorithm to solve the k-common repeated substring problem, we would have to run the algorithm m times (once for each k, where $1 \leq k \leq m$). Obviously, the resulting worst-case time complexity would be $O(m \cdot n)$. Here, we present the first $O(n)$ time algorithm for the k-common repeated substring problem. Our solution is based on the generalized enhanced suffix array of $T^{(1)}, T^{(2)}, \ldots, T^{(m)}$. For ease of presentation, we use constant time *range minimum queries* (`RMQs`) in certain subproblems, but the algorithms can be implemented without these.

2 Preliminaries

Let Σ be a finite ordered *alphabet* and let T be a string (also called text) of length $|T| = n$ over Σ. For $1 \leq i \leq n$, $T[i]$ denotes the *character at position* i in T. For $i \leq j$, $T[i..j]$ denotes the *substring* of T starting with the character at position i and ending with the character at position j. Furthermore, for $1 \leq i \leq n$, T_i denotes the ith suffix $T[i..n]$ of T.

i	$\mathsf{SA}[i]$	$\mathsf{SA}^{-1}[i]$	$\mathsf{LCP}[i]$	$T_{\mathsf{SA}[i]}$	$T^{\#}_{\mathsf{SA}[i]}$	$text[i]$
1	5	9	0	$\#_1$aac$\#_2$caac$\#_3$	$\#_1$	1
2	9	14	0	$\#_2$caac$\#_3$	$\#_2$	2
3	14	6	0	$\#_3$	$\#_3$	3
4	6	10	0	aac$\#_2$caac$\#_3$	aac$\#_2$	2
5	11	1	3	aac$\#_3$	aac$\#_3$	3
6	3	4	1	ac$\#_1$aac$\#_2$caac$\#_3$	ac$\#_1$	1
7	7	7	2	ac$\#_2$caac$\#_3$	ac$\#_2$	2
8	12	11	2	ac$\#_3$	ac$\#_3$	3
9	1	2	2	acac$\#_1$aac$\#_2$caac$\#_3$	acac$\#_1$	1
10	4	13	0	c$\#_1$aac$\#_2$caac$\#_3$	c$\#_1$	1
11	8	5	1	c$\#_2$caac$\#_3$	c$\#_2$	2
12	13	8	1	c$\#_3$	c$\#_3$	3
13	10	12	1	caac$\#_3$	caac$\#_3$	3
14	2	3	2	cac$\#_1$aac$\#_2$caac$\#_3$	cac$\#_1$	1

Fig. 1 The enhanced suffix array of the text $T =$ acac$\#_1$aac$\#_2$caac$\#_3$ coincides with the generalized enhanced suffix array of the texts acac, aac, and caac

The *suffix array* SA of the string T is an array of integers in the range 1 to n, specifying the lexicographic ordering of the n suffixes of the string T; see Fig. 1. That is, $T_{\mathsf{SA}[1]}, T_{\mathsf{SA}[2]}, \ldots, T_{\mathsf{SA}[n]}$ is the sequence of suffixes of T in ascending lexicographic order. The suffix array can be constructed in linear time [8–10]. The *inverse suffix array* SA^{-1} is an array of size n such that $\mathsf{SA}^{-1}[\mathsf{SA}[i]] = i$ for each i with $1 \le i \le n$; see Fig. 1. Obviously, the inverse suffix array can be constructed in linear time from the suffix array.

The *lcp-array* LCP is an array containing the lengths of the longest common prefix between every pair of consecutive suffixes in SA. We use $lcp(u, v)$ to denote the length of the longest common prefix between strings u and v. Thus, the lcp-array is an array of integers in the range 1 to n such that $\mathsf{LCP}[1] = 0$ and $\mathsf{LCP}[i] = lcp(T_{\mathsf{SA}[i-1]}, T_{\mathsf{SA}[i]})$ for $2 \le i \le n$; see Fig. 1. The lcp-array can be computed in linear time from the suffix array and its inverse [11]. Because suffixes appear in lexicographical order in the suffix array, the equality $lcp(T_{\mathsf{SA}[p]}, T_{\mathsf{SA}[q]}) = \min_{p<l\le q}\{\mathsf{LCP}[l]\}$ holds for all indices p and q with $1 \le p < q \le n$. The value $lcp(T_{\mathsf{SA}[p]}, T_{\mathsf{SA}[q]})$ can be computed in constant time, after a linear time preprocessing of the lcp-array. To be precise, after the preprocessing, so-called range minimum queries of the form $\mathtt{RMQ}(p, q) = \arg\min_{p\le l\le q}\{\mathsf{LCP}[l]\}$ can be answered in constant time [12, 13] and $lcp(T_{\mathsf{SA}[p]}, T_{\mathsf{SA}[q]}) = \mathsf{LCP}(\mathtt{RMQ}(p + 1, q))$. For instance, in the example of Fig. 1, $lcp(T_{\mathsf{SA}[4]}, T_{\mathsf{SA}[9]})$ is found by the range minimum query $\mathtt{RMQ}(5, 9) = 6$ and the table look-up $\mathsf{LCP}[6] = 1$.

Let $T^{(1)}, T^{(2)}, \ldots, T^{(m)}$ be texts of sizes $n_1, n_2, \ldots, n_m$, respectively. The *generalized suffix array* of these texts is an array specifying the lexicographic ordering of all suffixes of the strings $T^{(1)}\#_1, T^{(2)}\#_2, \ldots, T^{(m)}\#_m$, where the $\#_j$, $1 \le j \le m$, are pairwise distinct separator symbols that do not occur in any of the strings. We assume that $\#_1 < \#_2 < \cdots < \#_m$ and all other characters in the alphabet Σ are larger than the separator symbols. It is not difficult to see that—with respect

to the resulting lexicographic ordering—the suffix array of the concatenated string $T = T^{(1)}\#_1 T^{(2)}\#_2 \ldots T^{(m)}\#_m$ (which is of size $n = m + \sum_{l=1}^{m} n_l$) coincides with the generalized suffix array of $T^{(1)}, T^{(2)}, \ldots, T^{(m)}$; see Fig. 1.

The *text array* corresponding to a generalized suffix array is an array of integers in the range 1 to n. It is defined for $1 \le i \le n$ by $\mathit{text}[i] = j$, where j is the number of the text $T^{(j)}$ in which $T_{\mathsf{SA}[i]}$ starts; see Fig. 1. The text array can be computed in linear time from the inverse suffix array as follows. For all j with $1 \le j \le m$ and all i with $j + \sum_{l=1}^{j-1} n_l \le i \le j + \sum_{l=1}^{j} n_l$ set $\mathit{text}[\mathsf{SA}^{-1}[i]] = j$.

For a suffix $T_{\mathsf{SA}[i]}$ of T, the prefix of $T_{\mathsf{SA}[i]}$ that ends at the first separator symbol is denoted by $T^{\#}_{\mathsf{SA}[i]}$. More precisely, if suffix $T_{\mathsf{SA}[i]}$ starts in text j (i.e., $\mathit{text}[i] = j$), then $T^{\#}_{\mathsf{SA}[i]} = T[\mathsf{SA}[i]..j + \sum_{l=1}^{j} n_l]$. In the following we identify $T_{\mathsf{SA}[i]}$ with $T^{\#}_{\mathsf{SA}[i]}$. That is, when we write $T_{\mathsf{SA}[i]}$, we actually mean $T^{\#}_{\mathsf{SA}[i]}$.

The generalized suffix array of $T^{(1)}, T^{(2)}, \ldots, T^{(m)}$ together with the corresponding lcp-array and text array will be called the *generalized enhanced suffix array* of $T^{(1)}, T^{(2)}, \ldots, T^{(m)}$. As we have seen, it can be constructed in linear time.

3 The *k* Common Substring Problem

Given m strings $T^{(1)}, T^{(2)}, \ldots, T^{(m)}$ such that $T = T^{(1)}\#_1 T^{(2)}\#_2 \ldots T^{(m)}\#_m$ is of length n, the *k-common substring problem* is to find for all k, $2 \le k \le m$, a longest substring common to at least k of the strings. In the following, for each k, let ℓ_k denote the length of a longest string with this property. Our algorithms for this problem are based on the following theorem.

Theorem 1 *ω is a longest substring common to at least k strings if and only if there exist indices p and q with $1 \le p < q \le n$ such that*

1. $|\{\mathit{text}[p], \mathit{text}[p+1], \ldots, \mathit{text}[q]\}| \ge k$,
2. *ω is a common prefix of $T_{\mathsf{SA}[p]}, T_{\mathsf{SA}[p+1]}, \ldots, T_{\mathsf{SA}[q]}$ and $lcp(T_{\mathsf{SA}[p]}, T_{\mathsf{SA}[q]}) = |\omega|$,*
3. $\mathsf{LCP}[p] < |\omega|$ *and* $\mathsf{LCP}[q+1] < |\omega|$ *hold true,*[1]
4. $|\omega| = \ell_k$.

Proof "if": (1) and (2) imply that ω is a substring common to at least k strings and (4) implies that ω is a longest string with this property.

"only if": If ω is a longest substring common to at least k strings, then $|\omega| = \ell_k$ and there are indices p' and q' with $1 \le p' < q' \le n$ such that $|\{\mathit{text}[p'], \mathit{text}[p'+1], \ldots, \mathit{text}[q']\}| \ge k$, ω is a common prefix of $T_{\mathsf{SA}[p']}, T_{\mathsf{SA}[p'+1]}, \ldots, T_{\mathsf{SA}[q']}$, and $lcp(T_{\mathsf{SA}[p]}, T_{\mathsf{SA}[q]}) = |\omega|$. Let p be the largest index with $p \le p'$ and $\mathsf{LCP}[p] < |\omega|$, and let q be the smallest index with $q' \le q$ and $\mathsf{LCP}[q+1] < |\omega|$. These indices p and q satisfy (1)–(4). □

[1] In fact, we must also consider the boundary case $q = n$. To avoid the need of case distinctions like "$\mathsf{LCP}[q+1] < |\omega|$ or $q = n$", we add the entry $\mathsf{LCP}[n+1] = 0$ to the lcp-array.

Of course, we do not know ℓ_k. That is why we successively compute all strings having properties (1)–(3) and keep track of the currently longest string with these properties.

3.1 A Naive Solution

We use an array A of size $m-1$ that stores for each k, $2 \leq k \leq m$, a pair (lcs, idx), where lcs is the length of a (currently) longest substring ω common to at least k strings and idx is an index such that $T[\mathsf{SA}[idx]..\mathsf{SA}[idx]+lcs-1]=\omega$. In what follows, we denote the first and second component of an array element $A[k]$ by $A[k].lcs$ and $A[k].idx$, respectively. Initially, $A[k]=(0,\bot)$ for all k with $2 \leq k \leq m$.

Moreover, we employ a doubly linked list consisting of exactly m elements. For each text $T^{(j)}$, $1 \leq j \leq m$, there is exactly one element in the list and a pointer $textptr[j]$ to that element. Furthermore, there is a pointer LV to the last element in the list. (The name LV is an acronym for *last visited* because—as we shall see later—LV points to the element that corresponds to the last visited index in the suffix array.) Every element e in the list is a pair (lcp, idx), where lcp is an lcp-value and idx is a position in the suffix array. We denote the first and second component of a list element e by $e.lcp$ and $e.idx$, respectively. Initially, the list has the form $[(0,1),(0,2),\dots,(0,m-1),(|T_{\mathsf{SA}[m]}|,m)]$ and for all j, $1 \leq j \leq m$, $textptr[j]$ points to the element with second component j. This is because the m lexicographically smallest suffixes are $\#_1,\#_2,\dots,\#_m$; cf. Fig. 1. Observe that $|T_{\mathsf{SA}[m]}| = |\#_m| = 1$.

Our algorithm linearly scans the enhanced suffix array starting at index $m+1$, and for each i with $m+1 \leq i \leq n$, it first calls the procedure *lcp_update* with parameter $\mathsf{LCP}[i]$ and then the procedure *list_update* with parameter i.

- *lcp_update*($\mathsf{LCP}[i]$) linearly scans the list from right-to-left (the rightmost element can be found with the LV pointer) and compares the value $e.lcp$ of the current element e with $\mathsf{LCP}[i]$. Suppose that e is the k-th element from right-to-left. If $e.lcp \geq \mathsf{LCP}[i]$, then we compare the (currently best) value $A[k].lcs$ with $e.lcp$. In case $A[k].lcs \leq e.lcp$, we update $A[k]$ by the assignment $A[k] \leftarrow e$. Furthermore, in the case $e.lcp > \mathsf{LCP}[i]$, the value $e.lcp$ is updated by the assignment $e.lcp \leftarrow \mathsf{LCP}[i]$, and the next list element is considered. Otherwise (i.e., $e.lcp < \mathsf{LCP}[i]$) the procedure stops the scan of the list.
- *list_update*(i) updates the list by standard list operations as follows. It deletes the element to which $textptr[text[i]]$ points from the list and adds a new element $(|T_{\mathsf{SA}[i]}|, i)$ at the end of the list. The pointers LV and $textptr[text[i]]$ are updated so that they both point to the added element.

After completion of the linear scan, a final procedure call *lcp_update*(0) ensures that values which are still in the list are also taken into account. Figures 2 and 3 illustrate the algorithm for the example from Fig. 1.

Each execution of the procedure *lcp_update* takes $O(m)$ time, while each execution of the procedure *list_update* takes constant time. Because these procedures are called $O(n)$ times, the worst-case time complexity of the naive algorithm is in $O(m \cdot n)$.

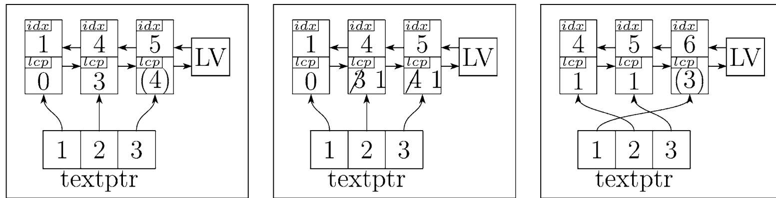

Fig. 2 *Left*: List after $i = 5$ has been processed. The last suffixes of the texts 1, 2, and 3 are $T_{\mathsf{SA}[1]} = \#_1$, $T_{\mathsf{SA}[4]} = \mathtt{aac}\#_2$, and $T_{\mathsf{SA}[5]} = \mathtt{aac}\#_3$. The longest common prefixes of these suffixes with $T_{\mathsf{SA}[i]}$ have length 0, 3, and 4. Furthermore, $A[2] = A[3] = (0, \perp)$. *Middle*: In step $i = 6$, the function *lcp_update* is called with lcp-value $\mathsf{LCP}[6] = 1$ and the *lcp*-values of list elements are being updated. Moreover, $A[2]$ is set to $(3, 4)$. *Right*: List after the procedure call *list_update*(6)

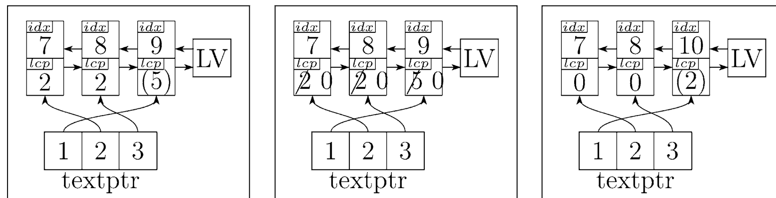

Fig. 3 *Left*: List after $i = 9$ has been processed. *Middle*: In iteration $i = 10$, the function *lcp_update* is called with *lcp*-value $\mathsf{LCP}[10] = 0$ and $A[3]$ is set to $(2, 7)$. *Right*: List after the procedure call *list_update*(10)

We show that for each i with $m \leq i \leq n$ the algorithm maintains the following invariants:

1. The *idx*-values of the list elements are in strict ascending order (left-to-right).
2. If $textptr[j]$ points to the element e, then $text[e.idx] = j$ and $text[p] \neq j$ for all p with $e.idx < p \leq i$. In words, the element e to which $textptr[j]$ points corresponds to the last suffix seen so far that belongs to text $T^{(j)}$.
3. The *lcp*-values of the list elements are in ascending order (from left-to-right).
4. Each element e in the list satisfies $e.lcp = lcp(T_{\mathsf{SA}[e.idx]}, T_{\mathsf{SA}[i]})$.

It is straightforward to verify that our algorithm maintains the invariants (1)–(3). We prove by induction that the crucial property (4) is an invariant and use the fact that for all indices p and q with $1 \leq p < q \leq n$ the equality $lcp(T_{\mathsf{SA}[p]}, T_{\mathsf{SA}[q]}) = \min_{p<l\leq q}\{\mathsf{LCP}[l]\}$ holds true. For $i = m$ property (4) holds. The induction hypothesis states that after index $i-1$ (where $i - 1 \geq m$) has been considered, every element e in the list satisfies $e.lcp = lcp(T_{\mathsf{SA}[e.idx]}, T_{\mathsf{SA}[i-1]}) = \min_{e.idx<l\leq i-1}\{\mathsf{LCP}[l]\}$. After the procedure call *lcp_update*$(\mathsf{LCP}[i])$, we have

$$\begin{aligned} e.lcp &= \min\{lcp(T_{\mathsf{SA}[e.idx]}, T_{\mathsf{SA}[i-1]}), \mathsf{LCP}[i]\} \\ &= \min_{e.idx<l\leq i}\{\mathsf{LCP}[l]\} = lcp(T_{\mathsf{SA}[e.idx]}, T_{\mathsf{SA}[i]}). \end{aligned}$$

Hence property (4) is satisfied. This is also true after the procedure call *list_update*(i) because for the new element $e_{new} = (i, |T_{\mathsf{SA}[i]}|)$, we have $e_{new}.lcp = |T_{\mathsf{SA}[i]}| = lcp(T_{\mathsf{SA}[e_{new}.idx]}, T_{\mathsf{SA}[i]})$.

Theorem 2 *The algorithm solves the k-common substring problem.*

Proof Recall that ℓ_k denotes the length of a longest substring common to at least k strings, where $2 \le k \le m$. Furthermore, let q be the largest index for which there is an index p, $1 \le p < q \le n$, so that the suffixes $T_{\mathsf{SA}[p]}, T_{\mathsf{SA}[p+1]}, \dots, T_{\mathsf{SA}[q]}$ have a common prefix of length ℓ_k and $|\{text[p], text[p+1], \dots, text[q]\}| \ge k$. Because q is the largest index with this property, it follows that $\mathsf{LCP}[q+1] < \ell_k$. Now consider the list before *lcp_update*($\mathsf{LCP}[q+1]$) is called. By the fourth invariant, we know that each element e in the list satisfies $e.lcp = lcp(T_{\mathsf{SA}[e.idx]}, T_{\mathsf{SA}[q]})$. Let e' be k-th element in the list (from right-to-left). As the suffixes $T_{\mathsf{SA}[p]}, T_{\mathsf{SA}[p+1]}, \dots, T_{\mathsf{SA}[q]}$ have a common prefix of length ℓ_k and $|\{text[p], text[p+1], \dots, text[q]\}| \ge k$, the first k elements (from right-to-left) in the list have an *lcp*-value $\ge \ell_k$. Moreover, $e'.lcp = \ell_k$ because otherwise there would be a longer substring common to k strings. Consequently, when the procedure *lcp_update*($\mathsf{LCP}[q+1]$) is called, $A[k]$ is correctly updated by $A[k] \leftarrow e'$. By the choice of index q, $A[k]$ remains unchanged from that point on. □

3.2 A Linear Time Solution

To obtain a linear time solution, we combine all elements of the doubly linked list having the same *lcp*-value into intervals. This implies that every interval can be identified by its unique lcp-value. Because the *lcp*-values of the elements are in ascending order (from left-to-right) in the list, it follows that the *lcp*-values of intervals are in strict ascending order (from left-to-right) in the list. To represent intervals, each list element now has the five components (*lcp*, *idx*, *begin*, *end*, *size*).

- *begin*: Pointer (from the last element of the interval) to the first element of the interval.
- *end*: Pointer (from the first element of the interval) to the last element of the interval.
- *size*: Number of elements in the interval.

The new components *begin*, *end*, and *size* are only relevant for the first and the last element of an interval.

We can access all intervals by following the *begin* pointers as follows. We start with the LV pointer and find the last element of the first interval I_1 (from right-to-left). Following the *begin* pointer of that last element, we reach the first element e_1 of I_1. Note that the size information can be used to compute the position k_1 (from right-to-left) of e_1 in the list, namely $k_1 = e_1.size$. Then we find the element left to it with the help of the usual links of the doubly linked list. This element is the last element of the second interval I_2 and we can reach the first element e_2 of I_2 by following the begin pointer of the last element of I_2. The position k_2 (from right-to-left) of e_2 in the list is $k_2 = k_1 + e_2.size$. In this way, we can proceed until all intervals have been found.

It will also be necessary to access an interval of a certain *lcp*-value in constant time. To this end, we use an array $intptr[1..n]$ of *interval pointers*. To be precise, $intptr[j]$ points to the first element of the interval with *lcp*-value j.

Initially, the list contains the elements $(0, 1), (0, 2), \ldots, (0, m-1), (|T_{\mathsf{SA}[m]}|, m)$ divided into two intervals: The first $m-1$ elements (from left-to-right) form an interval and the second interval solely consists of the last element.

The linear time algorithm has the same structure as the naive algorithm. It satisfies invariants (1)–(2), and invariants (3)–(4) for all elements that are the first element of an interval. The algorithm linearly scans the enhanced suffix array starting at index $m+1$, and for each i with $m+1 \le i \le n$, it first calls the procedure *lcp_update* and then the procedure *list_update*.

- *lcp_update*($\mathsf{LCP}[i]$) linearly scans the list of intervals from right-to-left as described above. Let e be the first element of the current interval and let k be its position (from right-to-left) in the list. If $e.lcp \ge \mathsf{LCP}[i]$, then we compare the (currently best) value $A[k].lcs$ with $e.lcp$. In case $A[k].lcs \le e.lcp$, we update $A[k]$ by the assignment $A[k] \leftarrow (e.lcp, e.idx)$. Then the next interval (provided it exists) is considered. If its *lcp*-value is greater than or equal to $\mathsf{LCP}[i]$, it will be set to be the current interval and so on. Otherwise, if the *lcp*-value of the next interval is strictly smaller than $\mathsf{LCP}[i]$, then e is the first element of a new interval with *lcp*-value $\mathsf{LCP}[i]$ and size k. Consequently, we update these values by $e.lcp \leftarrow \mathsf{LCP}[i]$ and $e.size \leftarrow k$. Furthermore, we set $e.end \leftarrow e'$ and $e'.begin \leftarrow e$, where e' is the last element of the list (LV points to it). Finally, the interval pointer *intptr*[$\mathsf{LCP}[i]$] must point to e and it is updated accordingly.
- *list_update*(i) deletes the element $\tilde{e}$ from the list to which *textptr*[*text*[i]] points. However, this must be done with care. First, the size of the interval to which $\tilde{e}$ belongs must be decreased by one. Second, if $\tilde{e}$ is the first or the last element of its interval, then *begin* and *end* pointers must be modified accordingly. Nevertheless, these updates can be done in constant time provided that the first and last element of the interval to which $\tilde{e}$ belongs can be found in constant time. According to invariant (4), this interval has *lcp*-value $lcp(T_{\mathsf{SA}[\tilde{e}.idx]}, T_{\mathsf{SA}[i]})$, and the *lcp*-value can be identified in constant time by $\mathsf{LCP}[\mathtt{RMQ}(\tilde{e}.idx+1, i)]$. Then, one finds the first element $\overline{e}$ of the interval to which $\tilde{e}$ belongs by following the interval pointer *intptr*[$\mathsf{LCP}[\mathtt{RMQ}(\tilde{e}.idx+1, i)]$], while the last element of that interval can be found with the help of the pointer $\overline{e}.end$. Moreover, the procedure *list_update*(i) adds a new interval at the end of the list. This interval consists of the element $e = (|T_{\mathsf{SA}[i]}|, i, e, e, 1)$, i.e., both pointers $e.begin$ and $e.end$ point to e itself. Finally, the pointers LV, *textptr*[*text*[i]], and *intptr*[$|T_{\mathsf{SA}[i]}|$] are updated so that they all point to the added element.

Figure 4 illustrates how the algorithm works.

In contrast to the naive algorithm, when the algorithm updates an entry in the array A, say $A[k]$, then it does not check whether other entries $A[k']$ with $k' < k$ have to be updated as well. This is because the algorithm directly jumps from the last element of an interval to the first element and skips the elements in between them. Consequently, in the final state of the array A, there may be entries $A[k]$ and $A[k']$ with $2 \le k' < k \le m$ such that $\ell_k = A[k].lcs > A[k'].lcs$. This means that the algorithm has found a string common to k strings that is longer than the (currently longest) string common to k' strings, where $k' < k$. Therefore, in a final phase, the algorithm scans the array A from right-to-left and for $j = m$ downto $j = 3$ it tests

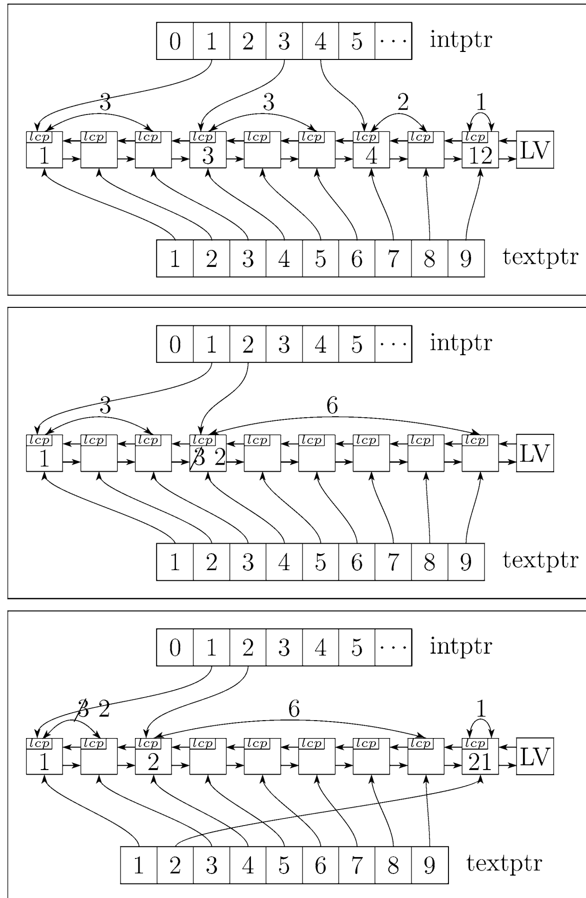

Fig. 4 In our fictitious example, there are nine texts and the upper figure depicts the point of departure. For each element e in the list, the component $e.idx$ is omitted. Furthermore, the component $e.lcp$ is only shown for the relevant elements, viz. the first element of intervals. The *begin* and *end* pointers of an interval are drawn as an *arc with two arrowheads*, and the size of the interval is represented by the *number above this arc*. In the next iteration i, we have $\mathsf{LCP}[i] = 2$, $text[i] = 2$, and $|T_{\mathsf{SA}[i]}| = 21$. The *middle figure* shows the situation after the procedure call *lcp_update*(2), while the *lower figure* depicts the situation after the procedure call *list_update*(i)

whether $A[j].lcs > A[j-1].lcs$ holds true. If so, then it updates $A[j-1]$ by the assignment $A[j-1] \leftarrow A[j]$. The correctness of the algorithm now follows from the invariants as in Sect. 3.1.

Let us turn to the overall complexity of the algorithm. In each iteration, at most two intervals are created. Thus, the algorithm creates at most $2n$ intervals. When the procedure *lcp_update*($\mathsf{LCP}[i]$) reads an interval, it either overwrites this interval with the new interval or it stops at this interval. Clearly, every interval can be overwritten only once. Thus, in all iterations the procedure *lcp_update* can overwrite at most $2n$ intervals. Hence it takes $O(n)$ time. Since the same is true for the procedure *list_update*, the overall running time of the algorithm is $O(n)$.

4 The k Common Repeated Substring Problem

Given m strings $T^{(1)}, T^{(2)}, \ldots, T^{(m)}$ such that $T = T^{(1)}\#_1 T^{(2)}\#_2 \ldots T^{(m)}\#_m$ is of length n and m positive integers $x_1, \ldots, x_m$, the *k-common repeated substring problem* is to find, for all k with $1 \leq k \leq m$, a longest string ω for which there are at

least k strings $T^{(i_1)}, T^{(i_2)}, \ldots, T^{(i_k)}$ $(1 \le i_1 < i_2 < \cdots < i_k \le m)$ such that ω occurs at least x_{i_j} times in $T^{(i_j)}$ for each j with $1 \le j \le k$. Again, let ℓ_k denote the length of a longest string with this property.

Our algorithms for this problem are based on the following analogon to Theorem 1.

Theorem 3 *ω is a longest string for which there are at least k strings $T^{(i_1)}$, $T^{(i_2)}, \ldots, T^{(i_k)}$ $(1 \le i_1 < i_2 < \cdots < i_k \le m)$ such that ω occurs at least x_{i_j} times in $T^{(i_j)}$ if and only if there exist indices p and q with $1 \le p < q \le n$ such that*

1. *For each j with $1 \le j \le k$: $|\{l \mid p \le l \le q \text{ and } text[l] = i_j\}| \ge x_{i_j}$*
2. *ω is a common prefix of $T_{\mathsf{SA}[p]}, T_{\mathsf{SA}[p+1]}, \ldots, T_{\mathsf{SA}[q]}$ and $lcp(T_{\mathsf{SA}[p]}, T_{\mathsf{SA}[q]}) = |\omega|$,*
3. *$\mathsf{LCP}[p] < |\omega|$ and $\mathsf{LCP}[q+1] < |\omega|$,*
4. *$|\omega| = \ell_k$.*

Proof Similar to the proof of Theorem 1. □

4.1 A Naive Solution

Again, array A stores the currently best solutions, but this time for each k with $1 \le k \le m$. The doubly linked list now contains $\sum_{j=1}^{m} x_j$ many elements, and each element e in the list is now a triple $(lcp, idx, flag)$, where the new component $e.flag$ is a Boolean flag. The initial list is obtained by adding, for each text T_j, $x_j - 1$ many elements of the form $(lcp, idx, flag) = (0, -1, flag)$ to the left of the doubly linked list $[(0, 1, flag), (0, 2, flag), \ldots, (0, m-1, flag), (|T_{\mathsf{SA}[m]}|, m, flag)]$. For each text T_j, the flag of the leftmost element in the list that belongs to T_j is set to *true*, while the flags of all other elements are set to *false*. For each $1 \le j \le m$, there is not one text pointer $textptr[j]$ but there is an array $textptr[j][0..x_j - 1]$ containing x_j many text pointers. Initially, $textptr[j][0]$ points to the leftmost element of text T_j in the list, $textptr[j][1]$ points to the second leftmost element of text T_j in the list, etc. So $textptr[j][x_j - 1]$ points to the rightmost element of text T_j in the list, namely $(0, j, flag)$ if $j \ne m$ and $(|T_{\mathsf{SA}[m]}|, m, flag)$ if $j = m$. Furthermore, there is an array $LM[1..m]$ such that entry $LM[j]$ is an integer in the range 0 to $x_j - 1$. (LM is an acronym for leftmost.) Initially, $LM[j] = 0$ for each j with $1 \le j \le m$.

Our modified algorithm also satisfies the invariants (1), (3), and (4) of Sect. 3.1 and, for each $1 \le j \le m$, the following modified invariant (2): Suppose that $l = LM[j]$ and the text pointers

$$textptr[j][l], textptr[j][l+1], \ldots, textptr[j][x_j - 1],$$
$$textptr[j][0], textptr[j][1], \ldots, textptr[j][l-1]$$

point to the elements $e_1, \ldots, e_{x_j}$ (i.e., $textptr[j][l]$ points to e_1, $textptr[j][l+1]$ points to e_2, etc.), then

(a) $text[e_p.idx] = j$ for all $1 \le p \le x_j$,
(b) the sequence of indices $e_1.idx, \ldots, e_{x_j}.idx$ is strictly increasing, and
(c) $|\{q \mid e_1.idx \le q \le e_{x_j}.idx \text{ and } text[q] = j\}| = x_j$.

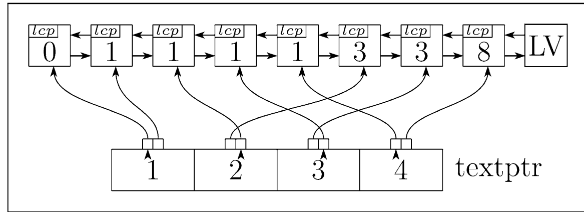

Fig. 5 For each text $T^{(j)}$, there are x_j elements in the list (here $x_1 = x_2 = x_3 = x_4 = 2$), and the pointers in the *textptr*$[j]$ array point to them. The pointer to the leftmost element corresponding to text $T^{(j)}$ is indicated by an arrowhead at the *textptr*$[j]$ array

In words, the elements $e_1, \ldots, e_{x_j}$ correspond to the last x_j suffixes seen so far that belong to text $T^{(j)}$ and *textptr*$[j][LM[j]]$ points to the leftmost of these elements in the list. An illustration can be found in Fig. 5. Moreover, the Boolean flag of an element is set if and only if one of the pointers *textptr*$[1][LM[1]]$,*textptr*$[2][LM[2]], \ldots,$ *textptr*$[m][LM[m]]$ points to it. In other words, if we scan the list, then the Boolean flag of the current element e tells us whether e is the leftmost element for some $1 \le j \le m$ or not.

As in Sect. 3.1 the algorithm linearly scans the enhanced suffix array starting at index $m + 1$, and for each i with $m + 1 \le i \le n$, it first calls the procedure *lcp_update* with parameter $\mathsf{LCP}[i]$ and then the procedure *list_update* with parameter i.

- We modify procedure *lcp_update* from Sect. 3.1 as follows. There, k is the number of elements from the rightmost element in the list to the current element e. Here, k is the number of elements from the rightmost element to e whose Boolean flag is set.
- *list_update*(i) updates the list as follows. If *text*$[i] = j$, then *list_update*(i) deletes the element to which *textptr*$[j][LM[j]]$ points from the list and adds a new element $(|T_{\mathsf{SA}[i]}|, i, \mathit{flag})$ at the end of the list, where $\mathit{flag} = \mathit{true}$ if $x_j = 1$ and $\mathit{flag} = \mathit{false}$ otherwise. The pointer LV is updated so that it points to the added element. Furthermore, by the assignment $LM[j] \leftarrow ((LM[j] + 1) \ \mathtt{mod} \ x_j)$ we make sure that *textptr*$[j][LM[j]]$ points to the leftmost element belonging to text $T^{(j)}$, and we set the flag of this leftmost element.

The worst-case time complexity of the algorithm is obviously in $O(n \cdot \sum_{j=1}^{m} x_j)$.

4.2 A Linear Time Solution

As in Sect. 3.2, we combine all elements having the same *lcp*-value into intervals. The elements in the list have the same five components, only the meaning of the *size* component is different. It now stores the number of elements in the interval whose Boolean flag is set.

Again, the algorithm linearly scans the enhanced suffix array starting at index $m + 1$. The procedure *lcp_update* is the same as in Sect. 3.2, only the procedure *list_update* needs the following slight changes.

If *text*$[i] = j$, then *list_update*(i) deletes the element $\tilde{e}$ to which *textptr*$[j][LM[j]]$ points. The interval to which $\tilde{e}$ belongs can be found as in Sect. 3.2 and the size component of the first element of the interval is decreased by one. By the assignment $LM[j] \leftarrow ((LM[j] + 1) \ \mathtt{mod} \ x_j)$ we make sure that *textptr*$[j][LM[j]]$ points to

the new leftmost element belonging to text $T^{(j)}$, find the interval to which this element belongs (again by a range minimum query and a look-up in the LCP array), and increase the size component of the first element of this interval by one.

It follows as in Sect. 3.2 that the worst-case time complexity of this algorithm is $O(n)$.

5 Conclusions

We have presented a new linear time algorithm for the k-common substring problem. In contrast to previous algorithms, small modifications to this algorithm are sufficient to solve a natural generalization of this problem—the k-common repeated substring problem—in optimal time. To the best of our knowledge, the modified algorithm is the first $O(n)$ time solution to the k-common repeated substring problem.

It should be pointed out that the algorithms can easily be modified so that they output more information about the solutions to the problem under consideration. We explain this for the k-common substring problem, but the same is true for the k-common repeated substring problem. In our exposition, array A stores for each k a pair (lcs, idx), where lcs is the length of a longest substring ω common to at least k strings and idx is an index such that $T[\mathsf{SA}[idx]..\mathsf{SA}[idx] + lcs - 1] = \omega$. However, the algorithms actually compute indices p and q with $1 \le p < q \le n$ such that ω is a common prefix of $T_{\mathsf{SA}[p]}, T_{\mathsf{SA}[p+1]}, \ldots, T_{\mathsf{SA}[q]}$ and $lcp(T_{\mathsf{SA}[p]}, T_{\mathsf{SA}[q]}) = |\omega| = lcs$; cf. Theorem 1. Thus, if array A stores pairs of the form $(lcs, [p..q])$, then the interval $[p..q]$ contains at least one position from each of the k texts at which ω occurs. Moreover, if the array A stores pairs of the form $(lcs, [[p_1..q_1], \ldots, [p_r..q_r]])$ such that the list $[[p_1..q_1], \ldots, [p_r..q_r]]$ contains all intervals $[p_s..q_s]$ satisfying Theorem 1, then the algorithms can output all longest substrings common to at least k strings and not just one.

Pseudo-code and implementations of the algorithms can be found at http://www.uni-ulm.de/in/theo/research/seqana.html.

Acknowledgements We thank the anonymous reviewers for their constructive comments.

References

1. Hui, L.C.K.: Color set size problem with applications to string matching. In: Proc. 3rd Annual Symposium on Combinatorial Pattern Matching. Lecture Notes in Computer Science, vol. 644, pp. 230–243. Springer, Berlin (1992)
2. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**(2), 323–350 (1977)
3. Apostolico, A.: The myriad virtues of subword trees. In: Combinatorial Algorithms on Words, pp. 85–96. Springer, Berlin (1985)
4. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1999)
5. Weiner, P.: Linear pattern matching algorithms. In: Proc. 14th IEEE Annual Symposium on Switching and Automata Theory, pp. 1–11. IEEE, New York (1973)
6. Lee, I., Iliopoulos, C.S., Park, K.: Linear time algorithm for the longest common repeat problem. J. Discrete Algorithms **5**(2), 243–249 (2007)

7. Lee, I., Pinzon-Ardila, Y.J.: A simple algorithm for finding exact common repeats. IEICE Trans. **90D**(12), 2096–2099 (2007)
8. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Proc. 30th International Colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science, vol. 2719, pp. 943–955. Springer, Berlin (2003)
9. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Proc. 14th Annual Symposium on Combinatorial Pattern Matching. Lecture Notes in Computer Science, vol. 2676, pp. 200–210. Springer, Berlin (2003)
10. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In: Proc. 14th Annual Symposium on Combinatorial Pattern Matching. Lecture Notes in Computer Science, vol. 2676, pp. 186–199. Springer, Berlin (2003)
11. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Proc. 12th Annual Symposium on Combinatorial Pattern Matching. Lecture Notes in Computer Science, vol. 2089, pp. 181–192. Springer, Berlin (2001)
12. Berkman, O., Vishkin, U.: Recursive star-tree parallel data structure. SIAM J. Comput. **22**(2), 221–242 (1993)
13. Fischer, J., Heun, V.: A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In: Proc. 1st International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies. Lecture Notes in Computer Science, vol. 4614, pp. 459–470. Springer, Berlin (2007)