Computing Runs on a General Alphabet

Dmitry Kosolobov

Ural Federal University, Ekaterinburg, Russia

Abstract
We describe a RAM algorithm computing all runs (=maximal repetitions) of a given string of length \( n \) over a general ordered alphabet in \( O(n \log^3 n) \) time and linear space. Our algorithm outperforms all known solutions working in \( \Theta(n \log \sigma) \) time provided \( \sigma = n^{\Omega(1)} \), where \( \sigma \) is the number of distinct letters in the input string. We conjecture that there exists a linear time RAM algorithm finding all runs.

Keywords: runs, general alphabet, maximal repetitions, linear time, repetitions

1. Introduction
Repetitions of strings are fundamental objects in both stringology and combinatorics on words. In some sense the notion of run, introduced by Main in \([13]\), allows to grasp the whole repetitive structure of a given string in a relatively simple form. Recall that a run of a string is a nonextendable (with the same minimal period) substring whose minimal period is at most half of its length. In \([8]\) Kolpakov and Kucherov showed that any string of length \( n \) contains \( O(n) \) runs and proposed an algorithm computing all runs in linear time on an integer alphabet \( \{0, 1, \ldots, n^{O(1)}\} \) and \( O(n \log \sigma) \) time on a general ordered alphabet, where \( \sigma \) is the number of distinct letters in the input string. In \([1]\) Bannai et al. described another interesting algorithm computing all runs in \( O(n \log \sigma) \) time. Modifying the approach of \([1]\), we prove the following theorem.

Theorem. For a general ordered alphabet, there is an algorithm that computes all runs in a string of length \( n \) in \( O(n \log^3 n) \) time and linear space.

This is in contrast to the result of Main and Lorentz \([14]\) who proved that any algorithm deciding whether a string over a general unordered alphabet has at least one run requires \( \Omega(n \log n) \) comparisons in the worst case.

Our algorithm outperforms all known solutions provided \( \sigma = n^{\Omega(1)} \). It should be noted that the algorithm of Kolpakov and Kucherov can hardly be improved in a similar way since it strongly relies on a structure (namely, the Lempel-Ziv decomposition) that cannot be computed in \( o(n \log \sigma) \) time on a general ordered alphabet (see \([1]\)).

Based on some theoretical observations of \([1]\), we conjecture that one can further improve our result.

Conjecture. For a general ordered alphabet, there is a linear time algorithm computing all runs.

2. Preliminaries
A string of length \( n \) over the alphabet \( \Sigma \) is a map \( \{1, 2, \ldots, n\} \rightarrow \Sigma \), where \( n \) is referred to as the length of \( w \), denoted by \( |w| \). We write \( w[i] \) for the \( i \)th letter of \( w \) and \( w[i..j] \) for \( w[i]w[i+1]\ldots w[j] \). A string \( u \) is a substring (or a factor) of \( w \) if \( u = w[i..j] \) for some \( i \) and \( j \). The pair \((i, j)\) is not necessarily unique; we say that \( i \) specifies an occurrence of \( u \) in \( w \). A string can have many occurrences in another string. A substring \( w[i..j] \) [respectively, \( w[i..n] \)] is a prefix [respectively, suffix] of \( w \). An integer \( p \) is a period of \( w \) if \( 0 < p < |w| \) and \( w[i] = w[i+p] \) for all \( i = 1, \ldots, |w| - p \). For integers \( i \) and \( j \), the set \( \{k \in \mathbb{Z} : i \leq k \leq j\} \) (possibly empty) is denoted by \([i..j]\). Denote \([i..j] = [i..j-1] \) and \([i..j] = [i+1..j]\).

A run of a string \( w \) is a substring \( w[i..j] \) whose period is at most half of the length of \( w[i..j] \) and such that both substrings \( w[i-1..j] \) and \( w[i+1..j] \), if defined, have strictly greater minimal periods than \( w[i..j] \).

Hereafter, \( w \) denotes the input string of length \( n \).
In the longest common extension (LCE) problem one has the queries \(LCE(i, j)\) returning for given positions \(i\) and \(j\) of \(w\) the length of the longest common prefix of the suffixes \(w[i..n]\) and \(w[j..n]\). It is well known that one can perform the LCE queries in constant time after a preprocessing of \(w\) requiring \(O(n \log \sigma)\) time, where \(\sigma\) is the number of distinct letters in \(w\) (e.g., see \([1]\)). It appears that the time consumed by the LCE queries is dominating in the algorithm of \([1]\); namely, one can easily prove the following lemma.

**Lemma 1** (see \([1]\)). Suppose we can compute any sequence of \(O(n)\) LCE queries on \(w\) in \(O(nf(n))\) time for some function \(f(n)\); then we can find all runs of \(w\) in \(O(n + f(n))\) time.

In what follows we describe an algorithm that computes \(O(n)\) LCE queries in \(O(n \log \frac{3}{2} n)\) time and thus prove Theorem using Lemma 1. The key notion in our construction is a difference cover. Let \(k \in \mathbb{N}\). A set \(D \subseteq [0..k]\) is called a difference cover of \([0..k]\) if for any \(x \in [0..k]\), there exist \(y, z \in D\) such that \(y - z \equiv x \pmod{k}\). Clearly \(|D| \geq \sqrt{k}\). Conversely, for any \(k \in \mathbb{N}\), there is a difference cover of \([0..k]\) with \(O(\sqrt{k})\) elements and it can be constructed in \(O(k)\) time (see \([4]\)).

**Example.** The set \(D = \{1, 2, 4\}\) is a difference cover of \([0..5]\).

\[
\begin{array}{c|cccc}
  x & 0 & 1 & 2 & 3 & 4 \\
  y, z & 1, 1 & 2, 1 & 1, 4 & 4, 1 & 1, 2
\end{array}
\]

![Difference cover example](image)

(\text{the figure is from [3]}.)

**Lemma 2** (see \([4]\)). Let \(D\) be a difference cover of \([0..k]\). For any integers \(i, j\), there exists \(d \in [0..k]\) such that \((i - d) \mod k \in D\) and \((j - d) \mod k \in D\).

3. Longest Common Extensions

At the beginning, our algorithm fixes an integer \(\tau\) (the precise value of \(\tau\) is given below). Let \(D\) be a difference cover of \([0..\tau^2]\) such that \(|D| = O(\tau)\). Denote \(M = \{i \in [1..n] : i \mod \tau^2 \in D\}\). Obviously, we have \(|M| = O(\frac{n}{\tau})\). Our algorithm builds in \(O(\frac{3}{2}(\tau^2 + \log n)) = O(\frac{3}{2} \log n + n\tau)\) time a data structure that can calculate \(LCE(i, j)\) in constant time for any \(i, j \in M\). To compute \(LCE(i, j)\) for arbitrary \(i, j \in [1..n]\), we simply naively compare \(w[i..n]\) and \(w[j..n]\) from left to right until we reach positions \(i + d\) and \(j + d\) such that \(i + d \in M\) and \(j + d \in M\), and then we obtain \(LCE(i, j) = d + LCE(i + d, j + d)\) in constant time. By Lemma 2 we have \(d < \tau^2\) and therefore, the value \(LCE(i, j)\) can be computed in \(O(\tau^2)\) time. Thus, our algorithm can execute any sequence of \(O(n)\) LCE queries in \(O(\frac{3}{2} \log n + n\tau^2)\) time. Putting \(\tau = \lceil \log^2 n \rceil\), we obtain \(O(\frac{3}{2} \log n + n\tau^2) = O(n \log^2 n)\). Now it suffices to describe the data structure for the LCE queries on the positions \(M\).

The data structure that we build in the preprocessing step is the minimal in the number of vertices compacted trie \(T\) such that for any \(i \in M\), the string \(w[i..n]\) can be spelled out on the path from the root to some leaf of \(T\) (see Figure 1). We store the labels on the edges of \(T\) as pointers to substrings of \(w\). The trie \(T\) is commonly referred to as a sparse suffix tree. Obviously, \(T\) occupies \(O(n)\) space. For simplicity, we assume that \(w[n]\) is a special letter that does not occur in \(w[1..n-1]\), so, for each \(i \in M\), the suffix \(w[i..n]\) corresponds to some leaf of \(T\).

Let \(i, j \in M\). It is straightforward that \(LCE(i, j)\) is equal to the length of the string written on the path from the root of \(T\) to the nearest common ancestor of the leaves corresponding to the suffixes \(w[i..n]\) and \(w[j..n]\). Using the construction of \([6]\), one can preprocess \(T\) in \(O(\frac{3}{2} \log n)\) time such that the nearest common ancestor of any two leaves can be found in constant time. So, to finish the prove, it remains to describe how to build \(T\) in \(O(n(\tau^2 + \log n))\) time.

In general our construction is similar to that of \([10]\). We use the fact that the set \(M\) has the “period” \(\tau^2\), i.e., for any \(i \in M\), we have \(i + \tau^2 \in M\) provided \(i + \tau^2 \leq n\). Our algorithm consecutively inserts the suffixes \(\{w[i..n] : i \in M\}\) in \(T\) from right to left. Suppose for some \(k \in M\), we already have a compacted trie \(T\) that contains the suffixes \(w[i..n]\) for all \(i \in M \cap (k..n]\). We are to insert the suffix \(w[k..n]\) in \(T\). To perform the insertion efficiently, we maintain four additional data structures.

1. An order on the leaves of \(T\). We store all leaves of \(T\) in a linked list in the lexicographical order of the corresponding suffixes. We maintain on this list the order maintenance data structure of \([2]\) that
allows to determine whether a given leaf precedes another leaf in the list in constant time. The insertion in this list takes constant amortized time. Hereafter, we say that a leaf \( x \) of \( T \) precedes [respectively, succeeds] another leaf \( y \) if \( x \) precedes [respectively, succeeds] \( y \) in the list of leaves.

2. Slow LCE queries. Denote by \( i_1, i_2, \ldots, i_n \) the sequence of all positions \( M \cap (k..n) \) in the increasing lexicographical order of the corresponding suffixes \( w[i_1..n], w[i_2..n], \ldots, w[i_n..n] \). For each \( i_p \in M \cap (k..n) \), we associate with the leaf corresponding to the suffix \( w[i_p..n] \) the value \( LCE(i_p, i_{p+1}) \). It is easy to see that for any \( i_p, i_q \in M \cap (k..n) \) such that \( p < q \), we have \( LCE(i_p, i_q) = \min\{LCE(i_p, i_{p+1}), LCE(i_{p+1}, i_{p+2}), \ldots, LCE(i_{q-1}, i_q)\} \). According to this observation, we store all leaves of \( T \) in an augmented balanced search tree \( C \) that allows to calculate \( LCE(i_p, i_q) \) for any such \( i_p \) and \( i_q \) in \( O(\log n) \) time. It is well known that the insertion in \( C \) of a new leaf with an associated LCE value requires \( O(\log n) \) amortized time.

3. The “top” part of \( T \). We maintain a compact trie \( S \) that contains the strings \( w[i..i+\tau^2] \) for all \( i \in M \cap (k..n) \) (we assume \( w[j] = w[n] \) for all \( j > n \) and thus \( w[i..i+\tau^2] \) is always well defined). Informally, \( S \) is the “top” part of \( T \), so, we augment each vertex of \( S \) with a link to the corresponding vertex of \( T \). We maintain on \( S \) the data structure of [9] supporting the insertions in \( O(\tau^2 + \log n) \) amortized time. Let \( x \) be a leaf of \( S \) corresponding to a string \( w[i..i+\tau^2] \). We augment \( x \) with a balanced search tree \( B_x \) that contains the leaves of \( T \) corresponding to all suffixes \( w[j..n] \) such that \( w[j-\tau^2..j] = w[i..i+\tau^2] \) in the order induced by the list of all leaves of \( T \) (see Figure 2). One can easily show that \( S \) together with the associated search trees occupies \( O(\tau^2) \) space in total.

4. Dynamic weighted ancestors. We maintain on \( T \) the dynamic weighted ancestor data structure of [9] that, for any given vertex \( x \) and an integer \( c \), can find in \( O(\log n) \) time the nearest ancestor of \( x \) such that the length of the string written on the path from the root to this ancestor is less than \( c \). When we insert a new vertex in \( T \), the modification of this structure takes \( O(\log n) \) amortized time.

Example. Let \( \tau^2 = 4 \). The set \( D = \{0, 1, 3\} \) is a difference cover of \( [0..\tau^2] \). Consider the string \( w = abababababbb \); the emphasized positions are from \( M = \{ i \in [1..n] : \ (i \mod \tau^2) \in D \} \). The sparse suffix tree of \( w \) is presented in Figure 1. Figure 2 depicts the corresponding compacted trie \( S \); each leaf of \( S \) is augmented with a balanced search tree of certain leaves of \( T \) (see the description above).

The construction of \( T \). Now to insert \( w[k..n] \) in \( T \), we first insert \( w[k..k+\tau^2] \) in \( S \) in \( O(\tau^2 + \log n) \) time. If \( S \) does not contain \( w[k..k+\tau^2] \), then we attach a new leaf in \( T \) using the links from \( S \) to \( T \) and modify in an obvious way all related data structures: the list of leaves of \( T \), the newly created balanced search tree associated with the new leaf of \( S \), the balanced search tree \( C \), and the dynamic weighted ancestor data structure on \( T \). The modifications require \( O(\log n) \) amortized time.

Now suppose \( S \) contains \( w[k..k+\tau^2] \). Denote by \( v \) the leaf of \( S \) corresponding to \( w[k..k+\tau^2] \). Let \( y \) be the leaf of \( T \) corresponding to the suffix \( w[k+\tau^2..n] \) (recall that \( k+\tau^2 \in M \)). In \( O(\log n) \) time we obtain the immediate predecessor and successor of \( y \) in the search tree \( B_v \), denoted by \( x \) and \( z \), respectively. Notice that \( x \) is the immediate predecessor only in the set of all leaves contained in \( B_v \), but it may not be the immediate predecessor in the whole list of leaves of \( T \); the situation with \( z \) is similar. Let \( x \)
and $z$ correspond to suffixes $w[i_x..n]$ and $w[i_z..n]$, respectively. Since $w[i_x-\tau^2..i_x] = w[i_z-\tau^2..i_z] = w[k..k+\tau^2]$, it is straightforward that the suffixes $w[i_x-\tau^2..n]$ and $w[i_z-\tau^2..n]$ are, respectively, the immediate predecessor and successor of the suffix $w[k..n]$ in the set of all suffixes inserted in $T$. Hence, we must insert $w[k..n]$ between these suffixes.

It is easy to see that $LCE(k,i_x-\tau^2) = \tau^2 + LCE(k+\tau^2,i_x)$ and $LCE(k,i_z-\tau^2) = \tau^2 + LCE(k+\tau^2,i_z)$. The values $LCE(k+\tau^2,i_x)$ and $LCE(k+\tau^2,i_z)$ can be computed in $O(\log n)$ time using the balanced search tree $C$. Without loss of generality consider the case $LCE(k,i_x-\tau^2) \geq LCE(k,i_z-\tau^2)$. We find the position where we insert a new leaf in $T$ using the weighted ancestor query on the value $LCE(k,i_x-\tau^2)$ and the leaf of $T$ corresponding to the suffix $w[i_x-\tau^2..n]$. We finally modify all related data structures in an obvious way: the list of leaves of $T$, the balanced search trees $B_k$ and $C$, and the dynamic weighted ancestor data structure on $T$. These modifications require $O(\log n)$ amortized time.

**Time and space.** The insertion of a new suffix in $T$ takes $O(\tau^2 + \log n)$ amortized time. Thus, the construction of $T$ consumes overall $O(n(\tau^2 + \log n))$ time as required. The whole data structure occupies $O(\frac{n}{\tau^2})$ space.

4. Conclusion

It seems that further improvements in the considered problem may be achieved by more and more efficient longest common extension data structures on a general ordered alphabet. One even might conjecture that there is a data structure that can execute any sequence of $k$ $LCE$ queries on a string of length $n$ over a general ordered alphabet in $O(k + n)$ time. Although we do not yet have a theoretical evidence for such strong results.

Another interesting direction is a generalization of our result for the case of online algorithms (e.g., see [7] and [12]).

References

[1] H. Bannai, T. I. S. Inenaga, Y. Nakashima, M. Takeda, K. Tsuruta, The “runs” theorem, arXiv preprint arXiv:1406.0263v4.
[2] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, J. Zito, Two simplified algorithms for maintaining order in a list, in: Algorithms-ESA 2002, vol. 2461 of LNCS, Springer, 2002, pp. 152–164.
[3] P. Bille, I. L. Gørtz, B. Sach, H. W. Vildhøj, Time-space trade-offs for longest common extensions, J. of Discrete Algorithms 25 (2014) 42–50.
[4] S. Burkhardt, J. Kärkkäinen, Fast lightweight suffix array construction and checking, in: CPM 2003, vol. 2676 of LNCS, Springer, 2003.
[5] G. Franceschini, R. Grossi, A general technique for managing strings in comparison-driven data structures, in: ICALP 2004, vol. 3142 of LNCS, Springer, 2004.
[6] D. Harel, R. E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM Journal on Computing 13 (2) (1984) 338–355.
[7] J.-J. Hong, G.-H. Chen, Efficient on-line repetition detection, Theoretical Computer Science 407 (1) (2008) 554–563.
[8] R. Kolpakov, G. Kucherov, Finding maximal repetitions in a word in linear time, in: FOCS 1999, IEEE, 1999.
[9] T. Kopelowitz, M. Lewenstein, Dynamic weighted ancestors, in: SODA 2007, SIAM, 2007.
[10] D. Kosolobov, Faster lightweight Lempel-Ziv parsing, arXiv preprint arXiv:1504.06712.
[11] D. Kosolobov, Lempel-Ziv factorization may be harder than computing all runs, in: STACS 2015, vol. 30 of LIPIcs, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
[12] D. Kosolobov, Online detection of repetitions with backtracking, in: CPM 2015, Springer, 2015.
[13] M. G. Main, Detecting leftmost maximal periodicities, Discrete Applied Mathematics 25 (1) (1989) 145–153.
[14] M. G. Main, R. J. Lorentz, Linear time recognition of squarefree strings, in: Combinatorial Algorithms on Words, Springer, 1985, pp. 271–278.