New simple efficient algorithms computing powers and runs in strings

Maxime Crochemore, C.S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, Krzysztof Stencel, T. Waleń

New Simple Efficient Algorithms Computing Powers and Runs in Strings

Maxime Crochemore[1,3], Costas Iliopoulos[1,4], Marcin Kubica[2],
Jakub Radoszewski[⋆,2], Wojciech Rytter[2,5], Krzysztof Stencel[2,5], and
Tomasz Waleń[2]

[1] King's College London, London WC2R 2LS, UK
maxime.crochemore@kcl.ac.uk, csi@dcs.kcl.ac.uk
[2] Dept. of Mathematics, Computer Science and Mechanics,
University of Warsaw, Warsaw, Poland
[kubica,jrad,rytter,stencel,walen]@mimuw.edu.pl
[3] Université Paris-Est, France
[4] Digital Ecosystems & Business Intelligence Institute,
Curtin University of Technology, Perth WA 6845, Australia
[5] Dept. of Math. and Informatics,
Copernicus University, Toruń, Poland

Abstract. Three new simple $O(n \log n)$ time algorithms related to repeating factors are presented in the paper. The first two algorithms employ only a basic textual data structure called the Dictionary of Basic Factors. Despite their simplicity these algorithms not only detect existence of powers but also find all primitively rooted cubes (as well as higher powers) and all cubic runs. Our third $O(n \log n)$ time algorithm computes all runs and is probably the simplest known efficient algorithm for this problem. It uses additionally the Longest Common Extension function, however, due to relaxed running time constraints, a simple $O(n \log n)$ time implementation can be used. At the cost of logarithmic factor (in time complexity) we have novel algorithmic solutions for several classical string problems which are much simpler than (usually quite sophisticated) linear time algorithms.

Keywords: run, repetition, square, cube (in a string), Dictionary of Basic Factors

1 Introduction

In this paper, we present algorithms finding various types of repetitions in a string: powers (e.g. squares or cubes), cubic runs and runs. Finding repetitions is a fundamental problem in text processing and has numerous applications. Examples of such applications, an explanation of the motivation and related topics can be found in the survey [8].

Various problems related to finding repetitions in a string have already been studied. For the problem of finding all distinct squares, a linear time algorithms are known [14,17,18]. It is also known, that the maximal number of distinct squares in a string is linear [13].

Multiple approaches to searching for squares in a string can be found in the literature, however most of the existing algorithms are rather complex. The first approach, is to check if a string is square-free. $O(n \log n)$ time algorithms for this problem have been presented in [23,24] (the latter one is randomized). The optimal $O(n)$ time algorithms are described in [5,23].

[⋆] Corresponding author. Some parts of this paper were written during the corresponding author's Erasmus exchange at King's College London

Maxime Crochemore, Costas Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, Krzysztof Stencel, Tomasz Waleń: New Simple Efficient Algorithms Computing Powers and Runs in Strings, pp. 138–149 .
Proceedings of PSC 2010, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04597-8 © Czech Technical University in Prague, Czech Republic

Another approach is to find all occurrences of primitively rooted squares in a string. A number of $O(n \log n)$ time algorithms reporting all such occurrences can be found in [2,4,19,22,25,26]. Due to the lower bound shown in [6] these algorithms are optimal.

Yet another approach is to report all occurrences of squares in a string. If we denote the number of such occurrences by z, then both $O(n \log n + z)$ time algorithms [20,22,26] and $O(n + z)$ time algorithms [14,17,18] are known for this problem.

Finally, there are recent results related to on-line square detection (that is, when letters of u are given one by one), improving the time complexity from $O(n \log^2 n)$ [21] to $O(n \log n)$ [16] and $O(n)$ [3].

Let u be a string of length n over a bounded alphabet. In Section 3 a very simple $O(n \log n)$ time algorithm checking whether u contains any kth string power is presented. The algorithm reports all occurrences of primitively rooted kth powers for any $k \geq 3$, in particular, primitively rooted cubes. As a by-product we obtain an alternative, algorithmic proof of the fact [6] that the maximal number of such occurrences is $O(n \log n)$. The output of the algorithm is later on used to list all cubic runs in u, within the same time complexity.

From the aforementioned literature, the papers [2,4,22] deal also with powers of arbitrary (integer) exponent, however the techniques used there (e.g., suffix trees, Hopcroft's factor partitioning) are much more sophisticated than the techniques applied in this paper. The $O(n \log n)$ time algorithm for a single square detection from [23] is in some sense similar to the algorithm presented in this paper. However it is less versatile than ours: we see no simple modification adapting it to detect all occurrences of primitively rooted higher powers.

In Section 4, we present an application of the algorithm finding all occurrences of primitively rooted cubes to find all cubic runs. This algorithm also runs in $O(n \log n)$ time and it does not use any additional advanced techniques.

Finally, in Section 5, we give an algorithm reporting all runs in a string, in $O(n \log n)$ time. It is significantly simpler than all known $O(n \log n)$ time algorithms present implicitly in [2,4,22] and than the optimal $O(n)$ time algorithm [17,18]. The only non-trivial technique used in our algorithm is the Longest Common Extension function. It can be either implemented as described in [10,15] — very efficiently, but using quite sophisticated machinery, or less efficiently, but in a much simpler way, what is sufficient to obtain $O(n \log n)$ time complexity.

2 Preliminaries

We consider *words* (*strings*) over a bounded alphabet Σ, $u \in \Sigma^*$. The empty word is denoted by ε. The positions in u are numbered from 1 to $|u|$. For $u = u_1 u_2 \dots u_n$, by $u[i\,..\,j]$ we denote a *factor* of u equal to $u_i \dots u_j$ (in particular $u[i] = u[i\,..\,i]$). Words $u[1\,..\,i]$ are called prefixes of u, words $u[i\,..\,n]$ suffixes of u, whereas words that are both a prefix and a suffix of u are called *borders* of u.

We say that a positive integer p is a *period* of the word $u = u_1 \dots u_n$ if $u_i = u_{i+p}$ holds for all i, $1 \leq i \leq n - p$. Periods and borders correspond to each other, i.e. u has a period p if and only if it has a border of length $n - p$, see e.g. [7,12].

A *run* (also called a maximal repetition) in a string u is such an interval $[i\,..\,j]$, that:

- the shortest period p of the associated factor $u[i\,..\,j]$ satisfies $2p \leq j - i + 1$,

– the interval can be extended neither to the left nor to the right, without violating the above property, that is, $u[i-1] \neq u[i+p-1]$ and $u[j-p+1] \neq u[j+1]$, provided that the respective characters exist.

A *cubic run* is a run $[i\,.\,.\,j]$ for which the shortest period p satisfies $3p \leq j-i+1$. We identify a run (or a cubic run) with a corresponding triple (i,j,p).

If $w^k = u$ (k is a positive integer) then we say that u is the kth power of the word w. A *square* (*cube*) is the 2nd (3rd) power of some nonempty word.

2.1 Dictionary of Basic Factors

Dictionary of Basic Factors is a simple, yet powerful data-structure. It is widely used in this paper. For a word u of length n, the *Dictionary of Basic Factors* of u (denoted by $\mathsf{DBF}(u)$) consists of a sequence of arrays $Name_t[\,]$, for $0 \leq t \leq \lfloor \log n \rfloor$. Array $Name_t[\,]$ contains information about factors of u of length 2^t — $Name_t[i]$ contains information about word $u[i\,.\,.\,i+2^t-1]$, for $1 \leq i \leq n-2^t+1$. More precisely, value of $Name_t[i]$ is the rank of $u[i\,.\,.\,i+2^t-1]$ among other factors of length 2^t. Hence, values of elements of all the arrays $Name_t[\,]$ are in the range from 1 to n. The important property of $\mathsf{DBF}(u)$, that we exploit, is that $u[i\,.\,.\,i+2^t-1] \leq u[j\,.\,.\,j+2^t-1]$ if and only if $Name_t[i] \leq Name_t[j]$. DBF has a variety of known applications in the field of text and sequence algorithms, see e.g. [11].

$\mathsf{DBF}(u)$ requires $O(n \log n)$ space and can be constructed in $O(n \log n)$ time [12]. $Name_0[\,]$ contains information about consecutive characters of u. So, $Name_0[\,]$ can be computed in $O(n)$ time, by sorting all the letters appearing in u and mapping characters of u to numbers from 1 on. Having computed $Name_t[\,]$, one can easily compute $Name_{t+1}[\,]$ in $O(n)$ time. Factor $u[i\,.\,.\,i+2^{t+1}-1]$ is a concatenation of factors $u[i\,.\,.\,i+2^t-1]$ and $u[i+2^t\,.\,.\,i+2^{t+1}-1]$. Hence, it can be represented by a pair $(Name_t[i], Name_t[i+2^t])$. Then, all such pairs can be sorted lexicographically (in $O(n)$ time) and mapped onto their ranks, that is integers from 1 on. Figure 1 shows the DBF for an example string.

Text	a	b	b	a	a	b	b	a	b	b	a
$Name_0[\,]$	1	2	2	1	1	2	2	1	2	2	1
	$(1,2)$	$(2,2)$	$(2,1)$	$(1,1)$	$(1,2)$	$(2,2)$	$(2,1)$	$(1,2)$	$(2,2)$	$(2,1)$	
$Name_1[\,]$	2	4	3	1	2	4	3	2	4	3	
	$(2,3)$	$(4,1)$	$(3,2)$	$(1,4)$	$(2,3)$	$(4,2)$	$(3,4)$	$(2,3)$			
$Name_2[\,]$	2	5	3	1	2	6	4	2			
	$(2,2)$	$(5,6)$	$(3,4)$	$(1,2)$							
$Name_3[\,]$	2	4	3	1							

Figure 1. Example of DBF computation for word *abbaabbabba*. Factor *abba* appears three times in this word and is represented in $Name_2[\,]$ by 2.

Using DBF, one can compare factors of arbitrary length, as given in the following Lemma, see [12].

Lemma 1. *Having precomputed* $\mathsf{DBF}(u)$*, any two factors of* u *can be compared in* $O(1)$ *time.*

Proof. Let $u[i\,.\,.\,j]$ and $u[i'\,.\,.\,j']$ be the two factors that should be compared. We can assume, that $j-i = j'-i'$, since otherwise they have different lengths, cannot be equal and can be compared by trimming the longer factor.

Let t be such an integer, that $2^t \leq j - i < 2^{t+1}$. Then, it is enough to compare $u[i \mathinner{..} i+2^t-1]$ with $u[i' \mathinner{..} i'+2^t-1]$, and $u[j-2^t+1 \mathinner{..} j]$ with $u[j'-2^t+1 \mathinner{..} j']$. This however can be done by comparing $Name_t[i]$ with $Name_t[i']$, and $Name_t[j-2^t+1]$ with $Name_t[j'-2^t+1]$. □

Later on, we show, that the memory complexity of the presented algorithm can be reduced to $O(n)$, although it uses $\mathsf{DBF}(u)$. To do this, we cannot store all the arrays $Name_t[\ ]$. Instead, we should store just a fixed number (e.g. one) of such arrays, and design the algorithm in such a way, that are used in the ascending order of t. Then, new arrays can be computed when needed, replacing previously used arrays. Still, it is possible to compare factors in $O(1)$ time, provided that the ratio between the length of the compared factors and 2^t is bounded, as expressed by the following Lemma.

Lemma 2. *Let t be a fixed number between 0 and $\lfloor \log n \rfloor$, and let $Name_t[\ ]$ be one of the arrays constituting $\mathsf{DBF}(u)$. It is possible to compare factors of u of length l, using just $Name_t[\ ]$, in constant time, provided that: $l \geq 2^t$ and $\frac{l}{2^t} = O(1)$.*

Proof. The proof is similar to the proof of the previous Lemma. The compared factors can be covered using $O(1)$ factors of length 2^t. Hence, it is enough to compare $O(1)$ pairs of elements of $Name_t[\ ]$. □

3 Detecting String Powers

Let u be a word of length n. The following algorithm tests if u contains a kth power, for $k \geq 2$. It exploits $\mathsf{DBF}(u)$ and two other auxiliary data-structures, denoted by POWERS and $Prev$.

POWERS is a list on which the result is accumulated. Each occurrence of a power of the form $(u[pos \mathinner{..} pos+root-1])^k$ (i.e. kth power of a factor of length $root$, starting at position pos) is represented by a pair $(root, pos)$. The output of the algorithm is a list of pairs denoting kth powers. We allow the same power to be inserted multiple times — at the end the list is sorted and the repetitions are removed.

$Prev[1 \mathinner{..} n]$ is an array of positions in the text, such that $Prev[Name_t[j]]$ is the most recent occurrence of $u[j \mathinner{..} j+2^t-1]$ preceding j, or -1 if there is none.

For all values of t, the algorithm scans the text, and for each position j it checks (in constant time) if factors of u of length 2^t, starting at $Prev[Name_t[j]]$ and j generate a power. Examples of how the algorithm works can be found in Fig. 2 and Table 1.

```
Algorithm DetectPowers($(u, n, k)$)
{detect $k$th string powers in a word $u$, $|u| = n$}
$Name \leftarrow \mathsf{DBF}(u)$
$\mathsf{POWERS} \leftarrow \emptyset$
for $t \leftarrow 0$ to $\lfloor \log n \rfloor$ do
    $Prev \leftarrow (0, 0, \ldots, 0)$
    for $j \leftarrow 1$ to $n - 2^t + 1$ do
        $name \leftarrow Name_t[\,j\,]$
        $pos \leftarrow Prev[\,name\,]$
        $root \leftarrow j - pos$
        if $u[pos \mathinner{..} pos + k \cdot root - 1]$ is (really) a $k$th power
            {constant time test due to DBF} then
            $\mathsf{POWERS}.insert((root, pos))$
        $Prev[\,name\,] \leftarrow j$
$RadixSort(\mathsf{POWERS})$ with repetitions removed
return $\mathsf{POWERS}$
```

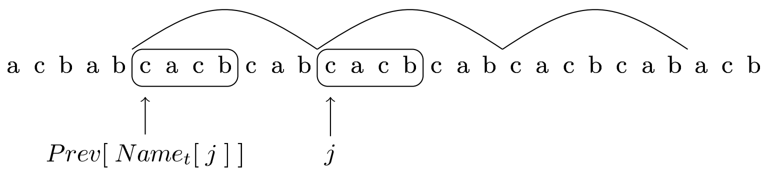

Figure 2. The basic factor *cacb* of rank $t = 2$ at position $j = 13$ generates a cube $(cacbcab)^3$ starting at position $pos = 6$. The same cube is generated for $t = 3$ and $j = 13$, for the basic factor *cacbcabc*.

a	b	b	a	b	a	a	b	b	a	a	b	a	b	b	a	b	a	a	b	a	b	b	a	a	b	b	a	b	a	a	b
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

Table 1. For the Thue-Morse word of length 32 the algorithm reports the following squares (pairs $(root, pos)$): $t = 0$: (1, 2), (2, 3), (1, 6), (1, 8), (1, 10), (2, 11), (1, 14), (2, 15), (1, 18), (2, 19), (1, 22), (1, 24), (1, 26), (2, 27), (1, 30); $t = 1$: (4, 5), (3, 12), (3, 16), (4, 21); $t = 2$: (8, 9). In particular, the algorithm reports all squares in the Thue-Morse words.

In the analysis of the algorithm we use some combinatorics of primitive words. The *primitive root* of a word u is the shortest word w, such that $w^k = u$ for some positive integer k. We call a word u *primitive* if it equals its primitive root, otherwise it is called *non-primitive*. Primitive words admit a so-called *synchronizing property*, as given in the following Lemma, see [7].

Lemma 3 (Synchronizing property of primitive words). *A nonempty word is primitive if and only if it occurs as a factor in its square only as a prefix and a suffix.*

Theorem 4. *DetectPowers algorithm reports only primitively rooted powers in the word u.*

Proof. Obviously, all positions reported by the algorithm represent kth powers. Thus we only need to show that no non-primitively-rooted powers are reported.

Consider lines 7–12 of the algorithm, for some t and j. Assume that $pos \neq 0$. To conclude the proof of the theorem, it suffices to show that the word $w \overset{\text{def}}{=} u[pos\,.\,.\,j-1]$ is always primitive.

Assume to the contrary, that $w = v^m$, for some v and $m \geq 2$. Let $z = u[j\,.\,.\,j + 2^t - 1] = u[pos\,.\,.\,j + 2^t - 1]$. There are two cases (see Fig. 3):

a) Let us assume, that $|w| \geq 2^t$. Then z is a prefix of of w and $|v|$ is a period of z.
b) Let us assume, that $|w| < 2^t$. Then w is a prefix of z and $u[j\,.\,.\,j + 2^t - |w| - 1]$ is a border of z.

In both cases z appears also at position $j - |v|$. Hence, $Prev[Name_t[j]] \geq j - |v| > pos$, this contradiction concludes the proof. □

The following two theorems conclude that the algorithm correctly checks if the word contains any kth power (i.e., whether the word is kth-power-free or not), and also reports (among others) all kth powers of specific type, depending on the value of parameter k (2 or ≥ 3).

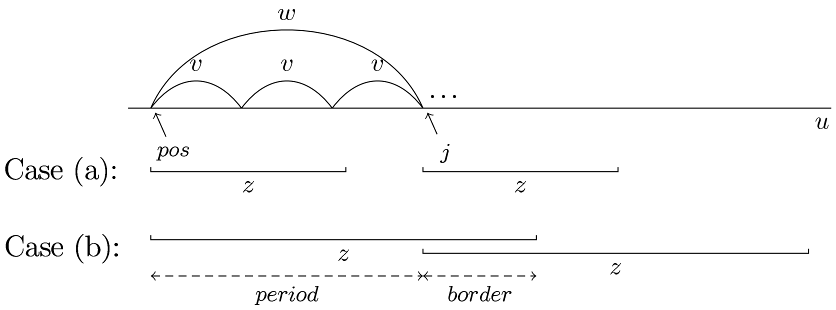

Figure 3. Illustration of the proof of Theorem 4; case (a): $|z| \leq |w|$, case (b): $|z| > |w|$.

Theorem 5. *For $k = 2$, the DetectPowers algorithm finds all occurrences of shortest squares in u.*

Proof. Let v^2 be any shortest square occurring in u at position i. Note that v must be primitive. Let s be such an integer, that $2^s \leq |v| < 2^{s+1}$. Consider the step of the algorithm in which $t = s$, $j = i + |v|$. We show that the algorithm reports the square v^2 in this step, i.e., $pos = i$. Obviously $pos \geq i$, hence it suffices to show that this value cannot be greater than i.

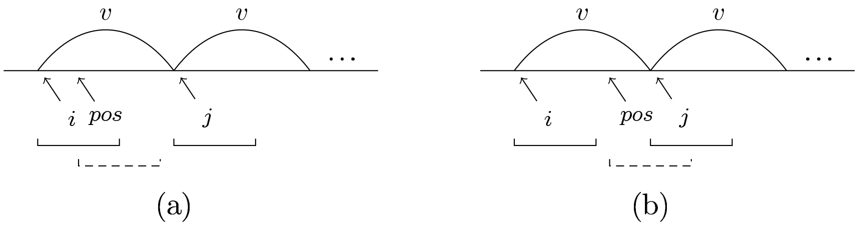

Figure 4. Illustration of the proof of Theorem 5. (a) If $pos - i < 2^t$ then the occurrences of w at positions i and pos overlap. (b) If $pos - i \geq 2^t$ then the occurrences of w at positions pos and j overlap.

If this was the case, the factor $w \stackrel{\text{def}}{=} u[j\,..\,j + 2^t - 1]$ would occur in u at positions i, pos and j, thus forming an overlap, see Fig. 4. However, an overlap of a string of length 2^t corresponds to a square in u with primitive root shorter than 2^t, what contradicts the fact that v^2 is the shortest square in u. □

Theorem 6. *For a given $k \geq 3$, the DetectPowers algorithm finds all occurrences of primitively rooted kth powers in u.*

Proof. Assume that there is an occurrence of v^k, for v primitive, which starts at position i in u. Let integer s be defined as $2^{s-1} < |v| \leq 2^s$. Let us consider the step of the algorithm in which $t = s$, $j = i + |v|$. We show that in this step $pos = i$, this concludes that the considered power is reported by the algorithm.

Let us note that $pos \geq i$, since $2^t < 2|v|$, and therefore:

$$u[i\,..\,i + 2^t - 1] \;=\; u[j\,..\,j + 2^t - 1] \;\stackrel{\text{def}}{=}\; w,$$

see Fig. 5a. We prove the inequality $pos \leq i$ by contradiction. Assume that $i < pos < j$. Then the prefix of w of length $|v|$, that is the word v, would occur in u at positions:

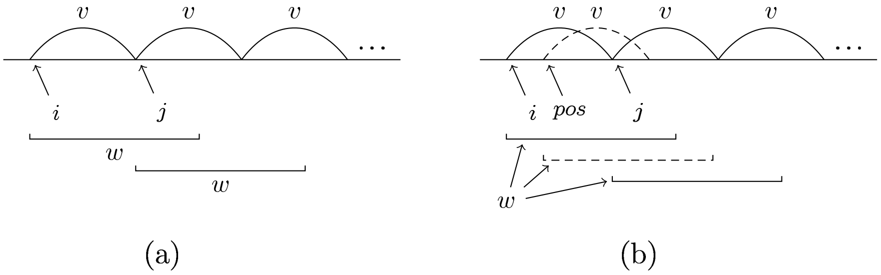

Figure 5. Illustration of the proof of Theorem 6

i, *pos* and j (see Fig. 5.b). This is not possible, however, due to the synchronizing property of primitive words (Lemma 3). □

Remark. It is easy to see that the stronger claim (from Theorem 6) does not hold in the case of $k = 2$ (Theorem 5). That is, not all primitively rooted squares are detected by the algorithm. Among others, the squares for which the primitive root admits a very long border, e.g., $((ab)^m a)^2$, may not be reported.
Finally, let us consider the complexity of the algorithm DetectPowers.

Theorem 7. *The time complexity of the DetectPowers algorithm is $O(n \log n)$. Moreover, for $k = O(1)$ the algorithm can be modified to require only $O(n)$ space and still satisfy the properties from Theorems 4–6.*

Proof. The analysis of the time complexity is straightforward — the outer loop of the algorithm makes $O(\log n)$ iterations and in each iteration the inner loop runs in $O(n)$ time.

The space complexity of the presented implementation is also $O(n \log n)$ due to the space requirements of the DBF, however it can be reduced to $O(n)$ if only two consecutive rows of the table *Name* are stored in the memory.

This causes a difficulty only in the kth-power-test in line 8, since the value of *root* can be arbitrary. However, along with the proofs of Theorems 5 and 6, we can immediately return $false$ in the test if the parameter *root* is not in the interval $[2^t, 2^{t+1})$ (for squares) or the interval $(2^{t-1}, 2^t]$ (for higher powers), and still the output of the algorithm will fulfill the requirements.

Thus the predicate reduces to testing equality of words of length $(k-1) \cdot root = c \cdot 2^{t-1}$, where $1 \le c = O(1)$ for $k = O(1)$, thus can be performed using only $Name_{t-1}[\,]$ in constant time (Lemma 2). □

4 Application of the DetectPowers Algorithm for Cubic Runs

In this section we show how to use the output of the DetectPowers algorithm to compute, in a simple manner, all cubic runs in a string u of length n in $O(n \log n)$ time. Cubic runs [9] are special type of runs in which the period is at least 3 times shorter than the run, hence they characterize strong periodic properties of a word.

Let L be the output of the DetectPowers algorithm for u and $k = 3$. It is a sorted list of pairs with repetitions removed. Moreover, without the loss of generality, we can assume, that it is sorted in ascending lexicographical order of pairs. Let us define a *special sublist* of L as a maximal continuous subsequence of L of the form (per, i), $(per, i+1)$, $\ldots$, $(per, i+s)$. Note, that such a sublist corresponds to a cubic run $(i,\ i+s+3\cdot per-1,\ per)$.

Example 8. For the following list of pairs $(root, pos)$:

$$L = (2,3),\ (2,4),\ (2,5),\ (4,8),\ (4,9),\ (4,28),\ (4,29),\ (4,30),\ (4,31),\ (5,18)$$

the corresponding cubic runs are:

$$(3,10,2),\ (8,20,4),\ (28,42,4),\ (18,32,5).$$

Thus we obtain:

Restriction 9. *There is a bijection between special sublists of L and cubic runs.*

The following algorithm scans the list of cubes and glues together its special sublists into cubic runs, utilizing Observation 9.

```
Algorithm DetectCubicRuns($u, n$)
 1: {list all cubic runs in the word $u$, $|u| = n$}
 2: $L \leftarrow$ DetectPowers($u, n, 3$)
 3: $L.append((-1,-1))$
 4: CRUNS $\leftarrow \emptyset$
 5: $prev\_root \leftarrow prev\_pos \leftarrow start \leftarrow -1$
 6: for all $(root, pos) \in L$ do
 7:     if $(root, pos) = (prev\_root, prev\_pos + 1)$ then
 8:         $prev\_pos \leftarrow pos$
 9:     else if $start \geq 0$ then
10:         CRUNS.$insert((start,\ prev\_pos + 3 \cdot prev\_root - 1,\ prev\_root))$
11:         $start \leftarrow prev\_pos \leftarrow pos$
12:         $prev\_root \leftarrow root$
13: return CRUNS
```

Theorem 10. *The DetectCubicRuns algorithm computes all cubic runs in a string u of length n in $O(n \log n)$ time.*

Proof. Due to Theorem 7, line 2 of the algorithm runs in $O(n \log n)$ time. The time complexity of the rest of the algorithm is $O(|L|+n)$, where $|L|$ denotes the number of elements in the list L. Time complexity of lines 6–13 is clearly $O(|L|)$. Finally, again due to Theorem 7, $|L| = O(n \log n)$, which yields $O(n \log n)$ total time complexity of the DetectCubicRuns algorithm.

Due to Theorems 4 and 6, the list L contains all occurrences of all primitively rooted cubes in the word u. In the for-all-loop (lines 6–12) the algorithm glues together cubes forming special sublists of L, thus forming the same cubic run. Hence, the output of the algorithm comprises exactly all the cubic runs in u. □

5 Detecting Runs

In this section we describe a different, however still very simple algorithm which reports all (ordinary) runs in a string u of length n in $O(n \log n)$ time. In the following pseudocode, in the for-loop we consider candidates for runs with period *per*. Verification of existence of runs is performed using *the longest common prefix* (*lcpref* in short) and *the longest common suffix* (*lcsuf* in short) queries, also called *longest common extension* queries (see also Fig. 6). Here, $lcpref(a,b)$ denotes the length of the longest common prefix of suffixes $u[a\,..\,n]$ and $u[b\,..\,n]$, similarly $lcsuf(a,b)$ denotes the length of the longest common suffix of prefixes $u[1\,..\,a]$ and $u[1\,..\,b]$.

The obtained list of candidates RUNS may contain the same run listed several times and additionally with periods being multiples of its shortest period. Therefore, in the end (lines 11–14) we remove such repetitions, leaving at most one triple (i,j,per) for given i,j, with the smallest corresponding value of period per.

```
Algorithm DetectRuns($u,n$)
 1: {list all runs in the word $u$, $|u| = n$}
 2: RUNS $\leftarrow \emptyset$
 3: for $per \leftarrow 1$ to $n$ div 2 do
 4:    $pos \leftarrow per$
 5:    while $pos + per \le n$ do
 6:       $left \leftarrow lcsuf(pos,\ pos + per)$
 7:       $right \leftarrow lcpref(pos,\ pos + per)$
 8:       if $left + right > per$ then
 9:          RUNS.$insert((pos - left + 1,\ pos + per + right - 1,\ per))$
10:       $pos \leftarrow pos + per$
11: RadixSort(RUNS) {triples sorted lexicographically}
12: $prev \leftarrow (-1,-1)$;
13: for all $(i,j,per) \in$ RUNS do
14:    if $prev = (i,j)$ then RUNS.$delete((i,j,per))$; else $prev \leftarrow (i,j)$
15: return RUNS
```

The following theorem shows correctness of the DetectRuns algorithm, i.e., that it computes exactly all distinct runs in a string.

Theorem 11.

(a) Each run (a,b,q) in the word u is inserted to the list RUNS (line 9) at least once.
(b) Every triple (a,b,p) inserted to RUNS in line 9 of the algorithm corresponds to a run (a,b,q) in u with $q \mid p$.

Proof. (a) Let $a+r$, for $0 \le r < q$, be any of the first q positions of the run. Then, by the definition of a run, the following inequalities hold, see Fig. 6:

$$\begin{aligned} lcsuf(a+r, a+r+q) &= r+1 \\ lcpref(a+r, a+r+q) &\ge q-r. \end{aligned}$$

Hence, if $per = q$ and $pos = a + r$ then the condition in line 8 of the algorithm is true and the run (a,b,q) is reported. However, this happens for $r \equiv -a \pmod{per}$, i.e., in the mth step of the while-loop, where $m = \lceil a/per \rceil$.

(b) Clearly any triple (a,b,p) inserted into the list RUNS in line 9 of the algorithm corresponds to an interval $[a\,..\,b]$ in u with period (not necessarily shortest) equal to p and repeating at least twice within the interval, i.e., $2p \le b-a+1$. Moreover, this interval is not extendable to either side without violating this periodicity.

Let q be the shortest period of the factor $u[a\,..\,b]$. Note that $q \mid p$, since otherwise, by Fine & Wilf's Periodicity Lemma [7,12], $\gcd(p,q)$ would be a shorter period of this factor. To show that $[a\,..\,b]$ is a run with period q, it suffices to prove that this interval is not extendable to either size with regard to the period q.

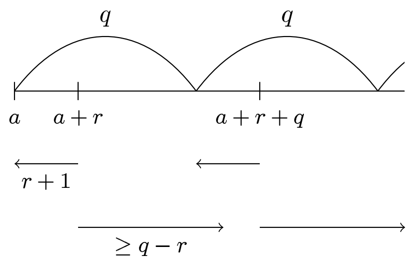

Figure 6. Graphical interpretation of $lcsuf(a{+}r, a{+}r{+}q)$ and $lcpref(a{+}r, a{+}r{+}q)$ for a run (a, b, q) and $0 \leq r < q$

Assume to the contrary that the interval is extendable to the left (the other case is analogical). Then we have:

$$u[a-1] \;=\; u[a-1+q] \;=\; u[a-1+q+(p/q-1)\cdot q] \;=\; u[a-1+p]$$

and consequently $[a\,..\,b]$ would be extendable to the left w.r.t. the period p, a contradiction. □

Now let us analyze the time complexity of the algorithm. It mostly depends on the time complexity of the *lcpref* and *lcsuf* queries. Their efficient implementation involves the Longest Common Prefix (LCP) and Suffix Arrays (SUF) (computed in $O(n)$ time), and Range Minimum Queries (RMQ) (with $O(n)$ preprocessing time and $O(1)$ query time) [10]. Techniques used in the efficient implementation of these data structures are rather complex. However, without increasing overall time complexity of the algorithm, we can compute the Suffix Array and preprocess RMQ in $O(n \log n)$ time. Hence, we can use much simpler machinery.

The Suffix Array of u can be computed in $O(n \log n)$ time using $\mathsf{DBF}(u)$ — all the suffixes are sorted lexicographically and can be compared in $O(1)$ time using $\mathsf{DBF}(u)$. Then, the LCP array can be simply computed in $O(n)$ time using the Suffix Array.

Preprocessing of RMQ data-structure in $O(n \log n)$ time resembles computation of DBF a lot. The main difference is that instead of computing ranks of factors we compute positions of minimal elements in ranges. Then, we can find a minimum in the given range by covering the given range by two ranges whose size is a power of two, and comparing their minimal elements. Hence, $O(1)$ query time is preserved.

Thus we obtain $O(n \log n)$ preprocessing time and $O(1)$ query time for the *lcpref* and *lcsuf* queries, what yields the time complexity of the algorithm specified in the following theorem.

Theorem 12. *The time complexity of the DetectRuns algorithm is $O(n \log n)$.*

Proof. For a given value of per, the while-loop performs at most n/per steps, each in constant time. The time complexity of the for-loop is therefore

$$O\left(\sum_{per=1}^{\lfloor n/2 \rfloor} \frac{n}{per}\right) = O(n \log n)$$

and this is also the maximum size of the list RUNS.

All remaining operations in the algorithm are: *lcpref*/*lcsuf* preprocessing which is performed in $O(n \log n)$ time, and sorting and removing duplicates from RUNS, both performed in $O(|\mathsf{RUNS}|) = O(n \log n)$ time. In total, we obtain the aforementioned time complexity of the algorithm. □

6 Acknowledgements

The authors thank Tomasz Kociumaka for the idea of the proof of Theorem 6.

References

1. A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words*, vol. F12 of NATO ASI Series, Springer-Verlag, 1985.
2. A. Apostolico and F. P. Preparata: *Optimal off-line detection of repetitions in a string.* Theor. Comput. Sci., 22 1983, pp. 297–315.
3. G.-H. Chen, J.-J. Hong, and H.-I. Lu: *An optimal algorithm for online square detection*, in CPM, A. Apostolico, M. Crochemore, and K. Park, eds., vol. 3537 of Lecture Notes in Computer Science, Springer, 2005, pp. 280–287.
4. M. Crochemore: *An optimal algorithm for computing the repetitions in a word.* Inf. Process. Lett., 12(5) 1981, pp. 244–250.
5. M. Crochemore: *Transducers and repetitions.* Theor. Comput. Sci., 45(1) 1986, pp. 63–86.
6. M. Crochemore, S. Z. Fazekas, C. S. Iliopoulos, and I. Jayasekera: *Bounds on powers in strings*, in Developments in Language Theory, M. Ito and M. Toyama, eds., vol. 5257 of Lecture Notes in Computer Science, Springer, 2008, pp. 206–215.
7. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on Strings*, Cambridge University Press, 2007.
8. M. Crochemore, L. Ilie, and W. Rytter: *Repetitions in strings: Algorithms and combinatorics.* Theor. Comput. Sci., 410(50) 2009, pp. 5227–5235.
9. M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen: *On the maximal number of cubic runs in a string*, in LATA, A. H. Dediu, H. Fernau, and C. Martín-Vide, eds., vol. 6031 of Lecture Notes in Computer Science, Springer, 2010, pp. 227–238.
10. M. Crochemore, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Walen: *Efficient algorithms for two extensions of LPF table: The power of suffix arrays*, in SOFSEM, J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, and B. Rumpe, eds., vol. 5901 of Lecture Notes in Computer Science, Springer, 2010, pp. 296–307.
11. M. Crochemore and W. Rytter: *Usefulness of the Karp-Miller-Rosenberg algorithm in parallel computations on strings and arrays.* Theor. Comput. Sci., 88(1) 1991, pp. 59–82.
12. M. Crochemore and W. Rytter: *Jewels of Stringology*, World Scientific, 2003.
13. A. S. Fraenkel and J. Simpson: *How many squares can a string contain?* J. of Combinatorial Theory Series A, 82 1998, pp. 112–120.
14. D. Gusfield and J. Stoye: *Linear time algorithms for finding and representing all the tandem repeats in a string.* J. Comput. Syst. Sci., 69(4) 2004, pp. 525–546.
15. L. Ilie and L. Tinta: *Practical algorithms for the longest common extension problem*, in SPIRE, J. Karlgren, J. Tarhio, and H. Hyyrö, eds., vol. 5721 of Lecture Notes in Computer Science, Springer, 2009, pp. 302–309.
16. J. Jansson and Z. Peng: *Online and dynamic recognition of squarefree strings*, in MFCS, J. Jedrzejowicz and A. Szepietowski, eds., vol. 3618 of Lecture Notes in Computer Science, Springer, 2005, pp. 520–531.
17. R. M. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in Proceedings of the 40th Symposium on Foundations of Computer Science, 1999, pp. 596–604.
18. R. M. Kolpakov and G. Kucherov: *On maximal repetitions in words.* J. of Discr. Alg., 1 1999, pp. 159–186.
19. S. R. Kosaraju: *Computation of squares in a string (preliminary version)*, in CPM, M. Crochemore and D. Gusfield, eds., vol. 807 of Lecture Notes in Computer Science, Springer, 1994, pp. 146–150.
20. G. M. Landau and J. P. Schmidt: *An algorithm for approximate tandem repeats*, in CPM, A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, eds., vol. 684 of Lecture Notes in Computer Science, Springer, 1993, pp. 120–133.
21. H.-F. Leung, Z. Peng, and H.-F. Ting: *An efficient online algorithm for square detection*, in COCOON, K.-Y. Chwa and J. I. Munro, eds., vol. 3106 of Lecture Notes in Computer Science, Springer, 2004, pp. 432–439.

22. M. G. Main and R. J. Lorentz: *An O(n log n) algorithm for finding all repetitions in a string.* J. Algorithms, 5(3) 1984, pp. 422–432.
23. M. G. Main and R. J. Lorentz: *Linear time recognition of squarefree strings*, in Apostolico and Galil [1], pp. 271–278.
24. M. O. Rabin: *Discovering repetitions in strings*, in Apostolico and Galil [1], pp. 279–288.
25. A. O. Slisenko: *Detection of periodicities and string matching in real time.* J. Soviet Math., 22 1983, pp. 1316–1386.
26. J. Stoye and D. Gusfield: *Simple and flexible detection of contiguous repeats using a suffix tree.* Theor. Comput. Sci., 270(1-2) 2002, pp. 843–856.