New Simple Efficient Algorithms Computing Powers and Runs in Strings

M. Crochemore[a,c], C. S. Iliopoulos[a,d], M. Kubica[e], J. Radoszewski[e,1,*], W. Rytter[e,f,2], K. Stencel[e,f], T. Waleń[b,e]

[a] *King's College London, London WC2R 2LS, UK*
[b] *Laboratory of Bioinformatics and Protein Engineering, International Institute of Molecular and Cell Biology in Warsaw, Poland*
[c] *Université Paris-Est, France*
[d] *Digital Ecosystems & Business Intelligence Institute, Curtin University of Technology, Perth WA 6845, Australia*
[e] *Dept. of Mathematics, Informatics and Mechanics, University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland*
[f] *Dept. of Math. and Informatics, Copernicus University, ul. Chopina 12/18, 87-100 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 (in particular, squares) in a string 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 obtain novel algorithmic solutions for several classical string problems which are much simpler than (usually quite sophisticated) linear time algorithms.

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

*Corresponding author. Tel.: +48-22-55-44-484, fax: +48-22-55-44-400. Some parts of this paper were written during the corresponding author's Erasmus exchange at King's College London.

Email addresses: maxime.crochemore@kcl.ac.uk (M. Crochemore), csi@dcs.kcl.ac.uk (C. S. Iliopoulos), kubica@mimuw.edu.pl (M. Kubica), jrad@mimuw.edu.pl (J. Radoszewski), rytter@mimuw.edu.pl (W. Rytter), stencel@mimuw.edu.pl (K. Stencel), walen@mimuw.edu.pl (T. Waleń)

[1] The author is supported by grant no. N206 568540 of the National Science Centre.
[2] The author is supported by grant no. N206 566740 of the National Science Centre.

1. Introduction

We present algorithms finding various types of repetitions in a string: powers (e.g., squares or cubes), cubic runs and runs. The k-th power of a string u, u^k, is simply composed of k juxtaposed occurrences of this string. The square and the cube are obviously the second and the third power. A string u is called periodic with the period p if $u_i = u_{i+p}$ holds for all i. A run is a periodic factor of the string u in which the shortest period repeats at least twice. A run must be maximal, i.e., if extend it by one letter its period increases. A cubic run is similar, but is has to contain at least 3 occurrences of the period. Finding powers and runs in a given string is a fundamental problem in text processing and has numerous applications: it occurs frequently in pattern matching, text compression and computational biology. A more detailed explanation of the motivation and related topics can be found in the survey [8].

Multiple algorithms for finding various kinds of repetitions in a string have already been presented. The largest part of the related literature deals with different approaches to searching for squares in a string. Most of the existing algorithms are rather complex. The first approach is to check if a string contains any square factor at all, or, otherwise, is square-free. Denote by n the length of the considered string. $O(n \log n)$ time algorithms for square-free testing are presented in [24, 25] (the latter one is randomized). The optimal $O(n)$ time algorithms are described in [5, 24].

For the problem of finding all distinct squares, linear time algorithms are known [10, 16, 19]. It is also known that the maximal number of distinct squares in a string is linear with respect to the length of the string: this number never exceeds $2n$ [15].

Another approach is to find all occurrences of primitively rooted squares in a string, that is, factors of the string in which the shortest period occurs exactly twice. A number of $O(n \log n)$ time algorithms reporting all such occurrences can be found in [2, 4, 20, 23, 26].

Due to the lower bound shown in [6] these algorithms are optimal.

Yet another approach is to report simply all occurrences of squares in a string. Denote the number of such occurrences by z, note that z could be $\Theta(n^2)$. Both $O(n \log n + z)$ time algorithms [21, 23, 26] and $O(n + z)$ time algorithms [16, 19] are known for this problem.

Finally, there are recent results related to on-line square detection (that is, when the letters of the string are given one by one), improving the time complexity from $O(n \log^2 n)$ [22] to $O(n \log n)$ [18] 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 k-th string power is presented. The algorithm utilizes a simple, yet powerful textual data structure called the Dictionary of Basic Factors. The algorithm also reports all occurrences of primitively rooted k-th 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 in a string of length n is $O(n \log n)$.

From the aforementioned literature, the papers [2, 4, 23] 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 [24] 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 the algorithm from [24] to detect all occurrences of primitively rooted higher powers.

In Section 4 we present an application of our power-detecting algorithm to find all cubic runs in a string. This algorithm also works 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, 23] and than the optimal $O(n)$ time algorithm [19]. The only non-trivial technique used in our algorithm is the Longest Common Extension function. It can be either implemented as described in [12, 17] — very efficiently, but using quite sophisticated machinery — or in a much simpler way, using the Dictionary of Basic Factors, which is sufficient to obtain $O(n \log n)$ time complexity.

This is a full version of the paper [9].

2. Preliminaries

We consider *strings* (words) over a bounded alphabet Σ. The empty string 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 $u[i] = u[i\,..\,i]$). Strings $u[1\,..\,i]$ are called prefixes of u, strings $u[i\,..\,n]$ are called suffixes of u, whereas strings 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 string $u = u_1 \dots u_n$ if $u_i = u_{i+p}$ holds for all i, $1 \le i \le n-p$. 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, 14].

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 $2p \le j-i+1$,
- 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.

We identify a run with a corresponding triple (i, j, p). A *cubic run* is a run $[i\,..\,j]$ for which the shortest period p satisfies $3p \le j-i+1$. Figure 1 shows all the runs and cubic runs in a sample string.

If $w^k = u$, where u and w are nonempty strings and k is a positive integer, then we say that u is the k-th power of the string w. A *square* (*cube*) is the 2nd (3rd) power of a string.

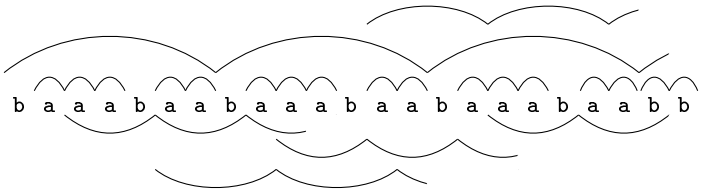

Figure 1: All runs that can be found in the string baaabaabaaabaabaaabaabb are indicated by wavy curves. There are exactly 4 cubic runs among them.

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
$Pair_0([\,],[+2^0])$	$(1,2)$	$(2,2)$	$(2,1)$	$(1,1)$	$(1,2)$	$(2,2)$	$(2,1)$	$(1,2)$	$(2,2)$	$(2,1)$	
$Name_1[\,]$	2	4	3	1	2	4	3	2	4	3	
$Pair_1([\,],[+2^1])$	$(2,3)$	$(4,1)$	$(3,2)$	$(1,4)$	$(2,3)$	$(4,2)$	$(3,4)$	$(2,3)$			
$Name_2[\,]$	2	5	3	1	2	6	4	2			
$Pair_2([\,],[+2^2])$	$(2,2)$	$(5,6)$	$(3,4)$	$(1,2)$							
$Name_3[\,]$	2	4	3	1							

Figure 2: An example of DBF computation for the string abbaabbabba. The factor abba appears three times in this string and is represented in $Name_2[\,]$ by 2.

2.1. Dictionary of Basic Factors

For a string u of length n, the *Dictionary of Basic Factors* of u [7, 14] (denoted by $\mathsf{DBF}(u)$) consists of a sequence of arrays $Name_t[\,]$, for $0 \le t \le \lfloor \log n \rfloor$. The array $Name_t[\,]$ contains information about factors of u of length 2^t, these factors are called the basic factors of u. The value of $Name_t[i]$ is the rank of $u[i\,..\,i+2^t-1]$ (in the lexicographic order) among all the factors of length 2^t. Hence, $u[i\,..\,i+2^t-1] \le u[j\,..\,j+2^t-1]$ if and only if $Name_t[i] \le Name_t[j]$. The elements of all the arrays $Name_t[\,]$ are in the range from 1 to n. DBF has a variety of known applications in the field of text and sequence algorithms, see, e.g., [13].

$\mathsf{DBF}(u)$ requires $O(n \log n)$ space and can be constructed in $O(n \log n)$ time [14]. $Name_0[\,]$ contains information about consecutive characters of u. Therefore $Name_0[\,]$ can be computed in $O(n \log n)$ time, by sorting all the characters appearing in u and mapping them to numbers from 1 on. Having computed $Name_t[\,]$, one can easily compute $Name_{t+1}[\,]$ in $O(n)$ time. The factor $u[i\,..\,i+2^{t+1}-1]$ is a concatenation of the 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 2 shows the DBF for an example string.

Using DBF, one can check equality of factors of arbitrary length, as given in the following lemma, see [14].

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

A small remark is necessary on finding the aforementioned number t. A naïve method would lead to the complexity $O(\log n)$. However, we can augment DBF with the array of values $t[1\,..\,n]$ such that for each $i = 1, 2, \ldots, n$, the inequality $2^{t[i]} \leq i < 2^{t[i]+1}$ holds. This array can be computed in linear time at the creation of DBF. Thus, it does not change the complexity of DBF construction.

Later on, we show that the memory complexity of the presented algorithm can be reduced to $O(n)$, although it uses $\mathsf{DBF}(u)$. For this, we cannot store all the arrays $Name_t[\,]$. Instead, we may store just a fixed number of such arrays, and design the algorithm in such a way, that the arrays 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 a constant number of factors of length 2^t. Hence, it suffices to compare $O(1)$ pairs of elements of $Name_t[\,]$.

Lemma 3 provides a useful tool for checking if a given factor of u is a power of a shorter string, with a given exponent.

Lemma 3. *Having precomputed* $\mathsf{DBF}(u)$*, one can check if a given factor of* u *has period* p *in* $O(1)$ *time.*

Proof. This test can be reduced to checking equality of two factors, since a factor of length ℓ has a period p if and only if it has a border of length $\ell - p$.

3. Detecting String Powers

Let u be a string of length n. The following DetectPowers algorithm tests if u contains a k-th power, for a given $k \geq 2$. If $k = 2$ then the algorithm reports

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.

all the occurrences of shortest squares in u (see the following Theorem 5). If $k \geq 3$ then a stronger property holds: its output consists of all the occurrences of primitively rooted k-th powers (this is due to Theorems 6 and 7).

The algorithm exploits $\mathsf{DBF}(u)$ and an auxiliary array of positions denoted as $Prev[1\,..\,n]$. Here $Prev[Name_t[j]]$ is the most recent occurrence of $u[j\,..\,j+2^t-1]$ preceding j, or -1 if there is no such previous occurrence.

The result is accumulated in a list POWERS. Each occurrence of a power of the form $(u[pos\,..\,pos+root-1])^k$ (i.e., the k-th 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 k-th powers. We allow the same occurrence of a power to be inserted multiple times — at the end the list is sorted and the repetitions are removed.

For each value of t, the algorithm scans the text from left to right. For each position j it finds the position pos of the previous occurrence of the factor $u[j\,..\,j+2^t-1]$ (using the arrays $Name_t[\,]$ and $Prev[\,]$). We then check if the factor $u[pos\,..\,j-1]$ repeats k consecutive times in u. This test can be done in constant time due to Lemma 3.

Examples of how the algorithm works can be found in Fig. 3 and Table 1.

Algorithm DetectPowers(u, k)
1: { detect k-th string powers in a string u, $|u| = n$ }
2: $Name \leftarrow \mathsf{DBF}(u)$
3: POWERS $\leftarrow \emptyset$
4: **for** $t \leftarrow 0$ **to** $\lfloor \log n \rfloor$ **do**
5: $\quad Prev \leftarrow (-1, -1, \ldots, -1)$
6: $\quad$ **for** $j \leftarrow 1$ **to** $n - 2^t + 1$ **do**
7: $\quad\quad name \leftarrow Name_t[j]$
8: $\quad\quad pos \leftarrow Prev[name]$
9: $\quad\quad root \leftarrow j - pos$
10: $\quad\quad$ **if** $pos \neq -1$ and $u[pos\,..\,pos + k \cdot root - 1]$ is (really) a k-th power
$\quad\quad\quad$ { constant time test due to DBF} **then**
11: $\quad\quad\quad$ POWERS.$append((root, pos))$
12: $\quad\quad Prev[name] \leftarrow j$
13: RadixSort(POWERS) with repetitions removed
14: **return** POWERS

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

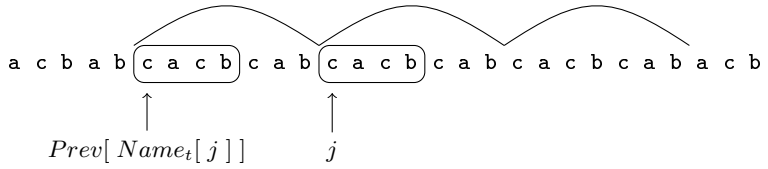

Figure 3: The basic factor `cacb` of rank $t = 2$ at position $j = 13$ generates a cube $(\texttt{cacbcab})^3$ starting at position $pos = 6$. The same cube is generated for $t = 3$ and $j = 13$, for the basic factor `cacbcabc`.

some positive integer k. Then, u is called a *primitively rooted* k-th power. 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 4 (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.*

The following two theorems conclude that the algorithm correctly checks if the string contains any k-th power (i.e., whether the string is k-th-power-free or not), and also reports all the occurrences of shortest squares (for $k = 2$) and all the occurrences of primitively rooted k-th powers for $k \geq 3$.

Theorem 5. *For $k = 2$, the DetectPowers algorithm finds all the 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., that $pos = i$. Obviously $pos \geq i$; it suffices to show that this value cannot be greater than i.

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, which contradicts the fact that v^2 is a shortest square in u, since $|v| \geq 2^t$ due to the choice of the parameter t.

Theorem 6. *For a given $k \geq 3$, the DetectPowers algorithm finds all the occurrences of primitively rooted k-th 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

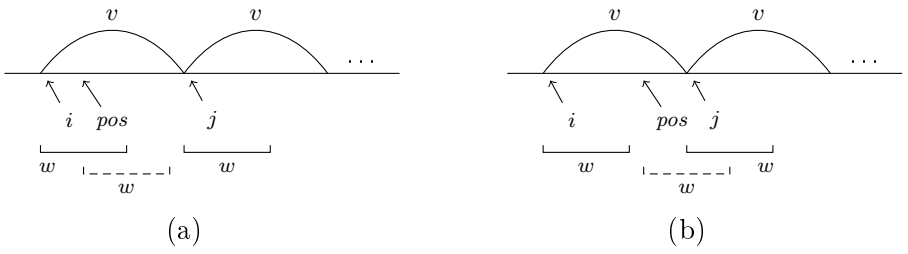

Figure 4: Illustration of the proof of Theorem 5. (a) If $pos - i < 2^t$ then the occurrences of w at positions i and pos overlap. (b) If $pos - i \geq 2^t$ then the occurrences of w at positions pos and j overlap.

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 occurrence of the power v^k is reported by the algorithm.

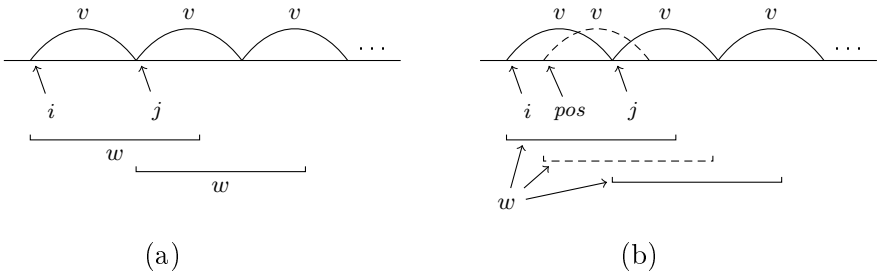

Figure 5: Illustration of the proof of Theorem 6.

Let us note that $2^t < 2\,|v|$, therefore:

$$u[i\,..\,i + 2^t - 1] \;=\; u[j\,..\,j + 2^t - 1]. \tag{1}$$

We denote the factor (1) as w (see Fig. 5a). From (1) we obtain that $pos \geq i$. 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: i, pos and j (see Fig. 5b). This is not possible, however, due to the synchronizing property of primitive words (Lemma 4).

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., $((\mathtt{ab})^m\mathtt{a})^2$, may not be reported.

We conclude the analysis of correctness of DetectPowers algorithm by showing that it only reports occurrences of primitively rooted powers.

Theorem 7. *The DetectPowers algorithm reports only primitively rooted powers in the string u.*

Proof. Obviously, all pairs reported by the algorithm represent k-th 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 $pos \neq -1$. To conclude the proof of the theorem, it suffices to show that the word $z \stackrel{\text{def}}{=} u[pos\,..\,j-1]$ is always primitive.

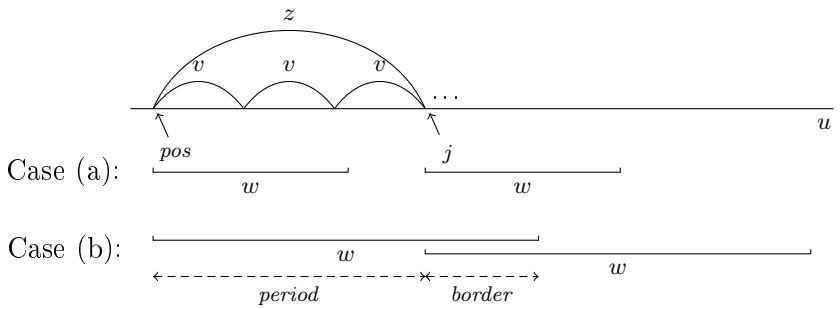

Figure 6: Illustration of the proof of Theorem 7; case (a): $|w| \leq |z|$, case (b): $|w| > |z|$.

Assume, to the contrary, that $z = v^m$, for some string v and integer $m \geq 2$. Let

$$w = u[j\,..\,j+2^t-1] = u[pos\,..\,pos+2^t-1].$$

There are two cases (see Fig. 6):

a) Let us assume that $|z| \geq 2^t$. Then w is a prefix of z, hence $|v|$ is a period of w.

b) Let us assume that $|z| < 2^t$. Then w has a border of length $2^t - |z|$. Hence, $|z|$ is a period of w, therefore $|v|$ is a period of w.

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

Finally, let us consider the complexity of the algorithm.

Theorem 8. *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 $O(n)$ additional space and still report the shortest k-th powers in u.*

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 at a time.

This causes a difficulty only in the k-th-power-test in line 10, since the value of *root* can be arbitrarily large or arbitrarily small comparing to 2^t. 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). Indeed, if the value of *root* does not belong to the corresponding range, exactly the same pair $(root, pos)$ would be reported for a different value of t which meets the interval constraints.

Thus, if $k = O(1)$, the predicate reduces to testing equality of words of length $(k-1) \cdot root = c \cdot 2^{t-1}$, where $c \geq 1$, $c = O(1)$, hence 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 rather straightforward manner, all the cubic runs in a string u of length n in $O(n \log n)$ time. Cubic runs [11] are a special type of runs in which the period is at least 3 times shorter than the run, hence they characterise strong periodic properties of a string.

Let L be the output of the DetectPowers algorithm for u and $k = 3$. It is a sorted list of pairs of the form $(root, pos)$, each denoting a k-th power of a factor of length *root*, starting at position *pos*, with repetitions removed. Let us define a *special sublist* of L as a maximal continuous subsequence of L of the form $(r, i),\ (r, i+1),\ \ldots,\ (r, i+s)$. Note that such a sublist corresponds to a cubic run $(i,\ i+s+3 \cdot r-1,\ r)$.

Example 9. *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, represented as triples (starting position, ending position, period), are:

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

Theorems 6 and 7 imply that L contains *all* primitively rooted cubes in u. Thus we conclude:

Observation 10. *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 10.

```
Algorithm DetectCubicRuns(u)
 1: { list all cubic runs in the string u, |u| = n }
 2: L ← DetectPowers(u, 3)
 3: L.append((−1, −1))
 4: CRUNS ← emptyList
 5: prev_root ← prev_pos ← start_pos ← −1
 6: for all (root, pos) ∈ L do
 7:    if (root, pos) = (prev_root, prev_pos + 1) then
 8:       prev_pos ← pos
 9:    else
10:       if start_pos ≥ 0 then
11:          CRUNS.append((start_pos, prev_pos + 3 · prev_root − 1, prev_root))
12:       start_pos ← prev_pos ← pos
13:       prev_root ← root
14: return CRUNS
```

Theorem 11. *The DetectCubicRuns algorithm finds all the cubic runs in a string u of length n in $O(n \log n)$ time.*

Proof. The correctness of the DetectCubicRuns algorithm is due to Observation 10. As for the complexity of the algorithm, due to Theorem 8, 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. Indeed, the time complexity of lines 6–12 is clearly $O(|L|)$. Finally, again due to Theorem 8, $|L| = O(n \log n)$, which yields $O(n \log n)$ total time complexity of the DetectCubicRuns algorithm.

5. Detecting Runs

In this section we describe another algorithm using the Dictionary of Basic Factors, which reports all (ordinary) runs in a string u of length n in $O(n \log n)$ time. The structure of all runs provides a succinct representation for all the repetitions in a string, since there are $O(n)$ runs in a string of length n [19], and it is conjectured [19] that the number of runs in a string of length n is always strictly less than n. Knowing all the runs in u, one can compute all distinct k-th powers in u (for any given $k \geq 2$) and find the local periods [14] in u in linear time in a much simpler way than computing the powers and local periods from scratch [10].

In the following pseudocode, in the for-loop in line 3 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 *the longest common extension* queries. Here, $lcpref(a, b)$ denotes the length of the longest common prefix of the suffixes $u[a \,..\, n]$ and $u[b \,..\, n]$, similarly $lcsuf(a, b)$ denotes the length of the longest common suffix of the prefixes $u[1 \,..\, a]$ and $u[1 \,..\, b]$.

Example 12. *Let* $u =$ `abaababaabaab`. *Then* $lcpref(1,4) = 3$, $lcpref(4,5) = 0$ *and* $lcsuf(5,10) = lcsuf(5,13) = lcsuf(10,13) = 5$.

```
Algorithm DetectRuns(u)
 1: { list all runs in the string u, |u| = n }
 2: RUNS ← emptyList
 3: for per ← 1 to ⌊n/2⌋ do
 4:    pos ← per
 5:    while pos + per ≤ n do
 6:       left ← lcsuf(pos, pos + per)
 7:       right ← lcpref(pos, pos + per)
 8:       if left + right > per then
 9:          RUNS.append((pos − left + 1, pos + per + right − 1, per))
10:       pos ← pos + per
11: RadixSort(RUNS) { triples sorted lexicographically }
12: prev ← (−1, −1);
13: for all (i, j, per) ∈ RUNS do
14:    if prev = (i, j) then
15:       RUNS.delete((i, j, per))
16:    else
17:       prev ← (i, j)
18: return RUNS
```

Let us analyze the conditions in lines 6–9 of the algorithm. Assume that for a given starting position *pos* and period *per*, the sum of *left* and *right*, where

$$left = lcsuf(pos,\ pos + per) \quad \text{and} \quad right = lcpref(pos,\ pos + per),$$

is greater than *per*. Thus, we know that the following two pairs of factors are equal:

$$\begin{aligned} u[pos - left + 1\,..\,pos] &= u[pos + per - left + 1\,..\,pos + per] \\ u[pos\,..\,pos + right - 1] &= u[pos + per\,..\,pos + per + right - 1], \end{aligned}$$

see also Fig. 7. This means that the factor

$$u[pos - left + 1\,..\,pos + per + right - 1] \tag{2}$$

of length $per + left + right - 1$ has a border of length $left + right - 1$, hence it has a period *per*. Since $left + right > per$, period *per* repeats within the factor (2) at least twice. This does *not* necessarily mean that this factor is a run with period *per*, however we will prove (Theorem 13(b)) that the factor (2) is a run with the shortest period being a divisor of *per*.

After we examine existence of a run for a given *pos* we advance by *per*. As we show in the following Theorem 13(a), this way during the execution of the while-loop from line 5 we will detect each run with the shortest period equal to *per*.

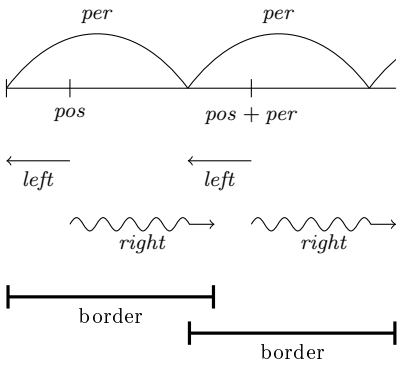

Figure 7: Detecting a run with period *per* using the longest extension queries. The equality of strings represented by straight arrows is implied by the value of *lcsuf*, while the equality of strings indicated by wavy arrows is a consequence of the value of *lcpref*. The combined straight and wavy arrows form a border of the considered factor (2).

Finally, the obtained list of candidates, denoted as RUNS, may contain the same run listed several times and additionally with periods being multiples of its shortest period. Therefore, in the end (lines 11–17) we remove such repetitions, leaving at most one triple (i, j, per) for given i, j, with the smallest corresponding value of period *per*. The following theorem shows correctness of the DetectRuns algorithm, i.e., that it computes exactly all the runs in a string.

Theorem 13.

(a) Each run (a, b, q) in the string u is inserted into the list RUNS (line 9) at least once.

(b) Every triple (a, b, p) inserted to the list 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 $0 \leq r < q$, be any of the first q positions of the run. Then, since the considered run has the period q, the following inequalities hold:

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

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. This happens in the m-th step of the while-loop, where $m = \lceil a/per \rceil$, for $r \equiv -a \pmod{per}$.

(b) As we have already argued, 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., $2p \leq b - a + 1$. Moreover, this interval is not extendible 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, 14], $\gcd(p, q)$ 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 extendible to either side with regard to the period q.

Assume to the contrary that the interval is extendible to the left (the other case is analogical). Then we would 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 extendible to the left w.r.t. the period p, a contradiction.

Now let us analyse the time complexity of the algorithm. It mostly depends on the time complexity of the *lcpref* and *lcsuf* queries. We will focus on *lcpref*, since the computation of *lcsuf* is symmetric. The efficient implementation of *lcpref* queries involves Suffix Arrays, computed in $O(n)$ time [7], and Range Minimum Queries (RMQ), with $O(n)$ preprocessing time and $O(1)$ query time [12]. The techniques used in the efficient implementation of these data structures are rather complex. However, without increasing the overall time complexity of the algorithm, we can compute the Suffix Arrays and preprocess Range Minimum Queries in $O(n \log n)$ time. For this we can use much simpler machinery. Let us first recall the notion of suffix arrays.

Suffix Arrays. The suffix array of the string u consists in three tables: SUF, LCP and RANK. The SUF array stores the list of positions in u sorted according to the increasing lexicographic order of suffixes starting at these positions, i.e.:

$$u[\mathsf{SUF}[1]\,..\,n] \;<\; u[\mathsf{SUF}[2]\,..\,n] \;<\; \ldots \;<\; u[\mathsf{SUF}[n]\,..\,n].$$

Thus, indices of SUF are ranks of the respective suffixes in the increasing lexicographic order. The LCP array is also indexed by the ranks of the suffixes, and stores the lengths of the longest common prefixes of consecutive suffixes in SUF. We set $\mathsf{LCP}[1] = -1$ and, for $1 < i \le n$, we define:

$$\mathsf{LCP}[i] = \mathit{lcpref}(\mathsf{SUF}[i-1], \mathsf{SUF}[i]).$$

Finally the RANK table is an inverse of the SUF array:

$$\mathsf{SUF}[\mathsf{RANK}[i]] = i \quad \text{for } i = 1, 2, \ldots, n.$$

As we already noted, all tables comprising the suffix array can be constructed in $O(n)$ time [7]. However, since we aim at $O(n \log n)$ overall complexity, we show simpler methods to compute the suffix array in $O(n \log n)$ time.

By Lemma 1 we know that using $\mathsf{DBF}(u)$ we can compare factors of u in $O(1)$ time. Therefore the SUF table for u can be simply computed by sorting suffixes using the comparison provided by $\mathsf{DBF}(u)$. Having SUF, we can compute the LCP table in $O(n \log n)$ time. For each pair of adjacent suffixes in SUF we

find their longest common prefix by binary search using $\mathsf{DBF}(u)$ and again Lemma 1. We start assuming that the longest common prefix is between 0 and M (maximum possible length). Then, when we know that the LCP is between b and e, we check in $O(1)$ time (using $\mathsf{DBF}(u)$) if the prefixes of length $\lceil (b+e)/2 \rceil$ are the same. The answer allows halving the interval. Finally, computation of the RANK table from the SUF table is straightforward.

As the last piece of the puzzle, we need an RMQ data-structure over LCP, since, as we show further, we need to answer queries for the minimum value in a subrange of the LCP table in $O(1)$ time. Preprocessing of RMQ data-structure in $O(n \log n)$ time resembles the computation of DBF. The main difference is that instead of computing ranks of factors we compute positions of minimal elements in ranges of length 2^t. Then, we can find a minimum in a given range by covering the range by two ranges whose size is a power of two, and comparing their minimal elements. Hence, $O(1)$ query time is preserved.

In order to compute $\mathit{lcpref}(i, j)$ we observe that the suffixes starting at the positions i and j are somehow located in SUF. Let x be $\min(\mathsf{RANK}[i], \mathsf{RANK}[j])$ and y be $\max(\mathsf{RANK}[i], \mathsf{RANK}[j])$. Then:

$$\mathit{lcpref}(i, j) = \min\{\mathsf{LCP}[x+1], \mathsf{LCP}[x+2], \ldots, \mathsf{LCP}[y]\},$$

provided that $i \neq j$, see [7]. However, this is exactly the RMQ query over the LCP array that can be answered in $O(1)$ time.

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

Theorem 14. *The time complexity of the DetectRuns algorithm is $O(n \log n)$.*

Proof. For a given value of *per*, the while-loop in line 5 performs at most n/per steps, each in constant time. The time complexity of the for-loop in line 3 is therefore

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

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, editors. *Combinatorial Algorithms on Words*, volume 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.*, 22:297–315, 1983.

[3] G.-H. Chen, J.-J. Hong, and H.-I. Lu. An optimal algorithm for online square detection. In A. Apostolico, M. Crochemore, and K. Park, editors, *CPM*, volume 3537 of *Lecture Notes in Computer Science*, pages 280–287. Springer, 2005.

[4] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.*, 12(5):244–250, 1981.

[5] M. Crochemore. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986.

[6] M. Crochemore, S. Z. Fazekas, C. S. Iliopoulos, and I. Jayasekera. Bounds on powers in strings. In M. Ito and M. Toyama, editors, *Developments in Language Theory*, volume 5257 of *Lecture Notes in Computer Science*, pages 206–215. Springer, 2008.

[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):5227–5235, 2009.

[9] M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, K. Stencel, and T. Walen. New simple effcient algorithms computing powers and runs in strings. In *Proceedings of the 15th Prague Stringology Conference, PSC*, pages 138–149, 2010.

[10] M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen. Extracting powers and periods in a string from its runs structure. In E. Chávez and S. Lonardi, editors, *SPIRE*, volume 6393 of *Lecture Notes in Computer Science*, pages 258–269. Springer, 2010.

[11] 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 A. H. Dediu, H. Fernau, and C. Martín-Vide, editors, *LATA*, volume 6031 of *Lecture Notes in Computer Science*, pages 227–238. Springer, 2010.

[12] 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 J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, and B. Rumpe, editors, *SOFSEM*, volume 5901 of *Lecture Notes in Computer Science*, pages 296–307. Springer, 2010.

[13] M. Crochemore and W. Rytter. Usefulness of the Karp-Miller-Rosenberg algorithm in parallel computations on strings and arrays. *Theor. Comput. Sci.*, 88(1):59–82, 1991.

[14] M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2003.

[15] A. S. Fraenkel and J. Simpson. How many squares can a string contain? *J. of Combinatorial Theory Series A*, 82:112–120, 1998.

[16] 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):525–546, 2004.

[17] L. Ilie and L. Tinta. Practical algorithms for the longest common extension problem. In J. Karlgren, J. Tarhio, and H. Hyyrö, editors, *SPIRE*, volume 5721 of *Lecture Notes in Computer Science*, pages 302–309. Springer, 2009.

[18] J. Jansson and Z. Peng. Online and dynamic recognition of squarefree strings. In J. Jedrzejowicz and A. Szepietowski, editors, *MFCS*, volume 3618 of *Lecture Notes in Computer Science*, pages 520–531. Springer, 2005.

[19] R. M. Kolpakov and G. Kucherov. On maximal repetitions in words. *J. of Discr. Alg.*, 1:159–186, 1999.

[20] S. R. Kosaraju. Computation of squares in a string (preliminary version). In M. Crochemore and D. Gusfield, editors, *CPM*, volume 807 of *Lecture Notes in Computer Science*, pages 146–150. Springer, 1994.

[21] G. M. Landau and J. P. Schmidt. An algorithm for approximate tandem repeats. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *CPM*, volume 684 of *Lecture Notes in Computer Science*, pages 120–133. Springer, 1993.

[22] H.-F. Leung, Z. Peng, and H.-F. Ting. An efficient online algorithm for square detection. In K.-Y. Chwa and J. I. Munro, editors, *COCOON*, volume 3106 of *Lecture Notes in Computer Science*, pages 432–439. Springer, 2004.

[23] M. G. Main and R. J. Lorentz. An O(n log n) algorithm for finding all repetitions in a string. *J. Algorithms*, 5(3):422–432, 1984.

[24] M. G. Main and R. J. Lorentz. Linear time recognition of squarefree strings. In Apostolico and Galil [1], pages 271–278.

[25] M. O. Rabin. Discovering repetitions in strings. In Apostolico and Galil [1], pages 279–288.

[26] J. Stoye and D. Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. *Theor. Comput. Sci.*, 270(1-2):843–856, 2002.