Extracting powers and periods in a string from its runs structure

Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen

Extracting Powers and Periods in a String from its Runs Structure

Maxime Crochemore[1,3], Costas Iliopoulos[1,4], Marcin Kubica[2], Jakub Radoszewski[2⋆], Wojciech Rytter[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,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. A breakthrough in the field of text algorithms was the discovery of the fact that the maximal number of runs in a string of length n is $O(n)$ and that they can all be computed in $O(n)$ time. We study some applications of this result. New simpler $O(n)$ time algorithms are presented for a few classical string problems: computing all distinct kth string powers for a given k, in particular squares for $k = 2$, and finding all local periods in a given string of length n. Additionally, we present an efficient algorithm for testing primitivity of factors of a string and computing their primitive roots. Applications of runs, despite their importance, are underrepresented in existing literature (approximately one page in the paper of Kolpakov & Kucherov, 1999). In this paper we attempt to fill in this gap. We use Lyndon words and introduce the Lyndon structure of runs as a useful tool when computing powers. In problems related to periods we use some versions of the Manhattan skyline problem.

1 Introduction

The structure of all runs in a string provides succinct and very useful information about periodic properties of the string. Several basic applications of this structure were given in [13]. We present some other algorithmic applications of runs and simplify already known algorithms.

First we consider the problem of computing all *distinct* kth powers in a string of length n, for a given k. It is a known fact that the number of distinct squares ($k = 2$) does not exceed $2n$ [8, 11, 12] and for cubes ($k = 3$) there is a $0.8n$ bound [14], which implies same bound for any value $k \geq 4$. Gusfield &

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

Stoye [10] present an $O(n)$ time algorithm for computing all the distinct squares. Unfortunately, this algorithm is complicated and uses suffix trees which are a rather heavyweight data structure and add a logarithmic factor depending on the size of alphabet in most implementations. We present a much simpler $O(n)$ time algorithm which computes all distinct kth powers in a string of length n using suffix arrays instead of suffix trees.

Another application of the runs structure is the computation of local periods which are related to the critical factorizations of a string [5]. The known $O(n)$ time algorithm by Duval et al. [6] employs several different techniques modified in a non-trivial way. We present an equally efficient but simpler algorithm using the solution of the Manhattan Skyline Problem.

Finally, we consider factor-primitivity queries, which consist in checking, for any factor of a given word, whether it is primitive and what is its primitive root. This problem has potential applications in data compression, in particular, in run-length encoding and its derivatives. We provide a solution to this problem with $O(n \log^{\epsilon} n)$ preprocessing time, for any $\epsilon > 0$, and $O(\log n)$ query time.

2 Preliminaries

Let u be a word of length n, $u = u[1\,..\,n]$, over a bounded alphabet Σ. We say that an integer p is the (shortest) *period* of $u[1\,..\,n]$ (notation: $p = \mathsf{per}(u)$) if p is the smallest positive integer such that $u[i] = u[i+p]$ holds for all $1 \le i \le n-p$.

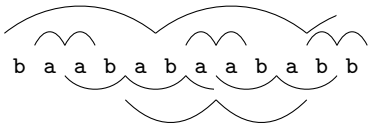

Fig. 1. The structure of runs in the word `baababaababb`. The word contains 3 runs with period 1, 2 runs with period 2, 1 run with period 3 and 1 run with period 5

A *run* v (a maximal repetition) in the word u is an interval $[i\,..\,j]$ such that the shortest period $p = \mathsf{per}(v)$ of the associated factor $u[i\,..\,j]$ satisfies $2p \le j - i + 1$, and the interval cannot be extended 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 letters exist. Denote by $\mathcal{R}(u)$ the set of all runs in u, each represented as a triple (i, j, p). It is known that $|\mathcal{R}(u)| = O(n)$ [4] and all elements of $\mathcal{R}(u)$ can be computed in $O(n)$ time [13] (a more practical algorithm for computing all runs is given in [2]).

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 a nonempty word. The *primitive root* of a word u, denoted $\mathsf{root}(u)$, is the shortest word w such that

$w^k = u$ for some positive integer k. We call a word u *primitive* if $\mathsf{root}(u) = u$, otherwise it is called *non-primitive*.

Let us recall two useful data structures in string processing.

Suffix Arrays. The suffix array of the word 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. Denote by $lcp(i,j)$ the length of the longest common prefix between $u[i\,..\,n]$ and $u[j\,..\,n]$ (for $1 \le i,j \le n$). Then, we set $\mathsf{LCP}[1] = -1$ and, for $1 < r \le n$, we have:

$$\mathsf{LCP}[r] = lcp(\mathsf{SUF}[r-1], \mathsf{SUF}[r]).$$

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

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

All tables comprising the suffix array can be constructed in $O(n)$ time [3].

Range Minimum Queries. Define the range minimum query data structure (RMQ, in short) as follows. Assume that we are given an array $A[1\,..\,n]$ of integers. This array is preprocessed to answer the following form of queries: for an interval $[a\,..\,b]$ (for $1 \le a \le b \le n$), find the minimum value $A[k]$ for $a \le k \le b$.

The best known RMQ data structures have $O(n)$ preprocessing time and $O(1)$ query time, using only $O(n)$ bits of space [7, 15]. The RMQ data structure on the LCP array enables the computation of longest common extensions, i.e., longest common prefixes between any two suffixes of a string in $O(1)$ time, with $O(n)$ time preprocessing.

3 Lyndon Representations of Runs

Let u be a word of length n. By $rot(u,c)$ let us denote a *cyclic rotation* of the word u obtained by moving (c mod n) first letters of u to its end. We say that the words u and $rot(u,c)$ are *cyclically equivalent*. A word that is both primitive and lexicographically minimal in the class of its cyclic rotations is called a *Lyndon word*. We define the *Lyndon root* of a word u, $\mathsf{lroot}(u)$, as the (only) Lyndon word cyclically equivalent to $\mathsf{root}(u)$. We define the Lyndon root of a run $v = (i,j,p)$ in u, $\mathsf{lroot}(v)$, as $\mathsf{lroot}(u[i\,..\,i+p-1])$, note that this notion is slightly different from the corresponding notion for words.

Denote by $u_{(a)}$ a prefix of the word u of length a and by $u^{(a)}$ a suffix of u of length a. Each run v can be uniquely represented (*Lyndon representation*) in the following form:

$$v \;\doteq\; \lambda^{(a)} \cdot \lambda^m \cdot \lambda_{(b)} \tag{1}$$

where $\lambda = \mathsf{lroot}(v)$ and $0 \le a, b < \mathsf{per}(v)$, see Fig. 2. We say that v is a λ-run. We will divide all runs of $\mathcal{R}(u)$ into maximal groups of λ-runs.

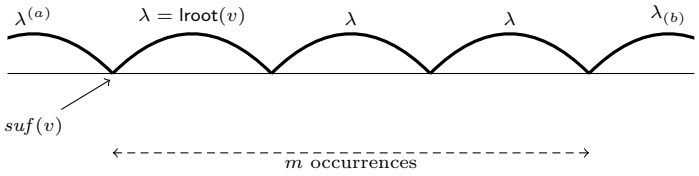

Fig. 2. A graphical view of the Lyndon representation of a run $v = \lambda^{(a)} \cdot \lambda^m \cdot \lambda_{(b)}$

For a run $v = (i, j, p)$, define $suf(v)$, $suf(v) \ge i$, as the smallest index for which:

$$u[suf(v) \,..\, suf(v) + p - 1] = \mathsf{lroot}(v),$$

see Fig. 2. This parameter, together with the period $\mathsf{per}(v)$, provides a unique characterization of the Lyndon root of the run. Additionally define $rank(v) = \mathsf{RANK}[suf(v)]$.

Lemma 1. *The values of $suf(v)$ and $rank(v)$ for any run v in a word u of length n can be computed in $O(1)$ time assuming $O(n)$ time preprocessing.*

Proof. Let $v = (i, j, p)$. The value of $rank(v)$ can be computed using RMQ on the interval $I = [i \,..\, i + p - 1]$ of the table RANK. Indeed, the prefixes of length p of the suffixes $\{u[d \,..\, n] \,:\, d \in I\}$ are exactly all cyclic rotations of $\mathsf{lroot}(v)$. Recall that RMQ for an array of length n can be implemented with $O(n)$ preprocessing time and $O(1)$ query time. Finally, $suf(v) = \mathsf{SUF}[rank(v)]$. □

Theorem 1. *The set $\mathcal{R}(u)$ of all runs within u can be decomposed into pairwise disjoint classes $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_t$ corresponding to runs with equal Lyndon roots in $O(n)$ time, where $n = |u|$.*

Proof. We start the proof of the theorem with the following claim.

Claim 2 *The equality of Lyndon roots of runs (represented as pairs of the form $(\mathsf{per}(v), rank(v))$) in u can be tested in $O(1)$ time with $O(n)$ preprocessing time. Moreover, if $L = v_1, v_2, \ldots, v_a$ is a list of all runs in u with period p sorted in ascending order of the values of parameter rank, then all runs in u with the same Lyndon root λ, $|\lambda| = p$, form a sublist of L composed of a number of consecutive elements.*

Proof. The Lyndon roots of two runs v_1 and v_2 are equal if and only if $\mathsf{per}(v_1) = \mathsf{per}(v_2)$ and the longest common prefix of suffixes at positions $suf(v_1)$ and $suf(v_2)$

is at least $\mathsf{per}(v_1)$. Recall that longest common prefixes of arbitrary suffixes can be computed using RMQ on the LCP array, which proves the first part.

As for the second part of the claim, assume that for three runs v_1, v_2 and v_3 we have $\mathsf{per}(v_1) = \mathsf{per}(v_2) = \mathsf{per}(v_3) = p$, $rank(v_1) < rank(v_2) < rank(v_3)$ and $\mathsf{lroot}(v_1) = \mathsf{lroot}(v_3)$. Then

$$lcp(suf(v_1), suf(v_3)) \ \geq \ p,$$

however due to the rank inequalities we have

$$lcp(suf(v_1), suf(v_3)) \ = \ \min(lcp(suf(v_1), suf(v_2)),\ lcp(suf(v_2), suf(v_3))).$$

Therefore

$$lcp(suf(v_1), suf(v_2)) \ \geq \ p$$

and consequently $\mathsf{lroot}(v_1) = \mathsf{lroot}(v_2) = \mathsf{lroot}(v_3)$. □

Using Claim 2, the requested decomposition of $\mathcal{R}(u)$ can be obtained in $O(n)$ time in the following three steps, recall that $|\mathcal{R}(u)| = O(n)$.

1. Compute the values of $suf(v)$ and $rank(v)$ for all runs in $\mathcal{R}(u)$ — $O(n)$ time in total due to Lemma 1.
2. Represent all runs v in u as pairs $(\mathsf{per}(v), rank(v))$, sort all such pairs lexicographically — $O(n)$ time using radix sort.
3. Group runs with equal Lyndon roots — due to Claim 2 the groups consist in consecutive runs in the sorted order of pairs, and equality of Lyndon roots of runs can be tested in $O(1)$ time with $O(n)$ time preprocessing, what gives $O(n)$ time complexity of this step. □

Define the *compact Lyndon representation* of a run $v = (i, j, p)$ as a tuple:

$$v \ \triangleq \ (i, j, p, a, m, b, \ell) \tag{2}$$

where ℓ is the length of v and a, m, b are defined as in the (ordinary) Lyndon representation (1). Due to the following lemma, the compact Lyndon representations of runs can be computed efficiently:

Lemma 3. *The compact Lyndon representation of runs (represented as (i, j, p)) in a word u of length n can be computed in $O(1)$ time with $O(n)$ time preprocessing.*

Proof. For a run $v = (i, j, p)$ of length $\ell = j - i + 1$, knowing the value of $suf(v)$ the compact Lyndon representation of v can be computed using the following additional formulas:

$$a = suf(v) - i, \qquad m = \lfloor (\ell - a)/p \rfloor, \qquad b = \ell - a - mp.$$

Hence, the statement is a consequence of Lemma 1. □

4 Inferring Powers from Runs

Denote by $\#powers(u,k)$ the total number of *distinct* kth powers in a string u. In this section we present an algorithm for efficiently computing this function as well as reporting the corresponding powers. By reporting we mean returning the vector $POWERS$ such that, for each i, $POWERS[i]$ is the set of periods of all kth powers which have the last occurrence starting at position i. These sets have cardinality at most two [8, 11, 12].

Each kth power w^k (for $k \geq 2$) occurring in u corresponds to a run v containing this occurrence for which $\mathsf{per}(v) = |\mathsf{root}(w)|$, we say that w^k is *induced* by the run. If $\mathsf{lroot}(w) = \lambda$ then we call w^k λ-*compatible*. Note that two runs may induce the same power only if their Lyndon roots are equal.

For a λ-run v define $maxpower(v)$ as the maximal natural β such that some cyclic rotation of $\lambda^{k\beta}$ is induced by v.

Observation 4 *If v is a run of length ℓ with period p then $maxpower(v) = \lfloor \ell/(kp) \rfloor$.*

The following lemma shows a correspondence between Lyndon representation of a run and the set of induced distinct kth powers.

Lemma 5. *Let v be a λ-run with period p and let $\beta = maxpower(v)$. Then all powers induced by v are:*

- *all cyclic rotations of $\lambda^{k\alpha}$ for $\alpha < \beta$*
- *cyclic rotations $rot(\lambda^{k\beta}, c)$ for $c \in \mathcal{I}(v)$, where $\mathcal{I}(v) \subseteq [0\,..\,p)$ is a union of at most two intervals.*

Proof. Let $v \doteq \lambda^{(a)} \cdot \lambda^m \cdot \lambda_{(b)}$ be a run of length ℓ with period p.

Note that for a given α the run v induces all cyclic rotations $rot(\lambda^{k\alpha}, c)$ for $c \in [p-a,\ p-a+\ell-kp\cdot\alpha]$. In particular, for $\alpha < \beta$, we obtain all distinct cyclic rotations, since $\ell - kp \cdot \alpha \geq p$. For $\alpha = \beta$, the aforementioned interval for the value of c must be treated modulo p and forms either a single subinterval of $[0\,..\,p)$ or a sum of at most two intervals $\mathcal{I}(v)$. For $\alpha > \beta$, no cyclic rotation of the word $\lambda^{k\alpha}$ is present in v, since $|\lambda^{k\alpha}| > |v|$. □

Let $maxruns(u,\lambda)$ be the set of λ-runs of u with maximal value of $maxpower(v)$. Denote by $\#powers_\lambda(u,k)$ the number of λ-compatible k-powers in u. The following lemma is a consequence of Lemma 5.

Lemma 6. *For a word u let $\beta(\lambda) = \max\{maxpower(v) \ : \ v \in \lambda\text{-}runs(u)\}$. Then*

$$\#powers_\lambda(u,k) \ = \ (\beta(\lambda)-1)\cdot|\lambda| + \left| \bigcup_{v \in maxruns(u,\lambda)} \mathcal{I}(v) \right|,$$

$$\#powers(u,k) \ = \ \sum_{\lambda} \#powers_\lambda(u,k).$$

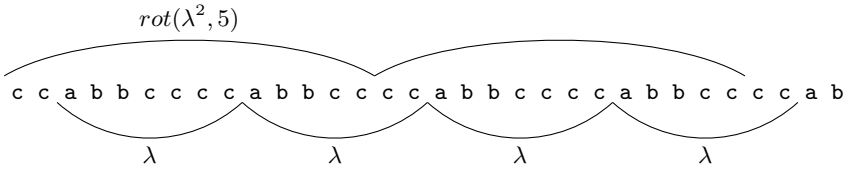

Fig. 3. The run $\lambda^{(2)}\lambda^4\lambda_{(2)}$ with the Lyndon root $\lambda = \texttt{abbcccc}$ induces all possible distinct squares cyclically equivalent to λ^2 and 5 squares cyclically equivalent to λ^4, that is, $maxpower(v) = 2$ and $\mathcal{I}(v) = [0\,..\,2] \cup [5\,..\,6]$

Theorem 2. *For a given word u of length n, the value $\#powers(u, k)$ can be computed and all distinct kth powers in u can be reported in $O(n)$ time.*

Proof. The value $\#powers(u, k)$ can be computed using the formulas from Lemma 6, assuming that we have the decomposition of $\mathcal{R}(u)$ from Theorem 1 and the compact Lyndon representations of all runs, which are necessary to compute the values of $\beta(\lambda)$ and $\mathcal{I}(v)$ (see the formulas in Lemma 5). The only difficulty is to find the size of the union of the sets $\mathcal{I}(v)$ for a given group of λ-runs $\mathcal{R}_y$ in $O(|\mathcal{R}_y|)$ time. Note that this can be performed in a simple way if the sets to be unionned form a list of intervals sorted in non-decreasing order (intervals treated as pairs). Due to Lemma 5, each set $\mathcal{I}(v)$ can be divided into a constant number of intervals. Finally, all intervals across all the groups $\mathcal{R}_y$ can be sorted using radix sort in $O(n)$ time.

The algorithm reporting all powers is a natural extension of the algorithm computing $\#powers(u, k)$ using the exact formulas from Lemma 5, we omit the technical description of the algorithm in this version of the paper. □

Denote by $\#occ\text{-}powers(u, k)$ the total number of *occurrences* of kth powers in a string u. We end this section presenting a formula for $\#occ\text{-}powers(u, k)$ which can be evaluated in a straightforward manner to obtain an $O(n)$ time algorithm, where $n = |u|$. Note that the value of the formula can be $\Theta(n^2)$.

Theorem 3.

$$\#occ\text{-}powers(u, k) = \sum_{(i,j,p)\in\mathcal{R}(u)} c(i, j, p) \cdot (j - i + 2 - kp/2) - c(i, j, p)^2 \cdot kp/2$$

where

$$c(i, j, p) = \left\lfloor \frac{j - i + 2}{kp} \right\rfloor. \tag{3}$$

The proof of the theorem will be included in the full version of the paper.

5 Computation of Local Periods

By $P = \{p_1, p_2, \ldots, p_{n-1}\}$ we denote the set of inter-positions that are located *between* pairs of consecutive letters of $u[1\,..\,n]$. We say that a square ww is centered at inter-position p_i of u if both of the following conditions hold, for $x = u[1\,..\,i]$ and $y = u[i+1\,..\,n]$:

- w is a suffix of x or x is a suffix of w
- w is a prefix of y or y is a prefix of w.

We define the *local period* at inter-position p_i (notation: $\mathsf{localper}[i]$) as $|w|$, where ww is the shortest square centered at this inter-position, see also Fig. 4. Clearly, for any p_i there are three possible cases:

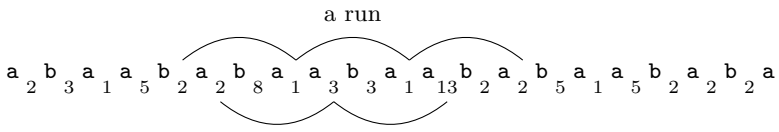

Fig. 4. A Fibonacci string with local periods at all its inter-positions. Local period at inter-position p_9 of the string is 3, since the smallest period q of a run which completely covers the factor of the string corresponding to the interval $[9-q+1\,..\,9+q]$ equals 3

Case A: $|w| \leq \min(|x|, |y|)$, i.e., ww is an *internal* square of u.

Case B: $\min(|x|, |y|) < |w| \leq \max(|x|, |y|)$, i.e., ww is a *left-external* square (if $|w| > |x|$) or a *right-external* square (if $|w| > |y|$).

Case C: $\max(|x|, |y|) < |w|$, i.e., ww is a *both-sides-external* square.

We handle Cases A-C separately. In Case A we use the structure of runs in u and perform a reduction to the Manhattan Skyline Problem. In Cases B and C we use the border array from the Morris-Pratt algorithm, which is a simple alternative to a modified Boyer-Moore shift function used for this purpose in [6].

Case A: internal local periods. The problem of the internal local periods can be reduced in $O(n)$ time to the (restricted min-version) of the following problem:

Restricted Manhattan Skyline Problem

Input:
given a set $\mathcal{S}$ of $O(n)$ subintervals of $[1\,..\,n-1]$ with natural heights of size $O(n)$;

Output:
the table $f[t] = \min\{height([i\,..\,j]) : t \in [i\,..\,j],\ [i\,..\,j] \in \mathcal{S}\}$, $t \in [1\,..\,n-1]$.

Indeed, note that any internal local period corresponds to a primitively rooted square in u, induced by one of the runs of u, see also Fig. 4. Each run $v = (a, b, q)$ in u induces such squares with root q at inter-positions $p_{a+q-1}, p_{a+q}, \ldots, p_{b-q}$. Thus for each inter-position p_i we need to find the shortest period of a run (i.e., height of an interval from the Manhattan Skyline Problem) inducing a square at this inter-position.

The following two lemmas show how to utilize the described reduction to construct a linear time algorithm for computing internal local periods.

Lemma 7. *Assume initially $X = \emptyset$ and all considered intervals $[i\,..\,j]$ are from the universe $[1\,..\,m]$. Then the sequence of $O(m)$ pairs of operations:*

$$\{ \quad \textit{list-all-elements}([i\,..\,j] \setminus X); \quad X \leftarrow X \cup [i\,..\,j]; \quad \} \tag{4}$$

can be implemented in $O(m)$ time.

Proof. The implementation uses a restricted version of the find/union data structure, in which we are allowed to union only adjacent subintervals. Thus the *structure* of union operations forms a static tree (here it is a path graph) and therefore $O(m)$ find/union operations can be performed in $O(m)$ time [9].

In the algorithm the universe $[1\,..\,m+1]$ (extended to the right by a sentinel) is partitioned into maximal segments of elements of X followed by a single element which is not in X: all elements in such a segment form a single find/union component which stores the index of its rightmost position. The operations (4) are implemented by traversing the components intersecting the interval $[i\,..\,j]$, reporting their rightmost elements and unionning them one by one. □

Lemma 8. *The internal local periods can be computed in linear time.*

Proof. We showed that the problem can be reduced to the restricted Manhattan Skyline Problem. This problem can be solved in $O(n)$ time as follows.

```
Sort intervals from S according to their heights (in increasing order);
Initialize X = ∅;
for each interval [i..j] ∈ S (in the sorted order) do
    for each t ∈ list-all-elements([i..j] \ X) do
        f[t] ← height([i..j]);
    X ← X ∪ [i..j];
```

According to Lemma 7, the set operations in the above pseudocode can be implemented in linear time. This completes the proof. □

Case B: one-side-external local periods. Recall that a word that is both a prefix and a suffix of a word u is called a *border* of the word u; a border of u is called proper if it is shorter than u. Denote by $\mathsf{border}[i]$, for $i = 1, 2, \ldots, n$, the length of the longest proper border of $u[1\,..\,i]$. Recall that the border array can be computed in $O(n)$ time, as in the Morris-Pratt algorithm [5].

The following lemma shows how the border array can be used to compute left-external local periods, the case of right-external local periods is symmetric and can be treated similarly by considering the reversed word u. The proof of the lemma will be present in the full version of the paper.

Lemma 9.

(a) If the local period at inter-position p_i is left-external (and not right external) then there exists $j > i$ such that $\mathsf{border}[j] = i$ and $\mathsf{localper}[i] = j - i$.
(b) If $\mathsf{border}[j] = i$ for any $j = 2, 3, \ldots, n$ and $i > 0$ then $\mathsf{localper}[i] \leq j - i$.

Due to Lemma 9, the localper array can be updated in $O(n)$ time by considering all left-external local periods corresponding to the values $\mathsf{border}[j]$ for all $j = 1, 2, \ldots, n$.

Case C: both-sides-external local periods. Consider a both-sides-external local period at inter-position p_i of u. If b is the longest overlap between $u[i{+}1\,..\,n]$ and $u[1\,..\,i]$, i.e., the longest suffix of the former word which is also a prefix of the latter word, then $\mathsf{localper}[i] = n - b$, see Fig. 5. Note that b is the length of the longest border of u which is not longer than $\min(i, n - i)$.

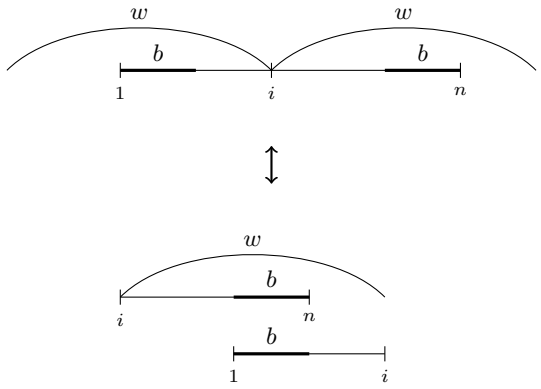

Fig. 5. The correspondence between both-sides-external local periods and borders

Recall that the lengths of all proper borders of u are iterations of the form $\mathsf{border}^{(j)}[n]$. This concludes an $O(n)$ time algorithm which updates the localper array obtained after the previous cases considering all both-sides-external local periods, filling the array from its middle to its sides.

Combining the solutions to Cases A-C, we obtain the following result.

Theorem 4. *All local periods of a string u of length n can be computed in $O(n)$ time (in a simple way) using the runs structure of u and the border array.*

6 Factor-Primitivity Queries

For a given string u of length n, we define a *factor-primitivity query* as follows: for the indices a, b, $1 \leq a \leq b < n$, check whether the factor $u[a\,..\,b]$ is primitive, and if not, find the length of its primitive root. Let us introduce a notion relating runs with factor-primitivity queries. We say that a run (i, j, p) *completely covers* an occurrence of a factor $u[a\,..\,b]$ in u if $i \leq a, b \leq j$.

Lemma 10. *Let p be the minimum period of a run completely covering an occurrence of a factor w in a string u (or $p = \infty$ if no such run exists). If $p < |w|$ and $p \mid |w|$ then $|\mathsf{root}(w)| = p$; otherwise w is primitive.*

Proof. Assume first that $q \stackrel{\text{def}}{=} |\mathsf{root}(w)| < |w|$. Then also $\mathsf{per}(w) = q$, see [5]. Hence, w is completely covered by a run with period q and, obviously, by no run with period smaller than q.

On the other hand, if $|\mathsf{root}(w)| = |w|$ then any run completely covering w and having period p satisfies $p = |w|$ or $p \nmid |w|$. This concludes the proof. □

The conclusion of Lemma 10 can again be interpreted using the notion of Manhattan skyline, see Fig. 6.

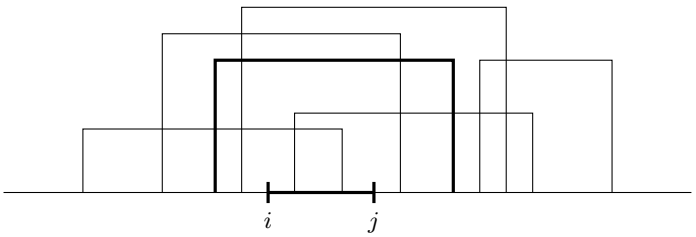

Fig. 6. The buildings in the skyline correspond to runs in a string u and their heights correspond to their periods. When checking primitivity of a factor $w = u[i\,..\,j]$ we look for the lowest building such that w is completely "under its roof"

In our algorithm we utilize yet another interpretation of the problem. To each run (i, j, p) in a word u ($|u| = n$) we assign a point (i, j) in the 2-dimensional plane, and define the value of this point as $f((i, j)) \stackrel{\text{def}}{=} p$. Denote the set of all such points by V. By Lemma 10, to find the primitive root of any factor $u[a\,..\,b]$ of u, it suffices to compute the value

$$\min\{f((i, j)) \ : \ 1 \leq i \leq a,\ b \leq j \leq n,\ (i, j) \in V\}.$$

This is exactly a *2D range search for minimum* query, which can be answered in the RAM model in: $O(\log^{1+\epsilon} m)$ query time with $O(m)$ preprocessing time, $O(\log m \log\log m)$ query time with $O(m \log\log m)$ preprocessing time, or $O(\log m)$ query time with $O(m \log^{\epsilon} m)$ preprocessing time, where $m = |V| = |\mathcal{R}(u)|$ and ϵ is an arbitrary positive real [1]. Thus we obtain the next result.

Theorem 5. *For a given string u of length n, using the runs structure of u we can answer factor-primitivity queries in $O(n \log^{\epsilon} n)$ preprocessing time, for any $\epsilon > 0$, and $O(\log n)$ query time.*

References

1. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988.
2. G. Chen, S. J. Puglisi, and W. F. Smyth. Fast and practical algorithms for computing all the runs in a string. In B. Ma and K. Zhang, editors, *CPM*, volume 4580 of *Lecture Notes in Computer Science*, pages 307–315. Springer, 2007.
3. M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings.* Cambridge University Press, 2007.
4. M. Crochemore, L. Ilie, and W. Rytter. Repetitions in strings: Algorithms and combinatorics. *Theor. Comput. Sci.*, 410(50):5227–5235, 2009.
5. M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2003.
6. J.-P. Duval, R. Kolpakov, G. Kucherov, T. Lecroq, and A. Lefebvre. Linear-time computation of local periods. *Theor. Comput. Sci.*, 326(1-3):229–240, 2004.
7. J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In B. Chen, M. Paterson, and G. Zhang, editors, *ESCAPE*, volume 4614 of *Lecture Notes in Computer Science*, pages 459–470. Springer, 2007.
8. A. S. Fraenkel and J. Simpson. How many squares can a string contain? *J. of Combinatorial Theory Series A*, 82:112–120, 1998.
9. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC)*, pages 246–251, 1983.
10. 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.
11. L. Ilie. A simple proof that a word of length n has at most $2n$ distinct squares. *J. of Combinatorial Theory Series A*, 112:163–164, 2005.
12. L. Ilie. A note on the number of squares in a word. *Theoretical Computer Science*, 380:373–376, 2007.
13. R. M. Kolpakov and G. Kucherov. On maximal repetitions in words. *J. of Discr. Alg.*, 1:159–186, 1999.
14. M. Kubica, J. Radoszewski, W. Rytter, and T. Walen. On the maximal number of cubic subwords in a string. In J. Fiala, J. Kratochvíl, and M. Miller, editors, *IWOCA*, volume 5874 of *Lecture Notes in Computer Science*, pages 345–355. Springer, 2009.
15. K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007.