CHAPTER 8

Periodic Structures in Words

8.0. Introduction

Repetitions (periodicities) in words are important objects that play a fundamental role in combinatorial properties of words and their applications to string processing, such as compression or biological sequence analysis. Using properties of repetitions allows one to speed up pattern matching algorithms.

The problem of efficiently identifying repetitions in a given word is one of the classical pattern matching problems. Recently, searching for repetitions in strings received a new motivation, due to the biosequence analysis. In DNA sequences, successively repeated fragments often bear an important biological information and their presence is characteristic for many genomic structures (such as telomer regions for example). From a practical viewpoint, satellites and alu-repeats are involved in chromosome analysis and genotyping, and thus are of major interest to genomic researchers. Thus, different biological studies based on the analysis of tandem repeats have been done, and even databases of tandem repeats in certain species have been compiled.

In this chapter, we present a general efficient approach to computing different periodic structures in words. It is based on two main algorithmic techniques – a special factorization of the word and so-called longest extension functions – described in Section 8.3. Different applications of this method are described in Sections 8.4, 8.5, 8.6, 8.7, and 8.8. These sections are preceded by section 8.2 devoted to a combinatorial enumerative properties of repetitions. Bounding the maximal number of repetitions is necessary for proving complexity bounds of corresponding search algorithms.

8.1. Definitions and preliminary results

Consider a word $w = a_1 \cdots a_n$. A *position* w is any integer ℓ with $1 \leq \ell \leq n$. Any word x such that there exist $i \leq j$ with $x = a_i \cdots a_j$ is called a *factor* of w. If a specific pair of integers (i, j) with this property is meant, we speak about an *occurrence* of the factor x. We say that this occurrence starts at position i and ends at position j in w and denote it $w[i..j]$. We say that a factor occurrence $v = w[i..j]$ *contains* a position ℓ of w, if $i \leq \ell \leq j$.

Recall (see Chapter 1) that an integer p is called *a period* of w if $a_i = a_{i+p}$, for all i such that $1 \leq i, i+p \leq n$. Equivalently, p is a period of w iff $w[1..n-p] = w[p+1..n]$. If p is a period of w, any factor u of w of length p is called a *root* of w. In other words, u is a root of w iff w is a factor of u^n for some natural n.

Each word w has a *minimal period* that we will denote $p(w)$. The roots u of w such that $|u| = p(w)$ are called *cyclic roots*. We also call the cyclic root $w[1..p(w)]$ the *prefix cyclic root* of w, and the cyclic root $w[n - p(w) + 1..n]$ the *suffix cyclic root* of w.

The rational number $e(w) = |w|/p(w)$ is called *the exponent* of w. If $e(w) \geq 2$, then w is called a *repetition* (*periodicity*). If $k = e(w)$ is an integer greater than 1, w can be written as $u^k = uu \cdots u$ (k times) and is called an *integer power* (k-power, or *tandem array* in biological literature). A word which is not an integer power is called *primitive*. An integer power of even exponent is commonly called a *square* (or a *tandem repeat*). These are words of the form uu for some word u. An integer power of exponent 2 is called a *primitively-rooted square*, as it corresponds to a square uu where u is primitive. In general, any word w of minimal period p and exponent e can be written as $u^k v$, where u is a primitive word, $|u| = p$, v is a proper prefix of u and $e = k + \frac{|v|}{|u|}$.

The following proposition specifies some properties of repetitions.

Proposition 8.1.1. *A word r of length m is a repetition of minimal period $p \leq m/2$ if and only if one of the following conditions holds:*

(i) $r[1..m-p] = r[p+1..m]$, and p is the minimal number with this property,

(ii) any factor of r of length $2p$ is a square, and p is the minimal number with this property.

From now on, we will be interested in repetitions occurring as factors of some word, that is in factor occurrences $r = w[i..j]$ with $e(r) \geq 2$. A *maximal repetition* in a word w, is a repetition $r = w[i..j]$ such that
(i) if $i > 1$, then $p(w[i-1..j]) > p(w[i..j])$, and
(ii) if $j < n$, then $p(w[i..j+1]) > p(w[i..j])$.
In other words, a maximal repetition is a repetition $r = w[i..j]$ such that no factor of w that contains r as a proper factor has the same minimal period as r. For example, the factor 10101 in the word $w = 1011010110110$ is a maximal repetition (with period 2), while the factor 1010 is not. Other maximal repetitions of w are prefix 10110101101 (period 5), suffix 10110110 (period 3), prefix 101101 (period 3), and the three occurrences of 11 (period 1). (Note that different repetitions can be equal words.)

Any repetition in a word can be extended to a unique maximal repetition, that we will call the *corresponding* maximal repetition. For example, the repetition 1010 in word $w = 1011010110110$ corresponds to the maximal repetition 10101 obtained by the extension by one letter to the right.

A basic result about periods is the Fine and Wilf's theorem:

Theorem 8.1.2 (Fine and Wilf). *If w has periods p_1, p_2, and $|w| \geq p_1 + p_2 - gcd(p_1, p_2)$, then $gcd(p_1, p_2)$ is also a period of w.*

Two factor occurrences $w[i..j]$ and $w[k..\ell]$ are said to *overlap* if their intervals $[i..j]$ and $[k..\ell]$ have a non-empty intersection. The *overlap* of the two factors is then the factor $w[r..s]$ where $[r..s]$ is the intersection of $[i..j]$ and $[k..\ell]$. The following lemma will be used in the sequel.

Lemma 8.1.3. *Two distinct maximal repetitions with the same period p cannot have an overlap of length greater than or equal to p.*

Proof. From a case analysis of relative positions of two repetitions of period p, it follows that if they intersect on at least p letters, at least one of them is not maximal. ∎

A repetition r occurring in a word w is said to *have a root* in some factor occurrence of w, if r overlaps with this occurrence by at least $p(r)$ letters. Also, we say that a repetition r *has a root on the right (respectively on the left) of a position ℓ of w* with the meaning that r overlaps by at least $p(r)$ letters with the suffix $w[\ell+1..n]$ (respectively, the prefix $w[1..\ell-1]$).

The following reformulation of Lemma 8.1.3 will be also useful.

Corollary 8.1.4. *If two maximal repetitions of w contain a position ℓ and have a root on the right (on the left) of ℓ, then either these maximal repetitions have different minimal periods or they coincide.*

We will also use the following proposition which is again a consequence of the Fine and Wilf theorem.

PROPOSITION 8.1.5. *If u is a primitive word, then u cannot be an internal factor of uu (that is, a factor which is not a prefix or suffix).*

Proof. If u is an internal factor of uu, then $u = xy = yx$, where x, y are nonempty words. We show that x and y are powers of the same word, by induction on the length of xy. The claim holds if $|x| = |y|$. Otherwise, assume $|x| > |y|$. Then $x = yz$ for some nonempty word z. It follows that $zy = yz$. By induction, y and z are powers of the same word, and so are x and y. Thus, u is not primitive. ∎

8.2. Counting maximal repetitions

Before considering algorithmic issues related to repetitions, we briefly study in this section the combinatorial question of the number of repetitions occurring in a word. The main point here is to show that considering maximal repetitions leads to a compact linear-space representation of all repetitions. This result will motivate building an efficient algorithm for computing all maximal repetitions, and, on the other hand, will be used to obtain bounds on its algorithmic complexity.

8.2.1. Counting squares: an upper bound

We start with the question of how many square occurrences can a word contain. Clearly, if all squares are counted, a word can contain a quadratic number of those (e.g. 1^n). If we restrict our attention to primitively-rooted squares, the following result holds.

THEOREM 8.2.1. *The number of occurrences of primitively-rooted squares in a word of length n is $O(n \log n)$.*

The proof is based on the following "lemma of three squares":

LEMMA 8.2.2. *Consider three squares x^2, y^2, z^2 and assume that z^2 is a prefix of y^2 and y^2 is a prefix of x^2. Assume that z is a primitive word. Then $|z| < |x|/2$.*

Proof. Denote $w = x^2$. Assume that $|z| \geq |x|/2$. Let $p = |y| - |z|$ and $k = |z| - p = 2|z| - |y|$. Observe that prefix $w[1..k]$ of z is also a suffix of z, and therefore $w[1..k] = w[p+1..k+p] = w[p+1..|z|]$. We show by induction that for every i, $k+1 \leq i \leq k+p$, we have $w[i] = w[i+p]$. We have $w[i] = w[i+|y|] = w[i+|y|-|x|]$, as $i+|y| > 2|z| \geq |x|$. By induction, the latter is equal to $w[i+|y|-|x|+p] = w[i+|y|+p] = w[i+p]$. We showed that z (and even z^2) occurs at position $p+1$ in w. By Proposition 8.1.5, this contradicts that z is primitive. ∎

Proof of Theorem 8.2.1 Lemma 8.2.2 implies that the number of primitively-rooted squares occurring as prefixes of a word w is less than $2\log_2|w|$. Indeed, from Lemma 8.2.2, it follows that if w has as its prefixes i primitively-rooted squares, then $|w| > 2^{i/2}$. Theorem 8.2.1 follows. ∎

8.2.2. Repetitions in Fibonacci words

Fibonacci words are words over the binary alphabet $\{0,1\}$ defined recursively by $f_0 = 0$, $f_1 = 1$, $f_n = f_{n-1}f_{n-2}$ for $n \geq 2$. The length of f_n, denoted F_n, is the n-th Fibonacci number. Fibonacci words have numerous interesting combinatorial properties and often provide a good example to test conjectures and analyze algorithms on words.

The following lemma summarizes some properties of Fibonacci words that we will need in this section.

LEMMA 8.2.3. *(i) For $n \geq 2$, f_n and the word $f_{n-2}f_{n-1}$ share a common prefix of length $F_n - 2$.*
(ii) For every n, f_n is a primitive word.
(iii) Every repetition occurring in Fibonacci words has the minimal period F_k for some k, and has a cyclic root f_k.
(iv) Fibonacci words contain no repetition of exponent 4.

Proof. (i) is easily proved by induction. To prove (ii), observe that for $n \geq 2$, f_n contains F_{n-1} occurrences of 1, as can be easily proved by induction. If f_n is not primitive that is $f_n = w^k$ for a non-empty word w and $k \geq 2$, then both $F_n = |f_n|$ and F_{n-1} (the number of 1's in f_n) are divisible by k which is a contradiction as F_n and F_{n-1} are mutually prime, which can be again easily proved by induction. Proving (iii) and (iv) is beyond the scope of this chapter (see Notes). ∎

We now prove that Fibonacci words realize the asymptotically largest number of square occurrences, according to Theorem 8.2.1. This shows, in particular, that the $O(n \log n)$ bound of Theorem 8.2.1 is asymptotically tight.

THEOREM 8.2.4. *Fibonacci word f_n contains $\Theta(F_n \log F_n)$ occurrences of primitively-rooted squares.*

Proof. Let S_n be the number of occurrences of primitively-rooted squares in f_n. By induction on n, we will show that $S_n \geq \frac{1}{6}F_n \log_2 F_n$, for $n \geq 5$. For $n = 5$ and $n = 6$, the inequality is verified directly. Assume now that $n \geq 7$. Consider the decomposition $f_n = f_{n-1}f_{n-2}$ and call the position between f_{n-1} and f_{n-2} the *frontier*. Clearly, all squares in f_n are divided into those which lie entirely in f_{n-1} or f_{n-2} and those which *cross the frontier*, i.e. overlap both with f_{n-1} and with f_{n-2}. Therefore,

$$S_{n+1} = S_n + S_{n-1} + \widehat{S}_{n+1},$$

where $\widehat{S}_{n+1}$ is the number of primitively-rooted squares crossing the frontier. By Lemma 8.2.3(i), the prefix of f_{n+1} of length $(F_{n+1}-2)$ is also a prefix of the word $f_{n-1}f_n = f_{n-1}f_{n-1}f_{n-2}$. Since f_{n-2} is a prefix of f_{n-1}, then f_{n+1} contains $(F_{n-2}-1)$ squares of period F_{n-1} that cross the frontier. Since f_{n-1} is a primitive word (Lemma 8.2.3(ii)), all those squares are primitively-rooted. Since

$$f_{n+1} = f_n f_{n-1} = f_{n-1}f_{n-2}f_{n-2}f_{n-3}$$

and f_{n-3} is a prefix of f_{n-2}, then the suffix $f_{n-2}f_{n-2}f_{n-3}$ of f_{n+1} contains $F_{n-3}+1$ squares of period F_{n-2} that cross the frontier. Since f_{n-2} is a primitive word, all those squares are primitively-rooted too.

We obtain that $\widehat{S}_{n+1} \geq (F_{n-2}-1)+(F_{n-3}+1) = F_{n-1}$. Therefore,

$$\begin{aligned}
S_{n+1} &\geq S_n + S_{n-1} + F_{n-1} \geq \frac{1}{6}F_n \log_2 F_n + \frac{1}{6}F_{n-1}\log_2 F_{n-1} + F_{n-1}\\
&= \frac{F_{n+1}}{6}[\frac{F_n}{F_{n+1}}(\log_2 \frac{F_n}{F_{n+1}} + \log_2 F_{n+1}) + \frac{F_{n-1}}{F_{n+1}}(\log_2 \frac{F_{n-1}}{F_{n+1}} + \log_2 F_{n+1})\\
&\quad + \frac{6F_{n-1}}{F_{n+1}}] = \frac{F_{n+1}}{6}(\log_2 F_{n+1} + \Delta),
\end{aligned}$$

where

$$\Delta = \frac{F_n}{F_{n+1}}\log_2 \frac{F_n}{F_{n+1}} + \frac{F_{n-1}}{F_{n+1}}\log_2 \frac{F_{n-1}}{F_{n+1}} + \frac{6F_{n-1}}{F_{n+1}}.$$

Both the first and the second term of the expression Δ is greater than or equal to $\min_{0<x<1}(x\log_2 x) = -\frac{\log_2 e}{e}$. The third term is greater than $3/2$, since $F_{i+1} < 2F_i$ for every i. We derive that

$$\Delta > -\frac{2\log_2 e}{e} + \frac{3}{2} > -\frac{3}{e} + \frac{3}{2} > 0.$$

We conclude that $S_{n+1} \geq \frac{F_{n+1}}{6}\log_2 F_{n+1}$. The upper bound follows from Theorem 8.2.1. ∎

In view of Theorem 8.2.1, one might want to estimate the maximal number of occurrences of primitively-rooted integer powers in a word, and conjecture that this number is asymptotically smaller than the number of square occurrences, since each primitively-rooted integer power of exponent e represents $e-1$ primitively-rooted squares. In particular, one might want to count the occurrences of *non-extensible* primitively-rooted integer powers, that is those primitively-rooted integer powers u^k, $k \geq 2$, which are not followed or preceded by another occurrence of u. Fibonacci words provide a counter-example to this hypothesis, as by Lemma 8.2.3, all integer powers contained in them are of exponent two or three, and therefore the maximal number of their occurrences is still $\Theta(F_n \log F_n)$, as implied by Theorem 8.2.4.

What happens if we count maximal repetitions instead of occurrences of integer powers or just squares? Note that a word can contain much less maximal repetitions than integer powers: e.g. if v is a square-free word over a three-letter alphabet, then word $v\$v\$v$ contains $|v|+1$ non-extensible integer powers (here squares) but only one maximal repetition. What is the number of maximal repetitions in Fibonacci words? The following theorem gives the exact answer.

THEOREM 8.2.5. *Let R_n be the number of maximal repetitions in f_n. Then for all $n \geq 4$, $R_n = 2F_{n-2} - 3$.*

Proof. As in the proof of Theorem 8.2.4, we divide all maximal repetitions in f_n into those which lie entirely in f_{n-1} or f_{n-2} and those which cross the frontier, i.e. overlap with f_{n-1} and with f_{n-2}. We call such repetitions *crossing repetitions* of f_n. Overlaps of a crossing repetition with f_{n-1} and f_{n-2} are called the *left part* and the *right part* respectively. Note first that the left part and the right part of a maximal repetition cannot be both of exponent greater than or equal to 2, since Fibonacci words don't have factors of exponent 4 (Lemma 8.2.3(iv)). If either the left or the right part is of exponent at least 2, then this repetition is an extension of a maximal repetition of respectively f_{n-1} or f_{n-2}. This implies that the only new maximal repetitions of f_n that should be counted are crossing maximal repetitions with both right and left part of exponent smaller than 2. We call those repetitions *composed* repetitions of f_n. Let $c(n)$ be their number. The following lemma allows to compute $c(n)$.

LEMMA 8.2.6. *Let R_n be the number of occurrences of maximal repetitions in the Fibonacci word f_n and set $R_n = R_{n-1} + R_{n-2} + c(n)$. Then for all $n \geq 8$, $c(n) = c(n-2)$.*

Proof. Consider the representation

$$f_n = f_{n-1}|f_{n-2} = f_{n-2}f_{n-3}|f_{n-3}f_{n-4} = f_{n-2}[f_{n-3}|f_{n-4}]f_{n-5}f_{n-4} \qquad (8.2.1)$$

where $|$ denotes the frontier, $n \geq 5$, and square brackets delimit the occurrence of f_{n-2} with the same frontier as for the whole word f_n (we call it the *central occurrence* of f_{n-2}). By Lemma 8.2.3(iii), the minimal period of every repetition in Fibonacci words is equal to F_k for some k. Since $F_{n-3} > F_{n-4} > 2F_{n-6}$, it follows from (8.2.1) that if a composed repetition of f_n has the minimal period F_k for $k \leq n-6$, then it is also a composed repetition of f_{n-2} and therefore is counted in $c(n-2)$. Vice versa, every composed repetition of f_{n-2} with the period F_k for $k \leq n-6$, is also a composed repetition of f_n. We now examine crossing maximal repetitions of f_n with minimal periods F_{n-2}, F_{n-3}, F_{n-4}, F_{n-5}.

Crossing repetitions with the minimal period F_{n-2}. The last term of (8.2.1) shows that square $(f_{n-2})^2$ is a prefix of f_n that crosses the frontier. As $F_{n-1} < 2F_{n-2}$, the corresponding maximal repetition does not have a square in its left or right part and therefore is composed for f_n. Since $F_{n-2} > F_n/3$, any two maximal repetitions of f_n with the period F_{n-2} overlap by more than F_{n-2} letters. By Lemma 8.1.3, f_n has only one maximal repetition with the period F_{n-2}. Trivially, the repetition under consideration is not a repetition for the central occurrence of f_{n-2}.

Crossing repetitions with the minimal period F_{n-3}. From the decomposition $f_n = f_{n-2}f_{n-3}|f_{n-3}f_{n-4}$ (see (8.2.1)), there is a square $(f_{n-3})^2$ of period F_{n-3} crossing the frontier. The corresponding maximal repetition does not extend to the left of the left occurrence of f_{n-3}, as the last letters of f_{n-3} and f_{n-2} are different (the last letters of f_i's alternate). Therefore, this repetition does not have a square in its left or right part, and thus is composed for f_n. As this maximal repetition has a root both on the left and on the right of the frontier, it is the only repetition with the period F_{n-3} crossing the frontier (see Corollary 8.1.4). Again, from length considerations, it is not a maximal repetition for the central occurrence of f_{n-2}.

Crossing repetitions with the minimal period F_{n-4}. Since

$$\begin{aligned} f_n &= f_{n-1}|f_{n-4}f_{n-5}f_{n-4} = f_{n-1}|f_{n-4}f_{n-5}f_{n-5}f_{n-6} \\ &= f_{n-1}|f_{n-4}f_{n-5}f_{n-6}f_{n-7}f_{n-6} = f_{n-1}|(f_{n-4})^2 f_{n-7}f_{n-6}, \end{aligned}$$

there is a square $(f_{n-4})^2$ on the right of the frontier. However, this maximal repetition does not extend to the left (i.e. does not cross the frontier), since the last letters of f_{n-4} and f_{n-1} are different.

On the other hand,

$$\begin{aligned} f_n &= f_{n-2}[f_{n-3}|f_{n-4}]f_{n-5}f_{n-4} = f_{n-3}f_{n-4}[f_{n-4}f_{n-5}|f_{n-5}f_{n-6}]f_{n-5}f_{n-4} \\ &= f_{n-3}f_{n-4}[f_{n-4}\underbrace{f_{n-5}|f_{n-6}}_{f_{n-4}}f_{n-7}f_{n-6}]f_{n-5}f_{n-4}, \text{ for } n \geq 6. \end{aligned}$$

This reveals a maximal repetition with the period F_{n-4} which crosses the frontier. However, this is not a composed repetition of f_n, as it has a square on the left of the frontier. On the other hand, the restriction of this repetition to the central occurrence of f_{n-2} is a composed repetition for f_{n-2}.

There is no other repetition with the period F_{n-4} crossing the frontier, since such would overlap with one of the two above by more than one period, which would contradict Lemma 8.1.3. In conclusion, there is one composed repetition with the period F_{n-4} in the central occurrence of f_{n-2} and no such repetition in f_n.

Crossing repetitions with the minimal period F_{n-5}. Rewrite

$$f_n = f_{n-2}[f_{n-4}f_{n-5}|f_{n-5}f_{n-6}]f_{n-5}f_{n-4}$$

which shows that there is a square of period F_{n-5} crossing the frontier. Since the frontier is the center of this square, the latter corresponds to the only crossing repetition with the period F_{n-5}. However, this repetition is not a composed repetition for f_n, as it has a square in its right part, as shown by the following transformation:

$$\begin{aligned} f_n &= f_{n-2}[f_{n-4}f_{n-5}|f_{n-5}f_{n-6}]f_{n-6}f_{n-7}f_{n-4} \\ &= f_{n-2}[f_{n-4}f_{n-5}|f_{n-5}\underbrace{f_{n-6}]f_{n-7}}_{f_{n-5}}f_{n-8}f_{n-7}f_{n-4}, \text{ for } n \geq 8. \end{aligned}$$

On the other hand, the restriction of this repetition to the central occurrence of f_{n-2} is a composed repetition for f_{n-2}. Thus, there is one composed maximal repetition with the period F_{n-5} in the central occurrence of f_{n-2} and no such repetition in f_n.

In conclusion, two new composed repetitions arise in f_n in comparison to f_{n-2}, but two composed maximal repetitions of the central occurrence of f_{n-2} are no more composed in f_n, as they extend in f_n to form a square in its right or left part. This shows that $c(n) = c(n-2)$ for $n \geq 8$. ■

Proof of Theorem 8.2.5 (continued): A direct counting shows that $R_0 = 0,\ R_1 = 0,\ R_2 = 0,\ R_3 = 0,\ R_4 = 1,\ R_5 = 3,\ R_6 = 7,\ R_7 = 13$. Therefore, $c(3) = 0,\ c(4) = 1,\ c(5) = 2,\ c(6) = 3,\ c(7) = 3$. Since $c(n) = c(n-2)$ for all $n \geq 8$, then $c(n) = 3$ for all $n \geq 6$. We then have the recurrence relation $R_n = R_{n-1} + R_{n-2} + 3$ for $n \geq 6$ with boundary conditions $R_4 = 1,\ R_5 = 3$. Substituting $R_n = 2R'_{n-2} - 3$, we get the relation $R'_n = R'_{n-1} + R'_{n-2}$ for $n \geq 4$, where $R'_2 = 2,\ R'_3 = 3$. This defines exactly the Fibonacci numbers. Thus, $R'_n = F_n$ for $n \geq 2$, and $R_n = 2F_{n-2} - 3$ for $n \geq 4$. ■

Note that by Lemma 8.2.3(iv), any maximal repetition in a Fibonacci word has the exponent smaller than 4, and therefore not only the number of maximal repetitions is linearly bounded, but also the sum of their exponents is linearly bounded on the word length.

8.2.3. Counting maximal repetitions

The results on Fibonacci words suggest a conjecture that arbitrary words contain only a linear number of maximal repetitions and moreover, that the sum of their exponents is linearly-bounded too. In this section we confirm these conjectures. Later in Section 8.4, this result will allows us to derive a linear-time algorithm for identifying all maximal repetitions in a word.

Theorem 8.2.7. *Let $\mathcal{R}(w)$ be the set of all maximal repetitions in a word w of length n (over an arbitrary alphabet), and let $E(w) = \sum_{r\in\mathcal{R}(w)} e(r)$. Then $E(w) = O(n)$.*

The existing proof of Theorem 8.2.7 is very technical and is done by a tedious case analysis. We don't include it here and refer the reader to Kolpakov and Kucherov 2000b. Finding a simple proof of Theorem 8.2.7 remain an open problem.

As a corollary of Theorem 8.2.7, we obtain that the number of maximal repetitions in a word over an arbitrary alphabet is linearly-bounded in the length of the word.

Theorem 8.2.8. *Let $\mathcal{R}(w)$ be the set of all maximal repetitions in a word w of length n. Then* $\mathrm{Card}\,(\mathcal{R}(w)) = O(n)$.

Together with the previous results of Section 8.2, this confirms that the set of maximal repetitions is a more compact representation of all repetitions than

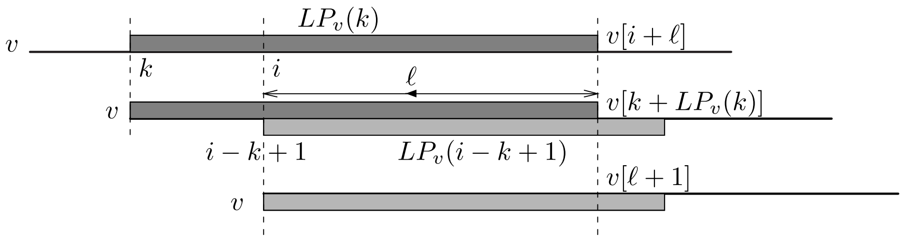

Figure 8.1. Case $LP_v(i-k+1) > \ell$

the set of primitively-rooted squares or primitively-rooted non-extensible integer powers.

8.3. Basic algorithmic tools

From now on, we will be interested in the algorithmic question: How to efficiently compute different types of repetitions in a given word? In the following sections, we present algorithms that allow to compute efficiently all maximal repetitions, as well as other types of repetitions. All those algorithms are based on a common general technique, and therefore a "secondary goal" of this chapter is to demonstrate the power of this approach. The technique is based on two main tools that we describe in this section.

8.3.1. Longest extension functions

The first tool is longest extension functions. Computing those functions (which are basically arrays of integer values, indexed by word positions) is an important component of the algorithms to be presented. Longest extension functions come in different variants – here we present a basic formulation and we will refer to it afterwards.

Consider a word v of length n. For each position i of v, we want to compute the longest factor of v which starts at position i and is also a prefix of v. Formally, we want to compute, for all $i \in [1..n]$, the value $LP_v(i)$ defined as maximal $\ell > 0$ such that $v[1..\ell] = v[i..i+\ell-1]$ and $LP_v(i) = 0$ if no such positive ℓ exists. Note that $LP_v(1) = n$ and, by convention, we always set $LP_v(n+1) = 0$. For example, for $v = 101101011011$, the values of $LP_v(i)$ for $i = 1, \ldots, 13$ are respectively $12, 0, 1, 3, 0, 6, 0, 1, 4, 0, 1, 1, 0$.

We now describe an algorithm that computes LP_v in $O(n)$ time. The algorithm processes v from left to right and computes $LP_v(i)$ for all positions i successively. The computation is based on the following idea. Assume we have computed $LP_v(j)$ for all $j < i$, and we are about to compute $LP_v(i)$. Assume we have stored a position $k < i$ that maximizes $k + LP_v(k)$, and assume that $k + LP_v(k) > i$. Set $\ell = k + LP_v(k) - i$ and consider the value $LP_v(i-k+1)$. If

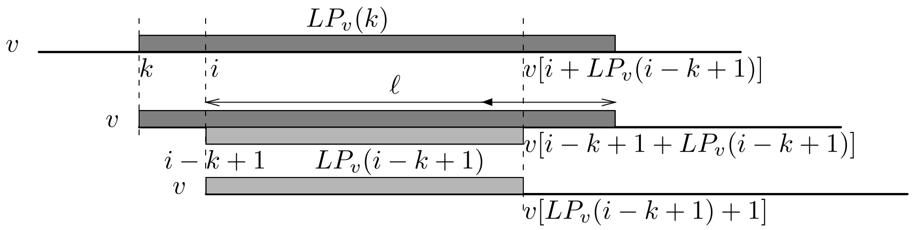

Figure 8.2. Case $LP_v(i - k + 1) < \ell$

$LP_v(i-k+1) > \ell$ (see Figure 8.1), then we claim that $LP_v(i) = \ell$. Indeed, it is easily seen that $v[i..i+\ell-1] = v[1..\ell]$. On the other hand, $v[\ell+1] \neq v[i+\ell]$, as $v[i+\ell] = v[k+LP_v(k)] \neq v[LP_v(k)+1]$, and $v[LP_v(k)+1] = v[\ell+1]$. Therefore, $LP_v(i) = \ell$.

If $LP_v(i-k+1) < \ell$ (see Figure 8.2), then we show that $LP_v(i) = LP_v(i-k+1)$. Again, $v[i..i+LP_v(i-k+1)-1] = v[1..LP_v(i-k+1)]$, but $v[i+LP_v(i-k+1)] \neq v[LP_v(i-k+1)+1]$, since $v[i+LP_v(i-k+1)] = v[i-k+1+LP_v(i-k+1)]$ and $v[i-k+1+LP_v(i-k+1)] \neq v[LP_v(i-k+1)+1]$.

Putting together the two cases above, if $LP_v(i-k+1) \neq \ell$, then $LP_v(i) = \min\{\ell, LP_v(i-k+1)\}$. The only case when $LP_v(i)$ cannot be computed immediately is $LP_v(i-k+1) = \ell$. In this case, we keep on reading letters $v[i+\ell], v[i+\ell+1], \ldots$ while $v[i+\ell+j] = v[\ell+j]$, and thus compute the value $LP_v(i)$. Position i is then stored as the new value of k. The resulting linear-time algorithm Longest-Prefix-Extension is shown below.

Function LP_v can be generalized to compute, for all positions of v, the longest factor that starts at this position and is a prefix of *another* fixed word. Formally, for two words $v[1..n]$ and $w[1..m]$, we define $LP_{v|w}$ to be the function associating with every position $i \in [1..n]$ of v the maximal length of a factor of word vw which starts at position i and is a prefix of w. Note that this factor starts inside v but can overlap with w. All values of $LP_{v|w}$ can be computed in time $O(m+n)$.

```
Longest-Prefix-Extension(v[1..n])
LP(1) ← n
j ← 0
while j ≤ n − 2 and v[j + 1] = v[j + 2] do
    j ← j + 1
LP(2) ← j
k ← 2
for i ← 3 to n do
    ℓ ← k + LP(k) − i
    if LP(i − k + 1) ≠ ℓ and ℓ ≥ 0 then
        LP(i) ← min(ℓ, LP(i − k + 1))
    else j ← max(0, ℓ)
        while i + j ≤ n and v[i + j] = v[j + 1] do
            j ← j + 1
        LP(i) ← j
        k ← i
return LP
```

Symmetrical functions can be defined with respect to suffixes instead of prefixes. For a word $v[1..n]$, we define $LS_v(i)$ to be the length of the longest factor of v that ends at position i and is a suffix of v. For $v[1..n]$ and $w[1..m]$, $LS_{w|v}(i)$, $i \in [1..n]$ is the maximal length of a factor of wv that ends at position $m + i$ in wv and is a suffix of w. Both these functions can be computed in time linear in the involved words, using an algorithm similar to Longest-Prefix-Extension. In particular, the function computing $LS_{w|v}$ will be called Longest-Suffix-Extension(w, v) in the sequel.

8.3.2. s-factorization and Lempel-Ziv factorization

The second basic algorithmic tool is a special factorization of the word, that will allow to speed up our repetition-finding algorithms. There are several variants of this factorization that we will use in the following sections. The difference is of technical nature and the choice of the definition will be basically guided by the convenience in describing the algorithm. We distinguish two factorizations that we call *s-factorization* and *Lempel-Ziv factorization*, following the terms under which they have been introduced in the literature. Each of those factorizations comes in two variants, thus yielding four different definitions.

Let w be an arbitrary word. The *s-factorization* of w (respectively, *s-factorization with non-overlapping copies*) is the factorization $w = f_1 f_2 \cdots f_k$, where f_i's are defined inductively as follows:

- $f_1 = w[1]$, and if letter a occurring in w immediately after $f_1 f_2 \cdots f_{i-1}$ does not occur in $f_1 f_2 \cdots f_{i-1}$, then $f_i = a$.
- otherwise, f_i is the longest factor occurring in w immediately after $f_1 f_2 \cdots f_{i-1}$ that occurs in $f_1 f_2 \cdots f_{i-1} f_i$ other than as a suffix (respectively, that occurs in $f_1 f_2 \cdots f_{i-1}$).

The *Lempel-Ziv factorization of* w (respectively, *Lempel-Ziv factorization with non-overlapping copies*) is the factorization $w = f_1 f_2 \cdots f_k$, where f_i's are defined inductively as follows:

- $f_1 = w[1]$,
- for $i \geq 2$, f_i is the shortest factor occurring in w immediately after $f_1 f_2 \cdots f_{i-1}$ that does not occur in $f_1 f_2 \cdots f_{i-1} f_i$ other than as a suffix (respectively, that does not occur in $f_1 f_2 \cdots f_{i-1}$ at all).

In the s-factorization of the word w, we look for the longest factor f_i starting after $f_1 f_2 \cdots f_{i-1}$ which has a copy on the left. In the s-factorization with non-overlapping copies, this copy is required to be non-overlapping with f_i. In the Lempel-Ziv factorization, we look for the shortest word that does not have an occurrence on the left. In other words, we extend by one letter the longest word which does have a copy on the left. If the Lempel-Ziv factorization with non-overlapping copies is considered, then the copy is required not to overlap with f_i.

As an example, consider the word $w = 1100101010000$. Its s-factorization is $1|1|0|0|10|1010|000$, and the s-factorization with non-overlapping copies of w is $1|1|0|0|10|10|100|00$. The Lempel-Ziv factorization of w is $1|10|01|010100|00$ and the Lempel-Ziv factorization without copy overlap is $1|10|01|010|1000|0$.

If $w = f_1 f_2 \cdots f_k$ is the s-factorization (respectively, Lempel-Ziv factorization), we call each f_i an *s-factor* (respectively, *LZ-factor*) of w.

A remarkable feature of all considered factorizations is that each of them can be computed in a time linear in the length of the word. This can be done in different ways, using a data structure like the suffix tree or the DAWG (Directed Acyclic Word Graph). A possible algorithm consists in computing the factorization along with constructing the data structure in an on-line fashion. A factorization provides a very useful information about the structure of repeated factors and the possibility to compute the factorization in linear time makes of it a key algorithmic tool that we will use throughout the rest of this chapter.

8.4. Finding all maximal repetitions in a word

According to the results of Section 8.2, maximal repetitions are important structures, as they encode, in a most compact way, all repetitions occurring in the word. If the set of maximal repetitions is known, repetitions of any other type can be extracted from it: primitively- or non-primitively rooted squares, cubes, etc.

In this section, we show that the set of all maximal repetitions can be computed very efficiently, namely in a time linear in the length of the word. The linear time bound is supported by Theorems 8.2.7, 8.2.8 that guarantee that the output itself is of linear size (assuming that each maximal repetition is represented in constant space, e.g. by the start position, the period and the total length).

We first consider the following auxiliary problem. Assume we are given two words $x = x[1..m]$, $y = y[1..n]$, and consider their concatenation $v = xy =$

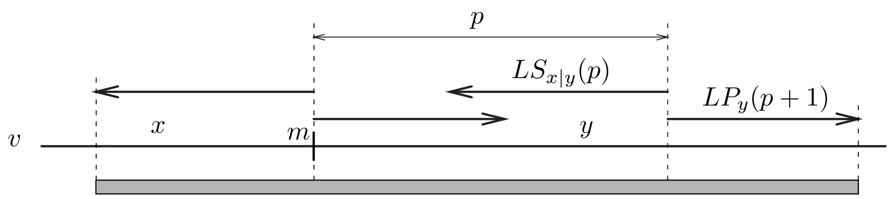

Figure 8.3. Illustration to Theorem 8.4.1

$v[1..m+n]$. We want to find all maximal repetitions $r = v[i..j]$ in the word v that *contain the frontier between* x *and* y, i.e. such that $i \leq m+1$ and $j \geq m$. Every such repetition belongs (non-exclusively) to one of two classes: the repetitions which have a root in y and those which have a root in x. Note that by Corollary 8.1.4, for every p, $1 \leq p \leq n$, there is at most one maximal repetition with a period p that contains the frontier between x and y and has a root in y. This shows, in particular, that the number of such repetitions is not greater than n. Similarly, the number of repetitions that contain the frontier and have a root in x is not greater than m, and thus, the number of maximal repetitions in $v = xy$ which contain the frontier between x and y is bounded by $(m+n)$.

Let us focus on maximal repetitions r which have a root in y, those which have a root in x are found similarly. Consider longest extension functions LP_y and $LS_{x|y}$ (see Section 8.3). The following theorem holds.

Theorem 8.4.1. *For $1 \leq p \leq n$, there exists a maximal repetition with a period p in $v = xy$ that contains the frontier between x and y and has a root in y iff*

$$LS_{x|y}(p) + LP_y(p+1) \geq p. \tag{8.4.1}$$

If the inequality holds, this maximal repetition is $v[m - LS_{x|y}(p) + 1..m + p + LP_y(p+1)]$ (see Figure 8.3).

Proof. Assume there is a square $s = v[\ell..\ell+2p-1]$ of period p that contains the frontier between x and y, i.e. such that $\ell \leq m+1$ and $\ell+2p-1 \geq m$. Assume that s has a root in y, i.e. $\ell+p > m$. Observe that prefix $y[1..\ell+p-m-1]$ of y is equal to $y[p+1..\ell+2p-m-1]$. This implies that $LP_y(p+1) \geq \ell+p-m-1$. On the other hand, suffix $x[\ell..m]$ of x is equal to $v[\ell+p..m+p]$, and then $LS_{x|y}(p) \geq m-\ell+1$. Inequality (8.4.1) follows.

On the other hand, if (8.4.1) holds, then any factor $v[\ell..\ell+2p-1]$ with $m - LS_{x|y}(p) + 1 \leq \ell \leq m + LP_y(p+1) - p + 1$ is a square of period p. Therefore, $r = v[m - LS_{x|y}(p) + 1..m + p + LP_y(p+1)]$ is a repetition. It remains to see that r is maximal, i.e. extended to the right and left as far as possible. This follows from the definition of the longest extension functions LS and LP. ∎

Theorem 8.4.1 yields an algorithm for computing all maximal repetitions which contain the frontier between x and y and have a root in y. The algorithm, called Right-Repetitions, is shown below. It assumes that each repetition is represented by a pair of its start and end positions.

```
Right-Repetitions(x, y)
LP_y ← Longest-Prefix-Extension(y)
LS_{x|y} ← Longest-Suffix-Extension(x, y)
R ← ∅
for p ← 1 to |y| do
    if LS_{x|y}(p) + LP_y(p + 1) ≥ p then
        r ← (m − LS_{x|y}(p) + 1, m + p + LP_y(p + 1))
        R ← R ∪ {r}
return R
```

The repetitions containing the frontier between x and y and having a root in x are computed similarly (function Left-Repetitions hereafter). As the functions Longest-Prefix-Extension(y) and Longest-Suffix-Extension(y) run in time $O(|y|)$ and $O(|x|+|y|)$ respectively, all maximal repetitions in $v = xy$ which contain the frontier between x and y can be computed in time $O(|v|)$.

Let us now come back to the problem of computing all maximal repetitions in a word. The s-factorization is another useful tool for building a linear-time algorithm for this problem, due to the following theorem.

Theorem 8.4.2. *Let $w = f_1 f_2 \cdots f_k$ be the s-factorization of w. Let r be a maximal repetition in w that contains the frontier between f_{i-1} and f_i and ends inside f_i. Then the prefix of r which is a suffix of $f_1 \cdots f_{i-1}$ is smaller than $|f_i| + 2|f_{i-1}|$.*

Proof. Assume $r = w[\ell..m]$ and denote b_i the position of the last letter of f_i, i.e. $b_i = |f_1 f_2 \cdots f_i|$. The theorem asserts that if $\ell \le b_{i-1}+1$ and $b_{i-1}+1 \le m \le b_i$, then $b_{i-1} + 1 - \ell \le |f_i| + 2|f_{i-1}|$.

Consider the suffix cyclic root $r' = w[m - p(r) + 1..m]$ of r. Observe that r' has a copy $p(r)$ letters to the left. If r' starts at a position before the start of f_{i-1}, i.e. $m - p(r) \le b_{i-2}$, then it includes entirely the factor f_{i-1} and at least the first letter of f_i. This contradicts that f_{i-1} is the *longest* factor occurring on the left, according to the definition of s-factorization. Therefore, $m-p(r) > b_{i-2}$ and then $p(r) = |r'| < |f_{i-1}| + |f_i|$.

On the other hand, r cannot extend to the left of $b_{i-2} + 1$ by $p(r)$ letters or more, as this would again contradict the definition of f_{i-1}. Thus, the part of r before the start of f_{i-1} is bounded by $|f_{i-1}| + |f_i|$. The theorem follows. ∎

For the simplicity of presentation, we assume that the last letter of w does not occur elsewhere in w. Given the s-factorization $w = f_1 f_2 \cdots f_k$, we consider, for each s-factor f_i, $i \in [2..k]$, all maximal repetitions that end either at the last position of f_{i-1}, or at some position of f_i except the last one. Formally, these are repetitions $r = w[\ell..m]$ such that $b_{i-1} \le m < b_i$, where $b_i = |f_1 f_2 \cdots f_i|$. Cleary, each maximal repetition of w belongs to exactly one such class. Note that since the last letter of w is unique, the last s-factor f_k consists of one letter and there is no maximal repetitions that occur as suffixes of w.

We further split all considered repetitions into two sets that we call *repetitions of type 1 and 2*:

repetitions of type 1: maximal repetitions r that contain the frontier between f_{i-1} and f_i and end at a position strictly smaller than the end of f_i,

repetitions of type 2: maximal repetitions r that occur properly inside f_i.

Repetitions of type 1 are repetitions $r = w[\ell..m]$ such that either $m = b_{i-1}$, or $b_{i-1} + 1 \le m \le b_i - 1$ and $\ell \le b_{i-1} + 1$. By Theorem 8.4.2, the former cannot extend by more than $|f_{i-1}| + 2|f_{i-2}|$ to the left of f_{i-1}, and therefore its length is bounded by $2|f_{i-1}| + 2|f_{i-2}|$, and the latter cannot extend by more than $|f_i| + 2|f_{i-1}|$ to the left of f_i. Joining both cases together, a repetition of type 1 cannot extend by more than $\max\{2|f_{i-1}| + 2|f_{i-2}|, |f_i| + 2|f_{i-1}|\} = 2|f_{i-1}| + \max\{2|f_{i-2}|, |f_i|\}$ to the left of f_i.

Therefore, to find all repetitions of type 1, we consider the word $t_i f_i$, where t_i is the suffix of $f_1 \cdots f_{i-1}$ of length $2|f_{i-1}| + \max\{2|f_{i-2}|, |f_i|\}$ (t_i is the whole word $f_1 \cdots f_{i-1}$ if its length is smaller than $2|f_{i-1}| + \max\{2|f_{i-2}|, |f_i|\}$). We then have to find in $t_i f_i$ all maximal repetitions that contain the frontier between t_i and f_i and don't include the last letter of f_i. This can be done in time $O(|f_{i-2}| + |f_{i-1}| + |f_i|)$ using longest extension functions, as described above. Summing up over all s-factors, all repetitions of type 1 in w can be found in time $O(n)$.

Every repetition of type 2 occurs entirely inside some s-factor f_i of the s-factorization, and each f_i has an earlier occurrence in w. Therefore, each maximal repetition of type 2 has another occurrence on the left. This implies, in particular, that finding all repetitions of type 1 guarantees finding all *distinct* maximal repetitions, and in particular all *leftmost* occurrences of distinct maximal repetitions.

We are left with the problem of finding all repetitions of type 2. Here is how this can be done.

During the computation of the s-factorization we store, for each s-factor f_i, a pointer to an earlier occurrence of f_i in w. Computing such a pointer does not affect the linear time complexity of s-factorization. Let v_i be this earlier occurrence of f_i, and let Δ_i be the difference between the position of f_i and the position of v_i. Obviously, each repetition of type 2 occurring inside f_i is a copy of a maximal repetition occurring inside v_i shifted by Δ_i to the right.

We first sort, using bucket sort, all maximal repetitions of type 1, found at the first stage, into n lists $end[1], ..., end[n]$ such that list $end[j]$ contains all maximal repetitions with end position j. Then we process all lists $end[j]$ in the increasing order of j and sort the repetitions again, using bucket sort, into n lists $start[1], ..., start[n]$ according to their start position. After this double sorting, the repetitions with the same start position j are sorted inside the list $start[j]$

in the increasing order of their end positions. As there is a linear number of repetitions of type 1, both sorting procedures take a linear time.

We will use the same lists $start[j]$ to store maximal repetitions of type 2. For each s-factor f_i and for each internal position j inside f_i, we have to find all maximal repetitions starting at this position and ending strictly inside f_i. We then have to find all maximal repetitions from the list $start[j - \Delta_i]$ which end inside v_i, and then shift them by Δ_i to the right. Note that these repetitions may be either of type 1, or previously found repetitions of type 2. We look through the list $start[j - \Delta_i]$ and retrieve its prefix consisting of those maximal repetitions which end inside v_i. Then we shift each of these maximal repetitions by Δ_i and append a modified copy of this prefix to the head of the list $start[j]$. Note that the data structure is preserved, as all appended repetitions must end before any of repetitions of type 1 previously stored in the list $start[j]$. Since we process f_i's from left to right, no maximal repetition can be missed. Thus, we recover all repetitions of type 2 and after all f_i's have been processed, the data structure contains all maximal repetitions of both types.

Note that when we retrieve a prefix of the list corresponding to some position in v_i, each repetition in this prefix results in a new maximal repetition of type 2 in f_i. This shows that the time spent on processing the lists is proportional to the number of newly found maximal repetitions. Theorem 8.2.7 states that the number of all maximal repetitions is linear in the length of w. This proves that the whole algorithm takes a linear time.

The whole algorithm for computing all maximal repetitions is summarized below.

```
Maximal-Repetitions(w[1..n])
(f_1, ..., f_k) ← s-factorization of w
▷ first stage
R ← ∅
for i ← 1 to k do
    t_i ← suffix of f_1 ··· f_{i-1} of length 2|f_{i-1}| + max{2|f_{i-2}|, |f_i|}
    R'_i ← Right-Repetitions(t_i, f_i)
    R''_i ← Left-Repetitions(t_i, f_i)
    R ← R ∪ R'_i ∪ R''_i
▷ second stage
for each r = (j, ℓ) ∈ R do
    add r to list end[ℓ]
for ℓ ← 1 to n do
    for each r = (j, ℓ) from list end[ℓ] do
        add r to list start[j]
for i ← 1 to k do
    for j ← b_{i-1} + 1 to b_i do
        for each r from list start[j − Δ_i] such that |r| < b_i − j + 1 do
            add repetition r' = (j, j + |r| − 1) to list start[j]
            R ← R ∪ {r'}
return R
```

The complexity of MAXIMAL-REPETITIONS follows from the above discussion:

THEOREM 8.4.3. MAXIMAL-REPETITIONS *finds all maximal repetitions in a word of length n in time $O(n)$.*

The set of all maximal repetitions provides an exhaustive information about the repetition structure of the word. It allows to extract all repetitions of other types, such as (primitively- or non-primitively-rooted) squares, cubes, or integer powers. Thus, all these tasks can be done in time $O(n+T)$ where T is the output size.

As another application, the set of maximal repetitions allows to determine, in linear time, the number $d_k(i)$ of primitively-rooted integer powers of a given exponent k, starting at each position i of the word. Here is how this can be done. For each position $i \in [1..n]$ of the input word w, we create two counters $b(i)$ and $c(i)$, initially set to 0. For each repetition $r = w[\ell..m]$, we increment $b(\ell)$ and $c(m - kp(r) + 1)$ by 1 ($[\ell..m - kp(r) + 1]$ is the interval, where primitively-rooted k-powers induced by repetition r start). By Theorem 8.2.8, the number of updates is linear. To compute the numbers $d_k(i)$, we scan all positions from left to right applying the following iterative procedure: $d_k(1) = b(1)$, $d_k(i+1) = d_k(i) + b(i) - c(i-1)$, $i = 2..|w|$. Note that the algorithm can be extended to all (not necessarily primitively-rooted) k-powers. In this case, we increment $b(\ell)$ by $\lfloor e(r)/k \rfloor$, and we increment by 1 each $c(j)$, for $j = m - kp(r) + 1, m - 2kp(r) + 1, \ldots, m - \lfloor e(r)/k \rfloor kp(r) + 1$. Here, Theorem 8.2.7 guarantees that the number of updates is linear. Finally, note that the procedure can be easily modified in order to count fractional repetitions of a given exponent, as well as to repetitions ending (or centered) at each position.

8.5. Finding quasi-squares in two words

The linear-time algorithm for computing all maximal repetitions presented in the previous section allows us to compute all squares in a word of length n in time $O(n + S)$, where S is the number of those squares. In this section we consider a problem of finding *quasi-squares* that generalizes the problem of computing usual squares. Besides that the problem of quasi-squares is interesting in its own, it will be used later in Section 8.6.1.

Assume we are given two words u, v of equal length, $|u| = |v| = n$, $n \geq 2$. We say that words u, v contain a *quasi-square* iff for some $1 \leq \ell \leq n$ and $p > 0$, we have $u[\ell..\ell+p-1] = v[\ell+p..\ell+2p-1]$. p is called the *period* of the quasi-square, and words $u[\ell..\ell + p - 1], v[\ell + p..\ell + 2p - 1]$ are called respectively its *left root* and *right root*. Obviously, if $u = v$, then the quasi-squares are usual squares.

Denote $\mathcal{QS}(u, v)$ the set of all quasi-squares of words u, v. We show that $\mathcal{QS}(u, v)$ can be computed in time $O(n \log n + S)$, where $S = \mathrm{Card}\,(\mathcal{QS}(u, v))$. The algorithm we propose is based only on longest extension functions and is similar to the algorithm based on Theorem 8.4.1 for finding maximal repetitions containing a given position. An advantage of the proposed solution is that the output quasi-squares are naturally grouped into runs of quasi-squares of the same period starting at successive positions in the word. This runs are analogous to repetitions for usual non-gapped squares. We will use this feature of the algorithm later in Section 8.6.

Assume $n = 2m$, and denote $\mathcal{QS}_m(u, v)$ the subset of $\mathcal{QS}(u, v)$ consisting of those quasi-squares $(u[\ell..\ell + p - 1], v[\ell + p..\ell + 2p - 1])$ which contain the frontier in the middle of u and of v, more precisely such that $\ell \le m + 1$ and $\ell + 2p - 1 \ge m$. $\mathcal{QS}_m(u, v)$ is the union of two subsets $\mathcal{QS}^l_m(u, v)$ and $\mathcal{QS}^r_m(u, v)$ consisting of quasi-squares containing the middle frontier in their left root or right root respectively. Consider the set $\mathcal{QS}^l_m(u, v)$ ($\mathcal{QS}^r_m(u, v)$ is treated similarly). $\mathcal{QS}^l_m(u, v)$ consists of quasi-squares $(u[\ell..\ell + p - 1], v[\ell + p..\ell + 2p - 1])$ verifying $m + 1 - p \le \ell \le m + 1$.

Consider the following longest extension functions defined on all positions $i \in [1..m]$ of $v[m + 1..n]$:

$$\widehat{LP}(i) = \min\{LP_{v[m+1..n]|u[m+1..n]}(i), m - i + 1\}$$
$$\widehat{LS}(i) = \min\{LS_{u[1..m]|v[m+1..n]}(i), i\}$$

In words, $\widehat{LP}(i)$ is the length of the longest common prefix of $v[m + i..n]$ and $u[m + 1..n]$, and $\widehat{LS}(i)$ is the length of the longest common suffix of $u[1..m]$ and $v[m + 1..m + i]$.

Let $u[\ell..\ell + p - 1], v[\ell + p..\ell + 2p - 1]$ be a quasi-square of period p belonging to $\mathcal{QS}^l_m(u, v)$. By an argument similar to Theorem 8.4.1, we have

$$\widehat{LS}(p) + \widehat{LP}(p + 1) \ge p. \tag{8.5.1}$$

Vice versa, if for some $p \in [1..m]$, inequality (8.5.1) holds, then there exists a quasi-square of period p from $\mathcal{QS}^l_m(u, v)$. More precisely, the following lemma holds.

LEMMA 8.5.1. *For $p \in [1..m]$, there exists a quasi-square of $\mathcal{QS}^l_m(u, v)$ of period p iff inequality (8.5.1) holds. When (8.5.1) holds, there is a family of quasi-squares of period p from $\mathcal{QS}^l_m(u, v)$, with the left roots starting at each position of the interval*

$$[m + 1 - \widehat{LS}(p) \;..\; m + 1 + \min\{\widehat{LP}(p + 1) - p, 0\}]. \tag{8.5.2}$$

To use Lemma 8.5.1 as an algorithm for computing $\mathcal{QS}^l_m(u, v)$, we have to compute the values $\widehat{LP}(i)$, $\widehat{LS}(i)$ for $i \in [1..m]$. All these values can be computed in time $O(m)$, as explained in Section 8.3.1.

We conclude that all quasi-squares of $\mathcal{QS}^l_m(u, v)$ can be computed in time $O(m + \mathrm{Card}\left(\mathcal{QS}^l_m(u, v)\right))$. Similarly, all quasi-squares of $\mathcal{QS}^r_m(u, v)$ can be computed in time $O(m + |\mathcal{QS}^r_m(u, v)|)$, and thus all quasi-squares of $\mathcal{QS}_m(u, v)$ are computed in time $O(m + \mathrm{Card}\,(\mathcal{QS}_m(u, v)))$. A straightforward divide-and-conquer algorithm gives the running time $O(n \log n + \mathrm{Card}\,(\mathcal{QS}(u, v)))$ for finding all quasi-squares in u, v.

THEOREM 8.5.2. *The set $\mathcal{QS}(u,v)$ of all quasi-squares in words u, v of length n can be found in time $O(n \log n + \mathrm{Card}\,(\mathcal{QS}(u,v)))$.*

8.6. Finding repeats with a fixed gap

In this section, we are interested in the problem of computing all factors of a given word repeated within a specified distance, rather then contiguously. In other words, we want to find all factor occurrences uvu, where the size of v, called the *gap*, is equal to a pre-specified constant δ. We will show that using the algorithm of the previous sections, all such occurrences can be found in time $O(n \log \delta + S)$, where S is the number of them. Thus, if δ is considered constant, we obtain an $O(n+S)$ time bound.

8.6.1. Algorithm for finding repeats with a fixed gap

Let $\delta > 0$ be an integer, called the *gap*. An occurrence in w of a factor $r = uvu$, where $|u| > 0$ and $|v| = \delta$, is called a *δ-gapped repeat* (for short, *δ-repeat*) in w. The left occurrence of u is called the *left copy* of r, and the right one the *right copy*. For a δ-repeat r, the length $|u|$ is called the *copy length*. The problem is to find all δ-repeats in a given word.

Let $w = a_1 \cdots a_n$ be a word of length n. Without loss of generality, we assume that a_n does not occur elsewhere in w. In this section, we use the Lempel-Ziv factorization (see Section 8.3.2). Let $w = f_1 \cdots f_k$ be the Lempel-Ziv factorization of w.

To describe the algorithm, we need some notation. Let $b_0, b_1, \ldots, b_{k-1}, b_k$ be the end positions of f_i's, that is $b_0 = 0$, and $b_i = |f_1 \cdots f_i|$ for $1 \le i \le k$. We also denote $\ell_i = |f_i|$, $i = 1, \ldots, k$. For every $i = 1, \ldots, k-1$, if $\ell_i > \delta$, then we decompose $f_i = f_i' f_i''$, where $|f_i'| = \delta$.

Let us split the set $\mathcal{GR}$ of all δ-repeats into the set $\mathcal{GR}'$ of those δ-repeats which contain a frontier between LZ-factors and the set $\mathcal{GR}''$ of the δ-repeats located properly inside LZ-factors. We now concentrate on the δ-repeats of $\mathcal{GR}'$, and further split $\mathcal{GR}'$ into (disjoint) subsets $\mathcal{GR}_i'$, $i = 1, \ldots, k-1$, where $\mathcal{GR}_i'$ consists of those δ-repeats which contain the frontier between f_i and f_{i+1} but don't contain the frontier between f_{i+1} and f_{i+2}. Furthermore, each $\mathcal{GR}_i'$ is split into the following subsets:

(a) $r \in \mathcal{GR}_i^{lrt}$ iff the left copy of r contains the frontier between f_i and f_{i+1},
(b) $r \in \mathcal{GR}_i^{rrt}$ iff the right copy of r contains the frontier between f_i and f_{i+1},
(c) $r \in \mathcal{GR}_i^{mrt}$ iff $\ell_{i+1} > \delta$ and the right copy of r contains the frontier between f_{i+1}' and f_{i+1}'' but does not contain the frontier between f_i and f_{i+1},
(d) $r \in \mathcal{GR}_i^{mid}$ iff the right copy of r contains neither the frontier between f_i and f_{i+1}, nor, if $\ell_{i+1} > \delta$, the frontier between f_{i+1}' and f_{i+1}''.

Cases (a) and (b) cover the situation when the frontier between f_i and f_{i+1} is contained respectively in the left and right copy of r. Otherwise, this frontier is contained in the gap between the copies. Cases (c) and (d) distinguish whether the right copy contains the frontier between f_{i+1}' and f_{i+1}'' or not, provided that $\ell_{i+1} > \delta$.

We now consider each of cases (a)-(d) separately and show how to find the corresponding set of δ-repeats.

(a) Finding δ-repeats of $\mathcal{GR}_i^{lrt}$. Let $r \in \mathcal{GR}_i^{lrt}$ be a δ-repeat with copy length p. Since r does not contain the frontier between f_{i+1} and f_{i+2}, then $\delta + p < \ell_{i+1}$, and therefore $p \leq \ell_{i+1} - 1 - \delta$. In particular, $\mathcal{GR}_i^{lrt}$ is empty whenever $\ell_{i+1} \leq \delta + 1$. Assume now that $\ell_{i+1} > \delta + 1$.

Let t_i be the suffix of $f_1 \cdots f_i$ of length $\ell_{i+1} - 1 - \delta$. We define the following longest extension functions on all positions $i \in [1..\ell_{i+1} - 1]$ of f_{i+1}:

$$\widehat{LP}(i) = LP_{f_{i+1}[1..\ell_{i+1}-1]}(i)$$
$$\widehat{LS}(i) = LS_{t_i|f_{i+1}[1..\ell_{i+1}-1]}(i)$$

Similarly to Theorem 8.4.1, an occurrence of a δ-repeat $r \in \mathcal{GR}_i^{lrt}$ implies

$$\widehat{LS}(\delta + p) + \widehat{LP}(\delta + p + 1) \geq p. \tag{8.6.1}$$

Conversely, if for some $p \in [1..\ell_{i+1} - 1 - \delta]$, inequation (8.6.1) holds, then there exists a δ-repeat of $\mathcal{GR}_i^{lrt}$ with copy length p. To summarize, the following lemma holds.

LEMMA 8.6.1. *For $p \in [1..\ell_{i+1} - 1 - \delta]$, there exists a δ-repeat of $\mathcal{GR}_i^{lrt}$ with copy length p iff inequality (8.6.1) holds. When (8.6.1) holds, there is a family of δ-repeats of $\mathcal{GR}_i^{lrt}$ with copy length p, starting at each position of the interval*

$$[b_i + 1 - \min\{\widehat{LS}(\delta + p), p\} \;..\; b_i + 1 + \min\{\widehat{LP}(\delta + p + 1) - p, 0\}]. \tag{8.6.2}$$

Lemma 8.6.1 gives a method of computing $\mathcal{GR}_i^{lrt}$. Compute the longest extension functions $\widehat{LS}$ and $\widehat{LP}$. According to Section 8.3.1, this computation can be done in linear time in the length of involved words, that is in time $O(\ell_{i+1})$. Then, all δ-repeats of $\mathcal{GR}_i^{lrt}$ can be computed in time $O(\ell_{i+1} + \mathrm{Card}\left(\mathcal{GR}_i^{lrt}\right))$.

(b) Finding δ-repeats of $\mathcal{GR}_i^{rrt}$. Consider a δ-repeat $r \in \mathcal{GR}_i^{rrt}$ with copy length p. From the definition of Lempel-Ziv factorization, it follows that the right copy of r starts strictly after the start of f_i. On the other hand, from the definition of $\mathcal{GR}_i^{rrt}$, it ends strictly before the end of f_{i+1}. Therefore, $p \leq \ell_i + \ell_{i+1} - 2$.

We then proceed similarly to case (a). Using appropriate longest extension functions computed in time $O(\ell_i + \ell_{i+1})$, all δ-repeats of $\mathcal{GR}_i^{rrt}$ can be reported in time $O(\ell_i + \ell_{i+1} + \mathrm{Card}\left(\mathcal{GR}_i^{rrt}\right))$.

(c) Finding δ-repeats of $\mathcal{GR}_i^{mrt}$. Note that this case is defined only when $\ell_{i+1} > \delta$. Consider a δ-repeat $r \in \mathcal{GR}_i^{mrt}$ with copy length p. The right copy of r occurs inside $w[b_i + 2..b_i + \ell_{i+1} - 1]$, and therefore $p \leq \ell_{i+1} - 2$.

Again, using appropriate longest extension functions, all δ-repeats of $\mathcal{GR}_i^{mrt}$ can be reported in time $O(\ell_{i+1} + \mathrm{Card}\left(\mathcal{GR}_i^{mrt}\right))$.

(d) Finding δ-repeats of $\mathcal{GR}_i^{mid}$. Consider now a δ-repeat $r \in \mathcal{GR}_i^{mid}$ with copy length p. Denote $m_i = \min\{\delta, \ell_{i+1}\}$. The right copy of r occurs inside $f_{i+1}[2..m_i - 1]$, and therefore $p \le m_i - 2$.

This case differs from cases (a)-(c) in that we cannot *a priori* select a position contained in the right or left copy of r. Therefore, we cannot apply directly the technique of longest extension functions. We reduce this case to the problem of finding quasi-squares, considered in Section 8.5.

Since the start position of the right copy belongs the interval $[b_i + 2..b_i + m_i - 1]$, the end position of the left copy belongs to the interval $w[b_i + 1 - \delta..b_i + m_i - 2 - \delta]$. Since $p \le m_i - 2$, the left copy of r is contained in the word $w[b_i - \delta - m_i + 4..b_i + m_i - 2 - \delta]$.

Consider the word $w' = w[b_i - \delta - m_i + 4..b_i + m_i - 1 - \delta]$. The length of w' is $(2m_i - 4)$. Let $\#$ be another fresh letter. Denote by w'' the word $\#^{m_i-2}w[b_i + 2..b_i + m_i - 1]$.

Lemma 8.6.2. *There exists a δ-repeat $r \in \mathcal{GR}_i^{mid}$ iff there exists a quasi-square in words w', w''. Each such quasi-square corresponds to a δ-repeat $r \in \mathcal{GR}_i^{mid}$.*

Proof. Consider a δ-repeat $r \in \mathcal{GR}_i^{mid}$ with the left copy $w[\ell..\ell+p-1]$ and the right copy $w[\ell+\delta+p..\ell+\delta+2p-1]$. The right copy is a factor of $w[b_i+2..b_i+m_i-1]$, and therefore starts at position $(\ell + \delta + p - b_i + m_i - 3)$ in w''. The left copy starts at position $\ell - (b_i - \delta - m_i + 4) + 1 = \ell + \delta - b_i + m_i - 3$ in w' and therefore this forms a quasi-square of period p in w' and w''.

Inversly, assume there is a quasi-square $w'[j..j+p-1] = w''[j+p..j+2p-1]$. We must have $[j + p..j + 2p - 1] \subseteq [m_i - 1..2m_i - 4]$. This implies that $w[b_i - \delta - m_i + j + 3..b_i - \delta - m_i + j + p + 2] = w[b_i - m_i + j + 3..b_i - m_i + j + p + 2]$, and hence a δ-repeat starting at position $(b_i - \delta - m_i + 3 + j)$ in w. ∎

In view of Theorem 8.5.2, all quasi-squares in w', w'' can be found in time $O(m_i \log m_i)$. We conclude that all δ-repeats of $\mathcal{GR}_i^{mid}$ can be reported in time $O(m_i \log m_i + \mathrm{Card}\left(\mathcal{GR}_i^{mid}\right))$. Using $m_i = \min\{\delta, \ell_{i+1}\}$, rewrite this bound as $O(\ell_{i+1} \log \delta + \mathrm{Card}\left(\mathcal{GR}_i^{mid}\right))$.

Putting together cases (a)-(d), all δ-repeats of $\mathcal{GR}'_i$ can be found in time $O(\ell_i) + O(\ell_{i+1} \log \delta) + O(\mathrm{Card}\left(\mathcal{GR}'_i\right))$. Summing up over all $i = 1..k$, we obtain that all δ-repeats of $\mathcal{GR}'$ can be found in time $O(n \log \delta + \mathrm{Card}\left(\mathcal{GR}'\right))$.

Finding δ-repeats of $\mathcal{GR}''$ can be done using a technique similar to the second stage of Maximal-Repetitions from Section 8.4. The key observation here is that each δ-repeat of $\mathcal{GR}''$ occurs inside some factor f_i (i.e. does not contain positions $b_i + 1$ and b_{i+1}). By definition of the factorization, each such δ-repeat is a copy of another δ-repeat occurring to the left. When constructing the Lempel-Ziv factorization, we store, for each factor $f_i = va$, a reference to an earlier occurrence of v. δ-repeats occurring in f_i are located inside v, and are retrieved from its copy by the method used in the second stage of Maximal-Repetitions. The running time of this stage is $O(n + \mathrm{Card}\left(\mathcal{GR}''\right))$.

We conclude with the final result of this section.

THEOREM 8.6.3. *The set $\mathcal{GR}$ of all δ-repeats in a word of length n can be found in time $O(n \log \delta + \mathrm{Card}\,(\mathcal{GR}))$.*

8.6.2. Finding δ-repeats with fixed gap word

The algorithm of Section 8.6.1 can be modified in order to find all δ-repeats with a fixed word between the two copies. Assume v is a fixed word of length δ. Denote by $\mathcal{GR}_v$ the set of δ-repeats of the form uvu, where $|u| \geq 1$. We show that all those repeats can be found in time $O(n \log \delta + \mathrm{Card}\,(\mathcal{GR}_v))$. To do that, we first find, using any linear-time string matching algorithm (for example, the Knuth-Morris-Pratt algorithm) all start occurrences of v in w. For each position i of v, we compute the position $next(i)$, defined as the closest start position of v to the right of i.

From the algorithm of Section 8.6.1 for finding the set $\mathcal{GR}'$, it should be clear that all the δ-repeats of $\mathcal{GR}'$ can be represented by $O(n \log \delta)$ families each consisting of δ-repeats with a given copy length and starting at all positions from a given interval. In other words, each family can be specified by an interval $[i..j]$ and a number p, and encodes all δ-repeats with copy length p starting at positions from $[i..j]$.

From this description, using function $next(i)$, we can easily extract all δ-repeats of $\mathcal{GR}_v$ in time proportional to the number of those. For that, we first assume that each family is specified by the interval of start positions of the gap between the left and right copies (as the copy length p is known for each family, the translation can be trivially computed by just adding p to the interval of start positions). Then we process all the families and extract from each interval those positions which are start positions of an occurrence of v. Using function $next$, this can be easily done in time proportional to the number of such positions.

After processing all families, we have found all δ-repeats from the set $\mathcal{GR}'_v = \mathcal{GR}_v \cap \mathcal{GR}'$ in time $O(n \log \delta + \mathrm{Card}\left(\mathcal{GR}'_v\right))$. Then, using a procedure for finding δ-repeats from $\mathcal{GR}''$, described in Section 8.6.1, we find all δ-repeats from $\mathcal{GR}''_v = \mathcal{GR}_v \cap \mathcal{GR}''$ in time $O(n + \mathrm{Card}\left(\mathcal{GR}''_v\right))$. As $\mathcal{GR}_v = \mathcal{GR}'_v \cup \mathcal{GR}''_v$, all δ-repeats from $\mathcal{GR}_v$ are found in time $O(n \log \delta + \mathrm{Card}\,(\mathcal{GR}_v))$.

8.7. Computing local periods of a word

In this section we focuse on the important notion of *local periods*, that characterize a local periodic structure at each location of the word. The local period at a given position is the root size of the smallest square centered at this position. An importance of local periods is evidenced by the fundamental Critical Factorization Theorem that asserts that there exists a position in the word (and a corresponding factorization), for which the local period is equal to the global period of the word.

Consider a word $w = a_1 \cdots a_n$ over a finite alphabet. Let $w = uv$ be a factorization of w such that $|u| = i$. We say that a non-empty square xx *is* *centered at position i of w* (or *matches w at central position i*) iff the following conditions hold:

(i) x is a suffix of u, or u is a suffix of x,

(ii) x is a prefix of v, or v is a prefix of x.

In the case when x is a suffix of u and x is a prefix of v, we have a square occurring inside w. We call it an *internal square*. If v is a proper prefix of x (respectively, u is a proper suffix of x), the square is called *right-external* (respectively, *left-external*).

The smallest square centered at a position i of w is called the *minimal local* square (hereafter simply *minimal*, for shortness). The *local period* at position i of w, denoted $MLP_w(i)$, is the period of the minimal square centered at this position[1].

Note that for each position i of w, $MLP_w(i)$ is well-defined, and $1 \leq MLP_w(i) \leq |w|$. The relation between local periods and the (global) period of the word is established by the fundamental Critical Factorization Theorem.

THEOREM 8.7.1 (Critical Factorization Theorem). *For each word w, there exists a position i (and the corresponding factorization $w = uv$, $|u| = i$) such that $MLP_w(i) = p(w)$. Moreover, such a position exists among any $p(w)$ consecutive positions of w.*

In this section, we show how the techniques of the previous sections can be used to compute *all* local periods in a word in time $O(n)$, assuming a constant-size alphabet. The method consists of two parts. We first show, in Section 8.7.1, how to compute all *internal* minimal squares. Then, in Section 8.7.2 we show how to compute left- and right-external minimal squares, in particular for those positions for which no internal square has been found. Both computations will be shown to be linear-time, and therefore computing all local periods can be done within linear time too.

8.7.1. Computing internal minimal squares

Finding internal minimal squares amounts to compute, for each position of the word, the smallest square that is centered at this position and occurs entirely inside the word, provided that such a square exists. Thus, throughout this section we will be considering only squares occurring inside the word and, for the sake of brevity, omit the adjective "internal".

The general approach is to use the algorithm for computing maximal repetitions from Section 8.4 in order to retrieve squares which are minimal for some position. One modification of MAXIMAL-REPETITIONS we make here is that we use the s-factorization *with non-overlapping copies* (see Section 8.3.2) instead of the regular s-factorization, used in Section 8.4. This modification, however, does not affect any properties of MAXIMAL-REPETITIONS, including its linear time bound.

[1] Note that the period of a square xx is $|x|$ and not the minimal period of word xx which can be smaller.

Let us now focus on the first stage of MAXIMAL-REPETITIONS. At this step, we find, for each s-factor $f_j = w[b_{j-1}+1..b_j]$, all maximal repetitions that start before b_{j-1} and end inside f_j. According to Corollary 8.1.4, for each possible period p, there can be at most two such repetitions. Each maximal repetition is a run of squares occurring at successive positions in the word. For our purpose here, it will be convenient for us to think of a repetition as an interval of center positions of squares it contains.

In this section, we will need to compute, for a maximal repetition, a *subrun* of squares it contains, that is a subinterval of the corresponding interval of center positions. To illustrate this, consider Theorem 8.4.1. The maximal repetition found according to the theorem corresponds to squares centered at positions $[m + p - LS_{x|y}(p) \;..\; m + LP_y(p+1)]$. If we want to compute only squares centered at positions greater than or equal to m and starting at positions less than or equal to m (as it will be the case below), the interval of centers should be restricted to $[m + \max\{p - LS_{x|y}(p), 0\} \;..\; m + \min\{LP_y(p+1), p\}]$. A similar interval restriction has been done in the previous section for computing δ-repeats.

We present now a linear-time algorithm for computing all internal minimal squares in a given word w. The general description of the algorithm is as follows. First, we compute, in linear time, the s-factorization of w with non-overlapping copies and keep, for each factor f_j, a reference to its non-overlapping left copy. Then we process all factors from left to right and compute, for each factor f_j, all minimal squares ending in this factor. For each computed minimal square, centered at position i, the corresponding value $MLP_w(i)$ is set. After the whole word has been processed, positions i for which values $MLP_w(i)$ have not been assigned are those for which no internal square centered at i exists. For those positions, minimal squares are external, and they will be computed at the second stage, presented in Section 8.7.2.

Let $f_j = w[b_{j-1}+1..b_j]$ be the current factor, $\ell_j = b_j - b_{j-1}$, and let $w[b_{j-1} + 1 - \Delta_j \;..\; b_j - \Delta_j]$ be its non-overlapping left copy (i.e. $\Delta_j \geq \ell_j$). If for some position $b_{j-1} + i$, $1 \leq i < \ell_j$, the minimal square centered at $b_{j-1}+i$ occurs entirely inside f_j, that is $MLP_w(b_{j-1}+i) \leq \min\{i, \ell_j - i\}$, then $MLP_w(b_{j-1}+i) = MLP_w(b_{j-1}+i-\Delta_j)$. Note that $MLP_w(b_{j-1}+i-\Delta_j)$ has been computed before, as the minimal square centered at $b_{j-1}+i$ ends before the beginning of f_j. Based on this observation, we retrieve, in time $O(|f_j|)$, all values $MLP_w(b_{j-1}+i)$ which correspond to squares occurring entirely inside f_j. Therefore, it remains to find those values $MLP_w(b_{j-1}+i)$ which correspond to minimal squares that end in f_j and extend to the left beyond the frontier between f_j and f_{j-1}.

To do this, we use the technique of computing runs of squares from Section 8.4. The idea is to compute all candidate squares and test which of them are minimal. However, this should be done carefully as this can break down the linear time bound, due to a possible super-linear number of all squares. The main trick is to keep squares in runs and to show that there is only a linear number of individual squares which need to be tested for minimality.

We are interested in squares starting at positions less than or equal to b_{j-1}

and ending inside f_j. All these squares are divided into those which are centered inside f_j and those centered to the left of f_j. Two cases are symmetrical and therefore we concentrate on squares centered at positions $[b_{j-1}..b_{j-1}+\ell_j-1]$. We compute all such squares *in the increasing order of periods*. For each $p \in [1..\ell_j]$ we compute the run of all squares of period p centered at positions belonging to the interval $[b_{j-1}..b_{j-1}+\ell_j-1]$, starting at a position less than or equal to b_{j-1}, and ending inside f_j, as explained above. Assume we have computed a run of such squares of period p, and assume that $q < p$ is the maximal period value for which squares have been previously found. If $p \geq 2q$, then we check each square of the run whether it is minimal or not by checking the value $MLP_w(b_{j-1}+i)$. If this square is not minimal, then $MLP_w(b_{j-1}+i)$ has been already assigned a positive value before. Indeed, if a smaller square centered at $b_{j-1}+i$ exists, it has necessarily been already computed by the algorithm (recall that squares are computed in the increasing order of periods). If no positive value $MLP_w(b_{j-1}+i)$ has yet been set, then we have found the minimal square centered at $b_{j-1}+i$. Since there is at most p considered squares of period p (their centers belong to the interval $[b_{j-1}..b_{j-1}+p-1]$), checking all of them takes at most $2(p-q)$ individual checks (as $q \leq p/2$ and $p-q \geq p/2$).

Now assume $p < 2q$. Consider a square $s_q = w[c_q-q+1..c_q+q]$ of period q and center c_q, which has been previously found by the algorithm (square of period q in Figure 8.4). We now prove that we need to check for minimality only those squares s_p of period p which have their center c_p verifying one of the following inequalities :

$$|c_p - c_q| \leq p - q, \text{ or} \tag{8.7.1}$$
$$c_p \geq c_q + q \tag{8.7.2}$$

In words, c_p is located either within distance $p-q$ from c_q, or beyond the end of square s_q.

Lemma 8.7.2. *Let $s_p = w[c_p-p+1..c_p+p]$ be the minimal square centered at some position c_p. Let $s_q = w[c_q-q+1..c_q+q]$ be another square with $q < p$. Then one of inequations (8.7.1),(8.7.2) holds.*

Proof. By contradiction, assume that neither of them holds. Consider the case $c_p > c_q$, case $c_p < c_q$ is symmetric. The situation with $c_p > c_q$ is shown in Figure 8.4. Now observe that word $w[c_q+1..c_p]$ has a copy $w[c_q-q+1..c_p-q]$ (shown with empty strip in Figure 8.4) and that its length is (c_p-c_q). Furthermore, since $c_p - c_q > p - q$ (as inequation (8.7.1) does not hold), this copy overlaps by $p-q$ letters with the left root of s_p. Consider this overlap $w[c_p-p+1..c_p-q]$ (shadowed strip in Figure 8.4). It has a copy $w[c_p+1..c_p+(p-q)]$ and another copy $w[c_p-(p-q)+1..c_p]$ (see Figure 8.4). We thus have a smaller square centered at c_p, which proves that square s_p is not minimal. ■

By Lemma 8.7.2, we need to check for minimality only those squares s_p which verify, with respect to s_q, one of inequations (8.7.1),(8.7.2). Note that there are at most $2(p-q)$ squares s_p verifying (8.7.1), and at most $p-q$ squares

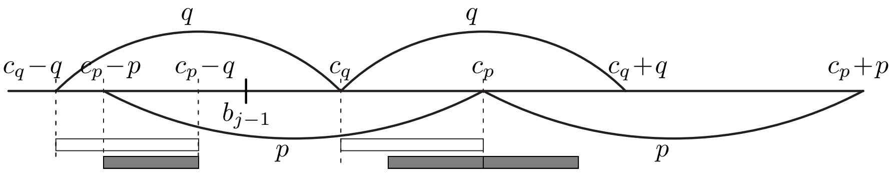

Figure 8.4. Case where neither of inequations (8.7.1),(8.7.2) holds (subcase $c_p > c_q$)

s_p verifying (8.7.2), the latter because s_p must start before the current factor, i.e. $c_p \le b_{j-1} + p$. We conclude that there are at most $3(p-q)$ squares of period p to check for minimality, among all squares found for period p. Summing up the number of all individual checks results in a telescoping sum, and we obtain that processing all squares centered in the current factor can be done in time $O(|f_j|)$.

The Right-Local-Squares algorithm for computing minimal squares centered inside f_j is given below. It is based on algorithms Right-Repetitions and Maximal-Repetitions from Section 8.4. In particular, t_i is defined as in the Maximal-Repetitions algorithm.

A similar algorithm applies to the squares centered on the left of f_j. Note that after processing f_j, all minimal squares ending in f_j have been computed.

To summarize, we need to check for minimality only $O(|f_{j-1}|+|f_j|)$ squares, among those containing the frontier between f_j and f_{j-1}, each check taking a constant time. We also need $O(|f_j|)$ time to compute minimal squares occurring inside f_j. Processing f_j takes then time $O(|f_{j-1}| + |f_j|)$ overall, and processing the whole word takes time $O(n)$.

Theorem 8.7.3. *All internal minimal squares in a word of length n can be computed in time $O(n)$.*

Right-Local-Squares(f_i)

▷ computing minimal squares centered inside f_i
$LS_{t_i|f_i} \leftarrow$ Longest-Suffix-Extension(t_i, f_i)
$LP_{f_i} \leftarrow$ Longest-Prefix-Extension(f_i)
$q \leftarrow 0$
for $p \leftarrow 1$ **to** ℓ_i **do**
  $t_i \leftarrow$ suffix of $f_1 \cdots f_{i-1}$ of length $(2|f_{i-1}| + \max\{2|f_{i-2}|, |f_i|\})$
  $\widehat{LS}(p) \leftarrow \min\{LS_{t_i|f_i}(p), p\}$
  $\widehat{LP}(p+1) \leftarrow \min\{LP_{f_i}(p+1), p\}$
  if $\widehat{LS}(p) + \widehat{LP}(p+1) \geq p$ **then**
    $I \leftarrow [p - \widehat{LS}(p)..\widehat{LP}(p+1)]$ ▷ interval of square centers
    if $p < 2q$ **then**
      $c'_q \leftarrow c_q - b_{j-1}$
      $I \leftarrow I \cap ([c'_q - (p-q)..c'_q + (p-q)] \cup [c'_q + q..\ell_j - 1])$
    for each $i \in I$ **do**
      if $MLP_w(b_{j-1} + i)$ is undefined **then**
        $MLP_w(b_{j-1} + i) \leftarrow p$
    $q \leftarrow p$
    $c_q \leftarrow b_{j-1} + p - \widehat{LS}(p)$

8.7.2. Computing external minimal squares

The algorithm of the previous section allows to compute all *internal* minimal squares of a word. Here we show how to compute *external* minimal squares for those positions which don't have internal squares centered at them.

Consider a word w of length n. We first consider squares which are right-external but not left-external. Those squares are centered at positions in the right half of the word. The case of squares which are left-external but not right-external is symmetrical.

For each position i in the right half of w, we compute a value $RS(i)$ equal to the period of the smallest right-external and not left-external square centered at i, provided such a square exists. We show that all values $RS(i)$ can be computed in linear time using longest extension functions (Section 8.3.1).

Consider a right-external square of period p centered at some position $i \in [\lceil n/2 \rceil..n-1]$, where $n - i < p \leq i$. Observe that $w[i-p+1..n-p] = w[i+1..n]$. This implies that $LS_w(n-p) \geq n-i$. Conversly, if for some $p \in [1..n-1]$, $LS_w(n-p) > 0$, then there exists a family of squares of period p centered at positions $i \in [n - LS_w(n-p)..n-1]$. For $i > n-p$, the square is right-external, otherwise it is internal.

This implies the following algorithm for computing minimal right-external squares. Compute LS_w for all positions of w. For each $j \in [1..n]$, set $LNS_w(j) = LS_w(j)$ if $LS_w(j) < n - j$, and $LNS_w(j) = n - j - 1$ otherwise. For each center position $i \in [\lceil n/2 \rceil..n-1]$, we need to compute the minimal p such that $LNS_w(n-p) \geq n-i$.

Consider all pairs $(j, LNS_w(j))$ for $j \in [1..n]$. If for some pair $(j, LNS_w(j))$, there exists a pair $(j', LNS_w(j'))$ such that $LNS_w(j') \geq LNS_w(j)$ and $j' > j$, then $(j, LNS_w(j))$ carries no useful information for computing minimal right-external squares. We then delete all such pairs $(j, LNS_w(j))$ from consideration by looping through all j from n to 1 and deleting those for which the value $LNS_w(j)$ is smaller than or equal to $\max_{j'>j}\{LNS_w(j')\}$. We then sort the remaining pairs $(j, LNS_w(j))$ in the decreasing order of $LNS_w(j)$. Using bucket sort, this can be done in $O(n)$ time and space.

We now set the values $RS(i)$ as follows. For the first element $(j_0, LNS_w(j_0))$ of the list, we set $RS(i) = 0$ for all $i \in [\lceil n/2 \rceil..n - LNS_w(j_0) - 1]$. We then scan through the ordered list of pairs and for each element $(j, LNS_w(j))$, look at the next element $(j', LNS_w(j'))$, $LNS_w(j) > LNS_w(j')$. For all $i \in [n - LNS_w(j)..n - LNS_w(j') - 1]$, set $RS(i) = n - j$. We then have the following

Lemma 8.7.4. *For each $i \in [\lceil n/2 \rceil..n - 1]$, $RS(i)$ is the smallest period of a right-external square centered at i if such a square exists, and $RS(i) = 0$ otherwise.*

We now turn to squares that are both right-external and left-external. Consider such a square of period p, centered at some position i. Observe that $w[1..n-p] = w[p+1..n]$. Therefore, there exists a *border* of w of size $n-p < n/2$. The largest border corresponds to smallest square. On the other hand, the period of this square is equal to the minimal period $p(w)$ of w. Note that, in general, each local period cannot be greater than $p(w)$, and all minimal squares which are both right-external and left-external have the period equal to $p(w)$.

It is well known that $p(w)$ can be easily computed in linear time (see Chapter 1). We then obtain an $O(n)$ algorithm for computing all minimal squares: first, using Theorem 8.7.3 we compute all minimal internal squares; then, using Lemma 8.7.4 and the above remark, we compute the minimal external squares for those positions for which no internal square has been found at the first stage. This proves the main result.

Theorem 8.7.5. *For a word of length n, all local periods $MLP_w(i)$ can be computed in time $O(n)$.*

8.8. Finding approximate repetitions

In many practical applications, such as DNA sequence analysis, considered repetitions admit a certain variation between copies of the repeated pattern. In other words, repetitions under interest are *approximate repetitions* and not necessarily exact repetitions only. Computing approximate repetitions is the subject of this section.

The simplest notion of approximate repetition is an *approximate square*. An approximate square in a word is a factor uv, where u and v are within a given distance k and the distance measure could be one of those usually used in practical applications, such as Hamming distance or Levenshtein (or edit) distance. Here we focus on the Hamming distance, when the variation between repeated copies can be only letter replacements. An important motivation here is to define structures encoding families of approximate squares, analogous to maximal repetitions in the exact case. In Section 8.8.1, we define two basic structures that we call *K-repetitions* and *K-runs*, where K the number of allowed errors. In Section 8.8.2, we show that all K-repetitions can be found in time $O(nK \log K+S)$, where S is their number. In Section 8.8.3 we show that the same bound holds for K-runs: all of them can be found in time $O(nK \log K+R)$, where R is their number. The latter result implies, in particular, that all approximate squares can be found in time $O(nK \log K+T)$ (T their number). All those algorithms require only $O(n)$ of working space.

8.8.1. K-repetitions and K-runs

Let $h(\cdot,\cdot)$ be the Hamming distance between two words of equal length, that is $h(u,v)$ is the number of mismatches (letter differences at corresponding positions) between u and v. For example, $h(baaacb, bcabcb) = 2$.

A word $s = uv$, such that $|u| = |v|$, is called a *K-square* iff $h(u,v) \leq K$. Reusing the terminology of the exact case, we call $p = |u| = |v|$ the *period* of s, and words u, v the *left* and *right root* of s respectively.

We now want to define a more global structure which would be able to capture "long approximate repetitions", generalizing repetitions of arbitrary exponent in the exact case. As opposed to the exact case, Conditions (i)-(ii) of Proposition 8.1.1 generalize to different notions of approximate repetition. Condition (i) gives rise to the strongest of them: A word $r[1..n]$ is called a *K-repetition* of period p, $p \leq n/2$, iff $h(r[1..n-p], r[p+1..n]) \leq K$.

Equivalently, a word $r[1..n]$ is a K-repetition of period p, if the number of i such that $r[i] \neq r[i+p]$ is at most K. For example, *abaa abba cbba cb* is a 2-repetition of period 4. *abc abc abc abd abd abd abd abd* is a 1-repetition of period 3 but *abc abc abc abb abc abc abc abb* is not.

Another point of view, expressed by Condition (ii) of Proposition 8.1.1, considers a repetition as an encoding of squares it contains. Projecting this to the approximate case, we come up with the notion of run of approximate squares: A word $r[1..n]$ is called a *run of K-squares*, or a *K-run*, of period p, $p \leq n/2$, iff for every $i \in [1..n-2p+1]$, the factor $s = r[i..i+2p-1]$ is a K-square of period p.

Similarly to the exact case, when we are looking for approximate repetitions occurring in a word, it is natural to consider *maximal* approximate repetitions. Those are repetitions extended to the right and left as much as possible provided that the corresponding definition is still verified. Note that the notion of maximality applies both to K-repetitions and to K-runs: in both cases we can extend each repetition to the right and left as long as it verifies the corresponding definition. We will always be interested in maximal K-repetitions and K-runs, without mentioning it explicitly. Note that for both definitions, the maximality requirement implies that if $r = w[i..j]$ is an approximate repetition of period p in $w[1..n]$, then $w[j+1] \neq w[j+1-p]$ (provided $j < n$) and $w[i-1] \neq w[i-1+p]$ (provided $i > 1$). Furthermore, if $w[i..j]$ is a maximal K-repetition, it contains

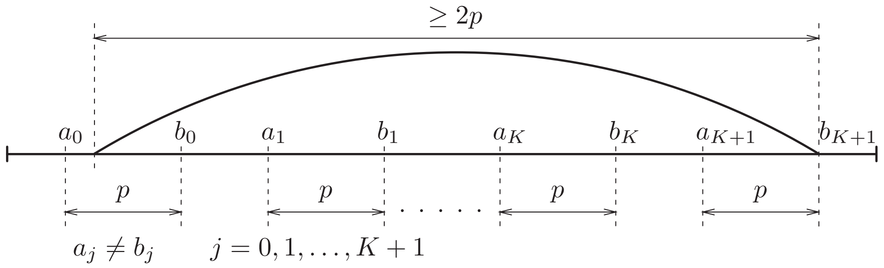

Figure 8.5. Maximal K-repetition

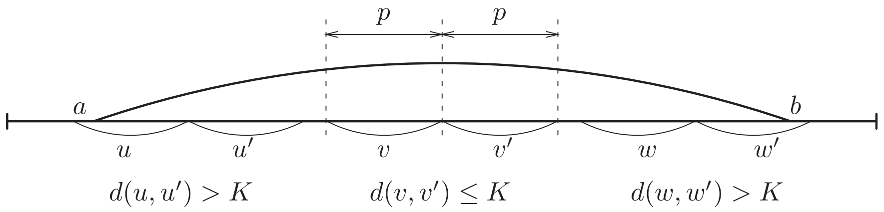

Figure 8.6. Maximal K-run

exactly K mismatches $w[\ell] \neq w[\ell + p]$, $i \leq \ell, \ell + p \leq j$, unless the whole word w contains less than K mismatches (to simplify the presentation, we exclude this latter case from consideration).

Figure 8.5 illustrates the definition of (maximal) K-repetitions and Figure 8.6 that of (maximal) K-run.

Example 8.8.1. The following Fibonacci word contains three 3-runs of of period 6. They are shown in regular font, in positions aligned with their occurrences. Two of them are identical, and contain each four 3-repetitions, shown in italic for the first run only. The third run is a 3-repetition in itself.

010010 100100 101001 010010 010100 1001

10010 100100 101001
10010 100100 10
0010 100100 101
10 100100 10100
0 100100 101001
1001 010010 010100 1
10 010100 1001

In general, each K-repetition is a factor of a K-run of the same period. On the other hand, a K-run in a word is the union of all K-repetitions it contains. Observe that a K-run can contain as many as a linear number of K-repetitions with the same period. For example, the word $(000\,100)^n$ of length $6n$ is a 1-run of period 3, which contains $(2n-1)$ 1-repetitions.

In general, the following lemma holds.

LEMMA 8.8.2. *Let $w[1..n]$ be a K-run of period p and let s be the number of mismatches $w[i] \neq w[i+p]$, $1 \leq i, i+p \leq n$ (equivalently, $s = h(w[1..n-p], w[p+1..n])$). Then w contains $(s-K+1)$ K-repetitions of period p.*

K-runs and K-repetitions provide respectively the weekest and strongest notions of repetitions with K mismatches, and therefore "embrace" all practically relevant repetitions.

8.8.2. Finding K-repetitions

In this subsection we describe how to find, in a given word w, all maximal K-repetitions occurring in w (K is a given constant).

We assume we fixed a minimal bound p_0 for the period of repetitions we are looking for. For example, p_0 can be taken to be $K+1$ having in mind that if a period $p \leq K$ is allowed, then any factor of length $2p$ would be a K-square. This assumption, however, is pragmatic and does not affect the method nor the complexity bounds.

From a general point of view, we are going to apply the same approach as the one used in Sections 8.4-8.7. However, the case of approximate repetitions is more complex and requires a number of modifications.

We start with describing the modification of the basic problem of finding repetitions containing a given position of the word (Theorem 8.4.1 of Section 8.4). Recall that a factor $w[i..j]$ of w is said to *contain* a position ℓ of w iff $i \leq \ell \leq j$. A factor $w[i..j]$ is said to *touch* a position ℓ of w iff $i-1 \leq \ell \leq j+1$. Here, it will be convenient for us to specify the problem as follows: Given a word $w[1..n]$ and a distinguished position $\ell \in [2..n-1]$, find all K-repetitions in w that touch ℓ. Similar to Section 8.4, we distinguish two (non-disjoint) classes of K-repetitions according to whether they have a root on the right or on the left of position ℓ. We focus on K-repetitions of the first class, those of the second class are found similarly.

To apply a method similar to the one of Theorem 8.4.1, we need a generalisation of longest extension functions (Section 8.3.1) that compares factors up to a given Hamming distance. Formally, given a word w and a position ℓ, for every $k = 0, \ldots, K$, we compute the following functions on $p \in [p_0..n-\ell]$:

$$LP^{(k)}_{w,\ell}(p) = \max\{j \mid h(w[\ell+p..\ell+p+j-1], w[\ell+1..\ell+j]) \leq k\}, \quad (8.8.1)$$

$$LS^{(k)}_{w,\ell}(p) = \max\{j \mid h(w[\ell+p-j+1..\ell+p], w[\ell-j+1..\ell]) \leq k\}. \quad (8.8.2)$$

$LP^{(k)}_{w,\ell}(p)$ is the length of the longest factor of w starting at position $\ell+p$ and equal, within k mismatches, to the factor of the same length starting at $\ell+1$.

$LS^{(k)}_{w,\ell}(p)$ is the length of the longest factor ending at position $\ell + p$ and equal, within k mismatches, to the factor ending at position ℓ. These functions are generalizations of longest extension functions considered in Section 8.3.1 and can be computed in time $O(nK)$ using suffix trees combined with the lowest common ancestor computation in a tree (see Gusfield 1997).

Consider now a K-repetition r of period p that has a root on the right of a position ℓ. Note that position $\ell+p$ of w is contained in r, and that r is uniquely defined by the number of mismatches $w[i+p] \neq w[i]$, $i \geq \ell+1$, occurring in r. Let k be the number of those mismatches. The following theorem is a generalization of Theorem 8.4.1.

Theorem 8.8.3. *Let w be a word of length n and let ℓ, $1 < \ell < n$, be a distinguished position of w. There exists a K-repetition of period p which touches position ℓ, and has a root on the right of ℓ, iff for some $k \in [0..K]$,*

$$LS^{(K-k)}_{w,\ell}(p) + LP^{(k)}_{w,\ell}(p+1) \geq p. \tag{8.8.3}$$

When (8.8.3) holds, this repetition starts at position $\left(\ell - LS^{(K-k)}_{w,\ell}(p) + 1\right)$ and ends at position $\left(\ell + p + LP^{(k)}_{w,\ell}(p)\right)$.

Theorem 8.8.3 provides an $O(nK)$ algorithm for finding all considered K-repetitions: compute longest extension functions (8.8.1), (8.8.2) (this takes time $O(nK)$) and then check inequation (8.8.3), for each $k = 0, \ldots, K$ and all $p \in [p_0..\ell-1]$ (this takes time $O(nK)$ too). Every time the inequation is verified, a K-repetition is identified. The computation is summarized in the following algorithm:

Mismatch-Right-Repetitions(w, ℓ)

▷ Find K-repetitions of w which have a root on the right of position ℓ
for all $k = 0, \ldots, K$ **do**
    ▷ compute longest extension functions (8.8.1), (8.8.2)
    $LP^{(k)}_{w,\ell} \leftarrow$ Mismatch-Prefix-Extension(w, ℓ, k)
    $LP^{(k)}_{w,\ell} \leftarrow$ Mismatch-Suffix-Extension(w, ℓ, k)
$\mathcal{R} \leftarrow \emptyset$
for $p \leftarrow p_0$ **to** $\min\{n - \ell + 1, n/2\}$ **do**
    for $k \leftarrow 0$ **to** K **do**
        if $LP^{(k)}_{w,\ell}(p+1) + LS^{(k)}_{w,\ell}(p) \geq p$ **then**
            $r \leftarrow (\ell - LS^{(K-k)}_{w,\ell}(p) + 1, \ell + p + LP^{(k)}_{w,\ell}(p))$
            $\mathcal{R} \leftarrow \mathcal{R} \cup \{r\}$
return $\mathcal{R}$

Finding repetitions having a root on the left of position ℓ is a symmetric problem that can be solved within the same time bound (hereafter referred to as algorithm Mismatch-Left-Repetitions).

We are now ready to describe an extension of the algorithm Maximal-Repetitions from Section 8.4 to compute all K-repetitions in a word w. Consider the Lempel-Ziv factorization $w = f_1 f_2 \cdots f_m$ (Section 8.3.2). The last position of an LZ-factor f_i will be called the *head* of f_i. The algorithm consists of three stages. The **first stage** is based on the following two lemmas.

Lemma 8.8.4. *The suffix of length p of a K-repetition of period p cannot contain $K+1$ consecutive LZ-factors.*

Proof. Each LZ-factor contained in the suffix of length p of a K-repetition must contain at least one mismatch with the letter located p positions to the left. Indeed, if it does not contain a mismatch, it has an exact copy occurring earlier, which contradicts the definition of the Lempel-Ziv factorization. All those mismatches belong to the repetition and there are at most K of them. Therefore, the suffix of length p contains at most K LZ-factors. ■

Divide w into consecutive *blocks* of $(K+2)$ LZ-factors. Let $w = B_1 \cdots B_{m'}$ be the partition of w into such blocks. The last letter of block B_i will be called the *head* of this block. At the first stage, we find, for each block B_i, those K-repetitions which touch the head of B_{i-1} but do not touch that of B_i. The following lemma is analogous to Theorem 8.4.2.

Lemma 8.8.5. *Assume that a K-repetition r touches the head of B_{i-1} but not that of B_i. Then the length of the prefix of r which is a suffix of $B_1 \cdots B_{i-1}$ is bounded by $|B_i| + 2|B_{i-1}|$.*

Proof. Lemma 8.8.4 implies that the suffix of r of period length cannot start before the first letter of B_{i-1}. Therefore, the period of r is bounded by $|B_{i-1}B_i|$. On the other hand, by an argument similar to Lemma 8.8.4, r cannot extend by more than one period to the left of B_{i-1}. This is because otherwise each of the LZ-factors of B_{i-1}, except possibly the last one, would correspond to a mismatch in r, and thus r would contain at least $(K+1)$ mismatches which is a contradiction. Therefore, the length of the prefix of r which is a suffix of $B_1 \cdots B_{i-2}$ is at most $|B_{i-1}| + |B_i|$. ■

Based on Lemma 8.8.5, we apply Mismatch-Right-Repetitions and Mismatch-Left-Repetitions algorithms: Consider the word $w_i = v_i B_i$, where v_i is the suffix of $B_1 \cdots B_{i-1}$ of length $(2|B_{i-1}| + |B_i|)$. Then find, by Mismatch-Right-Repetitions and Mismatch-Left-Repetitions, all K-repetitions in w_i touching the head of B_{i-1} and discard those which touch the head of B_i. The resulting complexity is $O(K(|B_{i-1}| + |B_i|))$.

After processing all blocks, we find all repetitions touching block heads. Observe that repetitions resulting from processing different blocks are distinct. Summing up over all blocks, the resulting complexity of the first stage is $O(nK)$. The repetitions which remain to be found are those which lie entirely within a block – this is done at the next two stages.

At the **second stage** we find all K-repetitions inside each block B_i which touch factor heads other than the block head (i.e. the head of the last LZ-factor of the block). For each B_i, we proceed by the following divide-and-conquer procedure:

Process-Block(B)
▷ Compute K-repetitions which touch factor heads inside a block B
$\mathcal{R} \leftarrow \emptyset$
divide $B = f_i f_{i+1} \cdots f_{i+s}$ into two sub-blocks
  $B' = f_i \cdots f_{\lfloor s/2 \rfloor}$ and $B'' = f_{\lfloor s/2 \rfloor + 1} \cdots f_{i+s}$
▷ compute K-repetitions which touch the head of B'
$h \leftarrow$ position of the head of B'
$\mathcal{R}_1 \leftarrow$ Mismatch-Right-Repetitions(B, h)
$\mathcal{R}_2 \leftarrow$ Mismatch-Left-Repetitions(B, h)
$\mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{R}_1 \cup \mathcal{R}_2$
▷ process recursively B' and B''
$\mathcal{R}' \leftarrow$ Process-Block(B')
$\mathcal{R}'' \leftarrow$ Process-Block(B'')
$\mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{R}' \cup \mathcal{R}''$
return $\mathcal{R}$

The algorithm Process-Block has $\lceil \log_2 K \rceil$ levels of recursion, and since at each step the word is split into disjoint sub-blocks, the whole complexity of the second stage is $O(nK \log K)$.

Finally, at the **third stage**, it remains to compute the K-repetitions which occur entirely inside each Lempel-Ziv factor, i.e. don't contain its first position and don't touch its head. By definition of Lempel-Ziv factorization, each LZ-factor without its head has a (possibly overlapping) copy on the left. Therefore, each of these K-repetitions has another occurrence in that copy. Using this observation, these K-repetitions can be found using the same technique as at the second stage of Maximal-Repetitions: during the construction of the Lempel-Ziv factorization we keep, for each LZ-factor wa, a pointer to a copy of w on the left. Then process all LZ-factors from left to right and recover repetitions occurring inside each LZ-factor from its left copy in the same way that it was done at the second stage of Maximal-Repetitions. The complexity of this stage is $O(n + S)$, where S is the number of repetitions found.

The following theorem summarizes this section.

Theorem 8.8.6. *All K-repetitions in a word of length n can be found in time $O(nK \log K + S)$ where S is the number of K-repetitions found.*

The algorithm K-Repetitions given below summarizes the three stages of the computation of K-repetitions.

K-Repetitions(w)
$(f_1, \cdots, f_m) \leftarrow$ Lempel-Ziv factorization of w
partition $f_1, \cdots, f_m$ into blocks $B_1 \cdots B_{m'}$ of $K+2$ consecutive factors
$\triangleright$ **first stage**
$\mathcal{R} \leftarrow \emptyset$
for $i \leftarrow 2$ **to** m' **do**
    $v_i \leftarrow$ suffix of $B_1 \cdots B_{i-1}$ of length $(2|B_{i-1}| + |B_i|)$
    $\ell \leftarrow |v_i|$
    $\mathcal{R}'_i \leftarrow$ Mismatch-Right-Repetitions($v_i B_i, \ell$)
    $\mathcal{R}''_i \leftarrow$ Mismatch-Left-Repetitions($v_i B_i, \ell$)
    $\mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{R}'_i \cup \mathcal{R}''_i$
$\triangleright$ **second stage**
for $i \leftarrow 1$ **to** m' **do**
    $\mathcal{R}_i \leftarrow$ Process-Block(B_i)
    $\mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{R}_i$
$\triangleright$ **third stage**
for each factor f_j **do**
    retrieve all K-repetitions which occur entirely inside f_j using a procedure similar to the **second stage** of Maximal-Repetitions

8.8.3. Finding K-runs

We now describe an algorithm for finding all K-runs in a word. The general structure of this algorithm is the same as algorithm K-Repetitions – it has three stages playing similar roles. However, the case of K-runs will require a considerable modification and additional algorithmic techniques, especially at the third stage.

At the first and second stages, the key difference is the type of objects we are looking for: instead of computing K-repetitions we now compute *subruns* of K-squares. Formally, a K-subrun is a family of K-squares occurring at successive positions. In other words, a K-subrun is a K-run which is not necessarily maximal. In this section, we *identify a subrun with the interval of end positions of the squares it contains* (note that a different convention has been adopted in Section 8.7.1 where we identified a subrun with the interval of central positions of its squares).

At each point of the first and the second stages when we search for repetitions touching some head position ℓ, we now compute subruns containing those K-squares which touch ℓ, i.e. K-subruns belonging to the interval $[\ell - 1..\ell + 2p]$. As in the case of K-repetitions, we split those squares into those having a root on the left of ℓ (belonging to the interval $[\ell - 1..\ell + p - 1]$) and those having a root on the right of ℓ (belonging to the interval $[\ell + p..\ell + 2p]$). Note that here, however, these two cases are disjoint. The modification of the Mismatch-Right-Repetitions algorithm is the algorithm Mismatch-Right-Subruns below. It computes the subruns of K-squares touching $w[\ell]$ and having a root on the right of it.

Mismatch-Right-Subruns(w, ℓ)

1 ▷ Find subruns of K-squares touching position ℓ in w
▷ and having the right root on the right of ℓ
2 **for** all $k = 0, \ldots, K$ **do**
3 ▷ compute longest extension functions defined by (8.8.1), (8.8.2)
4 $LP^{(k)}_{w,\ell} \leftarrow$ Mismatch-Prefix-Extension(w, ℓ, k)
5 $LP^{(k)}_{w,\ell} \leftarrow$ Mismatch-Suffix-Extension(w, ℓ, k)
6 $\mathcal{L} \leftarrow$ empty list
7 **for** $p \leftarrow p_0$ **to** $\min\{n - \ell + 1, n/2\}$ **do**
8 **for** $k \leftarrow 0$ **to** K **do**
9 **if** $LP^{(k)}_{w,\ell}(p+1) + LS^{(k)}_{w,\ell}(p) \geq p$ **then**
10 $lbound(p,k) \leftarrow \max\{\ell + 2p - LS^{(k)}_{w,\ell}(p), \ell + p\}$
11 $rbound(p,k) \leftarrow \min\{\ell + p + LP^{(k)}_{w,\ell}(p), \ell + 2p\}$
12 $r \leftarrow$ subrun $(lbound(p,k), rbound(p,k))$ of period p
13 **if** $rbound(p,k-1)$ is defined **and**
$lbound(p,k) \leq rbound(p,k-1) + 1$ **then**
14 merge r with the subrun computed for $k-1$
15 **else** add r to $\mathcal{L}$
16 **else** $rbound(p,k) \leftarrow$ undefined
17 **return** $\mathcal{L}$

A major additional difficulty in computing K-runs is that we have to *assemble* them from subruns. To perform the assembling, we need to store subruns in an additional data structure that allows to maintain links between subruns and to merge adjacent subruns into bigger runs. Finally, we have to ensure that the number of subruns we come up with and the work spent on processing them do not increase the resulting complexity bound.

The assembling occurs already in the function Mismatch-Right-Subruns, as intervals (subruns) found for different values of k (**for**-loop at line 8) may overlap or immediately follow each other, in which case we join them into a bigger subrun (lines 13-14). Moreover, those intervals can be disjoint, in which case we organize them in a linked list. A more formal description of the data structure will be given later.

The list of subruns of K-squares that touch $w[l]$ and have a root on the left of it is computed similarly. Once computed, it has to be concatenated with the list computed by Mismatch-Right-Subruns and the rightmost subrun of left-rooted squares has to be merged with the leftmost subrun of right-rooted squares, if those subruns are adjacent.

We now describe the three stages of the algorithm in more details. Given an input word w, we compute the Lempel-Ziv factorization $w = f_1 \cdots f_k$. Unlike the case of K-repetitions, we use here the Lempel-Ziv factorization *with non-overlapping copies* (see Section 8.3.2). We then divide the factorisation into blocks $B_1, \ldots, B_{m'}$, each containing $(K+2)$ consecutive LZ-factors. **At the first stage**, we compute subruns of those K-squares which touch the heads of all blocks $B_1, \ldots, B_{m'}$. For each block B_i, we find the subruns of

K-squares which touch the head of B_i but not that of B_{i+1}. This is done using algorithm Mismatch-Right-Subruns and its couterpart for the left-rooted squares, together with Lemma 8.8.5. Let $\widehat{b}_i$ be the head position of B_i. Then for each period p, K-subruns of period p found at this step belong to the interval $[\widehat{b}_i - 1 \mathinner{..} \min\{\widehat{b}_i + 2p, \widehat{b}_{i+1} - 2\}]$. We call this interval the *explored interval* for $\widehat{b}_i$ and p. For each p, K-subruns found at this step can be seen as non-intersecting subintervals of this explored interval. These K-subruns are stored into a double-linked list, say $\mathcal{L}(i, p)$, in the increasing order of positions. (We leave it to the reader to check that such a list can be easily computed by the algorithm Mismatch-Right-Subruns by making at each step a constant amount of extra work.) For $p > (\widehat{b}_i - \widehat{b}_{i-1})/2 - 1$, the explored interval for $\widehat{b}_{i-1}$ has to be joined with the explored interval for $\widehat{b}_i$, thus forming a bigger explored interval. Accordingly, lists $\mathcal{L}(i-1, p)$ and $\mathcal{L}(i, p)$ are joined. Note that if the last subrun of $\mathcal{L}(i-1, p)$ turns out to be adjacent to the first subrun of $\mathcal{L}(i, p)$, then those two subruns are merged into a single one. All additional operations take a constant time, and the resulting complexity of the first stage is still $O(nK)$.

The second stage is modified in a similar way. Let b_i denote the head position of factor f_i. Recall that at each call of Process-Block we are searching for K-squares occurring between some factor head $b_{j'}$ and another factor head $b_{j''}$, and touching some factor head b_i $(j' < i < j'')$. Moreover, no factor head between $b_{j'}$ and $b_{j''}$ has been processed yet. In this case, the explored interval is $[\max\{b_{j'} + 2p + 1, b_i - 1\} \mathinner{..} \min\{b_i + 2p, b_{j''} - 2\}]$, and we may have to merge it either with the previous explored interval, or with the next one, or both.

After the first and the second stages, we have computed lists of subruns of K-squares that touch all factor heads. Each list stores subruns of K-squares of some fixed period p touching heads of some successive factors $f_i, f_{i+1}, ..., f_j$. Note that for each factor and each period, the corresponding list exists but can be empty. Each such list is accessed through two pointers, associated with the corresponding leftmost (f_i) and rightmost (f_j) factors. We denote these pointers $\mathit{left}_i(p)$ and $\mathit{right}_j(p)$ respectively. These pointers are needed, in particular, for merging explored intervals at the second stage. An important remark is that at each moment there are only $O(n)$ pointers that need to be stored. The key observation is that for each factor head b_i, pointer $\mathit{left}_i(p)$ should be defined only for periods $p \leq (b_i - b_{i'})/2 - 1$, where $b_{i'}$ is the closest head on the left of b_i that has been processed before. Similarly, pointer $\mathit{right}_i(p)$ is defined only for periods $p \leq (b_{i''} - b_i)/2 - 1$, where $b_{i''}$ is the closest head to the right of b_i that has been processed before. On the other hand, all pointer manipulations add only a constant amount of work to each step of the second stage, and then the time complexity of the second stage stays $O(nK \log K)$.

At **the third stage**, we have to find those K-subruns which lie entirely inside LZ-factors. For each period, potential occurrences of these K-subruns correspond precisely to the gaps between explored intervals. Thus, the third stage can be also seen as closing up, for each period, the gaps between explored intervals. The goal of the third stage is to construct, for each period p, a single list $\mathcal{L}[p]$ of all K-subruns of period p occurring in the word. In the beginning of the third stage, $\mathcal{L}[p]$ is initialized to $left_1(p)$.

As before, the key observation here is the fact that each LZ-factor without its head has a copy on the left (here required to be non-overlapping), and the idea is again to process w from left to right and to retrieve the K-subruns occurring inside each LZ-factor from its copy. However, the situation here is different in comparison to the previous section. One difference is that here subruns have to be "copied forward" when we process a factor copy, rather than to be retrieved at the time of processing the factor itself, as done at the second stage of Maximal-Repetitions. Moreover, we may have to "cut out", from a longer list, a sublist of K-subruns belonging to a factor copy and then to "fit" it into the gap between two explored intervals. The "cutting out" may entail splitting K-subruns which span over the borders of the factor copy, and "fitting into" may entail merging those K-subruns with K-subruns from the neighboring explored intervals. Below we describe the algorithm for the third stage, which copes with these difficulties. The algorithm Runs-Third-Stage given below provides a detailed description of the third stage.

During the computation of the Lempel-Ziv factorization, for each LZ-factor $f_i = va$ we choose a copy of v occurring earlier and point from the end position of this copy to the head position of f_i. It may happen that one position has to have several pointers, in which case we organize them in a list. We traverse w from left to right and maintain the rightmost K-run, of each period, which starts before the current position. This K-run is called the *active run* and is denoted $A[p]$ in the Runs-Third-Stage algorithm. To this purpose, we also maintain the following invariant: at the moment we arrive at a position i, we have the list, denoted $S[i]$, of all K-subruns which start at this position. The lists $S[i]$ are maintained according to the following general rule: for each K-subrun starting at the current position, we assign the start position of the next K-subrun in the list provided that this K-subrun exists (instructions 16-19 of Runs-Third-Stage). If the next subrun does not exist, we set a special flag *islast* in order to do it later.

When we arrive at the end position of a copy of a LZ-factor, we have to copy "into the factor" all the K-subruns which this copy contains. Therefore, we scan *backwards* the K-subruns contained in the copy and copy them into the factor (instructions 24-29). After copying these K-subruns, we bridged two explored intervals into one interval, and linked together the two corresponding lists of K-subruns, possibly inserting a new list of K-runs in between (line 30). Copying K-subruns in the backward direction is important for the correction of the algorithm – this guarantees that no K-subruns are missed. It is also for this reason that we need the copy to be non-overlapping with the factor.

The final part of the algorithm (lines 31-40) treats the situation when before executing line 30, $right_{s-1}(p)$ actually refers to the current list $\mathcal{L}[p]$. If, in addition, $\mathcal{L}[p]$ was empty before but became non-empty after the execution of line 30, we have to add the first subrun of $\mathcal{L}[p]$ to the corresponding list $S[i']$ (lines 31-35). If the active run $A[p]$ was the last run in $\mathcal{L}[p]$ before the execution of line 30 but is not the last one after this execution, we have to update the list $S[i']$ for the start position i' of the next subrun (instructions 36-40).

```
Runs-Third-Stage()
▷ A[p] is maintained to be the last considered run of period p
▷ S[i] is maintained to be the list of runs starting at i
for each period p do
    ℒ[p] ← left_1(p)
    islast[p] ← false
    if ℒ[p] is not empty then
        r ← first run of ℒ[p]
        i ← start position of r
        add r to S[i]
        isempty[p] ← false
    else isempty[p] ← true
for each position i ∈ [1..n] do
    for each run r ∈ S[i] do
        p ← period of r
        A[p] ← r
        if r is not the last run in ℒ[p] then
            r′ ← run next to r in ℒ[p]
            i′ ← first position of r′
            add r′ to S[i′]
        else islast[p] ← true
    for each factor copy v ending at position i do
        s ← index of the factor corresponding to v
        for each period p ≤ |v|/2 do
            r ← A[p]
            while r is defined and r contains K-squares inside v do
                r′ ← subrun of all these K-squares
                r″ ← copy of r′ in f_s
                add/merge r″ to/with the head of list left_s(p)
                r ← predecessor of r in ℒ[p]
            link/merge right_{s−1}(p) to/with left_s(p)
            if isempty[p] and ℒ[p] is no more empty then
                r′ ← first run of ℒ[p]
                i′ ← start position of r′
                add r′ to S[i′]
                isempty[p] ← false
            if islast[p] and A[p] is no more the last run in ℒ[p] then
                r′ ← run next to A[p] in ℒ[p]
                i′ ← start position of r′
                add r′ to S[i′]
                islast[p] ← false
```

After the whole word has been traversed, no more gaps between explored intervals exist anymore. This means that for each period p, $\mathcal{L}[p]$ is the list of all K-subruns of period p occurring in the word, which are actually the searched runs.

The complexity of the third stage is $O(n+S)$, where S is the number of resulting K-runs. We show this by an amortized analysis of the Runs-Third-Stage algorithm. Specifically, we show that the *total* number of iterations of each loop in Runs-Third-Stage is either $O(n)$ or $O(S)$. Each iteration of the **for**-loop at line 13 processes a new K-run starting at position i. Therefore there are $O(S)$ iterations of this loop during the whole execution. Each iteration of the **for**-loop at line 21 treats a copy of a distinct Lempel-Ziv factor. Furthermore, the number of iterations of the nested **for**-loop at line 23 is the half of the length of the corresponding factor. Therefore, the total number of iterations of both **for**-loops is $O(n)$. On the other hand, the **while**-loop at line 25 iterates $O(n+S)$ times, as at each iteration, except possibly the first and the last one, it computes a new K-subrun, which becomes a completed K-run at that point. Thus, the *overall* time spent by all internal loops is $O(n+S)$. The main **for**-loop (line 3) makes obviously $O(n)$ iterations, and this completes the proof that the whole complexity of the third stage is indeed $O(n+S)$.

Putting together the three stages, we obtain the main result of this section.

Theorem 8.8.7. *All K-runs can be found in time $O(nK\log K+S)$ where n is the word length and S is the number of K-runs found.*

Once all K-runs have been found, we can easily output all K-squares. We then have the following result.

Corollary 8.8.8. *All K-squares can be found in time $O(nK\log K+S)$ where n is the word length and S is the number of K-squares found.*

8.9. Notes

Section 8.0. We refer to Storer 1988 for applications of repetitions to compression techniques. Galil and Seiferas 1983, Crochemore and Rytter 1995, Cole and Hariharan 1998 illustrate how repetitions are used in pattern matching. Kolpakov et al. 2003 discusses the role of repetitions in DNA sequences. More on biological origin and function of repeated sequences in genome sequences can be learnt, e.g., in Brown 1999.

Section 8.1. Basic definitions and results of word combinatorics, including Proposition 8.1.1 and Theorem 8.1.2 and Proposition 8.1.5, can be found in Lothaire 1997. The exponent is called the *order* in Chapter 8 of Lothaire 2002. Maximal repetitions were called *maximal periodicities* in Main 1989 and *runs* in Iliopoulos et al. 1997.

Section 8.2. The proof of Lemma 8.2.2 is attributed to D. Hickerson and was communicated to us by M. Crochemore and D. Gusfield. The lemma is a weaker and easier-to-prove version of a result from Crochemore and Rytter 1995 asserting that under conditions of Lemma 8.2.2, the stronger inequality $|y|+|z|\leq|x|$

holds. The latter implies that the number of primitively-rooted squares occurring as prefixes of a word w is less than $\log_\varphi |w|$ (φ is the golden ratio), which is a better bound than $\log_{\sqrt{2}} |w|$ implied by Lemma 8.2.2 (cf Theorem 8.2.1). Moreover, the bound $\log_\varphi |w|$ is asymptotically tight, as realized by Fibonacci words.

Lemma 8.2.3(i) is a "folklore result" (see e.g. Pirillo 1997), together with the fact that the common prefix of $f_{n-1}f_{n-2}$ and $f_{n-2}f_{n-1}$ of length $F_n - 2$ is a palindrome. Lemma 8.2.3(ii) appeared in De Luca 1981. The proof given here was communicated to us by J. Berstel. Theorem 8.2.4 was proved in Crochemore 1981.

Lemma 8.2.3(iii) was proved in the PhD thesis of P. Séébold (1985). Lemma 8.2.3(iv) appeared in Karhumäki 1983. Later, repetitions in Fibonacci words have been extensively studied in Mignosi and Pirillo 1992, Pirillo 1997 where it was proved, in particular, that they contain no repetition of exponent greater than $2+\varphi = 3.618..$ but do contain repetitions of exponent greater than $2+\varphi-\varepsilon$ for every $\varepsilon > 0$.

An exact formula for the number of squares Fibonacci word f_n was obtained in Fraenkel and Simpson 1999. It implies that this number is asymptotically $\frac{2}{5}(3-\varphi)nF_n + O(F_n) \approx 0.7962 \cdot F_n \log_2 F_n + O(F_n)$.

It is interesting to note that if we count *distinct* squares rather than square occurrences, their maximal number is asymptotically linearly bounded on the word length. In Fraenkel and Simpson 1999, it has been shown that Fibonacci word f_n contains $2(F_{n-2}-1) = 2(2-\varphi)F_n + O(1)$ distinct squares. In Fraenkel and Simpson 1998, it has been proved that the number of distinct squares in general words of length n is bounded by $2n$ (for an arbitrary alphabet). It is conjectured that this number is actually smaller than n. Thus, in contrast to square occurrences, the maximal number of distinct squares is linear.

A linear bound on the number of maximal repetitions in Fibonacci words was first obtained in Iliopoulos et al. 1997 by presenting a linear-time algorithm enumerating all of those. The direct formula given here was obtained in Kolpakov and Kucherov 2000b. Since Fibonacci words don't contain exponents greater than $(2+\varphi)$, Theorem 8.2.5 implies that the sum of exponents of all maximal repetitions in f_n is no greater than $(2+\varphi)(2F_{n-2}-3) = 2(2-\varphi)(2+\varphi)F_n + O(1) = 2(3-\varphi)F_n + O(1) \approx 2.764 \cdot F_n$. While an exact formula for the sum of exponents is not known, a more precise estimate was obtained in Kolpakov and Kucherov 2000b, where it was shown that the sum of exponents of all maximal repetitions in Fibonacci word f_n is $(C \cdot F_n + o(F_n))$, where $1.922 \leq C \leq 1.926$.

A complete proof of Theorem 8.2.7 can be found in Kolpakov and Kucherov 2000b.

On a different but related topic, several studies have been done on the *minimal*, rather than maximal, number of repetitions in words. In particular, it is well-known (Lothaire 1997) that an arbitrary long word with no squares (and therefore no repetitions at all) can be constructed on a three-letter alphabet. In Fraenkel and Simpson 1995 it was shown that for the binary alphabet, there exists an infinite word containing three *distinct squares* (e.g. 00, 11, 0101) and three is the minimal bound. Complementary, in Kucherov et al. 2003 it was shown that the minimal number of square occurrences in an infinite binary word is, in the limit, a constant fraction of the word length, and that this constant is 0.55080....

Section 8.3. Longest extension functions have been introduced in Main and Lorentz 1984; a closely related idea was used independently in Crochemore 1983. The algorithm presented here for computing longest extension functions is a refinement of the algorithm of Main and Lorentz 1984.

Using s-factorization (called f*-factorization* in Crochemore and Rytter 1994) for finding repetitions was first proposed in Crochemore 1983. The Lempel-Ziv factorization is directly related to the Lempel-Ziv compression algorithm (Ziv and Lempel 1977) and to the underlying definition of complexity of a string (Lempel and Ziv 1976). A discussion on two types of factorization can be found in Gusfield 1997. The linear-time computation of Lempel-Ziv factorization was used in Rodeh et al. 1981. Linear-time construction algorithms for the suffix tree are described in McCreight 1976, Ukkonen 1995 and for the DAWG in Blumer et al. 1985, Crochemore 1986.

Section 8.4. First papers on finding repetitions in words are Crochemore 1981, Slisenko 1983. Crochemore 1981 proposed an $O(n \log n)$ algorithm for finding all occurrences of non-extensible primitively-rooted integer powers in a word. Using a suffix tree technique, Apostolico and Preparata 1983 described an $O(n \log n)$ algorithm for finding all *right-maximal* repetitions, which are repetitions that cannot be extended *to the right* without increasing the period. Main and Lorentz 1984 proposed another algorithm that finds all maximal repetitions in $O(n \log n)$ time. They also pointed out the optimality of this bound under the assumption of unbounded alphabet and under the restriction that the algorithm is based only on letter comparisons.

Crochemore 1983 described a simple and elegant *linear-time* algorithm for finding a square in a word (and thus checking if a word is repetition-free). Another linear algorithm checking whether a word contains a square was proposed in Main and Lorentz 1985.

Using s-factorization, Main 1989 proposed a *linear-time* algorithm which finds all *leftmost* occurrences of distinct maximal repetitions in a word. This algorithm basically corresponds to the first stage of MAXIMAL-REPETITIONS. In particular, Theorems 8.4.2 and 8.4.1 are from Main 1989. The linear-time algorithm presented here is from Kolpakov and Kucherov 1999.

As far as other related works are concerned, Kosaraju 1994 described an $O(n)$ algorithm which, given a word, finds for each position the shortest square starting at this position. Stoye and Gusfield 1998 proposed several algorithms that are based on a unified suffix tree framework. Their results are based on an algorithm which finds in time $O(n \log n)$ all *branching tandem repeats*.

Section 8.5, 8.6. The results of those sections are from Kolpakov and Kucherov 2000a. A more general problem has been considered in Brodal et al. 2000: find all gapped repeats with a gap size belonging to a specified interval. The proposed algorithm has the time complexity $O(n \log n + S)$, where S is the size of the output.

Section 8.7. We refer to Duval 1998, Duval et al. 2001 for studies of properties of local periods. The Critical Factorization Theorem is presented in Lothaire 1997, Choffrut and Karhumäki 1997, Lothaire 2002. For recent developments of the Critical Factorization Theorem see Mignosi et al. 1995.

The results of Section 8.7 are based on Duval et al. 2003.

Section 8.8. The problem of finding approximate squares for both Hamming and edit distances has been first studied in Landau and Schmidt 1993. Computing generalized longest extension functions in time $O(nK)$ can be done by a method based on the suffix tree and the computation of the *nearest common ancestor* described in Gusfield 1997.

The results presented in the section are from Kolpakov and Kucherov 2003.

Algorithms of Sections 8.4 and 8.8 have been implemented in the `mreps` software for finding tandem repeats in DNA sequences. For more information about the software and experimental results obtained with it, we refer to the Web-site `http://www.loria.fr/mreps/` and the publication Kolpakov et al. 2003.