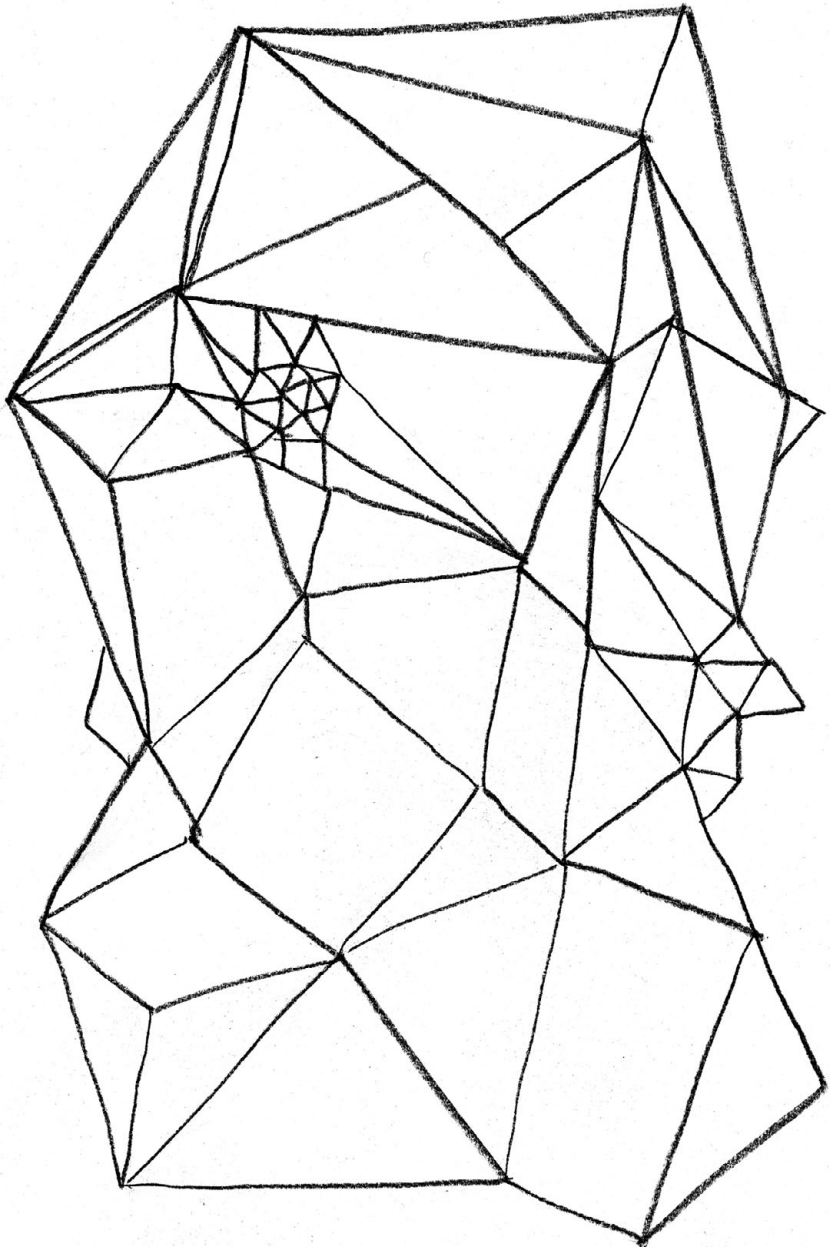
5 Regularities in Words



https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

72 Three Square Prefixes

The combinatorial analysis of square prefixes of a word leads to several consequences useful to design algorithms related to periodicities.

Three non-empty words u, v and w satisfy the square-prefix condition if u^2 is a proper prefix of v^2 and v^2 a proper prefix of w^2. For example, when $u = \texttt{abaab}$, $v = \texttt{abaababa}$, $w = \texttt{abaababaabaabab}$, the prefix condition is met:

```
abaababaab
abaababaabaababa
abaababaabaababababaabaabaabab
```

but u^2 is not a prefix of v nor v^2 a prefix of w, which otherwise would provide a trivial example.

Question. Show that if u^2, v^2 and w^2 satisfy the square-prefix condition and $|w| \leq 2|u|$ then $u, v, w \in z^2z^*$ for some word z.

The conclusion implies in particular that u is not primitive. In fact, this implication holds true if both the square-prefix condition and the inequality $|w| < |u| + |v|$ are met (Three-Square-Prefix Lemma). But the statement in the above question has a stronger conclusion that says we are essentially in the trivial situation where $w^2 = \mathtt{a}^k$ or the like.

Question. Give infinitely many examples of word triples that satisfy the square-prefix condition and for which both $|u|+|v| = |w|$ and u is primitive.

The next question provides a consequence of the Three-Square-Prefix Lemma or of the first statement. The exact upper bound or even a tight bound on the concerned quantity is still unknown.

Question. Show that less than $2|x|$ (distinct) primitively rooted squares can be factors of a word x.

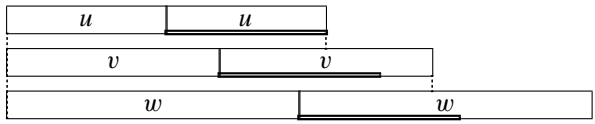
Another direct consequence of the Three-Square-Prefix Lemma is that a word of length n has no more than $\log_\Phi n$ prefixes that are primitively rooted

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

squares. The golden mean Φ comes from the recurrence relation for Fibonacci numbers in the second question.

Solution

Assume that $|w| \leq 2|u|$ as displayed in the picture.

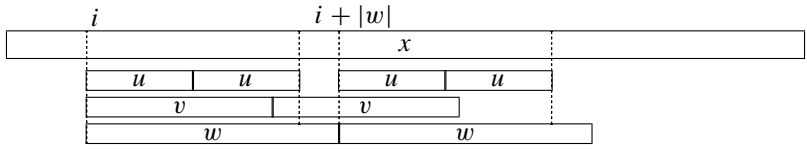


The condition in the first question implies that the three occurrences of u at positions $|u|$, $|v|$ and $|w|$ pairwise overlap. Thus the word u has periods $|v| - |u|$ and $|w| - |v|$ whose sum is no more than $|u|$, and then $q = \gcd(|v| - |u|, |w| - |v|)$ is also a period of u due to the Periodicity Lemma. The word $z = u[0\,..\,p]$, where p is the (smallest) period of u, is a primitive word and as such occurs in u only at positions kp for $k = 0, \ldots, \lfloor |u|/p \rfloor$. Period p is also a divisor of q because $q < |u|/2$.

The word z occurs at position $|u|$ on w^2 and then at position $|u| + |v| - |w|$ on u. Since $|u| + |v| - |w|$ and $|w| - |v|$ are multiples of p, their sum $|u|$ is, and then u is an integer power of z; thus $u \in z^2z^*$. The same holds for v and w because $|v| - |u|$ and $|w| - |v|$ are multiples of $p = |z|$.

The infinite word $\mathbf{s}$, limit of the sequence defined by $s_1 = \texttt{aab}$, $s_2 = \texttt{aabaaba}$ and $s_i = s_{i-1}s_{i-2}$ for $i \geq 3$, contains an infinity of prefix triples that answer the second question. The first triple lengths are $(3, 7, 10)$, $(7, 10, 17)$, $(10, 17, 27)$. The infinite Fibonacci word shows a similar behaviour.

To count the number of primitively rooted squares that are factors of a word x, assign to each its rightmost starting position on x. If ever a position i is assigned to three squares u^2, v^2 and w^2 like in the picture below, due to the statement of the first question, since u is primitive, the shortest square u^2 is a proper prefix of w. Then u^2 reoccurs at position $i + |w|$, which contradicts the fact that i is the rightmost starting position of u^2. Therefore, no more than two squares can be assigned to a given position. And since the last position of x is not considered, the total number of primitively rooted square factors is less than $2|x|$.



https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

Notes

The Three-Square-Prefix Lemma and consequences are by Crochemore and Rytter [97] (see also [74, chapter 9] and [176, chapters 8 and 12]). The first statement and variations on the lemma are by Bai et al. [22].

The problem of counting square factors and the present result are by Fraenkel and Simpson [118]. Direct simple proofs are by Hickerson [141] and Ilie [146]. Slightly improved upper bounds are by Ilie [147] and by Deza et al. [103].

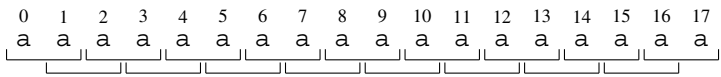
73 Tight Bounds on Occurrences of Powers

The problem considers lower bounds on the number of occurrences of integer powers occurring in a word.

An integer power is a word in the form u^k for some non-empty word u and some integer $k > 1$. The size of the set of square factors (Problem 72) and the number of runs (Problem 86) in a word are known to be linear in the length of the word. This contrasts with the number of occurrences of integer powers that does not satisfy this property.

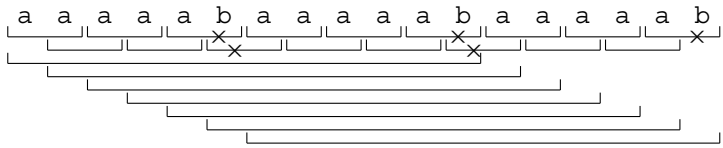
To avoid trivial lower bounds we consider primitively rooted integer powers, that is, powers of the form u^k, where u is a primitive word (i.e., not itself a power).

To start with, let us consider the word $\mathtt{a}^n$. Though it contains a quadratic number occurrences of squares, it contains exactly $n - 1$ occurrences of primitively rooted squares (underlined below).



But if a few occurrences of a are changed to b in the word (see below) the number of primitively rooted squares increases, although some occurrences of short squares disappear (when n is large enough).

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press



Consider the sequence of words defined by

$$\begin{cases} x_0 = \mathtt{a}^5\mathtt{b}, \\ x_{i+1} = (x_i)^3\mathtt{b}, \quad \text{for } i \geq 0. \end{cases}$$

Question. Show that x_i contains asymptotically $\Omega(|x_i| \log |x_i|)$ occurrences of primitively rooted squares.

In fact, the property on squares also holds for powers of any integer exponent $k \geq 2$.

Question. For a given integer $k \geq 2$, define a sequence of words y_i, for $i \geq 0$, containing asymptotically $\Omega(|y_i| \log |y_i|)$ occurrences of primitively rooted kth powers.

Notice the bound is tight due to the upper bound on square prefixes in Problem 72.

Solution

Consider the sequence of words x_i of length ℓ_i and let c_i be the number of occurrences of primitively rooted squares in x_i.

We have (looking at $(x_0)^3$ in the above picture and accounting for the suffix occurrence of $\mathtt{bb}$ in x_1)

$$\begin{cases} \ell_0 = 6, & c_0 = 4, \\ \ell_1 = 19, & c_1 = 20. \end{cases}$$

Note that all short squares appear in each occurrence of $\mathtt{a}^5\mathtt{b}$ in x_1 and that $\mathtt{a}^5\mathtt{b}$ itself is a primitive word. The same property holds true by induction for all squares occurring in x_i. This produces the recurrence relations, for $i > 0$

$$\begin{cases} \ell_1 = 19, & c_1 = \ell_1 + 1, \\ \ell_{i+1} = 3\ell_i + 1, & c_{i+1} = 3c_i + \ell_i + 2. \end{cases}$$

Then, asymptotically we get $\ell_{i+1} \approx 3^i \ell_1$, $c_{i+1} > 3^i c_1 + i3^{i-1}\ell_1$ and $i \approx \log |x_{i+1}|$, which proves the statement of the first question.

When k is the exponent of considered powers, for some positive integer m, we can define the sequence of words

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

$$\begin{cases} y_0 = \mathtt{a}^m\mathtt{b}, \\ y_{i+1} = (y_i)^{k+1}\mathtt{b}, \quad \text{for } i > 0, \end{cases}$$

which induces

$$\begin{cases} \ell_{i+1} = (k+1)\ell_i + 1, \quad c_{i+1} = (k+1)c_i + \ell_i + 2, \end{cases}$$

and also leads to an $\Omega(|y_i| \log |y_i|)$ lower bound on the number of occurrences of primitively rooted kth powers.

Notes

The lower bound on primitively rooted squares holds for Fibonacci words [64]. The proof uses the fact that Fibonacci words have no factors that are 4th powers. The bound has also been shown by Gusfield and Stoye [135].

The asymptotic lower bound for occurrences of kth power is shown in [72], which inspired the present proof.

74 Computing Runs on General Alphabets

The goal of the problem is to design an algorithm for computing runs in a word without any extra assumption on the alphabet. To say it differently, the algorithm should use the equality model on letters, that is, use only $=/\neq$ letter comparisons when necessary.

Problem 87 deals with computing runs on linear-sortable alphabets, a necessary condition to obtain a linear-time algorithm.

A run in a word x is a maximal periodicity or a maximal occurrence of a periodic factor of x. Formally, it is an interval of positions $[i \,..\, j]$ for which the (smallest) period p of $x[i \,..\, j]$ satisfies $2p \leq j - i + 1$, and both $x[i-1] \neq x[i+p-1]$ and $x[j+1] \neq x[j-p+1]$ when the inequalities make sense. The centre of run $[i \,..\, j]$ is the position $i + p$.

To avoid reporting the same run twice, they can be filtered out according to their centre. To do so, we say that a run is right centred in the product uv of words if it starts at a position on u and has its centre on v. And we say it is left centred in uv if its centre is on u and it ends in v.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

Question. Design a linear-time algorithm that finds right-centred runs occurring in a product uv of two words.

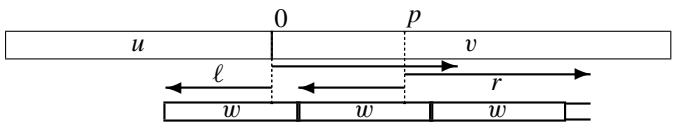
[**Hint:** Use prefix tables, see Problem 22.]

Question. Design an algorithm that computes all the runs in a word of length n in time $O(n \log n)$ in the equality model.

[**Hint:** Use a divide-and-conquer approach.]

Solution

To answer the first question, we can look for the sought runs in the increasing order of their periods. As shown in the picture, given a potential period p of a run, we just have to check how long the associated factor $v[0\,..\,p-1]$ matches to its left and to its right. These are longest common extensions (LCE) from two positions, for instance $r = lcp(v, v[p\,..\,|v|-1)$. If the sum of extension lengths is at least the period a run is detected.



The length r of the right extension is simply given by the prefix table of v. The length ℓ of the left extension is computed similarly with the prefix table of $z = u^{\mathrm{R}}\#v^{\mathrm{R}}u^{\mathrm{R}}$, where # does not appear in uv.

If the condition $\ell \le p$ holds at line 6 in the algorithm below, the potential run is centred on v, as required. The *offset*, position on x of one of its factors, uv, is added to report runs as intervals of positions on x (instead of uv) in Algorithm RUNS below.

```
RIGHT-CENTRED-RUNS(u, v non-empty words, offset)
1  pref_v ← PREFIXES(v)
2  pref_z ← PREFIXES(u^R#v^Ru^R)
3  for p ← 1 to |v| − 1 do
4      r ← pref_v[p]
5      ℓ ← pref_z[|u| + |v| − p + 1]
6      if ℓ ≤ p and ℓ + r ≥ p then
7          Output run [|u| − ℓ .. |u| + p + r − 1] + offset
```

The design of LEFT-CENTRED-RUNS to compute runs of uv centred on u follows the same scheme and is done symmetrically.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

The running time of both algorithms depends on the complexity to compute a prefix table. It is linear in the input length, as shown in Problem 22. Moreover, only $=/\neq$ comparisons are used during the calculation.

Eventually, to compute all runs in a word x, the process divides x into two words of similar length as in the algorithm below. Runs are obtained by calling RUNS$(x, n, 0)$. As a consequence of the running times of the two previous algorithms, the whole computation runs in time $O(n \log n)$ in the comparison model.

```
RUNS(x non-empty word of length n, offset)
1  if n > 1 then
2      (u, v) ← (x[0 .. ⌊n/2⌋], x[⌊n/2⌋ + 1 .. n − 1])
3      RUNS(u, ⌊n/2⌋ + 1, offset)
4      RUNS(v, n − ⌊n/2⌋ − 1, offset + ⌊n/2⌋ + 1)
5      RIGHT-CENTRED-RUNS(u, v, offset)
6      LEFT-CENTRED-RUNS(u, v, offset)
```

Note that some runs may be reported several times by the algorithm. This happens when a long run in the first half of the word overflows on the second half of it. Some filtering is needed to get a clean list of runs.

Notes

The present technique to compute runs is presented in [84] together with other solutions running in the same time according to the computational model. In this model, the algorithm is optimal due to a result by Main and Lorentz [179], who gave a $\Omega(n \log n)$ lower bound for the detection of a square in a word.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

75 Testing Overlaps in a Binary Word

The goal of the problem is to design an efficient algorithm to test whether a binary word contains an overlap factor. An overlap is a factor whose exponent is larger than 2. A word contains an overlap if equivalently it has a factor of the form $auaua$, where a is a letter and u is a word.

The Thue–Morse word $\mu^\infty(\mathtt{a})$ is an example of an infinite overlap-free word. It is generated by the Thue–Morse morphism μ (defined by $\mu(\mathtt{a}) = \mathtt{ab}$ and $\mu(\mathtt{b}) = \mathtt{ba}$) that preserves overlap-freeness of words.

For a binary word x we define its decomposition $uyv = x$, formally a triple (u, y, v): $|u|$ is the smallest position on x of a longest factor y that belongs to $\{\mathtt{ab}, \mathtt{ba}\}^+$. The decomposition is called an RS-factorisation if $u, v \in \{\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{bb}\}$. RS-factorisations are transformed into words in $\{\mathtt{ab}, \mathtt{ba}\}^*$ by the partial functions f or g as follows (c and d are letters and the bar function exchanges letters a and b):

$$f(uyv) = \begin{cases} y & \text{if } u = v = \varepsilon \\ \bar{c}cy & \text{if } u = c \text{ or } cc \text{ and } v = \varepsilon \\ yd\bar{d} & \text{if } u = \varepsilon \text{ and } v = d \text{ or } dd \\ \bar{c}cyd\bar{d} & \text{if } u = c \text{ or } cc \text{ and } v = d \text{ or } dd \end{cases}$$

$$g(uyv) = \begin{cases} y & \text{if } u = v = \varepsilon \\ \bar{c}cy & \text{if } u = c \text{ or } cc \text{ and } v = \varepsilon \\ yd\bar{d} & \text{if } u = \varepsilon \text{ or } c \text{ or } cc \text{ and } v = d \text{ or } dd \end{cases}$$

OverlapFree(x non-empty binary word)

```
1  while |x| > 6 do ▷ below c and d are letter variables
2      uyv ← RS − factorisation(x)
3      if uyv is not an RS-factorisation then
4          return FALSE
5      if [u = cc and (ccc or cc̄cc̄c prefix of uy)] or
         [v = dd and (ddd or dd̄dd̄dd suffix of uy)] then
6          return FALSE
7      if (u = c or u = cc) and (v = d or v = dd) and
         uyv is a square then
8          x ← μ^{-1}(g(uyv))
9      else x ← μ^{-1}(f(uyv))
10 return TRUE
```

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

Question. Show that Algorithm OverlapFree runs in linear time for testing if its binary input word is overlap free.

Solution

The proof of correctness of OverlapFree is out of the scope of the problem but we give a few properties that are used to do it. The proof relies on the property of decomposition of overlap-free words used in the algorithm. To state it, let O and E be the sets

$$O = \{\mathtt{aabb}, \mathtt{bbaa}, \mathtt{abaa}, \mathtt{babb}, \mathtt{aabab}, \mathtt{bbaba}\},$$

$$E = \{\mathtt{abba}, \mathtt{baab}, \mathtt{baba}, \mathtt{abab}, \mathtt{aabaa}, \mathtt{bbabb}\}.$$

Let x be an overlap-free binary word. Then, if x has a prefix in O, $x[j] \neq x[j-1]$ for each odd position j satisfying $3 \leq j \leq |x| - 2$. And, if x has a prefix in E, $x[j] \neq x[j-1]$ for each even position j satisfying $4 \leq j \leq |x|-2$. Consequently, if the word is long enough, it has a long factor that belongs to $\{\mathtt{ab}, \mathtt{ba}\}^+$. Namely, if $|x| > 6$, x uniquely factorises into uyv, where $u, v \in \{\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{bb}\}$ and $y \in \{\mathtt{ab}, \mathtt{ba}\}^+$.

Iterating the decomposition, the word x uniquely factorises into

$$u_1 u_2 \dots u_r \cdot \mu^{r-1}(y) \cdot v_r \dots v_2 v_1,$$

where $|y| < 7$ and $u_s, v_s \in \{\varepsilon, \mu^{s-1}(\mathtt{a}), \mu^{s-1}(\mathtt{b}), \mu^{s-1}(\mathtt{aa}), \mu^{s-1}(\mathtt{bb})\}$.

As for the running time of OverlapFree, note that instructions in the while loop execute in time $O(|x|)$. Since the length of x is essentially halved at each step by the action of the Thue–Morse morphism, this results in a total linear-time execution of the loop. The last test is done on a word of length at most 6, and therefore takes constant time, which proves the whole algorithm runs in time $O(|x|)$.

Notes

Most properties of overlap-free words concerned by this problem have been shown by Restivo and Salemi [207], who deduced the polynomial growth of their number according to the length. The present algorithm is by Kfoury [158], who proved tighter properties on overlap-free words and eventually reduced slightly the previous bound on the number of overlap-free words of a given length.

The present algorithm gives a direct solution to the question. A more generic solution that requires more tools is given by Problem 87 with an algorithm that computes all runs in a word. To detect overlap-freeness with it, it suffices to check that the exponent of all runs is exactly 2 (it cannot be smaller by the run definition). The latter algorithm also runs in linear time on binary words.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

76 Overlap-Free Game

The game relies on the notion of overlaps occurring in words. A word contains an overlap (factor of exponent larger than 2) if one of its factors is of the form $avava$ for a letter a and a word v.

The overlap-free game of length n is played between two players, Ann and Ben, on the alphabet $A = \{0, 1, 2, 3\}$. Players extend an initially empty word by alternately appending a letter to the word. The game ends when the length of the emerging word is n.

We assume that Ben makes the first move and that n is even. Ann wins if there is no overlap in the final word. Otherwise, Ben is the winner.

Ann's winning strategy. Let $d \in A$ be the letter Ann adds during the kth move. If Ben just added the letter c, d is defined by

$$d = c \oplus \mathbf{f}[k],$$

where $x \oplus y = (x + y) \bmod 4$ and $\mathbf{f} = f^{\infty}(1)$ is the infinite square-free word obtained by iterating the morphism f defined on $\{1,2,3\}^*$ by $f(1) = 123$, $f(2) = 13$ and $f(3) = 2$ (see Problem 79). Word $\mathbf{f}$ and a series of moves look like

f		1		2		3		1		3		2		1		2	⋯
moves	0	1	2	0	0	3	2	3	3	2	3	1	1	2	1	3	⋯

Question. Show that Ann always wins against Ben in the overlap-free game of any even length n when using Ann's strategy.

[**Hint:** The sum of letters of any odd-length factor of $\mathbf{f}$ is not divisible by 4.]

Solution

To answer the question we definitely use the fact the word $\mathbf{f}$ is square free but also use here the crucial property stated in the hint.

Proof of the hint. Let $\alpha = |v|_1$, $\beta = |v|_2$ and $\gamma = |v|_3$ be the respective number of occurrences of letters 1, 2 and 3 in v. Due to its morphic definition the word $\mathbf{f}$ is composed of blocks 123, 13 and 2. Hence there is always a single occurrence of 1 between any two (not adjacent) consecutive occurrences of 3's. This implies $|\alpha - \gamma| \leq 1$.

If $|\alpha - \gamma| = 1$, $\alpha + 2\beta + 3\gamma$ is not divisible by 2 and consequently not divisible by 4.

Otherwise $\alpha = \gamma$ and then β is odd because the length $\alpha + \beta + \gamma = |v|$ is odd. This implies $2\beta \bmod 4 = 2$. Hence $\alpha \oplus 2\beta \oplus 3\gamma = 2\beta \bmod 4 = 2$, and the sum of letters of v is not divisible by 4, which achieves the hint's proof.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

Correctness of Ann's strategy. We prove it by contradiction. Assume that at some moment in the game the word w gets an overlap, which is then of the form $cvcvc$ for $c \in A$. Let us distinguish two cases.

Case $|cv|$ is even. Choosing either $u = cv$ or $u = vc$, the word w contains a square uu for which $|u|$ is even and its first letter is a Ben's move in the game. The square looks like

$$uu = b_1a_1b_2a_2 \dots b_ka_k\ b_1a_1b_2a_2 \dots b_ka_k,$$

where b_i's correspond to Ben's moves and a_i's to Ann's moves. Denoting $x \ominus y = (x - y) \bmod 4$, the word $e_1e_2 \dots e_k\ e_1e_2 \dots e_k$, where $e_i = (b_i \ominus a_i)$ is a square in **f**. So this case is impossible because **f** is square free.

Case $|cv|$ is odd. As above, the word w contains a square uu for which $|u|$ is odd and its first letter corresponds to a Ben's move. Observe that $|u| > 1$, since the second letter is from Ann's move and is different from Ben's move.

We demonstrate the proof for $|u| = 7$, which clearly shows the pattern of the general proof. Let $u = b_1a_1\ b_2a_2\ b_3a_3\ b_4$, where b_i are Ben's moves and a_i's are Ann's moves. The square is of the form

$$uu = b_1a_1\ b_2a_2\ b_3a_3\ b_4b_1\ a_1b_2\ a_2b_3\ a_3b_4.$$

Consequently **f** contains the factor $e_1e_2e_3e_4e_5e_6e_7$, where

$e_1 = a_1 \ominus b_1$, $e_2 = a_2 \ominus b_2$, $e_3 = a_3 \ominus b_3$, $e_4 = b_1 \ominus b_4$,
$e_5 = b_2 \ominus a_1$, $e_6 = b_3 \ominus a_2$, $e_7 = b_4 \ominus a_3$.

We get

$$e_1 \oplus e_2 \oplus e_3 \oplus e_4 \oplus e_5 \oplus e_6 \oplus e_7 \ = \ 0,$$

writing the sum as
$(a_1 \ominus b_1) \oplus (b_1 \ominus b_4) \oplus (b_4 \ominus a_3) \oplus (a_3 \ominus b_3) \oplus (b_3 \ominus a_2) \oplus (a_2 \ominus b_2) \oplus (b_2 \ominus a_1)$.
But this is impossible because from the hint the sum of letters of an odd-length factor of **f** is not divisible by 4.

To conclude, since no case is possible, w contains no overlap and Ann's strategy causes her to win.

Notes

The solution of the problem is a version of the Thue game strategy presented in [132]. Note Ben has a simple winning strategy if the game is played with only three letters and Ann sticks to a similar strategy.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

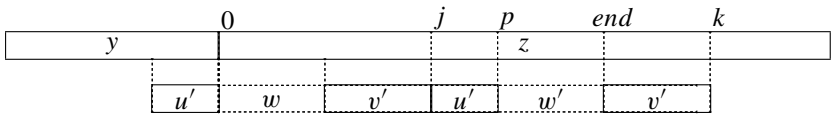
77 Anchored Squares

When searching, in a divide-and-conquer way, a word for square factor it is natural to look for squares in the product of two square-free words. The problem deals with the latter question and extends to a square-freeness test running in $O(n \log n)$ time on a word of length n.

The method based on prefix tables (see Problem 74) achieves the same goal but requires tables of size $O(n)$ while the present solution needs only a few variables in addition to input.

Let y and z be two square-free words. Algorithm RIGHT tests if yz contains a square centred in z only. Other squares in the product can be found symmetrically.

The algorithm examines all possible periods of a square. Given a period p (see picture), the algorithm computes the longest common suffix $u' = z[j \,..\, p-1]$ between y and $z[0 \,..\, p-1]$. Then it checks if $z[0 \,..\, j-1]$ occurs at position p on z, scanning z from the right position $k-1$ of the potential square. If it is successful, a square is found.



RIGHT(y, z non-empty square-free strings)

```
1   (p, end) ← (|z|, |z|)
2   while p > 0 do
3       j ← min{q : z[q .. p − 1] suffix of y}
4       if j = 0 then
5           return TRUE
6       k ← p + j
7       if k < end then
8           end ← min{q : z[q .. k − 1] suffix of z[0 .. j − 1]}
9           if end = p then
10              return TRUE
11      p ← max{j − 1, ⌊p/2⌋}
12  return FALSE
```

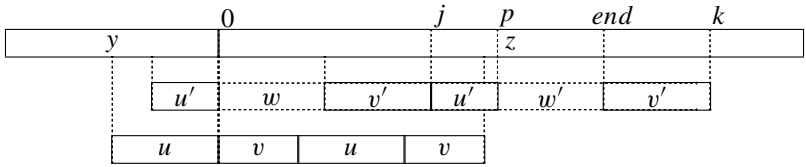
https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

Question. Show both that Algorithm RIGHT returns true if and only if the word yz contains a square centred in z and that its running time is $O(|z|)$ with only constant extra space.

In Algorithm RIGHT the role of variable *end* and instructions at lines 7 and 11 are crucial to get the running time announced in the question. Note that it does not depend on the length of y.

Solution

Correctness of RIGHT. This relies on the next statement, whose combinatorial proof is left to the reader. It is illustrated by the following picture, in which $u' = z[j\,..\,p-1]$ is the longest common suffix of y and $z[0\,..\,p-1]$ computed at line 3, and v' is the longest common suffix of $z[0\,..\,j-1]$ and $z[p\,..\,k-1]$ possibly computed at line 8. A test in the algorithm can be added to discard an empty u', since it cannot lead to a square because z is square-free.



***Lemma* 6** *Let y and z be two square-free words and vuv be the shortest prefix of z for which u is a suffix of y. Let u' and v' be as described above, and w and w', $|w| = |w'|$, as in the picture.*

Assume that vu is a proper prefix of $wv'u'$. Then, vu is a proper prefix of wv' or $|vu| \leq |wv'u'|/2$. The word vuv is also a prefix of $wv'u'w'$.

The correctness follows from the conclusion of the lemma after checking that u' and v' are correctly computed with indices j and *end* respectively. The next value of p assigned at line 11 applies the first conclusion of the lemma. The second conclusion is used at line 7 after the assignment of the variable k to skip a useless computation of v' when the condition is not met.

Running time of RIGHT. The worst-case running time of Algorithm RIGHT relies on the maximum number of letter comparisons, which we evaluate. Let p' and p'', $p' > p''$, be two consecutive values of the variable p during a run of the algorithm; that is, p' is the value of p when entering the while loop, and p'' is its value at the end of the loop execution.

If a test is added to discard an empty u' we have $p'' = p' - 1$ after 1 comparison. Otherwise we have $p'' = \max\{j' - 1, \lfloor p'/2 \rfloor\}$, where j' is the value of j after execution of line 3. If $j' - 1$ is the maximum, the number of

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

comparisons at this line is $p' - p''$. Otherwise the number of comparisons is no more than $2(p' - p'')$. Summing up on all the executions of the loop we get no more than $2|z|$ letter comparisons at line 3.

Due to the role of the variable *end*, positive letter comparisons on letters of $z[p\,..\,end - 1]$ at line 8 are all at different positions on z, which gives a maximum of $|z|$ comparisons. Besides, there is at most one negative comparison for each value of p. Then no more than $2|z|$ letter comparisons at line 8. Therefore the total number of letter comparisons is no more than $4|z|$, yielding a $O(|z|)$ running time.

Notes

The bound on the number of letter comparisons performed by Algorithm RIGHT on words y and z is $2|z| - 1$ when y is a Zimin word (see Problem 43) and $z = \#y$ for a letter # not appearing in y, for example when $y =$ `abacabadabacaba`.

The first design of Algorithm RIGHT with the constant extra space feature is by Main and Lorentz [179]. The slight improvement given here appears in [66].

A solution to the question using prefix tables or analogue tables, like in Problem 74, is described in [74, 98]. The computation of j and of *end* in Algorithm RIGHT are often referred to as Longest Common Extensions (LCEs). They can be found in constant time after some preprocessing when the alphabet is linearly sortable; see, for example, the method designed by Fischer and Heun in [115]. This latter type of solution is used in Problem 87.

Solutions of the question with a dual Algorithm LEFT lead to an algorithm that tests the square-freeness of a word of length n and runs in $O(n \log n)$ time using only constant extra space. The optimality is proved in [179]. On a fixed-size alphabet, it also leads to a linear-time square-freeness test (see [67]) using a factorisation of the word similar to the factorisation by Lempel and Ziv described in Chapter 6.

Extension to the computation in $O(n \log n)$ time of runs occurring in a word of length n is treated in [84].

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

78 Almost Square-Free Words

Testing if a word that contains no short squares is square free can be done in a more efficient and simpler way than with the methods treating ordinary words. This is the object of the problem.

A word w is said to be *almost square free* if it does not contain any square factor of length smaller than $|w|/2$. Such words have a useful property stated in the observation, in which $Occ(z, w)$ denotes the set of starting positions of occurrences of z in the word w.

Observation 1. If z is a factor of length $|w|/8$ of an almost square-free word w, then z is non-periodic (its smallest period is larger than $|z|/2$), $|Occ(z, w)| < 8$ and $Occ(z, w)$ can be computed in linear time and constant space.

Under the hypothesis of the observation, the computation of $Occ(z, w)$ is realised, for example, by Algorithm NaiveSearch, a naive version of Algorithm KMP.

```
NaiveSearch(z, w non-empty words)
1  (i, j) ← (0,0)
2  Occ(z, w) ← ∅
3  while j ≤ |w| − |z| do
4      while i < |z| and z[i] = w[j + i] do
5          i ← i + 1
6      if i = |z| then
7          Occ(z, w) ← Occ(z, w) ∪ {j}
8      (j, i) ← (j + max{1, ⌊i/2⌋}, 0)
9  return Occ(z, w)
```

Question. Design an algorithm that checks in linear time with constant space if an almost square-free word w is square free, assuming for simplicity that $|w| = 2^k$, $k \geq 3$.

[**Hint:** Use a factorisation of w into short factors, and apply Algorithm NaiveSearch and Observation 1.]

Solution

The idea of the solution is to factorise w into short blocks that a large square cannot miss.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

To do so, let $\ell = 2^{k-3}$ and $z_r = w[r \cdot \ell \,..\, r \cdot \ell + \ell - 1]$ for $r = 0, 1, \ldots, 7$. Let also $Z = \{z_0, z_1, \ldots, z_7\}$.

Consider the operation $\mathit{TestSquare}(p, q)$ that checks if there is a square of length $2(q - p)$ in w containing positions p, q, $p \leq q$. The operation is easily performed in time $O(n)$ and constant space using extensions to the left and to the right like in Problem 74. Based on the operation, the following fact is a straightforward observation.

Observation 2. If w is almost square free then it contains a square if and only if

$$\exists z \in Z \; \exists p, q \in \mathit{Occ}(z, w) \; \mathit{TestSquare}(p, q) = \text{True}.$$

We know that the sets Z and $\mathit{Occ}(z, w)$ are of constant size. Now the required algorithm is a direct implementation of Observation 1 and of Observation 2, using a constant number of executions of Algorithms NaiveSearch and TestSquare (the latter implements $\mathit{TestSquare}(p, q)$). Since each of them works in linear time and constant space, this achieves the answer.

Notes

The above method easily extends to test if a word of length $n = 2^k$ that has no square factor of length smaller than 2^3 is square free. This yields an algorithm running in time $O(n \log n)$ and in constant space. The sketch is as follows. For each $m = 3, 4, \ldots, k$ in this order, the algorithm checks if overlapping segments of length 2^m are square free assuming that they are almost square free. The segments that overlap are chosen by intervals of length 2^{m-1}. As soon as a square is found the algorithm stops and reports its occurrence. Since for a given m the total length of segments is $O(n)$ this leads to an overall $O(n \log n)$ running time.

The presented algorithm is adapted from a method by Main and Lorentz [180].

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

79 Binary Words with Few Squares

The goal of the problem is to exhibit binary words containing the fewest number of (distinct) square factors.

A square is a word whose exponent is even; it is of the form $u^2 = uu$ for a non-empty word u. The longest words on the binary alphabet $\{0,1\}$ containing no square as factor are 010 and 101. But there are square-free infinite words on three-letter alphabets. One of them, on the alphabet $\{a,b,c\}$, is obtained by iterating the morphism f defined by

$$\begin{cases} f(a) = abc \\ f(b) = ac \\ f(c) = b, \end{cases}$$

which gives the infinite square-free word

$$\mathbf{f} = f^\infty(a) = abcacbabcbacabcacbacabcb\cdots$$

despite the fact that f does not preserve square-freeness of words, since $f(aba) = abcacabc$ that contains the square $(ca)^2$.

A cube is a word whose exponent is a multiple of 3.

Question. Show that no infinite binary word contains less than 3 squares. Show that no infinite binary word that contains only 3 squares avoids cubes, that is, is cube free.

Let g be the morphism from $\{a,b,c\}^*$ to $\{0,1\}^*$ defined by

$$\begin{cases} g(a) = 01001110001101 \\ g(b) = 0011 \\ g(c) = 000111. \end{cases}$$

Note that $g(ab)$ contains the three squares, 0^2, 1^2 and 10^2, as well as the two cubes 0^3 and 1^3.

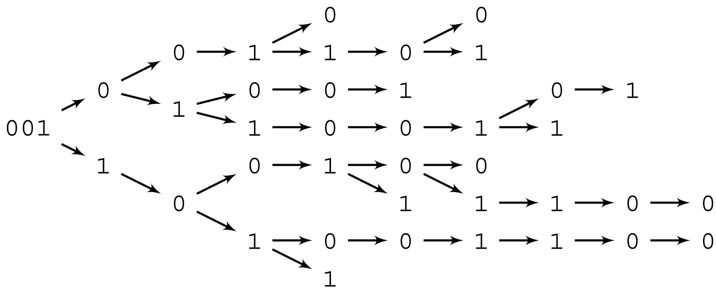
Question. Show there are only three squares and two cubes occurring in $\mathbf{g} = g(f^\infty(a))$.

[**Hint:** Consider distances between consecutive occurrences of 000.]

Solution

Checking the first assertion is a mere verification on the trie of binary words. Similarly, a word containing exactly three squares and no cube has maximal length 12, which can be checked with the next trie.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press



To prove the property of **g** we consider occurrences of `000` in it. In fact, distances between two consecutive occurrences are in $\{7, 11, 13, 17\}$:

$$\begin{array}{lll}
g(\texttt{ac}) & = \texttt{0100111}\underline{\texttt{000}}\texttt{1101}\ \underline{\texttt{000}}\texttt{111} & 7\\
g(\texttt{abc}) & = \texttt{0100111}\underline{\texttt{000}}\texttt{1101}\ \texttt{0011}\ \underline{\texttt{000}}\texttt{111} & 11\\
g(\texttt{ca}) & = \underline{\texttt{000}}\texttt{111}\ \texttt{0100111}\underline{\texttt{000}}\texttt{1101} & 13\\
g(\texttt{cba}) & = \underline{\texttt{000}}\texttt{111}\ \texttt{0011}\ \texttt{0100111}\underline{\texttt{000}}\texttt{1101} & 17.
\end{array}$$

Factors of **g** containing few occurrences of `000` have a bounded length; then it can be checked directly they do not have more squares than expected. We show it holds for other factors by contradiction.

Assume **g** contains a (large enough) square w^2 with an even number of occurrences of `000`. Let us consider the two consecutive occurrences on each side of the centre of the square and consider their distance is 7. This implies the centre of the square is in the occurrence of `1101` inside $g(\texttt{ac})$. Since the set $\{g(\texttt{a}), g(\texttt{b}), g(\texttt{c})\}$ is a prefix code, possibly taking a conjugate of the square yields that it is of the form $g(\texttt{c}v\texttt{ac}v\texttt{a})$ for some word $v \in \{\texttt{a},\texttt{b},\texttt{c}\}^*$. This is a contradiction since $f^\infty(\texttt{a})$ is square free.

Cases in which the distance between consecutive occurrences of `000` is 11, 13 or 17 are dealt with similarly.

Assume now w^2 contains an odd number of occurrences of `000`. Then w is of the form $\texttt{0}y\texttt{00}$ or symmetrically $\texttt{00}y\texttt{0}$ for a binary word y. Taking a conjugate as above produces a square in $f^\infty(\texttt{a})$, a contradiction.

Notes

The square-free word **f** is given with a different construction and a proof in Problem 80 after a translation with the alphabetic morphism α defined by $\alpha(\texttt{1}) = \texttt{c}$, $\alpha(\texttt{2}) = \texttt{b}$ and $\alpha(\texttt{3}) = \texttt{a}$.

The existence of an infinite binary word with only three squares and two cubes was initially proved by Fraenkel and Simpson [117]. Simpler proofs are by Rampersad et al. [205] and by Badkobeh [18] (see related questions in [19]). The present proof with the morphism g is from [18].

80 Building Long Square-Free Words

A word is square free if it does not contain any factor of the form uu for a non-empty word u. Generating long square-free words is meaningful only for alphabets of size at least three because the longest square-free words on a two-letter alphabet like {a, b} are aba and bab.

The goal of the problem is to design an algorithm generating long square-free words in an almost real-time way. Algorithm SQUAREFREEWORD does it using the function bin-parity(n) that denotes the parity (0 if even, 1 if odd) of the number of 1's in the binary representation of the natural number n. The delay between computing two outputs is proportional to the evaluation of that function.

```
SQUAREFREEWORD
1  prev ← 0
2  for n ← 1 to ∞ do
3      ▷ prev = max{i : i < n and bin-parity(i) = 0}
4      if bin-parity(n) = 0 then
5          output (n − prev)
6          prev ← n
```

The generated word α starts with: 3 2 1 3 1 2 3 2 1 2 3 1 $\cdots$.

Question. Show Algorithm SQUAREFREEWORD constructs arbitrarily long square-free words over the ternary alphabet {1, 2, 3}.

[**Hint:** The condition at line 4 holds only when n is the position of an occurrence of a in the Thue–Morse word **t**.]

Solution

The question is related to the overlap-freeness of the Thue–Morse word **t** (it contains no factor of the form $cucuc$ for a letter c and a word u). Running Algorithm SQUAREFREEWORD up to $n = 18$ gives the output 321312321. Assigning letter a to position n if the condition at line 4 holds and letter b if not, we get the table below, where the third row gives the output $n - prev(n)$ if the condition holds, associated with the current value of n.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a	b	b	a	b	a	a	b	b	a	a	b	a	b	b	a	b	a	a
3			2		1	3			1	2		3			2		1	

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

The algorithm exploits the following definition of **t**: $\mathbf{t}[n] = \mathtt{a}$ if and only if bin-parity$(n) = 0$. This equality is easily derived from other definitions of **t** in Chapter 1.

Word α. The square-freeness of the word α computed by Algorithm SQUAREFREEWORD relies on the fact that **t** is overlap free.

Let τ be the morphism from $\{1,2,3\}^*$ to $\{\mathtt{a},\mathtt{b}\}^*$ defined by $\tau(1) = \mathtt{a}$, $\tau(2) = \mathtt{ab}$ and $\tau(3) = \mathtt{abb}$. Note that **t** factorises uniquely on the suffix code $\{\mathtt{a},\mathtt{ab},\mathtt{abb}\}$. Algorithm SQUAREFREEWORD outputs i when the factor $\tau(i)\mathtt{a}$ is virtually detected in **t**.

Assume by contradiction that the output word contains a (non-empty) square factor uu. Then $\tau(uu)$ appears in **t**. But since both $u = \mathtt{a}v$ for a word v and the occurrence of $\tau(uu)$ is immediately followed by letter a, **t** contains the overlap $\mathtt{a}v\mathtt{a}v\mathtt{a}$, a contradiction. Therefore the output word α is square free.

Note that $a = h^\infty(3)$, where the morphism h, analogue to f in Problem 79, is defined by: $h(3) = 321$, $h(2) = 31$ and $h(1) = 2$.

Notes

A proof of the Thue–Morse word overlap-freeness may be found in [175, chapter 2]. The correctness of SQUAREFREEWORD also follows combinatorial proofs from the same chapter.

We give three alternative constructions of infinite square-free words β, γ, δ, omitting technical proofs:

- $\beta[i] = \mathtt{c}$ if $\mathbf{t}[i] = \mathbf{t}[i+1]$ and $\beta[i] = \mathbf{t}[i]$ otherwise.
- $\gamma[i] = \mathtt{c}$ if $\mathbf{t}[i-1] = \mathbf{t}[i]$ and $\gamma[i] = \mathbf{t}[i]$ otherwise.
- $\delta[0] = 0$ and, for $n > 0$, $\delta[n]$ is
  $\min\{k \geq 0 : k \neq \delta[\lfloor n/2 \rfloor]$ and $\delta[0\,..\,n-1] \cdot k$ is square free$\}$.
  The word δ can be computed using the following formula:
  if $h(n) = 1$ **then** $\delta[n] = 0$
  else if bin-parity$(n) = 1$ **then** $\delta[n] = 1$
  else $\delta[n] = 2$,
  where, for $n > 0$, $h(n)$ is the parity of the length of the block of 0's at the end of the binary representation of n.

Despite different constructions the four defined words α, β, γ and δ are essentially almost the same (after renaming letters and in some cases removing the first letter). The number of square-free words of length n over a ternary alphabet, $sqf(n)$, is known to grow exponentially with n, as proved by Brandenburg [42] and later tuned by several authors. The first values of $sqf(n)$ are listed in the table:

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

n	1	2	3	4	5	6	7	8	9	10	11	12	13
$sqf(n)$	3	6	12	18	30	42	60	78	108	144	204	264	342
n	14	15	16	17	18	19	20	21	22	23	24		
$sqf(n)$	456	618	798	1044	1392	1830	2388	3180	4146	5418	7032		

In contrast, the number of overlap-free binary words of length n over a binary alphabet only grows polynomially, as shown by Restivo and Salemi [207] (see Problem 75).

81 Testing Morphism Square-Freeness

Square-free morphisms are word morphisms that preserve word square-freeness. These morphisms provide a useful method to generate by iteration square free words. The problem aim is to give an effective characterisation of square-free morphisms, which yields a linear-time test according to the morphism length on a fixed-size alphabet.

A square free morphism h satisfies: $h(x)$ is square free when x is. We also say h is k-square free if the condition is satisfied for $|x| \leq k$. In general k-square-freeness does not imply square-freeness. For example, h_1 is a shortest square-free morphism from $\{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$ to itself, but h_2 from $\{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$ to $\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{e}\}^*$ is not square free although it is 4-square free.

$$\begin{cases} h_1(\mathtt{a}) = \mathtt{abcab} \\ h_1(\mathtt{b}) = \mathtt{acabcb} \\ h_1(\mathtt{c}) = \mathtt{acbcacb} \end{cases} \qquad \begin{cases} h_2(\mathtt{a}) = \mathtt{deabcbda} \\ h_2(\mathtt{b}) = \mathtt{b} \\ h_2(\mathtt{c}) = \mathtt{c}. \end{cases}$$

The following characterisation is based on the notion of pre-square. Let z be a factor of $h(a)$, $a \in A$. Its occurrence at position i is called a pre-square if there is a word y for which ay (resp. ya) is square free and z^2 occurs in $h(ay)$ at position i (resp. in $h(ya)$ with centre i). It is clear that if some $h(a)$ has a pre-square h is not a square-free morphism. The converse holds up to an additional condition.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

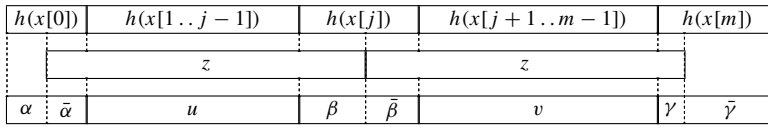
Question. Show that a morphism h is square free if and only if it is 3-square free and no $h(a)$, $a \in A$, contains a pre-square factor.

[**Hint:** Discuss cases using the picture below that displays a square z^2 in $h(x)$, where $x = x[0\,..\,m]$.]

Question. Show that for uniform morphisms 3-square-freeness implies square-freeness, and for morphisms on 3-letter alphabets 5-square-freeness implies square-freeness.

Solution

Pre-square condition. To prove the statement in the first question we only have to show that a non-square-free morphism breaks one of the two conditions because the converse is obvious.



Let $x = x[0\,..\,m]$, for which $h(x)$ contains a square z^2. Possibly chopping letters at the ends of x, we may assume the occurrence of z^2 starts in $h(x[0])$ and ends in $h(x[m])$ (see picture).

Note that if $h(a)$ is a prefix or a suffix of $h(b)$, $a \neq b$, the morphism is not even 2-square free. Therefore we can assume $\{h(a) : a \in A\}$ is a (uniquely decipherable) prefix and suffix code.

Let α, $\bar{\alpha}$, β, $\bar{\beta}$, γ and $\bar{\gamma}$ be as displayed in the picture.

First, if $\bar{\alpha} = \bar{\beta}$, by prefix codicity $x[1\,..\,j-1] = x[j+1\,..\,m-1]$, and then $\beta = \gamma$. Since x is square free, $x[0] \neq x[j]$ and $x[j] \neq x[m]$. Thus $x[0]x[j]x[m]$ is square free but $h(x[0]x[j]x[m])$ contains $(\bar{\alpha}\beta)^2$: h is not 3-square free.

Assume w.l.o.g. in the next cases that $\bar{\alpha}\delta = \bar{\beta}$ for $\delta \neq \varepsilon$.

Second, if $x[1] \neq x[j]$, let i be the smallest index for which δ is a prefix of $h(x[1\,..\,i])$. Then $x[j]x[1\,..\,i]$ is square free but $h(x[j]x[1\,..\,i])$ contains δ^2: there is a pre-square in $h(x[j])$.

Third, if $x[1] = x[j]$, $h(x[j])$ follows $\bar{\alpha}$ in z then $h(x[j\,..\,m])$ starts with $(\beta\bar{\alpha})^2$: there is a pre-square in $h(x[j])$.

Considering symmetric cases as above ends the proof.

Uniform morphism. When the morphism is uniform, it can just be remarked that the pre-square condition of the first statement is equivalent to the 2-square-free property, which is implied by the 3-square-free condition.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

From a 3-letter alphabet. Let A be a 3-letter alphabet. Assume there is a pre-square in $h(a)$, $a \in A$, and that y extends the pre-square into a square in $h(ay)$. Possibly chopping a suffix of y, the letter a can only reappear as its last letter. The word ay being square free on 3 letters, ya^{-1} is square free on 2 letters, which implies its length is at most 3. Therefore $|ay| \leq 5$, which resorts to the 5-square-free condition on h.

Example h_2 shows the bound 5 is optimal.

Notes

A square-free morphism h provides an interesting tool to generate an infinite square-free words: if $h(a)$ is of the form ay for some letter a and a non-empty word y, iterating h from a gives the square-free infinite word $h^\infty(a)$. Note, however, the morphism f of Problem 79 is not square-free but $f^\infty(\mathtt{a})$ is. More is presented by Berstel and Reutenauer in Lothaire [175, chapter 2]; see also [35].

The full proof of the first statement appears in [65] together with some consequences of the result.

82 Number of Square Factors in Labelled Trees

It is known that the number of (distinct) square factors in a given word is linear (see Problem 72). Unfortunately, the property does not hold for edge-labelled trees.

The problem shows a surprising lower bound based on relatively simple example trees.

Question. Prove that an edge-labelled binary tree of size n can contain $\Omega(n^{4/3})$ (distinct) square factors.

[**Hint:** Consider comb trees.]

Solution

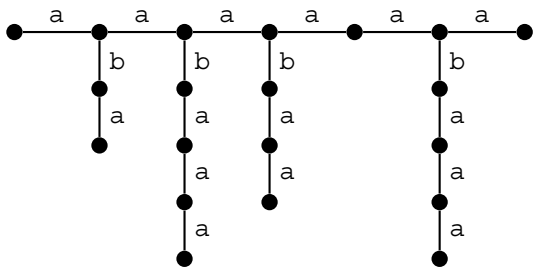
Denote by $\mathsf{sq}(T)$ the number of square factors along the branches of an edge-labelled tree T. To prove the result we consider a special family of very simple

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

trees, called combs, that achieves the largest possible number of squares in asymptotic terms.

A *comb* is a labelled tree that consists of a path called the *spine* with at most one *branch* attached to each node of the spine. All spine-edges are labelled by the letter a. Each branch is a path whose label starts with letter b followed by a number of a's. In the graphical example below the comb contains 14 square factors:

- $\mathtt{a}^2$, $(\mathtt{aa})^2$, $(\mathtt{aaa})^2$,
- all cyclic rotations of: $(\mathtt{ab})^2$, $(\mathtt{aab})^2$ and $(\mathtt{aaab})^2$,
- and the squares $(\mathtt{abaaa})^2$ and $(\mathtt{aaaba})^2$.



We show that there exists a family T_m of special combs that satisfy $\mathsf{sq}(T_m) = \Omega(|T_m|^{4/3})$. From this result one easily obtains $\mathsf{sq}(T) = \Omega(n^{4/3})$ for a tree T of size n.

We consider the integer $m = k^2$ and define the set

$$Z_m = \{1, \ldots, k\} \cup \{i.k : 1 \leq i \leq k\}.$$

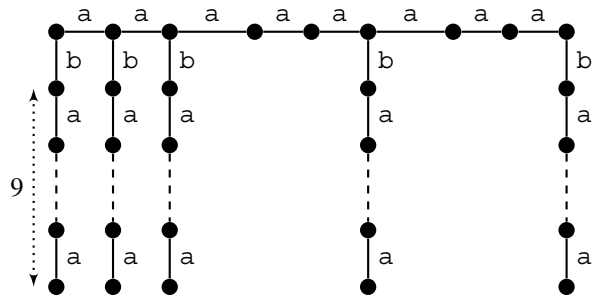
For example, $Z_9 = \{1, 2, 3, 6, 9\}$.

Observation. For each integer d, $0 < d < m$, there exist $i, j \in Z_m$ for which $i - j = d$.

Proof Each number d, $0 < d < m$, has a unique representation in the form $p\sqrt{m} - q$ where $0 < p, q \leq \sqrt{m}$. Choosing $i = p\sqrt{m}$ and $j = q$ gives the conclusion. ∎

The special comb T_m is then defined as follows: T_m consists of a spine of length $m - 1$ with vertices numbered from 1 to m and labelled by $\mathtt{a}^{m-1}$ and of branches labelled by $\mathtt{ba}^m$ attached to each vertex $j \in Z_m$ of the spine. The picture displays the special comb T_9 associated with $Z_9 = \{1, 2, 3, 6, 9\}$, with its spine and its five branches.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press



Fact. Each tree T_m satisfies $\mathsf{sq}(T_m) = \Omega(|T_m|^{4/3})$.

Proof The above observation implies that for every d, $0 < d < m$, there are two nodes i, j of degree 3 on the spine with $i - j = d$. Thus, T_m contains all squares of the form $(\mathtt{a}^i\mathtt{b}\mathtt{a}^{d-i})^2$ for $0 \leq i \leq d$.

Altogether this gives $\Omega(m^2)$ different squares. Since $m = k^2$, the size of T_m, its number of nodes, is $k(m+2) + (k-1)(k+m+1) = O(m\sqrt{m})$. Therefore, the number of squares in T_m is $\Omega(|T_m|^{4/3})$. ∎

Notes

The above result is optimal because the upper bound on the number of squares in labelled trees of size n is $O(n^{4/3})$. The combinatorial proof of this bound is much harder and can be found in [82].

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

83 Counting Squares in Combs in Linear Time

A comb is a labelled tree that consists of a path called the *spine* with at most one *branch* attached to each node of the spine. All spine-edges are labelled with the letter a. Each branch is a path whose label starts with the letter b followed by a number of a's.

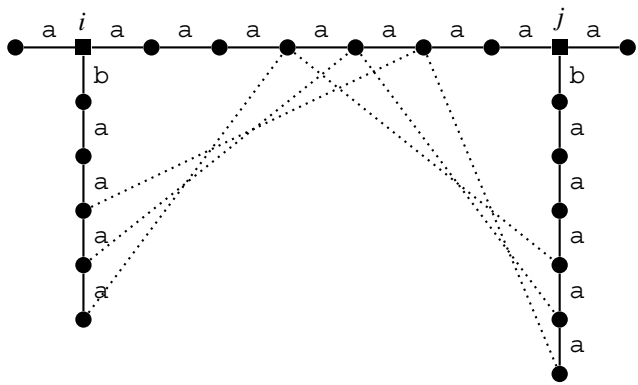
The number of (distinct) squares occurring on branches of a comb T can be superlinear (see Problem 82) but despite the lower bound it is possible to count them in linear time according to the tree size. This is the goal of the problem. This is done with a careful encoding of squares due to their global structure.

Question. Show how to compute in linear time the number of (distinct) square factors on branches of a binary comb.

Solution

We focus only on non-unary squares because it is clear that the number of unary squares (of period 1) in any labelled tree can be computed in linear time.

To get the expected running time a special encoding of all squares is required. It is based on admissible pairs of nodes of the spine. Such a pair (i, j) is called *admissible* if $d \leq p + q$, where d is the distance between i and j ($|(j - i)|$) and p, q are the numbers of occurrences of a's on the branches outgoing from i and from j respectively.



An *essential part* of a comb corresponds to an admissible pair of nodes (i, j) on the spine, the edges between them, and the two outgoing branches whose labels start with the letter b. All squares in each such essential part can be seen as a *package* of squares (set of conjugates of a single factor) represented by an interval.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

The above picture shows an admissible pair (i, j) with $d = 7$, $p = 4$ and $q = 5$. The non-unary squares generated by the essential part of the tree corresponding to this pair are $(\mathtt{a}^2\mathtt{ba}^5)^2$, $(\mathtt{a}^3\mathtt{ba}^4)^2$ and $(\mathtt{a}^4\mathtt{ba}^3)^2$ illustrated by the dotted lines.

More generally the set of squares corresponding to a pair (i, j) is of the form $\{\mathtt{a}^k\mathtt{ba}^k\mathtt{a}^{d-k}\mathtt{ba}^{d-k} : k \in [i', j']\}$, where $[i', j'] \subseteq [i, j]$. The set can then be represented by the pair $(d, [i', j'])$. In the above example $(7, [2, 4])$ represents the set of squares $\{(\mathtt{a}^2\mathtt{ba}^5)^2, (\mathtt{a}^3\mathtt{ba}^4)^2, (\mathtt{a}^4\mathtt{ba}^3)^2\}$.

Fact. The number of admissible pairs in a comb T is linear according to the comb size.

Proof Observe that if (i, j) is an admissible pair with distance d and p, q the numbers of a's on the branches outgoing from i and j, then $d \leq 2\max\{p, q\}$. Hence for a given node on the spine it is enough to consider nodes on the spine at distance at most k to the left and to the right from this node, where k is the number of a's on the branch outgoing from this node.

The total number of considered nodes is thus bounded by the total length of outgoing branches, which is $O(|T|)$. ∎

Fact. The number of (distinct) square factors in a comb T can be computed in linear time.

Proof To achieve linear running time, we group admissible pairs into sets associated with the same distance d between the nodes of pair. For each pair (i, j) the set of squares generated by this pair corresponds to an interval. These intervals (for distinct pairs) are not necessarily disjoint; however, we can make a union of all intervals in linear time. The resulting set is again a union of intervals and its total size can be easily computed. The sets of squares corresponding to distinct groups are disjoint. We sum the numbers for each group and get the final result. This completes the proof. ∎

Despite the fact that we can have super-linearly many distinct squares, in addition to unary squares, all of the other squares can be reported as a union of linearly many disjoint sets of the form

$$\{\mathtt{a}^k\mathtt{ba}^k\mathtt{a}^{d-k}\mathtt{ba}^{d-k} : k \in [i', j']\}.$$

Notes

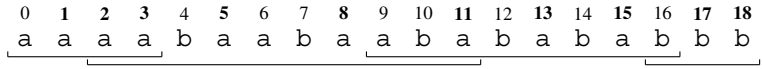
The present algorithm is adapted from the algorithm by Kociumaka et al. [164]. So far it is not known if squares can be counted in linear time for general trees.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

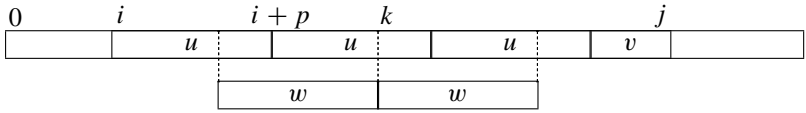
84 Cubic Runs

Cubic runs constitute a particular case of runs for which bounds are easier to evaluate. As runs they encompass different types of periodic factors in a word but to a lesser extent.

A ***cubic run*** in a word x is a maximal periodicity in x whose length is at least three times its period. More accurately, it is an interval $[i\,..\,j]$ of positions on x whose associated factor $u = x[i\,..\,j]$ satisfies $|u| \geq 3per(u)$ and that is not extensible to the left nor to the right with the same period. Cubic runs in aaaabaabaababababbb are underlined in the picture.



We consider an ordering $<$ on the alphabet and define special positions of a run $[i\,..\,j]$ in x as follows. Let p be the period of $x[i\,..\,j]$ and let w be the alphabetically smallest conjugate (rotation) of $x[i\,..\,i+p-1]$. Then k is a special position of the run if $x[i\,..\,j]$ contains a square ww centred at k. Special positions are shown in boldface in the above picture.



Question. Show that a cubic run has at least one special position and that two different cubic runs share no special position.

[**Hint:** Use the fact that the smallest conjugate of a primitive word, a Lyndon word, is border free.]

Question. Show both that the number of cubic runs in a word of length n is smaller than $n/2$ and that, for infinitely many n, it is at least $0.4n$.

[**Hint:** Consider the inverse alphabet ordering $<^{-1}$ and count cubic runs in words $x_m = (u^2\mathtt{a}^3v^2\mathtt{b}^3w^2\mathtt{c}^3)^m$, where $u = \mathtt{a}^3\mathtt{b}^3$, $v = \mathtt{b}^3\mathtt{c}^3$ and $w = \mathtt{c}^3\mathtt{a}^3$.]

Solution

At least one special position in each cubic run. Let $[i\,..\,j]$ be a cubic run, $p = per(x[i\,..\,j])$ and w the smallest conjugate of $x[i\,..\,i+p-1]$.

If $p = 1$ it is clear that all positions in the run except the first position are special, which shows there are at least two special positions for this type of cubic run.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

If $p > 1$ the square $x[i \,..\, i + 2p - 1]$ contains at least one occurrence of w, which is followed immediately by another occurrence of w in the run. Therefore there is at least 1 special position in this type of cubic run.

Different cubic runs share no special position. Assume some position k is the centre of occurrences of ww and of $w'w'$ associated with two different cubic runs. Due to the maximality condition, the runs, being different, have different periods. If for example w' is the shorter it is then a border of w. But since w is primitive (due to the definition of p) and a smallest conjugate, it is a Lyndon word, which is known to be border free, a contradiction. Thus two different cubic runs cannot share a special position.

Less than $n/2$ cubic runs. We have already seen that cubic runs with period 1 have at least two special positions. For the other cubic runs first note the associated prefix period contains at least two different letters. Then a second special position can be found using the inverse alphabet ordering (or the greatest conjugate of the prefix period) and, as above, this position is not shared by any other run.

Since position 0 on x cannot be special, the total number of special positions in a word of length n is less than n, which implies less than $n/2$ cubic runs.

Lower bound. Observe that for any $m > 0$, the word x_m contains at least $18m - 1$ cubic runs:

$$x_m = (\mathtt{a}^3\mathtt{b}^3\mathtt{a}^3\mathtt{b}^3\ \ \mathtt{a}^3\ \ \mathtt{b}^3\mathtt{c}^3\mathtt{b}^3\mathtt{c}^3\ \ \mathtt{b}^3\ \ \mathtt{c}^3\mathtt{a}^3\mathtt{c}^3\mathtt{a}^3\ \ \mathtt{c}^3)^m.$$

Indeed, there are $15m$ cubic runs of period 1 whose associated factors are $\mathtt{a}^3$, $\mathtt{b}^3$ or $\mathtt{c}^3$; $2m$ cubic runs of period 6 with factors $(\mathtt{a}^3\mathtt{b}^3)^3$ and $(\mathtt{b}^3\mathtt{c}^3)^3$; and $m - 1$ cubic runs of the form $(\mathtt{c}^3\mathtt{a}^3)^3$.

Note that for $m > 2$ the whole word x_m forms an additional cubic run. Hence, in this case the word x_m has length $45m$ and contains at least $18m$ cubic runs. Thus, for $m > 2$, the number of cubic runs in x_m is not less than $0.4|x_m| = 0.4n$.

Notes

Slightly improved lower and upper bounds on the number of cubic runs in a word are shown in [85, 86].

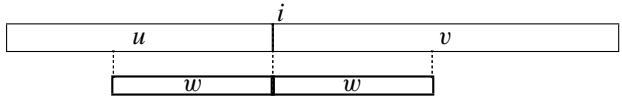
Using an argument similar to the one above, Harju and Kärki in [137] introduced the notion of frame, square whose root is border-free, and derive upper and lower bounds on the number of frames in binary words, bounds that are close to the above bounds.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

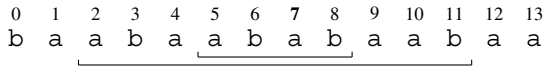
85 Short Square and Local Period

The notion of local periods in words provides a more accurate structure of its repetitiveness than its global period. The notion is central to that of critical positions (see Problem 41) and their applications.

Finding the local period at a given position i on a word x is the question of the problem. The local period $\ell per(i)$ is the period of a shortest non-empty square ww centred at position i and possibly overflowing x to the left or to the right (or both).



Question. Show how to compute all non-empty squares centred at a given position i on x in time $O(|x|)$.



For $x = \texttt{baabaababaabaa}$, squares centred at 7 are $\texttt{(abaab)}^2$, $\texttt{(ab)}^2$ and the empty square. There is no non-empty square centred at 6 or 9, for example.

Question. If there exists a shortest non-empty square of period p centred at position i on x, show how to find it in time $O(p)$.

[**Hint:** Double the length of the search area.]

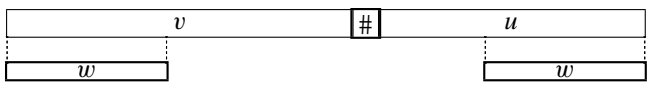
Here are a few local periods for the above example word: $\ell per(7) = 2$ period of $\texttt{(ab)}^2$, $\ell per(1) = 3$ period of $\texttt{(aab)}^2$, $\ell per(6) = 8$ period of $\texttt{(babaabaa)}^2$ and $\ell per(9) = 5$ period of $\texttt{(aabab)}^2$.

Question. Design an algorithm to compute the local period p at position i on x in time $O(p)$.

[**Hint:** Mind the situation where there is no square centred at i.]

Solution

Squares Let $u = x[0\,..\,i-1]$, $v = x[i\,..\,|x|-1]$ and # a letter not in x.



https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

The borders w of $v\#u$ produce the squares centred at i on x. If the mere concatenation vu is used instead of $v\#u$, only its borders no longer than $\min\{|u|, |v|\}$ produce the sought squares.

Thus squares can be found with the border table of $v\#u$ whose computation take linear time (see Problem 19) like the whole process.

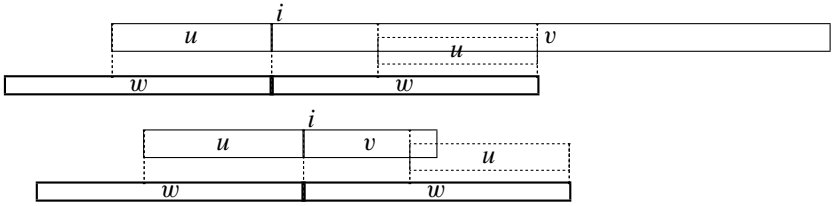
Shortest square. Given a length ℓ, $0 < \ell \leq \min\{|u|, |v|\}$, a square of period p no more than ℓ can be found as above considering borders of $x[i\,..\,i+\ell-1]\#x[i-\ell\,..\,i-1]$ instead of $v\#u$. Running ℓ from 1 to at most $4p$ allows the detection of a square of period p.

The whole process takes time

$$O(\Sigma\{\ell : \ell = 1, 2, 4, \ldots, 2^e, \text{ with } p \leq 2^e < 2p\}) = O(p).$$

Local period. If there is a non-empty square centred at i, the local period at i is the period of the shortest such square.

If there is no non-empty square centred at i, the square ww whose period is the local period at i may overflow to the left or to the right (or both). The picture below shows overflows to the left.



To detect an overflow to the left, v is searched online for u with, for example, Algorithm KMP whose output is to be adapted to cope with the situation displayed in the bottom picture. This produces a potential local period p_1. Checking symmetrically for an overflow to the right by searching u^{R} for v^{R} gives another potential local period p_2. Eventually the sought local period is the minimum of the two.

The whole computation then runs in time $O(p)$, which answers the question.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

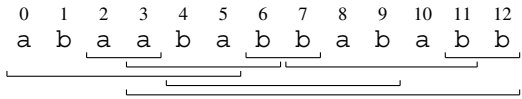
Notes

The computation of a border table is treated in Problem 19 and works on general alphabets, similarly to Algorithm KMP described in Problem 26. See also textbooks on Stringology [74, 96, 98, 134, 228] and other textbooks on algorithms.

On fixed-size alphabets the computation of all local periods of a word can be done in linear time [106] using a factorisation of the word similar to its Lempel–Ziv factorisation; see Chapter 6.

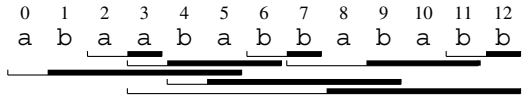
86 The Number of Runs

A ***run*** is a maximal periodicity occurring in a word. Formally, a run in x is an interval $[i\,..\,j]$ of positions on x whose associated factor $x[i\,..\,j]$ is periodic (i.e., its smallest period p satisfies $2p \leq |x[i\,..\,j]| = (j - i + 1)$) and the periodicity does not extend to the right nor to the left (i.e., $x[i - 1\,..\,j]$ and $x[i\,..\,j + 1]$ have larger periods when defined). The eight runs in abaababbababb are underlined in the picture.



We consider an ordering $<$ on a word alphabet and the corresponding lexicographic ordering denoted $<$ as well. We also consider the lexicographic ordering $\widetilde{<}$, called the reverse ordering, inferred by the inverse alphabet ordering $<^{-1}$. Each run $[i\,..\,j]$ is associated with its greatest suffix according to one of the two orderings as follows. Let $p = per(x[i\,..\,j])$. If $j + 1 < n$ and $x[j + 1] > x[j - p + 1]$ we assign to the run the position k for which $x[k\,..\,j]$ is the greatest proper suffix of $x[i\,..\,j]$ according to $<$. Otherwise, k is the starting position of the greatest proper suffix of $x[i\,..\,j]$ according to $\widetilde{<}$. The position k assigned this way to a run is called its **special position**. These positions are intimately linked to Lyndon words (defined in Chapter 1), subject of the first question. The thick lines below show the greatest suffixes associated with runs in abaababbababb.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press



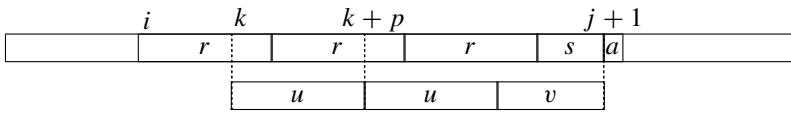
Question. Show that, if the special position k of a run of period p is defined according to $\widetilde{\preceq}$ (resp. $\prec$), $x[k\,..\,k+p-1]$ is the longest Lyndon factor of x starting at position k according to $\prec$ (resp. $\widetilde{\preceq}$).

[**Hint:** The special position k of a run $[i\,..\,j]$ of period p satisfies $k \leq i+p$; see Problem 40.]

Question. Show two distinct runs have no special position in common and deduce that the number of runs in a word is smaller than its length.

Solution

Special position. Let $[i\,..\,j]$ be a run of period p with special position k. To answer the first question, note that $x[k\,..\,k+p-1]$ is a Lyndon word because it is smaller than all its proper suffixes according to $\prec$. Consider a longer factor $x[k\,..\,j']$ for $k+p \leq j' \leq j$. It has period p which is smaller than its length; equivalently it is not border free, which shows it is not a Lyndon word for any of the two orderings.



There is nothing else to prove if $j+1 = |x|$. Assume then that $j' > j$ and $a = x[j+1]$. The picture displays the greatest suffix of $x[i\,..\,j]$ according to $\widetilde{\preceq}$, that is, $x[k\,..\,j] = u^e v$ of period $|u|$ in which v is a proper prefix of u. Since $x[j+1] < x[j-p+1]$, we get $x[k+p\,..\,j+1] < x[k\,..\,j-p+1]$, which leads to $x[k+p\,..\,j'] < x[k\,..\,j']$ and shows that $x[k\,..\,j']$ is not a Lyndon word according to $\prec$.

Therefore, $x[k\,..\,k+p-1]$ is the longest Lyndon factor of x starting at position k. Note the roles of the two orderings are perfectly symmetric.

Number of runs. Let us answer the second question by contradiction, assuming two runs share the same special position k. The position cannot be defined with the same ordering for the two runs due to the above result. Being defined by the two different orderings, the only possibility is that only one run has period 1. But then $x[k-1] = x[k]$, which is impossible for the special position k on a non-unary run.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

Since position 0 cannot be a special position, at most $n-1$ positions can be special positions of runs in a word of length n. The previous result then implies that there are less that n runs, as stated.

Notes

The concept of a run, also called a maximal periodicity or the maximal occurrence of a repetition, coined by Iliopoulos et al. [149] when analysing repetitions in Fibonacci words, has been introduced to represent in a succinct manner all occurrences of repetitions in a word. It is known that there are only $O(n)$ of them in a word of length n from Kolpakov and Kucherov [167], who proved it in a non-constructive way.

The first explicit bound was later on provided by Rytter [214]. Several improvements on the upper bound can be found in [77, 80, 102, 203]. Kolpakov and Kucherov conjectured that this number is in fact smaller than n, which has been proved by Bannai et al. [26]. The present proof, very similar to their proof, appears in [91]. Fischer et al. [116] gave a tighter upper bound of $22n/23$ on the number of runs.

With the above notations, remark that if $k + 2p \leq j$, $k + p$ can also be considered a special position with the same property. In particular, a run whose associated word starts with a cube has at least two special positions. This gives an upper bound of $n/2$ for the maximum number of cubic runs in a word of length n (see Problem 84 and more in [25] and [86]).

87 Computing Runs on Sorted Alphabet

The aim of the problem is to show that all runs (maximal periodicities) in a word drawn from a linearly sortable alphabet can be computed in linear time.

The solution is based on the results of Problem 86, where it is shown that a run possesses a special position from which starts a longest Lyndon factor of the whole word. Tracking the longest Lyndon factors has to be done according to the alphabet ordering $<$ but also to its inverse $<^{-1}$. When a longest Lyndon factor is located, simple extensions from two positions to the right and to the left (like in Problem 74) can confirm the starting position of the Lyndon factor is a special position of a run whose period is the length of the Lyndon factor.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

To do so we first define the ***Lyndon table*** *Lyn* of a (non-empty) word x. For a position i on x, $i = 0, \ldots, |x| - 1$, $Lyn[i]$ is the length of the longest Lyndon factor starting at i:

$$Lyn[i] = \max\{\ell : x[i \,..\, i + \ell - 1] \text{ is a Lyndon word}\}.$$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$x[i]$	a	b	b	a	b	a	b	a	a	b	a	b	b	a	b	a
$Lyn[i]$	3	1	1	2	1	2	1	8	5	1	3	1	1	2	1	1

Question. Show that Algorithm LONGESTLYNDON correctly computes the table *Lyn*.

LONGESTLYNDON(x non-empty word of length n)

```
1  for i ← n − 1 downto 0 do
2      (Lyn[i], j) ← (1, i + 1)
3      while j < n and x[i .. j − 1] < x[j .. j + Lyn[j] − 1] do
4          (Lyn[i], j) ← (Lyn[i] + Lyn[j], j + Lyn[j])
5  return Lyn
```

Question. Extend Algorithm LONGESTLYNDON to compute all runs occurring in a word.

[**Hint:** Use the longest common extensions like in Problem 74.]

Question. What is the total running time of the algorithm if a comparison of two factors is done with the help of the ranks of their associated suffixes in the alphabetic order and if the longest common extension techniques are used?

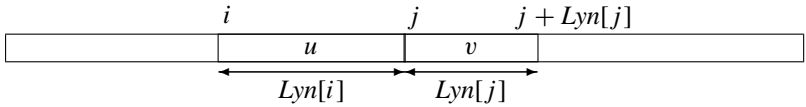
Solution

Proofs rely on the following well-known properties of Lyndon words that may be found in [175]. First, if u and v are Lyndon words and $u < v$ then uv is also a Lyndon word (and $u < uv < v$). Second, each non-empty word factorises uniquely as $u_0 u_1 u_2 \cdots$, where each u_i is a Lyndon word and $u_0 \geq u_1 \geq u_2 \geq \cdots$. In addition, u_0 is the longest Lyndon prefix of the word. The factorisation can be computed using the *Lyn* table of the word to jump from a factor to the next one. But the table contains more information than the factorisation.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

The factorisation of the above example word abbababaababbaba is abb · ab · ab · aababbab · a, corresponding to the subsequence 3, 2, 2, 8, 1 of values in its Lyndon table.

Correctness of LongestLyndon. The invariant of the for loop, when processing position i, is: $Lyn[k]$ is computed for $k = i+1, \ldots, n-1$ and $u[i+1\,..\,j-1] \cdot u[j\,..\,j+Lyn[j]-1] \cdots$, where $j = i+1+Lyn[i+1]$, is the Lyndon factorisation of $x[i+1\,..\,n-1]$.



The current factor u starting at position i, initially $x[i]$, is compared to its next factor v. If $u < v$, u is replaced by uv and the comparison continues with the successor of uv. The while loop stops when the current factor u becomes no smaller than its next factor. It is clear when the loop stops that u is the longest Lyndon factor starting at i and then $Lyn[i] = |u|$. It is also clear that we get the Lyndon factorisation of $x[i\,..\,n-1]$, which achieves the proof of the invariant and of the correctness of LongestLyndon.

Computing runs. To compute all runs in the word x, we just check for each position i if it is the special position of a run whose word period is $x[i\,..\,i+Lyn[i]-1]$. This is done by computing lengths ℓ and r of longest common extensions (LCEs) of the period to the left and to the right and by checking if $\ell + r \geq Lyn[i]$. If the inequality holds a run is reported.

```
Runs(x non-empty word of length n)
 1  for i ← n − 1 downto 0 do
 2      (Lyn[i], j) ← (1, i + 1)
 3      while j < n and x[i .. j − 1] < x[j .. j + Lyn[j] − 1] do
 4          (Lyn[i], j) ← (Lyn[i] + Lyn[j], j + Lyn[j])
 5      ℓ ← |lcs(x[0 .. i − 1], x[0 .. i + Lyn[i] − 1])|
 6      r ← |lcp(x[i .. |x| − 1], x[i + Lyn[i] .. |x| − 1])|
 7      if ℓ + r ≥ Lyn[i] then
 8          output run [i − ℓ .. i + Lyn[i] + r − 1]
```

More precisely, ℓ is the length of the longest common suffix of $x[0\,..\,i-1]$ and $x[0\,..\,i+Lyn[i]-1]$, while r is the length of the longest common prefix of $x[i\,..\,|x|-1]$ and $x[i+Lyn[i]\,..\,|x|-1]$. They are set to null if $i = 0$ and if $i + Lyn[i] = n$ respectively.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

The above process has to be repeated for the ordering $\tilde{<}$ associated with the inverse alphabet ordering.

Running time of RUNS. First note that the number of word comparisons performed at line 3 by RUNS is less than $2|x|$. Indeed there is at most one negative comparison at each step. And there are less than $|x|$ positive comparisons because each reduces the number of factors of the Lyndon factorisation of x. Therefore, to get a linear-time algorithm we have to discuss how to compare words and to compute LCE.

Comparison of words at line 3 of algorithms can be realised using ranks of suffixes due to the next property.

Property. Let u be a Lyndon word and $v \cdot v_1 \cdot v_2 \dots v_m$ be the Lyndon factorisation of a word w. Then $u < v$ if and only if $uw < w$.

Proof Assume $u < v$. If $u \ll v$ then $uw \ll vv_1v_2 \dots v_m = w$. Otherwise u is a proper prefix v. Let $e > 0$ be the largest integer for which $v = u^e z$. Since v is a Lyndon word, z is not empty and we have $u^e < z$. Since u is not a prefix of z (by definition of e) nor z a prefix of u (because v is border free) we have $u \ll z$. This implies $u^{e+1} \ll u^e z = v$ and then $uw < w$.

Conversely, assume $v \leq u$. If $v \ll u$ we obviously have $w < uw$. It remains to consider the situation where v is a prefix of u. If it is a proper prefix, u writes vz for a non-empty word z. We have $v < z$ because u is a Lyndon word. The word z cannot be a prefix of $t = v_1v_2 \cdots v_m$ because v would not be the longest Lyndon prefix of w, a contradiction with a property of the factorisation. Thus, either $t \leq z$ or $z \ll t$. In the first case, if t is a prefix of z, $w = vt$ is a prefix of u and then of uw, that is, $w < uw$. In the second case, for some suffix z' of z and some factor v_k of t we have $z' \ll v_k$. The factorisation implies $v_k \leq v$. Therefore, the suffix z' of u is smaller than its prefix v, a contradiction with the fact that u is a Lyndon word. ∎

For each starting position i of a suffix of x, $i = 0, \dots, |x| - 1$, let Rank$[i]$ be the rank of the suffix $x[i \,..\, |x| - 1]$ in the increasing alphabetic list of all non-empty suffixes of x (ranks range from 0 to $|x| - 1$).

Due to the property, the inequality $x[i \,..\, j - 1] < x[j \,..\, j + Lyn[j] - 1]$ at line 3 of the two previous algorithms rewrites Rank$[i]$ < Rank$[j]$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$x[i]$	a	b	b	a	b	a	b	a	a	b	a	b	b	a	b	a
$Lyn[i]$	3	1	1	2	1	2	1	8	5	1	3	1	1	2	1	1
Rank$[i]$	7	15	12	4	11	3	9	1	5	13	6	14	10	2	8	0

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

Note the Lyndon factorisation can be recovered by following the longest decreasing sequence of ranks from the first rank. It is $(7, 4, 3, 1, 0)$ in the above example, corresponding to positions $(0, 3, 5, 7, 15)$ on x and its factorisation $\mathtt{abb} \cdot \mathtt{ab} \cdot \mathtt{ab} \cdot \mathtt{aababbab} \cdot \mathtt{a}$.

As for the running time, when the table Rank of suffix ranks is precomputed, the comparison of words at line 3 can be realised in constant time. It is known that the table of ranks, inverse of the sorted list of suffixes (Suffix array), can be computed in linear time under the hypothesis that the alphabet is linearly sortable.

Instructions at lines 5–6 can executed as LCE queries and as such computed in constant time after a linear-time preprocessing under the same hypothesis (see, e.g., [115]). Therefore the whole algorithm Runs works in linear time when the alphabet is linearly sortable.

Notes

Algorithm LongestLyndon can be slightly changed to compute the Lyndon forest of a word. The forest comprises the list of Lyndon trees corresponding to factors of the Lyndon factorisation of the word.

The Lyndon tree of a Lyndon word is associated recursively with the (right) standard factorisation of a Lyndon word w not reduced to a single letter: w can be written uv, where v is chosen either as the smallest proper non-empty suffix of w or as the longest proper Lyndon suffix of w, which yields the same suffix. The word u is then also a Lyndon word and $u < v$ (see [175]).

The structure of a Lyndon tree has been shown to be the same as that of the Cartesian tree of ranks of suffixes by Hohlweg and Reutenauer [142]. Algorithm LongestLyndon proceeds like a right-to-left construction of a Cartesian tree (https://en.wikipedia.org/wiki/Cartesian_tree).

The relation between Suffix arrays and Lyndon factorisations is examined by Mantaci et al. in [184]

Franek et al. ([119]) present several algorithms to compute the Lyndon table.

The reader can refer to the review by Fischer and Heun [115] concerning LCE queries. More advanced techniques to implement them over a general alphabet and to compute runs can be found in [83, 128] and references therein.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

88 Periodicity and Factor Complexity

The property stated in the problem provides a useful condition to detect the periodicity of infinite words.

An infinite word x (indices run through natural numbers) is said to be ultimately periodic or simply u-periodic if it can be written yz^∞ for some (finite) words y and z, $z \neq \varepsilon$.

Let $F_x(n)$ denote the number of (distinct) factors of length n occurring in the infinite word x. The function F_x is called the factor (or subword) complexity function of x.

Question. Show that an infinite word x is u-periodic if and only if F_x is bounded by a constant.

Solution

If x is u-periodic it can be written yz^∞, where z is primitive and either y is empty or y and z end with two distinct letters. With this normalised representation of x, we get $F_x(n) = |yz|$ for every length $n \geq |yz|$, which shows that F_x is bounded by a constant.

Conversely, assume that F_x is bounded by an integer constant $m > 0$. Since $F_x(\ell) \leq F_x(\ell+1)$ for every length ℓ, the bound implies that $F_x(n) = F_x(n+1)$ for some length n. This implies that all occurrences of each length-n factor v are followed by the same letter b_v in x. Consequently, we can consider the next factor function *next* defined on non-empty factors u of length $n+1$ as follows: $next(u) = vb_v$ where $u = av$ for a letter a.

Let w be the prefix of length n of the infinite word x. There exist p and s such that $next^s(w) = next^{s+p}(w)$, since there are only finitely many factors of length n. Thus, x is u-periodic with period p starting from position s. This completes the proof.

Notes

The u-periodicity of x is also equivalent to the condition $F_x(n) \leq n$ for some length n.

The set of boundary infinite words x for which $F_x(n) = n + 1$, for every n, is known as the set of infinite Sturmian words. They are non-u-periodic infinite words with the minimal factor complexity. In particular, the infinite Fibonacci word has this property.

More on the subject is in the book by Allouche and Shallit [7] and in the tutorial by Berstel and Karhumäki [34].

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

89 Periodicity of Morphic Words

The problem shows that it is possible to test whether an infinite word generated by a (finite) morphism is periodic.

An infinite morphic word is obtained by iterating a morphism θ from A^+ to itself, where $A = \{\mathtt{a}, \mathtt{b}, \ldots\}$ is a finite alphabet. To do so, we assume that θ is prolongable over the letter $\mathtt{a}$, that is, $\theta(\mathtt{a}) = \mathtt{a}u$ for $u \in A^+$. Then $\Theta = \theta^\infty(\mathtt{a})$ exists and is $\mathtt{a}u\theta(u)\theta^2(u)\cdots$. The infinite word Θ is a fixed point of θ, that is, $\theta(\Theta) = \Theta$.

The infinite word Θ is periodic if it can be written z^∞ for some (finite) words z, $z \neq \varepsilon$.

To avoid unnecessary complications we assume that the morphism θ is both irreducible, which means that any letter is accessible from any letter (for any $c, d \in A$ the letter d appears in $\theta^k(c)$ for some integer k), and is elementary, which means it is not the product $\eta \circ \zeta$ of two morphisms $\zeta : A^+ \longrightarrow B^+$ and $\eta : B^+ \longrightarrow A^+$, where B is an alphabet smaller than A. The second condition implies that θ is injective on A^* and on A^∞.

Question. For an irreducible and elementary morphism θ prolongable over letter $\mathtt{a}$, design an algorithm that checks if $\Theta = \theta^\infty(\mathtt{a})$ is periodic and that runs in time $O(\Sigma\{|\theta(b)| : b \in A\})$.

[**Hint:** Θ is periodic if and only if it has no bispecial letter, that is, occurrences of each letter in Θ are all followed by a unique letter.]

The morphism ρ defined by $\rho(\mathtt{a}) = \mathtt{ab}$, $\rho(\mathtt{b}) = \mathtt{ca}$ and $\rho(\mathtt{c}) = \mathtt{bc}$ complies with the conditions and produces the periodic word $\rho^\infty(\mathtt{a}) = \mathtt{abcabcabc}\cdots = (\mathtt{abc})^\infty$. None of the letter is bispecial.

On the contrary, Fibonacci morphism ϕ, defined by $\phi(\mathtt{a}) = \mathtt{ab}$ and $\phi(b) = a$, also satisfies the conditions but generates the non- (ultimately) periodic Fibonacci word $\phi^\infty(\mathtt{a}) = \mathtt{abaababa}\cdots$. In it letter $\mathtt{a}$ is bispecial, since its occurrences are followed either by $\mathtt{a}$ or by $\mathtt{b}$, while occurrences of letter $\mathtt{b}$ are all followed by $\mathtt{a}$.

Solution

The decision algorithm builds on the combinatorial property: Θ is periodic if and only if it has no bispecial letter. Intuitively, if Θ has an infinite number of bispecial factors, its factor complexity is not bounded and it is not ultimately periodic (see Problem 88).

If the condition holds, that is, if Θ contains no bispecial letter, each letter is fully determined by the letter occurring before it. And since all letters of

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

the alphabet appear in Θ the period word corresponds to a permutation of the alphabet. Thus the period of Θ is $|A|$.

Conversely, let us assume that Θ contains a bispecial letter and prove it is not periodic.

Let b be a bispecial letter, that is, bc and bd appear in Θ, for two distinct letters c and d. Due to the irreducibility of θ, the letter $\mathtt{a}$ appears in $\theta^k(b)$ for some k. Since θ is injective, $\theta^k(bc) \neq \theta^k(bd)$. Let i and j be starting positions of $\theta^k(bc)$ and $\theta^k(bd)$ on Θ. Since θ is injective on A^∞, $\Theta[i\,..\,\infty) \neq \Theta[j\,..\,\infty)$. Their longest common prefix v is then bispecial and contains the letter $\mathtt{a}$.

We show more generally that for any bispecial factor v of Θ, v containing the letter $\mathtt{a}$, there exists a longer factor with the same property.

Let i and j be two positions on Θ with $\Theta[i\,..\,i+m] = vc$ and $\Theta[j\,..\,j+m] = vd$, and c and d be distinct letters. Let $y = \Theta[i\,..\,\infty)$ and $z = \Theta[j\,..\,\infty)$. Then, again from the injectivity of θ on A^∞, we get $\theta(y) \neq \theta(z)$. Let $\theta(v)u$ be the longest common prefix of $\theta(y)$ and $\theta(z)$. So there exist two letters e and f, $e \neq f$, for which $\theta(v)ue$ and $\theta(v)uf$ are factors of Θ. Since v contains $\mathtt{a}$, $|\theta(v)u| > |v|$.

Repeating the argument, we get an infinite sequence of bispecial factors of Θ. For each such v of length n we have $F_\Theta(n+1) > F_\Theta(n)$ ($F_\Theta(n)$ is the number of factors of length n occurring in Θ) because any (length-n) word has a prolongation in Θ and v has two. This implies that $\lim_{i\to\infty} F_\Theta(i) = \infty$ and shows (see Problem 88) that Θ is not periodic, not even ultimately periodic.

The algorithm derived from the combinatorial property consists in testing if Θ contains a bispecial letter, which can be implemented to run in time $O(\Sigma\{|\theta(b)| : b \in A\})$.

Notes

The present proof of the combinatorial property is derived from the original proof by Pansiot [201] and can be found in the book by Kůrka [171, chapter 4]. The notion of an elementary morphism is from Rozenberg and Salomaa [209]. The decidability of the ultimate periodicity for non-elementary morphic words is also proved in [201].

A common property on morphisms is primitivity, an analogue to primitivity of integer matrices, a property stronger than irreducibility (the exponent k is the same for all pairs of letters). But a weaker condition can lead to the same conclusion, like when all letters appear in $\theta^k(\mathtt{a})$ for some $k > 0$. With such a condition, the above proof applies to the following morphism ξ that is not irreducible and produces $\Xi = \xi^\infty(\mathtt{a}) = \mathtt{abcdabcd}\cdots = (\mathtt{abcd})^\infty$. The same word is produced by the irreducible morphism ψ.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

$$\begin{cases} \xi(\mathtt{a}) = \mathtt{abcda} \\ \xi(\mathtt{b}) = \mathtt{b} \\ \xi(\mathtt{c}) = \mathtt{c} \\ \xi(\mathtt{d}) = \mathtt{d} \end{cases} \qquad \begin{cases} \psi(\mathtt{a}) = \mathtt{abcd} \\ \psi(\mathtt{b}) = \mathtt{b} \\ \psi(\mathtt{c}) = \mathtt{c} \\ \psi(\mathtt{d}) = \mathtt{dabcd} \end{cases}$$

More on the topic appears in the section 'Shift Spaces' of [8].

90 Simple Anti-powers

A dual notion of periodicity or local periodicity is that of anti-powers, which is introduced in the problem.

A word $u \in \{1, 2, \ldots, k\}^+$ is an anti-power if each of its letters appear exactly once in it. It is a permutation of a subset of the alphabet, that is, $alph(u) = |u|$.

Question. Show how to locate in time $O(n)$ anti-powers of length k occurring in a word $x \in \{1, 2, \ldots, k\}^n$.

For example, 13542 and 54231 occur in 341354231332 $\in \{1, 2, \ldots, 5\}^+$ at positions 2 and 4 and are its only anti-powers of length 5.

Solution

The problem can be extended to locate the longest anti-power ending at any position j on x. To do so, let $antip[j]$ be

$$\max\{|u| : u \text{ antipower suffix of } x[0\,..\,j]\}.$$

The table corresponding to the word 341354231332 shows its two anti-powers of length 5 13542 and 54231 ending respectively at positions 6 and 8 since $antip[6] = antip[8] = 5$.

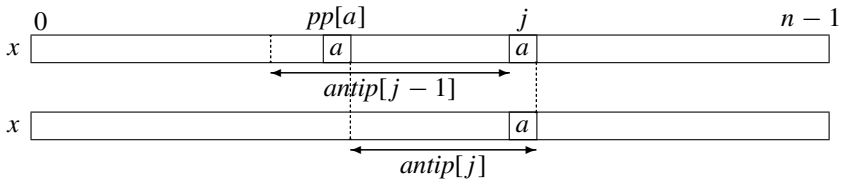
j	0	1	2	3	4	5	6	7	8	9	10	11
$x[j]$	3	4	1	3	5	4	2	3	1	3	3	2
$antip[j]$	1	2	3	3	4	4	**5**	4	**5**	2	1	2

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

The computation of table *antip* associated with x solves the question because an anti-power of length k ends at position j on x if $antip[j] = k$. Algorithm AntiPowers computes *antip* for a word in $\{1, 2, \ldots, k\}^+$.

```
AntiPowers(x ∈ {1,2,...,k}^+)
 1  for each a ∈ {1,2,...,k} do
 2      pp[a] ← −1
 3  pp[x[0]] ← 0
 4  antip[0] ← 1
 5  for j ← 1 to |x| − 1 do
 6      a ← x[j]
 7      if j − pp[a] > antip[j − 1] then
 8          antip[j] ← antip[j − 1] + 1
 9      else antip[j] ← j − pp[a]
10      pp[a] ← j
11  return antip
```

Algorithm AntiPowers computes the table sequentially and uses an auxiliary array *pp* to do it. The array indexed by letters stores at a given step the previous position $pp[a]$ of occurrences of each letter a met so far.



The correctness of AntiPowers is rather straightforward. Indeed, if the current letter a at position j does not occur in the longest anti-power ending at position $j-1$, the length of the anti-power ending at j is one unit more (line 8). Otherwise, as illustrated by the picture, $x[pp[a] + 1 \,..\, j - 1]$ is an anti-power not containing a, which gives the length $j - pp[a]$ of the longest anti-power ending at j (line 9).

It is clear that the running time of AntiPowers is $O(n)$.

Notes

The notion of an anti-power introduced by Fici et al. [113] refers to a word that is a concatenation of blocks of the same length but pairwise distinct. Authors show that every infinite word contains anti-powers of any anti-exponent (number of block). In [20], Badkobeh et al. design an optimal algorithm to locate these anti-powers with a specified anti-exponent. The above algorithm is the first step of their solution. See also [165].

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

91 Palindromic Concatenation of Palindromes

Palindromes constitute another type of regularity different from periodicity. They appear naturally in data folding when the process requires segments of data to be matched like in some genomic sequences. The problem deals with palindromes occurring in a product of palindromes.

Given a finite set of words X, computing the number of all palindromes in X^2 can be easily done in $n \cdot |X|$ time, where n is the total length of words in X. However, there is a much simpler and more efficient method when X is itself a set of palindromes.

Question. Given a finite set X of binary palindromes whose total length is n, design an algorithm computing the number of (distinct) palindromes in X^2 and running in time $O(n + |X|^2)$.

[**Hint:** When x and y are palindromes, xy is also a palindrome if and only if $xy = yx$.]

Solution

The algorithm below is based on the crucial combinatorial property stated in the hint. Let us start proving it.

Let x and y be palindromes. If xy is a palindrome then we have $x \cdot y = (x \cdot y)^{\mathrm{R}} = y^{\mathrm{R}} \cdot x^{\mathrm{R}} = y \cdot x$.

Conversely, if $xy = yx$ then x and y have the same primitive root (consequence of Lemma 2), which is also palindromic. Consequently it follows that $xy = (xy)^{\mathrm{R}}$.

From the property, the algorithm reduces to considering words in X that have the same primitive root. We execute the following algorithm:

- Compute the root of each word.
- After roots are lexicographically sorted, split them into groups with the same root.
- In each group Y, compute the number of palindromes in Y^2. As the roots are the same we only need to compute the size of the set $\{|u| + |v| : u, v \in Y\}$, which can be done in $O(|Y|^2)$ time.

The last step can be performed in time $|Y|^2$ for each group, and altogether in time $O(|X|^2)$ since the sum of sizes of Y's is $|X|$. Sorting and computing the roots takes $O(n)$ time on a fixed-size alphabet. Consequently the algorithm works in the required $O(n + |X|^2)$ time.

Notes

The problem appeared in the 13th Polish Olympiad in Informatics, 2006.

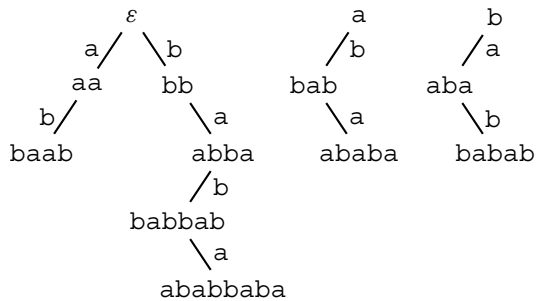
https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

92 Palindrome Trees

The notion of a *palindrome forest* $\mathcal{P}(x)$ provides a structural representation of all palindromes occurring in a word x. The data structure is used to perform different operations on the set of palindromic factors, such as to access efficiently to the longest palindrome ending at a given position on x or to count the number of occurrences of each palindrome in x.

The forest consists of a collection of trees in which each represents all palindromic factors in a similar way as a Suffix tree represents all factors. Suffix links are also part of the structure to get an efficient construction. However, palindrome forests are simpler than Suffix trees, since each edge is labelled by a single letter.

Each node of $\mathcal{P}(x)$ is a palindrome occurring in x. From a node z, there is an edge $z \xrightarrow{a} aza$ labelled by the letter a if aza is a palindrome in x. The empty word ε is the root of the tree for even palindromes. And each letter occurring in w is the root of a tree for odd palindromes having the letter at their centre. The picture shows the forest $\mathcal{P}(\texttt{ababbababaab})$ that comprises three palindrome trees.



Each node of the trees can be represented by an interval $[i \,..\, j]$ of positions corresponding to an occurrence of the palindrome $x[i \,..\, j]$. The palindrome is also fully determined by the path from the root to the node.

Question. Assume the alphabet is of constant size. Show how to construct the palindrome forest of a word in linear time according to the word length.

[**Hint:** Use suffix links.]

Solution

Algorithm PALINDROMEFOREST builds the palindrome forest of its input word x. The main trick of the construction is to augment the structure with

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

suffix links defined as follows. For a non-empty palindrome u its suffix link points to the longest palindrome that is a proper suffix of u. It is denoted by $palsuf(u)$ and may be the empty word. A suf-ancestor of u is any node accessible from u, including u itself, by iterating suffix links.

Assume u is a palindromic suffix of $x[0\,..\,i-1]$. Let $upward(u, x[i])$ be either the lowest suf-ancestor v of u for which $x[i]vx[i]$ is a suffix of $w[0\,..\,i]$ or the empty word ε.

To build the forest of x, the algorithm processes the word online. Initially, the forest consists of the roots of its trees, that is, nodes ε and a, for letters a occurring in x. Suffix links on nodes are maintained during the process, and the variable u of the algorithm stores the longest palindrome that is a suffix of the prefix of x read so far.

Inside the main for loop, the computation of the next value of u that includes the current letter $x[i]$ is done with the crucial help of *upward* at lines 4–7. The rest of the step at lines 9–15 consists in updating the forest in case a new node has to be added.

PalindromeForest(x non-empty word)

```
 1  initialise the forest P
 2  u ← x[0]
 3  for i ← 1 to |x| − 1 do
 4      v ← upward(u, x[i])
 5      if v = ε and x[i − 1] ≠ x[i] then
 6          u ← x[i]
 7      else u ← x[i]vx[i]
 8      if u ∉ P then
 9          add node u and edge v --x[i]--> u to P
10          v ← upward(palsuf(v), x[i])
11          if v = ε then
12              if x[i − 1] ≠ x[i] then
13                  palsuf(u) ← x[i]
14              else palsuf(u) ← ε
15          else palsuf(u) ← x[i]vx[i]
16  return P
```

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

The algorithm works in linear time mostly because the number of steps in computing *upward* shortens proportionally the depth of u and of $palsuf(v)$ in the forest. In addition, each of these two depths increases by at most one unit in each iteration.

Notes

The tree structure of palindromes has been investigated by Rubinchik and Shur in [210], where it is called an *eertree*. It has been later used in the design of several algorithms related to palindromes.

93 Unavoidable Patterns

Patterns of the problem are defined with a specific alphabet of variables in addition to the finite alphabet $\mathtt{A} = \{\mathtt{a}, \mathtt{b}, \ldots\}$. Variables are from the infinite alphabet $\mathtt{V} = \{\alpha_1, \alpha_2, \ldots\}$. A pattern is a word whose letters are variables. A typical pattern is $\alpha_1\alpha_1$: it appears in a word that contains a square. The aim of the problem is to produce unavoidable patterns.

A word $w \in \mathtt{A}^+$ is said to contain a pattern $P \in \mathtt{V}^*$ if $\psi(P)$ is a factor w for some morphism $\psi : alph(P)^+ \to \mathtt{A}^+$. If not, w is said to avoid P. A pattern is *avoidable* if there are infinitely many words of $\mathtt{A}^+$ avoiding it, which is equivalent (because $\mathtt{A}$ is finite) to the existence of an infinite word in $\mathtt{A}^\infty$ whose finite factors avoid it. For example, $\alpha_1\alpha_1$ is avoidable if the alphabet has at least three letters, but is unavoidable on a binary alphabet (see Problem 79).

Zimin patterns Z_n are standard examples of unavoidable patterns. They are defined, for $n > 0$, by

$$Z_0 = \varepsilon \text{ and } Z_n = Z_{n-1} \cdot \alpha_n \cdot Z_{n-1}.$$

In particular, a word contains the Zimin pattern Z_n if it contains a factor whose Zimin type is at least n (see Problem 43). For example, the word `aaaaabaabbaaaabaabb` contains Z_3 since its factor `aaabaabbaaaabaa` is the image of $Z_3 = \alpha_1\alpha_2\alpha_1\alpha_3\alpha_1\alpha_2\alpha_1$ by the morphism ψ defined by

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

$$\begin{cases} \psi(\alpha_1) = \mathtt{aa} \\ \psi(\alpha_2) = \mathtt{ab} \\ \psi(\alpha_3) = \mathtt{bba} \end{cases}$$

Question. Show that Zimin patterns Z_n, $n > 0$, are unavoidable.

Solution

Let k be the size of the alphabet $\mathtt{A}$. Define the sequence of lengths, for $n > 1$, by

$$\ell_1 = 1 \text{ and } \ell_n = (\ell_{n-1} + 1) \cdot k^{\ell_{n-1}} + \ell_{n-1} - 1,$$

and consider the following observation before answering the question.

Observation. Any word of length ℓ_n, $n > 1$, in $\mathtt{A}^*$ has a factor of the form uvu, where $|u| = \ell_{n-1}$ and $|v| > 0$.

Proof Any word w of length ℓ_n contains $(\ell_{n-1} + 2) \cdot k^{\ell_{n-1}}$ factors of length ℓ_{n-1}. Since the number of distinct factors of length ℓ_{n-1} is at most $k^{\ell_{n-1}}$ there is a word u of length ℓ_{n-1} having at least $\ell_{n-1} + 2$ occurrences in w. Consequently there are two occurrences at distance at least $\ell_{n-1} + 1$ and there should be a non-empty word v between these occurrences. The word uvu is a factor of the required form. ∎

To answer the question, it is enough to show that each word of length ℓ_n, $n > 0$, contains the Zimin pattern Z_n.

The proof is by induction on n. Obviously each non-empty word contains the pattern Z_1. Assuming that each word of length ℓ_{n-1} contains Z_{n-1} we are to show that any word w of length ℓ_n contains Z_n. Due to the above observation w contains a factor of the form uvu, where $|u| = \ell_{n-1}$ and $|v| > 0$.

By the inductive hypothesis u contains Z_{n-1}, hence $u = u_1u_2u_3$, where $u_2 = \psi(Z_{n-1})$ for a morphism $\psi : \{\alpha_1, \alpha_2, \ldots, \alpha_{n-1}\}^+ \to \mathtt{A}^+$. Then w contains the factor $u_2 \cdot z \cdot u_2$, where $z = u_3vu_1$. Extending ψ by setting $\psi(\alpha_n) = z$, w contains a morphic image of Z_n. This completes the proof.

Notes

Denote by $f(n)$ the length of a longest binary word not containing Z_n. Due to the unavoidability result $f(n)$ is finite. However, finiteness here almost meets infinity, since for instance $f(8) \geq 2^{(2^{16})} = 2^{65536}$ (see [48]). Even for short patterns values of $f(n)$ may be large; for example, there are binary words of length 10482 avoiding Z_4.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press

Pattern unavoidability is decidable as proved by Zimin (see [176, chapter 3]). An algorithm can be based on Zimin words, since it is known that a pattern P containing n variables is unavoidable if and only if it is contained in Z_n. In other words, Zimin words are unavoidable patterns and they contain all unavoidable patterns.

However, the existence of a deterministic polynomial-time algorithm for the pattern avoidability problem is still an open question. It is only known that the problem is in the NP class of complexity.

https://doi.org/10.1017/9781108835831.006 Published online by Cambridge University Press