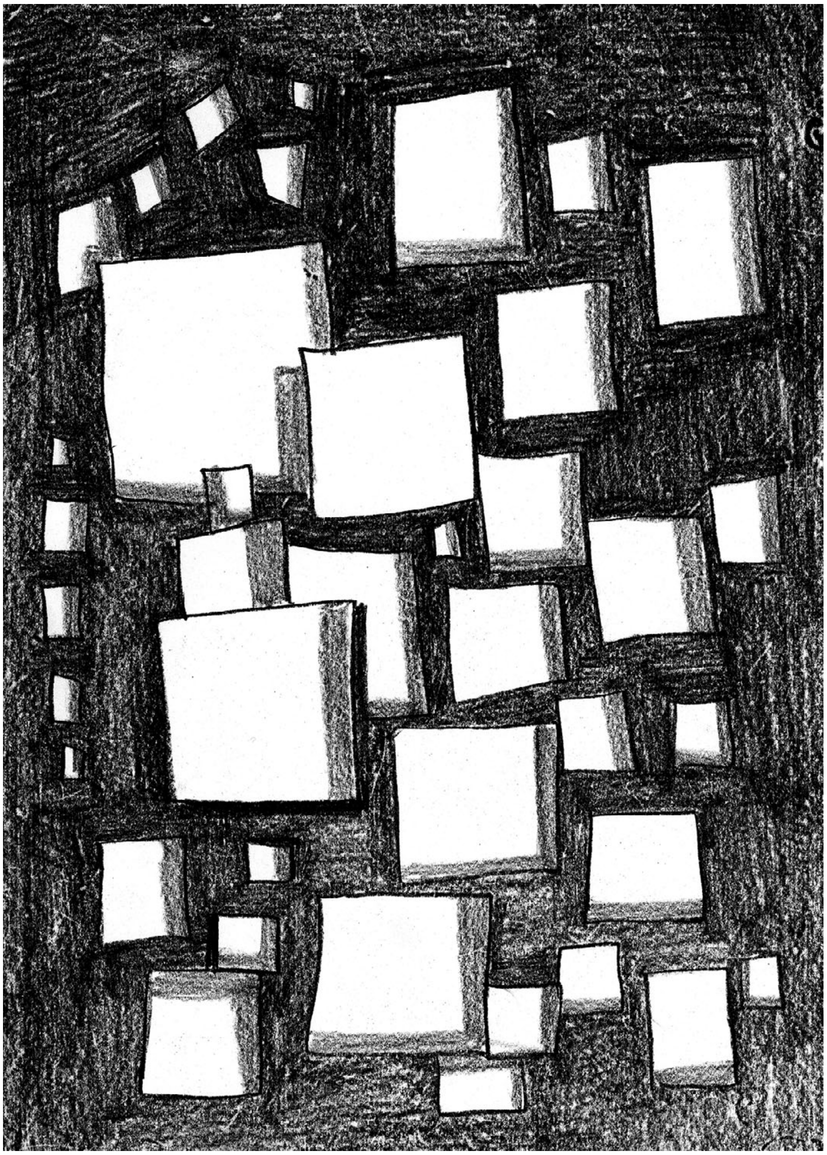
7 Miscellaneous



108 Binary Pascal Words

Pascal's triangle displays binomial coefficients following Pascal's rule:

$$\binom{n}{i} = \binom{n-1}{i-1} + \binom{n-1}{i}.$$

The regular structure of the triangle permits fast access to coefficients. In the problem, the nth binary Pascal word P_n is the nth row of Pascal's triangle modulo 2, that is, for $0 \le i \le n$:

$$P_n[i] = \binom{n}{i} \bmod 2.$$

Here are the resulting words P_n for $0 \le n \le 6$:

```
P0  =  1
P1  =  1  1
P2  =  1  0  1
P3  =  1  1  1  1
P4  =  1  0  0  0  1
P5  =  1  1  0  0  1  1
P6  =  1  0  1  0  1  0  1
```

Question. Given the binary representations $r_k r_{k-1} \cdots r_0$ of n and $c_k c_{k-1} \cdots c_0$ of i, show how to compute in time $O(k)$ the letter $P_n[i]$ and the number of occurrences of 1 in P_n.

[**Hint:** Possibly use Lucas's Theorem.]

Theorem [Lucas, 1852]. If p is a prime number and $r_k r_{k-1} \cdots r_0$, $c_k c_{k-1} \cdots c_0$ are the based p representations of the respective integers r and c with $r \ge c \ge 0$, then

$$\binom{r}{c} \bmod p = \prod_{i=0}^{k} \binom{r_i}{c_i} \bmod p.$$

Solution

The following property leads to a $O(k)$-time algorithm for computing letters in P_n.

Property 1. $P_n[i] = 1 \iff \forall j\ 1 \le j \le k\ (r_j = 0 \Rightarrow c_j = 0)$.

Proof The equivalence is a consequence of Lucas's Theorem. In our situation $p = 2$ and $r_j, c_j \in \{0, 1\}$. Then

$$\binom{r_j}{c_j} \bmod 2 = \mathtt{1} \iff (r_j = \mathtt{0} \Rightarrow c_j = \mathtt{0}),$$

which directly implies the property. ∎

Example. $P_6[4] = \binom{6}{4} \bmod 2 = \mathtt{1}$, since the binary representations of 6 and 4 are `110` and `010` respectively.

To answer the second part of the question let $g(n)$ denote the number of occurrences of `1` in the binary representation of the non-negative integer n. The following fact provides a simple algorithm to compute the number of occurrences of `1` in P_n in the required time.

Property 2. The number of ones in P_n is $2^{g(n)}$.

Proof Let $r_k r_{k-1} \cdots r_0$ and $c_k c_{k-1} \cdots c_0$ be the respective binary representations of n and i. Let

$$R = \{j : r_j = \mathtt{1}\} \text{ and } C = \{j : c_j = \mathtt{1}\}.$$

According to property 1 we have

$$\binom{n}{i} \bmod 2 = \mathtt{1} \iff C \subseteq R.$$

Hence the sought number equals the number of subsets C of R, which is $2^{g(n)}$, due to the definition of $g(n)$. This completes the proof. ∎

Notes

An easy description of Lucas's theorem is by Fine [114].

Among the many interesting properties of Pascal words let us consider the following. For a word $w = w[0\,.\,.\,k]$ and a set X of natural numbers, define

$$\mathit{Filter}(w, X) = w[i_1]w[i_2]\cdots w[i_t],$$

where $i_1 < i_2 < \cdots < i_t$ and $\{i_1, i_2, \ldots, i_t\} = X \cap [0\,.\,.\,k]$. Then, for a positive integer n and the set Y of powers of 2, we get the equality

$$\mathit{Filter}(P_n, Y) = \text{ reverse binary representation of } n.$$

A simple proof follows from the structure of the ith diagonal of Pascal triangle modulo 2, counting diagonals from left to right and starting with 0. The 0th diagonal consists of `1`'s, the next one consists of a repetition of `10`, and so on. Now considering the table whose rows are consecutive numbers, the columns of this table show similar patterns.

109 Self-Reproducing Words

Word morphisms are frequently used to produce finite or infinite words because they are defined by the images of single letters. In the problem we consider another type of function that may be regarded as context dependent and is a kind of sequential transducer.

The working alphabet is $A = \{0,1,2\}$. The image by h of a word $w \in A^+$ is the word

$$h(w) = (0 \oplus w[0])\,(w[0] \oplus w[1])\,(w[1] \oplus w[2]) \cdots (w[n-1] \oplus 0),$$

where $\oplus$ is the addition modulo 3. Iterating h from an initial word of A^+ produces longer words that have very special properties, as shown with the two examples.

Example 1. When the process is applied to the initial word $x = 1221$, it produces a list of ternary words starting with

$$\begin{array}{lcl}
h^0(x) & = & 1\ 2\ 2\ 1 \\
h^1(x) & = & 1\ 0\ 1\ 0\ 1 \\
h^2(x) & = & 1\ 1\ 1\ 1\ 1\ 1 \\
h^3(x) & = & 1\ 2\ 2\ 2\ 2\ 2\ 1 \\
h^4(x) & = & 1\ 0\ 1\ 1\ 1\ 1\ 0\ 1 \\
h^5(x) & = & 1\ 1\ 1\ 2\ 2\ 2\ 1\ 1\ 1 \\
h^6(x) & = & 1\ 2\ 2\ 0\ 1\ 1\ 0\ 2\ 2\ 1 \\
h^7(x) & = & 1\ 0\ 1\ 2\ 1\ 2\ 1\ 2\ 1\ 0\ 1 \\
h^8(x) & = & 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1 \\
h^9(x) & = & 1\ 2\ 2\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 2\ 2\ 1
\end{array}$$

In particular, $h^9(x) = x \cdot 00000 \cdot x$, which shows that the word x has been reproduced.

Example 2. Applied to $y = 121$, the associated list starts with

$$\begin{array}{lcl}
h^0(y) & = & 1\ 2\ 1 \\
h^1(y) & = & 1\ 0\ 0\ 1 \\
h^2(y) & = & 1\ 1\ 0\ 1\ 1 \\
h^3(y) & = & 1\ 2\ 1\ 1\ 2\ 1
\end{array}$$

Question. Show that for any word $w \in A^+$ the two properties hold:

(A) There is an integer m for which $h^m(w)$ consists of two copies of w separated by a factor of zeros (i.e., a factor in 0^*).

(B) If $|w|$ is a power of 3 then $h^{|w|}(w) = w\,w$.

Solution

Point (A). Let m be the minimal power of 3 not smaller than the length n of w and denote

$$\alpha(i) = \binom{m}{i} \bmod 3.$$

We use the following fact.

Observation. Let $i \in \{0, 1, \ldots, m\}$. Since m is a power of 3, then we have (obviously) $\alpha(i) = 1$ if $i \in \{0, m\}$; otherwise (slightly less obvious) $\alpha(i) = 0$.

We refer the reader to Problem 108 that shows a relation between the present process and Pascal's triangle.

Starting with the word `1`, after m steps, at position i on $h^m(\mathtt{1})$ we have the letter $\alpha(i)$.

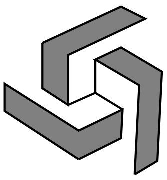
Due to the observation after m steps the contribution of a single `1` at position t to the letter at position $t + i$ is $\alpha(i)$. Therefore in the word $h^m(w)$ the prefix w remains as it is, since $\alpha(0) = 1$, and it is copied m positions to the right, since $\alpha(m) = 1$. Letters at other positions in $h^m(w)$ are `0`, since $\alpha(i) = 0$ for $i \notin \{0, m\}$. This solves point (A) of the question.

Point (B). Following the above argument, if $|w|$ is a power of 3 the word w is copied m positions to the right, which gives the word ww, since the prefix of size m is unchanged. This solves point (B) of the question.

Notes

When the alphabet is $A_j = \{0, 1, \ldots, j - 1\}$ and j is a prime number we can choose $m = \min\{j^i : j^i \geq n\}$, where n is the length of the initial word w. When j is not prime the situation is more complicated: now we can choose $m = j \cdot n!$, but in this case the word separating the two copies of w can contain non-zero values.

The problem presented here is from [13], where a 2-dimensional (more interesting) version is also presented.



110 Weights of Factors

The *weight* of a word on the alphabet $\{\mathtt{1},\mathtt{2}\}$ is the arithmetic sum of its letters. The problem deals with the weights of all the non-empty factors of a given word of length n. In this limited alphabet, the potentially maximal weight is $2n$ and the maximal number of different weights among factors is $2n-1$.

For example, the number of weights of factors of the word `2221122` is 10, namely they are $1,2,\dots,8,10,12$.

Question. Design a simple linear-time algorithm computing the number of different weights of non-empty factors of a word $x \in \{\mathtt{1},\mathtt{2}\}^+$.

Question. Show that after preprocessing the word x in linear time each query of the type 'is there is a non-empty factor of x of positive weight k?' can be answered in constant time. The memory space after preprocessing should be of constant size.

Solution

Before going to solutions, let us show some properties of weights. For a positive integer k let

$$\mathrm{SameParity}(k) = \{i : 1 \le i \le k \text{ and } (k-i) \text{ is even}\}.$$

The size of the set is $|\mathrm{SameParity}(k)| = \lceil \frac{k}{2} \rceil$.

Let $sum(i,j)$ denote the weight of the factor $x[i\,..\,j]$ of x, $i \le j$. Extremely simple solutions to the questions derive from the following fact.

Fact. If $k > 0$ is the weight of some factor of x then each element of SameParity(k) is also the weight of a non-empty factor of x.

Proof Let $sum(i,j) = k$ be the weight of $x[i\,..\,j]$, $i \le j$. If $x[i] = 2$ or $x[j] = 2$ then chopping the first or the last letter of $x[i\,..\,j]$ produces a factor with weight $k-2$ if it is non-empty. Otherwise $x[i] = x[j] = 1$ and after chopping both end letters we get again a factor with weight $k-2$ if it is non-empty. Iterating the process proves the fact. ■

Note that, if $x \in \mathtt{2}^+$, answers to questions are straightforward because x has $|x|$ different weights, 2, 4, …, $2|x|$. In the rest we assume x contains at least one occurrence of letter `1`. Let *first* and *last* be respectively the first and the last positions on x of letter `1` and let

$$s = sum(0, n-1) \text{ and } t = \max\{sum(first+1, n-1), sum(0, last-1)\}.$$

In other words, s is the weight of the whole word x and t is the maximum weight of a prefix or a suffix of x that is of different parity than s.

The next observation is a consequence of the above fact.

Observation. The set of weights of all non-empty factors of a word x is the union $\text{SameParity}(s) \cup \text{SameParity}(t)$.

Number of different weights. Since the number of different weights of non-empty factors of x is

$$|\text{SameParity}(s)| + |\text{SameParity}(t)| = \left\lceil \frac{s}{2} \right\rceil + \left\lceil \frac{t}{2} \right\rceil,$$

its computation amounts to compute s and t, which is done in linear time.

For the word `2221122`, we have $s = 12$, $t = \max\{5,7\} = 7$ and the number of weights of its factors is $\lceil \frac{12}{2} \rceil + \lceil \frac{7}{2} \rceil = 10$ as seen above.

Constant-time query. The preprocessing consists in computing the two values s and t corresponding to the word x, which is done in linear time. After preprocessing, the memory space is used only to store the values s and t.

Then, to answer the query 'is $k > 0$ the weight of a non-empty factor of x?' it suffices to check the condition

$$k \le t \text{ or } ((s-k) \text{ is non-negative and even}),$$

which is done in constant time.

Notes

What about larger alphabets, for example $\{1,2,3,4,5\}$? An efficient algorithm is still possible but there is nothing as nice and simple as the above solution.

Let x be a word whose length is a power of 2. An *anchored* interval $[i\,..\,j]$ is a subinterval of $[0\,..\,|x|-1]$ with i in the left half and j in the right half of the interval. The associated factor $x[i\,..\,j]$ of x is called an anchored factor. Using fast convolution, all distinct weights of anchored factors of a word x can be computed in time $O(|x|\log|x|)$. We can take characteristic vectors of the sets of weights of suffixes of the left half and of prefixes of the right half of x. Both vectors are of length $O(|x|)$. Then the convolution of these two vectors (sequences) gives all the weights of anchored factors.

Over the alphabet $\{1,2,3,4,5\}$, using a recursive approach, all distinct weights of factors of a word of length n are computed in time $O(n(\log n)^2)$ because the running time $T(n)$ to do it satisfies the equation $T(n) = 2T(n/2) + O(n\log n)$.

111 Letter-Occurrence Differences

For a non-empty word x, $diff(x)$ denotes the difference between the numbers of occurrences of the most frequent letter and of the least frequent letter in x. (They can be the same letter.)

For example,

$$diff(\texttt{aaa}) = 0 \text{ and } diff(\texttt{cabbcadbeaebaabec}) = 4.$$

In the second word, a and b are the most frequent letters, with five occurrences, and d the least frequent letter, with one occurrence.

Question. Design an $O(n|A|)$ running time algorithm computing the value $\max\{diff(x) : x \text{ factor of } y\}$ for a non-empty word y of length n over the alphabet A.

[**Hint:** First consider $A = \{\texttt{a},\texttt{b}\}$.]

Solution

Assume for a moment that $y \in \{\texttt{a},\texttt{b}\}^+$ and let us search for a factor x of y in which b is the most frequent letter. To do so, y is transformed into Y by substituting -1 for a and 1 for b. The problem then reduces to the computation of a factor with the maximum arithmetic sum and containing at least one occurrence of 1 and of -1.

Before considering a general alphabet, we consider a solution for the binary alphabet $\{\texttt{-1},\texttt{1}\}$ and introduce a few notations.

For a given position i on the word $Y \in \{\texttt{-1},\texttt{1}\}^+$, let sum_i be the sum $Y[0] + Y[1] + \cdots + Y[i]$ and let $pref_i$ be the minimum sum corresponding to a prefix $Y[0\,..\,k]$ of Y for which both $k < i$ and $Y[k+1\,..\,i]$ contains at least one occurrence of -1. If there is no such k, let $pref_i = \infty$.

The following algorithm delivers the expected value for the word Y.

```
MaxDifference($Y$ non-empty word on $\{-1,1\}$)
1  $(maxdiff, prevsum, sum) \leftarrow (0,0,0)$
2  $pref \leftarrow \infty$
3  for $i \leftarrow 0$ to $|Y|-1$ do
4      $sum \leftarrow sum + Y[i]$
5      if $Y[i] = -1$ then
6          $pref \leftarrow \min\{pref, prevsum\}$
7          $prevsum \leftarrow sum$
8      $maxdiff \leftarrow \max\{maxdiff, sum - pref\}$
9  return $maxdiff$
```

Algorithm MaxDifference implements the following observation to compute the maximal difference among the factors of its input.

Observation. Assume $pref \neq \infty$. Then the letter $Y[k]$ is `-1` and the difference $diff(Y[k+1\,..\,i])$ is $sum - pref$. Moreover $Y[k+1\,..\,i]$ has maximal $diff$ value among the suffixes of $Y[0\,..\,i]$.

In this way the problem is solved in linear time for a word on a two-letter alphabet.

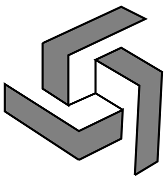
On a larger alphabet we apply the following trick. For any two distinct letters a and b of the word y, let $y_{a,b}$ be the word obtained by removing all other letters occurring in y. After changing $y_{a,b}$ to $Y_{a,b}$ on the alphabet {`-1`,`1`}, Algorithm MaxDifference produces the maximal difference among factors of $Y_{a,b}$, which is the maximal difference among factors of $y_{a,b}$ as well.

The required value is the maximum result among results obtained by running MaxDifference on $Y_{a,b}$ for all pairs of letters a and b separately.

Since the sum of lengths of all words $y_{a,b}$ is only $O(n|A|)$, the overall running time of the algorithm is $O(n|A|)$ for a word of length n over the alphabet A.

Notes

This problem appeared in the Polish Olympiad of Informatics for high school students in the year 2010.



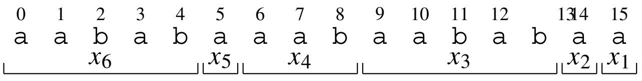
112 Factoring with Border-Free Prefixes

Searching for a border-free pattern in texts is done very efficiently without any sophisticated solution by BM algorithm (see Problem 33) because two occurrences of the pattern cannot overlap. When a pattern is not border free, its factorisation into border-free words may lead to efficient searching methods.

We say that a non-empty word u is border free if none of its proper non-empty prefix is also a suffix, that is, $Border(u) = \varepsilon$, or equivalently, if its smallest period is its length, that is, $per(u) = |u|$.

The aim of the problem is to show how a word factorises into its border-free prefixes.

Consider for example the word `aababaaabaababaa`. Its set of border-free prefixes is {`a`, `aab`, `aabab`} and its factorisation on the set is



Question. Show that a non-empty word x uniquely factorises into $x_k x_{k-1} \cdots x_1$, where each x_i is a border-free prefix of x and x_1 is the shortest border-free prefix of x.

The factorisation of x can be represented by the list of its factor lengths. On the preceding example it is $(5,1,3,5,1,1)$ and the list of factors is $x[0\,..\,4]$, $x[5\,..\,5]$, $x[6\,..\,8]$, $x[9\,..\,13]$, $x[14\,..\,14]$, $x[15\,..\,15]$.

Question. Design a linear-time algorithm for computing the border-free prefix factorisation of a word, namely the list of factor lengths.

Solution

Unique factorisation. Let $S(x)$ be the set of border-free prefixes of x. It is a suffix code, that is, if $u, v \in S(x)$ and u and v are distinct words none of them is a suffix of the other. Because on the contrary, if for example u is a proper suffix of v, since u is a non-empty prefix of v, v is not border-free, a contradiction. Then, any product of words in $S(x)$ admits a unique decomposition into such a product. This shows the uniqueness of the factorisation of x into words of $S(x)$, if the factorisation exists.

Let us prove that the factorisation exists. If x is border free, that is $x \in S(x)$, the factorisation contains only one factor, x itself. Otherwise, let u be the shortest non-empty border of x. Then u is border free, that is, $u \in S(x)$. Thus, we can iterate the same reasoning on the word xu^{-1} to get the factorisation. This yields a factorisation in which the last factor is the shortest element of $S(x)$, as required.

Factoring. The factorisation of a non-empty word x can be computed from its border table with the intermediate ***shortest-border table*** *shtbord*: *shtbord*$[\ell] = 0$ if $x[0\,..\,\ell-1]$ is border free and else is the length of its shortest border. The table is computed during a left-to-right traversal of the border table of x.

Here are the tables for the word x = `aababaaabaababaa`:

i		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$x[i]$		a	a	b	a	b	a	a	a	b	a	a	b	a	b	a	a
ℓ	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$border[\ell]$	—	0	1	0	1	0	1	2	2	3	4	2	3	4	5	6	7
$shtbord[\ell]$	—	0	1	0	1	0	1	1	1	3	1	1	3	1	5	1	1

The lengths ℓ of the border-free prefixes of x satisfy $border[\ell] = 0$, and are 1, 3 and 5 on the example.

Since the set $S(x)$ is a suffix code it is natural to compute the factorisation by scanning x from right to left. Following the above proof, lengths of factors are picked from the table of shortest non-empty borders until a border-free prefix is found.

```
FACTORISE($x$ non-empty word)
  $border \leftarrow$ BORDERS($x$)
  for $\ell \leftarrow 0$ to $|x|$ do
      if $border[\ell] > 0$ and $shtbord[border[\ell]] > 0$ then
          $shtbord[\ell] \leftarrow shtbord[border[\ell]]$
      else $shtbord[\ell] \leftarrow border[\ell]$
  $L \leftarrow$ empty list
  $\ell \leftarrow |x|$
  while $border[\ell] > 0$ do
      $L \leftarrow shtbord[\ell] \cdot L$
      $\ell \leftarrow \ell - shtbord[\ell]$
  $L \leftarrow shtbord[\ell] \cdot L$
  return $L$
```

As for the running time of Algorithm FACTORISE, it is clearly linear in addition to the computation of the border table of x, which can also be computed in linear time (see Problem 19). Therefore the overall process runs in linear time.

Notes

It is unclear whether the table of shortest non-empty borders can be computed as efficiently with the technique applied to produce the table of short borders in Problem 21.

113 Primitivity Test for Unary Extensions

A non-empty word x can be decomposed as $x = (uv)^e u$ for two words u and v where v is non-empty, $|uv| = per(x)$ is the (smallest) period of x and e is a positive integer. We set $tail(x) = v$ (not u).

For example, $tail(\texttt{abcd}) = \texttt{abcd}$ because the associated words are $u = \varepsilon$ and $v = \texttt{abcd}$, $tail(\texttt{abaab}) = \texttt{a}$ because $\texttt{abaab} = (\texttt{aba})^1\texttt{ab}$ and $tail(\texttt{abaababa}) = \texttt{ab}$ because $\texttt{abaababa} = (\texttt{abaab})^1\texttt{aba}$. The latter word is the Fibonacci word fib_4 and in general $tail(fib_n) = fib_{n-3}$, for $n \geq 3$.

The goal of the problem is to test whether xa^k is primitive when only little is known about the word x.

Question. Assume the only information on x is

- x is not unary (contains at least two distinct letters)
- $tail(x)$ is not unary or $tail(x) \in b^*$, for a letter b
- $\ell = |tail(x)|$

Show how to answer in constant time if the word xa^k is primitive, for an integer k and a letter a.

[**Hint:** $tail(x)$ is an obvious candidate to extend x to a non-primitive word.]

Solution

The solution relies on the following property, for a non-unary word x:

$$xa^k \text{ is not primitive} \Longrightarrow tail(x) = a^k.$$

Since the converse obviously holds, the property leads to a constant-time test under the hypotheses in the question because testing the primitivity of xa^k amounts to check if $tail(x) \in b^*$, that is, to check if $a = b$ and $k = \ell$. Although the property is simply stated it needs a tedious proof. We start with an auxiliary fact and go on with the crucial lemma.

Fact. If x is non-unary and $x = (uv)^e u = u'v'u'$, where $|uv| = per(x)$, $e > 0$ and $|u'| < |u|$, then v' is non-unary.

Indeed, if v' is unary, v also is, which implies that u is not (it cannot be unary with a letter different from the letter in v). Since the word u' is both a proper prefix and a suffix of u, we get $u = u'y = zu'$ for two non-empty words y and z that are respectively prefix and suffix of v' with $|y| = |z| = per(u)$. Then both are unary, implying u and x also are, a contradiction.

Unary-Extension Lemma. If x is non-unary, a is a letter, k is a positive integer and $a^k \neq tail(x)$, then xa^k is primitive.

Proof By contradiction, assume xa^k is non-primitive, that is

$$xa^k = z^j,\ j \geq 2,\ |z| = per(xa^k).$$

We deduce $|z| > k$, because otherwise both x would be unary and $|z| \neq per(x)$, since $a^k \neq tail(x)$. Since $|z|$ is a period of x, $|z| > per(x)$. We cannot have $j > 2$ because z^2 would be a prefix of x, which implies $|z| = per(x)$.

Consequently the only remaining case in this situation is when $j = 2$, that is

$$xa^k = z^2 = u'v'u'v',\ x = (uv)^i u = u'v'u',\ v' = a^k,$$

where $v' \neq v = tail(x)$, $|uv| = per(x)$ and $|u'v'| = per(xa^k)$. Let us consider two cases.

Case $|u'| < |u|$: This is impossible due to the above preliminary fact which implies that $v' = a^k$ is non-unary.

Case $|u'| > |u|$: Let us consider only the situation $i = 2$, that is, $x = (uv)^2u$. The general case $i > 1$ can be treated similarly.

Claim. $|u'| < |uv|$.

Proof (of the claim) By contradiction assume $|u'| \geq |uv|$. Then the word x admits periods $p = |uv|$ and $q = |x| - |u'| = |u'v'|$, where $p + q \leq |x|$. The Periodicity Lemma implies that p (as the smallest period) is a divisor of q. Hence p is also a period of xa^k (which has a period q). Consequently x and xa^k have the same shortest period, which is impossible. ■

Let w be such that $u'v' = uvw$. Due to the above claim w is a suffix of v' and consequently w is unary. The word u' is a prefix of uvu (as a prefix of x). This implies that $|w|$ is a period of uvu. Since w is unary, uv is also unary and the whole word x is unary, a contradiction.

In both cases we have a contradiction. Therefore the word xa^k is primitive as stated. ■

Notes

The solution to this problem is by Rytter in [215]. A time–space optimal primitivity test (linear time, constant space) is given in Problem 40 but the present problem provides a much faster solution in the considered particular case.

114 Partially Commutative Alphabets

The study of words over a partially commutative alphabet is motivated by the representation of concurrent processes in which letters are names of processes and commutativity corresponds to the non-concurrency of two processes.

We consider an alphabet A for which some pairs of letters commute. This means that we can transform the word $uabv$ to $ubav$, for any commuting pair (a,b). The corresponding (partial) commutativity relation on A denoted by $\approx$ is assumed to be symmetric.

Two words are equivalent (with respect to the commutativity relation), denoted by $u \equiv v$, if one word can be transformed into the other by a series of exchanges between adjacent commuting letters. Observe that $\equiv$ is an equivalence relation while $\approx$ usually is not.

For example, on the alphabet $A = \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}$

$$\mathtt{a} \approx \mathtt{b} \approx \mathtt{c} \approx \mathtt{d} \approx \mathtt{a} \ \Rightarrow\ \mathtt{abcdabcd} \equiv \mathtt{badbdcac}$$

due to the following commutations:

$$\begin{array}{cccccccc}
\underline{\mathtt{a}} & \underline{\mathtt{b}} & \mathtt{c} & \mathtt{d} & \mathtt{a} & \mathtt{b} & \mathtt{c} & \mathtt{d} \\
\mathtt{b} & \mathtt{a} & \underline{\mathtt{c}} & \underline{\mathtt{d}} & \mathtt{a} & \mathtt{b} & \mathtt{c} & \mathtt{d} \\
\mathtt{b} & \mathtt{a} & \mathtt{d} & \mathtt{c} & \underline{\mathtt{a}} & \underline{\mathtt{b}} & \mathtt{c} & \mathtt{d} \\
\mathtt{b} & \mathtt{a} & \mathtt{d} & \underline{\mathtt{c}} & \underline{\mathtt{b}} & \mathtt{a} & \mathtt{c} & \mathtt{d} \\
\mathtt{b} & \mathtt{a} & \mathtt{d} & \mathtt{b} & \mathtt{c} & \mathtt{a} & \underline{\mathtt{c}} & \underline{\mathtt{d}} \\
\mathtt{b} & \mathtt{a} & \mathtt{d} & \mathtt{b} & \mathtt{c} & \underline{\mathtt{a}} & \underline{\mathtt{d}} & \mathtt{c} \\
\mathtt{b} & \mathtt{a} & \mathtt{d} & \mathtt{b} & \underline{\mathtt{c}} & \underline{\mathtt{d}} & \mathtt{a} & \mathtt{c} \\
\mathtt{b} & \mathtt{a} & \mathtt{d} & \mathtt{b} & \mathtt{d} & \mathtt{c} & \mathtt{a} & \mathtt{c}
\end{array}$$

Question. Design an equivalence test that checks if $u \equiv v$ for two words u and v of length n in A^* and that runs in time $O(n|A|)$.

[**Hint:** Consider projections of words on pairs of letters.]

Solution

For two letters $a,b \in A$, $\pi_{a,b}(w)$ denotes the projection of w on the pair (a,b), that is, the word resulting from w by erasing all letters except them. Let $|w|_a$ denote the number of times letter a occurs in w. The next property is the basis of our solution.

Property. For two words u and v, $u \equiv v$ if and only if the following two conditions hold:

(i) $|u|_a = |v|_a$ for each letter $a \in A$; and

(ii) $\pi_{a,b}(u) = \pi_{a,b}(v)$ whenever a and b do not commute.

Proof It is clear that the conditions are satisfied if $u \equiv v$. Conversely, assume conditions (i) and (ii) hold. The proof is by induction on the common length of the two words.

Assume $u = au'$, where $a \in A$. We claim that we can move the first occurrence of letter a in v to the first position using the relation $\approx$. Indeed, if we are not able to do it, there is some non-commuting letter b occurring before a in v. Then $\pi_{a,b}(u) \neq \pi_{a,b}(v)$, a contradiction.

After moving a to the beginning of v we get the word av' with $av' \equiv v$ and the conditions (i) and (ii) hold for u' and v'. By the inductive assumption $u' \equiv v'$, and consequently $u = au' \equiv av' \equiv v$. This completes the proof. ■

The equivalence test consists in checking the above two conditions. Checking the first condition is obviously done in time $O(n|A|)$ (or even in time $O(n \log |A|)$ without any assumption of the alphabet).

The second condition it to check if $\pi_{a,b}(u) = \pi_{a,b}(v)$ for all pairs of letters a,b that do not commute. At a first glance this looks to produce a $O(n|A|^2)$ time algorithm. However, the sum of the lengths of all words of the form $\pi_{a,b}(u)$ is only $O(n|A|)$, which is also an upper bound on the running time of the algorithm.

Notes

The material in this problem is based on properties of partial commutations presented by Cori and Perrin in [61].

There is an alternative algorithm for the equivalence problem. We can define a canonical form of a word as its lexicographically smallest equivalent version. Hence given two words one can compute their canonical versions and test their equality. The computation of canonical forms is of independent interest.

115 Greatest Fixed-Density Necklace

A word is called a ***necklace*** if it is the lexicographically smallest word in its conjugacy class. The ***density*** of a word on the alphabet $\{0, 1\}$ is the number of occurrences of 1's occurring in it. Let $N(n,d)$ be the set of all binary necklaces of length n and density d.

The problem is concerned with the lexicographically greatest necklace in $N(n,d)$ with $0 \leq d \leq n$. For example, 00100101 is the greatest necklace in $N(8,3)$:

$$\{00000111, 00001011, 00001101, 00010011, 00010101, 00011001, 00100101\}.$$

And 01011011 is the greatest necklace in $N(8,5)$:

$$\{00011111, 00101111, 00110111, 00111011, 00111101, 01010111, 01011011\}.$$

The following intuitive property characterises the structure of largest necklaces.

***Lemma* 20** *Let C be the greatest necklace in $N(n,d)$:*

(i) *If $d \leq n/2$ then $C = 0^{c_0}10^{c_1}1 \cdots 0^{c_{d-1}}1$, where both $c_0 > 0$ and, for each $i > 0$, $c_i \in \{c_0, c_0 - 1\}$.*

(ii) *If $d > n/2$ then $C = 01^{c_0}01^{c_1} \cdots 01^{c_{n-d-1}}$, where both $c_0 > 0$ and, for each $i > 0$, $c_i \in \{c_0, c_0 + 1\}$.*

(iii) *In both cases, the binary sequence $w = (0, |c_1 - c_0|, |c_2 - c_0|, \ldots)$ is the largest necklace of its length and density.*

Question. Based on Lemma 20, design a linear-time algorithm for computing the greatest necklace in $N(n,d)$.

Solution

The lemma motivates the following two definitions when the (binary) word w is a necklace of length ℓ:

$$\phi_t(w) = 0^{t-w[0]}1\,0^{t-w[1]}1\,\cdots 0^{t-w[\ell-1]}1,$$

$$\psi_t(w) = 01^{t+w[0]}\,01^{t+w[1]}\,\cdots 01^{t+w[\ell-1]}.$$

The next two facts are rather direct consequences of the lemma and show that the functions ϕ_t and ψ_t preserve the lexicographic order. They also justify the recursive structure of the algorithm below.

Fact 1. A binary word w of length ℓ is a necklace if and only if $\phi_t(w)$ is a necklace for all $t > 0$.

Fact 2. A binary word w of length ℓ is a necklace if and only if $\psi_t(w)$ is a necklace for all $t \geq 0$.

The following algorithm then solves the problem. Point (iii) of the lemma allows to reduce the problem to a single much smaller problem by a single recursive call.

```
GreatestNecklace($n, d$ natural numbers, $d \leq n$)
 1  if $d = 0$ then
 2      return $0^n$
 3  elseif $d \leq n/2$ then
 4      $(t,r) \leftarrow (\lfloor n/d \rfloor, n \bmod d)$
 5      $w \leftarrow$ GreatestNecklace$(d, d-r)$
 6      return $\phi_t(w)$
 7  elseif $d < n$ then
 8      $(t,r) \leftarrow (\lfloor n/(n-d) \rfloor, n \bmod (n-d))$
 9      $w \leftarrow$ GreatestNecklace$(n-d, r)$
10      return $\psi_{t-1}(w)$
11  else return $1^n$
```

Example. For $n = 8$ and $d = 3$, we get $t = 2$, $r = 2$ and the recursive call gives GreatestNecklace$(3,1) = w =$ `001`. Eventually GreatestNecklace$(8,3)$ $= 0^{2-0}10^{2-0}10^{2-1}1$, which is `00100101`, as already seen above.

Example. For $n = 8$ and $d = 5$, we also get $t = 2$, $r = 2$ (since $n-d = 3$) and the recursive call gives GreatestNecklace$(3,2) = w =$ `011`. This produces GreatestNecklace$(8,5) = 01^{1+0}01^{1+1}01^{1+1}$, which is `01011011` as seen above.

The correctness of Algorithm GreatestNecklace essentially comes from the lemma and the two facts.

As for the running time of GreatestNecklace(n,d) note that recursive calls have to deal with words whose length is no more than $n/2$. Therefore the whole runs in linear time to generate the final word.

Notes

The material of the problem is by Sawada and Hartman in [218].

116 Period-Equivalent Binary Words

Two words are said to be period equivalent if they have the same set of periods or equivalently have the same length and the same set of border lengths. For example, `abcdabcda` and `abaaabaaa` are period equivalent since they share the same set of periods $\{4, 8, 9\}$ although their letters are not in one-to-one correspondence.

The goal of the problem is to show that a set of periods of a word can be realised by a binary word.

Question. Let w be a word over an arbitrarily large alphabet. Show how to construct in linear time a binary word x period equivalent to w.

[**Hint:** Consider border lengths instead of periods.]

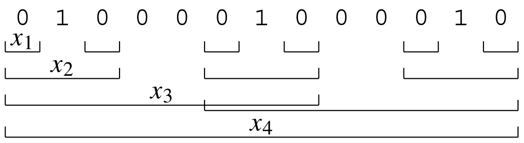
Solution

Dealing with the border lengths of w instead of its periodic structure is more convenient to solve the question and describe the corresponding algorithm. The border structure is given by the increasing list $\mathcal{B}(w) = (q_1, q_2, \dots, q_n)$ of lengths of non-empty borders of w with the addition of $q_n = |w| = N$. For example, $(1, 5, 9)$ is the list associated with `abcdabcda`.

To answer the question, from the list $\mathcal{B}(w)$ a sequence of words $(x_1, x_2, \dots, x_n)$, in which x_i is a binary word associated with the border list $(q_1, \dots, q_i)$, is constructed iteratively. The binary word period equivalent to w is $x = x_n$.

Let x_1 be a border-free word of length q_1. The word x_i of length q_i with longest border x_{i-1} is either of the form $x_i y_i x_i$ if this fits with its length or built by overlapping x_{i-1} with itself. Word y_i is unary and its letter is chosen to avoid creating undesirable borders.

Example. Let $(1, 3, 8, 13) = (q_1, q_2, q_3, q_4) = \mathcal{B}($`abacdabacdaba`$)$. Starting with the border-free word $x_1 =$ `0`, x_2 is built by inserting $y_2 =$ `1` between two occurrences of x_1. The word x_3 is built similarly from x_2 with the unary word $y_3 =$ `00`, whose letter is different from that of y_2. Eventually, $x_4 =$ `0100001000010` is built by overlapping two occurrences of x_3.



In Algorithm ALTERNATING that implements the method, *prefix*(z,k) denotes the prefix of length k of a word z when $k \le |z|$.

```
ALTERNATING($w$ non-empty word)
1  $(q_1, q_2, \ldots, q_n) \leftarrow \mathcal{B}(w)$
2  $(x_1, a) \leftarrow (\mathtt{0}\mathtt{1}^{q_1-1}, \mathtt{0})$
3  for $i \leftarrow 2$ to $n$ do
4      $gap_i \leftarrow q_i - 2q_{i-1}$
5      if $gap_i > 0$ then
6          $a \leftarrow 1 - a$
7          $x_i \leftarrow x_{i-1} \cdot a^{gap_i} \cdot x_{i-1}$
8      else $x_i \leftarrow prefix(x_{i-1}, q_i - q_{i-1}) \cdot x_{i-1}$
9  return $x_n$
```

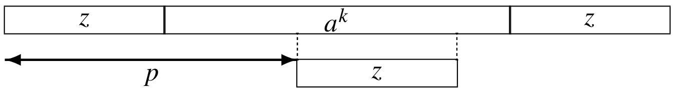
Why does Algorithm ALTERNATING work? The proof relies on the next result that is directly implied by the Unary-Extension Lemma (see Problem 113). When z is written $(uv)^e u$ and $|uv|$ is its (smallest) period, by definition $tail(z) = v$.

***Lemma* 21** *Let z be a non-unary binary word, k a positive integer and a a letter for which $a^k \neq tail(z)$. Then the word $x = za^kz$ has no period smaller than $|za^k|$, that is, has no border longer than z.*

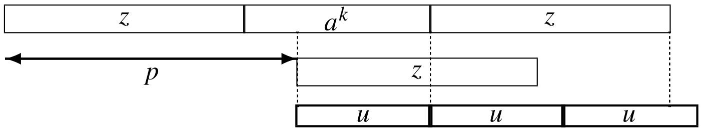
Proof The proof is by contradiction. Assume $x = za^kz$ has a period $p < |za^k|$ and consider two cases.

Case $p \le |z|$. The word x has two periods $|za^k|$ and p that satisfy $|za^k| + p \le |x|$. Applying the Periodicity Lemma to them, we deduce that $\gcd(|za^k|, p)$ is also a period of x. This implies that ua^k is non-primitive, since $\gcd(|za^k|, p) \le p < |ua^k|$. But this contradicts the Unary-Extension Lemma, whose conclusion is that ua^k is primitive under the hypothesis.

Case $|u| < p < |ua^k|$.



Inequalities of this case imply that there is an internal occurrence of z in za^kz, namely at position p. If $p \le k$, that is, $p + |z| \le |za^k|$ (see picture) this occurrence is internal to a^k, which contradicts the fact that z is non-unary.



Otherwise $p > k$, that is, $p + |z| > |za^k|$ (see picture). Then the last two occurrences of z overlap, which means that $|za^k| - p$ is a period of z. As a suffix of a^k, the prefix period $u = z[0\,..\,|za^k| - p - 1]$ of z is unary, which implies that z itself is unary, a contradiction again.
Therefore no period of za^kz is smaller than $|za^k|$. ■

To prove the correctness of Algorithm Alternating we have to show that filling the gap between occurrences of x_{i-1} with a unary word at line 7 does not generate a redundant period.

To do so, assume $q_1 > 1$. The case $q_1 = 1$ can be treated similarly starting with the first i for which x_i is non-unary. Algorithm Alternating has the following property: if $gap_i > 0$ then $x_i = x_{i-1}y_ix_{i-1}$ where $y_i = a^{gap_i} \neq tail(x_{i-1})$ and x_{i-1} is not unary. Due to the lemma $x_{i-1}y_ix_{i-1}$ has no border longer than x_{i-1}. Thus no redundant border is created, which shows the correctness of Alternating.

Computing the list of border lengths, for example, with Algorithm Borders from Problem 19, and running the algorithm takes linear time $O(N) = O(|w|)$, as expected.

Notes

The presented algorithm as well as more complicated algorithm for binary lexicographically first words are by Rytter [215].

Note that a sorted list $\mathcal{B} = (q_1, q_2, \ldots, q_n)$ corresponds to the list of border lengths of a word if and only if, for $\delta_i = q_i - q_{i-1}$ when $i > 1$,

$$\delta_{i-1} \mid \delta_i \Rightarrow \delta_{i-1} = \delta_i \text{ and } q_i + \delta_i \leq n \Rightarrow q_i + \delta_i \in \mathcal{B}.$$

This is a version of Theorem 8.1.11 in [176]. The above technique provides a compressed description of size $O(n)$, which can be of order $\log N$, of the output word of length N.

117 Online Generation of de Bruijn Words

A binary de Bruijn word of order n is a word of length 2^n on the alphabet $\{\mathtt{0},\mathtt{1}\}$ in which all binary words of length n occur cyclically exactly once. For example, 00010111 and 01000111 are (non-conjugate) de Bruijn words of order 3.

A de Bruijn word can be generated by starting from a binary word of length n and then repeating the operation NEXT(w) below. The operation computes the next bit of the sought de Bruijn word and updates the word w. The whole process stops when w returns to its initial word.

```
deBruijn($n$ positive integer)
1  $(x, w_0) \leftarrow (\varepsilon$, a binary word of length $n)$
2  $w \leftarrow w_0$
3  do  $w \leftarrow$ Next($w$)
4      $x \leftarrow x \cdot w[n-1]$
5  while $w \neq w_0$
6  return $x$
```

The operation NEXT needs only to be specified to get an appropriate on-line generation algorithm. Let $\overline{b}$ denote the negation of the bit b.

```
Next($w$ non-empty word of length $n$)
1  if $w[1\,..\,n-1] \cdot \mathtt{1}$ smallest in its conjugacy class then
2      $b \leftarrow \overline{w[0]}$
3  else $b \leftarrow w[0]$
4  return $w[1\,..\,n-1] \cdot b$
```

Question. Show that the execution of DEBRUIJN(n) generates a binary de Bruijn word of length 2^n.

Example. Let $n = 3$ and $w_0 = \mathtt{111}$. The values of w at line 4 of Algorithm DEBRUIJN are $\mathtt{11\underline{0}}, \mathtt{10\underline{1}}, \mathtt{01\underline{0}}, \mathtt{10\underline{0}}, \mathtt{00\underline{0}}, \mathtt{00\underline{1}}, \mathtt{01\underline{1}}, \mathtt{11\underline{1}}$. Underlined bits form the de Bruijn word 01000111.

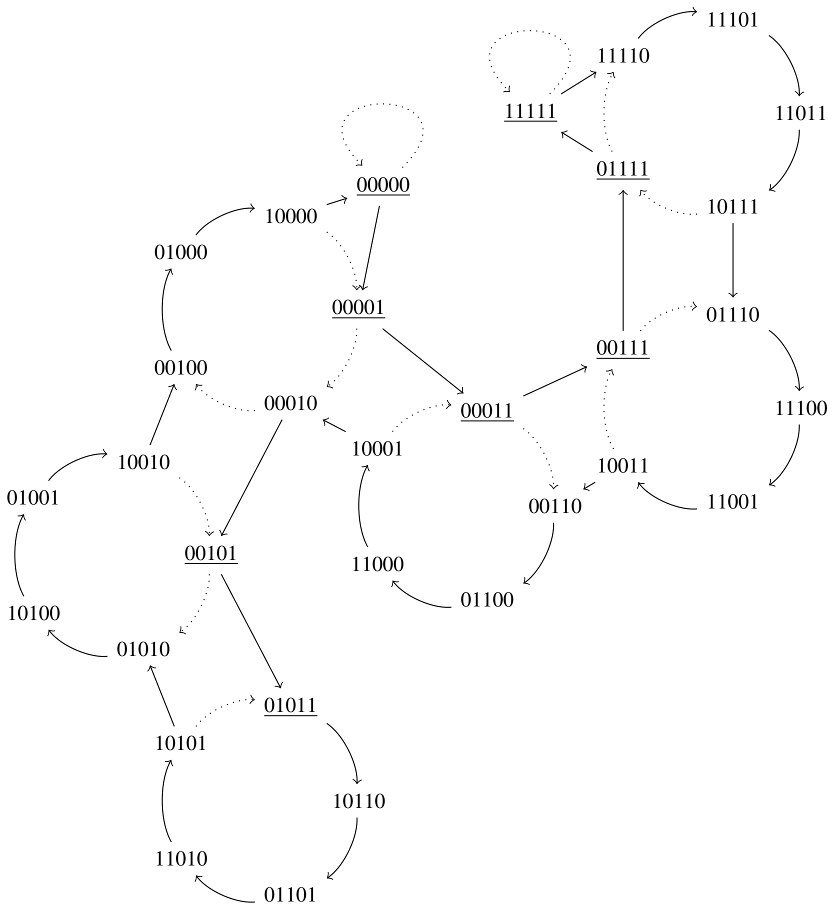
For $n = 5$ and $w_0 = \mathtt{11111}$ the consecutive values of w are
$\mathtt{1111\underline{0}}, \mathtt{1110\underline{1}}, \mathtt{1101\underline{1}}, \mathtt{1011\underline{1}}, \mathtt{0111\underline{0}}, \mathtt{1110\underline{0}}, \mathtt{1100\underline{1}}, \mathtt{1001\underline{1}}, \mathtt{0011\underline{0}}$,
$\mathtt{0110\underline{0}}, \mathtt{1100\underline{0}}, \mathtt{1000\underline{1}}, \mathtt{0001\underline{0}}, \mathtt{0010\underline{1}}, \mathtt{0101\underline{1}}, \mathtt{1011\underline{0}}, \mathtt{0110\underline{1}}, \mathtt{1101\underline{0}}$,

`1010`$\underline{\texttt{1}}$, `0101`$\underline{\texttt{0}}$, `1010`$\underline{\texttt{0}}$, `0100`$\underline{\texttt{1}}$, `1001`$\underline{\texttt{0}}$, `0010`$\underline{\texttt{0}}$, `0100`$\underline{\texttt{0}}$, `1000`$\underline{\texttt{0}}$, `0000`$\underline{\texttt{0}}$, `0000`$\underline{\texttt{1}}$, `0001`$\underline{\texttt{1}}$, `0011`$\underline{\texttt{1}}$, `0111`$\underline{\texttt{1}}$, `1111`$\underline{\texttt{1}}$, generating the de Bruijn word

`01110011000101101010010000011111.`

Solution

The correctness of Algorithm deBruijn can be proved by interpreting its run as a traversal of a tree whose nodes are shift cycles connected by two-way 'bridges'. Vertices of shift cycles are words of length n that are in the same conjugacy class. The representative of a cycle is its lexicographically minimal word (a necklace or Lyndon word if primitive). Edges in cycles stand for shifts, that is, are of the form $au \to ua$, where a is a single bit, and u is a word of length $n - 1$. Shift cycles form the graph G_n.



Graph G_n is transformed into the graph G'_n (see picture for G'_5) by adding bridges connecting disjoint cycles and by removing some cycle edges (dotted edges in picture) with the following procedure.

```
BRIDGES($G$ graph of cycles)
1  for each node $u$ of $G$ do
2      if $u\texttt{1}$ smallest in its conjugacy class then
3          remove edges $\texttt{1}u \to u\texttt{1}$ and $\texttt{0}u \to u\texttt{0}$
4          create edges $\texttt{1}u \to u\texttt{0}$ and $\texttt{0}u \to u\texttt{1}$
5  return modified graph
```

All solid edges in G'_n are associated with the function NEXT and used to traverse the graph. The graph G'_n consists of a single Hamiltonian cycle containing all words of length n. Bridges that connect a cyclic class of a word to another cyclic class of words with more occurrences of 0 form a (not rooted) tree whose nodes are cycles.

Observation. For a word w of length n, $v = \text{NEXT}(w)$ if and only if $w \to v$ is an edge of G'_n.

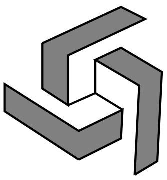
Hence the algorithm implicitly traverses the graph using the Hamiltonian cycle. This completes the proof of correctness.

Notes

The algorithm is by Sawada et al. [219]. The present proof is completely different from their. The for loop of function BRIDGES can be changed to the following:

```
1  for each node $u$ of $G$ do
2      if $\texttt{0}u$ smallest in its conjugacy class then
3          remove edges $\texttt{0}u \to u\texttt{0}$ and $\texttt{1}u \to u\texttt{1}$
4          create edges $\texttt{0}u \to u\texttt{1}$ and $\texttt{1}u \to u\texttt{0}$
```

Doing so we obtain a different graph of shift cycles connected by a new type of bridges and an alternative version of operation NEXT corresponding to the new Hamiltonian cycle.



118 Recursive Generation of de Bruijn Words

Following Problem 117, the present problem provides another method to generate a de Bruijn word on the alphabet $\mathtt{B} = \{\mathtt{0}, \mathtt{1}\}$. The method is recursive and its description requires a few preliminary definitions.

Let us first define *Shift*. When a word u occurs circularly exactly once in a word x, $|u| < |x|$, $\mathit{Shift}(x,u)$ denotes the conjugate of x admitting u as a suffix. For example, $\mathit{Shift}(\mathtt{001011101}, \mathtt{0111}) = \mathtt{010010111}$ and $\mathit{Shift}(\mathtt{001011101}, \mathtt{1010}) = \mathtt{010111010}$.

Let us then define the operation $\oplus$ between two words $x, y \in \mathtt{B}^N$, $N = 2^n$, for which the suffix u of length n of x has a unique circular factor occurrence in y: $x \oplus y$ denotes $x \cdot \mathit{Shift}(y,u)$. For example, with $n = 2$, $\mathtt{0010} \oplus \mathtt{1101} = \mathtt{0010\,1110}$ since $\mathit{Shift}(\mathtt{1101}, \mathtt{10}) = \mathtt{1110}$.

Eventually, for a binary word w, let $\Psi(w)$ be the binary word v of length $|w|$ defined, for $0 \le i \le |w| - 1$, by

$$v[i] = (w[0] + w[1] + \cdots + w[i]) \bmod 2.$$

For example, $\Psi(\mathtt{0010111011}) = \mathtt{0011010010}$. Also denote by $\overline{x}$ the complement of x, that is, its bitwise negation.

Question. Show that if x is a binary de Bruijn word of length 2^n then $\Psi(x) \oplus \overline{\Psi(x)}$ is a de Bruijn word of length 2^{n+1}.

[**Hint:** Count circular factors of $\Psi(x) \oplus \overline{\Psi(x)}$.]

Example. For the de Bruijn word $\mathtt{0011}$, we have $\Psi(\mathtt{0011}) = \mathtt{0010}$ and $\overline{\Psi(\mathtt{0011})} = \mathtt{1101}$. Then $\mathtt{0010} \oplus \mathtt{1101} = \mathtt{0010\,1110}$, which is a de Bruijn word of length 8.

Solution

Let $\mathit{Cfact}_k(z)$ denote the set of circular factors of length k of a word z. We start with the following fact.

Observation 1. If two words x and y of length N have a common suffix of length $n < N$, $\mathit{Cfact}_{n+1}(x \cdot y) = \mathit{Cfact}_{n+1}(x) \cup \mathit{Cfact}_{n+1}(y)$.

The second observation is a consequence of the first.

Observation 2. Let x and y be binary words of length $N = 2^n$. When both $\mathit{Cfact}_{n+1}(x) \cup \mathit{Cfact}_{n+1}(y) = \mathtt{B}^{n+1}$ and the suffix of length n of x belongs to $\mathit{Cfact}_n(y)$, the word $x \oplus y$ is a de Bruijn word.

Proof We can assume that x and y have the same suffix of length n because this does not change the result of the operation $\oplus$ and then we have $x \oplus y = x \cdot y$. Observation 1 implies, due to the hypothesis, that $Cfact_{n+1}(x \oplus y) = Cfact_{n+1}(x \cdot y) = Cfact_{n+1}(x) \cup Cfact_{n+1}(y) = \mathtt{B}^{n+1}$. Since $x \oplus y$ has length 2^{n+1}, every binary word of length $n+1$ occurs circularly exactly once in it. This means that $x \oplus y$ is a de Bruijn word as expected. ■

To answer the question, let x be a de Bruijn word of length 2^n. The operation Ψ satisfies the following two properties:

(i) $Cfact_{n+1}(\Psi(x)) \cap Cfact_{n+1}(\overline{\Psi(x)}) = \emptyset$;

(ii) $|Cfact_{n+1}(\Psi(x))| = 2^n$ and $|Cfact_{n+1}(\overline{\Psi(x)})| = 2^n$.

Properties (i) and (ii) imply that the words $\Psi(x)$ and $\overline{\Psi(x)}$ satisfy the assumptions of Observation 2. Consequently $\Psi(x) \oplus \overline{\Psi(x)}$ is a de Bruijn binary word of length 2^{n+1}.

Notes

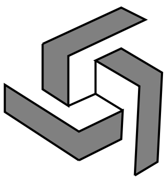
The recursive approach used to build de Bruijn words is from [206]. It is also a syntactic version of Lempel's algorithm that uses a special type of graph homomorphism, see [174].

It is an example of a simple application of algebraic methods in text algorithms. A more advanced application of algebraic methods is the generation of de Bruijn words based on so-called *linear shift registers* and related *primitive polynomials*, see [131].

The algorithm has a surprising graph-theoretic property. Assume we start with $w_2 = \mathtt{0011}$ and define, for $n \geq 3$,

$$w_n = \Psi(w_{n-1}) \oplus \overline{\Psi(w_{n-1})}.$$

Then, in the de Bruijn graph G_{n+1} of order $n+1$ having 2^n nodes, w_n corresponds to a Hamiltonian cycle C. After removing C and disregarding two single-node loops, the graph G_{n+1} becomes a big single simple cycle of length $2^n - 2$.

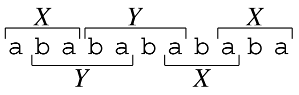


119 Word Equations with Given Lengths of Variables

A word equation is an equation between words whose letters are constants or variables. Constants belong to the alphabet $\mathtt{A} = \{\mathtt{a}, \mathtt{b}, \ldots\}$ and variables belong to the disjoint alphabet of unknowns $\mathtt{U} = \{X, Y, \ldots\}$. An equation is written $L = R$, where $L, R \in (\mathtt{A} \cup \mathtt{U})^*$ and a solution of it is a morphism $\psi : (\mathtt{A} \cup \mathtt{U})^* \to \mathtt{A}^*$ leaving constant invariant and for which the equality $\psi(L) = \psi(R)$ holds.

In the problem we assume that the length $|\psi(X)|$ is given for each variable X occurring in the equation and is denoted by $|X|$.

For example, $XY\mathtt{b}X = \mathtt{a}Y\mathtt{b}X\mathtt{ba}$ with $|X| = 3$ and $|Y| = 4$ admits a (unique) solution ψ defined by $\psi(X) = \mathtt{aba}$ and $\psi(Y) = \mathtt{baba}$:



On the contrary, the equation $\mathtt{a}XY = Y\mathtt{b}X$ has no solution, because $\mathtt{a}\psi(X)$ and $\mathtt{b}\psi(X)$ must be conjugate, which is incompatible with both $|\mathtt{a}\psi(X)|_{\mathtt{a}} = 1 + |\psi(X)|_{\mathtt{a}}$ and $|\mathtt{b}\psi(X)|_{\mathtt{a}} = |\psi(X)|_{\mathtt{a}}$, while the equation $\mathtt{a}XY = Y\mathtt{a}X =$ has $\mathtt{A}^{|X|}$ solutions when $|X| = |Y| - 1$.

Question. Given a word equation with the variable lengths, show how to check in linear time with respect to the equation length plus the output length if a solution exists.

Solution

Let ψ be a potential solution of the equation $L = R$. If $|\psi(L)| \neq |\psi(R)|$ according to the given variable lengths the equation has obviously no solution. We then consider that variable lengths are consistent and set $n = |\psi(L)| = |\psi(R)|$.

Let $G = (V, E)$ be the undirected graph defined by

- $V = \{0, 1, \ldots, n-1\}$, set of positions on $x = \psi(L) = \psi(R)$.
- E set of edges (i, j) where i and j correspond to the same relative position on two occurrences of $\psi(X)$ in $\psi(L)$ or in $\psi(R)$, for some variable X. For example, i and j can be first positions of occurrences.

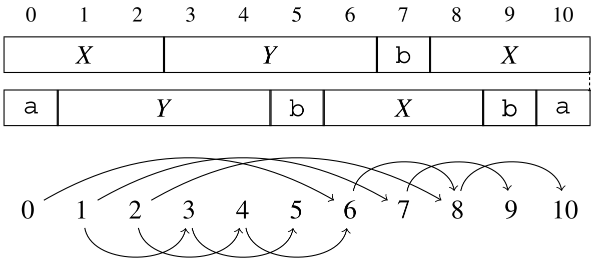
To build the graph, the list of positions on x covered by an occurrence of $\psi(X)$ in $\psi(L)$ or in $\psi(R)$ can be precomputed.

We say that two positions are *conflicting* if they index two distinct constants.

Observation. The equation is solvable if and only if there is no conflicting position in the same connected component of G.

Quadratic-time algorithm. After the graph is computed, its connected components are built and the condition on conflicting positions is checked. Using a standard efficient algorithm for computing the connected components the overall takes $O(n^2)$ time because the size of the graph is $O(n^2)$.

Example. Alignment and graph associated with the above equation $XY\texttt{b}X = \texttt{a}Y\texttt{b}X\texttt{ba}$ with $|X| = 3$ and $|Y| = 4$. The graph has two connected components $\{0,2,4,6,8,10\}$ and $\{1,3,5,7,9\}$. Positions 0 and 10 correspond to letter a and positions 5, 7 and 9 to letter b in the equation. There is no conflicting position and only one solution, producing the word ababababababa.



Linear-time algorithm. The algorithm is accelerated by reducing the number of edges of G in a simple way. It is enough to consider edges (i,j), $i < j$, where i and j are positions of consecutive occurrences of $\psi(X)$ in $\psi(L)$ or in $\psi(R)$ (see picture), possibly merging two lists. The connected components of the new graph satisfy the observation and the algorithm now runs in linear time because the size of the graph is linear, with at most two outgoing edges starting from each position.

Notes

When the lengths associated with variables are not given, the problem has been shown to be decidable by Makanin [181]. The problem is known to be NP-hard, but the big open question is its membership to the class of NP problems.

The fastest known algorithms work in exponential time (see [176, chapter 12] and references therein). If we knew that the shortest solution is of (only) single-exponential length then there is a simple NP algorithm to solve the problem. There is no known example of an equation for which a shortest solution is longer than a single exponential, but it is an open problem to prove it is always true.

120 Diverse Factors over a Three-Letter Alphabet

A word w is called diverse if the numbers of occurrences of its letters are pairwise different (some letters may be absent in w). The problem deals with diverse factors occurring in a word $x \in \{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$.

Example. The word `aab` is diverse but `aa` and the empty word are not. The word `abbccc` itself is diverse but the word `abcabcabc` has no diverse factor. The longest diverse factor of `cbaabacccbba` is `cbaabaccc`.

Obviously any word of length at most 2 has no diverse factor and a word of length 3 is not diverse if it is a permutation of the three letters. The straightforward observation follows.

Observation 1. The word $x \in \{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$, $|x| \geq 3$, has no diverse factor if and only if its prefix of length 3 is not diverse, that is, is a permutation of the 3 letters, and is a word period of x.

Question. Design a linear-time algorithm finding a longest diverse factor of a word x over a three-letter alphabet.

[**Hint:** Consider the Key property proved below.]

Key property. If the word $x[0\,..\,n-1] \in \{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$ has a diverse factor, it has a longest diverse factor $w = x[i\,..\,j]$ for which either $0 \leq i < 3$ or $n-3 \leq j < n$, that is,

$$w \in \bigcup_{i=0}^{2} \mathit{Pref}(x[i\,..\,n-1]) \cup \bigcup_{j=n-3}^{n-1} \mathit{Suff}(x[0\,..\,j]).$$

Solution

Since testing the condition in Observation 1 takes linear time it remains to consider the case where the word x contains a diverse factor. Before discussing the algorithm, we start with a proof of the Key property.

The proof of the property is by contradiction. Assume x has a longest diverse factor w for which $x = uwv$ with both $|u| \geq 3$ and $|v| \geq 3$. In other words x has a factor of the form $abcwdef$ for letters a, b, c, d, e and f. We consider all cases corresponding to the occurrence numbers of letters in w and in the neighbouring three positions of w, to the left and to the right in x, and assume w.l.o.g. that

$$|w|_{\mathtt{a}} < |w|_{\mathtt{b}} < |w|_{\mathtt{c}}.$$

The following observation limits considerably the number of cases to examine.

Observation 2. If w is a longest diverse factor of $abcwdef$, where a, b, c, d, e and f are single letters and $|w|_{\mathtt{a}} < |w|_{\mathtt{b}} < |w|_{\mathtt{c}}$, then

$$\mathtt{c} \notin \{c,d\} \text{ and } |w|_{\mathtt{a}} + 1 = |w|_{\mathtt{b}} = |w|_{\mathtt{c}} - 1.$$

Unfortunately there are still several cases to consider for letters a to f, but all of them lead to a contradiction with the non-extendability of the diverse factor w in the local window $abcwdef$. Eventually the question reduces to six cases whose proofs are left to the reader.

As a consequence of the Key property, the problem amounts to several applications of a simpler version: the diverse prefix and suffix problems, that is, compute a longest diverse prefix and a longest diverse suffix of a word. They are to be computed respectively for the three suffixes $x[i\,..\,n-1]$, $0 \le i < 3$, and for the three prefixes $x[0\,..\,j]$, $n-3 \le j < n$ of x to get the result.

Linear-time solution for the longest diverse prefix. We describe only the computation of a longest diverse prefix of x, since the other cases are either similar or symmetric.

Example. The longest diverse prefix of $y =$ `cbaabacccbba` is `cbaabaccc` as shown on the table at index 8. In fact it can be checked that it is the longest diverse factor of y.

i		0	1	2	3	4	5	6	7	8	9	10	11
$y[i]$		c	b	a	a	b	a	c	c	c	b	b	a
$\|y\|_{\mathtt{a}}$	0	0	0	1	2	2	3	3	3	**3**	3	3	4
$\|y\|_{\mathtt{b}}$	0	0	1	1	1	2	2	2	2	**2**	3	4	4
$\|y\|_{\mathtt{c}}$	0	1	1	1	1	1	1	2	3	**4**	4	4	4

The computation to find a longest diverse prefix of x is done on-line on x. The occurrence numbers of the three letters are computed in consecutive prefixes. The largest index where the vector has pairwise different numbers provides the sought prefix.

Notes

Note that a longest diverse factor can be much shorter than the word, like `ccbaaaaaa` in `aaaabccbaaaaaa`, and that the boundary distance 3 in the Key property cannot be reduced to 2: a counterexample is the word `abcacbacba` whose longest diverse factor is `cacbac`.

The problem appeared in the 25th Polish Olympiad in Informatics under the name ‘Three towers’.

121 Longest Increasing Subsequence

In this problem we consider a word x on the alphabet of positive integers. An increasing subsequence of x is a subsequence $x[i_0]x[i_1]\cdots x[i_{\ell-1}]$, where $0 \le i_0 < i_1 < \cdots < i_{\ell-1} < |x|$ and $x[i_0] \le x[i_1] \le \cdots \le x[i_{\ell-1}]$.

Example. Let x = `3 6 4 10 1 15 13 4 19 16 10`. A longest increasing subsequence of x is y = `3 4 10 13 16` of length 5. Another such subsequence is z = `3 4 10 13 19`. If `21` is appended to x this lengthen y and z to increasing subsequences of length 6. But if 18 is appended to x only y `18` becomes a longest subsequence, since z `18` is not increasing.

Question. Show that Algorithm Lis computes in place the maximal length of an increasing subsequence of a word x in time $O(|x| \log |x|)$.

```
Lis(x non-empty word over positive integers)
ℓ ← 1
for i ← 1 to |x| − 1 do
    (a, x[i]) ← (x[i], ∞)
    k ← min{j : 0 ≤ j ≤ ℓ and a < x|j]}
    x[k] ← a
    if k = ℓ then
        ℓ ← ℓ + 1
return ℓ
```

Example followed. The tables display x before and after a run of Algorithm Lis.

i	0	1	2	3	4	5	6	7	8	9	10
$x[i]$	3	6	4	10	1	15	13	4	19	16	10

i	0	1	2	3	4	5	6	7	8	9	10
$x[i]$	1	4	4	10	16	∞	∞	∞	∞	∞	∞

Inspecting carefully Algorithm Lis, it should be noticed that the word $x[0\,.\,.\,\ell-1]$ computed when the algorithm terminates is increasing but not usually a subsequence of x, as shown by the example, which leads to the next question.

Question. Design an algorithm that computes a longest increasing subsequence of a word x in time $O(|x| \log |x|)$.

Solution

Complexity. Note that values stored in the prefix $x[0 \mathinner{..} \ell - 1]$ of x satisfy $x[0] \leq x[1] \leq \cdots \leq x[\ell - 1]$ and are followed by ∞. (They can be different from the initial values in $x[0 \mathinner{..} \ell - 1]$.) Thus, the instruction at line 4 can be implemented to run in $O(\log |x|)$ time and in fact in $O(\log \ell)$ if ℓ is the length of a longest increasing subsequence of x. This amount to a total of $O(|x| \log |x|)$ or $O(|x| \log \ell)$ running time. It is clear that the required memory space in addition to the input is constant.

Correctness. The correctness of Algorithm LIS relies on this invariant of the for loop: for any j, $0 \leq j < \ell$, $x[j]$ is the smallest value, called $best[j]$, that can end an increasing subsequence of length j in $x[0 \mathinner{..} i]$.

This obviously holds at start with $j = 0$ and $x[0]$ unchanged. The effect of lines 4-7 is either to decrease $best[k]$ or to enlarge the previously computed longest increasing subsequence.

Longest Increasing Subsequence. Algorithm LIS can be upgraded to compute it. Values stored in $x[0 \mathinner{..} \ell]$ or rather their positions on x can be kept in a separate array with their predecessors inside an increasing subsequence. The main instruction to manage the new array, after initialisation, is to set the predecessor of the current element $x[i]$, when added to the array, to the predecessor of the element it replaces. When the algorithm terminates, a longest increasing subsequence is retrieved by traversing the predecessor links from the largest/rightmost value in the array.

Below is the predecessor array defined on indices for the above example, with which the two longest increasing subsequences are retrieved by tracing them back from indices 9 or 8.

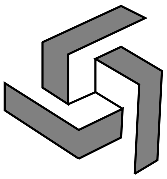
j	0	1	2	3	4	5	6	7	8	9	10
$pred[j]$	–	0	0	2	–	3	3	2	6	6	3

Notes

Observe that the algorithm solves a dual problem, and computes the smallest number of disjoint strictly increasing subsequences into which a given word can be split.

Computing a longest increasing subsequence is a textbook example of dynamic programming (see for example [226]).

When the word is composed of a permutation of the nth smallest positive integers the question is related to the representation of permutations with Young tableaux; see the chapter by Lascoux, Leclerc and Thibon in [176, Chapter 5] for a presentation of Schensted's algorithm in this context. In this case, the running time of the computation can be reduced to $O(n \log\log n)$ (see [241]) and even to $O(n \log\log \ell)$ (see [95] and references therein).



122 Unavoidable Sets via Lyndon Words

Lyndon words often surprisingly appear in seemingly unrelated problems. We show that they appear as basic components in certain decompositions, which are the main tool in the problem.

The problem deals with unavoidable sets of words. A set $X \subseteq \{\mathtt{0},\mathtt{1}\}^*$ is unavoidable if any infinite binary word has a factor in X.

To start with, let $\mathcal{N}_k$ be the set of necklaces of length k, $k > 0$. A necklace is the lexicographically smallest word in its conjugacy class. Each necklace is a power of a Lyndon word.

Example. The set $\mathcal{N}_3 = \{\mathtt{000},\mathtt{001},\mathtt{011},\mathtt{111}\}$ is avoidable, since $(01)^\infty$ has no factor in $\mathcal{N}_3$. But after moving the last $\mathtt{1}$ to the beginning we get the set $\{\mathtt{000},\mathtt{100},\mathtt{101},\mathtt{111}\}$ that is unavoidable.

Observation 1. If $X \subseteq \{\mathtt{0},\mathtt{1}\}^k$ is unavoidable then $|X| \geq |\mathcal{N}_k|$.

Indeed, for each $w \in \{\mathtt{0},\mathtt{1}\}^k$, length-$k$ factors of w^∞ are all conjugates of w and at least one of them should be in the unavoidable set X. The first question provides a more flexible condition to be unavoidable.

Question. Show that if for each necklace $y \in \{\mathtt{0},\mathtt{1}\}^*$, $|y| \geq 2k$, the word y^2 contains a word in $X \subseteq \{\mathtt{0},\mathtt{1}\}^k$ then X is unavoidable.

An ingenious construction of an unavoidable set is based on the notion of pre-primes. A pre-prime w is a prefix of a necklace.

Observation 2. A pre-prime is a prefix of a power of a Lyndon word.

The observation justifies the special decomposition of a pre-prime w as $u^e v$, where u is a Lyndon word, $e \geq 1$, $|v| < |u|$ and u is the shortest possible. Head and tail of w are defined by $\mathit{head}(w) = u^e$ and $\mathit{tail}(w) = v$.

Example. $w =$ `0101110` factorises as $\texttt{01}^2 \cdot \texttt{110}$, $\texttt{01011} \cdot \texttt{10}$ and $\texttt{010111} \cdot \texttt{0}$ over its Lyndon prefixes `01`, `01011` and `010111` respectively. Its special decomposition is $\texttt{01011} \cdot \texttt{10}$, $\mathit{head}(w) = \texttt{01011}$ and $\mathit{tail}(w) = \texttt{10}$.

Note that v is not necessarily a proper prefix of u, that is, $|u|$ is not usually the period of w. It is clear that such factorisation of a pre-prime always exists. The factorisation is the key-concept in this problem.

Question. Show that there is an unavoidable set $X_k \subseteq \{0,1\}^k$ of size $|\mathcal{N}_k|$. Consequently, $|\mathcal{N}_k|$ is the smallest size of such a set.

[**Hint:** Consider words of the form $\mathit{tail}(w) \cdot \mathit{head}(w)$.]

Solution

We first prove the statement in the first question, which provides a restricted condition for a subset of $\{0,1\}^k$ to be unavoidable, getting rid of infinity.

By contradiction, assume there is an infinite word x having no factor in $X \subseteq \{0,1\}^k$. Consider a word u with two non-overlapping occurrences in x, so that uvu is a factor of x and $|u|, |v| \geq k$. Let y be the necklace conjugate of uv. The hypothesis implies that there is a word $w \in X$ factor of y^2; however, due to the inequalities $|u|, |v| \geq k$ this word also occurs in uvu and thus in x. This contradicts the assumption that x does not contain any word from X and completes the proof.

A smallest unavoidable set. The sought unavoidable set is

$$X_k = \{\mathit{tail}(w) \cdot \mathit{head}(w) : w \in \mathcal{N}_k\}.$$

For example $X_4 = \{\texttt{0000}, \texttt{0001}, \texttt{1001}, \texttt{0101}, \texttt{1011}, \texttt{1111}\}$ and X_7 contains the 20 words (tails are underlined):

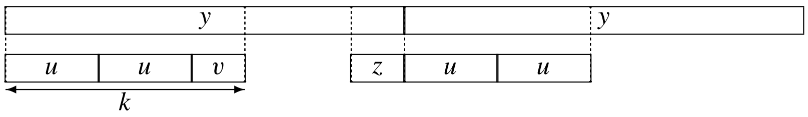
$$\begin{array}{l}
\texttt{0000000}, \texttt{0000001}, \underline{\texttt{1}}\texttt{000001}, \underline{\texttt{01}}\texttt{00001}, \underline{\texttt{11}}\texttt{00001}, \underline{\texttt{001}}\texttt{0001}, \underline{\texttt{011}}\texttt{0001},\\
\underline{\texttt{101}}\texttt{0001}, \underline{\texttt{111}}\texttt{0001}, \underline{\texttt{1}}\texttt{001001}, \underline{\texttt{01}}\texttt{00101}, \underline{\texttt{11}}\texttt{00101}, \underline{\texttt{011}}\texttt{0011}, \underline{\texttt{101}}\texttt{0011},\\
\underline{\texttt{111}}\texttt{0011}, \underline{\texttt{1}}\texttt{010101}, \underline{\texttt{11}}\texttt{01011}, \underline{\texttt{1}}\texttt{011011}, \underline{\texttt{111}}\texttt{0111}, \texttt{1111111}.
\end{array}$$

Before proving X_k is an answer to the second question, we state a useful property of necklaces and special decompositions; its technical proof is left to the reader (see Notes).

Observation 2. Let $w \in \{0,1\}^k$ be a pre-prime prefix of a necklace y of length at least $2k$ and with decomposition $u^e \cdot v$. Let z be the suffix of y of length $|v|$. Then $u^e \cdot z$ is a necklace with decomposition $u^e \cdot z$.

Theorem. The set X_k is unavoidable.

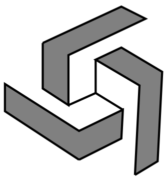
Proof Due to the first question it is enough to show that for each necklace y of size at least $2k$ the word y^2 has a factor in X_k. Let us fix any such y and let $u^e \cdot v$ be the decomposition of the pre-prime, prefix of length k of y. The goal is to find a factor of y^2 that belongs to X_k.



Let z be the suffix of length $|v|$ of y. According to Observation 2 the word $w = u^e z$ is a necklace and $u^e \cdot z$ is its decomposition. Hence $z \cdot u^e \in X_k$. Since z is a suffix of y and u^e is a prefix of y, zu^e is a factor of y^2 (see picture). This completes the proof. ■

Notes

The solution of the problem is by Champarnaud et al. [53]. Testing if a word is a pre-prime is addressed in Problem 42.

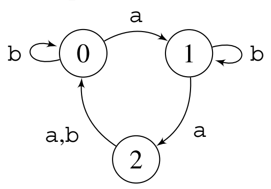


123 Synchronising Words

The problem deals with the synchronisation of the composition of functions from $I_n = \{0, 1, \ldots, n-1\}$ to itself for a positive integer n. For two such functions $f_{\texttt{a}}$ and $f_{\texttt{b}}$, a word $w \in \{\texttt{a},\texttt{b}\}^+$ encodes their composition $f_w = h_0 \circ h_1 \circ \cdots \circ h_{|w|-1}$ when $h_i = f_{\texttt{a}}$ if $w[i] = \texttt{a}$ and $h_i = f_{\texttt{b}}$ if $w[i] = \texttt{b}$. (Note functions h_i are applied to elements of I_n in decreasing order of i as usual, i.e., from right to left according to w.) The word w is said to be *synchronising* if the set $f_w(I_n)$ contains a single element.

A useful notion is that of a pair synchroniser: a word $u \in \{\texttt{a},\texttt{b}\}^*$ is a synchroniser of the pair (i, j), $i, j \in I_n$, if $f_u(i) = f_u(j)$.

Example. Consider the two functions: $g_{\texttt{a}}(i) = (i + 1) \bmod n$, $g_{\texttt{b}}(i) = \min\{i, g_{\texttt{a}}(i)\}$. For $n = 3$ they are illustrated by the automaton. As shown on the table below the word $w = \texttt{baab}$ is synchronising since the image of the set $\{0, 1, 2\}$ by g_w is the singleton $\{0\}$. The word obviously synchronises every pair but the table shows additional synchonisers like $\texttt{b}$ for the pair $(0, 2)$, $\texttt{baa}$ for the pair $(0, 1)$ and $\texttt{ba}$ for the pair $(1, 2)$.



w			b		a		a		b	
$g_w(0)$	=	0	←	2	←	1	←	0	←	0
$g_w(1)$	=	0	←	0	←	2	←	1	←	1
$g_w(2)$	=	0	←	2	←	1	←	0	←	2

For any positive integer n the word $w = \texttt{b}(\texttt{a}^{n-1}\texttt{b})^{n-2}$ is a synchronising word of the functions $g_{\texttt{a}}$ and $g_{\texttt{b}}$. It is more difficult to see that it is a shortest such word in this particular case. This yields a quadratic lower bound on the length of a synchronising words.

Question. Show that a pair of functions admits a synchronising word if and only if there exists a synchroniser for each pair (i, j) of elements of I_n.

[**Hint:** Compute a synchronising word from synchronisers.]

Question. Show how to check in quadratic time if a pair of functions admits a synchronising word.

[**Hint:** Check pair synchronisers.]

Solution

The ‘only if’ part of the statement in the first question is obvious because a synchronising word is a synchroniser of every pair of elements of I_n. Then we just have to prove the ‘if’ part, that is, show that a synchronising word exists when there is a synchroniser for each pair (i, j) of elements $i, j \in I_n$. Algorithm SyncWord constructs a global synchronising word.

```
SyncWord($f_{\mathtt{a}}, f_{\mathtt{b}}$ functions from $I_n$ to itself)
1  $J \leftarrow I_n$
2  $w \leftarrow \varepsilon$
3  while $|J| > 1$ do
4      $i, j \leftarrow$ any two distinct elements from $J$
5      $u \leftarrow$ a synchroniser of $(i, j)$
6      $J \leftarrow f_u(J)$
7      $w \leftarrow u \cdot w$
8  return $w$
```

Existence of pair synchronisers. To answer the second question, we need to check whether each pair of elements has a synchroniser or not.

To do so, let G be the directed graph whose nodes are pairs (i, j), $i, j \in I_n$, and whose edges of the form $(i, j) \to (p, q)$ are such that $p = f_a(i)$ and $q = f_a(j)$, for a letter $a \in \{\mathtt{a}, \mathtt{b}\}$.

Then the pair (i, j) has a synchroniser if and only if there is a path from (i, j) to a node of the form (p, p). Checking the property is done by a standard algorithm for traversing the graph.

If some pair has no synchroniser, the functions have no synchronising word by the first statement. Otherwise there is a synchronising word.

The running time for processing the graph is $O(n^2)$, as required. But running Algorithm SyncWord to get a synchronising word takes cubic time provided operations on words are done in constant time.

Notes

When the two functions are letters acting on the set of states of a finite automaton, a synchronising word is also called a reset word; see, for example, [29].

Although the above method works in quadratic time (it is enough to test the existence of local synchronisers) the actual generation of a synchronising word could take cubic time. This is due to the fact that the length of the generated word can be cubic. The so-called Cerny’s conjecture states that the upper bound on a synchronising word is only quadratic, but the best known upper bound is only $\frac{114}{685}n^3 + O(n^2)$ (improving on the best previous bound of $\frac{114}{684}n^3 + O(n^2)$); see [229] and references therein.

124 Safe-Opening Words

The problem addresses a special non-deterministic version of synchronising words. We are given a graph G with edges labelled by symbols and with a unique sink node s on which all edges loop. We are to find a synchronising word $\mathcal{S}$ for which each non-deterministically chosen path labelled by $\mathcal{S}$ goes to s independently of the starting node.

The problem is generally difficult but we consider a very special case called *safe-opening words*. It shows some surprising operations on words over the alphabet $\mathtt{B}_n = \{\mathtt{0},\mathtt{1}\}^n$.

Narrative description of the problem. The door of a *rotating safe* has a circular lock, which has n indistinguishable buttons on its circumference with equal spacing. Each button is linked to a switch on the other side of the door, invisible from outside. A switch is in state `0` (off) or `1` (on). In each move, you are allowed to press several buttons simultaneously. If all switches are turned on as a result, the safe door opens and remains open. Immediately before each move the circular lock rotates to a random position, without changing the on/off status of each individual switch. The initial configuration is unknown.

The goal is to find a sequence called a *safe-opening word*

$$\mathcal{S}(n) = A_1 \cdot A_2 \cdots A_{2^n-1}$$

of moves $A_i \in \mathtt{B}_n$ having a prefix that opens the safe.

Assuming button positions are numbered 1 to n from the top position clockwise, a move is described by an n-bit word $b_1 b_2 \cdots b_n$ with the meaning that button at position i is pressed if and only if $b_i = 1$. Positions are fixed though buttons can move, that is, change positions.

Example. It can be checked that the unique shortest safe-opening word for 2 buttons is $\mathcal{S}(2) = \mathtt{11} \cdot \mathtt{01} \cdot \mathtt{11}$.

Question. Let n be a power of two. Construct a safe-opening word $S(n)$ of length $2^n - 1$ over the alphabet $\mathtt{B}_n$.

Abstract description of the problem. Each move A_i is treated as a binary word of length n. Let $\equiv$ be the conjugacy (cyclic shift) equivalence. Let $G_n = (V, E)$ be the directed graph in which V is the set of binary words of length n, configurations of the circular lock, and edges are of the form, for $A \in \mathtt{B}_n$:

$$u \xrightarrow{A} (v \text{ xor } A)$$

for each $v \equiv u$ if $u \neq \mathtt{1}^n$, and $1^n \xrightarrow{A} 1^n$ otherwise.

Example. For $u = \texttt{0111}$ and $A = \texttt{1010}$ there are four nodes v conjugate of u, $\texttt{0111}$, $\texttt{1110}$, $\texttt{1101}$, $\texttt{1011}$, and consequently edges:

$$u \xrightarrow{A} \texttt{1101},\ u \xrightarrow{A} \texttt{0100},\ u \xrightarrow{A} \texttt{0111},\ u \xrightarrow{A} \texttt{0001}.$$

The aim is to find a word $\mathcal{S} = A_1 \cdot A_2 \cdots A_{2^n-1}$ in $\mathtt{B}_n^*$ for which each non-deterministically chosen path in G_n labelled by $\mathcal{S}$ leads to the sink $\mathtt{1}^n$ independently of the starting node.

Solution

Two operations on words $X, Y \in \mathtt{B}_n^*$ are defined to state recurrence relations:

$$X \odot Y = X \cdot Y[0] \cdot X \cdot Y[1] \cdot X \cdots X \cdot Y[N-1] \cdot X$$

is a word in $\mathtt{B}_n^*$ and, when $|X| = |Y| = N$, $X \otimes Y \in \Sigma_{2n}^*$:

$$X \otimes Y = X[0]Y[0] \cdot X[1]Y[1] \cdots X[N-1]Y[N-1].$$

For example, $(\texttt{01} \cdot \texttt{11} \cdot \texttt{10}) \otimes (\texttt{10} \cdot \texttt{11} \cdot \texttt{00}) = \texttt{0110} \cdot \texttt{1111} \cdot \texttt{1000}$.

Recurrence relations. Let $Z(n) = (\mathtt{0}^n)^{2^n-1}$, word of length $2^n - 1$ over $\mathtt{B}_n$. Let $\mathcal{S}(n)$ be a safe-opening word for $n \geq 2$. Then $\mathcal{S}(2n)$ can be computed as follows:

(i) $\mathcal{B}(n) = \mathcal{S}(n) \otimes \mathcal{S}(n)$, $\mathcal{C}(n) = Z(n) \otimes \mathcal{S}(n)$.

(ii) $\mathcal{S}(2n) = \mathcal{B}(n) \odot \mathcal{C}(n)$.

Example. Knowing that $\mathcal{S}(2) = \texttt{11} \cdot \texttt{01} \cdot \texttt{11}$, we get

$\mathcal{B}(2) = \texttt{1111} \cdot \texttt{0101} \cdot \texttt{1111}$,
$\mathcal{C}(2) = \texttt{0011} \cdot \texttt{0001} \cdot \texttt{0011}$ and
$\mathcal{S}(4) = \texttt{1111} \cdot \texttt{0101} \cdot \texttt{1111} \cdot \mathbf{0011} \cdot \texttt{1111} \cdot \texttt{0101} \cdot \texttt{1111} \cdot \mathbf{0001} \cdot \texttt{1111} \cdot \texttt{0101} \cdot \texttt{1111}$
$\cdot\, \mathbf{0011} \cdot \texttt{1111} \cdot \texttt{0101} \cdot \texttt{1111}$.

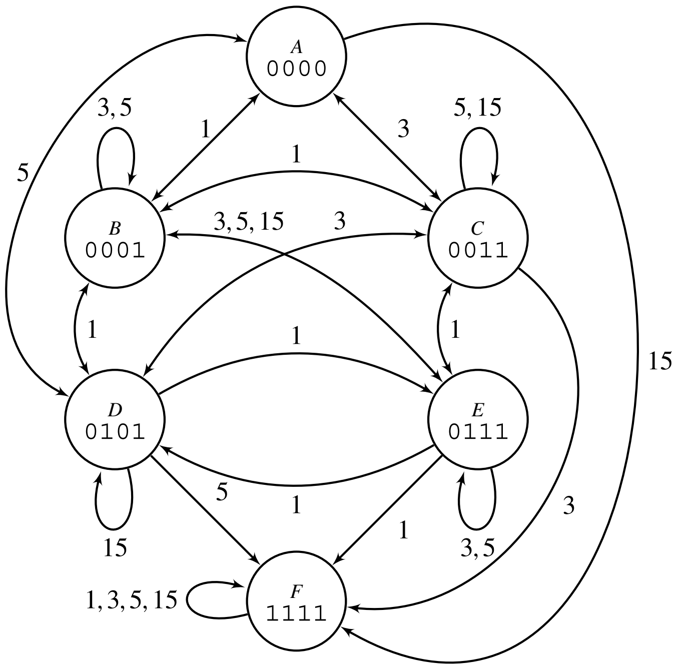
Claim. If n is a power of 2 the recurrence from Equation (ii) correctly generates safe-opening words.

Proof The word $\mathcal{B}(n)$ treats exactly in the same way buttons whose positions are opposite on the cycle. In other words, buttons at positions i and $i + n/2$ are both pressed or both non-pressed at the same time. Hence at some moment the word $\mathcal{B}(n)$ achieves the required configuration, if it starts from the configuration in which for each pair $(i, i + n/2)$ the corresponding buttons are synchronised, that is, in the same state.

This is precisely the role of the word $\mathcal{C}(n)$. After executing its prefix $C_1 \cdot C_2 \cdots C_i$, for some i, all pairs of opposite buttons are synchronised.

Then the forthcoming application of the whole word $\mathcal{B}(n)$ opens the safe, as required. ■

Illustration on the compacted graph. The alphabet $\mathtt{B}_n$ has exponential size but in the solution only its small part, denoted by $\mathtt{B}'_n$, is used. Instead of G_n let us consider its compacted version G'_n in which nodes are (smallest) representatives of conjugacy classes and edges have labels from $\mathtt{B}'_n$ only. The figure shows G'_4, in which letters `0001`, `0011`, `0101`, `1111` of $\mathtt{B}'_4$ are abbreviated as 1, 3, 5, 15, and nodes `0000`, `0001`, `0011`, `0101`, `0111`, `1111`, necklaces representing conjugacy classes, are abbreviated as A, B, C, D, E, F respectively.



Observe that, independently of the starting node in G'_4, every path labelled with 15, 5, 15, 3, 15, 5, 15, 1, 15, 5, 15, 3, 15, 5, 15 leads to `1111`.

The correctness of this sequence can be shown by starting from the whole set of nodes and applying consecutive transitions. At the end we should get the set $\{F\}$. Indeed we have

$$\{A,B,C,D,E,F\} \xrightarrow{15} \{B,C,D,E,F\} \xrightarrow{5} \{A,B,C,E,F\} \xrightarrow{15}$$
$$\{B,C,E,F\} \xrightarrow{3} \{A,B,E,D,F\} \xrightarrow{15} \{B,E,D,F\} \xrightarrow{5} \{A,B,E,F\}$$
$$\xrightarrow{15} \{B,E,F\} \xrightarrow{1} \{A,D,C,F\} \xrightarrow{15} \{D,C,F\} \xrightarrow{5} \{A,C,F\}$$
$$\xrightarrow{15} \{C,F\} \xrightarrow{3} \{A,D,F\} \xrightarrow{15} \{D,F\} \xrightarrow{5} \{A,F\} \xrightarrow{15} \{F\}.$$

Notes

The content of the problem is adapted from its original version by Guo at https://www.austms.org.au/Publ/Gazette/2013/Mar13/Puzzle.pdf. The length of safe-opening words is not addressed but it is shown that there is no such word if n is not a power of 2.

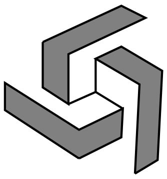
There is an alternative description of the safe-opening sequence. Assume binary words are represented as non-negative integers in a standard way. Then $\mathcal{S}(2) = 3 \cdot 1 \cdot 3$ and

$$\mathcal{S}(4) = 15 \cdot 5 \cdot 15 \cdot \mathbf{3} \cdot 15 \cdot 5 \cdot 15 \cdot \mathbf{1} \cdot 15 \cdot 5 \cdot 15 \cdot \mathbf{3} \cdot 15 \cdot 5 \cdot 15.$$

The recurrence equations (i) and (ii) now look much shorter:

$$\mathcal{S}(2n) = (\, 2^n \times \mathcal{S}(n) + \mathcal{S}(n)\,) \odot \mathcal{S}(n),$$

where the operations $+, \times$ are here component-wise arithmetic operations on sequences of integers.



125 Superwords of Shortened Permutations

On the alphabet of natural numbers, a word is an n-permutation if every number from $\{1, 2, \ldots, n\}$ appears exactly once in it (see Problems 14 and 15). A word is a shortened n-permutation (n-shortperm, in short) if it is an n-permutation with its last element removed. The bijection between standard permutations and shortened ones for a given n implies there are $n!$ shortened n-permutations.

The subject of the problem is the construction of a shortest superword for all n-shortperms. They are of length $n! + n - 2$, which meets the obvious lower bound. For example, `3213123` is a shortest superword for 3-shortperms since it contains all shortened 3-permutations

`32`, `21`, `13`, `31`, `12` and `23`.

Question. Show how to construct a superword of length $n!+n-2$ for shortened n-permutations in linear time w.r.t. the output length.

[**Hint:** Consider an Eulerian cycle in an appropriate graph.]

Solution

The problem reduces to finding an Eulerian cycle in a directed graph $\mathcal{J}_n$ (Jackson graph) that is very similar to a de Bruijn graph. The set V_n of nodes of $\mathcal{J}_n$ consists of all words that are $(n-2)$-combinations of elements in $\{1,2,\ldots,n\}$. For each $w = a_1a_2\ldots a_{n-2} \in V_n$ there are two outgoing edges:

$$a_1a_2\cdots a_{n-2} \xrightarrow{b} a_2\cdots a_{n-2}b,$$

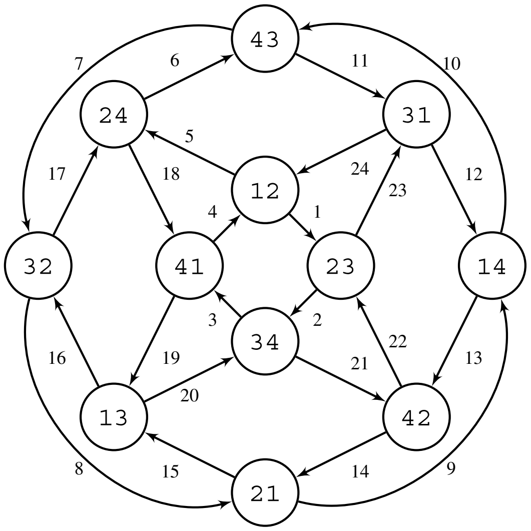
where $b \in \{1,2,\ldots,n\} - \{a_1,a_2,\ldots,a_{n-2}\}$. Each such edge labelled by b corresponds to the shortened permutation $a_1a_2\cdots a_{n-2}b$. The graph $\mathcal{J}_4$ is displayed in the picture below, where labels of edges are implicit.

Observation. If $b_1b_2\cdots b_m$ is the label of an Eulerian cycle starting from $a_1a_2\cdots a_{n-2}$, $a_1a_2\cdots a_{n-2}b_1b_2\ldots b_m$ is a shortest superword for n-shortperms.

Example. In $\mathcal{J}_4$, the Eulerian cycle `12` $\to$ `23` $\to$ `34` $\to$ `41` $\to$ `12` $\to$ `24` $\to$ `43` $\to$ `32` $\to$ `21` $\to$ `14` $\to$ `43` $\to$ `31` $\to$ `14` $\to$ `42` $\to$ `21` $\to$ `13` $\to$ `32` $\to$ `24` $\to$ `41` $\to$ `13` $\to$ `34` $\to$ `42` $\to$ `23` $\to$ `31` $\to$ `12` produces the superword of length $26 = 4!+4-2$ prefixed by `12`:

`12 341243214314213241342312.`

To answer the question it is sufficient to show that the graph $\mathcal{J}_n$ is an Eulerian graph.



***Lemma* 22** *For two n-shortperms with the same set of elements one is reachable from the other in the Jackson graph $\mathcal{J}_n$.*

Proof It is enough to show that for each shortperm $a_1a_2\cdots a_{n-2}$ there is a path in $\mathcal{J}_n$ to its cyclic shift $a_2\cdots a_{n-2}a_1$ and to $a_2a_1a_3\cdots a_{n-2}$ (transposition of the first two elements) because any permutation can be decomposed into a series of such permutations.

We sketch it for a shortperm of the form $\texttt{123}\cdots n-2$ using the representative example of `12345` for $n=7$. Let $a=6$ and $b=7$. In $\mathcal{J}_7$ we have the path

$$\texttt{12345} \to \texttt{2345}a \to \texttt{345}ab \to \texttt{45}ab\texttt{2} \to \texttt{5}ab\texttt{23} \to ab\texttt{234} \\ \to b\texttt{2345} \to \texttt{23451},$$

which shows that there is a path $\texttt{12345} \xrightarrow{*} \texttt{23451}$ from `12345` to its shift `23451`. There is also a path $\texttt{12345} \xrightarrow{*} \texttt{345}ab$. Then using them both we get the path

$$\texttt{12345} \xrightarrow{*} \texttt{345}ab \xrightarrow{*} ab\texttt{345} \longrightarrow b\texttt{3452} \longrightarrow \texttt{34521} \xrightarrow{*} \texttt{21345},$$

which corresponds to the transposition of the first two elements. This completes the sketch of the proof. ■

The lemma induces that the Jackson graph is strongly connected. It is also regular, that is, the in-degree and out-degree of each node equal 2. It is known that these conditions imply that it is an Eulerian graph. Following the observation we can extract a required superword from an Eulerian cycle.

Notes

The problem of constructing a shortest superword for all shortened n-permutations has been completely solved by Jackson [151, 211].

The problem is equivalent to finding a Hamiltonian cycle in the *line graph* $\mathcal{H}_n$ of $\mathcal{J}_n$. The nodes of $\mathcal{H}_n$, identified with shortened permutations, correspond to edges of $\mathcal{J}_n$: there is an edge (e,e') in $\mathcal{H}_n$ if and only if the starting node of the edge e' in $\mathcal{J}_n$ is the end node of e.

Edges of $\mathcal{H}_n$ can be labelled as follows. For each node $a_1a_2\cdots a_{n-1}$ of $\mathcal{H}_n$ the graph has two labelled edges

$$a_1a_2\cdots a_{n-1} \xrightarrow{1} a_2\cdots a_{n-1}a_1 \text{ and } a_1a_2\cdots a_{n-1} \xrightarrow{0} a_2\cdots a_{n-1}a_1a_n,$$

where $a_n \notin \{a_1,a_2,\ldots a_{n-1}\}$. The Eulerian tour in $\mathcal{J}_n$ corresponds now to a Hamiltonian cycle in $\mathcal{H}_n$. For example, the Eulerian cycle in $\mathcal{J}_4$ from the previous example is associated with the Hamiltonian cycle in $\mathcal{H}_4$, after adding one edge at the end:

$$\begin{array}{l}
\texttt{123} \xrightarrow{0} \texttt{234} \xrightarrow{0} \texttt{341} \xrightarrow{0} \texttt{412} \xrightarrow{1} \texttt{124} \xrightarrow{0} \texttt{243} \xrightarrow{1} \texttt{432} \xrightarrow{0} \\
\texttt{321} \xrightarrow{0} \texttt{214} \xrightarrow{0} \texttt{143} \xrightarrow{1} \texttt{431} \xrightarrow{1} \texttt{314} \xrightarrow{0} \texttt{142} \xrightarrow{1} \texttt{421} \xrightarrow{0} \\
\texttt{213} \xrightarrow{1} \texttt{132} \xrightarrow{0} \texttt{324} \xrightarrow{0} \texttt{241} \xrightarrow{0} \texttt{413} \xrightarrow{1} \texttt{134} \xrightarrow{0} \texttt{342} \xrightarrow{1} \\
\texttt{423} \xrightarrow{0} \texttt{231} \xrightarrow{1} \texttt{312} \xrightarrow{1} \texttt{123}.
\end{array}$$

The Hamiltonian cycle starting at 123 is identified by the word of labels $x = \texttt{000101000110101000101011}$.

Interestingly there is a family of words x_n describing a Hamiltonian cycle in the graph $\mathcal{H}_n$ and having a very compact description. We use the interesting operation $\odot$ on words defined as follows. For two binary words u and $v = v[0\,.\,.\,k-1]$, let

$$u \odot v = u\, v[0]\, u\, v[1]\, u\, v|2] \cdots u\, v[k-1].$$

Let $\overline{u}$ denote the operation of negating all symbols in u. Then for $n \geq 2$ let

$$x_2 = \texttt{00} \text{ and } x_{n+1} = \texttt{001}^{n-2} \odot \overline{x_n}.$$

For example, $x_3 = \texttt{00} \odot \texttt{11} = \texttt{001001}$ and $x_4 = \texttt{001} \odot \texttt{110110} = (\texttt{0011}\,\texttt{0011}\,\texttt{0010})^2$. It is shown in [211] that for $n \geq 2$ the word x_n describes a Hamiltonian cycle starting from $\texttt{123} \cdots n-1$ in $\mathcal{H}_n$.

The connection between words and Hamiltonian cycles happening here is similar to the relation between de Bruijn words and Hamiltonian cycles in de Bruijn graphs.