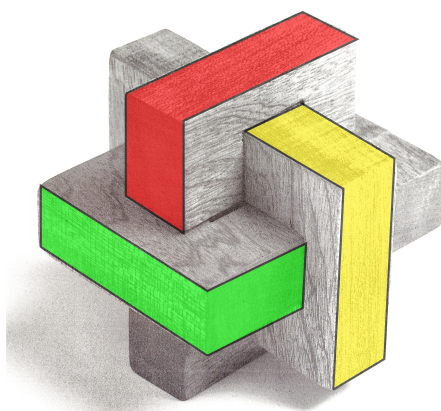


Extra Problems in Text Algorithms

Companion to the Book
"125 Problems in Text Algorithms"



Maxime Crochemore
Thierry Lecroq
Wojciech Rytter

November 14, 2024

Preface

This very preliminary text is related to “Algorithms on Texts”, also called “Algorithmic Stringology”. It is an extension of the book “125 Problems in Text Algorithms” (see reference [12]) providing, in the same compact style, more problems with solutions. We refer also to

<http://125-problems.univ-mlv.fr>

where all 150 problems (including these extra ones) are briefly announced.

The selected problems satisfy simultaneously three criteria: challenging, having short tricky solutions and solvable with only very basic background in stringology.

For the basics in stringology we refer to [12, Chapter 1].

July 2055

M. Crochemore, T Lecroq, W. Rytter
Paris, Rouen (France), Warsaw (Poland)

1	Extra Problems	1
126	Subsequence Covers	2
127	String attractors	5
128	Words with distinct cyclic k-factors	8
129	Shrinking a text by pairing adjacent symbols	11
130	Local periodicity lemma with one don't care symbol	14
131	Text index for patterns with one don't care symbol	16
132	1-Error Correcting Linear Hamming Codes	18
133	Huffman codes vs entropy	21
134	Compressed pattern matching in Thue-Morse words	23
135	Words representing combinatorial generations	25
136	Algorithm for 2-Anticovers	30
137	Short Supersequence of Shapes of Permutations	32
138	Computing short distinguishing subsequence	35
139	Yet another application of Suffix trees	37
140	Two longest subsequence problems	39
141	Two problems on Run-Length Encoded words	41
142	Maximal Number of (distinct) Subsequences	43
143	Avoiding Grasshopper repetitions	44
144	Counting unbordered words and relatives	46
145	Cartesian Tree Pattern-Matching	49
146	List-Constrained Square-Free Strings	52
147	Superstrings of shapes of permutations	55
148	Linearly generated words and primitive polynomials	58
149	An application of linearly generated words	61
150	Algorithm testing idempotent equivalence of words	64



1 Extra Problems

126 Subsequence Covers

A word x is a subsequence cover (s-cover, in short) of a word y if each position on y belongs to an occurrence of x as a subsequence of y .

Example. The word $x = 010$ is a (shortest) s-cover of $y = 0110110$ as well as of $y = 000011000$. However, $x = 010010$ is not because it is not a subsequence of them, nor $x = 0101$ because it does not s-cover their last position.

Question. Let y be a word in $\{0, 1, \dots, n-1\}^n$. Design a linear-time algorithm that checks if a given word x of length $m < n$ is an s-cover of y .

Solution

Let $y = y[0..n-1]$ and $x = x[0..m-1]$. Define the two lists of positions on y , $\mathbf{L} = (p_1, p_2, \dots, p_m)$ and $\mathbf{R} = (q_1, q_2, \dots, q_m)$, as the lexicographically first and last subsequences of positions on y , respectively, corresponding to x as a subsequence of y . Additionally, it is required that $p_1 = 0$ and $q_m = m-1$.

Note that \mathbf{L} or \mathbf{R} may not exist (see the above example). This can be tested readily in linear time with a greedy algorithm while computing the lists. Then, we assume up to now that \mathbf{L} and \mathbf{R} exist as defined. By definition, $x = y[p_1, p_2, \dots, p_m] = y[q_1, q_2, \dots, q_m]$.

Example. When $x = 020$ and $y = 021000201010120, many lists of positions on y are associated with x as a subsequence of y . Among them, the choosen lists are $\mathbf{L} = (0, 1, 3)$ and $\mathbf{R} = (11, 13, 14)$ (underlined letters).$

Observation 1. A position i on y is s-covered by x (as a subsequence) if there is a prefix p_1, p_2, \dots, p_{k-1} of \mathbf{L} and a suffix $q_{m-k+1}, q_{m-k+2}, \dots, q_m$ of \mathbf{R} for which $p_{k-1} < i < q_{m-k+1}$ and $x[k] = y[i]$.

To implement efficiently the observation, for i position on y define two auxiliary tables

$$\text{LEFT}[i] = |\{k \in \mathbf{L} : k < i\}|,$$

$$\text{RIGHT}[i] = |\{k \in \mathbf{R} : k > i\}|.$$

We also define, for $0 < i < m-1$:

$$P[i] = \max\{k : k \leq \text{LEFT}[i] + 1 \text{ and } x[k] = y[i]\} \cup \{0\}.$$

Example. For $y = 010210201$ and $x = 01201$ we get

$$\mathbf{L} = \{0, 1, 3, 5, 8\}, \quad \mathbf{R} = \{2, 4, 6, 7, 8\}$$

i	0	1	2	3	4	5	6	7	8
LEFT	0	1	2	2	3	3	4	4	4
RIGHT	5	5	4	4	3	3	2	1	0
P	1	2	1	3	2	4	3	4	5

Let Ψ be the predicate

$$\Psi(x, y) \stackrel{\text{def}}{=} \forall i \in [0 \dots m-1] \ P[i] > 0 \text{ and } P[i] + \text{RIGHT}[i] \geq |x|,$$

With this terminology Observation 1 restates as follows and leads to Algorithm s-COVER.

Observation 2. The word x is an s-cover of y if and only if $\Psi(x, y)$.

The algorithm can be written as the following pseudocode.

```

s-COVER ( $x, y$  non-empty words)
1  compute LEFT[ $i$ ], RIGHT[ $i$ ] for each  $i$ 
2  initially F[ $s$ ] = 0 for each letter  $s$  ;
3   $k \leftarrow 1$ 
4  F[ $x[0]$ ]  $\leftarrow 1$ 
5  ▷ Computing the table P
6  for  $i \leftarrow 1$  to  $|y| - 2$  do
7       $j \leftarrow \text{LEFT}[i]$ 
8      if  $i = p_{j+1}$  then
9          F[ $y[i]$ ]  $\leftarrow j + 1$ 
10     P[ $i$ ]  $\leftarrow \text{F}[y[i]]$ 
11 return  $\Psi(x, y)$ 

```

If the tables P, RIGHT are known then $\Psi(x, y)$ can be computed in linear time.

The computation of tables LEFT, RIGHT is very simple, we omit details. The table P is computed on-line using the auxiliary table F. This table satisfies:

in the moment immediately after we execute " $P[i] = \text{F}[y[i]]$ ", for each symbol s the value of $F(s)$ is the length of the longest prefix of x which ends with s and which is a subsequence of $y[0 \dots i]$.

Correctness follows from Observation 1 and Observation 2.

Notes

Our algorithm is a version of the one in [9].

The notion of an s-cover differs substantially from the notion of a standard cover:

- Two shortest s-covers of a same string can be distinct.
- Computing the length of a shortest s-cover is probably NP-hard.
- Every binary word of length at least 4 admits a nontrivial s-cover.
- In general, if the size k of the alphabet is fixed, then the length $\gamma(k)$ of the longest word without any nontrivial s-cover is finite, though exponential w.r.t. k . It is known that $\gamma(3) = 8$, $\gamma(4) = 19$. The exact value of $\gamma(5)$ is unknown.

127 String attractors

The notion of the **string attractor** provides a unifying framework for known dictionary-based compressors. A string attractor on a word x is a subset Att of positions on x for which each factor u of x has an occurrence covering at least one position in Att . That is, there are positions i, j and t satisfying $u = x[i..j]$, $t \in Att$ and $i \leq t \leq j$. We concentrate on attractors on two families of words.

Thue-Morse words. Thue-Morse words are defined as follows:

$$\tau_0 = \mathbf{a}, \quad \tau_{k+1} = \tau_k \overline{\tau_k}, \quad \text{for } k \geq 0,$$

where the bar morphism is defined by $\overline{\mathbf{a}} = \mathbf{b}$ and $\overline{\mathbf{b}} = \mathbf{a}$. Note the length of the k th Thue-Morse word is $|\tau_k| = 2^k$ and $\overline{\tau_{k+1}} = \overline{\tau_k} \tau_k$. It can be checked directly that $\{4, 6, 8, 12\}$ and $\{4, 8, 10, 12\}$ are attractors on τ_4 .

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\tau_4[i]$	a	b	b	a	b	a	a	b	b	a	a	b	a	b	b	a

Question. Construct an attractor of size at most 4 for Thue-Morse words τ_k , $k \geq 4$.

Solution

The clue of the solution is to consider middle positions in words. By the middle position of a word with even length $2 \times m$ we mean position m . Indeed such a position captures many (distinct) factors occurring in the word. For example, position 3 on **aaabbb** is covered by 12 factors and position m on $\mathbf{a}^m \mathbf{b}^m$ covered by $m(m+1)$ factors, a quadratic number with respect to the length of the word. Adding only position $m-1$ gives the attractor $\{m-1, m\}$. Let $Mid(x)$ be the set of factors of x that have an occurrence in x covering the middle of x and let $Fact(x)$ be the set of all nonempty factors of x . We have the following fact for $k \geq 4$.

Fact.

$$(a) \quad Fact(\tau_k) = Mid(\tau_k) \cup Fact(\tau_{k-1}) \cup Fact(\overline{\tau_{k-1}}).$$

$$(b) \quad Fact(\tau_k) = Mid(\tau_k) \cup Mid(\tau_{k-1}) \cup Mid(\overline{\tau_{k-1}}) \cup Mid(\overline{\tau_{k-2}}).$$

Proof Point (a) follows from the recursive description of τ_k . Then

$$Fact(\tau_k) = \bigcup_{i=1}^k Mid(\tau_i) \cup \bigcup_{i=1}^{k-1} Mid(\overline{\tau_i})$$

Now the thesis follows from the fact that τ_{k-2} is a central part of τ_k , and similarly $\overline{\tau_{k-2}}$ is a central part of $\overline{\tau_k}$. The same holds for τ_{k-2} , $\overline{\tau_{k-2}}$ and

their central parts. Hence, in the above equation it is enough to keep these four largest Thue-Morse words and their barred images. ■

Construction of an attractor on τ_k . Due to Fact 1 it is enough to take middle points in $\tau_k, \tau_{k-1}, \overline{\tau_{k-1}}, \overline{\tau_{k-2}}$. However all these words are parts of τ_k . When $k \geq 4$, from its recursive definition τ_k can be written as a composition of 8 fragments of the same length $A \cdot B \cdot B \cdot A \cdot B \cdot A \cdot A \cdot B$, where $\tau_{k-1} = ABBA$, $\overline{\tau_{k-1}} = BAAB$ and $\overline{\tau_{k-2}} = BA$. The 4 sought middle points are the middle points of the occurrences of $ABBABAAB$, $ABBA$, $BAAB$ and BA in τ_k . Then

$$\{2^{k-1}, 2^{k-2}, 2^{k-1} + 2^{k-2}, 2^{k-2} + 2^{k-3}\}$$

is an attractor on τ_k . Note that $2^{k-1} + 2^{k-3}$ can be substituted for $2^{k-2} + 2^{k-3}$ because there are two occurrences of BA in $ABBABAAB$.

Example. Splitting τ_5 of length 32 into 8 equal-length fragments and pointed positions of its attractor $\{16, 8, 24, 12\}$:

$$\begin{array}{cccccccc} & \bullet & & \bullet & & \bullet & & \bullet \\ \text{abba} \cdot \text{baab} \cdot \text{baab} \cdot \text{abba} \cdot \text{baab} \cdot \text{abba} \cdot \text{abba} \cdot \text{baab}. \end{array}$$

Another attractor on τ_5 is $\{16, 8, 24, 20\}$.

Attractors on Fibonacci words . The Fibonacci word fib_k is $\phi^k(\mathbf{a})$, $k \geq 0$, where the morphism ϕ is defined by $\phi(\mathbf{a}) = \mathbf{ab}$ and $\phi(\mathbf{b}) = \mathbf{a}$. The length of fib_k is the Fibonacci number F_{k+2} ($F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, \dots).

It can be checked directly that $\{4, 7\}$ and $\{6, 7\}$ are both attractors on fib_5 . Attractors of size 2 are obviously of the smallest size because two of its positions have to point on two different letters.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$fib_5[i]$	a	b	a	a	b	a	b	a	a	b	a	a	b

Question. Find two essentially distinct attractors of size 2 for each Fibonacci word fib_k , $k \geq 2$.

Solution

For Fibonacci words, using similar arguments as in the proof of Problem 127, it comes that $\{|fib_{k-1}| - 1, |fib_{k-2}| - 1\}$ is an attractor on fib_k . We show now a more attractive attractor consisting of two adjacent positions on fib_k , namely the last two positions of its prefix fib_{k-1} . In the proof we use a known property of Fibonacci words that is recalled first.

Lemma 1

Two consecutive Fibonacci words almost commute: more accurately, for $k \geq 2$, $fib_k fib_{k-1} = uw$ and $fib_{k-1} fib_k = u\bar{w}$, where $w = \mathbf{ab}$ if k is even and $w = \mathbf{ba}$ if k is odd.

Proof The proof is by induction on k . The conclusion can be checked for $k = 2$: $fib_2 fib_1 = aba \cdot ab = uab$ and $fib_1 fib_2 = ab \cdot aba = uba$, where $u = aba$.

For $k > 2$, by the definition of fib_k , $fib_k fib_{k-1} = fib_{k-1} fib_{k-2} fib_{k-1}$ and $fib_{k-1} fib_k = fib_{k-1} fib_{k-1} fib_{k-2}$. The induction hypothesis applied to fib_{k-1} and fib_{k-2} induces: $fib_{k-1} fib_{k-2} = vw$ and $fib_{k-2} fib_{k-1} = v\bar{w}$, with $w = ab$ if and only if $k - 1$ is even.

Setting $u = fib_{k-1}v$, the conclusion follows. \blacksquare

Proposition. The set $X_k = \{|fib_{k-1}| - 2, |fib_{k-1}| - 1\}$ of positions on fib_k is an attractor on fib_k for $k > 1$.

Example. Pointed positions of the attractor $\{6, 7\}$ on fib_5 .

• •
abaababa · abaab.

Note that $\{|fib_{k-1}|, |fib_{k-1}| + 1\}$ is not an attractor on fib_k , $k \geq 3$. For the example of fib_5 , the set $\{8, 9\}$ does not capture the factor **baba**.

Proof A direct examination shows that the result holds for $fib_2 = aba$ and $fib_3 = abaab$. Indeed, $\{0, 1\} = \{|fib_1| - 2, |fib_1| - 1\}$ is an attractor on fib_2 and $\{1, 2\} = \{|fib_2| - 2, |fib_2| - 1\}$ is an attractor on fib_3 . The rest of the proof is by induction on k . Let $k > 3$ and assume the result holds for fib_{k-2} . To prove the statement it is enough to show that each word in $Fact(u) \cup Fact(v)$ has an occurrence touching at least one position in X_k because this is obviously true for the other factors of fib_k .

Claim 1. Each word in $Fact(v)$ has an occurrence touching X_k .

Observe that fib_{k-2} is a prefix of the suffix v of fib_k . Due to the inductive assumption, X_k becomes an attractor on the prefix copy of fib_{k-2} . Then each factor of fib_k starting in v has an occurrence touching X_k .

Claim 2. Each word in $Fact(u)$ has an occurrence touching X_k .

This claim follows from the fact that u is a prefix of v . This completes the whole proof. \blacksquare

Notes

The relation between string attractors and text compression is by Kempa and Prezza [34]. Further results on attractors can be found in [41]. Testing if a set of positions form an attractor can be done with the algorithm described in [12, Problem 64]. Existence of 2-attractor is an NP-hard problem, see [19]. The explicit description of attractors on Thue-Morse words is by Kutsukake et al. [38]. Further results on attractors can be found in [41]. Testing if a set of positions form an attractor can be done with the algorithm described in [12, Problem 64].

128 Words with distinct cyclic k -factors

Assume the alphabet is $\{0,1\}$. A binary word v is called a cyclic factor of a word w , if v is a (standard) factor of w^∞ . A binary (cyclic) word w of length $k \leq n \leq 2^k$ is called a k -ring word if each cyclic k -factor of w occurs once. For example, the word $w = 000101101$ is a binary 4-ring word. Observe that string of length k is a k -ring if and only if it is primitive.

We refer to [12, Problem 18], [12, Problem 69] for the definition of de Bruijn graph G_k . Two edges of G_k are loops and in this problem we disregard these two edges. The nodes of G_k are binary words of length $k-1$, and edges correspond to words of length k . The number of edges of G_k is 2^k . The size of these graphs is $O(n)$ since we can choose minimal k such that $n \leq 2^k$. A **closed chain** (c-chain, in short) is a path C ending and starting in the same node and containing each of its edges exactly once. Denote by $w = \text{RingWord}(C)$ a word w resulting by spelling labels of consecutive edges of C .

Question. Design a linear time algorithm constructing a binary k -ring word, for given n, k , such that $k \leq n \leq 2^k$.

Equivalent formulation: construct a closed chain C of length n in G_k , then $\text{RingWord}(C)$ is a k -ring word of length n .

Solution

Assume the c-chains are represented as cyclic lists of consecutive nodes.

Fact 1. Assume H is a regular subgraph of G_k , such that each node of G_k is contained in an edge in H . Then we can compute a single c-chain $\text{GLUE}(H)$ in G_k of length n .

Proof Graphs G_k have the following simple property ([12, Problem 69]).

Claim. Assume we are given two node-disjoint c-chains C_1, C_2 , and an edge $u \rightarrow v$ in G_k , where $u \in C_1, v \in C_2$. Then in time $O(1)$ we can create a new c-chain $\text{merge}(C_1, C_2)$ of length $|C_1| + |C_2|$, whose set of nodes is the union of sets of nodes of C_1, C_2 .

Each connected component of H is an Eulerian graph. We can compute c-chains, containing all nodes of this component, using Euler algorithm in linear time. If there is one component we are done. Otherwise there should be two c-chains C_1, C_2 satisfying assumption of Fact 1, since H has no isolated nodes, and because G_k is a connected graph. Then, we replace C_1, C_2 by $\text{merge}(C_1, C_2)$ into a single c-chain. We iterate this

process until we get a single c-chain which is a required output. ■

We say that a set X of edge-disjoint c-chains is covering G_k if it contains each edge of G_k .

Fact 2. We are given a c-chain C in G_k . Then we can compute in time $O(|G_k|)$ a set $\text{Compl}(C, k)$ of c-chains such that $\text{Compl}(C, k) \cup \{C\}$ is covering G_k (in particular $\text{Compl}(C, k)$ has together $2^k - |C|$ edges).

Proof A directed graph is called *regular* if for each node the numbers of its out-going and in-going edges are equal (though can differ for distinct nodes). It is known that a directed graph has an Euler cycle if and only if it is regular and connected. We remove edges of C (but not nodes) and receive the graph $G_k - C$, afterword each connected component contains a directed Eulerian graph. Then $\text{Compl}(C, k)$ consists of Eulerian cycles of these components and the cycle C . ■

Each edge the form $a_1 a_2 \cdots a_{k-1} \xrightarrow{a_k} a_2 a_3 \cdots a_k$, in G_k corresponds to the node $a_1 a_2 \cdots a_k$ in G_{k+1} . Each c-chain C of length $n \leq 2^{k-1}$ in G_{k-1} corresponds to a simple cycle of length n in G_k , denote this cycle by $\Phi_k(C)$. The edges of C correspond to nodes of $\Phi_k(C)$.

Example. Below we show a c-chain C in G_3 and $\Phi_4(C)$.

$$\begin{aligned} C &= 01 \xrightarrow{0} 10 \xrightarrow{1} 01 \xrightarrow{1} 11 \xrightarrow{0} 10 \xrightarrow{0} 00 \xrightarrow{0} 00 \xrightarrow{1} 01 \\ \Phi_4(C) &= 010 \xrightarrow{1} 101 \xrightarrow{1} 011 \xrightarrow{0} 110 \xrightarrow{0} 100 \xrightarrow{0} 000 \xrightarrow{1} 001 \xrightarrow{0} 010 \end{aligned}$$

A pseudo-code of the recursive algorithm computing closed chain of length n in G_k , for $n \leq 2^k$, is given below.

Algorithm $\text{ComputeChain}(k, n)$

1. if $n \leq 2^{k-1}$ then
 $C := \text{ComputeChain}(k-1, n)$; return $\Phi_k(C)$
2. $H := G_k$
3. Let $n = 2^{k-1} + r$, $0 < r \leq 2^{k-1}$
4. $C := \text{ComputeChain}(k-1, r)$ (*recursive call*)
5. For each $C' \in \text{Compl}(C, k-1)$ remove edges of $\Phi_k(C')$ from H
 $(H$ has $2^k - (2^{k-1} - r)$ edges, it satisfies assumptions of Fact 1)
6. return $\text{GLUE}(H)$

The time complexity $T(n)$ of the algorithm satisfies $T(n) = O(T(n/2) + O(n))$. It implies $T(n) = O(n)$.

Notes

Our algorithm is a version of the algorithms in [54, 20]. Recently, in [21],

there were investigated words w , called *orientable sequences*, such that all cyclic length- n factors of w and w^R are distinct.

129 Shrinking a text by pairing adjacent symbols

One of the most powerful compression techniques is recompression. In this technique there are two crucial operations: shrinking unary runs and *pairing letters*.

A unary run is a maximal occurrence of a factor of length at least 2 that is a repetition of the same letter. The first phase of the recompression technique consists in shrinking each unary run into a single letter.

The second phase is to apply the operation $Compress(x, L, R)$, where (L, R) is a partition of the alphabet \mathbf{A} of letters, $L \cup R = \mathbf{A}$ and $L \cap R = \emptyset$. The compressed word $Compress(x, L, R)$ results from x by substituting a single letter (identifier of a pair of letters) for each occurrence of its 2-letter factors ab , whenever $a \in L$ and $b \in R$.

The **Pairing problem** consists in computing a partition (L, R) of the alphabet of x for which $|Compress(x, L, R)| \leq \frac{3}{4}|x|$.

Example. Consider the word $abcacbabcbac$. Let $L = \{a, c\}$, $R = \{b\}$. Then, substituting d for ab and e for cb produces the word $dcaedeac$ of length $8 < \frac{3}{4}12 = 9$. On the contrary, setting $L = \{a\}$, $R = \{b\}$ and substituting d for ab in the word $aaabbb$ containing two unary runs produces the word $aadbb$ of length $5 > \frac{3}{4}6 = 4.5$, which does not meet the above bound.

Question. Let x , $|x| \geq 2$, be a word over an integer alphabet containing no unary runs. Show how to compute in linear time a partition (L, R) of the alphabet of x for which $|Compress(x, L, R)| \leq \frac{3}{4}|x|$.

Solution

A solution to the pairing problem reduces to the following question.

$\frac{1}{4}$ -cut problem. Let $G = (V, E)$ be a directed multigraph without self-loops; the goal is to compute a partition (L, R) of the set V of vertices for which at least $\frac{1}{4}|E|$ arcs lead from L to R .

Lemma 2

The $\frac{1}{4}$ -cut problem can be solved in time $O(|V| + |E|)$.

Proof For $A, B \subseteq V$, let $E(A, B)$ be the set of arcs leading from A to B and let $\deg(A, B) = |E(A, B)|$.

We use the following algorithm.

```

PARTITION( $G = (V, E)$  a directed multigraph)
1   $(M, L, R) \leftarrow (V, \emptyset, \emptyset)$ 
2  while  $M$  not empty do
3      Let  $v \in M$ 
4      if  $2\deg(v, R) + \deg(v, M) \geq 2\deg(L, v) + \deg(M, v)$  then
5           $L \leftarrow L \cup \{v\}$ 
6      else  $R \leftarrow R \cup \{v\}$ 
7       $M \leftarrow M \setminus \{v\}$ 
8  return  $(L, R)$ 

```

To show the result we consider the potential expression

$$\mathbf{P} = 4\deg(L, R) + 2\deg(L, M) + 2\deg(M, R) + \deg(M, M)$$

and prove that its value cannot decrease throughout a run of the algorithm. Let us denote

$$a = \deg(v, R), \quad b = \deg(v, M), \quad c = \deg(L, v), \quad d = \deg(M, v)$$

and consider the effect of moving v from M to L on the four terms of \mathbf{P} :

- $\deg(L, R)$ increases by a ;
- $\deg(L, M)$ increases by b and decreases by c ;
- $\deg(M, R)$ decreases by a ;
- $\deg(M, M)$ decreases by b and decreases by d .

Overall, \mathbf{P} increases by $4a + 2(b - c) - 2a - (b + d) = 2a + b - 2c - d$. Then, this quantity is non-negative when the algorithm decides to move v to L .

Similarly, if v is moved from M to R , \mathbf{P} does not decrease using a similar argument.

Upon the end of the algorithm, we have $\mathbf{P} = 4\deg(L, R)$ due to $M = \emptyset$, while initially $\mathbf{P} = \deg(M, M) = |E|$ due to $M = V$. Since \mathbf{P} is nondecreasing, we conclude that $4\deg(L, R) \geq |E|$, which proves that $\deg(L, R) \geq \frac{1}{4}|E|$ as claimed.

Running time. To meet the expected running time the graph G is first preprocessed in linear time to compute the input and output degrees of vertices $v \in V$. Then, each iteration of the algorithm can be implemented in time $O(1 + \deg(v, V) + \deg(V, v))$, which yields a total running time of $O(|V| + |E|)$ as claimed. ■

Reduction of the pairing problem to $\frac{1}{4}$ -cut problem. Let x be a word of length at least 2 with no unary runs. Let $V = \text{alph}(x)$ be the set of letters occurring in x and let E be the set of edges $a \rightarrow b$, where ab is a factor of x and $a \neq b$. The number of edges from a to b , that

is, the output degree of a , is the number of occurrences of ab in x . Now the pairing problem reduces to the $\frac{1}{4}$ -cut in this graph and is solved by Algorithm PARTITION that computes a desired partition.

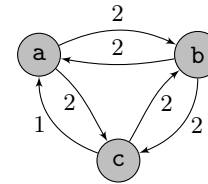
Example. Consider again the word **abcacbabcbac** of length 12. Let us run PARTITION on its associated graph (edges are labelled by the number of occurrences of their corresponding length-2 factor) and start with $M = \{a, b, c\}$, $L = \{\}$, $R = \{\}$.

Node $v = a$: $2 \times 0 + 4 \geq 2 \times 0 + 3$ gives $M = \{b, c\}$,
 $L = \{a\}$, $R = \{\}$.

Node $v = b$: $2 \times 0 + 2 \not\geq 2 \times 2 + 2$ gives $M = \{c\}$,
 $L = \{a\}$, $R = \{b\}$.

Node $v = c$: $2 \times 2 + 1 \geq 2 \times 2 + 0$ gives $M = \{\}$,
 $L = \{a, c\}$, $R = \{b\}$.

Then, factors **ab** and **cb** are replaced by new letters as seen above.



Notes

The recompression technique and the pairing problem can be found in [32]. This technique was successfully applied to many problems, especially to word equations. The newly created letters correspond to fragments of growing sizes. For recompressing a text, the process of pairing letters is iterated while receiving new letters. The whole process of creating new letters results globally in only a linear number of such letters, since meanwhile the size of the word decreases geometrically.

The recompression technique is technically very complicated, and details depend on the particular problem it is applied to.



130 Local periodicity lemma with one don't care symbol

The problem concerns periodicities occurring inside a word that contains one occurrence of a don't care symbol (also called hole or joker). It is a letter, denoted by $*$, that stands for any other letter of the alphabet, that is, it matches any letter including itself. For a string x , two of its letters, $x[i]$ and $x[j]$, are said to \approx -match, written $x[i] \approx x[j]$, if they are equal or one of them is the don't care symbol.

Further, an integer p is a **local period** of x if for each position i on x , $0 \leq i < |x| - p$, we have $x[i] \approx x[i + p]$. Recall the Periodicity lemma for usual words (see [12, Chapter 1]).

Lemma 3 (Periodicity lemma)

Let x be a word (without don't care symbol) and let p, q be periods of x that satisfy $p + q - \gcd(p, q) \leq |x|$. Then, $\gcd(p, q)$ is also a period of x .

The problem is related to an extension of the lemma to words in which only one don't care symbol occurs.

Question. (Local periodicity lemma) Let x be a word with one don't care symbol and p, q be two relatively prime local periods of x that satisfy $p + q \leq |x|$. Then, 1 is also a local period of x .

Question. Give an example word x with one don't care symbol having local periods $p = 5$ and $q = 7$ with $p + q - 1 = |x|$ but not having 1 as local period.

The example in this question shows the inequality in the first question is tight.

Solution

Let $n = |x|$ and assume $p + q = n$. The case $p + q < n$ can be easily reduced to this case.

Construct the graph $G(n, p, q)$ whose nodes are $0, 1, \dots, n - 1$ and whose undirected edges (i, j) are pairs of positions on x with $|i - j| \in \{p, q\}$. The Periodicity lemma implies that the graph is connected but to get the result we are to prove a stronger property in the next lemma, namely the biconnectivity of $G(n, p, q)$.

Lemma 4

Assume p, q are relatively prime and the word x has periods p, q , where $p + q = n$. If x has only one don't care symbol then x is unary.

Proof It is enough to show that the graph $G(n, p, q)$ is biconnected, that is, the removal of any single node, potentially a position of the don't care symbol, does not disconnect the graph.

It is easy to see that each node of $G(n, p, q)$ has degree 2. Hence the graph is a set of cycles. Due to the standard periodicity lemma (no don't care symbol) the graph $G(n-1, p, q)$ is connected. After removing the node 0 from $G(n, p, q)$ the remaining graph is isomorphic with $G(n-1, p, q)$, hence it is also connected (its nodes are $1, 2, \dots, n-1$). Consequently the whole graph $G(n, p, q)$ is a connected graph.

The graph $G(n, p, q)$ does not contain loops and consists of a set of disjoint simple cycle. Therefore it is just one big cycle, because it is connected. Hence $G(n, p, q)$ is biconnected, since a single simple cycle is biconnected. This completes the proof. ■

Solution to the second question. The word **ababaababa** of length 10 has periods 5 and 7. Note the Periodicity lemma does not apply to it since $5 + 7 - \gcd(5, 7) = 11 > 10$. The word $x = \mathbf{ababaababa}^*$ of length 11 has local periods 5 and 7 but obviously not period 1 as required, despite the equality $5 + 7 - 1 = |x|$.

Notes

A first proof of the Local periodicity lemma with one don't care symbol was given by Berstel and Boasson in [8], however our solution is different. A version of the Local periodicity lemma with two don't care symbols is rather nontrivial.

The notion of solid periodicities is thoroughly investigated by Kociumaka et al. in [36] in conjunction with words containing don't care symbols. An integer p is a solid period of x if there is a word z without don't cares and with period p for which $x \approx z$. If p is a local period it is not necessarily a solid period, see for example $x = \mathbf{a} * \mathbf{b}$ and $p = 1$.

Also the Solid periodicity lemmas are different. For example, if $|x| \geq 16$ has two don't cares and has solid periods 5, 7 then it should have 1 as a solid period. But this is not true for local periods, consider the example word $x = \mathbf{aaaaba} * \mathbf{aa} * \mathbf{abaaaa}$.



131 Text index for patterns with one don't care symbol

For a string w of size n over a (large) integer alphabet Σ we want to create a data structure $\mathbf{D}(w)$, of size $O(n \log n)$, called the *text index*, which allows to search for a pattern P in w in $O(|P|)$ time (usually $|P| \ll n$). The pattern P can contain a single occurrence of a special symbol $\theta \notin \Sigma$ called a *don't care* or a *wildcard*, which matches any other symbol in w . In this simplified problem we do not ask about time complexity of constructing $\mathbf{D}(w)$, since it is quite technical (see the notes). Our main aim here is only a small size of $\mathbf{D}(w)$ and fast searching of the pattern.

Combinatorics of trees.

We consider only trees with each internal nodes having at least two children. By a size of a tree we mean the number of its leaves. For each internal node v denote by T_v the subtree rooted at v . An edge $v \rightarrow u$, where v is the *parent* of u is called *heavy* if T_u has largest size among subtrees rooted at children of v (in case of ties we choose a single edge). Other edges are called *light*. If $v \rightarrow u$ is heavy then subtrees rooted at other children of v are called *light subtrees*.

Observe that each path from a given leaf to the root contains only logarithmically many light edges, hence each leaf belongs to logarithmically many light subtrees. Consequently we have the following fact.

Observation 1. The sum of sizes of all light subtrees is $O(|T| \log |T|)$.

Question. Construct the text index $\mathbf{D}(w)$ of size $O(n \log n)$ with searching time $O(|P|)$.

[Hint: Use Observation 1.]

Solution

We assume a word w ends with special endmarker. Let $ST(w)$ be the suffix tree of w . For a trie T' denote by $strings(T')$ the set of strings corresponding to paths $root \xrightarrow{*} leaf$ in T' .

Denote by $LightStrings(v)$ the set of strings corresponding to paths $v \xrightarrow{*} leaf$ in $ST(w)$ starting with light edges originating at v .

Let $NewTree(v)$ be a compacted trie T' such that

$\alpha \in strings(T')$ if and only if $(\exists a \in \Sigma) a\alpha \in LightStrings(v)$.

If q is the total size of light subtrees hanging at v then it can be easily seen that $|NewTree(v)| = O(q)$. (we do not ask about time complexity of constructing $NewTree(v)$), and we refer to [10].

A pseudocode of the construction of $\mathbf{D}(w)$ is given below, see also the figure.

Algorithm Construct $\mathbf{D}(w)$

For each original non-leaf node v of $ST(w)$ do

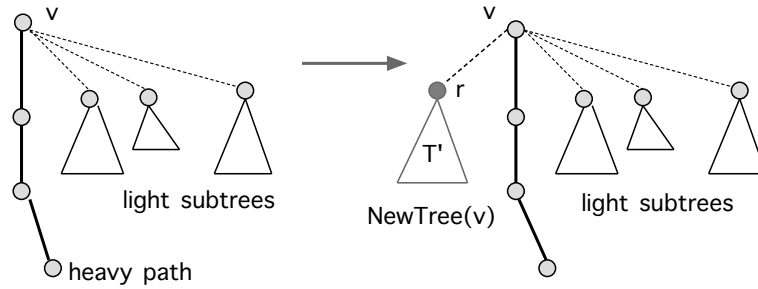
$T' = \text{NewTree}(v)$; $r := \text{root}(T')$

$r := \text{root}(T')$

$\text{next}(v) = r$; $\text{parent}(r) = v$

create additional edge $v \xrightarrow{\Theta} r$

Size of $\mathbf{D}(w)$. The total size of additional (after merging) trees is at most the total size of all light trees. Hence, due to Observation 1, $|\mathbf{D}(w)| = O(n \log n)$.



Searching the pattern P . We scan P and follow the downward path in T . However when we see θ in P we split the search. We go to $\text{Next}(v)$ and to the next node on the heavy path. Then we follow two disjoint paths in $\mathbf{D}(w)$. It still takes $O(|P|)$ time.

Notes

In case of k don't care symbols, for $k = O(1)$, one can construct the text index $\mathbf{D}(w)$ of size $O(n \log^k n)$ with searching time $O(|P|)$. Initially we proceed similarly as in case one error. Then we recursively process the newly added subtrees (in the merges) with respect to $k - 1$ don't cares.

In the case of don't cares in the pattern the bottleneck of time complexity of constructing $\mathbf{D}(w)$ is the construction of the tries $\text{NewTree}(v)$, it is technical and we refer to [10].

Our presentation is a version of the simplest case of an approximate text index presented in [10], where don't cares and edit operations in the text were allowed, however the general case is very technical.

132 1-Error Correcting Linear Hamming Codes

Sending a message through a noisy line may produce errors. The goal of the problem is to present a method for correcting a message in which only one error is assumed to occur. A message is a word of bits. To allow checking and correcting a possible transmission error in a word $w \in \{0, 1\}^*$, a very short word depending on it and easily computable, $f(w)$, is appended to w . The complete message to be sent is then $\text{code}(w) = w \cdot f(w)$.

We assume that the length of a message w is of the form $k = 2^r - r - 1$, $|f(w)| = r$ and the total length of the code is then $n = k + r = 2^r - 1$, for an integer $r > 2$. Hence, the size r of the additional part of the code is only logarithmic according to length of the message. Such codes are called (n, k) -codes. We consider only binary words and use linear algebra methods. A word $b_1 b_2 \cdots b_k$ is identified with the vector $[b_1, b_2, \dots, b_k]$.

Observation. The set of length- n binary words containing at least two occurrences of 1 has size $k = 2^r - r - 1$. For example, the set of such length-4 words has 11 elements, all 16 4-bit words except the 5 words 0000, 0001, 0010, 0100, 1000.

Example. A possible function f for a $(7, 4)$ -code is

$$f(b_0 b_1 b_2 b_3) = [b_0 + b_1 + b_2, b_0 + b_1 + b_3, b_0 + b_2 + b_3], \text{ and}$$

$$\text{code}(b_0 b_1 b_2 b_3) = [b_0, b_1, b_2, b_3, b_0 + b_1 + b_2, b_0 + b_1 + b_3, b_0 + b_2 + b_3],$$

where b_i are bits and the operation $+$ is *xor*.

We construct $f(w)$ as $M \times w^T$, multiplication of a $r \times k$ matrix M by the transposed vector associated with w . Then,

$$\text{code}^M(w) = [w, f(w)] = [w, M \times w^T].$$

Example (followed). The matrix of the above $(7, 4)$ -code is

$$M = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

For example $\text{code}^M(1010) = 1010010$, in this case $f(1010) = 010$. The length- n elements of $\text{Codes}_n^M = \{\text{code}_n^M(w) : w \in \{0, 1\}^k\}$ are called codewords and the set Codes_n^M is called a **Hamming code** if

$$(*) \quad \min \{ \text{Ham}(u, v) : u, v \in \text{Codes}_n^M, u \neq v \} \geq 3,$$

where $\text{Ham}(u, v)$ is the Hamming distance (number of mismatches).

Question. Build a matrix M for which Codes_n^M is a Hamming code.

[Hint: Use the observation.]

Question. Show how to correct the message assuming it contains at most one error.

Solution

Let I_r be the $r \times r$ identity matrix, and let the Parity checking matrix be horizontal concatenation of matrices M and I_r .

Observation. The columns of P are all nonzero binary words of size r . Following the above example where $r = 3$, it is

$$P = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

The next property of P is a reformulation of the definition of the function code. Observe that operations on matrices are modulo 2, and that the equality $x = y \bmod 2$ is equivalent to $x + y = 0 \bmod 2$. Let us denote by $\bar{0}$ the vector whose components are zeros.

Fact. $x \in \text{Codes}_n^M$ if and only if $P \times x^T = \bar{0}$.

We are ready to show property (*). Assume, by contradiction, that this property is false and that, for $u \neq v$, $u, v \in \text{Codes}_n^M$, we have $0 < \text{Ham}(u, v) < 3$. Let $x = u - v$ (subtraction modulo 2). Then x has exactly one or two occurrences of 1. We have also $P \times x^T = \bar{0}$.

If x has a single 1, then $P \times x^T$ is a single column of P , which cannot be $\bar{0}$ since all columns of P are nonzero vectors. Furthermore, if x has exactly two 1s then $P \times x^T$ is the sum of two columns of P . Then again, it cannot be a zero vector since every two distinct columns of P are linearly independent.

Hence, $P \times x^T \neq \bar{0}$, which contradicts the equality $P \times x^T = \bar{0}$ and completes the proof of property (*).

Larger example. Consider $r = 4$ and the 4×11 matrix for $(15, 11)$ -code

$$M = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

The function generating additional 4 bits for 11-bit messages is

$$f(b_0 b_1 \cdots b_{10}) = M \times [b_0, b_1, \dots, b_{10}]^T = [c_0, c_1, c_2, c_4], \text{ where}$$

$$c_0 = \sum_{i=0}^6 b_i, \quad c_1 = \sum_{i=3}^9 b_i, \quad c_2 = b_0 + b_1 + b_3 + b_4 + b_7 + b_8 + b_{10},$$

$$c_3 = b_0 + b_2 + b_3 + b_5 + b_7 + b_9 + b_{10}.$$

(Here, addition is modulo 2.)

Solution to the second question. Assume there is one error in the received message treated as a vector $y = x + \alpha$, where x is the message without error. The vector α contains exactly one element equals 1, say its i -th element. To locate the error we have to find i . We get

$$P \times y^T = P \times x^T + P \times \alpha^T = P \times \alpha^T,$$

since $P \times x^T = \bar{0}$. Then, $P \times \alpha^T$ is the i -th column of P and, since all columns of P are distinct, this uniquely determines the index i of the column as wanted.

Notes

Hamming codes have been introduced by R. W. Hamming in [26].



133 Huffman codes vs entropy

We consider n items with positive weights probabilities p_1, p_2, \dots, p_n satisfying $\sum_i p_i = 1$ and let $\mathbf{p} = (p_1, p_2, \dots, p_n)$. **Huffman algorithm** (see [12, Problem 99]) constructs a full binary tree (each non-leaf node has two children) with items assigned to its leaves. Let l_i be the depth of p_i , number of edges from the root to p_i . The average length of the **Huffman coding** of items is $\text{Huffman}(\mathbf{p}) = \sum_i p_i \cdot l_i$.

An important concept in information theory is **entropy**. The sequence \mathbf{p} is treated as a source of information and its entropy is

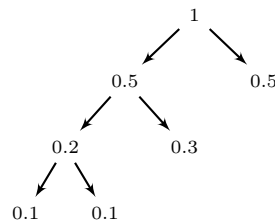
$$\text{Entropy}(\mathbf{p}) = - \sum_i p_i \cdot \log_2 p_i.$$

Property. A useful property for the solution below is: if p_1, \dots, p_n , and q_1, \dots, q_n are two sequences of positive integers with the same sum, then $-\sum_i p_i \log_2 q_i \geq -\sum_i p_i \log_2 p_i$. From the inequality, it follows directly: $\log_2 x \leq x - 1$, for $x > 0$.

Question. Show that $\text{Entropy}(\mathbf{p}) \leq \text{Huffman}(\mathbf{p}) \leq \text{Entropy}(\mathbf{p}) + 1$.

Example.

Besides is the Huffman tree corresponding to the sequence $\mathbf{p} = [0.1, 0.1, 0.3, 0.5]$. Then, $\text{Huffman}(\mathbf{p})$ is $0.1 \cdot 3 + 0.1 \cdot 3 + 0.3 \cdot 2 + 0.5 \cdot 1 = 1.7$ and $\text{Entropy}(\mathbf{p}) \approx 1.68548$. We have $1.68548 \leq 1.7 \leq 2.68548$.



Solution

The solution is built in three steps.

Fact 1.

- (i) In any full binary tree we have: $\sum_i 2^{-l_i} = 1$.
- (ii) The inequality $\sum_i 2^{-l_i} \leq 1$ implies that there is a binary (not necessarily full) tree with depths of leaves l_1, l_2, \dots, l_n .

Proof Proof of point (i). Choose two leaves that are children of the same node. They are at the same depth l . After removing these leaves their parent becomes a leaf at depth $l - 1$ and the whole sum $\sum_i 2^{-l_i}$ does not change. Eventually, we get a full binary tree with only 2 leaves. For this tree the above sum obviously equals 1.

Proof of point (ii). Assume now $\sum_i 2^{-l_i} \leq 1$. Take the maximum l_i . If there is another l_j with $l_j = l_i$, then create a node at depth $l_i - 1$ with 2 children. The depths l_i, l_j are removed and replaced by a single depth $l_i - 1$. If there is no such l_j , then create a new node at level $l_i - 1$ and having a single child.

This operation is iterated until the required tree is obtained. ■

Fact 2. $\text{Entropy}(\mathbf{p}) \leq \text{Huffman}(\mathbf{p})$.

Proof. Let $q_i = 2^{-l_i}$. We have $l_i = -\log_2 2^{-l_i}$, and due to Fact 1 $\sum_i p_i = \sum_i q_i$. Hence

$$\text{Huffman}(\mathbf{p}) = \sum_i p_i l_i = - \sum_i p_i \cdot \log_2 q_i \stackrel{\text{property}}{\geq} - \sum_i p_i \cdot \log_2 q_i = \text{Entropy}(\mathbf{p}).$$

Fact 3. $\text{Huffman}(\mathbf{p}) \leq \text{Entropy}(\mathbf{p}) + 1$.

Proof Let $l_i = \lceil -\log_2 p_i \rceil$. Then, $2^{-l_i} \leq p_i$.

Hence, $\sum_i 2^{-l_i} \leq \sum_i p_i = 1$, and due to Fact 1, there is a binary tree (not necessarily full) with depths of leaves l_1, l_2, \dots, l_n . The average path length in such a tree is

$$\begin{aligned} \sum_i p_i \cdot l_i &= \sum_i p_i \cdot \lceil -\log_2 p_i \rceil \leq \sum_i p_i \cdot (-\log_2 p_i + 1) \\ &= - \sum_i p_i \cdot \log_2 p_i + \sum_i p_i = \text{Entropy}(\mathbf{p}) + 1. \end{aligned}$$

Consequently, there is a tree whose cost is at most $\text{Entropy}(\mathbf{p}) + 1$. However, the Huffman tree realises the minimum cost, which shows that we have also $\text{Huffman}(\mathbf{p}) \leq \text{Entropy}(\mathbf{p}) + 1$ as required. ■

Notes

The relation between Huffman trees and entropy is from [50].



134 Compressed pattern matching in Thue-Morse words

The Thue-Morse binary word on the alphabet $\{0, 1\}$ is produced by iterating infinitely from 0 the **Thue-Morse morphism** μ from $\{0, 1\}^*$ to itself defined by

$$\mu(0) = 01, \mu(1) = 10.$$

Eventually, the iteration produces the infinite Thue-Morse word:

$$\mathbf{t} = 01101001100101101001011001101001 \dots$$

For a pattern $x \in \{0, 1\}^*$ of even length, let $\mu^{-1}(x)$ be the word z for which $\mu(z) = x$ if it exists and *nil* otherwise. In other words $\mu^{-1}(x) \neq \text{nil}$ if $x \in \{01, 10\}^*$. We also introduce the set $EVEN = \{0110, 1010, 0101, 1001\}$.

Let us denote by $first_4(x)$ the prefix of x of length 4, if there is any, by $first(x)$ and $last(x)$ the first and last letters of x , respectively, and by \bar{s} the negation of a bit s ($\bar{0} = 1$ and $\bar{1} = 0$).

The following algorithm tests in linear time and in a very simple way if a finite binary pattern x is a factor of \mathbf{t} .

```

TEST( $x$  non-empty word)
1  if  $x = \text{nil}$  then
2      return FALSE
3  if  $|x| < 4$  then
4      return ( $x \neq 111$  and  $x \neq 000$ )
5  if  $first_4(x) \notin EVEN$  then
6       $x \leftarrow \overline{first(x)} \cdot x$ 
7  if  $|x|$  is odd then
8       $x \leftarrow x \cdot \overline{last(x)}$ 
9  return TEST( $\mu^{-1}(x)$ )

```

Question. Show why this algorithm correctly tests in linear time if x is a factor of the infinite Thue-Morse word \mathbf{t} .

Solution

First note that if $\mu^{-1}(x) \neq \text{nil}$ then $|\mu^{-1}(x)| = \frac{1}{2}|x|$, which implies the linear running time. The correctness is a consequence of three simple observations.

(i) The set $EVEN$ is the set of all length-4 words that occur in \mathbf{t} starting at even positions.

(ii) A nonempty word x of even length starts at an even position in \mathbf{t} if and only if $\mu^{-1}(x)$ occurs in \mathbf{t} .

(iii) If $|x| < 4$ then x is a factor of \mathbf{t} if and only if $x \neq 111$ and $x \neq 000$.

The algorithm checks if $x = ayb$ starts at an even position using (i). If x starts at odd position we add \bar{a} at the beginning of x . Now if the length of x becomes odd we add \bar{b} at the end. In this way, we slightly change x forcing x to occur at an even position and to have an even length. Then the resulted word x is a factor of \mathbf{t} if and only if $\mu^{-1}(x)$ is. The correctness follows from the observations above.

Notes

Equivalent definitions of the Thue-Morse word can be found in [40, Chapter 2] and [12, Chapter 1], for example.

The infinite Fibonacci word \mathbf{f} is generated by iterating, starting from \mathbf{a} , the morphism ϕ from $\{\mathbf{a}, \mathbf{b}\}^*$ to itself defined by $\phi(\mathbf{a}) = \mathbf{ab}$, $\phi(\mathbf{b}) = \mathbf{a}$.

Following the same strategy as above, we get a very simple algorithm testing if a nonempty binary word is a factor of \mathbf{f} .

```

TEST-FIB( $x$  non-empty word)
1  if  $x = nil$  then return FALSE
2  if  $|x| = 1$  then return TRUE
3  if  $x = by$  then  $x \leftarrow aby$ 
4  if  $x = yba$  then  $x \leftarrow yb$ 
5  if  $x = yaa$  then  $x \leftarrow xb$ 
6  return TEST-FIB( $\phi^{-1}(x)$ )

```

For example:

TEST-FIB(\mathbf{baa}) = TEST-FIB($\phi^{-1}(\mathbf{abaab})$) = TEST-FIB(\mathbf{aba})
= TEST-FIB($\phi^{-1}(\mathbf{ab})$) = TEST-FIB(\mathbf{a}) = TRUE.

However

TEST-FIB(\mathbf{baaa}) = TEST-FIB($\phi^{-1}(\mathbf{abaaab})$) = TEST-FIB(\mathbf{abba})
= TEST-FIB($\phi^{-1}(\mathbf{abb})$) = TEST-FIB(nil) = FALSE.

Thue-Morse and Fibonacci words are examples of morphic words. Testing a pattern in any morphic word is the subject of [11, Problem 68]. We showed special cases of pattern-matching in compressed texts. The fastest algorithm for general case, using *recompression technique*, was presented in [31].



135 Words representing combinatorial generations

There are many interesting strings related to combinatorial generations, their characteristic feature is usually *high compressibility*. Here we discuss permutation generations. Usually they produce each successive permutation by applying some kind of a *basic operation*. The sequence of these operations, corresponding to the permutation ordering, is called the *generating sequence*. It is treated as a word over the alphabet consisting of names of basic operations. The most interesting are cases when this alphabet is small. We present in detail three very similar generation sequences $\alpha_n, \beta_n, \lambda_n$, then in the notes we mention briefly two others.

Assume the permutation of numbers $\{1, 2, \dots, n\}$ is stored in the table π , with positions numbered from zero.

We consider compression in terms of straight-line programs. A straight-line program, briefly SLP, is a context-free grammar that produces a single word w over a given alphabet Σ .

An SLP can be also defined as a sequence of recurrences (equations), using operations of concatenation of words. Compression by straight-line programs is also called Grammar-Based Compression.

Question. Construct a generating sequence α_n which generates all n -permutations using operations of prefix reversals and can be described by an SLP of size $O(n \log n)$. Show also that each SLP of any generating sequence for n -permutations is of $\Omega(n \log n)$ size.

Solution

In this generation the alphabet of basic operations is $\Sigma_n = \{1, 2, \dots, n-1\}$. The symbol i corresponds to the basic operation: reverse the prefix $\pi[0..i]$ of permutation π .

We define the word α_n by recurrences

$$\alpha_2 = 1; \alpha_{n+1} = \alpha_n \cdot (n \cdot \alpha_n)^n \text{ for } n > 2. \quad ((1.1))$$

For example $\alpha_3 = 12121$, $\alpha_4 = 12121312121312121312121$

α_n is a generating sequence: starting from the id-permutation $\pi = (\pi_0, \pi_1, \pi_2, \dots, \pi_{n-1}) = (1, 2, \dots, n)$, and consecutively applying operations from α_n all n -permutations are generated, each exactly once.

Example. Recall that we number positions in permutations starting from 0, but the n -permutations consist of numbers $1, 2, \dots, n$.

We have $\alpha_3 = 12121$ and the generation of $\{1, 2, 3\}$ is:

$$123 \xrightarrow{1} 213 \xrightarrow{2} 312 \xrightarrow{1} 132 \xrightarrow{2} 231 \xrightarrow{1} 321$$

The generation of all 24 permutations of $\{1, 2, 3, 4\}$ has the following structure.

$$1234 \xrightarrow{\alpha_3} 3214 \xrightarrow{3} 4123 \xrightarrow{\alpha_3} 2143 \xrightarrow{3} 3412 \xrightarrow{\alpha_3} 1432 \xrightarrow{3} 2341 \xrightarrow{\alpha_3} 4321$$

It is not exactly an SLP because of exponents. However X^n can be rewritten as $O(\log n)$ -length SLP in a strict sense. Hence the total size of all expressions defining α_n , using only concatenation, is $O(n \log n)$.

Fact 1. If we start with $x_0 x_1 \cdots x_{n-1}$ then α_n generates all permutations of $\{x_0, \dots, x_{n-1}\}$ and ends in $x_{n-1} x_{n-2} \cdots x_0$.

Proof Assume it is true for n . We show it holds for $n+1$. First notice that, due to inductive assumption,

$$x_0 x_1 \cdots x_{n-1} x_n \xrightarrow{\alpha_n} x_{n-1} x_{n-2} \cdots x_0 x_n \xrightarrow{n} x_n x_0 x_1 \cdots x_{n-1}$$

Hence each $\alpha_n \cdot n$ produces the left cyclic shift and α_{n+1} works like n left cyclic shifts, followed by reversing the prefix of size n . After n left cyclic shifts we get $x_1 \cdots x_{n-1} x_n x_0$. Then the last α_n reverses $x_1 \cdots x_{n-1} x_n$ and we get $x_n x_{n-1} x_{n-2} \cdots x_0$. This completes the proof. ■

An SLP of size k can generate only words of single exponential size $N = O(2^k)$, consequently $k = \Omega(\log N)$. We have $|\alpha_n| = n! - 1$, hence in this case $k = \Omega(\log n!)$, which is $\Omega(n \log n)$.

Modified reversing of prefixes. Now our basic operation $\mathbf{R}(k)$ (k , in short) consists in reversing a prefix of size k and moving it to the end of the word. In other words, if $x = uv$, $|u| = k$, then $\mathbf{R}(k)(x) = vu^R$.

For example $(1, 2, 3, 4, 5, 6) \xrightarrow{3} (4, 5, 6, 3, 2, 1)$. A permutation generator using operations $\mathbf{R}(k)$ corresponds to another compactly described generating sequences.

Question. Write a compact representation of the permutations generator using modified prefix reverses.

Solution

We construct the generation sequence $\beta_n = (k_1, k_2, k_3, \dots, k_m)$ of identifiers of actions $\mathbf{R}(k)$.

The word β_n can be defined by a recurrence,

$$\beta_2 = 1, \quad \beta_{n+1} = 1^n \prod_{i=1}^m ((a_i + 1) \cdot 1^n) \text{ where } a_1 a_2 \cdots a_m = \beta_n. \quad ((1.2))$$

Example. We have,

$$\beta_2 = 1, \quad \beta_3 = 11 \ 2 \ 11, \quad \beta_4 = 111 \ \mathbf{2} \ 111 \ \mathbf{2} \ 111 \ \mathbf{3} \ 111 \ \mathbf{2} \ 111 \ \mathbf{2} \ 111.$$

For $n = 3$ the output (sequence of generated permutations) is

$$\boxed{123} \xrightarrow{1} \boxed{231} \xrightarrow{1} \boxed{312} \xrightarrow{2} \boxed{213} \xrightarrow{1} \boxed{132} \xrightarrow{1} \boxed{321}.$$

For $n = 4$, the generation is:

$$\begin{array}{l} \boxed{123} \xrightarrow{4} 2341 \xrightarrow{1} 3412 \xrightarrow{1} 4123 \xrightarrow{2} \boxed{231} \xrightarrow{4} 3142 \xrightarrow{1} 1423 \xrightarrow{1} 4231 \xrightarrow{2} \\ \boxed{312} \xrightarrow{4} 1243 \xrightarrow{1} 2431 \xrightarrow{1} 4312 \xrightarrow{3} \boxed{213} \xrightarrow{4} 1342 \xrightarrow{1} 3421 \xrightarrow{1} 4213 \xrightarrow{2} \\ \boxed{132} \xrightarrow{4} 3241 \xrightarrow{1} 2413 \xrightarrow{1} 4132 \xrightarrow{2} \boxed{321} \xrightarrow{4} 2143 \xrightarrow{1} 1432 \xrightarrow{1} 4321 \end{array}$$

Observe how the sequence for $n = 4$ results from the sequence for $n = 3$ of boxed fragments in a recursive way.

If we replace α by β then Fact 1 remains true and correctness proof for the sequence β is very similar to that for α . The sequence β_n corresponds to the iterative algorithm C, which Knuth in his 4-th volume of “The art of computer programming” called “the simplest permutation generator of all”. We refer to Knuth’s book for correctness of the algorithm C, which also implies correctness of β .

Remark. An iterative generation with function $\mathbf{R}(k)$ is exceptionally simple. For a permutation x of $\{1, 2, \dots, n\}$ we compute the length k of the shortest prefix of x which is not a prefix of $n, n-1, n-2, \dots, 2, 1$. Then we perform $\mathbf{R}(k)$ on x to get the next permutation.

Question. Construct a compactly represented sequences λ_n of transpositions which are permutations generators, such that λ_n is a prefix of λ_{n+1} .

Solution

We construct sequences λ corresponding to Heap’s algorithm. Let $\langle i, j \rangle$ represent a transposition $exchange(x[i], x[j])$. The permutations are stored as $x[0, 1, \dots, n-1]$, where n is the number of elements. Define the words w_n , each of size $n-1$, for $0 \leq i \leq n-2$, as:

$$w_n[i] = \begin{cases} \langle 0, n-1 \rangle & \text{if } n \text{ is odd} \\ \langle i, n-1 \rangle & \text{if } n \text{ is even} \end{cases}$$

Then Heap’s algorithm [28] corresponds to the sequence λ_n of basic operations defined as follows

$$\lambda_2 = \langle 0, 1 \rangle; \quad \lambda_{n+1} = \lambda_n \cdot \prod_{i=0}^{n-1} (w_n[i] \cdot \lambda_n) \text{ for } n > 2. \quad ((1.3))$$

Example. We have: $\lambda_3 = \lambda_2 (\langle 0, 2 \rangle \lambda_2)^2 = \langle 0, 1 \rangle \langle 0, 2 \rangle \langle 0, 1 \rangle \langle 0, 2 \rangle \langle 0, 1 \rangle$.

$\lambda_4 = \lambda_3 \langle 0, 3 \rangle \lambda_3 \langle 1, 3 \rangle, \lambda_3 \langle 2, 3 \rangle \lambda_3. \lambda_5 = \lambda_4 (\langle 0, 4 \rangle \lambda_4)^4$.

Correctness of λ_n follows easily from the property:

- After performing λ_n starting with the permutation $1, 2, \dots, n$ we generate each permutation exactly once, and finish with $n, 2, 3, 4, \dots, n-1, 1$, if n is even, and with $2, 3, 4, \dots, n-1, 1$, if n is odd.

Notes

Our presentation of reversing-prefixes algorithm follows the Zaks algorithm [55] for permutation generation. The recurrences were given originally in terms of suffixes, but it is essentially equivalent to taking prefixes. In [49] the same permutation generating sequence α_n was described by a greedy algorithm. Each sequence α_n as a prefix of size $n! - 1$ of the sequence $\rho = (\rho_1, \rho_2, \rho_3, \dots)$, where

$$\rho_k = \max\{j : j! \text{ is a divisor of } k\}, \text{ for } k \geq 1.$$

We have $\rho = 1\ 2\ 1\ 2\ 1\ 3\ 1\ 2\ 1\ 2\ 1\ 3\ 1\ 2\ \dots$. ρ_n is the sequence of values of so called *factorial ruler* function. ρ_n can be also generated on-line using extra memory of size $O(n)$ in the following way.

Assume we know the *factorial representation* of $n - 1$:

$$n - 1 = a_1 \cdot 1 + a_2 \cdot 2! + a_3 \cdot 3! + \dots + a_m \cdot m!,$$

where $0 \leq a_i \leq i!$ for each i and $a_m \neq 0$. Then ρ_n equals the first position k such that $a_k < k$ or $k = m + 1$. We obtain factorial representation of n by increasing a_k by 1 and set $a_i = 0$ for all $i < k$.

Ehrlich algorithm. Gideon Ehrlich devised in [17] a tricky version of Zaks algorithm, this time the operation i corresponds to the transposition $x_0 \leftrightarrow x_i$, also called “star transposition”. We cite D. Knuth, as he has written in his 4-th volume of “The Art of Computer Programming”, fascicle 2, page 57: “The most amazing thing about this algorithm ... is that it works.” Knuth in his book also gives a sketch of correctness proof (exercise 55). We present here our stringologic version of Ehrlich algorithm showing its remarkable similarity to Zaks algorithm.

Assume \odot is the composition of functions from left to right, and $Shift_k(x)$ moves the k 'th element of x to the beginning of x . The sequence γ_n of *star transpositions* generating n -permutations is compactly represented, using morphisms h_n , as

$$\gamma_2 = 1, \quad \gamma_{n+1} = \gamma_n \prod_{i=1}^n (n h_n^i(\gamma_n)), \text{ for } n > 2, \quad ((1.4))$$

where $h_2 = \text{Identity} = [1, 2, 3, \dots, \infty]$, $h_{n+1} = h_n^{n+1} \odot Shift_n$

Example.

$$\gamma_4 = \gamma_3 h_3(\gamma_3) \gamma_3 h_3^2(\gamma_3) \gamma_3 h_3^3(\gamma_3) \gamma_3$$

$$\gamma_3 = 12121, \quad \gamma_4 = 12121\mathbf{3}21212\mathbf{3}12121\mathbf{3}21212.$$

$$\gamma_5 = 12121\mathbf{3}21212\mathbf{3}12121\mathbf{3}21212\mathbf{4}3131\mathbf{3}2....$$

$$\gamma_5 = \gamma_4 4 h_4(\gamma_4) 4 h_4^2(\gamma_4) 4 h_4^3(\gamma_4) 4 h_4^4(\gamma_4).$$

The morphism h_n involves only letters $1, 2, \dots, n-1$. Below we list the first few morphisms:

$$\begin{aligned} h_2 &= [1], \quad h_3 = [2, 1], \quad h_4 = [3, 1, 2], \quad h_5 = [4, 2, 3, 1,], \\ h_6 &= [5, 1, 2, 3, 4], \quad h_7 = [6, 4, 5, 1, 2, 3], \quad h_8 = [7, 3, 1, 2, 6, 4, 5], \\ h_9 &= [8, 5, 1, 7, 3, 4, 2, 6, 9...]. \end{aligned}$$

Steinhaus-Trotter-Johnson algorithm. In this algorithm we generate recursively all $(n-1)$ -permutations of $012\dots n-2$, then for each of them we insert the element $n-1$ in all possible places, traversing the $(n-1)$ -permutation alternately right-to-left or left-to-right.

Assume each permutation is a sequence $x[0], x[1] \dots x[n-1]$. The alphabet of basic actions is $\{0, 1, 2, \dots, n-2\}$. In this case the i -th action is "exchange $x[i]$ with $x[i+1]$ ". Denote by δ_n the corresponding sequence of basic operations generating n -permutations. We have $\delta_2 = 0$.

Define $w_n = 0, 1, 2, \dots, (n-2)$. If $n > 2$ and $\delta_{n-1} = a_1 a_2 \dots a_N$ then the sequence δ_n of actions is a word of length $n! - 1$ defined as

$$\delta_n = w_n^R (a_1 + 1) w_n a_2 w_n^R (a_3 + 1) w_n a_4 \dots a_{N-1} w_n^R (a_N + 1) w_n,$$

Example. $\delta_3 = 10101, \quad \delta_4 = w_4^R \mathbf{2} w_4 \mathbf{0} w_4^R \mathbf{2} w_4 \mathbf{0} w_4^R \mathbf{2} w_4.$

136 Algorithm for 2-Anticovers

A 2-anticover of a word x is a set of pairwise distinct factors of x of length 2 that cover the whole word. The notion is dual of the notion of a cover, for which a unique factor (or a finite number of them) covers the whole word. The duality is similar to that of powers and antipowers, where the word is a concatenation of the same factor or of distinct factors. Instead, for anticovers or covers the occurrences factors can overlap or just be adjacent.

Example. The set $\{\mathbf{ab}, \mathbf{aa}, \mathbf{ac}, \mathbf{ba}, \mathbf{cc}, \mathbf{ca}\}$ is a 2-anticover of the word **abaacbacca**.

a b a a c b a c c a
 └─┘ └─┘ └─┘ └─┘

Note the word **abaababbaab** has no 2-anticover because **ab** is both a prefix and a suffix of it.

The notion generalises obviously to k -anticover and, for example, the word **abaababbaa** admits the 3-anticover $\{\mathbf{aba}, \mathbf{aab}, \mathbf{bab}, \mathbf{baa}\}$:

a b a a b a b b a a
 └─┘ └─┘ └─┘

On an alphabet of size σ , since the number of words of length k is σ^k , no word of length larger than $k\sigma^k$ admits a k -anticover. This is why it is appropriate to consider an integer alphabet that is potentially infinite.

Question. Design a linear-time algorithm testing if the word x admits a 2-anticover, assume its alphabet is sortable in linear time (integer alphabet).

[**Hint:** Use a linear-time algorithm for the satisfiability of 2CNF formulas (CNF = conjunctive normal form). Each 2CNF formula is a conjunction of two-variable “clauses” (alternatives of variables and their negations). An example of a 2CNF formula is: $(v_2 \vee v_4) \wedge (v_1 \vee \neg v_3)$.]

Solution

As clauses we use also formulas of the type $(a \rightarrow b)$ because it is equivalent to $(\neg a \vee b)$.

For a set of Boolean variables $V = \{v_1, v_2, \dots, v_m\}$, we introduce the predicate

$$\Delta(V) \equiv |\{i : v_i = 1\}| \leq 1$$

and use the following fact.

Fact 1. The predicate $\Delta(V)$ can be written as an equivalent 2CNF formula of size $O(m)$ for $V = \{v_1, v_2, \dots, v_m\}$.

Proof We introduce variables α_i and β_i that are to be interpreted as

$$\alpha_i \equiv (\forall t \leq i \ v_t = \text{FALSE}) \text{ and } \beta_i \equiv (\forall t \geq i \ v_t = \text{FALSE}).$$

Then, Δ can be written as the following conjunction of implications:

$$\begin{aligned} & \forall i < m \ (v_i \rightarrow \beta_{i+1}) \wedge (\beta_i \rightarrow \beta_{i+1}) \\ & \wedge \forall i > 1 \ (v_i \rightarrow \alpha_{i-1}) \wedge (\alpha_i \rightarrow \alpha_{i-1}) \\ & \wedge \forall 1 \leq i \leq m \ (\alpha_i \rightarrow \neg v_i) \wedge (\beta_i \rightarrow \neg v_i). \end{aligned}$$

Consequently, $\Delta(v_1, \dots, v_m)$ is equivalent to a conjunction of $O(m)$ implications. ■

Construction of a 2-anticover. Let $x = a_1 a_2 \dots a_n$ (a_i letters) and $\text{Fact}_2(x)$ be the set of factors of length 2 of x . Let $\text{Occ}(v, x)$ denote the set of starting positions of occurrences of v in x .

We consider the Boolean variables x_i whose value is TRUE iff $a_i a_{i+1}$ is an element of our anti-cover. Now the problem reduces to the satisfiability of the 2CNF formula:

$$\begin{aligned} & \forall 1 < i < n \ (x_i \vee x_{i-1}) \wedge (x_1 \wedge x_{n-1}) \\ & \wedge \forall v \in \text{Fact}_2(w) \ \Delta(\{x_i : i \in \text{Occ}(v, x)\}). \end{aligned}$$

Then, for each i , $1 < i < n$ and $x_i = \text{TRUE}$, we choose $a_i a_{i+1}$ as an element of the 2-anticover. Otherwise we choose $a_{i-1} a_i$. We have also to take $a_1 a_2$ and $a_{n-1} a_n$ (hence $x_1 \wedge x_{n-1}$).

The second part of the formula says that each factor of length 2 is chosen at most once as a fragment in the 2-anticover.

To conclude, the word admits a 2-anticover if and only if the formula is true, which answers the question.

Notes

We briefly sketch a linear-time algorithm testing 2CNF satisfiability. Let V be the set of variables and their negations. We change the problem to a set of implications of type $A \rightarrow B$, where $A, B \in V$. Each implication can be viewed as a directed edge in the graph G whose V is the set of nodes. Then the formula is not satisfiable if and only if, for some variable v , both v and $\neg v$ are in the same strongly connected component. The strongly connected components of a graph can be computed in linear time.

The k -anticover was introduced in [2], where the above result is proved and it is shown that the 3-anticover problem is NP-complete.

Some algorithms related to covers are the subject of Problems 20 and 45 in [12]. Problem 90 deals with antipowers, see also [3].

137 Short Supersequence of Shapes of Permutations

An n -permutation is a length- n sequence (or word) of n distinct elements from $\{1, 2, \dots, n\}$. The aim of the problem is to build a short word \mathbf{S}_n , called a **superpattern** (supersequence of shapes), such that each n -permutation is order-equivalent to a subsequence of \mathbf{S}_n . The question is similar to finding a short supersequence but the order-preserving feature reduces drastically the length of the searched word. Indeed, the superpattern defined below has length $|\mathbf{S}_n| = (n^2 + n)/2$, which is almost half the length $n^2 - 2n + 4$ of the supersequence constructed in [12, Problem 15].

The word \mathbf{S}_n is drawn from the alphabet $\{1, 2, \dots, n+1\}$ as follows. Let α_n be the increasing sequence of all odd letters and β_n be the decreasing sequence of all even letters of the alphabet (α_n is an “ascending group” and β_n is a “descending group”). Alternation between ascending and descending groups is the main trick of the solution. Then, define,

$$\mathbf{S}_n = \begin{cases} (\alpha_n \beta_n)^{n/2} & \text{if } n \text{ is even,} \\ (\alpha_n \beta_n)^{\lfloor n/2 \rfloor} \alpha_n & \text{otherwise.} \end{cases}$$

Example. With $n = 8$, $\alpha_8 = 1\,3\,5\,7\,9$, $\beta_8 = 8\,6\,4\,2$ and

$\mathbf{S}_8 = 1\,3\,5\,7\,9\,8\,6\,4\,2\,1\,3\,5\,7\,9\,8\,6\,4\,2\,1\,3\,5\,7\,9\,8\,6\,4\,2$.

With $n = 7$, $\alpha_7 = 1\,3\,5\,7$, $\beta_7 = 8\,6\,4\,2$ and

$\mathbf{S}_7 = 1\,3\,5\,7\,8\,6\,4\,2\,1\,3\,5\,7\,8\,6\,4\,2\,1\,3\,5\,7$.

For a permutation $\pi = (\pi_1, \pi_2, \dots, \pi_n)$ of $\{1, 2, \dots, n\}$, let π^+ denote $(\pi_1 + 1, \pi_2 + 1, \dots, \pi_n + 1)$, permutation of $\{2, 3, \dots, n+1\}$.

An embedding of π in a word \mathbf{S} is an increasing sequence of positions (p_1, p_2, \dots, p_n) on \mathbf{S} that satisfies $\pi = \mathbf{S}[p_1]\mathbf{S}[p_2] \cdots \mathbf{S}[p_n]$.

Question. Show how to compute in linear time an order-preserving embedding of a given n -permutation π into \mathbf{S}_n .

[**Hint:** Show that π or π^+ is a (standard) subsequence of \mathbf{S}_n and can be found by a greedy algorithm. Note that π^+ is order equivalent to π .]

Solution

Following the hint, we show that π or π^+ is a subsequence of \mathbf{S}_n . To do it, we proceed indirectly as follows. We show that π and π^+ are subsequences of prefixes of lengths m_1 and m_2 , respectively, of an infinite word \mathbf{S} , where $m_1 + m_2 \leq 2|\mathbf{S}_n|$. Then, since \mathbf{S}_n is a prefix of \mathbf{S} , one of π or π^+ is a subsequence of \mathbf{S}_n .

Let $\mathbf{S} = (\alpha_n \beta_n)^\infty$. The positions on \mathbf{S} are numbered from 1 and are

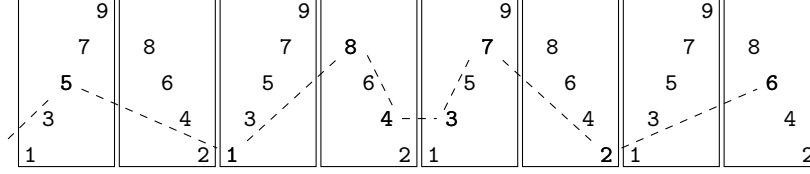


Figure 1.1 The route showing how Algorithm GREEDY processes the permutation $\pi = (5, 1, 8, 4, 3, 7, 2, 6)$, by successive jumps to the first next appropriate group. The sequence of jumps is 1, 2, 1, 0, 1, 0, 1, 2, for a total $\text{Jumps}(\pi) = 8$. The output is $(p_1, p_2, \dots, p_8) = (3, 10, 15, 20, 22, 27, 34)$

partitioned into consecutive disjoint intervals of alternative sizes $|\alpha_n|$ and $|\beta_n|$, called a groups. \mathbf{S}_n is the prefix of \mathbf{S} whose indices consist of n groups.

Let $\text{group}(j)$ be the number of the group containing j if $j > 0$, and set $\text{group}(0) = 0$.

From an n -permutation π , Algorithm GREEDY computes in a greedy manner an embedding (p_1, p_2, \dots, p_n) of π in \mathbf{S} .

```

GREEDY( $\pi = (\pi_1, \pi_2, \dots, \pi_n)$ , length- $n$  permutation)
1   $p_0 \leftarrow 0$ 
2  for  $i \leftarrow 1$  to  $n$  do
3       $p_i \leftarrow \min\{j > p_{i-1} : \pi_i = \mathbf{S}[j]\}$ 
4       $\text{jump}_\pi(i-1) \leftarrow \text{group}(p_i) - \text{group}(p_{i-1})$ 
5  return  $(p_1, p_2, \dots, p_n)$ 

```

Observation. Variable jump and the instruction at line 4 Note that $\text{jump}_\pi(i-1) \in \{0, 1, 2\}$ (see figure).

The algorithm is said to be *successful* if $p_n \leq |\mathbf{S}_n|$, which means that $\pi = \mathbf{S}_n[p_1 \dots p_n]$ because \mathbf{S}_n is a prefix of \mathbf{S} .

Example (followed). For $n = 8$ and $\pi = (2, 4, 6, 8, 4, 3, 2, 1)$ the algorithm is unsuccessful, but is not for $\pi^+ = (3, 5, 7, 9, 5, 4, 3, 2)$ since it returns $(2, 3, 4, 5, 12, 17, 20, 27)$ and $27 \leq |\mathbf{S}_n| = 36$.

The property has an equivalent formulation in terms of jumps. Let $\text{Jumps}(\pi)$ be the sum of jumps: $\text{jump}_\pi(0) + \text{jump}_\pi(1) + \dots + \text{jump}_\pi(n-1)$. Then, we get the following fact to characterise a success.

Fact 1. $p_n \leq |\mathbf{S}_n| \Leftrightarrow \text{Jumps}(\pi) \leq n$.

Since π and π^+ are obviously order-equivalent, it is enough to prove that Algorithm GREEDY is successful for at least one of π and π^+ . This amounts to show that π or π^+ is a (standard) subsequence of \mathbf{S}_n as computed by GREEDY.

Example. Let $n = 8$. For $\pi = (1, 2, 3, 4, 5, 6, 7, 8)$, $\text{Jumps}(\pi) = 8$ and $\text{Jumps}(\pi^+) = 9$. For $\pi = (2, 4, 6, 8, 7, 5, 3, 1)$, $\text{Jumps}(\pi) = 15$ and $\text{Jumps}(\pi^+) = 2$. In both cases $\text{Jumps}(\pi) + \text{Jumps}(\pi^+) = 2n + 1$, which is not accidental, and is a key point to correctness.

```

EMBEDDING( $\pi = (\pi_1, \pi_2, \dots, \pi_n)$   $n$ -permutation)
1   $(p_1, p_2, \dots, p_n) \leftarrow \text{GREEDY}(\pi)$ 
2  if  $p_n > |\mathbf{S}_n|$  then
3      return  $\text{GREEDY}(\pi^+)$ 
4  return  $(p_1, p_2, \dots, p_n)$ 

```

The proof of correctness of Algorithm EMBEDDING reduces to the following statement whose proof is after the observation.

Fact 2. $\text{Jumps}(\pi) + \text{Jumps}(\pi^+) = 2n + 1$. Hence, for an n -permutation π , either $\text{Jumps}(\pi) \leq n$ or $\text{Jumps}(\pi^+) \leq n$ and Algorithm GREEDY is successful for π or for π^+ .

Observation. When π_i and π_{i+1} are both even, p_i, p_{i+1} belong to descending groups. In this case, if $\pi_i > \pi_{i+1}$ then $\text{jump}(i) = 0$ else $\text{jump}(i) = 2$. Symmetrically, when they are both odd, they belong to the same ascending group and if $\pi_i > \pi_{i+1}$ then $\text{jump}(i) = 2$ else $\text{jump}(i) = 0$. When π_i and π_{i+1} are of distinct parities, $\text{jump}(i) = 1$.

Proof Let BothEven, BothOdd, Dif be the set of $i < n$ for which respectively both π_i, π_{i+1} are even, both are odd and they are of different parities. We introduce the sets:

$\mathbf{A}_{\text{even}} = \{0 < i < n : \pi_i < \pi_{i+1} \text{ and } i \in \text{BothEven}\},$
 $\mathbf{D}_{\text{even}} = \{0 < i < n : \pi_i > \pi_{i+1} \text{ and } i \in \text{BothEven}\},$
 $\mathbf{A}_{\text{odd}} = \{0 < i < n : \pi_i < \pi_{i+1} \text{ and } i \in \text{BothOdd}\},$
 $\mathbf{D}_{\text{odd}} = \{0 < i < n : \pi_i > \pi_{i+1} \text{ and } i \in \text{BothOdd}\}.$

For $0 < i < n$, we have

$$(\text{jump}_\pi(i), \text{jump}_{\pi^+}(i)) = \begin{cases} (0, 2) & \text{if } i \in \mathbf{A}_{\text{odd}} \cup \mathbf{D}_{\text{even}}, \\ (2, 0) & \text{if } i \in \mathbf{A}_{\text{even}} \cup \mathbf{D}_{\text{odd}}, \\ (1, 1) & \text{if } i \in \text{Dif}. \end{cases}$$

Hence, for $0 < i < n$, $\text{jump}_\pi(i) + \text{jump}_{\pi^+}(i) = 2$. This, together with equation $\text{jump}_\pi(0) + \text{jump}_{\pi^+}(0) = 3$, implies

$$\text{Jumps}(\pi) + \text{Jumps}(\pi^+) = 2(n - 1) + 3 = 2n + 1,$$

which completes the proof. ■

Notes

The present construction is adapted from the version in [43]. If the conjecture that the shortest superpattern has length $\frac{1}{2}n^2(1 + o(n))$ held, this would imply that our construction is asymptotically optimal.

138 Computing short distinguishing subsequence

The problem considers **distinguishing subsequences** between two different binary words (see [12, Problem 51]). Denoting by $\text{Subs}(x)$ the set of subsequences of a word x , a word z is said to distinguish x and y , $x \neq y$, if it is a subsequence of only one of them, that is, $z \in \text{Subs}(x) \Leftrightarrow z \notin \text{Subs}(y)$.

Question. Construct a distinguishing subsequence of length at most $\lceil (n+1)/2 \rceil$ for two distinct binary words of the same length n .

In fact, the above bound is optimal.

Question. For each $n > 0$, construct two distinct binary words of length n that do not have a distinguishing subsequence of length smaller than $\lceil (n+1)/2 \rceil$.

Solution

Let $\{\mathbf{a}, \mathbf{b}\}$ be the alphabet of the different words x and y of length n . First note that if the words have different numbers of occurrences of \mathbf{a} (or of \mathbf{b}) then both \mathbf{a}^k and \mathbf{b}^ℓ are distinguishing subsequences for some integers k and ℓ . Choosing the shorter answers the question. We then assume that the words have the same number of occurrences of \mathbf{a} (and then of \mathbf{b}).

For a word w and a natural number k , denote by $\text{pos}(w, k)$ the position on w of the k -th occurrence of \mathbf{b} if it exists. If not (when $w \in \mathbf{a}^*$ for example) $\text{pos}(w, k) = |w|$. Let

$$i = \min\{k : \text{pos}(x, k) \neq \text{pos}(y, k)\},$$

which is well defined because $x \neq y$ and at least one of them have occurrences of \mathbf{b} . We later assume w.l.o.g. that $\text{pos}(x, i) < \text{pos}(y, i)$.

Let $x = x_1 \cdot \mathbf{b} \cdot x_2$, where

$$x_1 = x[0 \dots \text{pos}(x, i) - 1] \text{ and } x_2 = x[\text{pos}(x, i) + 1 \dots n - 1]$$

and let z_1 and z_2 be the two sequences defined by

$$z_1 = \text{erase}(x_1, \mathbf{b}) \cdot \mathbf{ab} \cdot \text{erase}(x_2, \mathbf{a}),$$

$$z_2 = \text{erase}(x_1, \mathbf{a}) \cdot \mathbf{b} \cdot \text{erase}(x_2, \mathbf{b}),$$

where, for a word w and a symbol c , $\text{erase}(w, c)$ denotes the word resulting from w by erasing all the occurrences of letter c in it.

It is clear that z_1 and z_2 are both distinguishing subsequences for x and y . Additionally, since $|z_1| + |z_2| = n + 2$, at least one of the two subsequences is of length at most $\lceil (n + 1)/2 \rceil$.

Example 1. For $x = \text{ababababab}$ and $y = \text{ababaababb}$ of length 10, we have $i = 3$, $\text{pos}(x, 3) = 5$, $x_1 = \text{ababa}$, $x_2 = \text{abab}$. Eventually, we get the two distinguishing subsequences $z_1 = \text{aaa} \cdot \text{ab} \cdot \text{bb}$ and $z_2 = \text{bb} \cdot \text{b} \cdot \text{aa}$. The second has length $5 < \lceil (10 + 1)/2 \rceil$.

Example 2. For $x = \text{abababababa}$ and $y = \text{ababaaabbba}$ of length 11, we have $i = 3$, $\text{pos}(x, 3) = 5$, $x_1 = \text{ababa}$, $x_2 = \text{ababa}$. Eventually, we get the two distinguishing subsequences $z_1 = \text{aaa} \cdot \text{ab} \cdot \text{bb}$ and $z_2 = \text{bb} \cdot \text{b} \cdot \text{aaa}$. The second has length $6 = \lceil (11 + 1)/2 \rceil$.

Optimal bound. Let $n = 2m$, $x = (\text{ab})^m$ and $y = (\text{ba})^m$. Then, any binary word of length m is a subsequence of each of these two different words. Hence, they have a shortest distinguishing subsequence of length exactly $m + 1 = \lceil (n + 1)/2 \rceil$, for example, $\text{a}^m \text{b}$.

For $n = 2m + 1$ we can choose $x' = x \cdot \text{a}$ and $y' = y \cdot \text{a}$, for which ba^m is a shortest distinguishing subsequence of the expected length.

Notes

A standard solution to compute a shortest distinguishing subsequence of two words is a by-product of testing the equivalence of their minimal (deterministic) subsequence automata (see [12, Problem 51]) as an application of the UNION-FIND data structure, see [1].

There is a linear-time algorithm computing a shortest distinguishing subsequence of two different words. Such an algorithm was first announced by Imre Simon, but is has not been published by him. The first (quite complicated) published linear-time algorithm for this problem is by Gawrychowski et al. [24].



139 Yet another application of Suffix trees

In this problem, we show how the Suffix tree of a word can be used in three different ways to solve an example problem. For a string x , let $Sub[k]$ denote the number of (distinct) nonempty factors of x having an occurrence whose position starts in the interval $[0..k]$. For simplicity, assume that x ends with a unique symbol.

Question. Show how to compute the table $Sub[0..n-1]$ in linear time.

[Hint: Use a suffix tree.]

Solution

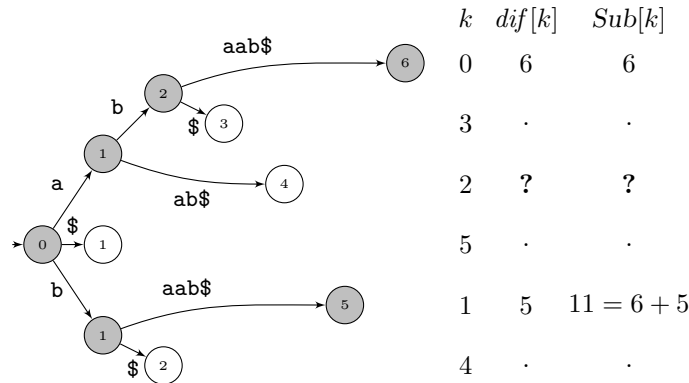
To compute the table Sub , it is enough to compute, for each position $k > 0$, the number $dif[k]$ of factors that start at k but not before.

TABLESUB(x word of length n)

```

1   $dif \leftarrow \text{TABLEDIF}(x)$ 
2  for  $k \leftarrow 0$  to  $n - 1$  do
3       $Sub[k] \leftarrow$  if  $k = 0$  then  $dif[k]$  else  $Sub[k - 1] + dif[k]$ 
4  return  $Sub$ 
```

Computing the table dif can be done in various ways.



Let $\mathcal{ST}(x)$ be the Suffix tree of $x = x[0..n-1]$ (see Notes). Recall that a node of $\mathcal{ST}(x)$ is (or can be identified with) a factor u of x . In the picture, each branching node (explicit node or fork) u displays $|u|$. The weight of an edge $u \rightarrow v$ in $\mathcal{ST}(x)$ is the absolute difference between the lengths of its end-nodes, that is, $|v| - |u|$. Each leaf is a non-empty suffix v and is labeled by its starting position on x , that is, $|x| - |v|$.

Algorithm 1. The picture illustrates a step in a run of Algorithm TABLEDIF1, just before processing the suffix **aab\$** of the word **abaab\$**. All nodes on the two longest branches are marked following the computation of $dif[0]$ and of $dif[1]$. Since the parent of leaf 2 is marked, $dif[2] = |ab\$| = 3$ and $Sub[2] = 11 + 3 = 14$.

```

TABLEDIF1( $x$  word of length  $n$ )
1  unmark all nodes of  $\mathcal{ST}(x)$ 
2  for  $k \leftarrow 0$  to  $n - 1$  do
3      from leaf  $k$ , go bottom-up until meeting a marked node
4      mark all visited nodes
5       $dif[k] \leftarrow$  sum of weights of visited edges
6  return  $dif$ 

```

Algorithm 2. Instead of running through all suffixes (with variable k), the algorithm below processes nodes in any order. But to do so, it recovers suffixes with the value $\min(v)$, minimum leaf in the subtree rooted at node v . The algorithm is as follows.

```

TABLEDIF2( $x$  word of length  $n$ )
1  for  $k \leftarrow 0$  to  $n - 1$  do
2       $dif[k] \leftarrow 0$ 
3  compute bottom-up  $\min(v)$  for each node  $v$  of  $\mathcal{ST}(x)$ 
4  for each non-root node  $v$  do
5       $(k, u) \leftarrow (\min(v), \text{parent of } v)$ 
6       $dif[k] \leftarrow dif[k] + |v| - |u|$ 
7  return  $dif$ 

```

Algorithm 3. The table dif can be computed during the construction of $\mathcal{ST}(x)$ by McCreight algorithm, which combined features in algorithms 1 and 2. Indeed, this algorithm adds exactly one edge at each iteration on suffix k . The weight of the edge is then added to $dif[k]$.

Notes

The Suffix tree of a word is described in [12, Chapter 1] and in references cited in its notes. McCreight algorithm for its construction is given in [15] and in [11, Section 5.2].

140 Two longest subsequence problems

There are many problems related to subsequences with specific properties (see the notes). We consider two simple problems of this type: for a word x , compute its lexicographically smallest subsequence of a given length k , and a longest palindromic subsequence.

The MinSub problem. For a word x drawn from an ordered alphabet, $\text{MinSub}(x, k)$ is defined as the lexicographically smallest subsequence of a given length k , $k \leq |x|$. For example, $\text{MinSub}(\text{bbbbbaeeecffddd}, 5) = \text{acddd}$. Note the subsequence may have several occurrences in x .

Question. For a word x of length n , design an algorithm that computes $\text{MinSub}(x, k)$ in time $O(n)$.

The LPS problem. In this second problem, the goal is to compute a longest palindromic subsequence $\text{LPS}(x)$ of the word x . For example, *abba* and *dccd* are possible answers for $\text{LPS}(\text{dcabcdba})$.

Question. Compute a longest palindromic subsequence of a word of length n in time $O(n^2)$.

Solution

The solution to the first problem is known as a folklore due to its simplicity, which is its main interest. Algorithm `MINSUB` is a modification of a very simple algorithm computing a lexicographically minimal subsequence.

It uses a stack handled with standard operations *top*, *pop* and *push*.

```

MINSUB( $x$  word of length  $n$ , integer  $k \leq n$ )
1   $rest \leftarrow |x| - k$ 
2   $S \leftarrow$  empty stack
3  for each letter  $a$  of  $x$ , sequentially do
4      while  $S$  non empty and  $a < \text{top}(S)$  and  $rest > 0$  do
5           $\text{pop}(S)$ 
6           $rest \leftarrow rest - 1$ 
7       $\text{push}(a, S)$ 
8  return  $S$ 

```

The variable *rest* takes care of the required length k . If there are not enough unread letters, the algorithm stops comparisons and adds to the stack the remaining unread symbols. In particular, if $k = |x|$ the

algorithm pushes to the stack the whole word x .

Correctness and complexity of the algorithm are straightforward.

Example. $\text{MinSub}(\text{baddbccega}, 7) = \text{abccega}$, because after reading the subsequence **ab** all remaining letters should be appended to get a word of length 7.

Solution to the second question. To compute an $\text{LPS}(x)$, the naive approach is to take $\text{LCS}(x, x^R)$. However, it does not work. For example, **abcd** (as one of possible answers) is an $\text{LCS}(\text{dcabdcba}, (\text{dcabdcba})^R)$ but is not a palindrome.

The problem **LMPS** (longest mutually palindromic subsequences) refines the above approach. $\text{LMPS}(w)$ returns two longest subsequences u and v (possibly the same) of w that satisfy $u = v^R$, together with their locations. In other words we look for the longest word y such that y and y^R occur (it could be the same occurrence if y is a palindrome) as subsequences of a given word w .

Formally, $\text{LMPS}(w)$ is a pair (α, γ) of longest increasing sequences of positions on w for which $w[\alpha]$ is the reverse of $w[\gamma]$. It does not guarantee that $w[\alpha]$ is a palindrome.

Example. For $w = \text{dcabdcba}$, $[(0, 1, 6, 7), (2, 3, 4, 5)]$ is a possible value of $\text{LMPS}(w)$. We have $w[\alpha] = \text{dcba}$ and $w[\gamma] = \text{abcd}$ and each word is the reverse of the other. However none of them is a palindrome, nevertheless w has a palindromic subsequence of length 4, namely **abba**.

Reduction of LPS to LMPS. Let $(\alpha, \gamma) = \text{LMPS}(x)$ and $u = x[\alpha]$. Then, it can be derived a palindromic subsequence of length $|u|$ of x .

Proof We consider only the case of odd $|u|$ since the even case is similar. Let u and v be mutually symmetric subsequences of x . Then, for the “middle” letter c , let $u = zcy$, $v = y^Rcz^R$, where $|z| = |y|$. Let p be the position of letter c on u and q its position on v . If $p \leq q$ then y^R is to the left of y and we get a palindromic subsequence y^Ry . If $p > q$ then we have a palindromic subsequence xx^R . ■

Reduction LMPS \Rightarrow LCS. For two words u and v of length n , let $\text{LCS}(u, v) = (\alpha, \beta)$, where α and β are longest increasing sequences of positions on u and on v respectively, for which $u[\alpha] = v[\beta]$. It is well known that this problem can be solved in $O(n^2)$ time. Then, we compute $(\alpha, \beta) = \text{LCS}(w, w^R)$. This gives a solution $(\alpha, \gamma) = (\alpha, \beta')$ to $\text{LMPS}(x)$, where β' results from β by numbering positions from the end.

Notes

There are other algorithmic problems related to subsequences having other specific properties: the longest palindromic subsequence, the longest subsequence that is a square or that is a highly periodic subsequence or that is a Lyndon word, see [37], [5], [6] and [30].

141 Two problems on Run-Length Encoded words

The run-length encoding of a binary word $x \in \{0, 1\}^*$ is

$$\text{RLE}(x) = 1^{p_0} 0^{p_1} \dots 1^{p_{s-2}} 0^{p_{s-1}},$$

where $s - 2 \geq 0$, $p_i > 0$ for $i = 0, \dots, s - 2$ and $p_{s-1} \geq 0$. The value $|\text{RLE}(x)|$ denotes the size of the compressed version of x . Note the length $|x|$ can be exponential w.r.t. $|\text{RLE}(x)|$.

A cover of a non-empty word x is one of its factors whose occurrences cover all positions on x . We refer to [12, Problem 45] where a list-oriented computation of covers in standard strings is presented using the prefix table of the word (see [12, Problem 22]). Our algorithm here follows similar lines, except that now it operates on sparse sets of (well chosen) positions.

Question. Let x be a word whose $\text{RLE}(x)$ is of size n . Show how to compute in linear time $O(n)$ the length of the shortest cover of x , assuming the cost of each arithmetic operation is a constant.

[Hint: Extend the list-based algorithm for shortest covers in [12, Problem 45] and the *sparse* prefix table.]

Question. Let a pattern x and a text y given in RLE-form of total size n . Show how to check if x occurs in y in $O(n)$ time.

Solution

Denote by $\text{Occ}(u, x)$ the sorted list of starting positions of occurrences of a word u in x . Assume that x is a non-unary word (otherwise the solution is trivial) and $\alpha = 1^k 0$ is a prefix of x , for $k > 0$.

Observation. Both $|\text{Occ}(\alpha, x)| \leq n$ and $\text{Occ}(\alpha, x)$ can be computed in time $O(n)$.

We use the “sparse” prefix table pref ([12, Problem 45]) defined (only) for positions in $\text{Occ}(\alpha, x)$: $\text{pref}(i)$ is the length of the longest prefix of x that has an occurrence starting at position i .

Let

$$\mathbf{L} = \{\ell : \text{pref}(i) = \ell \text{ for some } i \in \text{Occ}(\alpha, x)\}$$

and, for each length $\ell \in \mathbf{L}$, $\ell \geq 0$, let

$$\text{pref}^{-1}(\ell) = \{i : \text{pref}(i) = \ell\}.$$

Assume each set $\text{Occ}(\alpha, x)$, $\text{pref}^{-1}(\ell)$ and \mathbf{L} is represented as a linear ascending double-linked list. For $\ell \in \mathbf{L}$ denote by $\text{prev}(\ell)$ its predecessor in \mathbf{L} .

Then, Algorithm SHORTESTCOVER1 computes the length of the shortest cover of its input. In fact, it is almost the same solution as in [12, Problem 45], except that only positions in $Occ(\alpha, x)$ are considered.

```

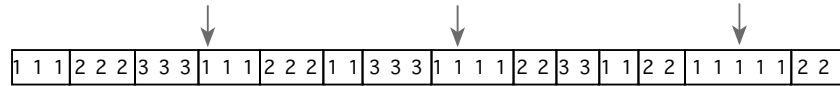
SHORTESTCOVER1( $x$  non-empty word)
1  compute the sparse prefix table of  $x$ 
2  compute  $\mathbf{L}$  and  $Occ(\alpha, x)$ 
3  compute the sets  $pref^{-1}(\ell)$  for  $\ell \in \mathbf{L}$ 
4   $\mathbf{F} \leftarrow Occ(\alpha, x)$ 
5   $\triangleright$  the sets  $\mathbf{L}$  and  $\mathbf{F}$  are represented as increasing lists
6  for  $\ell \in \mathbf{L}$  do
7      if  $\ell \neq \min(\mathbf{L})$  then
8          remove elements of  $pref^{-1}(prev(\ell))$  from  $\mathbf{F}$ 
9      update  $maxgap(\mathbf{F})$ 
10     if  $maxgap(\mathbf{F}) \leq \ell$  then
11         return  $\ell$ 
12 return null

```

We explain now how to compute the sparse prefix table $pref$. We can encode each block consisting of a maximal k -repetition of the same letter a as a single composite letter (a, k) . We discard the first block and the last block. The resulting word (consisting of encoded blocks) x' is of length $n - 2$.

Example. Let $x = 1^3 2^3 3^3 1^3 2^3 1^2 3^3 1^4 2^2 3^2 1^2 2^2 1^5 2^2$. The set \mathbf{F} consists of positions indicated by arrows in the figure below. We get $\alpha = 1^3 2$ and

$x' = (2, 3) (3, 3) (1, 3) (2, 3) (1, 2) (3, 3) (1, 4) (2, 2) (3, 2) (1, 2) (2, 2) (1, 5)$.



We compute the additional (full) prefix table $pref'$ for x' in $O(n)$ time using an algorithm for standard strings. Using the table $pref'$ it is easy to compute $pref(i)$ for each individual position $i \in Occ(\alpha, x)$ in the text x in $O(1)$ time, which is done globally in $O(n)$ time.

Solution to the second question. Let z be the word $x\#y\#$, where $\#$ does not occur in xy . Then, the RLE-encoding of z is of size $O(n)$. After computing the prefix table $pref$ of z for the special positions on z , it can be checked $pref(i) = |x|$ for each position $i \in Occ(a^k b, x)$ inside y , where a and b are letters and $a^k b$ is a prefix of x . Which solves the question.

142 Maximal Number of (distinct) Subsequences

For a word $x \in \{a, b\}^*$, let $\text{Subs}(x)$ denote the set of subsequences occurring in x , including the empty word, and let $\text{subs}(x) = |\text{Subs}(x)|$.

Question. For a word $x \in \{a, b\}^*$, design an efficient (polynomial time) algorithm computing $\text{subs}(x)$.

Question. What is a compact formula for the maximal number $S(n)$ of (distinct) subsequences of a binary word of length n .

Solution

The present solution uses the subsequence automaton of x (see [12, Problem 51]). Discarding labels of edges, the automaton is a directed acyclic graph with one initial node (source). The number of distinct paths from the source can be computed efficiently using the so called topological sorting. This number gives the number of (distinct) subsequences $\text{subs}(x)$.

Solution to the second question. Let F_k be the k -th Fibonacci number ($F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$).

Fact 1. $S(n) = F_{n+3} - 1$.

Proof The proof is by induction. It works for $n = 1$ and for $n = 2$ considering the word ab . Let $n > 2$. For any word abu of length n , where a, b are letters, the following inequality holds true

$$\text{subs}(abu) \leq \text{subs}(bu) + \text{subs}(u) + 1 \leq S(n-1) + S(n-2) + 1.$$

Then, using the induction hypothesis, it follows

$$\text{subs}(abu) \leq (F_{n+2} - 1) + (F_{n+1} - 1) + 1 = F_{n+3} - 1.$$

(The additional unit is for the empty word.)

To conclude, note that if letters a, b are distinct, the first inequality becomes an equality and the rest follows. ■

For the word $x = abab$, $\text{subs}(x) = 12$, that is F_{4+3} , since $\text{Subs}(x)$ is $\{\varepsilon, a, ab, aba, abab, aab, abb, aa, b, ba, bab, bb\}$.

Notes

The problem is from [18].

143 Avoiding Grasshopper repetitions

The problem deals with grasshopper subsequences of words. A grasshopper subsequence of a word x is a word of the form $x[i_1]x[i_2]\cdots x[i_k]$, where i_t is a position on x satisfying $i_{t+1} \in \{i_t + 1, i_t + 2\}$, for each t , $0 < t < k$. We can imagine a grasshopper jumping to the right by one or two positions.

The goal of the problem is related to long words that avoid grasshopper squares and grasshopper cubes over alphabets of size 3 and 6, respectively.

For example, **abbab** contains a grasshopper cube, namely **bbb**, while **bbaabbba** avoids grasshopper cubes.

Let $A = \{a, b, c\}$ and $A' = \{a', b', c'\}$ whose elements are called “primed” letters. For a word $v \in A^*$, the word $\Phi(v)$ over the alphabet $A \cup A'$ is defined using the coding (morphism):

$$a \rightarrow aa', b \rightarrow bb', c \rightarrow cc'.$$

For example, $\Phi(abc) = aa'bb'cc'$.

Question. Let $x \in A^+$, $y = \Phi(x)$ and z be a grasshopper square in y . Show how to compute in time $O(|z|)$ a (standard) square v in x of length at least $|z|/2$.

Question. For a given integer $n > 0$, build a word of length n over a 6-letter alphabet that avoids grasshopper cubes.

Solution

For a symbol $s \in A \cup A'$, let us define $unprime(s)$ by $a \rightarrow a$, $a' \rightarrow a$, $b \rightarrow b$, $b' \rightarrow b$, $c \rightarrow c$, $c' \rightarrow c$.

Algorithm RECOVER SQUARE below constructs a required square v in x . From the grasshopper square $z = z[0..k-1]$ in $y = \Phi(x)$ let us denote by $fill_gaps(z)$ the shortest factor of y that contains z and is in $(AA')^+$. In other words $\Phi^{-1}(fill_gaps(z))$ is well defined. In fact, $fill_gaps(z)$ results by filling “gaps” created by jumps with letters, and eventually expanding the starting and ending part of z ; the resulting string should start with a symbol in A and end with a symbol in A' . This is what the algorithm computes up to line 7 to get the word v . Additionally, if v is of odd length its last letter is removed to get the output.

Example. Let $x = abacba$. The word $\Phi(x)$ is $aa'bb'aa'cc'bb'aa'$ and contains the grasshopper square $z = a'baa'ba$.

Assume we start the algorithm with $z = a'baa'ba$. After executing

statement in line 6 we get $v = \mathbf{ababa}$. The removal of the last symbol of v is necessary and is done in line 9. Ultimately the algorithm produces the square $v = \mathbf{abab}$ in x .

```

RECOVERSQUARE( $x \in A^+$ ,  $z$  grasshopper square in  $\Phi(x)$ )
1  ( $k, v, i$ )  $\leftarrow$  ( $|z|, \varepsilon, 0$ )
2  while  $i < k$  do
3      if  $z[i] \in A$  and  $z[i+1] \in A'$  then
4          ( $s, i$ )  $\leftarrow$  ( $z[i], i+2$ )
5      else ( $s, i$ )  $\leftarrow$  ( $\text{unprime}(z[i]), i+1$ )
6           $v \leftarrow v \cdot s$ 
7   $\triangleright v = \Phi^{-1}(\text{fill\_gaps}(z))$ 
8  if  $|v|$  is odd then
9       $v \leftarrow$  ( $v$  without its last symbol)
10 return  $v$  (square in  $x$ )

```

The correctness of Algorithm RECOVERSQUARE follows from the fact that odd positions on y point to primed symbols and even positions to unprimed symbols. Hence, due to limited jumps (one or two steps), whenever we have a factor cd' , for $c \in A$ and $d' \in A'$, we know that $d' = c'$ and the factor becomes cc' that decodes to c .

Solution to the second question. Use any sufficiently long square-free word w over the 3-letter alphabet $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ and compute $\Phi(w)$. The solution to the previous question guarantees that $\Phi(w)$ avoids grasshopper cubes.

Notes

For a given integer $n > 0$ there is a word of length n over a 3-letter alphabet that avoids grasshopper cubes. To see it, let v be a cube-free word over the alphabet $\{\mathbf{a}, \mathbf{b}\}$. We create the required word w over the alphabet $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ by applying to v the coding defined by: $\mathbf{a} \rightarrow \mathbf{c}^2\mathbf{a}$, $\mathbf{b} \rightarrow \mathbf{c}^2\mathbf{b}$. The correctness can be proved similarly to the above solution for squares, see [16].

Our presentation is a version of constructions in [16]. Computer experiments show that 5 letters are not enough to avoid grasshopper squares: every word of length 23 over a 5-letter alphabet contains a grasshopper square. Besides, two letters are not enough to avoid grasshopper cubes. Hence, the numbers 3 and 6 here are minimal.

144 Counting unbordered words and relatives

We assume, for simplicity, that the alphabet is binary. A word is called *unbordered* if it has no proper border. For example the word *ababb* is unbordered, as well as the empty word. Denote by $\mathbf{u}(n)$ the number of unbordered binary words of length n . We have:

$$\mathbf{u}(0), \mathbf{u}(1), \mathbf{u}(3), \mathbf{u}(4), \dots = 1, 2, 2, 4, 6, 12, 20, 40, 74, \dots$$

Observation. A word w is unbordered if and only if it has no border of length at most $|w|/2$.

There are 2^{n-k} binary words with border of size k , for $k \leq n/2$. Due to the observation we have an exponential lower bound on $\mathbf{u}(n)$

$$\mathbf{u}(n) \geq 2^n - 2^{n-1} - 2^{n-2} - \dots - 2^{\lceil n/2 \rceil} \geq 2^{n/2}.$$

Denote by $\mathbf{v}(n)$ and $\mathbf{t}(n)$ the number of binary words of length n without nontrivial prefix palindrome of even length and odd length, respectively.

Question. Describe algorithms computing $\mathbf{u}(n)$, $\mathbf{v}(n)$ and $\mathbf{t}(n)$ in $O(n)$ time

Solution

We can use recurrences:

$$(*) \quad \mathbf{u}(2n+1) = 2 \cdot \mathbf{u}(2n), \quad \mathbf{u}(2n) = 2 \cdot \mathbf{u}(2n-1) - \mathbf{u}(n)$$

The first equality follows from the fact that a $(2n+1)$ -length word is unbordered if and only if it is unbordered after removing the “middle” letter. There are two possible letters, hence we have the coefficient 2.

Similarly, an even length word $w = aw_1bw_2$ of length $2n$, where $|w_1| = |w_2| = n-1$, is unbordered if and only if aw_1w_2 (of length $2n-1$) is unbordered and aw_1, bw_2 are not equal unbordered words ($\mathbf{u}(n)$ possibilities to exclude).

The computation of $\mathbf{v}(n)$ is easy due to equality $\mathbf{v}(n) = \mathbf{u}(n)$ for each n . We prove it the following operation. For two words $x = a_1a_2 \dots a_m$, $y = b_1b_2 \dots b_m$ denote $x \otimes y = a_1b_1a_2b_2 \dots a_mb_m$.

Now we construct a bijection \mathbf{F} in the following way.

$$\text{If } w = uav, |u| = |v|, |a| \leq 1 \text{ then } \mathbf{F}(w) = u \otimes v^R a.$$

Example. Let $w = abcd \circ \circ \circ \circ \bullet \star \star \star \star abcd$. We have

$$\mathbf{F}(w) = adbcbda \circ \star \circ \star \circ \star \circ \star \bullet.$$

Observe that w has a border $abcd$ and $\mathbf{F}(w)$ has prefix palindrome $adbcbda$.

It is easy to see that

$$w \text{ is bordered} \Leftrightarrow \mathbf{F}(w) \text{ has nontrivial even prefix palindrome}$$

Computing $\mathbf{t}(n)$. The numbers $\mathbf{t}(n), \mathbf{v}(n)$ are “almost” the same. The computation of $\mathbf{t}(n)$ is easy due to equality

$$(**) \quad \mathbf{v}(2n+1) = \mathbf{t}(2n+1), \quad \mathbf{t}(2n) = 2 \cdot \mathbf{t}(2n-1)$$

We justify the first equality using simple algebraic trick.

Denote by \oplus the operation of addition modulo 2. For a word

$w = a_1 a_2 \cdots a_m$ define

$$\mathbf{F}'(w) = b_1 b_2 \cdots b_{m-1}, \text{ where } b_i = a_i \oplus a_{i+1} \text{ for } i < m.$$

Observation. A word x is a nontrivial odd palindrome if and only if $\mathbf{F}'(x)$ is a nontrivial even palindrome.

Due to the observation we have a mapping \mathbf{F}' of the set of length- $(2n+1)$ words without odd palindromes onto the set of length- $2n$ words without even palindromes. \mathbf{F}' is not bijection, however it is a “2-bijection”:

$$|\mathbf{F}'(y)| = 2 \text{ for each word } y, \text{ such that } |y| \geq 2.$$

Consequently,

$$\mathbf{t}(2n+1) = 2 \cdot \mathbf{v}(2n) = \mathbf{v}(2n+1),$$

due to Equation (*).

The second equality in Equation (**) follows from the fact that for each length- $2n$ word w we can create two length- $(2n+1)$ words w_1, w_2 by inserting 0 or 1 in the middle. The word w has no nontrivial prefix palindrome of odd length if and only if w_1, w_2 have the same property.

Notes

There are other relations between unbordered words and palindromes. Prime palstars are even nonempty palindromes which are not a concatenation of smaller even nonempty palindromes. It is known that the number of prime palstars of length $2n$ equals $\mathbf{u}(n)$, see [46]. Another problem concerns the number $A_3(n)$ of ternary words without **any** non-trivial palindromic prefix, then we have a recurrence similar to (*):

$$A_3(n) = 3 A_3(n-1) - A_3(\lceil n/2 \rceil).$$

The relation between length- n unbordered words and words without even palindromic prefix is from [22]. The cardinalities of the sets of length- n unbordered words with fixed “weight” have been investigated in [27]. The weight of a binary word is the number of ones. Let $U(n, k)$ denote the number of length- n unbordered binary words of weight k . If $0 < k < n$ then

$$U(n, k) = U(n-1, k) + U(n-1, k-1) - \alpha(n, k) \cdot U(n/2, k/2),$$

where $\alpha(n, k) = 1$ if both n, k are even, otherwise $\alpha(n, k) = 0$. Interestingly, a different natural sequence of numbers of n -length sequences of $+1$ and -1 , not summing together to zero, looks initially the same as **u**: 1, 2, 2, 4, 6, 12, 20, 40. Afterwards it differs from **u**.

145 Cartesian Tree Pattern-Matching

In the problem we consider words drawn from a linear-sortable alphabet Σ of integers. Let $x = x[0..m-1]$ be a word of length m . The Cartesian Tree $\mathbf{CTree}(x)$ of x is a binary tree in which:

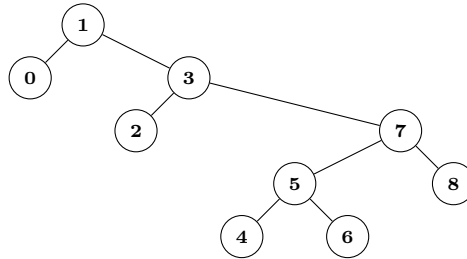
- the root is the position i of the minimal element $x[i]$ (if there are several occurrences of the minimal element, its leftmost position is chosen);
- the left subtree of the root is $\mathbf{CTree}(x[0..i-1])$;
- the right subtree of the root is $\mathbf{CTree}(x[i+1..m-1])$.

The *Cartesian tree pattern-matching* problem is naturally defined as follows: given a pattern x and a text y of length m and n respectively, find all factors of y that have the same Cartesian tree as x .

Example. Let $x = 3\ 1\ 6\ 4\ 8\ 6\ 7\ 5\ 9$, and

$$y = 10\ 12\ 16\ \underline{15\ 6\ 14\ 9\ 12\ 11\ 14\ 9\ 17}\ 12\ 10\ 12$$

The underlined factor $u = 15\ 6\ 14\ 9\ 12\ 11\ 14\ 9\ 17$ of y has the same Cartesian tree as x : $\mathbf{CTree}(u) = \mathbf{CTree}(x)$.



Question. Design an online linear time and space algorithm that builds the Cartesian tree of a word $x \in \Sigma^*$.

[Hint: Consider only nodes on the rightmost paths of the tree.]

Question. Design a linear-time algorithm for the Cartesian tree pattern-matching related to a pattern x and a text y in Σ^* .

[Hint: Find a linear representation of Cartesian trees and design a notion of border table (see [12, Problems 19 and 26]) adequate to the problem.]

Solution

The algorithm considers the right path of the tree (starting from the root and always going right) as a stack of positions. The Cartesian tree of $x[0]$ consists thus of a single root node 0 with a stack containing this element only. Then, given the Cartesian tree of a prefix $x[0..i]$ of x , with $0 \leq i < |x| - 1$, the Cartesian tree of $x[0..i+1]$ is obtained by popping from the stack all positions $j \leq i$ for which $x[j] > x[i+1]$. If no such element, $i+1$ is a leaf and inserted as the right child of the top element of the stack. Otherwise, let k be the last popped element from the stack for which $x[k] > x[i+1]$. There are two cases:

1. The stack is not empty. Let k' be its top element ($x[k'] < x[i+1]$). Then $i+1$ is inserted as the right child of k' and k becomes the left child of $i+1$.
2. The stack is empty. Then $i+1$ is inserted at the root of the Cartesian tree of $x[0..i+1]$ and k becomes the left child of $i+1$.

Eventually, $i+1$ is pushed on the stack.

In all cases, the new element $i+1$ is always the last element of the right path (thus the top element of the stack).

Since each index j can be popped only once from the stack, the whole online process takes linear worst-case time.

Solution to Cartesian tree pattern-matching. The present solution is based on the notion of a Parent-Distance array PD_w of a word w , which is defined as follows for $0 \leq i < |w|$:

$$\text{PD}_w[i] = \begin{cases} i - \max_{0 \leq j < i} \{j : w[j] \leq w[i]\} & \text{if such } j \text{ exists,} \\ 0 & \text{otherwise.} \end{cases}$$

The Parent-Distance representation has a one-to-one mapping to the Cartesian tree.

Below is the Parent-Distance representation of $x = 3 \ 1 \ 6 \ 4 \ 8 \ 6 \ 7 \ 5 \ 9$.

i	0	1	2	3	4	5	6	7	8
$x[i]$	3	1	6	4	8	6	7	5	9
$\text{PD}_x[i]$	0	0	1	2	1	2	1	4	1

The Parent-Distance representation of a word w can be computed in time $O(|w|)$ time using an algorithm similar to the one for building the Cartesian tree of w . Given the Parent-Distance representation of w , the Parent-Distance of a factor $w[i..j]$ of w satisfies:

$$\text{PD}_{w[i..j]}[k] = \begin{cases} 0 & \text{if } \text{PD}_w[i+k-1] \geq k, \\ \text{PD}_w[i+k-1] & \text{otherwise.} \end{cases}$$

Then, the Cartesian border table of w is defined in the following way: $\text{CTBord}[0] = -1$ and, for $1 \leq k < i$,

$$\text{CTBord}[i] = \max\{k : \mathbf{CTree}(w[0..k]) = \mathbf{CTree}(w[i-k+1..i])\}$$

Below is the Cartesian border table of $x = 3 \ 1 \ 6 \ 4 \ 8 \ 6 \ 7 \ 5 \ 9$.

i	0	1	2	3	4	5	6	7	8
$x[i]$	3	1	6	4	8	6	7	5	9
CTBord[i]	-1	0	0	1	2	3	4	1	2

Algorithm CTMATCH answers the second question. It first builds the Parent-Distance representation of x and y and builds the Cartesian border table of x . It uses a deque Q to represent the right path of the Cartesian tree of substring of y that matches the Cartesian tree of a prefix of x .

```

CTMATCH( $x, y$  non-empty words)
1   $PD_x, PD_y \leftarrow$  Parent-Distance representations of  $x$  and  $y$ 
2   $CTBord \leftarrow$  Cartesian border table of  $x$ 
3   $i \leftarrow -1$ 
4   $Q \leftarrow$  empty stack
5  for  $j \leftarrow 0$  to  $|y| - 1$  do
6      delete elements  $(v, k)$  from back of  $Q$  with  $v > y[j]$ 
7      while  $i > -1$  and  $PD_{y[j-i..j]}[i+1] = PD_x[i+1]$  do
8           $i \leftarrow CTBord[i]$ 
9          delete elements  $(v, k)$  from front of  $Q$  with  $k < j - i$ 
10     add  $(y[j], j)$  at the back of  $Q$ 
11      $i \leftarrow i + 1$ 
12     if  $i = |x| - 1$  then
13         output: match at position  $j - |x| + 1$ 
14          $i \leftarrow CTBord[i]$ 
15     delete elements  $(v, k)$  from front of  $Q$  with  $k < j - i$ 

```

Notes

Cartesian trees have been introduced by Vuillemin [53]. More information in https://en.wikipedia.org/wiki/Cartesian_tree with various applications of them. Multiple pattern Cartesian tree matching and a suffix tree for Cartesian tree matching is considered in [44]. Fast practical solutions are presented in [52].

An obvious application of this type of matching is to detect analogue ups and downs behaviour in time series without processing their absolute values.

There is a strong connection between Cartesian trees and (right) Lyndon trees. Indeed, the Lyndon tree of a word is a Cartesian tree built from the lexicographic rank of its suffixes (see [29, 13]). The notion of Lyndon tree is essential tool to deal with repetitions in words (see [4]).

146 List-Constrained Square-Free Strings

Let L be a list of finite alphabets (L_1, L_2, \dots, L_n) . A word $a_1 a_2 \dots a_n$ is said to be L -constrained if $a_i \in L_i$ for each i , $1 \leq i \leq n$. The aim is to find L -constrained square-free words of length n .

Example. For the list $L = (\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{e}\}, \{\mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}\}, \{\mathbf{a}, \mathbf{c}, \mathbf{d}, \mathbf{e}\}, \{\mathbf{c}, \mathbf{a}, \mathbf{b}, \mathbf{e}\}, \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}, \{\mathbf{b}, \mathbf{c}, \mathbf{d}\}, \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}\}, \{\mathbf{a}, \mathbf{c}, \mathbf{d}, \mathbf{e}\}, \{\mathbf{c}, \mathbf{a}, \mathbf{b}, \mathbf{d}\})$, among many others, the word **abcabdbca** is an L -constrained square-free word.

For simplicity, assume from now on that each L_i is of size 5. The constructed word u is treated as a stack: adding a symbol at the end corresponds to a *push* operation and removing the last symbol corresponds to a *pop* operation. Let pop^k be the sequence of k pop operations. Let also $\frac{1}{2}\text{square}(u)$ be the maximal half-length of the suffix of u that is a square.

Let $\mathbf{C} = \{1, 2, 3, 4, 5\}^{8n}$ and $\text{symbol}_j(t)$ denote the t -th symbol on the list L_j . Informally speaking, each element $c \in \mathbf{C}$ is treated as a “control sequence”. During the i -th iteration of Algorithm H below, the letter $\text{symbol}_j(c[i])$ is inserted at the j -th position of u by pushing it onto the stack.

The following function H runs a naive backtracking way controlled by the sequence $c \in \mathbf{C}$. The result is (u, β) , where u is a square-free word and β is an auxiliary value. We have $|u| \leq n$.

```

H( $c \in \mathbf{C}$ )
1  ( $u, i$ )  $\leftarrow$  (empty stack, 1)
2  while  $i \leq 8n$  and  $|u| < n$  do
3       $j \leftarrow |u| + 1$ 
4      push  $\text{symbol}_j(c[i])$ 
5      if  $u$  contains a suffix square then
6           $k \leftarrow \frac{1}{2}\text{square}(u)$ 
7           $u \leftarrow \text{pop}^k(u)$ 
8       $i \leftarrow i + 1$ 
9   $\beta \leftarrow$  the sequence of executed push and pop operations
10 ( $\beta$  is the sequence of symbols “push” and “pop”)
11 return ( $u, \beta$ )

```

Question. Show constructively that there exists an L -constrained square-free word of length $n = |L|$ if each set L_i of L is of size 5.

We say that $c \in \mathbf{C}$ is successful if $H(c) = (u, \beta)$, where $|u| = n$. Our

algorithm is to compute the function $H(c)$ for all possible $c \in \mathbf{C}$ and choose any c for which $H(c)$ is successful. Then we return u , where $H(c) = (u, \beta)$. It is enough to show that such c exists.

Observation 1. If $H(c) = (u, \beta)$ with $|u| < n$ then β contains $8n$ symbols “push” and $8n - |u|$ symbols “pop”.

$H(c)$ records the computation history: both the sequence of moves of the stack (pops and pushes) and the word u as final content of the stack, with $|u| \leq n$. This is sufficient to reconstruct the word c if $|u| < n$.

Solution

If $n \leq 5$ there is obviously an L -constrained square-free word of length n . Hence, we assume later $n \geq 6$.

It is enough to show that for at least one $c \in \mathbf{C}$ the algorithm is successful, in other words $H(c) = (u, \beta)$, where $|u| = n$. Define

$$V = \{H(c) : c \in \mathbf{C}\}.$$

The following fact says that in the unsuccessful case, that is, $|u| < n$, from (u, β) , the sequence of symbols pushed onto the stack can be recovered by reversing the algorithm. Hence, (u, β) uniquely determines the sequence $c = H^{-1}(u, \beta)$. If, for each $c \in \mathbf{C}$, $H(c) = (u, \beta)$ with $|u| < n$ then the function H is a one-to-one mapping. This implies the following fact

Observation 2. Assume that for each $c \in \mathbf{C}$ the algorithm is unsuccessful. Then $|V| \geq |\mathbf{C}|$

Now we show that our algorithm is successful. The proof is by contradiction. Assume the algorithm is unsuccessful for each c .

There are at most $2 \cdot 4^{8n}$ sequences consisting of $8n$ push operations and at most $8n$ pop operations. The number of possible values of u is at most $2 \cdot 5^n$. So, $|V| \leq 4 \cdot 5^n \cdot 4^{8n}$. Besides, $|\mathbf{C}| = 5^{8n}$. Together, this gives $|V| \leq 4 \cdot 5^n \cdot 4^{8n} < 5^{8n} = |\mathbf{C}|$ for $n > 5$.

Therefore, the unsuccessful assumption, due to Observation 2, leads to a contradiction, and proves that the algorithm is successful for at least one c .

Notes

Our presentation is a deterministic version of the probabilistic algorithm from [25], where list elements of size 4 are shown to work similarly as for size 5. But the proof needs certain properties of Catalan numbers. The above algorithm has a pessimistic exponential time. However, by choosing randomly a control sequence c , it is claimed in [25] that it gives a randomized linear-time algorithm.

It is conjectured that there are also list-constrained square-free words when list elements are of size 3. The conjecture was confirmed by Matthieu Rosenfeld [48] for the case where all list elements of size 3 are subsets of the same alphabet of size 4.

There are other square-free problems on “special” words, for example, Abelian square-free words with 4 letters, see [35], or circular square-free words with 3 letters of the length not in $\{5, 7, 9, 10, 14, 17\}$, see [51].

147 Superstrings of shapes of permutations

Two words u and v of the same length are said to be *order-equivalent*, written $u \approx v$, if $u[i] < u[j] \iff v[i] < v[j]$ for all pairs of positions i, j on the words. For a word u of length n with all letters distinct we define $\text{shape}(u)$ as the n -permutation of $\{1, 2, \dots, n\}$ order-equivalent to u . For example $\text{shape}(2, 5, 4) = (1, 3, 2)$. Define

$$\text{SHAPES}_n(w) = \{\text{shape}(u) : u \text{ is a factor of } w \text{ of length } n\}$$

For $n > 2$ there is no word containing exactly once each n -permutation, but surprisingly in order-preserving case such a word exists for each n .

A word of size $n! + n - 1$ containing shapes of all n -permutations is called a *universal word*, it is a superstring of shapes of all n -permutations. Obviously $n! + n - 1$ is the smallest length of such word.

Example. The word 3 5 1 0 5 1 2 3 of length $3! + 2$ is 3-universal: The sequence of shapes of its factors of length 3 is:

$$(2, 3, 1) \rightarrow (3, 2, 1) \rightarrow (2, 1, 3) \rightarrow (1, 3, 2) \rightarrow (3, 1, 2) \rightarrow (1, 2, 3).$$

Question. Construct, for a given n , a universal word of size $n! + n - 1$ (shortest possible).

[**Hint:** Use a construction similar to that for linear de Bruijn words.]

Solution

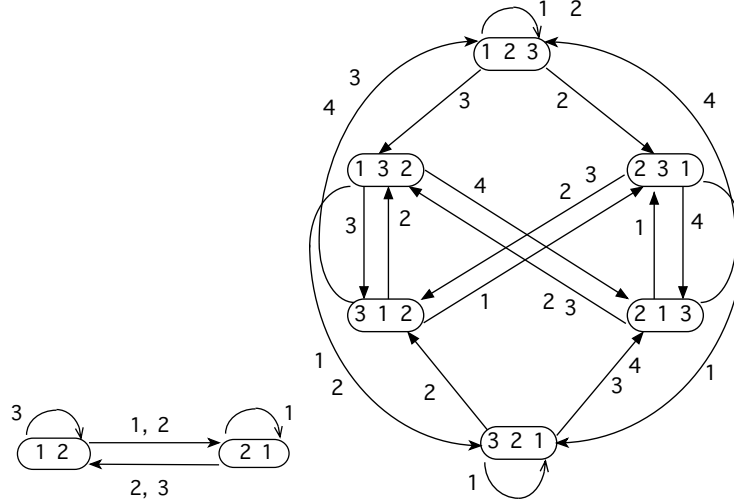
Denote by $\text{suf}(w)/\text{pref}(w)$ the suffix/prefix of length $n - 1$ of the word w . We construct a graph G_n . Its nodes are $(n - 1)$ -permutations and edges correspond to n -permutations. The edge corresponding to the n -permutation π is defined as

$$\text{shape}(\text{pref}_{n-1}(\pi)) \xrightarrow{k} \text{shape}(\text{suf}(\pi)),$$

where the label k is the last element of π , see the figure. For example the 5-permutation $(3, 5, 4, 1, 2)$ corresponds to the edge $(2, 4, 3, 1) \xrightarrow{2} (4, 3, 1, 2)$.

Let $\text{lift}(\alpha, a)$ be the operation of adding 1 to each element of α equal or larger than a . For example $\text{lift}((3, 2, 4, 1), 3) = (4, 2, 5, 1)$.

Observation. Assume α is a sequence of integers, $\text{suf}(\alpha)$ consists of distinct integers and α' results by replacing $\text{suf}(\alpha)$ with $\text{lift}(\text{suf}(\alpha), a)$. Then $\text{SHAPES}(\alpha') = \text{SHAPES}(\alpha)$.



We describe the following operation for $1 \leq k \leq n$ and a sequence α of distinct integers of length at least $n - 1$.

Extend_n(α, k):

$\beta := \text{suf}(\alpha)$

if $k < n$ then $a := k$ -th smallest element of β

else $a := \max(\beta) + 1$

$\alpha := \text{lift}(\alpha, a)$ (now $a \notin \alpha$)

$\alpha := \alpha \cdot a$

return α

Example. Let $\alpha = (4, 6, 7, 5, 1, 7, 6, 4, 3, 1)$. Then

$\text{Extend}_4(\alpha, 2) = (5, 7, 8, 6, 1, 8, 7, 5, 4, 1, 3)$,

$\text{Extend}_4(\alpha, 4) = (4, 7, 8, 6, 1, 8, 7, 4, 3, 1, 5)$.

Denote by $\text{EulerCycle}(G_n, \alpha)$ the sequence of labels of any Euler cycle of G_n starting in the node α (an $(n - 1)$ -permutation).

SUPERSTRING(n , positive integer)

1 $\alpha \leftarrow (1, 2, \dots, n - 1)$

2 $C := \text{EulerCycle}(G_n, \alpha)$ (Assume $C = c_1 c_2 \dots c_{n!}$)

3 **for** $i \leftarrow 1$ **to** $n!$ **do**

4 $\alpha \leftarrow \text{Extend}_n(\alpha, c_i)$

5 **return** α

Example. $\text{EulerCycle}(G_3, (12)) = 12112233$. The algorithm returns a universal word $\alpha = 78613245$. We can reduce the alphabet and get 56413245, with the same sequence of length-3 shapes.

It is easy to see that G_n is Eulerian. The computed α contains all permutations as shapes. It follows from the following fact.

Observation. Assume the edges of the Euler cycle given by labels $C = c_1 c_2 \dots c_n!$ correspond to sequence of permutation $\pi_1, \pi_2, \dots \pi_n!$. If $\text{SHAPES}(\alpha) = \{\pi_1, \pi_2, \dots \pi_{i-1}\}$ then

$$\text{SHAPES}_n(\text{Extend}(\alpha, c_i)) = \{\pi_1, \pi_2, \dots \pi_{i-1}, \pi_i\}$$

Notes

The resulting universal word produced in the algorithm presented here uses huge amount of letters, however the most interesting is that universal words exist at all. The algorithm is a version of that in [23], where it was also given construction of cyclic universal words.

There exist universal words with only $n + 1$ letters, but the construction is too complex to present it here, we refer to [33].

148 Linearly generated words and primitive polynomials

We consider sequences of length- n binary non-unary words (each containing at least one nonzero bit). There are $N = 2^n - 1$ such word. By \oplus denote operation *xor* on bits. Let $\alpha = (a_0, a_1, \dots, a_{n-1})$ be a (control) sequence of bits. The **LFSR-sequence** associated with α , denoted by $\text{LFSR}(\alpha)$, is the sequence $b_1 b_2 b_3 \dots b_{N+n-1}$ of bits, such that for $n < k < N + n - 2$.

$$b_1 b_2 \dots b_n = 0^{n-1} 1,$$

$$b_{k+1} = a_0 \cdot b_{k-n+1} \oplus a_1 \cdot b_{k-n+2} \oplus a_2 \cdot b_{k-n+3} \oplus \dots \oplus a_{n-1} \cdot b_k$$

For example for $\alpha = 11010$ and $n = 5$ the recurrence is

$$b_{k+1} = b_{k-4} \oplus b_{k-3} \oplus b_{k-1}.$$

We fix the starting prefix $0^{n-1}1$ for simplicity. Observe that $N + n - 2$ is the smallest length of a binary sequence containing each length- n nonzero word. Denote by $\text{GEN}(\alpha)$ the sequence of consecutive length- n factors in $\text{LFSR}(\alpha)$. Observe that $\text{GEN}(\alpha)$ is of length N .

Example. We have: $\text{LFSR}(110) = 001011100$,

$$\text{GEN}(110) = 001, 010, 101, 011, 111, 110, 100.$$

The polynomial related to $\text{LFSR}(\alpha)$, where $\alpha = a_0 a_1 \dots a_{n-1}$ is

$$W_\alpha = x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \dots + a_1x^1 + a_0$$

W_α is called a generating polynomial for $\text{LFSR}(\alpha)$. If all words in $\text{GEN}(\alpha)$ are different then the sequence $\text{LFSR}(\alpha)$ is called a *PN sequence* (pseudo-noise) sequence. The polynomial P is called *primitive* if all polynomials $x^i \bmod P(x)$ are distinct, for $1 \leq i \leq 2^n - 1$ (it is maximal number of nonzero binary polynomials of degree smaller than n). It is known, and surprising, that LFSR-sequences corresponding to primitive polynomials are PN sequences. In this way construction of PN sequences is reduced to construction of primitive polynomials, which is easier since one can use algebraic tools.

Question. Compute the m -th word of $\text{GEN}(\alpha)$ in $O(n^3 \log m)$ time, using matrix multiplications.

Question. Improve time complexity of computing the m -th word of $\text{GEN}(\alpha)$ to $O(n \log n \cdot \log m + n^2)$ time, using polynomial multiplications.

Solution

Each binary polynomial $V(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \dots + a_0$ of degree at most $n-1$ (some of a_i could equal zero) can be represented as the word $string(V) = a_{n-1}a_{n-2} \dots a_0$. The length of $string(V)$ equals n .

Example. For $n = 5$ and polynomials P of degree less than 5 each $string(P)$ is of length 5, we have

$$string(x^2 + 1) = 00101, \quad string(x) = 00010, \quad string(x^3) = 01000.$$

We create the $n \times n$ matrix A . The i -th column, for $1 \leq i \leq n$, equals $string(x^i \bmod W_\alpha)$ written bottom-up.

In particular the n -th column is the sequence a_0, a_1, \dots, a_{n-1} read top-down, and the i -th column, for $i < n$ is the sequence $0^i 10^{n-i-1}$, read also top-down.

Then $GEN(\alpha)$ is the sequence of the first rows of A^1, A^2, A^3, \dots . Consequently, the required result equals the first row of A^m .

Fast computation of A^m The computation of A^m can be done in time $O(n^3 \log m)$ by first computing all powers A^t , where t is a power of 2 not exceeding 2^n .

Example Consider $\alpha = a_0a_1a_2a_3a_4a_5 = 10100$ and

$$W_\alpha = x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0 = x^5 + x^2 + 1,$$

Then the consecutive first rows of A^i , for $i = 1, 2, 3, \dots$ give the sequence $GEN(\alpha)$. The first 6 powers of $A = A^1$ are:

$$\begin{aligned}
 A^1 &= \begin{array}{|c|c|c|c|c|} \hline \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \hline 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 \\ \hline \end{array} & A^2 &= \begin{array}{|c|c|c|c|c|} \hline \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \hline 0 & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 \\ \hline \end{array} \\
 A^3 &= \begin{array}{|c|c|c|c|c|} \hline \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \hline 0 & 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 1 & 0 & 1 \\ \hline 1 & 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 & 1 \\ \hline \end{array} & A^4 &= \begin{array}{|c|c|c|c|c|} \hline \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \hline 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 1 & 0 & 1 & 1 \\ \hline 0 & 0 & 1 & 0 & 1 \\ \hline 1 & 0 & 0 & 1 & 0 \\ \hline \end{array} \\
 A^5 &= \begin{array}{|c|c|c|c|c|} \hline \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \hline 0 & 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 1 & 1 & 0 \\ \hline 0 & 1 & 0 & 1 & 1 \\ \hline 0 & 0 & 1 & 0 & 1 \\ \hline \end{array} & A^6 &= \begin{array}{|c|c|c|c|c|} \hline \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ \hline 1 & 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 1 & 1 & 0 \\ \hline 0 & 1 & 0 & 1 & 1 \\ \hline \end{array}
 \end{aligned}$$

Solution

We go now to the second question, using the following observation.

Observation.

- The columns (read left-to-right) of the matrix A^m correspond to $string(x^m)$, $string(x^{m+1})$, \dots , $string(x^{m+n-1})$.
- The sequence $\text{GEN}(\alpha)$ consists of first rows of consecutive matrices.

In the example the first 6 words of $\text{GEN}(10100)$ are

00001, 00010, 00100, 01001, 10010, 00101.

The columns of A^6 correspond to:

$$x^6 \bmod W_\alpha = x + x^3, \quad x^7 \bmod W_\alpha = x^2 + x^4,$$

$$x^8 \bmod W_\alpha = 1 + x^2 + x^4, \quad x^9 \bmod W_\alpha = x + x^3 + x^4$$

$$x^{10} \bmod W_\alpha = 1 + x^4.$$

Instead of using matrix multiplication we can compute $x^m \bmod W_\alpha$, by computing first x^{2^i} for all i 's such that $2^i \leq m$. The cost $O(n^3)$ of matrix multiplication is reduced to $O(n \log n)$ time of polynomial multiplication using FFT. We need $O(\log m)$ such multiplications. Once we know x^m we can compute all columns of A^m , since they correspond to $x^m, x^{m+1}, \dots, x^{m+n-1}$, it needs $O(n^2)$ time. Altogether we need $O(n^2)$ time.

Notes

Using the observation it is relatively easy, though tedious, to show that primitive polynomials generate PN-sequences. It is enough to show that the top row of A^i determines the whole matrix A^i . Then, if a polynomial is primitive, the values of the first column correspond to powers of x , which are different due to polynomial primitivity, so all the first rows are distinct.

Hence if the polynomial W_α is primitive then LFSR generates $2^n - 1$ distinct words. The proof of this fact is nontrivial

However it is rather simple if the polynomial W_α is of the form $x^n + x^k + 1$ (it is a *trinomial*). Assume $x = uv$ is the first row in the table A^i , where $|u| = k$. Then, in this case the sequence of coefficients of $x^k \bmod W_\alpha$ equals vu . See the last example, where $W_\alpha = x^5 + x^2 + 1$. This gives a bijection between elements of $\text{GEN}(\alpha)$ and powers of x modulo W_α , and this set is of size $2^n - 1$ according to definition of polynomial primitivity.

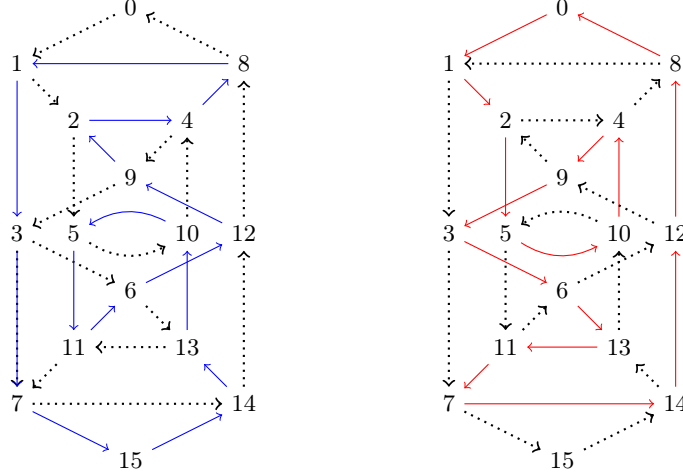
If we know any n consecutive values of $\text{GEN}(\alpha)$ then we can compute α in polynomial time using Berlekamp-Massey algorithm, see [7]. Linear feedback shift register and PN sequences were introduced by Solomon Golomb.

149 An application of linearly generated words

In this problem we want to decompose each binary de Bruijn graph G_{n+1} , disregarding two loops, into two edge-disjoint simple cycle. The word “simple” means that nodes do not repeat on the cycle. It is easy to see that each cycle should be of length $2^n - 1$. LFSR-sequences provide a surprisingly simple algorithm.

Assume the alphabet is $\{0, 1\}$. Denote by $\text{CycF}_m(w)$ the set of all cyclic length- m factors of a word w . A word of length 2^n is a de Bruijn word of rank n if $\text{CycF}_n(w) = 2^n$. We say that a word w is a semi-deBruijn word of rank n if $|w| = 2^n - 1$ and $\text{CycF}_n(w) = 2^n - 1$. Two semi-deBruijn words u, w are *orthogonal* if $\text{CycF}_{n+1}(w) \cap \text{CycF}_{n+1}(u) = \emptyset$. We are interested in finding two orthogonal semi-deBruijn words u, w .

A simple cycle of length $2^n - 1$ in G_n is called a *semi-Hamiltonian* cycle. Orthogonal semi-deBruijn words correspond to such cycles. The figure shows the decomposition of the graph G_5 without loops into two edge-disjoint semi-Hamiltonian cycles.



We refer to problems [12, Problem 18], [12, Problem 69] for the formal definition of de Bruijn graph G_{n+1} . The nodes of G_{n+1} are words of length n , edges correspond to words of length $n + 1$. The word $a_1 a_2 \cdots a_{n+1}$ corresponds to the edge

$$a_1 a_2 \cdots a_n \xrightarrow{a_{n+1}} a_2 a_3 \cdots a_{n+1}$$

We discard two loops in the graph and ask to compute two semi-Hamiltonian cycles covering all non-loop edges. In the figure each node i corresponds to the 4-bit binary representation of i .

Question. Assume you have a primitive polynomial $W(x)$ over Z_2 of degree n . Compute two orthogonal semi-deBruijn words u, w . Equivalently, compute two edge-disjoint semi-Hamiltonian cycles in G_{n+1} covering all non-loop edges.

[Hint: Use LFSR-sequences.]

Solution

Let α be the sequence of coefficients of $W(x)$, without coefficient at x^n , in the order of increasing powers of x^i : $\alpha = (a_0, a_1, \dots, a_{n-1})$. For a binary word w denote by \bar{w} its bitwise negation, for example $\overline{0011} = 1100$. Using LFSR the construction is as follows.

Algorithm.

```

 $w := \text{LFSR}(\alpha);$ 
 $u := \bar{w};$ 
remove the last  $n - 1$  letters in  $u$  and in  $w$ ;
return  $u, w$ 

```

Example. We have $\text{LFSR}(1001) = 000111101011001\underline{000}$. The algorithm deletes the underlined fragment and returns

$w = 000111101011001, u = 111000010100110$

Observe that w is the word corresponding to the cycle indicated on the left in the figure. u is the negation of w and corresponds to the remaining cycle. The words u, w are required words. The nodes of the cycle in G_6 related to w are:

0001 \rightarrow 0011 \rightarrow 0111 \rightarrow 1111 \rightarrow 1110 \rightarrow 1101 \rightarrow 1010 \rightarrow 0101
 \rightarrow 1011 \rightarrow 0110 \rightarrow 1100 \rightarrow 1001 \rightarrow 0010 \rightarrow 0100 \rightarrow 1000

This cycle, when nodes are written as decimal numbers is (see the figure)

1, 3, 7, 15, 14, 13, 10, 5, 11, 6, 12, 9, 24, 8,

Correctness of the algorithm follows directly from the following fact.

Fact. If α corresponds to a primitive polynomial then the word $\text{LFSR}(\alpha)$ and its bitwise negation have no common factor of length $n + 1$.

Proof We use the following property of α .

Claim. The number of 1's in α is even. The number of 1's in α_n cannot be odd, otherwise $\text{LFSR}(\alpha)$ would loop at 1^n

The proof is now by contradiction.

Assume $(n + 1)$ -length word vs , where s is a letter, is both in $\text{LFSR}(\alpha)$ and $\text{LFSR}(\bar{\alpha})$.

Then $vs, \bar{v}\bar{s} \in \text{LFSR}(\alpha)$. However the factors v and \bar{v} should be followed by the same letter, which follows from the claim and definition of LFSR - a contradiction. ■

Notes

The problem is related to the number of edge-disjoint simple cycles in de Bruijn graph. It is known that *maximal* number of such cycles in de Bruijn graph of rank n equals the number of conjugate (cyclic) classes of binary words of length n . It was a difficult problem known as Golomb's conjecture. We were interested here in *minimal* number of simple cycles containing all edges of de Bruijn graph. The considered problems is related to finding so called double helices. A double helix is a Hamiltonian cycle in de Bruijn such that after its deletion the remaining graph consists of one simple cycle and two loops. The notion of double helix was motivated by some problems in genetics. If we have two edge-disjoint semi-Hamiltonian cycles then it is easy to convert one of them into a double helix. Double helices were considered in the context of primitive polynomials in [42]. The algorithm presented here is algebraic and depends heavily on primitive polynomials. A different combinatorial construction (without use of primitive polynomials) of double helices was given in [47], where double helices correspond to so called complementary cycles: two Hamiltonian cycles of de Bruijn graph G_{n+1} which are edge-disjoint except 4 edges which are necessarily contained in each Hamiltonian cycle of G_{n+1} . Such two cycles can be trivially converted to edge-disjoint semi-Hamiltonian cycles. The algorithm presented here gives always words u, w which are negation of each other, the algorithm in [47] would give in many cases the words u, w not having this property.

150 Algorithm testing idempotent equivalence of words

We consider a relation between words of A^* , for a finite alphabet A , that identifies a square uu to its root u . More precisely, any factor u occurring in a word $x \in A^*$ can be replaced by uu , and any occurrence of uu can be replaced by u . Two words are idempotent equivalent if one can be transformed in the other using such replacements. This defines an equivalence relation \approx between words of A^* . For example, $\text{aababa} \approx \text{aba}$ since $\text{aababa} \approx \text{ababa} \approx \text{aba}$, and obviously $\text{a}^{10} \approx \text{a}^{111}$. A nontrivial example is $\text{bacbcabc} \approx \text{bacabc}$. For a given alphabet the number of equivalence classes is finite, but grows considerably fast. For alphabet sizes 1, 2, 3, 4, 5 the number of equivalence classes are respectively 1, 2, 7, 160, 332381. The goal of the problem is to design an efficient algorithm for testing the \approx -equivalence of two words. To do so, with each $x \in A^*$ is associated a (characteristic) quadruple $\Psi(x) = (p, a, b, q)$, where $a, b \in A$, pa is a shortest prefix and bq is a shortest suffix of x for which $\text{alph}(pa) = \text{alph}(bq) = \text{alph}(x)$ (Recall that $\text{alph}(u)$ is the set of letters occurring in u). For example, $\Psi(\text{ababbbcbcbcb}) = (\text{ababbb}, \text{c}, \text{a}, \text{bbcbcbcb})$. The sought algorithm is based on the following result (see Notes for reference).

Lemma 5 (Equivalence Criterion)

Let $x, y \in A^*$ be two words and their quadruples $\Psi(x) = (p, a, b, q)$ and $\Psi(y) = (p', a', b', q')$. Then, $x \approx y$ iff $p \approx p'$, $a = a'$, $b = b'$ and $q \approx q'$.

For example $\text{bacbcabc} \approx \text{bacabc}$ since $\Psi(\text{bacbcabc}) = (\text{ba}, \text{c}, \text{a}, \text{bc}) = \Psi(\text{bacabc})$.

Question. Assuming A is an integer alphabet (sortable in linear time), show how to check if $x \approx y$ in $(n \cdot |A|)$ time, where $n = |x| + |y|$.

Solution

We assume $\text{alph}(x) = \text{alph}(y)$ since otherwise x, y are certainly not equivalent.

First, we change the problem to testing the equivalence of two factors x, y of the same word $z = x\$y$, where $\$$ is new symbol. Observe that $\Psi(z) = (x, \$, \$, y)$.

Next, we restrict the set of factors of z as follows. Let $R(u) = |\text{alph}(u)|$ the rank of a word u . We say that a proper factor $z[i..j]$ of z is essential if

$$R(z[i..j]) + 1 = R(z[i..j+1]) \text{ or } R(z[i..j]) + 1 = R(z[i-1..j]).$$

Denote by E and E_k the set of all and of rank k essential factors of z respectively. Note that x and y are essential factors of z .

Data structure. Our main data structure to answer the question consists of collections of tables of two types: for each $k < |A|$, when $z[i..j] \in E_k$

$$RIGHT_k[i] = j \text{ and } LEFT_k[j] = i.$$

The tables are used to compute the quadruple of each $z[r..s] \in E_{k+1}$:

$$\Psi(z[r..s]) = (z[r..r'], z[r' + 1], z[s' - 1], z[s'..s]),$$

where $r' = RIGHT_k[r], s' = LEFT_k[s]$.

Note that all the tables and quadruples can be computed in total time $O(n \cdot |A|)$ since there are only $O(n \cdot |A|)$ essential factors.

Sketch of algorithm. It is based on a dynamic programming technique to compute, for each essential factor u , an identifier $ID(u)$ of its equivalence class. It is an integer in the range $[1..n]$ that must satisfy the condition: if $u, v \in E_k$

$$(*) \ u \approx v \iff ID(u) = ID(v).$$

The main step implements the Equivalence Criterion in Lemma 5.

```

EQUIVALENCE( $x, y$  words in  $A^+$ )
1   $z \leftarrow x\$y$ 
2  compute all tables  $RIGHT$  and  $LEFT$ 
3  for all  $u \in E_1$  do
4      compute  $ID(u)$ 
5  for  $k \leftarrow 2$  to  $|A|$  do
6      for all  $u \in E_k$  do
7           $\Psi(u) \leftarrow$  quadruple  $(p, a, b, q)$  corresponding to  $u$ 
8          radix-sort all quadruples and give the same  $ID$ 
              to words having the same quadruple  $\Psi$ 
9  return  $ID(x) = ID(y)$ 

```

The computation at lines 3-4 is straightforward because there factors of z have length 1. The whole computation has the required running time, mostly because both each radix-sort works in time $O(|E_k|)$ and we have $|E_k| = O(n)$ for each k .

Notes

The Equivalence Criterion Lemma is stated in [39]. The above computation is similar to the computation of the Dictionary of Basic factors of a word (see [12, Problem 66] or [14]). The present algorithm is a version of the one given in [45].

Bibliography

- [1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974. 36
- [2] M. Alzamel, A. Conte, S. Denzumi, R. Grossi, C. S. Iliopoulos, K. Kurita, and K. Wasa. Finding the anticover of a string. In I. L. Gørtz and O. Weimann, editors, 31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark, volume 161 of LIPIcs, pages 2:1–2:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. 31
- [3] G. Badkobeh, G. Fici, and S. J. Puglisi. Algorithms for anti-powers in strings. *Inf. Process. Lett.*, 137:57–60, 2018. 31
- [4] H. Bannai, T. I. S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta. The "runs" theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017. 51
- [5] H. Bannai, T. I. T. Kociumaka, D. Köppl, and S. J. Puglisi. Computing longest (common) Lyndon subsequences. In C. Bazgan and H. Fernau, editors, Combinatorial Algorithms - 33rd International Workshop, IWOCA 2022, Trier, Germany, June 7-9, 2022, Proceedings, volume 13270 of Lecture Notes in Computer Science, pages 128–142. Springer, 2022. 40
- [6] H. Bannai, T. I. T. Köppl. Longest bordered and periodic subsequences. *Inf. Process. Lett.*, 182:106398, 2023. 40
- [7] E. Berlekamp. Algebraic coding Theory. McGraw-Hill, 1968. 60
- [8] J. Berstel and L. Boasson. Partial words and a theorem of Fine and Wilf. *Theor. Comput. Sci.*, 218(1):135–141, 1999. 15
- [9] P. Charalampopoulos, S. P. Pissis, J. Radoszewski, W. Rytter, T. Walen, and W. Zuba. Subsequence covers of words. In D. Arroyuelo and B. Poblete, editors, String Processing and Information Retrieval - 29th International Symposium, SPIRE 2022, Concepción, Chile, November 8-10, 2022, Proceedings, volume 13617 of Lecture Notes in Computer Science, pages 3–15. Springer, 2022. 3
- [10] R. Cole, L. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In L. Babai, editor, Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004, pages 91–100. ACM, 2004. 16, 17

- [11] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. 392 pages. 24, 38
- [12] M. Crochemore, T. Lecroq, and W. Rytter. *125 Problems in Text Algorithms—with solutions*. Cambridge University Press, 2021. 334 pages. i, 7, 8, 14, 21, 24, 31, 32, 35, 36, 38, 41, 42, 43, 49, 61, 65
- [13] M. Crochemore and L. M. S. Russo. Cartesian and Lyndon trees. *Theoretical Computer Science*, 806:1–9, February 2020. <http://arxiv.org/abs/1712.08749>. 51
- [14] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994. 412 pages. 65
- [15] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific Publishing, Hong-Kong, 2002. 310 pages. 38
- [16] M. Debski, U. Pastwa, and K. Wesek. Grasshopper avoidance of patterns. *Electron. J. Comb.*, 23(4):4, 2016. 45
- [17] G. Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. ACM*, 20(3):500–513, 1973. 28
- [18] A. Flaxman, A. W. Harrow, and G. B. Sorkin. Strings with maximally many distinct subsequences and substrings. *Electron. J. Comb.*, 11(1), 2004. 43
- [19] J. Fuchs and P. Whittington. The 2-attractor problem is NP-complete. In O. Beyersdorff, M. M. Kanté, O. Kupferman, and D. Lokshtanov, editors, *41st International Symposium on Theoretical Aspects of Computer Science, STACS 2024, March 12-14, 2024, Clermont-Ferrand, France, volume 289 of LIPIcs*, pages 35:1–35:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. 7
- [20] D. Gabric, S. Holub, and J. O. Shallit. Maximal state complexity and generalized de Bruijn words. *Inf. Comput.*, 284:104689, 2022. 9
- [21] D. Gabric and J. Sawada. Efficient construction of long orientable sequences. *CoRR*, abs/2401.14341, 2024. 9
- [22] D. Gabric and J. O. Shallit. Borders, palindrome prefixes, and square prefixes. *Inf. Process. Lett.*, 165:106027, 2021. 48
- [23] A. L. L. Gao, S. Kitaev, W. Steiner, and P. B. Zhang. On a greedy algorithm to construct universal cycles for permutations. *Int. J. Found. Comput. Sci.*, 30(1):61–72, 2019. 57
- [24] P. Gawrychowski, M. Kosche, T. Koł, F. Manea, and S. Siemer. Efficiently testing Simon’s congruence. In M. Bläser and B. Monmege, editors, *STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*, volume 187 of *LIPIcs*, pages 34:1–34:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. 36

- [25] J. Grytczuk, J. Kozik, and P. Micek. New approach to nonrepetitive sequences. *Random Struct. Algorithms*, 42(2):214–225, 2013. 53
- [26] R. W. Hamming. Error detecting and error correcting codes. In Bell, editor, *Bell System Tech. J.*, volume 29, page 147–160, 1950. 20
- [27] T. Harju and D. Nowotka. Counting bordered and primitive words with a fixed weight. *Theor. Comput. Sci.*, 340(1):273–279, 2005. 48
- [28] B. R. Heap. Permutations by interchanges. *Comput. J.*, 6(3):293–298, 1963. 27
- [29] C. Hohlweg and C. Reutenauer. Lyndon words, permutations and trees. *Theor. Comput. Sci.*, 307(1):173–178, 2003. 51
- [30] T. Inoue, S. Inenaga, and H. Bannai. Longest square subsequence problem revisited. *CoRR*, abs/2006.00216, 2020. 40
- [31] A. Jez. Faster fully compressed pattern matching by recompression. *ACM Trans. Algorithms*, 11(3):20:1–20:43, 2015. 24
- [32] A. Jez. Recompression: A simple and powerful technique for word equations. *J. ACM*, 63(1):4:1–4:51, 2016. 13
- [33] J. R. Johnson. Universal cycles for permutations. *Discret. Math.*, 309(17):5264–5270, 2009. 57
- [34] D. Kempa and N. Prezza. At the roots of dictionary compression: string attractors. In I. Diakonikolas, D. Kempe, and M. Henzinger, editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 827–840. ACM, 2018. 7
- [35] V. Keränen. Abelian squares are avoidable on 4 letters. In W. Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13-17, 1992, Proceedings*, volume 623 of *Lecture Notes in Computer Science*, pages 41–52. Springer, 1992. 54
- [36] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen. A periodicity lemma for partial words. *Inf. Comput.*, 283:104677, 2022. 15
- [37] A. Kosowski. An efficient algorithm for the longest tandem scattered subsequence problem. In A. Apostolico and M. Melucci, editors, *String Processing and Information Retrieval, 11th International Conference, SPIRE 2004, Padova, Italy, October 5-8, 2004, Proceedings*, volume 3246 of *Lecture Notes in Computer Science*, pages 93–100. Springer, 2004. 40
- [38] K. Kutsukake, T. Matsumoto, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. On repetitiveness measures of Thue-Morse words. In C. Boucher and S. V. Thankachan, editors, *String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings*, volume 12303 of *Lecture Notes in Computer Science*, pages 213–220. Springer, 2020. 7

- [39] M. Lothaire. *Combinatorics on Words*. Addison-Wesley, 1983. Reprinted in 1997. 65
- [40] M. Lothaire. *Algebraic Combinatorics on Words*. *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2002. 24
- [41] S. Mantaci, A. Restivo, G. Romana, G. Rosone, and M. Sciortino. A combinatorial view on string attractors. *Theor. Comput. Sci.*, 850:236–248, 2021. 7
- [42] T. R. McConnell. DeBruijn strings, double helices, and the Ehrenfeucht-Mycielski mechanism. *CoRR*, 1303.6820, 2013. 63
- [43] A. Miller. Asymptotic bounds for permutations containing many different patterns. *J. Comb. Theory, Ser. A*, 116(1):92–108, 2009. 34
- [44] S. G. Park, A. Amir, G. M. Landau, and K. Park. Cartesian tree matching and indexing. In N. Pisanti and S. P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy, volume 128 of LIPIcs*, pages 16:1–16:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. 51
- [45] J. Radoszewski and W. Rytter. Efficient testing of equivalence of words in a free idempotent semigroup. In J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, and B. Rumpe, editors, *SOFSEM 2010: Theory and Practice of Computer Science*, volume 5901 of *Lecture Notes in Computer Science*, pages 663–671. Springer, 2010. 65
- [46] N. Rampersad, J. O. Shallit, and M. Wang. Inverse star, borders, and palstars. *Inf. Process. Lett.*, 111(9):420–422, 2011. 47
- [47] D. Repke and W. Rytter. On semi-perfect de Bruijn words. *Theor. Comput. Sci.*, 720:55–63, 2018. 63
- [48] M. Rosenfeld. Avoiding squares over words with lists of size three amongst four symbols. *CoRR*, abs/2104.09965, 2021. 53
- [49] J. Sawada and A. Williams. Greedy flipping of pancakes and burnt pancakes. *Discret. Appl. Math.*, 210:61–74, 2016. 28
- [50] C. E. Shannon. A mathematical theory of communication. *Bell Syst. Tech. J.*, 27(4):623–656, 1948. 22
- [51] A. M. Shur. On ternary square-free circular words. *Electron. J. Comb.*, 17(1), 2010. 54
- [52] S. Song, G. Gu, C. Ryu, S. Faro, T. Lecroq, and K. Park. Fast algorithms for single and multiple pattern cartesian tree matching. *Theor. Comput. Sci.*, 849:47–63, 2021. 51
- [53] J. Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980. 51
- [54] M. Yoeli. Binary ring sequences. *The American Mathematical Monthly*, 69(9):852–855, November 1962. 9

- [55] S. Zaks. A new algorithm for generation of permutations. BIT, 24(2):196–204, 1984. 28

Index