ФИЗИКА ЭЛЕМЕНТАРНЫХ ЧАСТИЦ И АТОМНОГО ЯДРА
2002. Т. 33. ВЫП. 7

УДК 539.12.01

GENERATING THE FIBONACCI CHAIN IN $O(\log n)$ SPACE AND $O(n)$ TIME

J. Patera[*]

Department of Mathematics, Faculty of Nuclear Science and Physical Engineering, Czech Technical University, Praha, Czech Republic

On the example of the Fibonacci chain we show how to efficiently generate infinite words defined by substitutions. We introduce substitution trees and we present an algorithm that uses them to generate such infinite words. We use a stack to generate the words in a completely different manner than in traditional breadth-first tree traversals. We show that the algorithm has $O(\log n)$ space complexity and $O(n)$ time complexity, where n is the length of the generated word. The aperiodic Fibonacci chain is used for construction of aperiodic pseudorandom number generators.

INTRODUCTION

In this paper we study a special kind of trees that we call *substitution trees*. We use them to represent iterations of substitutions on finite alphabets. The specifity of substitution trees is that, although being infinite, they contain only a very small finite number of different two-level subtrees. We use substitution trees for efficient iteration of the substitutions, which they represent, to generate a possibly infinite word.

The concept of substitution trees can be applied to any substitution, but we will illustrate it on the Fibonacci substitution and the Fibonacci chain.

We further introduce a new algorithm for breadth-first tree traversal that perfectly suits substitution trees. The general breadth-first (also called level-order) traversal, as described, for example, by Knuth [3], uses a queue to store up to one entire level of the tree, i. e., the queue size is $O(n)$, if n is the number of nodes on the given level. Our new algorithm requires only a stack of size $O(\log n)$, i. e., its size is equal to the tree depth.

We also analyze the time complexity of the new traversal. We show that the time needed to traverse from a node to the adjacent node on the same level is not constant, but its average value is bounded. Thus n nodes can be traversed in $O(n)$ time.

[*]e-mail: patera@km1.fjfi.cvut.cz

The results of this paper can be used, for example, for exact generation of special aperiodic self-similar point sets, called *cut and project sequences* [4, 5]. These sequences are essential for the design of a new type of aperiodic pseudo-random number generators (APRNG), introduced in [1] and studied, for example, in [2].

1. PRELIMINARIES

A substitution θ is a mapping of a finite alphabet $\mathcal{A}$ into the set $\mathcal{A}^*$ of finite words in the letters of $\mathcal{A}$ such that $\theta(a)$ is nonempty for any $a \in \mathcal{A}$. In particular, the *Fibonacci substitution* is defined on the alphabet $\mathcal{A} = \{A, B\}$ with substitution rules $\theta(A) = ABA$ and $\theta(B) = AB$. A sequence of j-th iterations of the substitution on the letter A is a sequence of words $\theta^j(A)$ of increasing length such that $\theta^{j+1}(A) = \theta^j(A)\omega_j$ for some words ω_j. The infinite word $\theta^\infty(A) = ABAABABA...$, to which $\theta^j(A)$ converges, is invariant with respect to this substitution and is called the *Fibonacci chain*. $|w|$ for some word w denotes the number of letters in w, we call it the length of w. Further details on substitutions can be found, e. g., in [6], and particularly on the Fibonacci substitution in [4].

The *Fibonacci numbers* will be very often used in this paper, they are defined by the following recurrence:

$$F_j = F_{j-1} + F_{j-2}, \quad F_0 = 0, \quad F_1 = 1. \tag{1}$$

Solving this recurrence, one obtains

$$F_j = \sqrt{5}^{-1}(\tau^j - \tau'^j) \tag{2}$$

and

$$\tau^j = F_j\tau + F_{j-1},$$

where $\tau = \dfrac{1}{2}(1 + \sqrt{5})$ and $\tau' = \dfrac{1}{2}(1 - \sqrt{5})$.

Let us recall that from (2) it follows that $F_j = O(\tau^j)$. On the other hand, if $f(k)$ is increasing for $k \in \mathbb{N}$ and $f(F_j) = O(j)$, then $f(j) = O(\log j)$.

Proposition. *Let θ be the Fibonacci substitution. The lengths of words $\theta^j(A)$ are $|\theta^j(A)| = F_{2j+2}$ for all non negative integers j.*

Proof. Let $p_A(j)$ and $p_B(j)$ denote the number of letters A and B, respectively, in $\theta^j(A)$. Clearly, $p_A(j+1) = 2p_A(j)+p_B(j)$, $p_B(j+1) = p_A(j)+p_B(j)$, $p_A(0) = 1$, and $p_B(0) = 0$. It can be shown easily by induction, that $p_A(j+1) = F_{2j+1}$ and $p_B(j) = F_{2j}$ and thus $|\theta^j(A)| = F_{2j+1} + F_{2j} = F_{2j+2}$. ■

This proposition gives the number of iterations of θ that are needed to generate n elements. Indeed, it can be used to find j such that $F_{2(j-1)+2} < n \leq F_{2j+2}$. In Sec. 2, we present an algorithm requiring just $j-1$ memory cells (besides the memory needed for the substitution itself) to generate the first n elements of the sequence.

2. SUBSTITUTION TREES

In this section we introduce an efficient way of representing iterations of substitutions, the substitution trees, and we show how to use them to generate the Fibonacci chain. Let t_n, for $n \geq 0$, be the n-th letter of the Fibonacci chain, with $t_0 = A$ being the starting letter.

To the Fibonacci substitution, a *substitution tree* can be assigned. It is a tree with nodes and edges both labeled. The root node is labeled as A (A is the starting letter of the substitution) and every other node is labeled either as A or B. Every A node has $|\theta(A)| = 3$ children nodes, labeled A, B, A, respectively, in the given order according to the word $\theta(A) = ABA$, and every B node has $|\theta(B)| = 2$ children nodes, labeled according to $\theta(B) = AB$ (see Fig. 1). The edge connecting a node with its i-th child node is labeled with the index i. We call the given node with all its descendant nodes a subtree. The subtree rooting at any A node is isomorphic to the entire tree. Therefore the j-th level of the tree coincides with the beginning of the $(j+1)$-th level, i. e., $\theta^{j+1}(A) = \theta^j(A)\omega_j$ for certain word ω_j, and thus

$$\theta^{j+1}(A) = A\omega_0\omega_1\omega_2\ldots\omega_j. \tag{3}$$

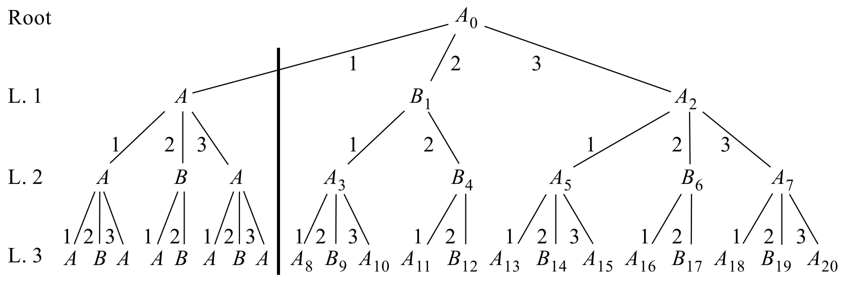
Root
L. 1
L. 2
L. 3
A_0
B_1
A_2
A_3
B_4
A_5
B_6
A_7
A_8 B_9 A_{10} A_{11} B_{12} A_{13} B_{14} A_{15} A_{16} B_{17} A_{18} B_{19} A_{20}

Fig. 1. The first 3 levels of the substitution tree for the Fibonacci substitution $\theta(A) = ABA$, $\theta(B) = AB$. The j-th level of the tree corresponds to the j-th iteration of θ. In generation of the j-th iteration of θ, the letters to the left of the thick line are skipped as they correspond to the already generated (j–1)-th level. To ease the notation, we write A_n or B_n if the n-th generated letter is A or B, respectively

Letters $t_0, \dots, t_n$ can be generated either by traversing a single sufficiently long level $\theta^j(A)$ from its left end, or using (3) by traversing the right parts of the levels of the tree, given by the words ω_j, starting at the first level.

Figure 1 shows the first three levels of the substitution tree for the Fibonacci substitution. The subscripts of nodes denote the order in which the letters t_n are generated.

We have described the idea of using substitution trees for generation. We further discuss technical issues of an implementation of the given algorithm.

Each of the substitution rules $\theta(A)$ and $\theta(B)$ appears repeatedly in many nodes, and therefore it is impractical to keep the entire tree in the computer memory or even to construct a long part of it, especially if the required length n of the Fibonacci chain is not known in advance (and thus the depth of the used part of the tree). The only data used for generation is a static set of substitution rules and a stack, as described below. The set of rules does not change within the generation although, the tree is being repeatedly extended by a new level. On the other hand, the stack grows by one item whenever a new tree level is started.

The way the substitution tree is traversed is the so-called *breadth-first traversal* which traditionally needs a *queue* to be performed. Instead of a queue, we use a stack in a completely different way than the stack is used in traditional breadth-first traversals [3]. The position of every node ν in a tree is uniquely determined by the path from ν to the root. The path is given by the sequence of all ancestors of ν and edges it passes through. The stack is here used to store the reversed sequence of nodes and edges from the root to the parent of the parent of ν. Thus the nodes on the two deepest levels are never stored in the stack. Each item of the stack consists of two parts $(letter, index)$: the node label and the label of the edge that leads to the node following in the path. We use the substitution rules represented by the nodes in the $(d-1)$-th level to generate the subordinate letters in the d-th level at once, without use of the stack. For example, when the letters $t_3 = A$ and $t_4 = B$ are being generated (refer to Fig. 1), the tree has two levels and the stack consists just of one item: $(A, 2)$, because t_3 and t_4 are generated by simple application of $\theta(B) = AB$ rooting at $t_1 = B$ in the first level; in this case $(A, 2)$ means the second child node of $t_0 = A$. Whereas when the third level is started and the letters $t_8 = A$, $t_9 = B$, and $t_{10} = A$ are generated, the stack has two items: $(A, 2)$ and $(B, 1)$, representing the second child node of A_0 and the first child of B_1.

The traversing algorithm works as follows. Since generally there are no direct links between adjacent nodes (i. e., through their common parent node) on any level, it is occasionally necessary to make a detour through a certain number of upper levels: to go up in the tree and then to go down again. For example, in Fig. 1 to go from B_{12} to A_{13} we must go back to the root. When the generator enters, say, the i-th subtree of a node ν, i. e., it goes down from ν to its subtree, then it pushes the node name ν and the index i to the stack. When all subtrees of ν of the given depth are traversed, it is necessary to pop the parent node and edge (a, i) from the stack. The popping is then repeated as long as $i = |\theta(a)|$ for the newly popped values of a and i. This condition checks whether the entire subtree of a was already traversed. After that the pair $(a, i+1)$ is pushed and the new a is defined as the letter $\theta(a)_{i+1}$. Finally it is necessary to go down the tree to reach the original level where the popping started. Every level stepped down is represented by pushing the pair $(a, 1)$ and defining new a as $\theta(a)_1$. If, in the above popping, the stack got empty and $i = |\theta(a)|$ still holds, then it means that the entire tree level was traversed, that a new one must be started, and that a new stack, with one more level than before, must be built. The stack is built in such a way that the first subtree of the tree root is skipped, i. e., the built path is the closest one to the right of the thick line in Fig. 1.

The generation algorithm is given formally in Algorithm 1. It generates the sequence $(t_i)_{i=0}^{N}$, where t_0 is the given starting letter A. The used $stack$ stores pairs $(letter, index)$. The variable l counts the number of stack items that were popped between generations of two consecutive letters and are thus necessary to be pushed back. If an entire level gets traversed, l is incremented. It signals that a deeper stack needs to be built because a new level is about to be started.

```
$stack := empty$; $a := t_0$; $n := 1$; $i := 2$;
while $(n \leq N)$ do
  while $(i \leq |\theta(a)|)$ do
    $t_n := \theta(a)_i$; $n := n+1$; $i := i+1$;
  enddo;
  $l := 0$;
  while $(not\ stack.isEmpty)$ and $(i > |\theta(a)|)$ do
    $stack.pop(a, i)$;
    $l := l + 1$; $i := i + 1$;
  enddo;
  if $(stack.isEmpty)$ then
    $a := t_0$; $i := 2$; $l := l + 1$;
  endif;
  repeat
    $stack.push(a, i)$;
    $a := \theta(a)_i$; $i := 1$; $l := l - 1$;
  until $(l = 0)$;
enddo;
```

Algorithm 1: Generating N letters of the Fibonacci chain using its substitution tree.

The fact that after generating n letters of the chain the stack contains as many pairs $(letter, index)$ as many levels the corresponding generated tree has, leads us to the following theorem.

Theorem 1. *The space complexity of Algorithm 1 for generation of n letters of the Fibonacci chain is $O(\log n)$.*

3. TIME COMPLEXITY OF SUBSTITUTION TREES

Performance of every algorithm should be studied from different points of view. In this section we examine the time characteristics of traversing substitution trees. We look at the average, best, and worst time needed for generation of a single letter.

The most time-consuming operations in the tree traversing are the stack operations. We will further refer to a pair of pushing and popping as to a single (stack) operation, assuming that the letter production time itself is neglectable.

The average number of operations per generated letter $\mathcal{M}(n)$ is the ratio of the total number of operations for all letters $\mathcal{N}(n)$ and the number of letters n. In the analysis of $\mathcal{M}(n)$, we will first consider the case $n = F_{2j+2}$, i. e., that an entire level is generated (see Proposition). When the right part ω_j of the j-th level, i. e., the letters t_i with $F_{2j} < i \le F_{2j+2}$, is being generated, every edge entering any node corresponding to t_i, $1 \le i \le F_{2j}$, is pushed to and popped from the stack, and it is done exactly once. Therefore the generation of the letters t_i, $F_{2j} < i \le F_{2j+2}$ requires $N_j := F_{2j}$ operations. In total, summing for all ω_j, generation of the sequence $(t_i)_{i=1}^{F_{2j+2}}$ requires

$$\mathcal{N}(F_{2j+2}) = \sum_{k=1}^{j} N_k = \sum_{k=1}^{j} F_{2k} < F_1 + \sum_{k=1}^{j} F_{2k} = F_{2j+1}$$

operations.

Obviously, $\mathcal{N}(n)$ is a nondecreasing function satisfying $\mathcal{N}(F_{2j+2}) \le F_{2j+1}$. Let now n be an arbitrary positive integer. We have $F_{2j} \le n \le F_{2j+2}$ for some j and therefore the average number of operations per letter is

$$\mathcal{M}(n) = \frac{\mathcal{N}(n)}{n} < \frac{\mathcal{N}(F_{2j+2})}{F_{2j}} < 2\tau\frac{\mathcal{N}(F_{2j+2})}{F_{2j+1}} < 2\tau\frac{F_{2j+1}}{F_{2j+1}} < 2\tau.$$

We used the estimate $F_{j+1}/F_j < 2\tau$ following from (2); in limit, the ratio F_{j+1}/F_j is τ.

These considerations lead us to the following theorem.

Theorem 2. *The Fibonacci chain can be generated using the substitution tree in $O(n)$ time, where n is the length of the generated word.*

We see that in average the number of operations per letter is uniformly bounded by a constant, independent of n. However, if $(t_i)_{i=0}^{n}$ has already been generated, the upper bound of number of operations needed to determine t_{n+1} is a function of n. When traversing a substitution tree, it is usually necessary to go up through the tree and then descend. For example, generating t_{13} from t_{12} requires going up to the tree root. The maximal number of operations per single letter is therefore equal to the tree depth, i. e., is proportional to $\log n$. On the other hand, the number of operations is 0 when t_n and t_{n+1} are two direct children nodes of the same node, such as t_{13} and t_{14}.

4. SKIPPING n LETTERS IN THE FIBONACCI CHAIN

Algorithm 1 in Sec. 2 generates the Fibonacci chain from the beginning. Some applications may require a sequence starting at an arbitrary n-th letter, for example the APRNG needs the ability of starting generation from any seed-point. This can be done naively by brute-force generating and discarding all the unnecessary letters. In this section we provide an efficient algorithm for starting generation from any n-th letter. At the end of the section we generalize it to the skipping of arbitrary n letters at any position.

Starting generation from the n-th letter is equivalent to building a stack simulating generation of the previous $n-1$ letters.

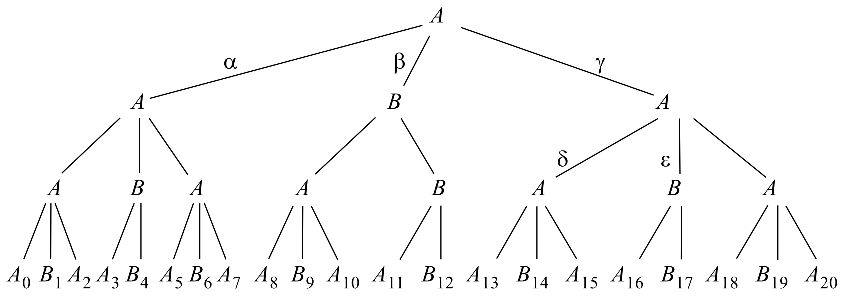
A
α
β
γ
A
B
A
δ
ε
A B A A B A B A
A0 B1 A2 A3 B4 A5 B6 A7 A8 B9 A10 A11 B12 A13 B14 A15 A16 B17 A18 B19 A20

Fig. 2. The process of building a stack for the generation starting from an n-th letter. It is illustrated on the letter t_{17} of the substitution tree for the Fibonacci substitution. The nodes α, β, γ, δ, and ε must be traversed before the letter t_{17} is reached

In the substitution tree for the Fibonacci substitution, the number of letters on any level of any subtree can always be determined. Let $\ell : \{A, B\} \times \mathbb{N} \to \mathbb{N}$ be defined as follows:

$$\ell(a, j) := \begin{cases} F_{2j+2}, & \text{for } a = A \\ F_{2j+1}, & \text{for } a = B \end{cases} .$$

According to Proposition and its generalization for letter B, $\ell(a,j)$ gives the number of letters on the j-th level of the tree for $\theta^j(a)$, i. e., $\ell(a,j) = |\theta^j(a)|$.

Let us assume that we want to skip n letters, i. e., to start generation from the $(n+1)$-th letter. We determine the level on which the $(n+1)$-th letter resides. We then build the path to it from the tree root and we store the path in the stack. The resulting stack is the desired stack. We illustrate the stack building on $n = 17$. The algorithm has three parts (refer to Fig. 2):

• Determine the tree depth d. Find d such that $|\theta^{d-1}(A)| \leq n < |\theta^d(A)|$, then the tree will have d levels and the constructed stack will have $d-1$ items. In our case, $d = 3$ since $(8 =)\ F_{2\cdot2+2} \leq n < F_{2\cdot3+2}\ (= 21)$.

• Build the stack by a single deterministic walk down the tree from the top to the right bottom, starting at the first node of the first level. In every node ν determine, using the ℓ function, whether the finite subtree rooting at this node should be skipped entirely or if it contains the $(n+1)$-th letter. If it does contain the letter, the path continues in this subtree and therefore we push the position of this descendant node (i. e., its parent ν and the edge connecting them) into the stack and we enter the subtree on the left-most node of its first level. If it does not contain the letter, we repeat the procedure on the right sibling node of ν (there always is such a node, this follows from the way the path is determined). The procedure is repeated recursively until the stack has $d-1$ items. We always start on the left-most node of the first level.

In our example, the left-most node is denoted by α. α contains $\ell(A,2) = 8$ letters which is less than 17. Therefore we skip it and proceed to the subtree root β. It contains $\ell(B,2) = 5$ letters which is less than $n - \ell(A,2) = 17 - 8 = 9$, therefore we also skip it. The subtree γ contains $\ell(A,2) = 8$ letters. It is more than $9 - 5 = 4$, therefore the path continues in this subtree. We push the pair $(A,3)$, the position of γ, into the stack and we proceed to the subtree δ. It contains $\ell(A,1) = 3$ letters, less than 4, so we proceed to ε. This subtree contains $\ell(B,1) = 2$ letters, more than $4 - 3 = 1$, therefore the path continues in this subtree. We push the pair $(A,2)$ into the stack, because we are going down to the subtree ε. The stack now contains $d-1 = 2$ items, therefore we reached the bottom of the tree and the stack is $(A,3)$, $(A,2)$.

• Finally, we determine the starting position in the substitution rule we found by skipping the remaining letters from the beginning. In our example, the first letter to be generated is $\theta(B)_2$ because $n - 8 - 5 - 3 = 1$.

The formal version of this informal description is shown in Algorithm 2. The lenghts of words $\theta^j(a)$ can be simply calculated by $\ell(a,j)$.

The algorithm just given can be generalized easily to skipping of arbitrary n letters from arbitrary position in the sequence. Indeed, instead of starting with empty stack and descending in the tree, we can use an already existing stack and start by climbing up the tree, using a procedure similar to the one mentioned in the second item of the above given description. After a certain number of steps, the direction of traversing will change and we will start going down toward the deepest level. It may happen that it is required to skip more letters than the number of letters remaining on the deepest level is. In such a case, a slight modification of Algorithm 2 can be applied to the remaining letters that did not fit in the previous level. The modification consists in skipping the first left subtree α of the tree root.

Theorem 3. *Skipping n letters in the Fibonacci chain requires $O(\log n)$ operations.*

```
$Stack := empty$; $a := t_0$;
**if** $(n > |\theta(a)|)$ **then**
  Find $d$ such that $|\theta^{d-1}(a)| \leq n < |\theta^d(a)|$;
  **repeat**
    $i := 1$; $d := d - 1$;
    **while** $(|\theta^d(\theta(a)_i)| \leq n)$ **do**
      $n := n - |\theta^d(\theta(a)_i)|$;
      $i := i + 1$;
    **enddo**;
    $stack.push(a, i)$;
    $a := \theta(a)_i$;
  **until** $(d = 1)$;
**endif**;
Start generating with $\theta(a)_{n+1}$.
```

Algorithm 2: Skipping n letters and creating the stack.

CONCLUSION

The concept of substitution trees and their traversals can be, without any modifications, applied to an arbitrary substitution on any finite alphabet. The cut and project sequences [5] used in the APRNG design [1] have generally much larger alphabets than just two letters. In any case, the $O(n)$ time and $O(\log n)$ space complexity of generation is preserved. For skipping n letters, the ability to determine the exact number of letters on any level of any tree is essential. In general, multiplications of matrices of substitutions [4] may be needed rather than a simple calculation of a recurrent sequence like (1). Besides the matrix operations, the algorithm also has $O(\log n)$ time complexity.

Skipping n letters does not necessarily require building the entire $O(\log n)$ stack. In cases when only few points are to be generated (compared to the number of letters to be skipped), a certain number of the top items will never be used. Indeed, it may happen that only a minor part of a single tree level is to be used without the need of going too far towards the tree root. If the upper bound on the number of letters to be generated is known in advance, Algorithm 2 can be modified in such a way that the stack items corresponding to the upper tree levels will not be pushed into the stack. If, for example, at most m letters starting at the letter t_n are about to be generated, the *stack. push* operation in Algorithm 2 can be ignored (i. e., not performed) as long as t_n and t_{n+m} are in the same subtree of the node currently traversed in the while loop of Algorithm 2.

Let us conclude with some numbers. In our most efficient implementation of the presented tree traversal, a single stack item consists just of one memory pointer. On a 32-bit computer, a 1 KB stack can be used to generate $F_{514} \sim 10^{107}$ letters of the Fibonacci chain, i. e., 10^{107} elements of a binary aperiodic sequence usable by the APRNG.

Acknowledgements. The author would like to thank L.-S. Guimond, Z. Masáková, and E. Pelantová for their stimulating discussions and many suggestions to the manuscript. This work was supported by the Bell Canada University Laboratory, by the NATO Collaborative Research Grant CRG 974230 and by the grant FRVŠ 1927/2001 of the Czech Ministry of Education. The author is also grateful for the hospitality of the Centre de Recherches Mathématiques in Montréal (Québec, Canada), where part of the work was done.

REFERENCES

1. *Guimond L.-S., Patera J.* Combining Random Number Generators Using Cut and Project Sequences // Czech. J. Phys. 2001. V. 26. P. 305–311.
2. *Guimond L.-S., Patera J.* Statistics and Implementation of Aperiodic Pseudorandom Number Generators. Preprint. 2000. Subm. to «Appl. Num. Math.».
3. *Knuth D. E.* The Art of Computer Programming: Fundamental Algorithms. 3rd ed. Addison-Wesley, 1997. V. 2.
4. *Luck J. M. et al.* The Nature of the Atomic Surfaces of Quasiperiodic Self-Similar Structures // J. Phys. A: Math. Gen. 1993. V. 26. P. 1951–1999.
5. *Masáková Z., Patera Jiří, Pelantová E.* Substitution Rules for Aperiodic Sequences of the Cut and Project Type // J. Phys. A: Math. Gen. 2000. V. 33. P. 8867–8886.
6. *Queffélec M.* Substitution Dynamical Systems: Spectral Analysis. Berlin: Springer, 1987.