Abstract

Data compression has been widely applied in many data processing areas. Compression methods utilize variable-length codes with shorter codewords assigned to symbols or groups of symbols that appear in the data frequently. Fibonacci, Elias-delta, and Elias-Fibonacci codes are representatives of these codes, and are often utilized for the compression of numbers. The time consumption of encoding as well as decoding algorithms is important for some applications in the data processing area. In this case, the efficiency of these algorithms is extremely important. Some works related to fast decoding of variable-length codes have been published in recent years. Fast encoding algorithms for these codes have not yet been studied. In this work, we introduce fast encoding and decoding algorithms. Our fast encoding algorithms are up to $12.5\times$ faster than conventional bit-oriented algorithms and our decoding algorithms are up-to $8.9\times$ faster than the conventional decoding algorithms.

Keywords: compression, variable-length code, fast encoding algorithm, Elias-delta code, Fibonacci codes of order 2 and 3, Elias-Fibonacci code
Abstrakt

Při zpracování dat jsou široce využívány kompresní metody. Tyto metody nejčastěji využívají kódy s proměnnou délkou, kdy kratší kóдовá slova jsou přiřazována symbolem nebo skupinám symbolů s vyšší frekvencí výskytu v kódovaných datech. K zástupcům těchto kódů patří kódy Fibonacci, Elias-delta a Elias-Fibonacci, které jsou využity i v této práci. Pro některé aplikace zpracování dat je časová náročnost a efektivita kódování a dekódování velmi důležitá, a proto byly nedávno publikovány práce zabývající se metodami rychlého dekódování. Metody pro rychlé kódování využívající výše zmíněných kódů prozatím publikovány nebyly. V této práci jsou představeny nové metody rychlého kódování a dekódování pro kódy Fibonacci, Elias-delta a Elias-Fibonacci. Rychlé kodovací algoritmy jsou až $12.5 \times$ rychlejší než běžné používané kódovací algoritmy a rychlé dekódovací algoritmy jsou až $8.9 \times$ rychlejší než běžné používané dekódovací algoritmy.

**Keywords:** komprese, kódy s proměnnou délkou, rychlé kódovací algoritmy, rychlé dekódovací algoritmy, Elias-delta, Fibonacci, Elias-Fibonacci
Acknowledgment

My thanks belongs to Doc. Michal Krátký for introducing me into this topic, for many useful comments regarding fast encoding and decoding algorithms, and helpful discussions from which many ideas have arisen.

My thanks belongs also to my family which supported me during all my studies and to my beloved girlfriend and to my one year old son who were so patient with me during long nights and days of working on this thesis.
Prohlášení

Prohlašuji, že jsem tuto disertační práci vypracoval samostatně s použitím odborné literatury a dalších informačních zdroj v této práci uvedené.

V Ostravě
Contents

I Introduction 1

1 Introduction 2

2 Preliminaries 4

II Variable-Length Codes 7

3 Variable-Length Codes 8

3.1 Outlines 8

3.2 Unary Code 8

3.3 Static and Adaptive Huffman Code 9

3.4 Golomb Codes 14

3.5 Elias Codes 15

3.5.1 Elias-gamma Code 15

3.5.2 Elias-delta Code 16

3.5.3 Elias-omega Code 16

3.5.4 Comparing Elias Codes 17

3.6 Fibonacci Codes 17

3.6.1 Fibonacci Code of Order 2 19

3.6.2 Fibonacci Codes of Higher Orders 21

3.7 Elias-Fibonacci Code 23

3.8 Comparison and Selection of Codes 24

4 Preliminaries of Encoding and Decoding Algorithms 28

5 Conventional Encoding and Decoding Algorithms 31

5.1 Introduction 31
5.2 Conventional Algorithms for Elias-delta Code .................. 31
5.3 Conventional Algorithms for Fibonacci Code of Order 2 .......... 32
5.4 Conventional Algorithms for Fibonacci Code of Order 3 .......... 34
5.5 Conventional Algorithms for Elias-Fibonacci Code ............... 36

III Fast Encoding and Decoding Algorithms 39

6 Prerequisites of Fast Fibonacci Encoding and Decoding Algorithms 40
6.1 Introduction ............................................. 40
6.2 Properties of Fibonacci Numbers .......................... 40
6.3 Interval of Numbers with the Same Bit-length of the Fibonacci Representation ........................................... 41
6.4 Extended Fibonacci Representation .......................... 42
6.5 Fibonacci Shift Operations and their Computation ............... 43
   6.5.1 Fibonacci Shift Operations ........................... 43
   6.5.2 Conventional Computation of Shift Operations .......... 44
   6.5.3 Efficient Computation of Fibonacci Right Shift ........ 45
   6.5.4 Efficient Computation of Extended Fibonacci Right Shift . 50
   6.5.5 Efficient Computation of Fibonacci Left Shift .......... 52
6.6 Efficient Computation of Nearest Lower Fibonacci Sum ......... 55

7 Fast Encoding Algorithms ........................................ 60
7.1 Introduction ............................................. 60
7.2 Fast Elias-delta and Elias-Fibonacci Encoding Algorithms ....... 60
7.3 Fast Encoding of Fibonacci Code of Order 2 .................... 62
7.4 Fast Encoding of Fibonacci Code of Order 3 .................... 65

8 General Principles of Fast Decoding Algorithms .................. 68
8.1 Introduction ............................................. 68
8.2 General Fast Decoding Algorithm ........................... 68
8.3 Identification of Automaton States with Brute-force Approach .... 70
8.4 Mapping Table Building .................................... 72
8.5 Automation Reduction Using Similarity of States ................ 73
   8.5.1 Automaton States of Elias-delta Code .................. 74
8.5.2 Automaton States for Elias-Fibonacci Code ................. 76
8.5.3 Automaton States for Fibonacci Codes ....................... 78
8.6 Automaton Reduction Using Shift Operation .................. 79
  8.6.1 Reduction Using Binary Shift Operation .................. 79
  8.6.2 Reduction Using Fibonacci Left Shift Operation .......... 82
8.7 Number of States After Reductions ............................ 85

9 Fast Decoding Algorithms ........................................ 87
  9.1 Outlines ..................................................... 87
  9.2 Fast Elias-delta and Elias-Fibonacci Decoding Algorithms .. 87
  9.3 Fast Fibonacci Decoding Algorithm .......................... 91
  9.4 Comparison with Other Works ................................. 93

IV Experiments and Applications .................................... 95

10 Experimental Results ............................................. 96
  10.1 Test Collections .............................................. 96
  10.2 Compression Ratio ............................................ 97
  10.3 Encoding Algorithms .......................................... 98
  10.4 Decoding Algorithms .......................................... 100
  10.5 Who Is the Winner? ............................................ 102

11 Applications of Fast Encoding and Decoding Algorithms ...... 104
  11.1 Compression of XML Node Streams ........................... 104
    11.1.1 Introduction ............................................. 104
    11.1.2 XML Model .............................................. 104
    11.1.3 Stream ADT .............................................. 105
    11.1.4 Compression of XML Node Streams ....................... 107
    11.1.5 Experimental Results .................................. 109
  11.2 Text Compression ............................................. 111
    11.2.1 Simple Dictionary-Based Compression Method .......... 112
    11.2.2 Experiments ............................................. 113

12 Conclusion ....................................................... 115
Nomenclature

Operators

<< the binary left shift

<<\_F the Fibonacci left shift

>> the binary right shift

>>\_F the Fibonacci right shift

>>\_F\_ext the Extended Fibonacci right shift

\[n\] truncation of the decimal part of a number \(n\)

\[\|n\|\] rounding of a real number \(n\)

\(AND\) the binary AND operation

\(NOT\) the binary NOT operation

\(OR\) the binary OR operation

\(*\) 1’s complement of the standard binary representation

Acronyms

\(0\_m\) \(m\) consecutive 0-bits

\(1\_m\) \(m\) consecutive 1-bits

\(F^{(m)}(n)\) the reversed Fibonacci code of order \(m\) for a number \(n\)

\(F^{(m)}_C(n)\) the Fibonacci code of order \(m\) for a number \(n\) originally defined by Apostolico and Fraenkel [12]

\(B'(n)\) the standard binary representation of a number \(n\) without the highest 1-bit

\(B(n)\) the standard binary representation of a number \(n\)

\(B_c(n)\) the standard binary representation of a number \(n\) with the fixed bit-length \(c\)

\(E_\delta(n)\) the Elias-delta code of a number \(n\)
\( E_{\gamma}(n) \) the Elias-gamma code of a number \( n \)

\( E_{\omega}(n) \) the Elias-omega code of a number \( n \)

\( EF(n) \) the Elias-Fibonacci code of a number \( n \)

\( ExFRSEIT \) the Extended Fibonacci right shift encoding interval table

\( f(n) \) the frequency of occurrence of a number \( n \)

\( F^{(m)}(n) \) the Fibonacci representation of order \( m \) for a number \( n \)

\( F_i \) the \( i \)-th Fibonacci number without any specification of order \( m \)

\( F_i^{(m)} \) the \( i \)-th Fibonacci number of order \( m \)

\( FL_{F_{L_{\max}}} \) the Fibonacci number with index \( L_{FL_{\max}} \)

\( FRSEIT \) the Fibonacci right shift encoding interval table

\( g \) the index of the Fibonacci sum

\( G(n) \) the Golomb code of a number \( n \)

\( H_s(n) \) the static Huffman code of a number \( n \)

\( k \) the parameter of the Fibonacci left or right shift

\( L \) \( L(n) \), the bit-length of \( n \)

\( L(F(n)) \) the bit-length of \( F(n) \)

\( L(n) \) the bit-length of \( n \)

\( L_F \) \( L(F(n)) \) – the bit-length of \( F(n) \)

\( L_{\max} \) the maximal bit-length of the maximal number (the bit-length of \( n_{\max} \))

\( L_{FL_{\max}} \) the maximal bit-length of the Fibonacci representation of \( L_{\max} \), i.e. \( L_{FL_{\max}} = L(F(L_{\max})) = L(F(L(n_{\max}))) \)

\( m \) the parameter of the Golomb code or the order of the Fibonacci representation or the Fibonacci code

\( n \) a number

\( n_{\max} \) a maximal number

\( NYT \) not yet transmitted – a special codeword in the Adaptive Huffman tree

\( P \) probability of occurrence

\( S \) the segment bit-length (chunk of bits of the fixed bit-length \( S \))

\( S_g^{(m)} \) the \( g \)-th Fibonacci sum of order \( m \)
$S_{MAP}$ the size of a mapping table

$U(n)$ the Unary code of a number $n$

$MAP$ a mapping table used in fast decoding algorithms

**Subscripts**

10 the standard decimal representation

2 the standard binary representation

$F^{(m)}$ a codeword of the reversed Fibonacci code of order $m$

$E_\delta$ a codeword of the Elias-delta code

$E_\gamma$ a codeword of the Elias-gamma code

$E_\omega$ a codeword of the Elias-omega code

$EF$ a codeword of the Elias-Fibonacci code

$F^{(m)}$ the Fibonacci representation of order $m$

$G$ a codeword of the Golomb code

$H_s$ a codeword of the static Huffman code

$U$ a codeword of the Unary code
Part I

Introduction
Chapter 1

Introduction

Data compression has been widely applied in many data processing areas. Various compression algorithms have been developed for processing text documents, images, video, etc. In particular, data compression is of the foremost importance and has been well researched as it has been presented in excellent books [54, 67].

In contrast with fixed-length codes, statistical methods utilize variable-length codes [55], with the shorter codes assigned to symbols or groups of symbols that have a higher probability of occurrence. People who design and implement variable-length codes have to deal with these two problems: (1) assigning codes that are uniquely decodable and (2) assigning codes with the minimum average size. Several variable-length codes such as Elias [23], Fibonacci [26, 12], Elias-Fibonacci [4], Golomb [30, 66], and Huffman codes [33] are well known representatives of variable-length codes.

Fast encoding/decoding algorithms have been intensively studied in a connection with video and audio encoders/decoders. Moreover, these studies often integrate the variable-length codes in the encoders and decoders. The result is the following recommendations: T.81 for a JPEG encoder [36] and H.261 for an MPEG encoder [35]. These encoders utilize Exponential Golomb [63] or Huffman codes. Many works and patents (e.g. [27, 38, 61]) deal with fast encoders/decoders for these codes. In this case, a codeword is predefined for each coded number; the size of the encoding/decoding table then corresponds to the size of the number domain used. Therefore, these algorithms can be used only for small domains of coded numbers, e.g. 8 or 16 bits; they are not useful for general universal codes where the domain can be large (e.g. 32 bit-length numbers). Since these domains are often used in the data processing area or text file compression [66, 58], other algorithms utilizing more general variable-length codes are investigated. Compression properties of the variable-length codes have been compared in several works [64, 3]. In these articles, Fibonacci, Elias-delta, and Elias-Fibonacci codes overcome other variable-length codes.

The time consumption of compression and decompression is sometimes critical; therefore, efficient decompression algorithms have been studied in many works related to decompression of data structures [56, 29, 21, 2] or text files [50, 16]. In the case of physical implementation of database systems [46], retrieval of compressed
data structure’s pages can be more efficient than retrieval of uncompressed pages due to the fact that the cost of decompression can be lower than the cost of page accesses to the secondary storage [3, 1]. Therefore, variable-length codes can be utilized for compression of data structure’s pages. In this case, pages are decompressed during every reading from the secondary storage into the main memory or items of a page are decompressed during every access to the page.

The variable-length codes for larger domains are not often utilized since efficient algorithms for encoding and decoding have been not known. There are only very few works dealing with fast decoding algorithms for Fibonacci codes [40, 41]. In this thesis, we propose fast encoding and decoding algorithms for Elias-delta, Fibonacci of order 2 and 3, and Elias-Fibonacci codes and we compare the efficiency of these algorithms with conventional algorithms.

Chapter 2 provides some preliminaries and basic definitions related to codes. In Chapter 3, we depict an introduction to variable-length codes and the new Elias-Fibonacci code is introduced. We briefly describe the main ideas of conventional and fast algorithms in Chapter 4 and we introduce the notation used in listings of algorithms. In Chapter 5, we present conventional encoding and decoding algorithms for the selected variable-length codes. In Chapter 6, we introduce a theoretical background to Fibonacci codes necessary for fast Fibonacci encoding and decoding algorithms. We introduce fast encoding algorithms in Chapter 7. Basic principles of fast decoding algorithms are presented in Chapter 8 and the fast decoding algorithms are described in Chapter 9. In Chapter 10, we compare conventional and fast algorithms and we also show some compression properties of these codes. In Chapter 11, we present two applications of fast encoding and decoding algorithms. We conclude this thesis in Chapter 12.
Chapter 2

Preliminaries

When we borrow a definition of a code from Oxford Dictionary [10], we can define a code as a system of words, letters, figures, or symbols used to represent others, especially for the purposes of secrecy. When we utilize a technical definition from Webster’s New World Telecom Dictionary [9], we can define a code as a system by which some combinations of bits is used within a computer and between computers to represent a character or a symbol, such as a letter, number, punctuation mark, or control character.

In this thesis, we aim our effort to binary codes, i.e. we define a code as a set of specific combinations of 0- and 1-bits of a specific total bit-length in order to represent a character, such as a letter, number, punctuation mark, or control character. In the following text, we consider only encoding and decoding of integers; characters, punctuation marks, or control sequences, etc. can be represented by integer values. This combination of 0- and 1-bits of a code is called a codeword. We use the following common notation: the leftmost bit in a codeword is the highest bit and the rightmost bit is the lowest bit. We denote a sequence of \( m \) consecutive 1-bits and 0-bits by \( 1_m \) and \( 0_m \), respectively.

Table 2.1: Standard binary representation of the bit-length 8 for some integers

<table>
<thead>
<tr>
<th>( n )</th>
<th>( B(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000(_2)</td>
</tr>
<tr>
<td>1</td>
<td>00000001(_2)</td>
</tr>
<tr>
<td>2</td>
<td>00000010(_2)</td>
</tr>
<tr>
<td>3</td>
<td>00000011(_2)</td>
</tr>
<tr>
<td>4</td>
<td>00000100(_2)</td>
</tr>
<tr>
<td>5</td>
<td>00000101(_2)</td>
</tr>
<tr>
<td>6</td>
<td>00000110(_2)</td>
</tr>
<tr>
<td>7</td>
<td>00000111(_2)</td>
</tr>
<tr>
<td>8</td>
<td>00001000(_2)</td>
</tr>
<tr>
<td>9</td>
<td>00001001(_2)</td>
</tr>
<tr>
<td>10</td>
<td>00001010(_2)</td>
</tr>
</tbody>
</table>
**Preliminaries**

*Encoding* is simply a process of replacing a coded number with a codeword. *Decoding* is an opposite process of replacing the codeword with the coded number. Both encoding and decoding can be defined by a translation table where each number is assigned by its codeword. In Table 2.1, we see an example of the translation table for the *standard binary representation* of the bit-length 8 for some integers.

The standard binary representation $B(n)$ of an integer $n$ is often used in computers to represent integers of the fixed bit-length of power of 2, i.e., 8, 16, 32 or 64 bits. We denote $L(n) = L(B(n))$ or simply $L$ the bit-length of a number $n$ in the standard binary representation.

A number $n$ with the standard binary representation $B(n) = a_{L-1}a_{L-2} \ldots a_2a_1a_0$ of the bit-length $L$ is obtained by the following sum:

$$n = \sum_{i=0}^{L-1} a_i2^i$$

The maximal bit-length $L_{\text{max}}$ for the standard binary representation of an integer $n$ in a set of integers has to be known in advance since it determines the fixed bit-length used to represent numbers in a computer. In this case, the maximal bit-length $L_{\text{max}}$ is computed as follows: $L_{\text{max}} = 1 + \lfloor \log_2 n_{\text{max}} \rfloor$, where $n_{\text{max}}$ is the maximal number in the set.

A *variable-length code*, as its name implies, consists of codewords with various bit-lengths. An important type of variable-length codes are *prefix codes* (or P-codes). A variable-length code is a prefix code when it satisfies the following prefix property: once a codeword $c$ is assigned to a number, no other codeword should start with the codeword $c$. Although it seems that only prefix codes are *uniquely decodable*, there exists uniquely decodable codes although they are not prefix codes [52]. Therefore, prefix codes are a subset of the *uniquely decodable codes*.

**Example 1** (A uniquely decodable code which is not the prefix code)

Consider a code with the four codewords: 0, 01, 011, 0111. This is not a prefix code since the first codeword 0 is the prefix of the others. However, given an encoded message 01011010111, there is no ambiguity and there is only one way to decode it: 01 011 01 0111, since each 0-bit means a delimiter. We only need to watch out the 0-bit, the beginning of each codeword, and the 1-bit before any 0-bit, the last bit of each codeword.

Uniquely decodable codes, which are not prefix codes, require looking ahead during a decoding process. This makes them not as efficient as prefix codes [52].

*Data compression* is an important application of variable-length codes. In this case, short codewords are assigned to numbers/symbols with a high frequency of occurrence (or frequency for short), whereas long codewords are assigned to numbers with a low frequency [55]. We denote the frequency of occurrence of a number $n$ by $f(n)$ in the following text.
We distinguish static and dynamic codes. The static code permanently assigns a codeword to a number by the translation table that never changes. A well known example of the fixed-length static code is ASCII or Unicode. Another example of the fixed-length static code is the standard binary representation presented above.

A variable-length static code assigns codewords to numbers according the frequencies of numbers in source data. Translation tables are constructed in advance; frequencies of numbers are obtained either by statistics or by analyzing the source data. During decoding, the translation table must be known; therefore, when the table is not generally known, it should be a part of encoded data. An example of the static variable-length code is the static Huffman code [33, 55].

Universal codes are static codes constructed with the assumption that the order of a number implies its frequency, more precisely:

$$\forall i : n_i < n_{i+1} \Rightarrow f(n_i) \geq f(n_{i+1})$$

If this assumption is satisfied, we can concentrate on the order of a number rather than on the frequency. Codewords of universal codes are assigned in the way that the number with the lowest index is assigned with the shortest codeword and the number with the highest index is assigned with the longest codeword and so on. Representatives of universal codes are for example Elias-delta, Elias-gamma [23], Golomb [30, 55], Elias-Fibonacci [4], and Fibonacci codes [26, 12].

**Example 2** (A set of numbers satisfying the assumption of Universal codes)

Let’s have 5 numbers \( n = \{1, 2, 3, 4, 5\} \) with frequencies of occurrence \( f(n) = \{10, 6, 6, 4, 1\} \). This set satisfies the assumption of Universal codes since \( 1 < 2 < 3 < 4 < 5 \) and \( f(1) \geq f(2) \geq f(3) \geq f(4) \geq f(5) \) since \( 10 \geq 6 \geq 6 \geq 4 \geq 1 \). The number 1 is assigned with the shortest codeword whereas the number 5 is assigned with the longest codeword of the Universal code.

Dynamic (or Adaptive) code varies over time, it means its translation tables vary. Codewords of dynamic codes usually depend on the frequency of occurrence, which changes over time when more data is processed. An example of dynamic codes is the adaptive Huffman code [37, 17].

In this thesis, we aim our effort to variable-length codes. In the following chapter, we give an overview of the most often used variable-length codes.
Part II

Variable-Length Codes
Chapter 3

Variable-Length Codes

3.1 Outlines

In this chapter, we give an overview of some variable-length codes. We start with the simplest Unary code in Section 3.2. Then we describe the popular static and adaptive Huffman codes in Section 3.3. Another widely used family of codes, the Golomb codes, are put forward in Section 3.4. In Sections 3.5, we focus on the well-known Elias codes. In Section 3.6, the Fibonacci family codes are described. Our new Elias-Fibonacci code, introduced in [4], is put forward in Section 3.7. In the last section, Section 3.8, these codes are compared with respect to compression ratio, especially for compression of large integers often used in data structures.

3.2 Unary Code

The Unary code is the simplest variable-length code. In this code, the number of 1-bits or 0-bits in a codeword is equal to the number it represents. A sequence of 0- or 1-bits is not a prefix code because when two different numbers are represented by a sequence of 0- or 1-bits, a lower number is a prefix of a higher number, e.g. 00 is a prefix of 000. To construct the valid prefix code from the sequence of 0- or 1-bits, we must use an additional sequence of bits as a delimiter. To create the 0-bit Unary code from the sequence of 0-bits, we replace the lowest 0-bit with the 1-bit delimiter (see Table 3.1 for some examples). On the other hand, the 0-bit delimiter serves as a delimitler of the 1-bit Unary code. The Unary code for a number \( n \) is denoted by the \( U(n) \) symbol and the codewords are denoted by a \( U \) subscript. The Unary code is not used standalone but it is often used as a part of more complex codes, e.g. in Golomb and Elias codes described in Section 3.4 and 3.5, respectively.
### 3.3 Static and Adaptive Huffman Code

The Huffman code [55, 33] is a popular variable-length code. Given a set of numbers and their frequencies, this code constructs a set of variable-length codewords with the shortest average bit-length for the numbers; numbers with higher frequencies obtain shorter codewords and vice versa.

The Huffman translation table is often represented by a binary tree, called the Huffman tree. Coded numbers are considered as leaf nodes of the binary tree. Each leaf node has a weight which is simply the frequency of the number. The parent node weight is the sum of its children weights. The edges are denoted with 0 or 1 bits. To obtain a codeword for a number, we need to follow the path from the root to a leaf with the number. Consequently, the bits of the codeword are read in edges of the path. The tree for the static Huffman code is built by using a bottom-up approach; starting with the leaves of the tree and working to the root. The following steps are involved for the construction of the tree:

1. The weights of leaf nodes are assigned with frequencies of numbers included in the leaf nodes. All leaf nodes are put into a priority queue.
2. Two nodes with the lowest weights are located and removed from the priority queue.
3. A new parent node of these two nodes is created and it is assigned with the weight equals to the sum of the children nodes’ weights.
4. The parent node is added in the priority queue.
5. The edges from the parent node to two child nodes are arbitrarily set to 0 and 1 bits.
6. Steps 2 – 5 are repeated until only one node remains in the priority queue. This node is designated as the root of the tree.
Example 3 (A construction of the static Huffman code)

Let s have five numbers \( n_i \) with frequencies of occurrence \( f(n_1) = 11, f(n_2) = 8, f(n_3) = 7, f(n_4) = 4, f(n_5) = 1 \). The construction of the translation tree for the code is depicted in Figure 3.1. These five numbers are encoded in the following steps A–D:

- **Step A**: numbers \( n_4 \) and \( n_5 \) with the lowest frequencies are removed from the priority queue. The new parent node \( p_1 \) is created and it is assigned by the weight \( f(p_1) = f(n_4) + f(n_5) = 5 \). This node is inserted into the priority queue.

- **Step B**: numbers \( p_1 \) and \( n_3 \) with the lowest frequencies are removed from the priority queue. The new parent node \( p_2 \) with the weight \( f(p_2) = f(p_1) + f(n_3) = 12 \) is inserted into the priority queue.

- **Step C**: numbers \( n_1 \) and \( n_2 \) with the lowest frequencies are removed from the priority queue. The new parent node \( p_3 \) with the weight \( f(p_3) = f(n_1) + f(n_2) = 19 \) is inserted into the priority queue.

- **Step D**: numbers \( p_2 \) and \( p_3 \) with the lowest frequencies are removed from the priority queue. The new root node \( R \) with the weight \( f(R) = f(p_3) + f(p_2) = 31 \) is created.

![Figure 3.1: The construction of the translation tree for the static Huffman code](image)

The translation table for this example is shown in Table 3.2. The static Huffman codewords are denoted by the \( H_s \) subscript.

The adaptive Huffman code \([37, 17]\) continuously updates the tree to gather the latest information on data entered so far (i.e. frequencies of numbers). The adaptive Huffman code utilizes the Huffman tree where the weight and unique id are assigned to each node; the id is set in a postorder traversal (see Figure 3.3). The weights must satisfy the siblings property, it means: When \( A \) is the parent node of \( B \) and \( B \) is the parent node of \( C \), then \( \text{Weight}(A) > \text{Weight}(B) > \text{Weight}(C) \). A set of nodes with same weights makes a block.
Table 3.2: A translation table for the static Huffman code

<table>
<thead>
<tr>
<th>$n$</th>
<th>$f(n)$</th>
<th>$H_s(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_1$</td>
<td>11</td>
<td>00$H_s$</td>
</tr>
<tr>
<td>$n_2$</td>
<td>8</td>
<td>01$H_s$</td>
</tr>
<tr>
<td>$n_3$</td>
<td>7</td>
<td>10$H_s$</td>
</tr>
<tr>
<td>$n_4$</td>
<td>4</td>
<td>110$H_s$</td>
</tr>
<tr>
<td>$n_5$</td>
<td>1</td>
<td>111$H_s$</td>
</tr>
</tbody>
</table>

We need a special codeword to identify a still not encoded number by the Huffman tree; it is called NYT (meaning “not yet transmitted” used in telecommunication theory [57]). Let us note that one node of the tree is always the NYT node and the initialized tree includes one node – the NYT node. Both encoder and decoder start with the root node of the tree with the maximal id in the tree. When we encode the NYT codeword, it has to be followed by a number to be encoded by a generic code.

![Flowchart](image)

Figure 3.2: Updating the adaptive Huffman tree

For every input number $n_i$ the following update procedure is executed (see Figure 3.2):
Variable-Length Codes

1. Search the tree for $n_i$. If $n_i$ is already inserted in the tree, make $n_i$ the actual node and continue with Step 2. If $n_i$ is not in the tree, add two child nodes to the NYT node; a leaf NYT node and a leaf node for the number $n_i$. Then increase the weight of the new $n_i$ node and the parent (the old NYT node). Let us note that all weights are rescaled as soon as the counter reaches the maximal value to avoid overflow. The parent NYT node is set as the actual node. Go to Step 4.

2. If the actual node does not have the highest id in its block (let us remind that a block is a set of nodes with the same weight), swap it with the node having the highest id together with its subtree, except if that node is its parent. We must note that the swapping of nodes means the swapping of numbers in the nodes and subtrees not the swapping of ids.

3. Increment the weight of the actual node.

4. If the actual node is not the root node go to the parent node and continue with Step 2. If the actual node is the root node, finish this procedure.

Example 4 (A construction of the adaptive Huffman code)

In this example, we assume that we encode the following sequence of numbers: $n_1$, $n_2$, $n_3$, $n_4$, $n_1$, $n_4$. These numbers are encoded in the following steps A–E (see the dynamic updating of the Huffman tree in Figure 3.3):

- **Step A**: The Huffman tree is initialized with the NYT node, the weight of this node is set to 0.

- **Step B**: The number $n_1$ is not in the tree; therefore, we create the new NYT and $n_1$ nodes. These new nodes are highlighted with gray color in Figure 3.3. We increment the weights of $n_1$ and the parent NYT node.

- **Step C**: The number $n_2$ is not in the tree; therefore, we create the new NYT and $n_2$ nodes as children of the NYT node. These new nodes are highlighted with gray color in the figure. We increment the weights of $n_2$, the parent NYT node, and the root node.

- **Step D.1**: The number $n_3$ is not in the tree; therefore, we create the new NYT and $n_3$ nodes as children of the NYT node. These new nodes are highlighted with gray color in Figure 3.3. We increment the weights of $n_3$ and the parent NYT node and set the parent of the NYT node as the actual node. This node is highlighted with the thicker line in the figure. This node has the same weight as other nodes in the same block (i.e. weight=1); therefore, we need to swap this node with a node with the highest id in its block. We also increase the weight of this node. The situation after the swapping of the nodes is depicted in Step D.2 of the figure.

- **Step E**: The encoding of $n_3,n_1,n_4$ is not depicted in Figure 3.3. The situation after processing these numbers is depicted in Step E.1 of the figure. Now the last number $n_4$ is encoded. This number is already in the tree; therefore, we set
the actual node to $n_4$, it is depicted with the thicker line in Step E.1. Since id of this node is not the highest one in its block, we swap this node with the $n_2$ node with the highest id in its block with weight=1. We increase the weight of the actual node and the parent node is set as the actual node. This situation is depicted in Step E.2 of Figure 3.3. The actual node is highlighted with the thicker line. We need to swap this node again with the node with the highest id (id=49) in its block with weight=2. After the swapping of the nodes, we increment the weights of the node with id=49 and the root node. The final situation is depicted in Step E.3.

Figure 3.3: A construction of the adaptive Huffman tree

Both static and adaptive Huffman codes require a computation of the Huffman tree. Evidently, the tree is suitable only for a small set of input numbers; it is not very good choice for a large set of input numbers because the Huffman tree becomes large and the computation is rather time consuming. Therefore, the Huffman code is not very useful in the case of compression of large number domains, e.g. the 32-bit integer domain, where all numbers are encoded, since the size of the Huffman tree is equal to the number of encoded values.
3.4 Golomb Codes

The Golomb code has been introduced by Solomon W. Golomb in the 1960s [30]. The Golomb code for a number \( n \) is parametrized with a parameter \( m \). It makes it especially useful where we are able to estimate a proper value of the parameter.

The Golomb code \( G(n) \) for a number \( n \) is defined as follows:

1. Compute three values \( q \) (quotient), \( r \) (remainder), and \( c \):
   \[
   q = \left\lfloor \frac{n}{m} \right\rfloor, \quad r = n - qm, \quad c = \lceil \log_2 m \rceil
   \]
2. Let \( U(q) \) is the 1-bit Unary encoding of \( q \).
3. Let \( B_c(r) \) is the standard binary representation of \( r \) with the bit-length \( c \).
4. If \( r < 2^c - m \) the Golomb codeword is the concatenation:
   \[
   G(n) = U(q)B_{c-1}(r),
   \]
   otherwise the Golomb codeword is the concatenation:
   \[
   G(n) = U(q)B_c(r + c).
   \]

Some examples of Golomb codewords for different values of the parameter \( m \) are shown in Table 3.3. We must note that the Golomb code \( m = 1 \) is equivalent with the Unary code.

The Golomb code is an optimal prefix code if coded values follow a geometric distribution. It makes the Golomb code highly suitable in situations where the occurrence of small values in an input data set is significantly more probable than the occurrence of large values [55]. Golomb codes are especially utilized when a compressed data set includes long runs of 0-bits and a small number of 1-bits. In this case, only 0-bits are encoded by the Golomb code, 1-bits are denoted as delimiters. It can be shown that the best value of the parameter \( m \) is the nearest integer to the following number [55]:

\[
    m = \left\lfloor \frac{-1}{\log_2 (P)} \right\rfloor,
\]

where \( P \) is the probability of occurrence of the 0-bit.

A subset of Golomb codes for the parameter \( m \) equals to powers of 2 (i.e. \( m = 2, 4, 8, 16 \) etc.) has been introduced by Robert F. Rice [53]. These codes are named Golomb-Rice codes.
whereas Elias-gamma can only encode numbers > 0. Consequently, it can encode the zero value, whereas Elias-gamma can only encode numbers > 0.

### 3.5 Elias Codes

_Elias codes_ were introduced by Peter Elias [23], he described three prefix codes. All codes are universal and intended for encoding of positive integers. He denoted the codes by Greek letters gamma, delta, and omega. He did not start with alpha and beta because he denoted the Unary code by alpha and the standard binary representation without insignificant zeros by beta.

#### 3.5.1 Elias-gamma Code

In the _Elias-gamma code_, a positive integer \( n \) is encoded as follows:

1. Let \( B(n) \) be the standard binary representation of \( n \) without insignificant 0-bits.
2. Let \( L \) be the bit-length of \( B(n) \).
3. The Elias-gamma codeword is then the concatenation:

\[
E_\gamma(n) = 0_{L-1}B(n)
\]

In other words, each codeword consists of a prefix and a binary part. The prefix includes the bit-length of the binary part encoded by the Unary code. Some examples of Elias-gamma codewords are shown in Table 3.5. The codewords of the Elias-gamma code are denoted with an \( E_\gamma \) subscript in this article.

The _Exponential Golomb_ (or _Exp-Golomb_) code is identical to the Elias-gamma code of the same number plus one [63]. Consequently, it can encode the zero value, whereas Elias-gamma can only encode numbers > 0.
Table 3.5: Examples of Elias-gamma codewords for some integers

<table>
<thead>
<tr>
<th>( n )</th>
<th>( B(n) )</th>
<th>( L )</th>
<th>( E_\gamma(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bit-length of ( B(n) )</td>
<td>Elias-gamma codeword</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1_{E_\gamma}</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
<td>0 10_{E_\gamma}</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>2</td>
<td>0 11_{E_\gamma}</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>3</td>
<td>00 100_{E_\gamma}</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>3</td>
<td>00 101_{E_\gamma}</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>3</td>
<td>00 110_{E_\gamma}</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>3</td>
<td>00 111_{E_\gamma}</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>4</td>
<td>000 1000_{E_\gamma}</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>4</td>
<td>000 1001_{E_\gamma}</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>4</td>
<td>000 1010_{E_\gamma}</td>
</tr>
<tr>
<td>100</td>
<td>1100100</td>
<td>7</td>
<td>0000000 1100100_{E_\gamma}</td>
</tr>
<tr>
<td>1000</td>
<td>1111101000</td>
<td>10</td>
<td>00000000001111101000_{E_\gamma}</td>
</tr>
<tr>
<td>( 10^4 )</td>
<td>100111000100000</td>
<td>14</td>
<td>0000000000000000 1001111000100000_{E_\gamma}</td>
</tr>
<tr>
<td>( 10^5 )</td>
<td>110000110101000000</td>
<td>17</td>
<td>000000000000000000000000 110000110101000000_{E_\gamma}</td>
</tr>
</tbody>
</table>

3.5.2 Elias-delta Code

In the **Elias-delta code**, a positive integer \( n \) is encoded as follows:

1. Let \( B(n) \) be the standard binary representation of \( n \) without insignificant 0-bits. Let \( B'(n) \) be \( B(n) \) without the highest 1-bit.
2. Let \( L \) be the bit-length of \( B(n) \) and \( B(L) \) be the standard binary representation of \( L \).
3. Let \( Z \) be the bit-length of \( B(L) \).
4. The Elias-delta codeword is then the concatenation:

\[
E_\delta(n) = 0_{Z-1} B(L) B'(n)
\]

This code also consists of a prefix and a binary part; however, both parts are more complicated than in the case of the Elias-gamma code. Some examples of codewords are shown in Table 3.6. The codewords of the Elias-delta code are denoted with an \( E_\delta \) subscript in this work.

3.5.3 Elias-omega Code

In the **Elias-omega code**, a positive integer \( n \) is recursively encoded as follows:

1. Initialize \( E_\omega(n) = 0 \).
2. If \( n = 1 \), END.
3. Let \( B(n) \) be the standard binary representation of \( n \) without insignificant 0-bits. Concatenate \( E_\omega(n) = B(n) E_\omega(n) \).
Table 3.6: Examples of Elias-delta codewords for some integers

<table>
<thead>
<tr>
<th>n</th>
<th>(B(n))</th>
<th>(0_{Z-1})</th>
<th>Bit-length of (L(n)) as the number of zeros -1</th>
<th>Bit-length of (B(n)) as the standard binary representation</th>
<th>(B(n)) without the highest bit</th>
<th>Elias-delta codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>(1_{E\delta})</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>(010_{E\delta})</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>(0101_{E\delta})</td>
</tr>
<tr>
<td>4</td>
<td>110</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>0</td>
<td>(01100_{E\delta})</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td>01</td>
<td>0</td>
<td>(01101_{E\delta})</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>0</td>
<td>11</td>
<td>10</td>
<td>0</td>
<td>(01110_{E\delta})</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>0</td>
<td>(01111_{E\delta})</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>0</td>
<td>100</td>
<td>00</td>
<td>0</td>
<td>(00100000_{E\delta})</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>0</td>
<td>100</td>
<td>01</td>
<td>0</td>
<td>(00100001_{E\delta})</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>0</td>
<td>100</td>
<td>010</td>
<td>0</td>
<td>(00100100_{E\delta})</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>1100100</td>
<td>0</td>
<td>111</td>
<td>100100</td>
<td>0</td>
<td>(00111100100_{E\delta})</td>
</tr>
<tr>
<td>1000</td>
<td>1111010000</td>
<td>0</td>
<td>1010</td>
<td>111010000</td>
<td>000</td>
<td>(001010111101000_{E\delta})</td>
</tr>
<tr>
<td>10^4</td>
<td>10011100010000</td>
<td>0</td>
<td>1110011100010000</td>
<td>0000011100010000</td>
<td>000001111001000010000_{E\delta}</td>
<td></td>
</tr>
</tbody>
</table>

4. Set \(n\) to the bit-length of \(B(n)-1\). Continue with Step 2.

Since the Elias-omega code recursively encodes the prefix, it is sometimes known as the recursive Elias code. Some examples of codewords are shown in Table 3.7. The codewords of the Elias-omega code are denoted with an \(E_{\omega}\) subscript in this work.

### 3.5.4 Comparing Elias Codes

In Table 3.8, we show bit-lengths of some codewords of the three Elias codes. The Elias-gamma code grows slowly; it is therefore a good candidate for encoding a data set where small integers are common and large ones are rare. Elias-delta performs slightly worse than Elias-gamma for small integers but becomes rapidly better for higher integers. The Elias-omega code stays somewhere between these two codes.

### 3.6 Fibonacci Codes

Fibonacci codes are closely related to the Fibonacci representation of an integer and are based on Fibonacci numbers [45]. A Fibonacci number \(F_i^{(m)}\) of order \(m\) is recursively defined as follows:

\[
F_i^{(m)} = F_{i-1}^{(m)} + F_{i-2}^{(m)} + \ldots + F_{i-m}^{(m)}, \text{ for } i \geq 1,
\]
Table 3.7: Examples of Elias-omega codewords for some integers

<table>
<thead>
<tr>
<th>$n$</th>
<th>$E_\omega(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$0_{E_\omega}$</td>
</tr>
<tr>
<td>2</td>
<td>$10_{E_\omega}$</td>
</tr>
<tr>
<td>3</td>
<td>$11_{E_\omega}$</td>
</tr>
<tr>
<td>4</td>
<td>$10100_{E_\omega}$</td>
</tr>
<tr>
<td>5</td>
<td>$10101_{E_\omega}$</td>
</tr>
<tr>
<td>6</td>
<td>$10110_{E_\omega}$</td>
</tr>
<tr>
<td>7</td>
<td>$10111_{E_\omega}$</td>
</tr>
<tr>
<td>8</td>
<td>$111000_{E_\omega}$</td>
</tr>
<tr>
<td>9</td>
<td>$111001_{E_\omega}$</td>
</tr>
<tr>
<td>10</td>
<td>$111010_{E_\omega}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>$101101100100_{E_\omega}$</td>
</tr>
<tr>
<td>1000</td>
<td>$111001111101000_{E_\omega}$</td>
</tr>
<tr>
<td>$10^4$</td>
<td>$111101100110001000_{E_\omega}$</td>
</tr>
<tr>
<td>$10^5$</td>
<td>$101001000011000010010000_{E_\omega}$</td>
</tr>
</tbody>
</table>

where $F_{-m+1}^{(m)} = F_{-m+2}^{(m)} = \ldots = F_{-2}^{(m)} = 0$
and $F_{-1}^{(m)} = F_0^{(m)} = 1$

In Table 3.9, we see examples of some Fibonacci numbers of order 2 and 3.

**Definition 1** (The Fibonacci representation of order $m$ for an integer $n$)
A number $F^{(m)}(n) = a_p a_{p-1} \ldots a_0, a_i \in \{0, 1\}, 0 \leq i \leq p$, is the Fibonacci representation of a positive integer $n$ iff:

$$\sum_{i=0}^{p} a_i F_i^{(m)} = n$$

and there is no run of $m$ consecutive 1-bits in $F^{(m)}(n)$.

The last condition enables to select a correct representation, e.g. for $n = 5$, we select the representation 1000 instead of 0110. Since this representation has the property of not containing any sequence of $m$ consecutive 1-bits [12], we can construct Fibonacci codewords; each codeword includes a sequence of $m$ consecutive 1-bits in the lowest bits as a delimiter. The Fibonacci representation of order $m$ for a number $n$ is denoted with $F^{(m)}(n)$ and a particular representation is denoted with an $F_i^{(m)}$ subscript in the following text. In Table 3.10, some examples of the Fibonacci representation of order 2 and 3 are shown.
### Table 3.8: Bit-lengths of some codewords for the three Elias codes

<table>
<thead>
<tr>
<th>$n$</th>
<th>Elias-gamma</th>
<th>Elias-delta</th>
<th>Elias-omega</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>100</td>
<td>13</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>1000</td>
<td>19</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>$10^4$</td>
<td>27</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>$10^5$</td>
<td>33</td>
<td>25</td>
<td>28</td>
</tr>
<tr>
<td>$10^6$</td>
<td>39</td>
<td>28</td>
<td>31</td>
</tr>
</tbody>
</table>

### Table 3.9: Some Fibonacci numbers of order 2 and 3

<table>
<thead>
<tr>
<th>$i$</th>
<th>$F_i^{(2)}$</th>
<th>$F_i^{(3)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>21</td>
<td>44</td>
</tr>
<tr>
<td>7</td>
<td>34</td>
<td>81</td>
</tr>
<tr>
<td>8</td>
<td>55</td>
<td>149</td>
</tr>
</tbody>
</table>

#### 3.6.1 Fibonacci Code of Order 2

The reverse Fibonacci code of order 2\(^1\) has been introduced in [26]. Since fast decoding algorithms [4, 42] have to read bits of codewords from the lowest to the highest bit, we must utilize the reverse Fibonacci representation. In the reverse Fibonacci representation, we reverse the bits so that the highest 1-bit becomes the lowest 1-bit and so on; when a bit of the reverse Fibonacci representation is read,\(^\text{1This code is commonly known as the Fibonacci code; however, we use the prefix reverse since it corresponds to the fact that the Fibonacci representation of this code is reversed. Consequently, we need to distinguish this code and the generalized Fibonacci codes of order ≥ 2 introduced by Apostolico and Fraenkel in [12] not using the reverse Fibonacci representation (see Section 3.6.2).}
Table 3.10: Fibonacci representations of order 2 and 3 for some numbers

<table>
<thead>
<tr>
<th>n</th>
<th>$F^{(2)}(n)$</th>
<th>$F^{(3)}(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1$_{F^{(2)}}$</td>
<td>1$_{F^{(3)}}$</td>
</tr>
<tr>
<td>2</td>
<td>1 0$_{F^{(2)}}$</td>
<td>1 0$_{F^{(3)}}$</td>
</tr>
<tr>
<td>3</td>
<td>1 0 0$_{F^{(2)}}$</td>
<td>1 0 1$_{F^{(3)}}$</td>
</tr>
<tr>
<td>4</td>
<td>1 0 1$_{F^{(2)}}$</td>
<td>1 0 0 0$_{F^{(3)}}$</td>
</tr>
<tr>
<td>5</td>
<td>1 0 0 0$_{F^{(2)}}$</td>
<td>1 0 0 1$_{F^{(3)}}$</td>
</tr>
<tr>
<td>6</td>
<td>1 0 0 1$_{F^{(2)}}$</td>
<td>1 0 1 0$_{F^{(3)}}$</td>
</tr>
<tr>
<td>7</td>
<td>1 0 1 0$_{F^{(2)}}$</td>
<td>1 0 1 1$_{F^{(3)}}$</td>
</tr>
</tbody>
</table>

we always know the order $i$ of the $F_i$ number in the representation. More details are depicted in Section 8.6.2 describing general principles of fast decoding algorithms. The reverse Fibonacci code of order 2, denoted by $F^{(2)}(n)$, is obtained by appending an additional 1-bit as the lowest bit to the reverse Fibonacci representation, i.e.:

$$F^{(2)}(n) = \text{reverse}(F(n)_{(2)})1$$

It can be done because two consecutive 1-bits do not appear in the Fibonacci representation $F^{(2)}(n)$. Some codewords of the reverse Fibonacci code of order 2 are shown in Table 3.11. Codewords of the reverse Fibonacci code of order 2 are denoted by an $F^{(2)}$ subscript in the following text.

Table 3.11: Some codewords of the reverse Fibonacci code of order 2

<table>
<thead>
<tr>
<th>n</th>
<th>$F^{(2)}(n)$</th>
<th>reverse $F^{(2)}(n)$</th>
<th>$\overline{F}^{(2)}(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1$_{F^{(2)}}$</td>
<td>1$_{F^{(3)}}$</td>
<td>11$_{F^{(2)}}$</td>
</tr>
<tr>
<td>2</td>
<td>10$_{F^{(2)}}$</td>
<td>01$_{F^{(3)}}$</td>
<td>011$_{F^{(2)}}$</td>
</tr>
<tr>
<td>3</td>
<td>100$_{F^{(2)}}$</td>
<td>001$_{F^{(3)}}$</td>
<td>0011$_{F^{(2)}}$</td>
</tr>
<tr>
<td>4</td>
<td>101$_{F^{(2)}}$</td>
<td>101$_{F^{(3)}}$</td>
<td>1011$_{F^{(2)}}$</td>
</tr>
<tr>
<td>5</td>
<td>1000$_{F^{(2)}}$</td>
<td>0001$_{F^{(3)}}$</td>
<td>00011$_{F^{(2)}}$</td>
</tr>
<tr>
<td>6</td>
<td>1001$_{F^{(2)}}$</td>
<td>1001$_{F^{(3)}}$</td>
<td>10011$_{F^{(2)}}$</td>
</tr>
<tr>
<td>7</td>
<td>1010$_{F^{(2)}}$</td>
<td>0101$_{F^{(3)}}$</td>
<td>01011$_{F^{(2)}}$</td>
</tr>
<tr>
<td>8</td>
<td>10000$_{F^{(2)}}$</td>
<td>00001$_{F^{(3)}}$</td>
<td>000011$_{F^{(2)}}$</td>
</tr>
<tr>
<td>16</td>
<td>100100$_{F^{(2)}}$</td>
<td>001001$_{F^{(3)}}$</td>
<td>0010011$_{F^{(2)}}$</td>
</tr>
<tr>
<td>32</td>
<td>1010100$_{F^{(2)}}$</td>
<td>0010101$_{F^{(3)}}$</td>
<td>00101011$_{F^{(2)}}$</td>
</tr>
</tbody>
</table>
### 3.6.2 Fibonacci Codes of Higher Orders

In the case of Fibonacci codes of order 3 and higher, it is not possible to create code-words by only appending \(1_{m-1}\) bits to the reverse Fibonacci representation \(F^{(m)}(n)\) as in the case of the reverse Fibonacci code of order 2. For example, let us suppose two Fibonacci representations \(F^{(3)}(5) = 101_F\) and \(F^{(3)}(12) = 1101_F\). If we append the 11 sequence to the reverse Fibonacci representations, we obtain codewords 10111 and 101111, respectively. Since the first codeword includes a sequence of three 1-bits in the lowest bits and the second codeword includes a sequence of four 1-bits in the lowest bits, these two codewords are not codewords of a prefix code. The 1-bit read after the lowest three 1-bits of the second codeword is taken as a part of the next codeword and it leads to an error.

In [12], authors introduced the generalized Fibonacci code of order \(m\) (\(m \geq 2\)). The authors also depicted a proof of the completeness of the code. Each codeword of this code includes a sequence \(1_m\) in the lowest bits as the delimiter. The Fibonacci sum \(S_g^{(m)}\) must be used to construct a valid prefix code (see Table 3.12 for some examples).

**Definition 2** (The Fibonacci sum)

The Fibonacci sum \(S_g^{(m)}\) is defined as follows:

\[
S_g^{(m)} = \begin{cases} 
0, & \text{for } g < -1 \\
\sum_{i=-1}^{g} F_i^{(m)}, & \text{for } g \geq -1
\end{cases}
\]

| \(g\) | \(S_{-2}^{(2)}\) | \(S_{-2}^{(3)}\) | \(S_{-1}^{(2)}\) | \(S_{-1}^{(3)}\) | \(S_0^{(2)}\) | \(S_0^{(3)}\) | \(S_1^{(2)}\) | \(S_1^{(3)}\) | \(S_2^{(2)}\) | \(S_2^{(3)}\) | \(S_3^{(2)}\) | \(S_3^{(3)}\) | \(S_4^{(2)}\) | \(S_4^{(3)}\) | \(S_5^{(2)}\) | \(S_5^{(3)}\) | \(S_6^{(2)}\) | \(S_6^{(3)}\) | \(S_7^{(2)}\) | \(S_7^{(3)}\) | \(S_8^{(2)}\) | \(S_8^{(3)}\) |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| -2  | 0   | 0   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| -1  | 1   | 1   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 0   | 2   | 2   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 1   | 4   | 4   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 2   | 7   | 8   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 3   | 12  | 15  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 4   | 20  | 28  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 5   | 33  | 52  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 6   | 54  | 96  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 7   | 88  | 177 |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 8   | 143 | 326 |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

Consequently, the generalized Fibonacci code \(F^{(m)}_C(n)\) originally defined by Apostolico and Fraenkel in [12] for a positive integer \(n\) is as follows:

1. If \(n = 1\) then \(F^{(m)}_C(n) = 1_m\). END.
2. If \(n = 2\) then \(F^{(m)}_C(n) = 01_m\). END.
3. Find $g$ such that $S_g^{(m)} - 2 < n \leq S_g^{(m)}$.

4. Compute $Q = n - S_g^{(m)} - 1$.

5. Compute $F^{(m)}(Q)$.

6. Let $L(F^{(m)}(Q))$ be the bit-length of $F^{(m)}(Q)$.

7. Compute $z = g - 1 - L(F^{(m)}(Q))$.

8. The Fibonacci code of order $m$ is then the concatenation:

$$F^{(m)}_C(n) = F^{(m)}(Q)0_z01_m$$

The bit-length of $F^{(m)}_C(n)$ is $m + g$.

Like in the case of the reverse Fibonacci code of order 2, we need a reverse variant of these codes appropriate for fast decoding algorithms. The reverse variant has been introduced by Shmuel T. Klein at al. in [41] and we use it in our fast encoding and decoding algorithms described later. This modification is denoted by $F^{(m)}(n)$ in the following text. The reverse Fibonacci code $F^{(m)}(n)$ is defined by the same procedure like $F^{(m)}_C(n)$, only the 8-th step is different:

8. The reverse Fibonacci code of order $m$ is then the concatenation:

$$F^{(m)}(n) = reverse(F^{(m)}(Q))0_z01_m$$

Using this algorithm for $m = 2$ we obtain the reverse Fibonacci code of order 2 described in Section 3.6.1. Some codewords for the $F^{(3)}(n)$ code are shown in Table 3.13.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$F^{(3)}(n)$</th>
<th>$\langle S_g^{(3)} \rangle$, $S_{g-1}^{(3)}$</th>
<th>$m + g$</th>
<th>$F^{(3)}(Q)$</th>
<th>$z$</th>
<th>$F^{(3)}(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$1_{F^{(3)}}$</td>
<td>$[0, 1]$</td>
<td>$3 + 0 = 3$</td>
<td>$F(0) = _$</td>
<td>-</td>
<td>$11_{F^{(3)}}$</td>
</tr>
<tr>
<td>2</td>
<td>$10_{F^{(3)}}$</td>
<td>$[1, 2]$</td>
<td>$3 + 1 = 4$</td>
<td>$F(0) = _$</td>
<td>-</td>
<td>$011_{F^{(3)}}$</td>
</tr>
<tr>
<td>3</td>
<td>$11_{F^{(3)}}$</td>
<td>$[2, 4]$</td>
<td>$3 + 2 = 5$</td>
<td>$F(0) = _$</td>
<td>1</td>
<td>$0011_{F^{(3)}}$</td>
</tr>
<tr>
<td>4</td>
<td>$100_{F^{(3)}}$</td>
<td>$[2, 4]$</td>
<td>$3 + 2 = 5$</td>
<td>$F(1) = 1_{F^{(3)}}$</td>
<td>0</td>
<td>$1011_{F^{(3)}}$</td>
</tr>
<tr>
<td>5</td>
<td>$101_{F^{(3)}}$</td>
<td>$[4, 8]$</td>
<td>$3 + 3 = 6$</td>
<td>$F(0) = _$</td>
<td>2</td>
<td>$00011_{F^{(3)}}$</td>
</tr>
<tr>
<td>6</td>
<td>$110_{F^{(3)}}$</td>
<td>$[4, 8]$</td>
<td>$3 + 3 = 6$</td>
<td>$F(1) = 1_{F^{(3)}}$</td>
<td>1</td>
<td>$10011_{F^{(3)}}$</td>
</tr>
<tr>
<td>7</td>
<td>$1000_{F^{(3)}}$</td>
<td>$[4, 8]$</td>
<td>$3 + 3 = 6$</td>
<td>$F(2) = 10_{F^{(3)}}$</td>
<td>0</td>
<td>$01111_{F^{(3)}}$</td>
</tr>
<tr>
<td>8</td>
<td>$1001_{F^{(3)}}$</td>
<td>$[4, 8]$</td>
<td>$3 + 3 = 6$</td>
<td>$F(3) = 11_{F^{(3)}}$</td>
<td>0</td>
<td>$110111_{F^{(3)}}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
3.7 Elias-Fibonacci Code

In this section, we describe the Elias-Fibonacci code: our new code introduced in [4]. Like other codes in this chapter, the Elias-Fibonacci code is a universal variable-length code for positive integers. It consists of two parts. The second part is $B(n)$, the binary representation of a number $n$. The first part is $L(n)$ (or $L$), the bit-length of $B(n)$, encoded with the reverse Fibonacci representation of order 2. We do not utilize a delimiter in this code; however, we utilize a sequence of two 1-bits between the end of the reverse Fibonacci representation $F^{(2)}(L)$ and the start of $B(n)$. In other words, if we reach two 1-bits in a codeword, we read $L$ and we know that we must read $L - 1$ bits to complete $B(n)$.

The name of the new code is inspired by terminology of the Elias-gamma code [23]. The prefix of the Elias-gamma code is encoded by the Unary code. In the case of the new code, the prefix is encoded by the reverse Fibonacci code of order 2. Therefore, we have chosen the name as a combination of the names Elias and Fibonacci.

The encoding algorithm for a positive integer $n$ is defined as follows:

1. Let $B(n)$ be the standard binary representation of $n$ without insignificant 0-bits.
2. Let $B'(n)$ be $B(n)$ without the highest 1-bit.
3. Let $L$ be the bit-length of $B(n)$.
4. Compute $F^{(2)}(L)$.
5. The codeword of the Elias-Fibonacci code is then the concatenation:

$$EF(n) = F^{(2)}(L)B'(n)$$

Some examples of codewords are shown in Table 3.14. The codewords of the Elias-Fibonacci code are denoted with an $EF$ subscript in this article.

In both conventional and fast encoding/decoding algorithms (see Sections 5.5, 7.2, and 9.2), we utilize another view on the Elias-Fibonacci code. The highest 1-bit of $B(n)$ completes the reverse Fibonacci representation of its bit-length. Consequently, the Elias-Fibonacci code for a positive integer $n$ is defined as follows:

1. Let $B(n)$ be the standard binary representation of $n$ without insignificant 0-bits.
2. Let $L$ be the bit-length of $B(n)$.
3. Compute $F^{(2)}(L)$. 
Table 3.14: Some codewords of the Elias-Fibonacci code

<table>
<thead>
<tr>
<th>n</th>
<th>$B(n)$</th>
<th>$L$</th>
<th>$F^{(2)}(L)$</th>
<th>$EF(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$11_{F^{(2)}}$</td>
<td>$11_{EF}$</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
<td>$011_{F^{(2)}}$</td>
<td>$011 00_{EF}$</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>2</td>
<td>$011_{F^{(2)}}$</td>
<td>$011 11_{EF}$</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>3</td>
<td>$0011_{F^{(2)}}$</td>
<td>$0011 00_{EF}$</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>3</td>
<td>$0011_{F^{(2)}}$</td>
<td>$0011 11_{EF}$</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>3</td>
<td>$0011_{F^{(2)}}$</td>
<td>$0011 10_{EF}$</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>3</td>
<td>$0011_{F^{(2)}}$</td>
<td>$0011 11_{EF}$</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>4</td>
<td>$1011_{F^{(2)}}$</td>
<td>$1011 000_{EF}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>110100</td>
<td>7</td>
<td>$01011_{F^{(2)}}$</td>
<td>$01011 100100_{EF}$</td>
</tr>
</tbody>
</table>

4. The Elias-Fibonacci codeword is then the concatenation:

$$EF(n) = reverse(F^{(2)}(L))B(n)$$

This definition enables us to skip one operation (removing the highest 1-bit in Step 2) necessary in the case of the previous definition.

### 3.8 Comparison and Selection of Codes

Encoding/decoding algorithms have been intensively studied in a connection with video and audio encoders/decoders. Moreover, these studies often integrate the variable-length codes in the encoders and decoders. The result is the following recommendations: T.81 for a JPEG encoder [36] and H.261 for an MPEG encoder [35]. These encoders utilize Exponential Golomb [63] or Huffman codes [49]. Many works and patents (e.g. [27, 38, 61]) therefore deal with fast encoders/decoders for these codes. In this case, a codeword is predefined for each coded number; the size of the encoding/decoding table then corresponds to the size of the number domain used. As result, these algorithms can be used only for small domains of coded numbers, e.g. 8 or 16 bits; they are not useful for general universal codes where the domain can be large (e.g. 32-bit-length numbers). Since these domains are often used in the data processing area or text file compression [66, 58], other algorithms utilizing more general variable-length codes have been investigated. The time consumption of decompression is sometimes more critical than the time of compression; therefore, efficient decompression algorithms have been studied in many works related to decompression of data structures [56, 29, 3] or text files [50, 16]. In these articles, Fibonacci, Elias-delta, and Elias-Fibonacci codes overcome other variable-length codes. Therefore, in our research we selected these universal codes and we developed fast encoding and decoding algorithms for these codes.
In Figure 3.4, we see a comparison of codeword bit-lengths for some variable-length codes and the standard binary representation. Evidently, Golomb-Rice codes are better than other codes only for a small range of values depending on the $m$ parameter. Before the range, the bit-length of codewords of the Golomb-Rice code is equal to $m$, whereas other codes produce shorter codewords. In the range, codewords of the Golomb-Rice code are shorter than other codes and, after the range, their bit-length grows up exponentially.

**Example 5** (A comparison of Golomb-Rice and Fibonacci of order 2 codes)

Let us take a comparison of the Golomb-Rice code with $m = 8$ and the Fibonacci code of order 2 in Figure 3.4. We see that codewords of the Fibonacci code of order 2 are shorter for coded numbers of the bit-length $< 5$. Let us consider the number 8 with the bit-length 4 of the standard binary representation; the codeword bit-length is 5 for the Fibonacci code and 8 for the Golomb-Rice code. Furthermore, codewords of the Golomb-Rice code are shorter than the Fibonacci code for numbers $[5, 12]$. Let us consider the number 987 with the bit-length 10 of the standard binary representation; the codeword bit-length is 16 for the Fibonacci code and 10 for the Golomb-Rice code. Finally, codewords of the Fibonacci code of order 2 are shorter than codewords of the Golomb-Rice code for coded numbers of the bit-length $> 12$. Let us consider the number 10,946 with the bit-length 12 of the standard binary representation, the codeword bit-length is 21 for the Fibonacci code and 40 for the Golomb-Rice code.

The Elias-gamma and Fibonacci of order 2 codes are useful for small number domains, i.e. the 8-bit domain. For large domains greater than 8-bits, Fibonacci of order 3, Elias-delta, and Elias-Fibonacci codes are right choice.

The Fibonacci code of order 3 produces shorter codewords than our Elias-Fibonacci code for numbers of the bit-length $< 40$ in the standard binary representation, whereas codewords of the Elias-Fibonacci code become shorter than the Fibonacci code of order 3 for numbers of the bit-length $\geq 40$.

In Figure 3.4, we also see compression properties of the variable-length codes. The standard binary representation has always the same bit-length whereas the bit-length of variable-length codes increase with a growing coded number. This implies that short codewords encoding input data must occur more frequent then long codewords. For example for 32-bit coded numbers, standard binary representations have the bit-length 32. We can compress input numbers only when they are assigned with shorter codewords than 32-bits on average. In the Fibonacci of order 3 code, all coded numbers $n \leq 2^{26}$ have codewords of the bit-length $< 32$.

Since binary parts of Elias-delta and Elias-Fibonacci codewords have the same bit-length, we must compare the bit-length of their prefixes if we want to compare the bit-length of their codewords. As we see in Table 3.15, the prefix of the Elias-Fibonacci codeword is longer only for coded number 1 ($L(n) = 1$).
Table 3.15: A comparison of prefixes for Elias-Fibonacci and Elias-delta codes

<table>
<thead>
<tr>
<th>Bit-length of $L = L(n)$</th>
<th>Standard binary representation</th>
<th>Elias-Fibonacci prefix $L(F^{(2)}(L))$</th>
<th>Elias-delta prefix $L(0_{L-1}B(L))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>7</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>10</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>11</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>13</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>14</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>1,024</td>
<td>16</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>2,048</td>
<td>17</td>
<td>23</td>
<td></td>
</tr>
</tbody>
</table>
Figure 3.4: A comparison of codeword bit-lengths for some variable-length codes and the standard binary representation (a) for coded numbers up-to 64 bits of the standard binary representation (b) for coded numbers of the bit-length 35–50 (b) for coded numbers of the bit-length 1–16.
Chapter 4
Preliminaries of Encoding and Decoding Algorithms

Conventional encoding and decoding algorithms, described in Chapter 5, process data bit-by-bit which is rather time consuming. The main idea of fast encoding and decoding algorithms, is to encode and decode data in a bigger chunk of bits called **segment**. In Figure 4.1, we see a general schema of conventional and fast encoding and decoding algorithms. The fast algorithms described in 7 then encode or decode codewords segment-by-segment instead of bit-by-bit. We see the segment is used to handle codewords – it is utilized in the output of encoding and in the input of decoding. The segments usually have the bit-length of 8, 16, 32 or more bits to fit in a processor register.

Our fast decoding algorithms are based on a finite automaton [65]. The state of the automaton depends on a value of the current segment read from an input stream and the previous automaton state. For each automaton state, a precomputed mapping table defines the output decoded numbers and the new automaton state.

For both conventional and fast encoding algorithms, we assume a two-step process encoding a set of integers into an output stream. In the first step, we encode a number in a temporary **buffer**. In the second step, this **buffer** is written in the output stream. A general algorithm encoding input numbers into an output stream is depicted in Algorithm 1. The first step is represented by the **Encode** function in Line 4, it is any conventional or fast encoding algorithm described in the following chapters. The rest of the algorithm represents the second step of a general encoding algorithm. It utilizes the **Remain** variable to store incomplete bytes with the **RemLen** bit-length. The function **SetNextByte(stream, byte)** writes a **byte** in the output **stream**. The **AND**, **OR**, **<<**, and **>>** are binary and, or, left shift, and right shift operators, respectively.

The length and the type of the temporary **buffer** depends on the bit-length of a codeword, target processor platform, and implementation environment. The best choice for storage of **buffer** is a processor register. If the bit-length of a codeword is greater than the bit-length of the register, we can use a memory array. In our implementation, we utilize a memory array for all codes.
Preliminaries of Encoding and Decoding Algorithms

**Figure 4.1:** A general schema of conventional and fast encoding and decoding algorithms for the 8bit segment

We utilize some functions in algorithms described in the following chapters. The conventional bit-oriented algorithms use the function \( \text{SetBit(buffer, i, bit)} \) setting the \( i \)-th bit of \( \text{buffer} \) to \( \text{bit} \), the function \( \text{GetBit(n, i)} \) returning the \( i \)-th bit of the standard binary representation of \( n \), and \( \text{GetNextBit(stream)} \) returning the next bit of the input stream.

In all fast algorithms, we utilize the function \( \text{SetBits(buffer, i, n, len)} \) which copies the \( len \) bits of the standard binary representation of \( n \) into the \( \text{buffer} \) from the \( i \)-th bit position and the function \( \text{GetBits(stream, len)} \) returning the \( len \) bits from the input \( \text{stream} \). The function \( \text{Eos(stream)} \) is utilized to test the end of the stream.

Moreover, in the Elias-delta and Elias-Fibonacci encoding algorithms, we utilize the function \( L(n) \) returning the bit-length of the standard binary representation of \( n \), i.e. the position of the highest 1-bit. The \( L(n) \) function can be implemented by an iterative process with while loop or by means of a lookup table called \( \text{LogTable256} \) [59]. In the case of Intel and AMD processors\(^1\), we can use the BSR assembly instruction which scans the register from the highest bit for the first occurrence of the 1-bit [34]. The processing time of this instruction depends on the bit-length of a number in the register; the count of CPU cycles increases when the

---

\(^1\)Intel\® 64 and IA-32 Architectures Software Developer’s Manual (available at www.intel.com), AMD64 Architecture Programmer’s Manual (available at www.amd.com)
number in the register decreases [25]. Consequently, we should use the implementation with the BSR instruction only when it is more efficient than the implementation using LogTable256. We compare all implementations later in Section 10.
Chapter 5

Conventional Encoding and Decoding Algorithms

5.1 Introduction

In this chapter, we describe conventional encoding and decoding algorithms for the following variable-length codes: Elias-delta, Fibonacci of order 2 and 3, and Elias-Fibonacci. These algorithms use bit-by-bit processing therefore they are not very efficient. The following algorithms encodes and decodes only one value. When more input numbers are considered, an encoding algorithm is used as a part of Algorithm 1. In the case of decoding more codewords in a bit stream, a decoding algorithm is repeated in a loop for each codeword.

5.2 Conventional Algorithms for Elias-delta Code

In Algorithm 2, a conventional algorithm encoding a number \( n \) with the Elias-delta code is shown [23]. This algorithm includes three blocks related to three parts of the Elias-delta code (see Section 3.5.2). In Line 1, the bit-length of an input number is computed. The number of zero bits \( Z + 1 \) in the first part of a codeword is computed in Line 2. In Lines 3–6, the sequence \( 0^{Z-1} \) is written in buffer. The standard binary representation of the bit-length of \( n \) is written in Lines 7–11. Finally, the standard binary representation of \( n \) without the highest 1-bit is written in buffer in Lines 12–15. The bit-length of the codeword is computed in Line 16.

Example 6 (Encoding a number with the Elias-delta code)

Let us consider \( n = 35 \). In Line 1, the bit-length \( L = 6 \) is computed. Since the bit-length of \( L \) is 3 we set \( Z = 3 \) (Line 2). In Lines 3–6, we write the \( Z - 1 = 3 - 1 = 2 \) zero bits starting from the position \( \text{pos} = Z + L - 1 = 3 + 6 - 1 = 8 \) (buffer = 00XXXXXXX). In Lines 7–11, we write the bit-length \( L = 110_2 \) as the standard binary representation starting from the position \( L - 1 = 5 \) (buffer = 00110XXXX). Finally, \( n = 35 = 100011_2 \) is written in buffer without
Conventional Encoding and Decoding Algorithms

the highest 1-bit (buffer = 0011000011). The bit-length of the Elias-delta codeword is \(\text{len} = L + 2 \times Z - 1 = 6 + 2 \times 3 - 2 = 10\)

```
input : n, a positive integer
output: buffer, an Elias-delta codeword of n with the bit-length len

1  \(L \leftarrow L(n)\);
2  \(Z \leftarrow L(L)\);
3  pos \leftarrow Z + L - 1;
4  for i = 0 to Z - 2 do
5      \(\text{SetBit(buffer, pos + i, 0)}\);
6  end
7  pos \leftarrow L - 1;
8  for i = 0 to Z - 1 do
9      \(\text{bit} \leftarrow \text{GetBit}(L, i)\);
10     \(\text{SetBit(buffer, pos + i, bit)}\);
11  end
12  for i = 0 to L - 2 do
13     \(\text{bit} \leftarrow \text{GetBit}(n, i)\);
14     \(\text{SetBit(buffer, i, bit)}\);
15  end
16  len \leftarrow L + 2 \times Z - 2;
```

Algorithm 2: A conventional encoding algorithm for the Elias-delta code

In Algorithm 3, we see a conventional Elias-delta decoding algorithm. This algorithm also includes three blocks related to three parts of the Elias-delta code (see Section 3.5.2). The sequence 0\(_Z-1\) is read from an input stream in the \(Z\) variable in Lines 3–6; we read the sequence until the first 1-bit is read, for each 0-bit, we increase the \(Z\) variable by 1. The highest 1-bit of \(L\) is set in Line 7. We read the remaining bits of \(L\) from the input stream in Lines 8–12. Since the standard binary representation of \(n\) in a codeword is encoded without the highest 1-bit, we set the highest 1-bit in the decoded number in Line 14. Remaining bits of \(n\) are read from the input stream in Lines 15–19.

5.3 Conventional Algorithms for Fibonacci Code of Order 2

In Algorithm 4, we see a conventional bit-oriented encoding algorithm for the Fibonacci code of order 2. Since we utilize the reverse Fibonacci code (see Section 3.6.1), bits of a Fibonacci codeword are written in the reverse order in \(buffer\). In the first two lines of the algorithm, we search for the highest 1-bit of the Fibonacci representation of \(n\). This iterative process starts with the lowest Fibonacci number and continues to higher Fibonacci numbers. Although we can start from higher Fibonacci numbers or use a binary search algorithm, we prefer the iteration from the lowest Fibonacci number, since we expect to encode small numbers with higher frequencies in compression algorithms.
Conventional Encoding and Decoding Algorithms

Algorithm 3: A conventional decoding algorithm for the Elias-delta code

```plaintext
input : stream including an Elias-delta codeword
output: n, a decoded number
1  Z ← 0; L ← 0; n ← 0;
2  bit ← GetNextBit(stream);
3  while bit = 0 do
4      Z ← Z + 1;
5      bit ← GetNextBit(stream);
6  end
7  SetBit(L,Z,1);
8  while Z > 0 do
9      Z ← Z − 1;
10     bit ← GetNextBit(stream);
11    SetBit(L,Z,bit);
12  end
13  L ← L − 1;
14  SetBit(n,L,1);
15  while L > 0 do
16     L ← L − 1;
17     bit ← GetNextBit(stream);
18    SetBit(n,L,bit);
19  end
```

Algorithm 4: A conventional encoding algorithm for the Fibonacci code of order 2

```plaintext
input : n, a positive integer
output: buffer, a Fibonacci code of order 2 codeword of n with the bit-length len
1  i ← 0;
2  while F_i^{(2)} ≤ n do i ← i + 1;
3  len ← i + 1;
4  i ← i − 1;
5  while i ≥ 0 do
6      if F_i^{(2)} ≤ n then
7         SetBit(buffer,len − i − 1,1);
8         n ← n − F_i^{(2)};
9      else
10         SetBit(buffer,len − i − 1,0);
11    end
12  end
13  SetBit(buffer,0,1);
```

Example 7 (Encoding a number by the Fibonacci code of order 2)
Let us consider n = 53. Due to the fact that F_7^{(2)} = 34 ≤ 53 < F_8^{(2)} = 55, we compute len = 9 and i = 7 in Lines 1–4. Since F_7^{(2)} = 34 ≤ 53, the 1-bit is set in the 1-st position of buffer (we use the reverse Fibonacci code). The position is computed by len − i − 1 = 9 − 7 − 1 = 1. The 0-th position is not known yet because it is reserved for the 1-bit delimiter: buffer = 1X. Then we compute n = 53 − 34 = 19 and i = i − 1 = 6. Since F_5^{(2)} = 21 > 19, we set the 0-bit in the 2-nd position of buffer: buffer = 01X and i = 5. Since F_5^{(2)} = 13 ≤ 19, we set the 1-bit in the 3-rd
Conventional Encoding and Decoding Algorithms

position: buffer = 101X, n = 19 − 13 = 6 and i = 4. Since $F^{(2)}_3 = 8 > 6$, we set the 0-bit in the 4-th position: buffer = 0101X and i = 3. Since $F^{(2)}_3 = 5 \leq 6$, we set the 1-bit in the 5-th position: buffer = 10101X, n = 6 − 5 = 1 and i = 2. Since $F^{(2)}_3 = 3 > 1$, we set the 0-bit in the 6-th position: buffer = 010101X and i = 1. Since $F^{(2)}_1 = 2 > 1$, we set the 0-bit in the 7-th position: buffer = 0010101X and i = 0. Since $F^{(2)}_0 = 1 \leq 1$, we set the 1-bit in the 8-th position: buffer = 10010101X, n = 0, and i = −1. Since i < 0, we continue with Line 14 where the delimiter is set; the 1-bit is set in the lowest bit: buffer = 100101011.

| input : stream including a codeword of the Fibonacci code of order 2 |
| output: n, a decoded number |
| 1 n ← 0; |
| 2 prevBit ← 0; |
| 3 i ← 0; |
| 4 while true do |
| 5 | bit ← GetNextBit(stream); |
| 6 | if prevBit = 1 and bit = 1 then break; |
| 7 | if bit = 1 then n ← n + $F^{(2)}_i$; |
| 8 | i ← i + 1; |
| 9 | prevBit ← bit; |
| 10 end |

Algorithm 5: A conventional decoding algorithm for the Fibonacci code of order 2

The decoding algorithm works in the opposite way (see Algorithm 5). In the prevBit variable we remember the previous read bit. We set prevBit, n, and i to 0 in Lines 1–3. In loop in Lines 4–10, we read bits in stream and compute the output number n. When both bit (the current bit) and prevBit (the previous bit) are 1-bits in Line 6, the current bit is the delimiter and the loop ends. If the current bit is the 1-bit in Line 7, we add the $F^{(2)}_i$ number to n. In Lines 7–8, we set $i = i + 1$ and prevBit to remember the previous read bit.

5.4 Conventional Algorithms for Fibonacci Code of Order 3

A conventional encoding algorithm for the Fibonacci code of order 3 is shown in Algorithm 6. The numbers 1 and 2 are directly encoded in Lines 1–12. In Lines 13–14, the correct interval of Fibonacci sums is found. The lower Fibonacci sum is utilized in Line 15 for the calculation of the $Q$ value. The second part of the code, $0_z$, is written in Line 19. The first part, the reverse Fibonacci representation of $Q$, is written in Lines 20–29. Finally, the $0(1_m)$ delimiter is written in Lines 30–34. Conventional encoding algorithms for Fibonacci codes of order $\geq 3$ are similar to this algorithm.

Example 8 (Encoding a number by the Fibonacci code of order 3)
Let us consider $n = 35$. Since $n = 35$ is in $(S_4,S_5) = (28,52)$, we compute $g = 6$
Conventional Encoding and Decoding Algorithms

input : \( n \), a positive integer

output: buffer, a Fibonacci code of order 3 codeword of \( n \) with the bit-length \( len \)

1. if \( n = 1 \) then
   2. \( len \leftarrow 3; \)
   3. SetBit(buffer,0,1);
   4. SetBit(buffer,1,1);
   5. SetBit(buffer,2,1);
2. else if \( n = 2 \) then
   3. \( len \leftarrow 4; \)
   4. SetBit(buffer,0,1);
   5. SetBit(buffer,1,1);
   6. SetBit(buffer,2,1);
   7. SetBit(buffer,3,0);
3. else
   4. \( g \leftarrow 2; \)
   5. while \( n \leq S_g - 2 \) do \( g \leftarrow g + 1; \)
   6. \( Q \leftarrow n - S_g - 2 - 1; \)
   7. \( i \leftarrow -1; \)
   8. while \( F_3^{i+1} \leq Q \) do \( i \leftarrow i + 1; \)
   9. \( len \leftarrow g + 3; \)
   10. for \( z = 4 \) to \( len - 2 - i \) do SetBit(buffer,\( z \),0);
   11. while \( i \geq 0 \) do
      12. if \( F_3^i \leq Q \) then
         13. SetBit(buffer,len - i - 1,1);
         14. \( Q \leftarrow Q - F_3^i; \)
      15. end
      16. else
         17. \( i \leftarrow i - 1; \)
      18. end
      19. SetBit(buffer,0,1);
      20. SetBit(buffer,1,1);
      21. SetBit(buffer,2,1);
      22. SetBit(buffer,3,0);
   23. end

Algorithm 6: A conventional encoding algorithm for the Fibonacci code of order 3

In Lines 13–14, In Line 15, we compute \( Q = n - S_g - 2 - 1 = 35 - 28 - 1 = 6 \). Since \( F_2^{(3)} < 6 \), we compute \( i = 2 \) in Lines 16–17. The bit-length of the output codeword is set to \( len = g + 3 = 9 \) in Line 18. Since \( len - 2 - i = 9 - 2 - 2 = 5 \), we set the second part of the code, \( 0_2 \), from the 4-th to the 5-th position (we use the reverse Fibonacci code) in Line 19: buffer = 00XXXX. In the while loop in Line 20, we set bits of the Fibonacci representation of \( Q \). Since \( F_2^{(3)} = 4 \leq 6 \) in Line 21, we compute \( len - i - 1 = 9 - 2 - 1 = 6 \) and set the 1-bit in the 6-th position in Line 22: buffer = 100XXXX. Then we compute \( n = 6 - 4 = 2 \) and \( i = i - 1 = 1 \). Since \( F_1^{(3)} = 2 \leq 2 \), we set the 1-bit in the 7-th position: buffer = 1100XXXX, \( n = 2 - 2 = 0 \) and \( i = 0 \). Since \( F_0^{(3)} = 1 > 0 \), we set the 0-bit in the 8-th position and \( i = -1 \): buffer = 01100XXXX. Since \( i < 0 \), we continue with Line 30. In Lines 30–33, we set the 0111 delimiter: buffer = 011000111.
A decoding algorithm for the Fibonacci code of order 3 is shown in Algorithm 7. In Lines 1–3, we read the first three bits. If these bits are 1-bits, we set $n = 1$ and the algorithm ends in Lines 4–7. In other cases, we read the next bit in stream. If the current three bits are the 1-bits, we set $n = 2$ and the algorithm ends in Lines 8–12. In Lines 15–22, we read bits one by one from the input stream until the three last bits are 1-bits. If we read the 1-bit in the $i$-th position, we add the $F_i^{(3)}$ number to $Q$ in Line 16. To remember the last four bits of a codeword read from the input stream (the delimiter 0111), we use four variables $bit_0$ – $bit_3$ in Lines 18–21. In Line 23, the index $g$ of the nearest lower Fibonacci sum is computed from the number of bits read so far. Finally, in Line 24, the number $n$ is computed from $Q$ by adding the nearest lower Fibonacci sum.

```
input : stream including a codeword of the Fibonacci code of order 3
output: n, a decoded number
1 bit3 ← GetNextBit(stream);
2 bit2 ← GetNextBit(stream);
3 bit1 ← GetNextBit(stream);
4 if bit1 = 1 and bit2 = 1 and bit3 = 1 then
5     n ← 1;
6     End;
7 end
8 bit0 ← GetNextBit(stream);
9 if bit0 = 1 and bit1 = 1 and bit2 = 1 then
10    n ← 2;
11    End;
12 end
13 Q ← 0;
14 i ← 0;
15 while not (bit0 = 1 and bit1 = 1 and bit2 = 1) do
16    if bit3 = 1 then Q ← Q + F_i^{(3)};
17    i ← i + 1;
18    bit3 ← bit2;
19    bit2 ← bit1;
20    bit1 ← bit0;
21    bit0 ← GetNextBit(stream);
22 end
23 g ← i + 1;
24 n ← Q + S_g^{(3)} + 1;
```

**Algorithm 7**: A conventional decoding algorithm for the Fibonacci code of order 3

### 5.5 Conventional Algorithms for Elias-Fibonacci Code

A conventional encoding algorithm (see Algorithm 8) utilizes the second definition of the Elias-Fibonacci code shown in Section 3.7. The bit-length of a coded number $n$ is computed in Lines 1–2. We need two variables $L1$ and $L2$ containing the bit-length since $L2$ is later iterated down to 0 during encoding the prefix part in Line 9.
The prefix part of the codeword is the reverse Fibonacci representation of \( L_1 \). The position of the highest bit of the prefix part is computed in Lines 3–4 and it is stored in the \( p \) variable. The bit-length of the whole codeword is computed in Line 5. The reverse Fibonacci representation of \( L_1 \) is written in \( \text{buffer} \) in Lines 6–14. The standard binary representation \( B(n) \) of the number \( n \) is written in \( \text{buffer} \) in Lines 15–18.

**Example 9** (Encoding a number by the Elias-Fibonacci code)

Let us consider \( n = 35 \). In Lines 1–2, the bit-length \( L_1 = L_2 = 6 \) is computed. In Lines 3–14, we encode the prefix part of a codeword. Bits of the reverse Fibonacci representation \( F(2)(6) = 1001 \) are written in the following steps. In Line 4, we compute the position of the highest bit of the reverse Fibonacci representation, \( p = 3 \), since \( F_3 = 5 \leq 6 < F_4 = 8 \). In Line 5, we compute the total bit-length of the codeword \( \text{len} = 3 + 6 + 1 = 10 \). In Lines 6–14, we write the bits of the reverse Fibonacci representation. Since \( F_3 = 5 \leq 6 \), we write the highest 1-bit in the position \( \text{len} - p - 1 = 10 - 3 - 1 = 6 \): \( \text{buffer} = 1XXXXXX \). We set \( L_2 = 6 - F_3 = 6 - 5 = 1 \) in Line 9 and \( p = 2 \) in Line 13. Since \( F_2 > L_2 (3 > 1) \), we write the 0-bit in the position \( \text{len} - p - 1 = 10 - 2 - 1 = 7 \): \( \text{buffer} = 01XXXXXX \). We set \( p = 1 \) in Line 13. Since \( F_1 > L_2 (2 > 1) \), we write the 0-bit in the position \( \text{len} - p - 1 = 10 - 1 - 1 = 8 \): \( \text{buffer} = 001XXXXXX \). We set \( p = 0 \) in Line 13. Since \( F_0 \leq L_2 (1 \leq 1) \), we write the 1-bit in the position \( \text{len} - p - 1 = 10 - 0 - 1 = 9 \): \( \text{buffer} = 1001XXXXXX \). We set \( L_2 = 0 \) in Line 9 and \( p = -1 \) in Line 13. Since \( p < 0 \), the while loop ends. The algorithm continues with Lines 15–18 where we write \( L_1 = 6 \) bits of the standard binary representation of \( n = 35_{10} = 100111_2 \) in \( \text{buffer} \): \( \text{buffer} = 1001100011_2 \).

```
input : \( n \), a positive integer
output: \( \text{buffer} \), an Elias-Fibonacci codeword of \( n \) with the bit-length \( \text{len} \)

1 \( L_1 \leftarrow L(n) \);
2 \( L_2 \leftarrow L_1 \);
3 \( p \leftarrow 0 \);
4 while \( F_{p+1} \leq L_1 \) do \( p \leftarrow p + 1 \);
5 \( \text{len} \leftarrow p + L_1 + 1 \);
6 while \( p \geq 0 \) do
7     if \( F_p \leq L_2 \) then
8         SetBit(\( \text{buffer} \), \( \text{len} - p - 1 \), 1);
9         \( L_2 \leftarrow L_2 - F_p \);
10     else
11         SetBit(\( \text{buffer} \), \( \text{len} - p - 1 \), 0);
12     end
13     \( p \leftarrow p - 1 \);
14 end
15 for \( i = 0 \) to \( L_1 - 1 \) do
16     \( \text{bit} \leftarrow \text{GetBit}(n, i) \);
17     SetBit(\( \text{buffer} \), \( \text{len} \), \( \text{bit} \));
18 end

Algorithm 8: A conventional encoding algorithm for the Elias-Fibonacci code
```

A decoding algorithm (see Algorithm 9) utilizes the first definition of the Elias-Fibonacci code shown in Section 3.7. The \( \text{prevBit} \) variable includes the previous
read bit. The while loop in Lines 4–11 is used to compute the bit-length $L$ of an output number $n$. Since the bit-length is encoded by the reverse Fibonacci code, we read bits one by one until the sequence 11 is identified in Line 7. The bit-length $L$ of the output number is computed in Line 8 by adding the Fibonacci number $F^{(2)}_i$ for each 1-bit read in the $i$-th position of the codeword. Since the highest 1-bit of $B(n)$ is read as a part of the prefix part of the codeword, we set the highest bit of the output number in Line 13. Finally, we read the remaining $L - 1$ bits of $B(n)$, i.e. we read $B'(n)$, from the input stream in Lines 14–18.

```
input: stream including a codeword of the Elias-Fibonacci code
output: n, a decoded number

1 n ← 0;
2 L ← 0;
3 prevBit ← 0;
4 i ← 0;
5 while true do
6     bit ← GetNextBit(stream);
7     if prevBit = 1 and bit = 1 then break;
8     if bit then L ← L + F^{(2)}_i;
9     i ← i +1;
10    prevBit ← bit;
11 end
12 L ← L - 1;
13 SetBit(n,L,1) ;
14 while L > 0 do
15     L ← L -1 ;
16     bit ← GetNextBit(stream);
17     SetBit(n,L,bit) ;
18 end
```

Algorithm 9: A conventional decoding algorithm for the Elias-Fibonacci code
Part III

Fast Encoding and Decoding Algorithms
Chapter 6

Prerequisites of Fast Fibonacci Encoding and Decoding Algorithms

6.1 Introduction

Before a description of fast Fibonacci encoding and decoding algorithms in Chapters 7–9, we need to introduce some prerequisites. These prerequisites are based on properties of the Fibonacci numbers described in Section 6.2. An interval of coded numbers with the same bit-length of the Fibonacci representation of order 2 is defined in Section 6.3. We utilize this interval in Section 8.5.2 to determine the maximal bit-length of the prefix of the Elias-Fibonacci code. In Section 6.4, we describe the Extended Fibonacci representation.

Different Fibonacci shifts (namely Fibonacci left shift, Fibonacci right shift and Extended Fibonacci right shift) are introduced in Section 6.5.1. The Fibonacci right shift is utilized in all fast Fibonacci encoding algorithms, the Fibonacci left shift is utilized in all fast Fibonacci decoding algorithms. Since computation of the Fibonacci shift operations is time consuming, an efficient computation is introduced in Sections 6.5.3–6.5.5. To encode the Fibonacci codes of order 3 and higher we need a computation of the nearest lower Fibonacci sum. Whereas an original algorithm is $O(n)$, in Section 6.6, we introduce an $O(2)$ algorithm.

6.2 Properties of Fibonacci Numbers

A Fibonacci number $F_i^{(m)}$ can be represented as a linear combination of the $i$-th powers of roots of the polynomial $P(m) = x^m - x^{m-1} - \ldots - x - 1$ [12, 42]. The $P(m)$ has only one real root with a norm $> 1$, which we shall denote by $\phi_m$, other $m - 1$ roots have a norm $< 1$. Therefore, when representing $F_i^{(m)}$ as a linear combination, the term with $\phi_m^i$ is the dominant one, whereas other terms rapidly
Prerequisites of Fast Fibonacci Encoding and Decoding Algorithms

become negligible with increasing $i$. Consequently, the $i$-th Fibonacci number of order $m$ can be computed by the following linear equation including only terms with $\phi_m^i$:

$$F_i^{(m)} = \|b_m\phi_m^i\|$$  \hspace{1cm} (6.2.1)

where $b_m$ is the coefficient of the dominating term and $\|\cdot\|$ means rounding to the closest integer. Let us note $\phi_2$ is also known as the golden ratio \([24, 22, 47]\).

In the case of Fibonacci of order 2, $\phi_2 = \frac{1+\sqrt{5}}{2} \approx 1.6180$ and $b_2 = \frac{3\sqrt{5}+5}{10}$ (see \cite{22} for more details). In the case of Fibonacci of order 3, $\phi_3 = \frac{1}{3} \left( \sqrt{19 + 3\sqrt{33}} + \sqrt{19 - 3\sqrt{33}} + 1 \right) \approx 1.8392868$ and $b_3 = 1.1374516$ (see \cite{60}). Some Fibonacci numbers calculated according to Equation 6.2.1 are shown in Table 6.1.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$b_2\phi_2^i$</th>
<th>$|b_2\phi_2^i|$</th>
<th>$b_3\phi_3^i$</th>
<th>$|b_3\phi_3^i|$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.17</td>
<td>1</td>
<td>1.14</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1.89</td>
<td>2</td>
<td>2.09</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3.07</td>
<td>3</td>
<td>3.85</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4.96</td>
<td>5</td>
<td>7.08</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>8.02</td>
<td>8</td>
<td>13.02</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>12.98</td>
<td>13</td>
<td>23.94</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>21.01</td>
<td>21</td>
<td>44.04</td>
<td>44</td>
</tr>
<tr>
<td>7</td>
<td>33.99</td>
<td>34</td>
<td>81</td>
<td>81</td>
</tr>
<tr>
<td>8</td>
<td>55</td>
<td>55</td>
<td>148.98</td>
<td>149</td>
</tr>
<tr>
<td>9</td>
<td>89</td>
<td>89</td>
<td>274.02</td>
<td>274</td>
</tr>
<tr>
<td>10</td>
<td>144</td>
<td>144</td>
<td>504</td>
<td>504</td>
</tr>
</tbody>
</table>

### 6.3 Interval of Numbers with the Same Bit-length of the Fibonacci Representation

In Section 8.5.2, we utilize the following theorem to obtain an interval of coded numbers with the same bit-length of the Fibonacci representation of order 2. It is utilized to compute the maximal bit-length of the prefix in the Elias-Fibonacci code.

**Theorem 1** (An interval of coded numbers with the same bit-length of the Fibonacci representation of order 2)

*Let $L_F = L(F^{(2)}(n))$ be the bit-length of the Fibonacci representation of order 2 of $n$.***
Prerequisites of Fast Fibonacci Encoding and Decoding Algorithms

Then

\[ \exists L_F : \forall n \in [F_{L_F - 1}, F_{L_F} - 1] \Rightarrow L(F^{(2)}(n)) = L_F \]

Proof. The lower bound of the interval represents the Fibonacci representation with only the highest 1-bit; other bits are 0-bits. Since the position of the 1-bit is \( L_F - 1 \), it represents the Fibonacci number \( F_{L_F - 1} \). The next interval’s lower bound is \( F_{L_F} \) and intervals are connected; therefore, the upper bound of the interval is \( F_{L_F} - 1 \). \( \square \)

Example 10 (An interval of coded numbers with the same bit-length of the Fibonacci representation of order 2)

Let us consider all Fibonacci representations of order 2 with the bit-length \( L_F = 6 \).

 Bounds of the interval with the bit-length 6 of the Fibonacci representation are as follows: \( F^{(2)}_5 = 13 \) and \( F^{(2)}_6 - 1 = 21 - 1 = 20 \). Consequently, coded numbers \( n \in [13, 20] \) have the bit-length 6 of the Fibonacci representation. The Fibonacci representations of the bounds are \( F^{(2)}(13) = 100000 \) and \( F^{(2)}(20) = 101010 \).

6.4 Extended Fibonacci Representation

In this section, we introduce another Fibonacci representation based on Fibonacci numbers \( F(m)_i \) with respect to the sign of the subscript \( i \) called the Extended Fibonacci representation. This representation is utilized in a computation of the Extended Fibonacci right shift described in Section 6.5.1.

To identify the negative part we use the symbol •; the values on the left of • belongs to Fibonacci numbers with the positive subscript and the values on the right belongs to Fibonacci numbers with the negative subscript. Note that the symbol • is an analogy of the decimal point in decimal numbers where both parts belong to positive and negative powers of 10.

Definition 3 (The extended Fibonacci representation of an integer \( n \))

A number \( F^{(m)}_{ext}(n) = a_p a_{p-1} \ldots a_0 \cdot a_{-1} a_{-2} \ldots a_{-q}, \ a_i \in \{0, 1\}, \ -q \leq i \leq p \), is the Extended Fibonacci representation of a positive integer \( n \) iff:

\[ \sum_{i=-q}^{p} a_i F^{(m)}_i = n \]

and there is no run of \( m \) consecutive 1-bits in \( F^{(m)}_{ext}(n) \).

One or more Extended Fibonacci representations are defined for one number \( n \). In Table 6.2, we see examples of Extended Fibonacci representations for some numbers denoted by an \( F^{(2)}_{ext} \) subscript. We must note that the first bit from the right of • represents the \( F^{(m)}_{-1} \) Fibonacci number and the bits in positions \( i < -1 \) do not change the sum since \( F^{(m)}_i = 0 \) for \( i < -1 \).

Example 11 (The extended Fibonacci representation of a number)

In Table 6.2, we see that there are two Extended Fibonacci representations for \( n = 1 \):
Table 6.2: Extended Fibonacci representations of order 2 with one Fibonacci number after • for some numbers

<table>
<thead>
<tr>
<th>$n$</th>
<th>$F^{(2)}_{ext}(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 • 1 $F^{(2)}_{ext}$</td>
</tr>
<tr>
<td>1</td>
<td>1 • 0 $F^{(2)}_{ext}$</td>
</tr>
<tr>
<td>2</td>
<td>10 • 0 $F^{(2)}_{ext}$</td>
</tr>
<tr>
<td>3</td>
<td>10 • 1 $F^{(2)}_{ext}$</td>
</tr>
<tr>
<td>3</td>
<td>100 • 0 $F^{(2)}_{ext}$</td>
</tr>
<tr>
<td>4</td>
<td>100 • 1 $F^{(2)}_{ext}$</td>
</tr>
<tr>
<td>4</td>
<td>101 • 0 $F^{(2)}_{ext}$</td>
</tr>
<tr>
<td>5</td>
<td>1000 • 0 $F^{(2)}_{ext}$</td>
</tr>
<tr>
<td>6</td>
<td>1000 • 1 $F^{(2)}_{ext}$</td>
</tr>
<tr>
<td>6</td>
<td>1001 • 0 $F^{(2)}_{ext}$</td>
</tr>
<tr>
<td>7</td>
<td>1010 • 0 $F^{(2)}_{ext}$</td>
</tr>
<tr>
<td>8</td>
<td>1010 • 1 $F^{(2)}_{ext}$</td>
</tr>
<tr>
<td>8</td>
<td>10000 • 0 $F^{(2)}_{ext}$</td>
</tr>
<tr>
<td>9</td>
<td>10000 • 1 $F^{(2)}_{ext}$</td>
</tr>
<tr>
<td>9</td>
<td>10001 • 0 $F^{(2)}_{ext}$</td>
</tr>
<tr>
<td>10</td>
<td>10010 • 0 $F^{(2)}_{ext}$</td>
</tr>
</tbody>
</table>

$0 \cdot 1_{F^{(2)}_{ext}}$ and $1 \cdot 0_{F^{(2)}_{ext}}$. However, there is generally an unlimited number of representations for $n = 1$, e.g. $1 \cdot 01_{F^{(2)}_{ext}}$ or $1 \cdot 000001_{F^{(2)}_{ext}}$ and so on.

Let us consider $n = 3$. Some Extended Fibonacci representations are as follows: $F^{(2)}_{ext}(3) = 10 \cdot 1$, $F^{(2)}_{ext}(3) = 100 \cdot 0$ or $F^{(2)}_{ext}(3) = 10 \cdot 101$. To calculate the $n$ value from $F^{(2)}_{ext}(3) = 10 \cdot 101$ we apply the following sum:

$$n = \sum_{i=-3}^{1} 1 \times F_{1} + 0 \times F_{0} + 1 \times F_{-1} + 0 \times F_{-2} + 1 \times F_{-3}$$

$$= 1 \times 2 + 0 \times 1 + 1 \times 1 + 0 \times 0 + 1 \times 0$$

$$= 3$$

6.5 Fibonacci Shift Operations and their Computation

6.5.1 Fibonacci Shift Operations

The Fibonacci shift operations, introduced in [4, 42, 6], are required for the bit manipulation in the fast encoding and decoding algorithms of the Fibonacci code described in Chapters 7–9.
**Definition 4** (The Fibonacci shift operations)
Let \( F(n) = a_p \ldots a_2a_1a_0 \) be the Fibonacci representation of an integer \( n \) and \( k \geq 0 \) be an integer. The \( k \)-th Fibonacci left shift \( n <<_F k \) is defined as follows:

\[
n <<_F k = a_p \ldots a_2a_1a_0 \underbrace{00 \ldots 0}_k
\]

and \( k \)-th Fibonacci right shift is defined as follows:

\[
n >>_F k = a_p \ldots a_{k+2}a_{k+1}a_k
\]

and \( k \)-th Extended Fibonacci right shift is defined as follows:

\[
n >>_{F, e} k = a_p \ldots a_{k+2}a_{k+1}a_k \cdot a_{k-1}a_{k-2} \ldots a_0
\]

Fibonacci and Extended Fibonacci right shifts are not inverse operations to the Fibonacci left shift since the Fibonacci right shift truncates the bits of \( F(n) \) from the position \( k - 1 \) and the Extended Fibonacci right shift truncates the bits of \( F(n) \) from the position \( k - 2 \) (these bits belong to insignificant Fibonacci numbers of the Extended Fibonacci representation since \( F_i^{(m)} = 0 \) for \( i < -1 \)).

**6.5.2 Conventional Computation of Shift Operations**
A conventional approach for a computation of the Fibonacci right shift \( n >>_F k \) (similarly the other Fibonacci shifts) is carried out in the following steps:

1. Encode \( n \) to the Fibonacci representation \( F(n) = a_p \ldots a_2a_1a_0 \)
2. Binary right shift the bits of \( F(n) \) by \( k \); we obtain:
   \( F(n) >> k = a_p \ldots a_{k+2}a_{k+1}a_k \)
3. Compute \( n >>_F k \) using the sum in Definition 1:

\[
n >>_F k = \sum_{i=0}^{(p-k)} (F(n) >> k)_i F_i^{(m)}
\]

**Example 12** (A conventional computation of the Fibonacci right shift)
Let us consider the computation of \( 220 >>_F 8 \):

1. Encode 220 to the Fibonacci representation \( F(220) = 10100110000_F \)
2. Binary shift the bits of \( F(n) \) by \( k \), we obtain: \( F(220) >> 8 = 101_F \)
3. Compute \( 220 >>_F 8 \) using the sum in Definition 1:

\[
\sum_{i=0}^{(11-8=3)} (101_F)_i F_i^{(m)} = 1 \times 3 + 0 \times 2 + 1 \times 1 = 4
\]
As result, $220 >>_F 8 = 4$.

The computation of the Fibonacci shifts is time consuming since $n$ needs to be encoded in Step 1 and the computation in Step 3 requires the sum of Fibonacci numbers for all 1-bits. As result, an efficient calculation of Fibonacci shifts is required; we describe that computation in the following sections.

### 6.5.3 Efficient Computation of Fibonacci Right Shift

**Introduction**

In this section, we describe an efficient computation of the Fibonacci right shift introduced in [6]. The computation is utilized during a fast Fibonacci encoding algorithm to obtain individual segments of the bit-length $S$ of the Fibonacci representation (see Sections 7.3 and 7.4).

Let us consider an interval $[n_{\text{min}}, n_{\text{max}}]$ of all numbers with the $k$-th Fibonacci right shift equal to $x$, i.e., $\forall n \in [n_{\text{min}}, n_{\text{max}}] : n >>_F k = x$. A table including all intervals for a domain is called the *Fibonacci right shift encoding interval table* (FRSEIT). Each row of FRSEIT includes the following values (see Table 6.3 for a fragment of FRSEIT):

- $k$ – the parameter of the Fibonacci right shift
- $x$ – the result of the Fibonacci right shift
- $[n_{\text{min}}, n_{\text{max}}]$ – the interval of numbers having the result $x$ of the Fibonacci right shift
- $F(x)$ – the Fibonacci representation of the result $x$
- $\text{enc}(F(x))$ – the encoded Fibonacci representation – the standard binary representation of $F(x)$. This encoding is symbolically represented by changing an $F$ subscript to a $2$ subscript (see Table 6.3 for an example).
- $L(F(x))$ – the bit-length of the Fibonacci representation $F(x)$

FRSEIT is then utilized to find the $k$-th Fibonacci right shift of $n$; for the $k$-th Fibonacci right shift we find the row where $n \in [n_{\text{min}}, n_{\text{max}}]$. In this row, we can directly read the result $x$ of the Fibonacci right shift $n >>_F k$ as well as the Fibonacci representation $F(x)$. When we consider a binary search algorithm, this procedure has logarithmic complexity.

**Example 13** (A computation of a row in FRSEIT)

The 7-th row of FRSEIT for $k = 8$ is computed as follows: the Fibonacci representation of $x = 7$ is $F(7) = 1010_F$, $\text{enc}(F(x)) = 1010_2 = 10_{10}$. The bit-length of the Fibonacci representation $L(F(7)) = 4$. 

45
To compute the interval \([n_{\text{min}}, n_{\text{max}}]\) of all numbers with the \(k\)-th Fibonacci right shift equal to \(x\), we need to use the \(k\)-th Fibonacci left shift of \(x\). In this way, we obtain the lower bound \(n_{\text{min}}\) of the interval. To obtain the upper bound \(n_{\text{max}}\) of the interval, we use the lower bound of the next interval \([n'_{\text{min}}, n'_{\text{max}}]\); the upper bound \(n_{\text{max}} = n'_{\text{min}} - 1\).

The lower bound \(n_{\text{min}} = 7 << F_8 = 8 = 322\) since \(F(7) << 8 = 1010_F << 8 = 101000000000_F = F(322)\). The next interval’s lower bound \(n'_{\text{min}} = (7 + 1) << F_8 = 8 = 377\) since \(F(7 + 1) << 8 = 10000_F << 8 = 1000000000000_F = F(377)\). The upper bound \(n_{\text{max}} = n'_{\text{min}} - 1 = 377 - 1 = 376\). As result, we obtain the interval \([322, 376]\).

Table 6.3: A fragment of FRSEIT for the Fibonacci code of order 2 and the 8-th and 16-th Fibonacci right shifts

<table>
<thead>
<tr>
<th>Interval for (k = 8)</th>
<th>Interval for (k = 16)</th>
<th>(x)</th>
<th>(F(x))</th>
<th>(\text{enc}(F(x)))</th>
<th>(L(F(x)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n_{\text{min}})</td>
<td>(n_{\text{max}})</td>
<td>(n'_{\text{min}})</td>
<td>(n'_{\text{max}})</td>
<td>(F_{\text{Bibonacci representation}})</td>
<td>(\text{Encoded Fibonacci representation})</td>
</tr>
<tr>
<td>0</td>
<td>54</td>
<td>0</td>
<td>2,583</td>
<td>0</td>
<td>(0_F)</td>
</tr>
<tr>
<td>55</td>
<td>88</td>
<td>2,584</td>
<td>4,180</td>
<td>1</td>
<td>(1_F)</td>
</tr>
<tr>
<td>89</td>
<td>143</td>
<td>4,181</td>
<td>6,764</td>
<td>2</td>
<td>(10_F)</td>
</tr>
<tr>
<td>144</td>
<td>198</td>
<td>6,765</td>
<td>9,348</td>
<td>3</td>
<td>(100_F)</td>
</tr>
<tr>
<td>199</td>
<td>232</td>
<td>9,349</td>
<td>10,945</td>
<td>4</td>
<td>(101_F)</td>
</tr>
<tr>
<td>233</td>
<td>287</td>
<td>10,946</td>
<td>13,529</td>
<td>5</td>
<td>(1000_F)</td>
</tr>
<tr>
<td>288</td>
<td>321</td>
<td>13,530</td>
<td>15,126</td>
<td>6</td>
<td>(1001_F)</td>
</tr>
<tr>
<td>322</td>
<td>376</td>
<td>15,127</td>
<td>17,710</td>
<td>7</td>
<td>(1010_F)</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>1,275</td>
<td>1,308</td>
<td>59,898</td>
<td>61,494</td>
<td>27</td>
<td>(1001001_F)</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>2,173</td>
<td>2,206</td>
<td>102,085</td>
<td>103,681</td>
<td>46</td>
<td>(10010101_F)</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

Size of FRSEIT

FRSEIT is utilized during a fast Fibonacci encoding algorithm to obtain individual segments of the bit-length \(S\) of the Fibonacci representation (see Sections 7.3 and 7.4). Since only some \(k\) of \(k\)-th Fibonacci right shifts are used, the size of FRSEIT depends on:

- The count of various Fibonacci representations with the bit-length \(\leq S\). This count is equal to \(F^m_S - 1\) since the bit of the order \(S\) does not already fit in the segment. When we use larger segments, the size of FRSEIT increases; on the other hand, encoding becomes faster.

- The count of \(k\)-th Fibonacci right shifts used; it depends on \(S\) and the bit-length of the largest Fibonacci representation of coded numbers. Fast encoding algorithms in Sections 7.3 and 7.4 utilize only the \(k\) multiples of \(S\).
These values are precisely described in the following two paragraphs.

The segment bit-length is usually one or two bytes, i.e., $S = 8$ or $S = 16$, respectively. In the case of Fibonacci of order 2, there are $F_8^2 - 1$ Fibonacci representations to fit in one segment; thus $F_8^2 - 1 = 54$ for $S = 8$ and $F_{16}^2 - 1 = 2,583$ for $S = 16$. In the case of Fibonacci of order 3, there are $F_8^3 - 1$ Fibonacci representations to fit in one segment; thus $F_8^3 - 1 = 148$ for $S = 8$ and $F_{16}^3 - 1 = 19,512$ for $S = 16$.

If the 8-bit segment is used, the 8-th Fibonacci right shift is applied to obtain the 8th–15th bits of the Fibonacci representation, the 16-th Fibonacci right shift is applied to obtain the 16th–23rd bits of the Fibonacci representation, and so on. We do not utilize any shift to retrieve the 0th–7th bits of the Fibonacci representation. Consequently, we do not need to know all $k$-Fibonacci right shifts; we only need to know them for some multiples of $S$. When we consider 32-bit-length integers, the bit-length of the largest Fibonacci representation of order 2 is $L(F(2)(2^{32})) = 46$; therefore, we need to utilize the Fibonacci right shift for $k \in \{8, 16, 24, 32, 40\}$ in the case of $S = 8$ and for $k \in \{16, 32\}$ in the case of $S = 16$. The bit-length of the largest Fibonacci representation of order 3 is $L(F(3)(2^{32})) = 37$; therefore, we need $k \in \{8, 16, 24, 32\}$ for $S = 8$ and $k \in \{16, 32\}$ for $S = 16$. In Table 6.3, we see some rows of FRSEIT for the 8-bit segment.

The size of FRSEIT for the segment bit-length $S$ is:

$$S_{FRSEIT} = (F_S - 1) \times \lceil \frac{2^b}{S} \rceil$$

(6.5.1)

where $b$ is the bit-length of the largest codeword. The size of FRSEIT for various segments and number domains is shown in Table 6.4. Evidently, tables are rather short even for 32-bit-length coded numbers.

<table>
<thead>
<tr>
<th>Domain Size</th>
<th>Fibonacci of order 2</th>
<th>Fibonacci of order 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8-bit</td>
<td>16-bit</td>
</tr>
<tr>
<td>8-bit</td>
<td>108</td>
<td>2,583</td>
</tr>
<tr>
<td>16-bit</td>
<td>162</td>
<td>5,166</td>
</tr>
<tr>
<td>32-bit</td>
<td>324</td>
<td>7,749</td>
</tr>
</tbody>
</table>

**Improved Searching in FRSEIT**

To improve logarithmic time complexity of binary searching in FRSEIT, we must consider properties of Fibonacci numbers shown in Section 6.2. Using these properties, we can estimate the result of the $k$-th Fibonacci right shift and the estimation is then utilized to search the correct result of the $k$-th Fibonacci right shift. The estimation is computed according to the following theorem.
Prerequisites of Fast Fibonacci Encoding and Decoding Algorithms

**Theorem 2** (An estimation of the $k$-th Fibonacci right shift)

Let an estimation of the $k$-th Fibonacci right shift be:

$$ n >>_F k \approx \left\| \frac{n}{\phi^k_m} \right\|, $$

where $m$ is the Fibonacci order and $k$ is the parameter of the Fibonacci right shift. The error of the estimation is:

$$ \varepsilon \in \{0, 1\}. $$

**Proof.** Each number can be represented by Fibonacci numbers (see Equation 6.2.1 and Definition 1):

$$ n = \sum_{i=0}^{p} a_i F_i = \sum_{i=0}^{p} a_i F_i = \sum_{i=0}^{p} a_i \left\| b_m \phi^i_m \right\| $$

Let us consider an interval $[n_{\min}, n_{\max}]$ of all numbers with the same $k$-th Fibonacci right shift $x$. The lower bound of the interval $n_{\min}$ does not contain 1-bits in indexes $i \in [0, k-1]$ of the Fibonacci representation (see Definition 4). Therefore we can compute $n_{\min}$ as follows:

$$ n_{\min} = \sum_{i=k}^{p} a_i F_i $$

$n_{\min} >>_F k$ is then computed without an error:

$$ x = n_{\min} >>_F k = \sum_{i=k}^{p} a_i F_i >>_F k = \sum_{i=k}^{p} a_i F_{i-k} = \sum_{i=k}^{p} a_i \left\| b_m \phi^{i-k}_m \right\| = $$

$$ = \sum_{i=k}^{p} a_i \left\| \frac{b_m \phi^i_m}{\phi^k_m} \right\| = \left\| \sum_{i=k}^{p} a_i \left\| \frac{b_m \phi^{i-k}_m}{\phi^k_m} \right\| \right\| = \left\| \frac{n_{\min}}{\phi^k_m} \right\| $$

Instead of estimating the error of truncated bits for other values in the interval $[n_{\min}, n_{\max}]$, we utilize the estimation error of the next interval’s lower bound. All numbers in the next interval $n' \in [n'_{\min}, n'_{\max}]$ have the $k$-th Fibonacci right shift $n' >>_F k = x + 1$. All numbers $n$ with the same $k$-th Fibonacci right shift $x$ belongs to the interval $n \in [n_{\min}, n_{\max}] = [n_{\min}, n'_ {\min})$. Consequently, the estimation of $x = n >>_F k \approx \left\| \frac{n}{\phi^k_m} \right\|$ belongs to the interval:

$$ \left\| \frac{n_{\min}}{\phi^k_m} \right\| \leq \left\| \frac{n}{\phi^k_m} \right\| < \left\| \frac{n'_{\min}}{\phi^k_m} \right\| $$

and we can write:

$$ \left\| \frac{n_{\min}}{\phi^k_m} \right\| \leq \left\| \frac{n}{\phi^k_m} \right\| \leq \left\| \frac{n_{\min}}{\phi^k_m} \right\| + 1 $$

As result, the error of the estimation $\varepsilon$ is not never > 1 and we can write $\varepsilon \in \{0, 1\}$. 

$\square$
input: \( n \), a positive integer, \( k \), the parameter of the Fibonacci right shift, \( m \), the Fibonacci order

output: \( x \), the result of the \( k \)-th Fibonacci right shift of order \( m \), i.e.,
\[
x = n >>_F k
\]

1. \( x' \leftarrow \left\lfloor \frac{n}{\phi_m^k} \right\rfloor \);
2. if \( FRSEIT[x'].n_{\min} > n \) then \( x' \leftarrow x' - 1 \);
3. \( x \leftarrow x' \);

Algorithm 10: A computation of the \( k \)-th Fibonacci right shift of \( n \)

For the computation of the correct result of the \( k \)-th Fibonacci right shift \( n >>_F k = x \), we use Algorithm 10. In Line 1, we estimate the result using Theorem 2; we obtain the estimation \( x' = \left\lfloor \frac{n}{\phi_m^k} \right\rfloor \). The \( x' \) value is also the row order in FRSEIT. In Line 2, we compare the input number \( n \) (the number being Fibonacci shifted) with the interval in the row order \( x' \). If the number is not in the interval, we simply subtract the estimated result by the value 1 and receive the correct result \( x \) of the \( k \)-th Fibonacci right shift. This means that our estimation method decreases logarithmic complexity of the computation to two attempts at the most.

The floating point division and rounding can be replaced by integer operations. For example, in the case of the 8-th Fibonacci right shift of order 2 we can use instead of:
\[
x' = \left\lfloor \frac{n}{\phi_2^8} \right\rfloor = \left\lfloor \frac{n}{1.6180348} \right\rfloor = \left\lfloor \frac{n}{46.9787} \right\rfloor
\]
the following estimation:
\[
x' = \left\lfloor \frac{n}{46.9787} \right\rfloor = \left\lfloor \frac{n}{46.9787} + \frac{1}{2} \right\rfloor \approx \left\lfloor \frac{n \times 1000 + 23,489}{46,978} \right\rfloor
\]

In our experiments, the computation using the integer operations is approximately 20× faster. For other shifts, we can similarly replace the floating point operations used in Theorem 2.

**Example 14** (An efficient computation of the Fibonacci right shift)
Let us consider \( n = 256 \) with the Fibonacci representation \( F(256) = 100001000010_F \). We calculate the 8-th Fibonacci right shift of \( n \), \( 256 >>_F 8 \), by the following procedure. First, we calculate the estimation of the 8-th Fibonacci right shift for this number by means of Theorem 2 as follows:
\[
256 >>_F 8 \approx \left\lfloor \frac{256}{\phi_8^8} \right\rfloor = \left\lfloor \frac{256}{1.6180348} \right\rfloor = \left\lfloor \frac{256}{46.9787} \right\rfloor = \left\lfloor 5.4493 \right\rfloor = 5
\]

After checking the 5th row of FRSEIT for the 8-th Fibonacci right shift (see Table 6.3), we see \( 256 \in [233; 287] \); therefore, the correct result of the 8-th Fibonacci right shift \( 256 >>_F 8 \) is 5. We can then read its Fibonacci representation \( F(5) = 1000_F \) in the row.
Let us consider another example; \( n = 280 \) with the Fibonacci representation \( F(280) = 100010100000_F \). We calculate the following estimation:

\[
280 >> F_8 \approx \frac{280}{46.9787} \approx 5.9601 = 6
\]

After checking the interval in the 6th row of FRSEIT, we see the number \( 280 \not\in [288; 321] \); therefore, the estimation is not correct and we must decrease the estimated value by 1. Consequently, the correct result of the 8-th Fibonacci right shift \( 280 >> F_8 \) is 5. We can then read its Fibonacci representation \( F(5) = 1000_F \) in the 5th row of the table.

### 6.5.4 Efficient Computation of Extended Fibonacci Right Shift

Bits in positions \( i = -2 \) and lower of the Extended Fibonacci representation belong to the insignificant Fibonacci numbers (see Definition 3); we can set all these bits to 0-bits, but the value it represents is not changed. Consequently, the Fibonacci representation and the Extended Fibonacci representation differ only in the bit in the position \(-1\). We can therefore compute the Extended Fibonacci right shift using the Fibonacci right shift; we find the function \( f \) of \( n >> F_{\text{ext}} k = f(n >> F_{k-1}) \).

In this section, this function is implemented by searching in the **Extended Fibonacci right shift encoding interval table** (ExFRSEIT).

Let us consider an interval \([n_{\text{min}}, n_{\text{max}}]\) of all numbers with the \((k-1)\)-th Fibonacci right shift \( x_{k-1} \), i.e., \( \forall n \in [n_{\text{min}}, n_{\text{max}}] : n >> F_{k-1} = x_{k-1} \). Each row of ExFRSEIT includes the following values (see Table 6.5 for a fragment of the table):

- \( k \) – the parameter of the Extended Fibonacci right shift
- \( x_{k-1} \) – the result of the \((k-1)\)-th Fibonacci right shift (see Definition 1)
- \([n_{\text{min}}, n_{\text{max}}]\) – the interval of numbers having the result \( x_{k-1} \) of the \((k-1)\)-th Fibonacci right shift
- \( F(x_{k-1}) \) – the Fibonacci representation of the result \( x_{k-1} \)
- \( L(F(x_{k-1})) \) – the bit-length of the Fibonacci representation of \( x_{k-1} \)
- \( x_k \) – the result of the \( k \)-th Extended Fibonacci right shift (see Definition 3)
- \( F_{\text{ext}}(x_k) \) – the Extended Fibonacci representation of the result \( x_k \)

### Example 15 (A computation of a row of ExFRSEIT)

The 7-th row of ExFRSEIT for \( k = 8 \) is computed as follows. Set \( x_{k-1} = 7 \); its Fibonacci representation is \( F(x_{k-1}) = F(7) = 1010_F \) and the bit-length of the Fibonacci representation \( L(F(x_{k-1})) = L(F(7)) = 4 \).
Prerequisites of Fast Fibonacci Encoding and Decoding Algorithms

**Algorithm 11**: A computation of the \( k \)-th Extended Fibonacci right shift of \( n \)

Table 6.5: A fragment of ExFRSEIT for the Fibonacci code of order 2 and the 8-th and 16-th Extended Fibonacci right shifts

<table>
<thead>
<tr>
<th>Interval for ( k = 8 )</th>
<th>Interval for ( k = 16 )</th>
<th>( x_{k-1} )</th>
<th>( F(x_{k-1}) )</th>
<th>( L(F(x_{k-1})) )</th>
<th>( F_{\text{ext}}(x_k) )</th>
<th>( x_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n_{\text{min}} )</td>
<td>( n_{\text{max}} )</td>
<td>( n_{\text{min}} )</td>
<td>( n_{\text{max}} )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>33</td>
<td>0</td>
<td>1,596</td>
<td>0</td>
<td>( 0_F )</td>
<td>0</td>
</tr>
<tr>
<td>34</td>
<td>54</td>
<td>1,597</td>
<td>2,583</td>
<td>1</td>
<td>( 1_F )</td>
<td>1</td>
</tr>
<tr>
<td>55</td>
<td>88</td>
<td>2,584</td>
<td>4,180</td>
<td>2</td>
<td>( 2_F )</td>
<td>2</td>
</tr>
<tr>
<td>89</td>
<td>122</td>
<td>4,181</td>
<td>5,777</td>
<td>3</td>
<td>( 3_{\text{ext}} )</td>
<td>3</td>
</tr>
<tr>
<td>123</td>
<td>143</td>
<td>5,778</td>
<td>6,764</td>
<td>4</td>
<td>( 4_{\text{ext}} )</td>
<td>4</td>
</tr>
<tr>
<td>144</td>
<td>177</td>
<td>6,765</td>
<td>8,361</td>
<td>5</td>
<td>( 5_{\text{ext}} )</td>
<td>5</td>
</tr>
<tr>
<td>178</td>
<td>198</td>
<td>8,362</td>
<td>9,548</td>
<td>6</td>
<td>( 6_{\text{ext}} )</td>
<td>6</td>
</tr>
<tr>
<td>199</td>
<td>232</td>
<td>9,549</td>
<td>10,945</td>
<td>7</td>
<td>( 7_{\text{ext}} )</td>
<td>7</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
</tbody>
</table>

To compute the interval \([n_{\text{min}}, n_{\text{max}}]\) of all numbers with the \((k-1)\)-th Fibonacci right shift \( x_{k-1} \), we utilize the \((k-1)\)-th Fibonacci left shift of \( x_{k-1} \). In this way, we obtain the lower bound \( n_{\text{min}} \) of the interval. To obtain the upper bound \( n_{\text{max}} \) of the interval, we use the lower bound of the next interval \([n'_{\text{min}}, n'_{\text{max}}]\): \( n_{\text{max}} = n'_{\text{min}} - 1 \).

Consequently, the lower bound \( n_{\text{min}} = 7 \ll_F 7 = 199 \) since \( F(7) \ll 7 = 1010_F \ll 7 = 10100000000_F = F(199) \). The next interval’s lower bound \( n'_{\text{min}} = (7 + 1) \ll_F 7 = 233 \) since \( F(7 + 1) \ll 7 = 10000_F \ll 7 = 10000000000_F = F(233) \). Consequently, \( n_{\text{max}} = n'_{\text{min}} - 1 = 233 - 1 = 232 \).

The extended Fibonacci representation is computed by the 1-st Extended Fibonacci right shift of \( F(x_{k-1}) = F(7) = 1010_F \), i.e., \( F_{\text{ext}}(x_k) = 1010_F \gg_F 1 = 101 \circ 0_{\text{ext}} \). The integer it represents is computed by the sum in Definition 3 as follows:

\[
x_k = 1 \times F_2 + 0 \times F_1 + 1 \times 0 + 0 \times F_{-1} = 1 \times 3 + 0 + 1 \times 1 + 0 \times 1 = 3 + 1 = 4.
\]

The computation of the Extended Fibonacci right shift is shown in Algorithm 11 and it follows Algorithm 10 for the efficient computation of the Fibonacci right shift. An estimation of \( n \gg_F k \) is computed in Line 1 of the algorithm by using the estimation of \( n \gg_F (k - 1) \), the estimation is corrected in Line 2. The result \( x_k \) of the Extended Fibonacci right shift is read in ExFRSEIT in Line 3.

**Example 16** (A computation of the 8-th Extended Fibonacci right shift)

Let us consider \( n = 200 \) with the Fibonacci representation \( F(200) = 10100000001_F \). We calculate the 7-th Fibonacci right shift of \( n \) using Algorithm 10: \( 200 \gg_F 7 = 7 \).
Then we directly read the result of the Extended Fibonacci right shift $200 \gg_{\text{Ext 8}} = 4$ in the 7-th row of ExFRSEIT.

### 6.5.5 Efficient Computation of Fibonacci Left Shift

In [4], we introduce an efficient computation of the Fibonacci left shift based on Fibonacci numbers and the Extended Fibonacci right shift. It is used in the fast Fibonacci decoding algorithm described in Section 9.3.

**Theorem 3** (The computation of the $k$-Fibonacci left shift for order $m = 2$)

The $k$-Fibonacci left shift of a number $n$ for order $m = 2$ is:

$$n \ll_F k = F_{k-1}^{(2)} \times n + F_{k-2}^{(2)} \times (n \gg_{\text{Ext 1}})$$

**Proof.** In this proof, we use $F(n)$ instead of $F^{(2)}(n)$ and $F_i$ instead of $F_i^{(2)}$. This theorem can be proved by mathematical induction. First, we show that the theorem holds for $k = 0$ and $k = 1$ (base case). Let us have $F(n) = a_0a_1a_2 \ldots a_p$ then

$$n \ll_F 0 = F_{-1} \times n + F_{-2} \times (n \gg_{\text{Ext 1}}) = 1 \times n + 0 \times (n \gg_{\text{Ext 1}}) = n$$

$$n \ll_F 1 = \sum_{i=0}^{p} a_i F_{i+1} = a_0F_1 + a_1F_2 + \ldots + a_pF_{p+1} = a_0(F_0 + F_{-1}) + a_1(F_1 + F_0) + \ldots + a_p(F_p + F_{p-1}) = (a_0F_0 + a_1F_1 + \ldots + a_pF_p) + (a_0F_{-1} + a_1F_0 + \ldots + a_pF_{p-1}) = \sum_{i=0}^{p} a_iF_i + \sum_{i=0}^{p} a_iF_{i-1} = n + (n \gg_{\text{Ext 1}}) = 1 \times n + 1 \times (n \gg_{\text{Ext 1}}) = F_0 \times n + F_{-1} \times (n \gg_{\text{Ext 1}})$$

By induction hypothesis, it is assumed that this theorem holds for all $j$, $0 \leq j < k$. We must prove the following equation:

$$n \ll_F k = F_{k-1} \times n + F_{k-2} \times (n \gg_{\text{Ext 1}})$$

52
The k-Fibonacci left shift for order \( m = 3 \)

The k-Fibonacci left shift of a number \( n \) for order \( m = 3 \) is:

\[
n <<_F F = \sum_{i=0}^{p} a_i F_{k+i} = \sum_{i=0}^{p} a_i F_{k+i-1} + \sum_{i=1}^{p} a_i F_{k+i-2}
\]

\[
= n <<_F (k - 1) + n <<_F (k - 2)
\]

\[
= F_{k-2} \times n + F_{k-3} \times (n >>_{F_{ext}} 1) + F_{k-4} \times n + (n >>_{F_{ext}} 1)
\]

\[
= (F_{k-2} + F_{k-3}) \times n + (F_{k-3} + F_{k-4}) \times (n >>_{F_{ext}} 1)
\]

\[
= F_{k-1} \times n + F_{k-2} \times (n >>_{F_{ext}} 1)
\]

\[\square\]

**Theorem 4** (The computation of the k-Fibonacci left shift for order \( m = 3 \))

The k-Fibonacci left shift of a number \( n \) for order \( m = 3 \) is:

\[
n <<_F k = F_{k-1}^{(3)} \times n + (F_{k-2}^{(3)} + F_{k-3}^{(3)}) \times (n >>_{F_{ext}} 1) + F_{k-2}^{(3)} \times (n >>_{F_{ext}} 2)
\]

**Proof.** In this proof, we use \( F(n) \) for \( F^{(3)}(n) \) and \( F_i \) for \( F_i^{(3)} \). This theorem can be proved by mathematical induction as well. First, we show that the statement holds for \( k = 0, k = 1, \) and \( k = 2 \) (base case). Let us have \( F(n) = a_0a_1a_2 \ldots a_p \) then

\[
n <<_F 0 = F_{-1} \times n + (F_{-2} + F_{-3}) \times (n >>_{F_{ext}} 1) + F_{-2} \times (n >>_{F_{ext}} 2)
\]

\[
= 1 \times n + (0 + 0) \times (n >>_{F_{ext}} 1) + 0 \times (n >>_{F_{ext}} 2)
\]

\[
= n
\]

\[
n <<_F 1 = \sum_{i=0}^{p} a_i F_i+1 = a_0 F_1 + a_1 F_2 + \ldots + a_p F_{p+1}
\]

\[
= a_0(F_0 + F_{-1} + F_{-2}) + a_1(F_1 + F_0 + F_{-1}) + \ldots + a_p(F_p + F_{p-1} + F_{p-2})
\]

\[
= (a_0 F_0 + a_1 F_1 + \ldots + a_p F_p) +
\]

\[
+ (a_0 F_{-1} + a_1 F_0 + \ldots + a_p F_{p-1})+
\]

\[
+ (a_0 F_{-2} + a_1 F_{-1} + \ldots + a_p F_{p-2})
\]

\[
= \sum_{i=0}^{p} a_i F_i + \sum_{i=0}^{p} a_i F_{i-1} + \sum_{i=0}^{p} a_i F_{i-2}
\]

\[
= n + (n >>_{F_{ext}} 1) + (n >>_{F_{ext}} 2)
\]

\[
= 1 \times n + (1 + 0) \times (n >>_{F_{ext}} 1) + 1 \times (n >>_{F_{ext}} 2)
\]

\[
= F_0 \times n + (F_{-1} + F_{-2}) \times (n >>_{F_{ext}} 1) + F_{-1} \times (n >>_{F_{ext}} 2)
\]
By induction hypothesis, it is assumed that this theorem holds for all $j$, $0 \leq j < k$. We must prove the following equation:

\[
\begin{align*}
\sum_{i=0}^{p} a_i F_{i+2} &= a_0 F_2 + a_1 F_3 + \ldots + a_p F_{p+2} \\
&= a_0 (F_1 + F_0 + F_{-1}) + a_1 (F_2 + F_1 + F_0) + \ldots + a_p (F_{p+1} + F_p + F_{p-1}) \\
&= (a_0 F_{i+1} + a_1 F_0 + \ldots + a_p F_{p+1}) + \\
&\quad + (a_0 F_0 + a_1 F_1 + \ldots + a_p F_p) + \\
&\quad + (a_0 F_{-1} + a_1 F_0 + \ldots + a_p F_{p-1}) \\
&= 2 \times n + 2 \times (n >> \text{ext} 1) + 1 \times (n >> \text{ext} 2) \\
&= F_1 \times n + (F_0 + F_{-1}) \times (n >> \text{ext} 1) + F_0 \times (n >> \text{ext} 2)
\end{align*}
\]

\[
\begin{align*}
\sum_{i=0}^{p} a_i F_{k+i} &= F_{k-1} \times n \\
&\quad + (F_{k-2} + F_{k-3}) \times (n >> \text{ext} 1) \\
&\quad + F_{k-2} \times (n >> \text{ext} 2)
\end{align*}
\]
The computation of the Fibonacci left shift according to Theorem 3 or 4 is used in the fast Fibonacci decoding algorithm described in Section 9.3. This computation requires the 1-st Extended Fibonacci right shift for Fibonacci of order 2 and the 2-nd Extended Fibonacci right shift for Fibonacci of order 3. Since this shift is applied only on individual segments and the number of integers in a segment is low, we do not need to apply a computation of the Extended Fibonacci right shift described in Section 6.5.4 for a general value of the parameter $k$; we utilize a precomputed table of the 1-st and 2-nd Extended Fibonacci right shifts for all integers in a segment.

For the 8-bit segment, there are 55 various values of the 1-st Extended Fibonacci right shift for Fibonacci of order 2 since the highest Fibonacci representation with the bit-length $L(F(2)) = 8$ is $F(2)(54) = 10101010$. In Table 6.6, we see all values of the 1-st Extended Fibonacci right shift for Fibonacci of order 2 and the 8-bit segment. For the 16-bit segment, there are 2,584 various values of the 1-st Extended Fibonacci right shift for Fibonacci of order 2 since the highest Fibonacci representation with the bit-length $L(F(2)) = 16$ is $F(2)(2,583) = 1010101010101010$.

Table 6.6: All values of the 1-st Extended Fibonacci right shift for Fibonacci of order 2 and the 8-bit segment

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n \gg F_{ext}$</th>
<th>$n$</th>
<th>$n \gg F_{ext}$</th>
<th>$n$</th>
<th>$n \gg F_{ext}$</th>
<th>$n$</th>
<th>$n \gg F_{ext}$</th>
<th>$n$</th>
<th>$n \gg F_{ext}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>6</td>
<td>20</td>
<td>12</td>
<td>30</td>
<td>19</td>
<td>40</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
<td>7</td>
<td>21</td>
<td>13</td>
<td>31</td>
<td>19</td>
<td>41</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>12</td>
<td>8</td>
<td>22</td>
<td>14</td>
<td>32</td>
<td>20</td>
<td>42</td>
<td>26</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>13</td>
<td>8</td>
<td>23</td>
<td>14</td>
<td>33</td>
<td>21</td>
<td>43</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>14</td>
<td>9</td>
<td>24</td>
<td>15</td>
<td>34</td>
<td>21</td>
<td>44</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>15</td>
<td>9</td>
<td>25</td>
<td>16</td>
<td>35</td>
<td>22</td>
<td>45</td>
<td>28</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>16</td>
<td>10</td>
<td>26</td>
<td>16</td>
<td>36</td>
<td>22</td>
<td>46</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>17</td>
<td>11</td>
<td>27</td>
<td>17</td>
<td>37</td>
<td>23</td>
<td>47</td>
<td>29</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>18</td>
<td>11</td>
<td>28</td>
<td>17</td>
<td>38</td>
<td>24</td>
<td>48</td>
<td>30</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>19</td>
<td>12</td>
<td>29</td>
<td>18</td>
<td>39</td>
<td>24</td>
<td>49</td>
<td>30</td>
</tr>
</tbody>
</table>

Example 17 (A computation of the Fibonacci left shift)

Let us consider $n = 32$ and its Fibonacci representation $F(2)(32) = 1010100$. The 1-st Extended Fibonacci right shift of 32 is: $32 \gg F_{ext} = 101010 \bullet 0_{F_{ext}} = 20$ (see Table 6.6). The 3-rd Fibonacci left shift is computed as follows by means of Theorem 3:

$$n \ll F = 3 = F_{3-1} \times 32 + F_{3-2} \times 20 = 3 \times 32 + 2 \times 20 = 96 + 40 = 136$$

6.6 Efficient Computation of Nearest Lower Fibonacci Sum

To encode the Fibonacci code of order 3 and higher by the conventional as well as fast algorithms (see Sections 5.4 and 7.4), we need to find the nearest lower Fibonacci sum; it is used to determine the codeword bit-length. A simple method is to use a binary search algorithm with logarithmic complexity finding the Fibonacci sum $S_g$ in a precomputed table according Definition 2 (see some examples of the Fibonacci
sum in Table 3.12). In this section, we describe a more efficient calculation using an estimation of the nearest lower Fibonacci sum introduced in [6]. In [12], we can find the proof of the following theorems.

**Theorem 5** (The number of codewords with the same bit-length)
The Fibonacci code of order $m$ contains precisely $F_{g-1}$ codewords of the bit-length not exceeding $m + g$, where $g \geq 0$.

**Theorem 6** (The computation of the Fibonacci sum)
Let $S_g = \sum_{i=-1}^{g} F_i$, for $g \geq -1$. Then

$$S_g = \frac{1}{m-1} \left( F_{g+2} + \sum_{i=0}^{m-3} (m - 2 - i) F_{g-i} - 1 \right) \quad (6.6.1)$$

where $g \geq -1, m \geq 2$.

Theorem 5 implies that the largest integer able to be represented by the Fibonacci code of order $m$ with the codeword bit-length $m + g$ is $S_{g-1}$. It also implies that each integer $n$ in a set of all integers $n$ with the same codeword bit-length $m + g$ belongs to the interval (see [12] for more detail):

$$n \in (S_{g-2} , S_{g-1}] \quad (6.6.2)$$

The lower bound of this interval $S_{g-2}$ is called the nearest lower Fibonacci sum. Our method finds the nearest lower Fibonacci sum more efficiently than the binary search algorithm; it is based on an estimation of the $g$ value – the index of the Fibonacci sum.

For $m = 3$, the Fibonacci sum can be calculated, according to Theorem 6 and Equation 6.2.1, by:

$$S_g = (F_{g+2} + F_g - 1)/2 = (\|b\phi_3^{g+2}\| + \|b\phi_3^g\| - 1)/2 \quad (6.6.3)$$

If we change indexes in Equation 6.6.3 we obtain:

$$S_{g-2} = (\|b\phi_3^g\| + \|b\phi_3^{g-2}\| - 1)/2 \quad (6.6.4)$$

When we follow Interval 6.6.2, we simply put $S_{g-2} \approx n$ for an estimation of the $g$ value. We obtain the following function:

$$n \approx \|b\phi_3^g\| + \|b\phi_3^{g-2}\| - 1$$

If we ignore the rounding of real numbers, we obtain, after some adjustments:

$$n \approx \frac{b\phi_3^g + b\phi_3^{g-2} - 1}{2}$$

$$2n + 1 \approx b\phi_3^g + b\phi_3^{g-2}$$

$$2n + 1 \approx b\phi_3^g (1 + \phi_3^{-2})$$

$$\frac{2n+1}{b(1+\phi_3^{-2})} \approx \phi_3^g$$
Prerequisites of Fast Fibonacci Encoding and Decoding Algorithms

an estimation of $g$ by means of the following function:

$$g \approx \log_{\phi_3} \frac{2n + 1}{b(1 + \phi_3^{-2})}$$

The computation of the logarithmic function is time consuming because the computation requires the usage of floating-point arithmetic. Therefore, we estimate the $g$ value by another logarithmic function which can be calculated without the usage of floating-point arithmetic; we use the bit-length of the standard binary representation of $n$ which is the logarithmic function $L = L(n) = \lceil \log_2(n + 1) \rceil$. Evidently, for the Fibonacci code of order 3, the value $\phi_3 = 1.8392868$ is close to the base of the logarithm (the value 2). To obtain the correct value $k$ from the estimated value, we define the $EstimateFibSum$ table including the following values:

- $L = L(n)$, the bit-length of the standard binary representation of $n$
- The $g$ value of the interval $(S_{g-2}, S_{g-1}]$.
- The interval $(S_{g-2}, S_{g-1}]$. This interval is calculated for the lowest $n$ with the bit-length $L$.

```
input : n, a positive integer
output: g, the index of the nearest lower Fibonacci sum $S_{g-2}$
1  L ←L(n);
2  g ←EstimateFibSum[L];
3  while $S_{g-1} < n$ do  g ← g + 1;
```

Algorithm 12: Finding the nearest lower Fibonacci sum

The $EstimateFibSum$ table, precomputed for 32 bit-length numbers, is shown in Table 6.7. In Algorithm 12, we see how to find the correct nearest lower Fibonacci sum using this table. Since each interval in the table is precomputed for the lowest number $n$ with the bit-length $L(n)$, the higher numbers with the same bit-length can belong to the next interval. Therefore, the algorithm works as follows:

- Compute $L = L(n)$, the bit-length of the standard binary representation of $n$ (Line 1 of the algorithm).
- Estimate the $g$ value by accessing the $L$-th row of $EstimateFibSum$ (Line 2).
- In a loop (Line 3), compare the $n$ number with the upper bound of $(S_{g-2}, S_{g-1}]$ in the $L$-th row of $EstimateFibSum$. If $n$ does not belong to this interval, increase the $g$ value and continue with the loop. Otherwise, $g$ is the correct value.

Table 6.7 also shows the maximal error of the $g$ value estimation. The estimation error is not greater than the difference of the $g$ value between two consecutive rows in the table. Evidently, the maximal number for a specific bit-length $L$ can
Prerequisites of Fast Fibonacci Encoding and Decoding Algorithms

Table 6.7: The EstimateFibSum table for 32 bit-length numbers

<table>
<thead>
<tr>
<th>L</th>
<th>g</th>
<th>(S_{g-2})</th>
<th>(S_{g-1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>15</td>
<td>28</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>28</td>
<td>52</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>52</td>
<td>96</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>96</td>
<td>177</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>177</td>
<td>326</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>326</td>
<td>600</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>600</td>
<td>1,104</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>2,031</td>
<td>3,736</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>3,736</td>
<td>6,872</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>6,872</td>
<td>12,640</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>12,640</td>
<td>23,249</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>23,249</td>
<td>42,762</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>42,762</td>
<td>78,652</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>78,652</td>
<td>144,664</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>144,664</td>
<td>266,079</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>489,396</td>
<td>900,140</td>
</tr>
<tr>
<td>21</td>
<td>23</td>
<td>900,140</td>
<td>1,655,616</td>
</tr>
<tr>
<td>22</td>
<td>24</td>
<td>1,655,616</td>
<td>3,045,153</td>
</tr>
<tr>
<td>23</td>
<td>25</td>
<td>3,045,153</td>
<td>5,600,910</td>
</tr>
<tr>
<td>24</td>
<td>26</td>
<td>5,600,910</td>
<td>10,301,680</td>
</tr>
<tr>
<td>25</td>
<td>27</td>
<td>10,301,680</td>
<td>18,947,744</td>
</tr>
<tr>
<td>26</td>
<td>28</td>
<td>18,947,744</td>
<td>34,850,335</td>
</tr>
<tr>
<td>27</td>
<td>29</td>
<td>34,850,335</td>
<td>64,099,760</td>
</tr>
<tr>
<td>28</td>
<td>30</td>
<td>64,099,760</td>
<td>117,897,840</td>
</tr>
<tr>
<td>29</td>
<td>31</td>
<td>117,897,840</td>
<td>216,847,936</td>
</tr>
<tr>
<td>30</td>
<td>32</td>
<td>216,847,936</td>
<td>398,845,537</td>
</tr>
<tr>
<td>31</td>
<td>33</td>
<td>398,845,537</td>
<td>733,591,314</td>
</tr>
</tbody>
</table>

belong to the interval specified in the next row of the table. Therefore, the maximal difference of \(g\) between two consecutive rows of the EstimateFibSum table is 2; see rows \(L \in \{11, 12\}\), \(L \in \{19, 20\}\), and \(L \in \{26, 27\}\). As result, the precise \(g\) value is found by maximally two loops of the cycle in the algorithm.

Let us note that the Fibonacci sum intervals for \(g \in \{12, 21, 29\}\) are not included in EstimateFibSum. It is therefore these Fibonacci sum intervals are always estimated for \(g - 1\) and the \(g\) value is corrected in the first loop of Algorithm 12.

**Example 18** (A computation of the nearest lower Fibonacci sum)

*Let us consider \(n = 2,048 = 1000\ 0000\ 0000_2\). The bit-length of the standard binary representation \(L = L(n) = 12\). In the 12th row of EstimateFibSum, we read the estimated value \(g = 13\) and the interval \(S_{g-2} = S_{13-2} = S_{11} = 2,031\), \(S_{g-1} = S_{13-1} = S_{12} = 3,736\). Since \(S_{11} \leq n \leq S_{12}\), it is a correct interval; the nearest lower Fibonacci sum is \(S_{11}\) and \(g = 13\).*

*Let us consider \(n = 511 = 1\ 1111\ 1111_2\). The bit-length of the standard binary representation is \(L = L(n) = 9\). In the 9th row of EstimateFibSum, we read the estimated value \(g = 9\) and the interval \(S_{g-2} = S_{9-2} = S_7 = 177\), \(S_{g-1} = S_{9-1} = S_8 = 326\). Since \(S_8 < n\), it is not a correct interval. We must check the next interval with \(g = 9 + 1 = 10\), and bounds \(S_{g-2} = S_{10-2} = S_8 = 326\), \(S_{g-1} = S_{10-1} = S_9 = 600\). Since \(S_8 \leq n \leq S_9\), the correct nearest lower Fibonacci sum is \(S_8\) and \(g = 10\).*
Since, a value can be maximally found in three Fibonacci sum intervals, i.e. two comparisons must be processed at the most, this estimation provides lower time complexity than the binary search algorithm. Moreover, we do not utilize floating-point arithmetic.
Chapter 7

Fast Encoding Algorithms

7.1 Introduction

Whereas the conventional encoding algorithms process data bit by bit, the fast algorithms process data by a higher number of bits. Since the Elias-delta and Elias-Fibonacci encoding algorithms are simpler, they encode numbers only using precomputed tables (see Section 7.2). Fast Fibonacci encoding algorithms described in Sections 7.3 and 7.4 utilize the efficient computation of the Fibonacci right shift described in Section 6.5.3 to compute individual segments of the Fibonacci representation in a codeword. In the case of Fibonacci of order 3, we also utilize the efficient computation of the nearest lower Fibonacci sum described in Section 6.6.

7.2 Fast Elias-delta and Elias-Fibonacci Encoding Algorithms

In fast Elias-delta and Elias-Fibonacci encoding algorithms, we separately encode both parts of the Elias-delta code $E_\delta(n) = 0_{Z-1}B(L)B'(n)$ as well as the Elias-Fibonacci code $EF(n) = reverse(F^{(2)}(L))B(n)$ (see Sections 3.5.2 and 3.7): the first part of codewords is a prefix, $0_{Z-1}B(L)$ or $reverse(F^{(2)}(L))$, and the second part is the binary part, $B(n)$ – the standard binary representation of $n$ or $B'(n)$ – the standard binary representation of $n$ without the highest 1-bit. The fast Elias-delta and Elias-Fibonacci encoding algorithms are shown in Algorithms 13 and 14, respectively. Although these algorithms encode a coded number in more bits than the conventional algorithms, they do not use the concept of $S$ bit-length segments like fast Fibonacci encoding algorithms described in next sections. Since encoding of $B(n)$ or $B'(n)$ is rather straightforward, we must aim our effort to encoding the bit-length in the prefix. Due to the fact that the count of various bit-lengths is relatively low, we can create short tables to encode the bit-length to the prefix of a codeword.
In more detail, the prefixes of Elias-delta and Elias-Fibonacci codes are directly encoded by means of tables called the \textit{EDTAB} and \textit{EFTAB} tables, respectively. These tables include the encoded prefixes for each bit-length $L(n)$ in the \textit{PREFIX} column. The bit-lengths of these prefixes are stored in the \textit{LEN} column. As mentioned previously, these tables are short; they include only 16 rows for all 16 bit-length coded numbers (see Table 7.1) and 32 rows for all 32 bit-length coded numbers. In Lines 2 and 3 of Algorithms 13 and 14, these tables are utilized, the prefix is then written in Line 4.

After encoding the prefix, $B(n)$ or $B'(n)$ must be written in \textit{buffer}. In the case of Elias-Fibonacci, we do not need any encoding of the standard binary representation; $B(n)$ is directly written in \textit{buffer}. However, in the case of Elias-delta, we must compute $B'(n)$. We know the bit-length $L(n)$ and we compute $B'(n)$ by means of the following function:

$$B'(n) = n \text{ AND } (1 << (L(n) - 1))^*$$ \hfill (7.2.1)

where $^*$ denotes the 1’s complement of the standard binary representation and \text{AND} is a binary operator. We can save some execution time by preparing the \textit{COMP} column of \textit{EDTAB} including all $(1 << (L(n) - 1))^*$ (see Table 7.1). Consequently, $B'(n)$ is computed as follows:

$$B'(n) = n \text{ AND } \text{EDTAB}[L(n)].\text{COMP}$$

This computation is utilized in Line 5 of Algorithm 13. In the last two lines of both algorithms, $B(n)$ or $B'(n)$ is written in \textit{buffer} and the bit-length is computed.

<table>
<thead>
<tr>
<th>$L = L(n)$</th>
<th>\textbf{EDTAB}</th>
<th>\textbf{EFTAB}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\textbf{PREFIX}</td>
<td>\textbf{LEN}</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>00100</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>00101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>00110</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>00111</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>0001000</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>0001001</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>0001010</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>0001011</td>
<td>7</td>
</tr>
<tr>
<td>12</td>
<td>0001100</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>0001101</td>
<td>7</td>
</tr>
<tr>
<td>14</td>
<td>0001110</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>0001111</td>
<td>7</td>
</tr>
<tr>
<td>16</td>
<td>000010000</td>
<td>9</td>
</tr>
</tbody>
</table>
Fast Encoding Algorithms

**Example 19 (The fast Elias-delta encoding algorithm)**

Let us have a number \( n = 58 \) and its standard binary representation \( 111010_2 \). In the first line of Algorithm 13, we compute \( L = L(58) = 6 \). In Lines 2–3, we obtain \( \text{prefixLen}=5 \) and \( \text{prefix}=00110 \) using EDTAB. The prefix is stored in \( \text{buffer} \) starting the position \( 6 - 1 = 5 \): \( \text{buffer}=00110XXXXX \) (see Line 4). In Line 5, we compute \( 58 \text{ AND } 1111111111011111_2 = 11010_2 = 26 \). This value is stored in \( \text{buffer} \) in Line 6: \( \text{buffer}=0011011010 \). The bit-length of the codeword is computed in the last line: \( \text{len}= 6 + 5 - 1 = 10 \).

### 7.3 Fast Encoding of Fibonacci Code of Order 2

A fast encoding algorithm for the Fibonacci of code order 2 is shown in Algorithm 15. This algorithm utilizes an efficient computation of the Fibonacci right shift by means of \( FRSEIT \) described in Section 6.5.3. In this algorithm, we use the names \( \text{encFx}, LFx, \) and \( N\text{min} \) for the \( FRSEIT \)'s columns \( \text{enc}(F(x)), L(F(x)), \) and \( n\text{min}, \) respectively (see Table 6.3). Two indices of \( FRSEIT[k][x] \) address the \( x \)-th row of \( FRSEIT \) and the \( k \)-th Fibonacci right shift.

A number \( n \) is encoded in parts and individual segments of the bit-length \( S \) are written in \( \text{buffer} \). Due to the fact that the bits of the Fibonacci representation are stored in the reverse order, we need to write segments and bits of the segments in the reverse order as well. When the bit-length of the Fibonacci representation is not divided by \( S \) without remainders, the bit-length of the highest segment is shorter than \( S \).
Fast Encoding Algorithms

Algorithm 15: A fast encoding algorithm for the Fibonacci code of order 2

In Lines 2–7 of Algorithm 15, the numbers lower than $F_S$ are directly encoded by $FRSEIT$. For larger numbers, we first need to find the maximal $k$ in Line 8; this value is the parameter of the Fibonacci right shift. The $k$ value also determines the number of segments used. The input number is encoded by $kS + 1$ segments in $kS$ steps. Since the bit-length of the highest segment can be shorter than $S$, this segment is separately encoded in Lines 9–14. The order $x$ of the current $FRSEIT$’s row is computed by the $k$-Fibonacci right shift in Line 9. The efficient computation of the $k$-Fibonacci right shift by means of Algorithm 10 is utilized. In the $x$-th row of $FRSEIT$, we read the bit-length of the Fibonacci representation included in the current segment in Lines 10 and the encoded Fibonacci representation in Line 11. The bits of the codeword are written in $buffer$ from the position 1 (Line 11); the position 0 is reserved for the 1-bit delimiter. The bit-length of the codeword is computed in Line 13. In Line 14, we subtract the lower bound $n_{\text{min}}$ from $n$. In this way, we remove the bits of the highest segment from $n$. In Lines 15–25, all other segments are similarly encoded and written in $buffer$. Finally, in Line 27, the 1-bit delimiter is written in $buffer$.

1The reverse operation is a part of the $\text{SetBits}$ function, another way is to compute and include the reverse bits in $FRSEIT$ by the $\text{reverse(enc(F(x)))}$ function.
Figure 7.1: An example of the fast Fibonacci encoding algorithm; the encoding of 17,327: Segment 2

**Example 20** (Encoding the number 17,327 with the fast Fibonacci of order 2 algorithm)

Let us have encoding the number 17,327 using the 8-bit segment \( (S = 8) \). The count of segments needed is computed in Line 8 of Algorithm 15. Since 17,327 ∈ \( [F_{16}; F_{24}] = [2,582; 121,393] \), we find the parameter \( k \) of the Fibonacci right shift for the highest segment in Line 8; \( k = 16 \). This implies the number of segments needed for encoding: \( k + S + 1 = 16 + 8 + 1 = 25 \).

Encoding the highest segment (Segment 2) is depicted in Figure 7.1. After the 16-th Fibonacci right shift of 17,327, we obtain the order \( x \) of the current FRSEIT’s row by \( x = 17,327 >> F_{16} = 7 \) (Line 9). The result of the shift is depicted in STEP 1 of the figure. In STEP 2, this result is encoded using the 7-th row of FRSEIT by \( \text{enc}(F(7)) = 1010 \) with the bit-length \( L(F(7)) = 4 \) in Lines 10–11. These bits are reversely written in buffer in Line 12 from the position 1 (see STEP 3 and STEP 4 of this figure); bits of the codeword not used in this computation are grey highlighted. The bit-length of the codeword is computed by \( \text{len} = L(F)x + k + 1 = 4 + 16 + 1 = 21 \) (Line 13). In FRSEIT, we also read the number 17,327 ∈ \( [15,127; 17,710] \); therefore, the value of other two segments (Segments 0 and 1) is computed by \( 17,327 - n_{\min} = 17,327 - 15,127 = 2,200 \) (Line 14).

Encoding the next segments is carried out in the loop in Lines 15–25. In Line 16, we compute the \( k \) parameter of the Fibonacci right shift for Segment 1 by \( k = k - S = 16 - 8 = 8 \). Encoding Segment 1 is depicted in Figure 7.2. After the 8-th Fibonacci right shift of 2,200 (the value of Segments 1 and 0) in Line 18, we obtain the order \( x \) of the current FRSEIT’s row by \( x = 2200 >> F_8 = 46 \). The result of the shift is depicted in STEP 1 of the figure. Consequently, the
Figure 7.2: An example of the fast Fibonacci encoding algorithm; the encoding of 17,327: Segment 1

encoded Fibonacci representation of \( x \) is retrieved in the 46-th row of FRSEIT; \( \text{enc}\, F\, x = \text{enc}(F(46)) = 10010101_2 = 149_{10} \) (Line 20). This situation is depicted in STEP 2 of the figure. In Line 21, we reversely write \( S \) bits of \( \text{enc}\, F\, x \) in buffer from the position \( \text{len} - k - S = 21 - 8 - 8 = 5 \) (see STEP 3 and STEP 4 of the figure); bits of the codeword not used in this computation are grey highlighted. In FRSEIT, we read that the number \( 2,200 \in [2,173;2,206] \); therefore, the value of the last segment (Segments 0) is computed by \( 2,200 - n_{\text{min}} = 2,200 - 2,173 = 27 \) (Line 19).

The last Segment (Segment 0) with the value 27 is directly encoded by the 27-th row of FRSEIT; \( \text{enc}\, F\, x = \text{enc}(F(27)) = 1001001_2 = 73_{10} \) (Line 20). Since \( k = 0 \), the \( k \)-th Fibonacci right shift is not applied in Line 18. In Line 21, we reversely write \( S \) bits of \( \text{enc}\, F\, x \) in buffer from the position \( \text{len} - k - S = 21 - 0 - 8 = 13 \). The loop ends since there is no other segments to encode. Finally, the 1-bit delimiter is written in buffer in the position 0 (Line 27).

### 7.4 Fast Encoding of Fibonacci Code of Order 3

A fast encoding algorithm for the Fibonacci code of order 3 is shown in Algorithm 16. Let us remember that each codeword of the Fibonacci code of order 3 includes the reverse Fibonacci representation of \( Q \) and the 0111 delimiter (see Section 3.6.2 for more details). The fast encoding algorithm of the Fibonacci code of order 3 is similar to the fast encoding algorithm of the Fibonacci code of order 2; however, it differs in some parts. Special cases, numbers 1 and 2, are encoded in Lines 2–8. In Line 9, we need to compute the bit-length of \( n \). The index \( g \) of the nearest lower Fibonacci
sum is computed in Lines 10–11. The efficient computation of the nearest lower Fibonacci sum by means of Algorithm 12 is utilized. The nearest lower Fibonacci sum is utilized to compute the $Q$ value in Line 12. In Line 13, we compute the bit-length of the codeword.

In Lines 14–15, we compute the maximal parameter $k$ of the Fibonacci right shift. The $k$ value also determines the number of segments used by the encoding algorithm. The input number is encoded by $kS + 1$ segments in $kS + 1$ steps. The Fibonacci representation of $Q$ is appended with 0-bits in Line 21 to extend the bit-length of the codeword to $m + g$ bits. The number of the 0-bits is computed in Line 20. The 0-bits are written from the position 4 since positions 0–3 are reserved for the 0111 delimiter.

The highest segment with the $\frac{k}{2}$ order is encoded in Lines 16–18. The order $x$ of the current FRSEIT’s row is computed by the $k$-Fibonacci right shift in Line 16. The efficient computation of the $k$-Fibonacci right shift by means of Algorithm 10 is utilized. In the $x$-th row of FRSEIT, we read the bit-length of the Fibonacci representation in Lines 17 and the encoded Fibonacci representation in Line 18. The bits of the codeword are written in buffer in Line 19.

Encoding other segments in Lines 22–33 is similar to the encoding of the Fibonacci code of order 2 (see Lines 14–24 of Algorithm 15); however, we use the $Q$ variable instead of the $n$ variable. The 0111 delimiter is written in Line 34.

**Example 21** (Encoding the number $n = 779$ with the fast encoding algorithm for Fibonacci of order 3)

Let us consider $n = 779 = 1100001011_2$. We compute its bit-length $L(779) = 10$ (see Line 9 of Algorithm 16). We estimate the $g$ value of the Fibonacci sum interval $(S_g−2; S_g−1]$ as $g = \text{EstimateFibSum}[10] = 10$ in Line 10; $(S_{g−2}; S_{g−1}) = (S_{10−2}; S_{10−1}) = (S_8; S_9) = (326; 600)$. Since $10 \notin (326; 600]$, we increment $g = 10 + 1 = 11$ (Line 11). Since $779 \in (S_{g−2}; S_{g−1}) = (S_{11−2}; S_{11−1}) = (S_9; S_{10}) = (600; 1, 104]$, we can continue with Line 12 where we compute $Q = n − S_{g−2} − 1 = n − S_9 − 1 = 779 − 600 − 1 = 178$. The bit-length of the codeword is computed by $\text{len} = g + 3 = 11 + 3 = 14$ (Line 13). Since $178 \in \left[\frac{F_8(3)}{3}; \frac{F_{16}(3)}{2}\right] = [149; 19, 513]$, we find the parameter $k$ of the Fibonacci right shift in Lines 14–15; $k = 8$. This implies that the number of segments needed for encoding is $\frac{k}{2} + 1 = \frac{8}{2} + 1 = 2$.

The highest segment (Segment 1) is encoded in Lines 16–18. The Fibonacci right shift is used to obtain the order $x$ of the current FRSEIT’s row by $x = Q >> F 8 = 100100101_F >> 8 = 1_F = 1$ (Line 16). In the 1-st row of FRSEIT, we read the encoded Fibonacci representation and the bit-length of $F(x)$; $\text{encFx} = \text{enc}(F(1)) = 1_2$ and the bit-length $LFx = L(F(1)) = 1$ (Lines 10–11). This bit is written in buffer in the position $\text{len} − k − LFx = 14 − 8 − 1 = 5$; buffer = XXXXXXXXXX (Line 20). The 0-bits are appended to make the codeword bit-length $m + g$ (Line 20–21). We write $z = \text{len} − k − LFx − 4 = 14 − 8 − 1 − 4 = 1$ 0-bits in buffer in the position 4; buffer = XXXXXXXXX10XXX.
**input**: $n$, a positive integer

**output**: buffer, a Fibonacci of order 3 codeword of $n$ with the bit-length $len$

```plaintext
1 \[ k \leftarrow 0 ; \]
2 \[ \text{if } n = 1 \text{ then} \]
3 \[ \quad \text{SetBits(buffer},0,111_2,3); \]
4 \[ \quad len \leftarrow 3; \]
5 \[ \text{else if } n = 2 \text{ then} \]
6 \[ \quad \text{SetBits(buffer},0,0111_2,4); \]
7 \[ \quad len \leftarrow 4; \]
8 \[ \text{else} \]
9 \[ \quad len \leftarrow L(n); \]
10 \[ \quad g \leftarrow \text{EstimateFibSum[len]}; \]
11 \[ \quad \text{while } S_g - 1 < n \text{ do } g \leftarrow g + 1; \]
12 \[ \quad Q \leftarrow n - S_g - 2 - 1; \]
13 \[ \quad len \leftarrow g + 3; \]
14 \[ \quad k \leftarrow 0; \]
15 \[ \quad \text{while } F_{3 + k}^3 < Q \text{ do } k \leftarrow k + S; \]
16 \[ \quad x \leftarrow Q \gg_F k; \]
17 \[ \quad LFx \leftarrow \text{FRSEIT}[k][x].LFx; \]
18 \[ \quad \text{encFx} \leftarrow \text{FRSEIT}[k][x].\text{encFx}; \]
19 \[ \quad \text{SetBits(buffer},len-k-LFx,\text{encFx},LFx); \]
20 \[ \quad z \leftarrow len - k - LFx - 4; \]
21 \[ \quad \text{SetBits(buffer},4,0,z); \]
22 \[ \quad Q \leftarrow Q - \text{FRSEIT}[k][x].Nmin; \]
23 \[ \quad \text{while } k > 0 \text{ do} \]
24 \[ \quad \quad k \leftarrow k - S; \]
25 \[ \quad \quad \text{if } F_k^3 \le Q \text{ then} \]
26 \[ \quad \quad \quad x \leftarrow Q \gg_F k; \]
27 \[ \quad \quad \quad Q \leftarrow Q - \text{FRSEIT}[k][x].Nmin; \]
28 \[ \quad \quad \quad \text{encFx} \leftarrow \text{FRSEIT}[k][x].\text{encFx}; \]
29 \[ \quad \quad \quad \text{SetBits(buffer},len-k\cdot\text{encFx},S); \]
30 \[ \quad \quad \text{else} \]
31 \[ \quad \quad \quad \text{SetBits(buffer},len-k-S,0,S); \]
32 \[ \quad \text{end} \]
33 \[ \text{end} \]
34 \[ \text{SetBits(buffer},0,0111_2,4); \]
```

**Algorithm 16**: A fast encoding algorithm for the Fibonacci code of order 3

The value included in Segment 0 is computed by $Q = Q - n_{\min} = 178 - 149 = 29$ in Line 22. Since the 0-th Fibonacci right shift would be computed in Line 24 for Segment 0, we do not compute any shift, and the Fibonacci representation $F(30) = 00100101_{F(3)}$ is reversely written from the position $len - k - S = 14 - 0 - 8 = 6$ in buffer in Line 29: buffer = 1010010010XXXX. Finally, in Line 34, the 0111 delimiter is written in buffer in positions 0 – 3: buffer = 10100100100111.
Chapter 8

General Principles of Fast Decoding Algorithms

8.1 Introduction

The fast decoding algorithms described in the following chapter process data segment by segment instead of bit by bit as the conventional decoding algorithms. In this chapter, we provide a theoretical background of the fast decoding algorithms. These algorithms are based on a finite automaton with precomputed mapping tables. A general fast decoding algorithm based on an automaton is described in Section 8.2. Section 8.3 describes an identification of the automaton states by a brute-force algorithm. In Section 8.4, we explain the building of mapping tables in more details. Since the number of automaton states is rather high, we propose two types of the automaton reduction: the first reduction using a similarity of automaton states is described in Section 8.5, the second reduction using a shift operation is depicted in Section 8.6. The number of automaton states after the reductions is depicted in 8.7.

8.2 General Fast Decoding Algorithm

The basic idea of all proposed fast algorithms is to decode codewords segment by segment instead of bit by bit. As previously proposed, these algorithms are based on a finite automaton. Each state of the automaton represents a position in the corresponding bit-oriented algorithm described in Chapter 5. A precomputed mapping table is used for each state of the automaton.

The first step of the automaton construction is an identification of all its states. The second step of the construction is to create a mapping table (MAP) for each automaton state and the segment bit-length. Each segment of an input stream and state define the order of a row in the mapping table.
Let $N_{\text{states}}$ denote the number of the automaton states, the size of the mapping table is computed as follows:

$$S_{\text{MAP}} = N_{\text{states}} \times 2^{S}$$  \hfill (8.2.1)

The precomputed mapping table allows a conversion of input stream’s segments directly to decoded numbers. The mapping table also defines the new automaton state for each segment. Evidently, the size of the mapping table exponentially increases with the segment bit-length. In Chapter 10, we show that the proposed method can produce very good results even for small segment bit-lengths like 1B. It is important since the 1B segment is handled quickly and it allows the mapping table to be a reasonable size to fit in L1d or L2 CPU’s caches.

In more details, each record of the mapping table includes the following attributes:

- **State** – the current automaton state
- **Segment** – bits of the input stream including a part of one or more codewords or one or more complete codewords
- **NewState** – the next state of the finite automaton
- **OutputCount** – the amount of numbers which are decoded after processing an actual segment. The maximum value is equal to $\left\lceil \frac{S}{L_{\text{min}}} \right\rceil + 1$, where $L_{\text{min}}$ is the minimal bit-length of a codeword. For example, $L_{\text{min}} = 2$ for the Fibonacci code of order 2, $L_{\text{min}} = 1$ for the Elias-delta code.
- **Numbers** – result decoded numbers
- **Ls** – bit-lengths of the result decoded numbers (if necessary)
- **U** – bits of the partially decoded number. Bits of the next segment(s) complete $U$.
- **L_U** – the bit-length of $U$
- **Rest** – the number of bits in the next segments needed to complete $U$

In Algorithm 17, a general fast decoding algorithm based on the finite automaton is depicted. Some attributes of the mapping table described above are not used in this basic algorithm. These attributes are utilized in code-specific fast algorithms proposed in Chapter 9. This algorithm works as follows:

1. Set the automaton to the initial state and the number of bits read to 0 (Lines 1–2).
2. Read the current segment from the input stream. This algorithm is finished if all segments are read (Lines 3–4).
input : stream – codewords, $S$ – the segment bit-length, $\text{bits}$ – the number of input bits
output: result – an array of decoded numbers

1. $\text{state} \leftarrow 0$;
2. $x \leftarrow 0$;
3. while $x < \text{bits}$ do
   4. $\text{segment} \leftarrow \text{GetBits}(\text{stream}, S)$;
   5. $\text{map} \leftarrow \text{MAP}[	ext{state}, \text{segment}]$;
   6. $\text{result} \leftarrow \text{result} \bigcup_{i=1}^{\text{map.\text{OutputCount}-1}} \text{map.\text{Numbers}[i]}$;
   7. $\text{state} \leftarrow \text{map.\text{NewState}}$;
   8. $x \leftarrow x + S$;
4. end

Algorithm 17: A general fast decoding algorithm based on the finite automaton

3. Find the current record in the mapping table: the record for the current segment and the state of the automaton (Line 5).
4. Output all numbers, i.e. the values in Numbers of the current record (Line 6).
5. Change the state of the automaton to a new one defined in the current record, meaning the NewState attribute, and continue with Step 2 (Line 7).
6. Increment the number of bits read by the segment bit-length $S$ (Line 8).

8.3 Identification of Automaton States with Brute-force Approach

The first step of the automation construction is an identification of all automaton states. A simple solution is to use a brute-force algorithm; however, it leads to many states of the automaton. In the brute-force approach, we must remember one complete codeword or incomplete codeword in each automaton state since the codeword can be stored in more input segments. Since the brute-force algorithm leads to many states of the automaton, in Section 8.5, two methods of the automaton reduction are proposed.

Let us consider that the maximal bit-length of a codeword is $L_{\text{max}}$. Consequently, each incomplete codeword is of the bit-length $L_U < L_{\text{max}}$. Each $L_U$ defines $2^{L_U}$ combinations of 0- and 1-bits. We obtain the total number of states for the brute-force approach when we summarize all readings of incomplete codewords plus one reading of the complete codeword as follows:

$$N_{\text{States}} = \sum_{L_U < L_{\text{max}}} 2^{L_U} + 1 = \sum_{x=1}^{L_{\text{max}}-1} 2^x + 1 = 2^{L_{\text{max}}-1} - 1 + 1 = 2^{L_{\text{max}}} - 1 \quad (8.3.1)$$
Example 22 (An identification of automaton states with the brute-force approach)

Let us consider that the maximal bit-length of a codeword is \( L_{\text{max}} = 3 \) and \( S = 8 \). One complete codeword and all incomplete codewords for the maximal bit-length are depicted in Figure 8.1. In this figure, the \( x \) symbol denotes a bit of an unknown value. Each line in the figure corresponds to one automaton state; therefore, there are \( N_{\text{states}} = 2^{L_{\text{max}}-1} = 2^3-1 = 7 \) states of the automaton.

More precisely, we define the following states of the brute-force approach:

- \( \text{state} = 0 \) – the initial state
- \( \text{state} = -1 \) – error state. When a state is the error state it means the input bit stream contains an unknown codeword.
- \( \text{states} > 0 \) – these automaton states are used to remember the state of a corresponding bit-oriented algorithm and the incomplete codeword. In the brute-force approach, bits of the incomplete codeword are stored in the \( U \) variable with the bit-length \( L_U \) as follows:

\[
L_U = \lfloor \log_2(\text{state} + 1) \rfloor
\]
\[
U = \text{state} - 2^{L_U} + 1
\]

Consequently, we can use a simple formula for the calculation of the state from the incomplete codeword \( U \) with the bit-length \( L_U \):

\[
\text{state} = \begin{cases} 
2^{L_U} - 1 + U, & \text{for } L_U < L_{\text{max}} \\
-1, & \text{for } L_U \geq L_{\text{max}}
\end{cases}
\]  

(8.3.2)

Example 23 (A state of the automaton)

Let us consider state = 6. The corresponding incomplete codeword stored in this state is obtained as \( U = 6 - 2^2 + 1 = 3 \) with \( L_U = \lfloor \log_2(6 + 1) \rfloor \approx \lfloor 2.807 \rfloor = 2 \). Therefore, the incomplete codeword is 11 (see the 6-th line in Figure 8.1).
For \( state = 0 \), we obtain \( U = L_U = 0 \) corresponding to the complete codeword. This state is the initial state of the automaton (see the last line in Figure 8.1). For \( state = 4 \), we obtain \( U = 1 \) and \( L_U = 2 \). In this case, the incomplete codeword stored in the automaton state is 01 (see the 4-th line in Figure 8.1).

8.4 Mapping Table Building

The second step of the automaton construction is to create the mapping table \( MAP \) for each automaton state and each segment value. An algorithm of building the mapping table based on the conventional bit-oriented algorithm is shown in Algorithm 18.

```
output: mapping table MAP
for state ← 0 to N_{states} − 1 do
  for segment ← 0 to 2^S − 1 do
    \{U, L_U\} ← GetBitsForState(state);
    SetBits(stream,0,U,L_U);
    SetBits(stream,L_U.segment,S);
    \{OutputCount, Numbers, U, L_U, Rest, Ls\} ← BitAlgorithm(stream);
    MAP[state,segment].NewState ← GetNewState(U,L_U);
    MAP[state,segment].Rest ← Rest;
    MAP[state,segment].U ← U;
    MAP[state,segment].L_U ← L_U;
    MAP[state,segment].segment ← segment;
    MAP[state,segment].OutputCount ← OutputCount;
    MAP[state,segment].Numbers ← Numbers;
    MAP[state,segment].Ls ← Ls;
  end
end
```

Algorithm 18: An algorithm of mapping table building

This algorithm includes two loops; the first loop covers all states of the automaton and the second loop covers all possible values of the segment. One row of MAP is set in each inner loop utilizing the following functions:

- \( GetBitsForState \) – returns an incomplete codeword in \( U \) and its bit-length in \( L_U \), these values are computed from the automaton state (see Section 8.3).
- \( SetBits \) – fills \( stream \) with \( U \) and \( L_U \) in Line 4 and with the segment value in Line 5.
- \( BitAlgorithm \) – is any bit-oriented conventional algorithm described in Chapter 5. This algorithm processes the \( stream \) filled in Lines 4–5. During processing of the stream, a conventional algorithm first processes bits filled in Line 4. As a result of this processing, the state of the conventional algorithm corresponds to the defined automaton state and it is prepared to process bits of the current segment filled in Line 5. When the \( stream \) is empty, the algorithm ends. Decoded numbers are stored in the \( OutputCount, Numbers, \) and \( Ls \).
variables. The other output values $U$ and $L_U$ contain an incomplete codeword and its bit-length, these values also define the new state of the automaton (Line 7). The $Rest$ variable defines the number of bits in the next segment(s) needed to complete the incomplete codeword $U$.

- **GetNewState** – this function computes a new state of the automaton for $U$ and $L_U$. This new state is set in the $NewState$ variable in Line 7. In other words, the new state corresponds to the incomplete codeword defined by $U$ and $L_U$. Consequently, there are as many states as the number of all incomplete codeword combinations. If any new state is not found, it indicates an error in the input stream and further decoding is not possible. In this case, $-1$ is returned.

**Example 24** (Building the mapping table by the brute-force algorithm)

Let us consider state $= 4$ and the 8-bit segment with the value: segment $= 134_{10} = 10000110_2$. Using GetBitsForState, we obtain the incomplete codeword $01$ (the codeword in Example 23 in Section 8.3). In this way, the stream is initialized with the bits $011000110$. If the Elias-delta code is used, the first 4 bits $0110$ are decoded as the number 2. Other bits (000110) are completed in the next segment(s); therefore, $Outcount = 1$, $Numbers = \{2\}$, $Ls = \{4\}$, $U = 6$, and $L_U = 6$. Using Equation 8.3.2, the new state is obtained by $NewState = 2^{L_U} - 1 + U = 2^6 - 1 + 6 = 69$.

**Example 25** (The size of the mapping table for the brute-force approach of the Elias-Fibonacci code)

Let us consider $S = 16$ and $n \in [0, 2^{32} - 1]$. The longest codeword bit-length is $L_{\text{max}} = 32 + 7 = 39$ (32 bits for the binary part and 7 bits for the prefix). Using Equations 8.3.1 and 8.2.1, we compute $N_{\text{states}} = 2^{L_{\text{max}} - 1} = 2^{39} - 1$ and $S_{\text{MAP}} = (2^{39} - 1) \times 2^{16} = (2^{39} - 1) \times 2^{16} = 2^{55} - 2^{16}$. The number of states as well as the size of the mapping table are very high.

Let us consider $S = 8$; we compute $N_{\text{states}} = 2^{L_{\text{max}} - 1} = 2^{39} - 1$ and $S_{\text{MAP}} = (2^{39} - 1) \times 2^8 = (2^{39} - 1) \times 2^8 = 2^{47} - 2^8$. The number of states as well as the size of the mapping table are still very high; therefore, we introduce two types of the automaton reduction: a reduction using a similarity of states is described in Section 8.5, a reduction using a shift operation is depicted in Section 8.6.

### 8.5 Automation Reduction Using Similarity of States

In the previous section, we shown that the number of automaton states for the brute-force approach can be high, especially for longer segment bit-lengths. Therefore, we need to reduce the automaton. When we analyse all automaton states created by the brute-force algorithm, we can identify some groups of similar states. The identification is based on an analysis of codewords described in Chapter 3. These similar states can be then supposed as one state without any influence on the automaton.
functionality; a reduction is made in this way. In [40, 41], authors introduce a similar type of the reduction for the Fibonacci code. In [4], we introduce the reduction for more codes.

In Sections 8.5.1, 8.5.2, and 8.5.3 we describe all groups of similar states for each code considered in this work: Elias-delta, Elias-Fibonacci, and Fibonacci of order 2 and 3, respectively. The automaton reduction has no influence on building the mapping table; the loop in Algorithm 18 must be carried in the same way; however, the number of states is reduced. We show results of the reduction in Section 8.7.

8.5.1 Automaton States of Elias-delta Code

For the identification of all automaton states, we analyse Elias-delta codewords $E_δ(n) = 0_{Z-1}B(L)B'(n)$ described in Section 3.5.2. The number of automaton states depends on the bit-length of a coded number. Let us remind that $L = L(n)$ denotes the bit-length of a coded number $n$ and $Z$ denotes the bit-length of $B(L)$; $Z = L(B(L)) = 1 + \lceil \log_2 (L) \rceil$. For a computation of all automaton states, we use the bit-length of the maximum coded number $Z_{max} = 1 + \lceil \log_2 (L_{max}) \rceil$. Each of three parts of the codeword partially decoded by a fast decoding algorithm must be stored in an automaton state. After the codeword is analysed, we identify the following groups of automaton states:

- **Initial state** – it represents the first state of the automaton. In this state, we read bits of a codeword from the first bit.

- **Zero states** – it represents the reading of the leading 0-bits of an Elias-delta codeword. The maximal number of 0-bits is equal to $Z_{max} - 1$; therefore, there is $Z_{max} - 1$ states of the automaton to remember these 0-bits. The number of the states expressed by means of $L_{max}$ is equal to $Z_{max} - 1 = 1 + \lceil \log_2 (L_{max}) \rceil - 1 = \lfloor \log_2 (L_{max}) \rfloor$.

**Example 26** (Zero states for the Elias-delta code)

Let us consider an example with the maximal bit-length of a coded number $L_{max} = 32_{10} = 100000_2$. We need $\lceil \log_2 (32) \rceil = 5$ 0-bits to encode the bit-length of $L_{max}$. In Figure 8.2, all combinations of 0-bits in the 8-bit segment are shown. Let us consider the third line of the figure; three 0-bits are included in the current segment; it means that the last two 0-bits are included in the next segment.

- **Bit-length states** – it represents the reading of $B(L) = B(L(n))$. If we consider only the bit-lengths of power of 2, it means $L \leq L_{max} \leq 2^{Z_{max}}$, the number of the states is equal to $L_{max} - 1$. This value is computed by considering two situations:
  1. $L = 2^{Z_{max}}$ (the longest number) – the number of the states is equal to $Z_{max}$.

General Principles of Fast Decoding Algorithms

Figure 8.2: An example of partially decoded zeros where \( L(n) = L_{\text{max}} = 32_{10} = 100000_2 \)

2. \( L < 2^{Z_{\text{max}}} \) – to compute the number of the states, we must summarize combinations for all \( Z < Z_{\text{max}} \). The summarization is as follows:

\[
\sum_{x=1}^{Z < Z_{\text{max}}} (2^x - 1) = \sum_{x=1}^{Z_{\text{max}}-1} (2^x - 1) = \sum_{x=1}^{Z_{\text{max}}-1} 2^x - (Z_{\text{max}} - 1) \\
= 2^{Z_{\text{max}}-1} - 1 - (Z_{\text{max}} - 1) = 2^{Z_{\text{max}}} - 1 - Z_{\text{max}}
\]

Finally, the overall number of the bit-length states is as follows:

\[
Z_{\text{max}} + 2^{Z_{\text{max}}} - 1 - Z_{\text{max}} = 2^{Z_{\text{max}}} - 1 = L_{\text{max}} - 1 \quad (8.5.1)
\]

**Example 27** (Bit-length states for the Elias-delta code)

*Let us consider the following 2 situations:*

1. We decode a codeword of \( n \) with the maximal bit-length \( L(n) = L_{\text{max}} = 32_{10} = 100000_2, Z_{\text{max}} = 5 \). We can obtain the following 5 states when \( L_{\text{max}} \) is read: 0000010000X, 000001000XX, 00000100XXX, 0000010XXXX, and 000001XXXXX.

2. Let us consider all numbers with the bit-length \( L(n) < L_{\text{max}} = 32, Z < Z_{\text{max}} = 5 \). To obtain all states, we need to summarize all sub-states for \( Z \in \{1, 2, 3, 4\} \). Therefore, we compute \( 2^4 - 1 = 15 \) states for \( z = 4 \), \( 2^3 - 1 = 7 \) for \( z = 3, 3 \) for \( z = 2 \), and 1 state for \( z = 1 \). All states for \( z = 3 \) are depicted in Figure 8.3. Consequently, the result is \( 1 + 3 + 7 + 15 = 26 \) states. The same value is computed by \( 2^{Z_{\text{max}}} - 1 - Z_{\text{max}} = 2^5 - 1 - 5 = 26 \).

As result, we obtain \( 26 + 5 = 31 \) states. The same result is computed by means of Equation 8.5.1: \( L_{\text{max}} - 1 = 32 - 1 = 31 \).

- **Standard binary states** – after the bit-length of the number \( L = L(n) \) is read from the first two parts of a codeword, we read the last part; the standard binary representation of a coded number without the highest 1-bit. The number of the states is equal to \( 2^{Z_{\text{max}}-1} - 1 \).
If we sum all above depicted states, we obtain the following equation for the number of states after the state similarity reduction:

\[ N_{\text{states}} = 1 + \left\lfloor \log_2 (L_{\text{max}}) \right\rfloor + L_{\text{max}} - 1 + 2^{Z_{\text{max}}-1} - 1 \quad (8.5.2) \]

\[ \begin{array}{cccc}
  \cdot & \cdot & \cdot & 0 \ 0 \ 0 \ 1 \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \\
  \cdot & \cdot & \cdot & 0 \ 0 \ 0 \ 1 \ 0 \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \\
  \cdot & \cdot & \cdot & 0 \ 0 \ 0 \ 1 \ 1 \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \\
  \cdot & \cdot & 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \\
  \cdot & \cdot & 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \\
  \cdot & \cdot & 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \\
  \cdot & \cdot & 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \ \underline{x} \\
\end{array} \]

Decoded Unread Segment

\[ \begin{array}{cccc}
  \underline{Z} & \underline{L} & \underline{Z} \\
\end{array} \]

Figure 8.3: All partially decoded bit-lengths where the number of zeros \( Z = 3 \)

### 8.5.2 Automaton States for Elias-Fibonacci Code

For the identification of all automaton states, we analyse Elias-Fibonacci codewords. An Elias-Fibonacci codeword includes two parts: the prefix part representing the bit-length of \( n \) in the reverse Fibonacci representation \( \text{reverse}(F(2)(b)) \) and the standard binary representation \( B(n) \): \( EF(n) = \text{reverse}(F(2)(b))B(n) \). Both parts of the codeword depends on the bit-length of a maximum coded number \( L_{\text{max}} \). Automaton states are as follows:

- **Initial state** – it represents the first state of the automaton; we read bits of a codeword from the first bit.

- **Prefix states** – it represents the reading of the prefix. We denote the maximal bit-length of the Fibonacci representation of \( L_{\text{max}} \) by \( L_{\text{FLmax}} = L(F(L_{\text{max}})) \). According to Theorem 1, \( \exists L_{\text{FLmax}} : L_{\text{max}} \in [F_{L_{\text{FLmax}}-1}, F_{L_{\text{FLmax}}-1}] \Rightarrow L(F(L_{\text{max}})) = L_{\text{FLmax}} \). When the prefix is read from the current segment, we must distinguish if the prefix is complete or it is completed in the next segments:

  1. **Incomplete prefix**

     All incomplete prefixes are of the bit-length \( L_{FL} < L_{\text{FLmax}} \). This number is the lower boundary of the above proposed interval. The number of all incomplete prefixes is computed by the sum of all possible values of the prefix; Theorem 6 is utilized in the following computation:
\[ \sum_{L_{FL} = 1}^{L_{FL_{max}} - 1} F_{L_{FL}} = S_{L_{FL_{max}} - 1} - 2 = F_{L_{FL_{max}} + 1} - 3 \]

2. Complete prefix

All prefixes with the bit-length \( L_{FL_{max}} \) are complete. According Theorem 5: \( n_{max} \in [F_{L_{FL_{max}} - 1}, F_{L_{FL_{max}} - 1}] \). All Fibonacci representations in this interval are of the bit-length \( L_{FL_{max}} \). All Fibonacci representations for numbers \( > n_{max} \) (with the bit-length of the standard binary representation \( L_{max} \)) are not used as prefixes. Therefore, there is exactly \( L_{max} - F_{L_{FL_{max}} - 1} + 1 \) prefixes of the bit-length \( L_{FL_{max}} \) for the Elias-Fibonacci code.

As result, we obtain the following number of prefix states:

\[
F_{L_{FL_{max}} + 1} - 3 + L_{max} - F_{L_{FL_{max}} - 1} + 1 \\
= (F_{L_{FL_{max}} - 1} + F_{L_{FL_{max}}}) - 3 + L_{max} - F_{L_{FL_{max}} - 1} + 1 \\
= F_{L_{FL_{max}}} + L_{max} - 2
\]

\[ (8.5.3) \]

**Example 28** (Prefix states for the Elias-Fibonacci code)

The number of prefix states for the Elias-Fibonacci code and 16 bit-length numbers is computed as it follows. The bit-length of the maximal number is \( L_{max} = 16 \), its Fibonacci representation is \( F(L_{max}) = F(16) = 100100 \) with the bit-length \( L_{FL_{max}} = L(F(L_{max})) = 6 \). Consequently, the bit-length of an incomplete prefix is shorter than 6. We obtain the following number of incomplete prefixes: \( F_{L_{FL_{max}} + 1} - 3 = F_{6+1} - 3 = 34 - 3 = 31 \). As result, states 1 – 31 define incomplete prefixes. There are \( L_{max} - F_{L_{FL_{max}} - 1} + 1 = L_{max} - F_{6-1} + 1 = 16 - 13 + 1 = 4 \) complete prefixes: states 32–35 define the complete prefixes. When Equation 8.5.3 is applied, we obtain the following number of all prefix states: \( F_{L_{FL_{max}}} + L_{max} - 2 = F_{6} + 16 - 2 = 21 + 16 - 2 = 35 \). All prefix states are shown in Table 8.1. Prefixes for \( L \in [0, 12] \) and \( L_{FL} = 6 \) are not possible since all Fibonacci representations are shorter than 6 bits. Similarly, prefixes for \( L > 16 \) and \( L_{FL} = 6 \) are not usable for 16-bit numbers.

In the case of 32 bit-length coded numbers, we need 64 prefix states for the Elias-Fibonacci code, since \( F(32) = 1010100_F \), \( L_{FL_{max}} = L(F(L_{max})) = 7 \), and the number of all prefix states is computed using Equation 8.5.3:

\[ F_{L_{FL_{max}}} + L_{max} - 2 = F_{7} + L_{max} - 2 = 34 + 32 - 2 = 64 \]

- **Standard binary states** – after the bit-length \( L = L(n) \) of \( n \) is read from the first two parts of a codeword, we read the last part; the standard binary representation of the coded number. The number of the states is equal to \( 2^{L_{max}} - 1 \).

If we sum all above depicted states, we obtain the following equation for the number of automaton states after the state similarity reduction:
$N_{states} = 1 + F_{L_{FL_{\text{max}}}} + L_{\text{max}} - 2 + 2^z_{\text{max}} - 1 \quad (8.5.4)$

Table 8.1: The prefix states for the Elias-Fibonacci code and 16 bit-length numbers

<table>
<thead>
<tr>
<th>State</th>
<th>reverse($F(L)$)</th>
<th>$L$</th>
<th>$L_{FL}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XXXXXXXX00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>XXXXXXXX01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>XXXXXXXX000</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>XXXXXXXX10</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>XXXXXXXX010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>XXXXXXXX0000</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>XXXXXXXX100</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>XXXXXXXX0100</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>XXXXXXXX0001</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>XXXXXXXX101</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>XXXXXXXX0000</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>XXXXXXXX1000</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>XXXXXXXX0100</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>XXXXXXXX0001</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>XXXXXXXX1010</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>XXXXXXXX00010</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>XXXXXXXX1001</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>XXXXXXXX0101</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>XXXXXXXX00000</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>XXXXXXXX10000</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>21</td>
<td>XXXXXXXX01000</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>22</td>
<td>XXXXXXXX00100</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>23</td>
<td>XXXXXXXX10100</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>24</td>
<td>XXXXXXXX00010</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>25</td>
<td>XXXXXXXX10010</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>26</td>
<td>XXXXXXXX01010</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>27</td>
<td>XXXXXXXX00001</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>28</td>
<td>XXXXXXXX10001</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>29</td>
<td>XXXXXXXX01001</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>30</td>
<td>XXXXXXXX00101</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>31</td>
<td>XXXXXXXX10101</td>
<td>12</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 8.2: The automaton states for Fibonacci Codes

8.5.3 Automaton States for Fibonacci Codes

Each codeword of the Fibonacci code ends with the $1_m$ delimiter. The number of states is derived from an idea that the 1-bits read from the start of the current segment can complete the $1_m$ delimiter (as well as a codeword) from the previous segment. An example of the Fibonacci code of order 2 is shown in Figure 8.4. The first bit in the current segment is a part of the delimiter; it completes the number 7 partially decoded in the previous segment.

Consequently, it is necessary to have $m$ states including the number of 1-bits in the end of the previous segment. These states and possible values of the previous segment are shown in Table 8.2 for both Fibonacci codes of order 2 and 3. All partially decoded parts without the delimiter of the bit-length $L_{\text{max}} - m - 1$ must be
Figure 8.4: An example of segments where the current segment completes the previous segment for the Fibonacci code of order 2.

remembered in each automaton state. As result, the number of automaton states for the Fibonacci code of order \( m \) after the state similarity reduction is as follows:

\[
N_{\text{states}} = m + 2^{L_{\text{max}} - m - 1} - 1
\]  

Table 8.2: Automaton states and all possible values of the previous segment for the Fibonacci codes of order 2 and 3

<table>
<thead>
<tr>
<th>Previous segment</th>
<th>State</th>
<th>Previous segment</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxxxxxx0</td>
<td>0</td>
<td>xxxxxxxx0</td>
<td>0</td>
</tr>
<tr>
<td>xxxxxxxx1</td>
<td>1</td>
<td>xxxxxxxx1</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>xxxxxxxx11</td>
<td>2</td>
</tr>
</tbody>
</table>

8.6 Automaton Reduction Using Shift Operation

The binary part \((B(n)\) and \(B'(n)\) in the case of the Elias-Fibonacci and Elias-delta codes, respectively) or the reversed Fibonacci representation \((\text{reverse}(F(n)^{(2)}))\) and \((\text{reverse}(F^{(m)}(Q)))\) in the case of the Fibonacci codes of order 2 and 3, respectively) of one codeword can be the longest part of the codeword and it can be stored in more than one segment. This situation is depicted in Figure 8.6 where the standard binary representation \(B(93, 586)\) is stored in three segments. A shift operation reduces states of the automaton related to this part of a codeword. In the case of Elias-Fibonacci and Elias-delta, we apply the binary shift operation; however, in the case of Fibonacci of order 2 and 3, we must apply the Fibonacci left shift operation. The utilization of both operations for the reduction is described in the following sections.

8.6.1 Reduction Using Binary Shift Operation

Let us consider that the standard binary representation \(B(n)\) or \(B'(n)\) of a codeword of \(n\) is stored in more than one segment. Let us remind that the most significant bit of \(B(n)\) or \(B'(n)\) is stored as the first bit. Let us assume that we decode one segment in each step of a decoding algorithm, it means we obtain a part of \(B(n)\)
or $B'(n)$ and this part is stored in $U$ with the bit-length $L_U$. We also assume that we know the bit-length $L(n)$ of $n$ from the prefix of a codeword (see Sections 3.5.2 and 3.7). The number $n$ is decoded by the following procedure:

1. Set $n = 0$ and $len = L(n)$
2. Read the segment and decode a part of $B(n)$ or $B'(n)$ and store a part of $B(n)$ or $B'(n)$ in the variable $U$
3. Compute $len = len - L_U$
4. Compute $n = n + U << len$
5. Finish this procedure if $len = 0$; otherwise continue with Step 2.

![Figure 8.5: The shift operation of the binary part of a codeword to obtain the coded number 613 stored in two segments](image)

**Example 29** (An application of the binary shift operation)

Let us consider the number $n = 613$ encoded by a code like Elias-delta or Elias-Fibonacci. Its standard binary representation $B(613)$ is stored in two segments of the bit-length $S = 8$. The first part of $B(n)$, $B(38)$, is stored in Segment 1. The second part of $B(n)$, $B(5)$, is stored in Segment 2. This situation is depicted in Figure 8.5. The bits depicted by dots in Segment 1 are already decoded, the bits denoted by $X$ in Segment 2 belongs to the next codeword. We assume that the bit-length of $B(n)$ is known in advance, e.g. the bit-length is a part of the codeword prefix in the Elias-delta and Elias-Fibonacci codes. In this case, $len = L(B(613)) = 10$.

When we read and decode Segment 1 in Step 2, we obtain $U = 38$ and $L_U = 6$. In Steps 3 and 4, we compute $len = 10 - 6 = 4$ and $n = 0 + 38 << 4 = 608$. Since $len > 0$ in Step 4, we continue with Step 2. After Segment 2 is read and decoded in Step 2, we obtain $U = 5$, $L_U = 4$. We compute $len = 4 - 4 = 0$ and $n = 608 + 5 << 0 = 613$ in Step 3 and 4. Since $len = 0$, this procedure ends; the encoded number $n = 613$.

In the brute-force approach, we remember a partly decoded number in each state of the automaton, as result, the number of automaton states is very high. The shift-based procedure described above is utilized to reduce the number of automaton states; we use two variables holding the partially decoded number (the $n$ variable) and the number of bits remaining to complete the decoded number (the $len$ variable). During the procedure we distinguish the following groups of states:

80
• \( \text{state} = 0 \) is the initial automaton state

• \( \text{state} \in [1; S] \) are the states related to the shift operation. In these states, we read parts of the binary representation of a codeword.

• \( \text{state} > S \) are other automaton states related to reading the codeword’s prefix.

Algorithm 19 shows a part of a fast Elias delta or Elias-Fibonacci decoding algorithm with a focus on a computation of these states using the shift operation. The complete fast Elias-delta and Elias-Fibonacci decoding algorithm is described later in Section 9.2. In Algorithm 19, we distinguish three steps processing three parts of \( B(n) \) or \( B'(n) \):

• The beginning of \( B(n) \) or \( B'(n) \) is processed in Step 1 of the algorithm in Lines 14–15. The \( \text{len} \) variable holding the remaining bits of \( B(n) \) or \( B'(n) \) is initialized with the value of the mapping table’s attribute \( \text{Rest} \) and \( n \) is set to \( U \) shifted by \( \text{Rest} \).

• The inner part of \( B(n) \) or \( B'(n) \) is processed in Step 2 in Lines 4–6. The inner part of \( B(n) \) or \( B'(n) \) is handled when \( \text{len} \geq S \) meaning that it still remains to read more than 1 segment. The \( \text{len} \) variable is decreased by the segment bit-length \( S \) in Line 4. In Line 5, the inner part is shifted and added to \( n \). Finally, in Line 6, the new state is set according the previously described rules.

• The end of \( B(n) \) or \( B'(n) \) is processed in Step 3 in Lines 10–11. In this case, \( n \) is increased by a decoded part of \( B(n) \) or \( B'(n) \) and, since \( n \) is completed, it is stored in the result array.

Figure 8.6: The standard binary representation \( B(n) \) of \( n = 93,586 \) stored in 3 segments

**Example 30** (The binary shift operation and setting the automaton state in the fast decoding algorithm)

In Figure 8.6, we see \( n = 93,568 \) with the standard binary representation \( B(n) = 10110110110010010 \) stored in 3 segments. In Segment 1, there is the beginning of \( B(n) \) with the value \( B(11) = 1011_2 \). In Segment 2, there is the inner part of \( B(n) \) with the value \( B(108) = 01101100_2 \). Finally, in Segment 3, there is the end of \( B(n) \) with the value \( B(18) = 10010_2 \). These parts are processed
in three steps of Algorithm 19. Segment 1 is processed in Step 1 (Lines 14–15),
\( \text{len} = \text{map.Rest} = 13 \) and \( n = \text{map.U} \ll \text{len} = 11 \ll 13 = 90,112 \). Segment 2 is processed in Step 2 (Lines 4–6), \( \text{len} = \text{len} - S = 13 - 8 = 5 \) and \( n = n + (\text{segment} \ll \text{len}) = 90,112 + (108 \ll 5) = 90,112 + 3,456 = 93,568 \). Since \( \text{len} < S \), we set the automaton state to \( \text{state} = \text{len} = 5 \). Finally, Segment 3 is processed in Step 3 (Lines 10–11). We obtain \( n = \text{Numbers}[0] + n = 18 + 93,568 = 93,586 \) and this value is stored in the result array.

The number of states related to decoding the binary part is dramatically reduced when Algorithm 19 is used. We only need \( S + 1 \) automaton states after the shift operation is applied in Line 5 of the algorithm. As result, the automaton includes only \( S \) of these states for the Elias-delta and Elias-Fibonacci codes and 32 bit-length coded numbers after the second reduction, although the number of these states is \( 2^{32} - 1 \) after the state similarity reduction (see Table 8.3).

The number of states for the Elias-delta and Elias-Fibonacci codes after the shift-based reduction:

\[
N_{\text{states}} = 1 + \left\lfloor \log_2 (L_{\text{max}}) \right\rfloor + L_{\text{max}} - 1 + S \quad (8.6.1)
\]

\[
N_{\text{states}} = 1 + F_{L_{\text{max}}} + L_{\text{max}} - 2 + S \quad (8.6.2)
\]

```plaintext
1 state ← map.NewState;
2 if \( \text{len} \geq S \) then
3     // Step 2: The inner part of B(n) or B'(n)
4     \text{len} ← \text{len} - S;
5     \text{n} ← \text{n} + (\text{segment} \ll \text{len});
6     if \( \text{len} < S \) then \text{state} ← \text{len};
7 else
8     if map.OutputCount > 0 then
9         // Step 3: The end of B(n) or B'(n)
10        \text{n} ← map.Numbers[0] + \text{n};
11        \text{result} ← \text{result} ∪ \text{n};
12     end
13     // Step 1. The beginning of B(n) or B'(n)
14     \text{len} ← map.Rest;
15     \text{n} ← (map.U \ll \text{len});
16 end
```

**Algorithm 19:** A part of a fast Elias delta or Elias-Fibonacci decoding algorithm with a focus on a computation of the automaton states using the shift operation.

### 8.6.2 Reduction Using Fibonacci Left Shift Operation

The binary shift operation can not be applied on the Fibonacci representation; we can not therefore utilize the reduction based on the operation in the case of the
Fibonacci code. In this section, we introduce a reduction based on the Fibonacci left shift operation depicted in Section 6.5.1. The Fibonacci shift manipulates with a Fibonacci representation in the same way as the binary shift manipulates with a standard binary representation.

Whereas we need to decode the Fibonacci representation \(\text{reverse}(F(n)^{(2)})\) for the Fibonacci code of order 2 (see Section 3.6.1), the Fibonacci representation \(\text{reverse}(F^{(m)}(Q))\) is decoded for the Fibonacci codes of higher order (see Section 3.6.2). These Fibonacci representations are located before the delimiter and the bits of the Fibonacci representation are stored in the reverse order; the most significant bit of the Fibonacci representation is stored as the last bit in an input stream. We assume that a Fibonacci representation can be stored in more than one segment. We also assume that one segment is decoded in each step of a decoding algorithm; we obtain a part of the Fibonacci representation of \(n\). This part is stored in \(U\) with the bit-length \(L_U\). The number \(n\) is obtained by the following procedure from the decoded parts:

1. Set \(n = 0\) and \(len = 0\)
2. Read and decode the current segment in the \(U\) variable and set \(L_U\).
3. Calculate \(n = n + U << F\ \text{len}\)
4. Calculate \(len = len + L_U\)
5. As long as the delimiter is not read from the current segment, continue with Step 2.

Since bits of the Fibonacci representation are stored in the reverse order, the bit-length \(len\) of the Fibonacci representation is not known in advance and parts of the Fibonacci representation are consecutively stored in the \(n\) variable. The coded number \(n\) is completed in the time when the delimiter is read. In this way, all states needed to remember the parts of the Fibonacci representation are eliminated. As result, in Table 8.3, we see the number of all automaton states is reduced to 2 and 3 for Fibonacci of order 2 and 3, respectively, although the number of states related to the parts of the Fibonacci representation is \(2^{45} - 1\) and \(2^{35} - 1\), respectively, after the 1st reduction.

The number of states for the Fibonacci codes after the shift-based reduction is as follows:

\[
N_{\text{states}} = m
\]  

(8.6.3)

**Example 31** (An application of the Fibonacci shift operation)

Let us consider a codeword of the number \(n = 226\). Its Fibonacci representation is \(F(n) = 10101001001_{F(2)}\) with the bit-length \(L(F(n)) = 11\). We assume that the Fibonacci representation is stored in two segments (see Figure 8.7). The first part, \(F(27)\), is stored in Segment 1. Let us remember that the bits are stored in the reverse
order and the Fibonacci code of order 2 utilizes the 1-bit delimiter. The bits depicted by dots in Segment 1 belongs to the previous codeword, whereas the bits denoted by X in Segment 2 belongs to the next codeword.

\[
\begin{array}{c}
\text{Segment 1} & \text{Segment 2} \\
\hline
F(226) & F(27) & F(7) \\
\hline
\hline
\end{array}
\]

\[
\begin{array}{c}
.100100101011xxx \\
U=27,L_U=7 \\
27+7 \ll F(7)=27+199=226 \\
\end{array}
\]

\[
\begin{array}{c}
U=7,L_U=4 \\
\end{array}
\]

Figure 8.7: The Fibonacci shift operation of a part of the Fibonacci representation to obtain a coded number stored in two segments.

After we read and decode Segment 1, we obtain \( U = 27 \) and \( L_U = 7 \) in Step 2 of the above procedure. Consequently, we compute \( n = 0 + 27 \ll F(0) = 27 \) and \( \text{len} = 7 \) in Steps 3 and 4. Since the delimiter is not reached in Step 5, we continue with Step 2. After we read and decode Segment 2, we obtain \( U = 7 \) and \( L_U = 4 \). We calculate \( n = 27 + 7 \ll F(7) = 27 + 1010_{F(2)} \ll F(7) = 27 + 1010_{F(2)} + F_5 \times (1010_{F(2)} \gg F(1)) = 27 + 21 \times 7 + 13 \times 4 = 27 + 199 = 226 \) and \( \text{len} = 11 \) (the Fibonacci left shift is computed by means of Theorem 3). Since the delimiter is reached in Step 5, the number is completely decoded.

```
1 if map.OutputCount > 0 then
2    // Step 3: The end of F(n)
3        n <- n + map.Numbers[0] \ll F(len);
4        result <- result \cup n;
5    // Step 1. The beginning of F(n)
6        n <- 0;
7        len <- 0;
8 end
9 // Step 2. The beginning and inner part of F(n)
10 if map.L_U > 0 then
11    n <- n + (map.U \ll F(len));
12    len <- len + map.L_U;
13 end
```

Algorithm 20: A part of a fast Fibonacci decoding algorithm with a focus on a computation of the Fibonacci shift operation

Following the procedure described above, a part of a fast Fibonacci decoding algorithm with a focus on a computation of the Fibonacci shift operation is shown in Algorithm 20. The complete fast Fibonacci decoding algorithm is described later in Section 9.2. In Algorithm 20, we distinguish three steps processing three parts of \( F(n) \):
General Principles of Fast Decoding Algorithms

- The beginning of $F(n)$ is processed in Step 1 and Step 2 of Algorithm 20 in Lines 6–13. The len variable holding the number of the remaining bits of $F(n)$ and $n$ are initialized to 0 in Lines 6–7. Since $len = 0$ in the case of the beginning of $F(n)$, the Fibonacci left shift operation is not applied in Line 11 and $n = U$. The len variable is increased by the bit-length $L_U$ in Line 12.

- The inner part of $F(n)$ is processed in Step 1 in Lines 10–13. The decoded inner part of $F(n)$ is shifted by len and it is gradually added to $n$ in Line 11. The len variable is increased by the bit-length $L_U$ in Line 12.

- The end of $F(n)$ is processed in Step 3 in Lines 3–4. In this case, $n$ is increased by the shifted decoded part of $F(n)$, and, since $n$ is completely decoded, it is stored in the result array.

\[
\begin{array}{c|c|c}
\text{Segment 1} & \text{Segment 2} & \text{Segment 3} \\
\hline
. . . 1 0 1 0 0 1 0 0 1 0 0 1 0 0 0 1 x x & F(4)=U.L=4 & F(10)=U.L=8 \\
F(3,033) & F(9)=U.L=5 \\
\end{array}
\]

Figure 8.8: The Fibonacci representation $F(n = 3,003_{10})$ stored in 3 segments

Example 32 (The Fibonacci shift operation in the fast decoding algorithm)  
In Figure 8.8, we see the number $n = 3,003_{10}$ as the Fibonacci representation $F(n) = 1001000100100101_F$ stored in 3 segments. Segment 1 includes the beginning of $F(n)$: the value $F(4) = 0101_F$. Segment 2 includes the inner part of $F(n)$: $F(10) = 00010010_F$. Finally, Segment 3 includes the end of $F(n)$: $F(9) = 10001_F$. These parts are processed with three steps in Algorithm 20. Segment 1 is processed in Step 1 and Step 2. The variables are initialized in Step 1 (Lines 6–7) to len = 0 and $n = 0$. In Step 2 (Lines 10–13), we compute $n = n + (\text{map}.U <<_F \text{len}) = 0 + 4 <<_F 0 = 4$ and len = len + $L_U = 0 + 4 = 4$. Segment 2 is processed in Step 2 (Lines 10–13): $n = n + (\text{map}.U <<_F \text{len}) = 4 + 10 <<_F 4 = 4 + 68 = 72$ and len = len + $L_U = 4 + 8 = 12$. Finally, Segment 3 is processed in Step 3 (Lines 3–4); $n = n + (\text{map}.\text{Numbers}[0] <<_F \text{len}) = 72 + (9 <<_F 12) = 72 + 2,961 = 3,033_{10}$. The value $n = 3,033_{10}$ is stored in the result array.

8.7 Number of States After Reductions

In Table 8.3, we see the result of both reductions, where '1st reduction' means the state similarity reduction and '2nd reduction' means the shift-based reduction. Since the number of states for the brute-force algorithm depends on the maximal bit-length $L_{\text{max}}$ of a codeword (see Equation 8.3.1), this table also includes this value. The number of states after the 1st reduction is computed using Equations 8.5.2, 8.5.4,
and 8.5.5. The number of states after the 2\textsuperscript{nd} reduction is computed using Equations 8.6.1, 8.6.2, and 8.6.3. Let us remember that the size of the mapping table is always $N_{\text{states}} \times 2^S$ by means of Equation 8.2.1. Evidently, the shift-based reduction extremely decreases the number of automaton states.

Table 8.3: The number of automaton states after the 1\textsuperscript{st} (utilizing a similarity of states) and the 2\textsuperscript{nd} reduction (utilizing a shift operation) for 32 bit-length numbers and the segment bit-length $S$.

<table>
<thead>
<tr>
<th>$N_{\text{states}}$</th>
<th>Elias-delta</th>
<th>Fibonacci $m = 2$</th>
<th>Fibonacci $m = 3$</th>
<th>Elias-Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_{\text{max}}$</td>
<td>41</td>
<td>46</td>
<td>37</td>
<td>39</td>
</tr>
<tr>
<td>1\textsuperscript{st} reduction</td>
<td>$2^{41} - 1$</td>
<td>$2^{46} - 1$</td>
<td>$2^{37} - 1$</td>
<td>$2^{39} - 1$</td>
</tr>
<tr>
<td>2\textsuperscript{nd} reduction</td>
<td>$37 + 2^{31} - 1$</td>
<td>$2 + 2^{45} - 1$</td>
<td>$3 + 2^{35} - 1$</td>
<td>$65 + 2^{32} - 1$</td>
</tr>
<tr>
<td>$37 + S$</td>
<td>2</td>
<td>3</td>
<td>$65 + S$</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 9

Fast Decoding Algorithms

9.1 Outlines

In this chapter, all fast decoding algorithms are described. The fast decoding algorithm for the Elias-delta and Elias-Fibonacci codes is put forward in Section 9.2. In Section 9.3, the fast decoding algorithm for the Fibonacci codes of order 2 and 3 is introduced. In Section 9.4, we compare our work with other works.

9.2 Fast Elias-delta and Elias-Fibonacci Decoding Algorithms

The fast decoding algorithms for the Elias-delta and Elias-Fibonacci codes are identical; however, the mapping tables are different since the conventional bit-oriented algorithms as well as automatons are different. This algorithm is shown in Algorithm 21.

In Lines 1–3, variables for the state, the current decoded number $n$, and the bit-length $len$ of the current decoded number are initialized. If $state < 0$, it indicates an error and the algorithm ends in Line 5. In Lines 6–7, the next segment is read from the input stream and it is decoded by means of the mapping table $MAP$. The new state is set in Line 8. Lines 10–16 cover a situation when the current decoded number is not finished in the actual segment. In other words, the shift operation is used if the number of remaining bits $len \geq S$. In Line 10, we compute the number of the remaining bits in the next segment $len = len - S$ which also defines the shift operation in Line 11. In Line 12, we decide whether there are some remaining bits of the codeword in next segment(s) after the shift operation.

After the shift operation, we output the number $n$ in the result array in Lines 13–14. In Line 16, we set the correct automaton state after the shift operation. Lines 18–25 cover the situation when previously decoded bits are finished in the actual segment. In Lines 18–21, output decoded numbers are written in the result; in one segment, the algorithm can output more then one decoded number. Since the first
Fast Decoding Algorithms

Algorithm 21: The fast Elias-delta and Elias-Fibonacci decoding algorithm

| input : stream including codewords of the Elias-delta or Elias-Fibonacci code, |
| MAP – Elias-delta or Elias-Fibonacci mapping table |
| output: result – an array of decoded numbers |

1. \( n \leftarrow 0; \)
2. \( \text{state} \leftarrow 0; \)
3. \( \text{len} \leftarrow 0; \)
4. while NOT Eos(stream) do
   5.     if state < 0 then break;
   6.     segment ← GetBits(stream, S);
   7.     map ← MAP[state, segment];
   8.     state ← map.NewState;
   9.     if len ≥ S then
      10.    len ← len − S;
      11.    \( n \leftarrow n + \text{(segment} \ll \text{len);} \)
      12.    if len = 0 then
      13.         result ← result \cup n;
      14.         \( n \leftarrow 0; \)
      15.      end
   16.     if len < S then state ← len;
   17.     else
      18.         if map.OutputCount > 0 then
      19.             \( n \leftarrow \text{map.} \text{Numbers}[0] + n; \)
      20.             result ← result \cup n;
      21.             result ← result \bigcup_{i=1}^{\text{map.} \text{OutputCount}-1} \text{map.} \text{Numbers}[i];
      22.         end
      23.         \( \text{len} \leftarrow \text{map.} \text{Rest}; \)
      24.         \( n \leftarrow \text{map.} \text{U} \ll \text{len}; \)
      25.     end
   26. end

decoded number can complete bits decoded in the previous segment by the shift operation in Lines 10–16, it is separately processed in Lines 19–20.

The rest of the numbers is written in the result array in Line 21. In Lines 23, len is initialized by the Rest attribute of the mapping table. It means that the remaining bits of the currently decoded number \( n \) are stored in the next segment(s). In Line 24, \( n \) is initialized with \( U \) and it is shifted by the number of the remaining bits len.

Example 33 (The fast Elias-delta decoding algorithm)
Let us consider an example of fast Elias-delta decoding in Figure 9.1. Rows of MAP accessed in this example are shown in Table 9.1.

After the first segment is read, we access the mapping table for state = 0 and segment = 104 in Lines 6–7 of the algorithm. The new state of the automaton is set to 23 in Line 8. This state is one of the zero states (see Section 8.5.1), it represents 3 zeros of the Elias-delta prefix. These bits belong to the codeword in next segment(s). Since len = 0, in Lines 18–22, we obtain the number 5 directly from the mapping table (see the Numbers attribute of MAP) and it is stored in the result array. In Lines 23–24, we initialize len = 0 and \( n = 0 \). Now, the second
Figure 9.1: An example of the fast Elias-delta decoding algorithm for $S = 8$

Table 9.1: Some rows of the mapping table for the fast Elias-delta decoding algorithm (m: and $S = 8$)

<table>
<thead>
<tr>
<th>State</th>
<th>Segment</th>
<th>NewState</th>
<th>Rest</th>
<th>$U$</th>
<th>OutputCount</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>104</td>
<td>23</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>{5}</td>
</tr>
<tr>
<td>3</td>
<td>139</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>{4, 3, 1}</td>
</tr>
<tr>
<td>23</td>
<td>128</td>
<td>3</td>
<td>3</td>
<td>16</td>
<td>0</td>
<td>{}</td>
</tr>
</tbody>
</table>

segment with the value 128 is read. In the mapping table, we access a row of MAP for state = 23 and segment = 128. The state 3 is set in Line 8. This state is one of the shift states (see Section 8.6.1), it indicates that there are 3 bits in the next segment to complete the codeword (see the Rest attribute of MAP). There are not any output numbers for this segment. We set $len = 3$ and $n = 16 << 3 = 128$ in Lines 24–25. The next segment with the value 139 starts with the remaining 3 bits; the sequence 1002 is decoded. We obtain the number 4 (see the Numbers attribute of MAP) which is added to $n$: $n = 128 + 4 = 132$ in Line 19. This value is stored in the result array with other two numbers 3 and 1 of Numbers in Lines 20–21.

Let us consider another example of fast Elias-delta decoding in Figure 9.2. Rows of MAP accessed in this example are shown in Table 9.2.

Figure 9.2: An example of the fast Elias-delta decoding algorithm for $S = 8$

After the first segment is read, we access the mapping table for state = 0 and segment = 25 in Lines 6–7 of Algorithm 21. The new state of the automaton
Table 9.2: Some rows of the mapping table for the fast Elias-delta decoding algorithm and $S = 8$

<table>
<thead>
<tr>
<th>State</th>
<th>Segment</th>
<th>NewState</th>
<th>Rest</th>
<th>$U$</th>
<th>OutputCount</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
<td>8</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>{}</td>
</tr>
<tr>
<td>2</td>
<td>133</td>
<td>37</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>{2}</td>
</tr>
<tr>
<td>8</td>
<td>115</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>{}</td>
</tr>
</tbody>
</table>

is set to 8 in Line 8. This state means that the shift operation is applied during the decoding of the next segment (see Section 8.6.1). In Lines 23–24, we initialize $\text{len} = 10$ and $n = 3 \ll 10 = 3,072$ (see the Rest and $U$ attributes of MAP). Now, the second segment with the value 115 is read. We apply the shift operation in Lines 10–16 since $\text{len} \geq S$ in Line 9. We compute $\text{len} = 10 - 8 = 2$ in Line 10 and $n = 3,072 + (115 \ll 2) = 3,532$ in Line 11. The next state is set to 2 in Line 16. The next segment with the value 133 starts with the remaining 2 bits; the sequence $10_2$ is decoded. We obtain the number 2 (see the Numbers attribute of MAP) which is added to $n$, $n = 3,532 + 2 = 3,534$, in Line 19. This value is stored in the result array and the new state is set to 37 in Line 8. This state is one of the bit-length states since the sequence 000101 is a partially decoded bit-length (see Section 8.5.1 for more details).

Example 34 (The fast Elias-Fibonacci decoding algorithm)

Let us consider an example of fast Elias-Fibonacci decoding in Figure 9.3. Rows of the mapping table accessed in this example are shown in Table 9.3.

<table>
<thead>
<tr>
<th>State</th>
<th>Segment</th>
<th>NewState</th>
<th>Rest</th>
<th>$U$</th>
<th>OutputCount</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>58</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>{6}</td>
</tr>
<tr>
<td>4</td>
<td>87</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>{5,3}</td>
</tr>
<tr>
<td>11</td>
<td>59</td>
<td>4</td>
<td>4</td>
<td>27</td>
<td>0</td>
<td>{}</td>
</tr>
</tbody>
</table>

Figure 9.3: An example of the fast Elias-Fibonacci decoding algorithm for $S = 8$

Table 9.3: Some rows of the mapping table for the fast Elias-Fibonacci decoding algorithm and $S = 8$

<table>
<thead>
<tr>
<th>State</th>
<th>Segment</th>
<th>NewState</th>
<th>Rest</th>
<th>$U$</th>
<th>OutputCount</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>58</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>{6}</td>
</tr>
<tr>
<td>4</td>
<td>87</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>{5,3}</td>
</tr>
<tr>
<td>11</td>
<td>59</td>
<td>4</td>
<td>4</td>
<td>27</td>
<td>0</td>
<td>{}</td>
</tr>
</tbody>
</table>

After the first segment is read, we access a row for segment = 58 and state = 0 of the mapping table in Lines 6–7. The next automaton state is set to 11 in Line 8. This state is one of the Elias-Fibonacci prefix states, it represents the partially decoded
prefix $10_2$ (see Section 8.5.2). This sequence is a part of the reverse Fibonacci representation of $L(437)$; $\text{reverse}(F(L(437))) = 9$. In the mapping table, we directly read the number 6 and it is stored in result in Line 20 of the algorithm. We set $n = 0$ and $len = 0$ in Lines 23–24. Now, the second segment with the value 59 is read and we access a row for segment = 59 and state = 11 of the mapping table in Lines 6–7. We set the new state to 4 in Line 8. This state is one of the shift states (see Section 8.6.1), it indicates that there are 4 bits in the next segment to complete the codeword. We set $len = 4$ and compute the incomplete number $n = 27 << 4 = 432$ in Lines 23–24. The next segment starts with the remaining 4 bits; the sequence $0101_2$ is decoded. In this state, we obtain the numbers 5 and 3 in the mapping table. The first number is added to $n$; we compute $n = 432 + 5 = 437$ in Line 19. The number 437 is stored in result in Lines 20–21 together with the second number 3.

### 9.3 Fast Fibonacci Decoding Algorithm

The fast Fibonacci decoding algorithm is the same for any order $m$ of the Fibonacci code. This algorithm is shown in Algorithm 22, it differs for various orders in the following parts:

- **Mapping table**: The mapping table is based on a different bit-oriented algorithm.
- **Fibonacci left shift operation**: The Fibonacci left shift operations are different for different orders (see Section 8.6.2).
- **Final adjustment**: $S_n + 1$ is added in the case of Fibonacci codes of order $m > 2$.

In Lines 1–3, the variables for `state`, the current output number $n$, and its bit-length $len$ are initialized. If $state < 0$, it indicates an error; therefore, the algorithm ends in Line 5. In Lines 6–7, the next segment is read from the input stream and it is decoded by means of the mapping table $MAP$. The new state is set in Line 8. If the codeword is finished, it is output in Lines 9–22. If the codeword’s bit-length $len > 0$, we need to use the Fibonacci left shift; it is applied in Line 11. In Lines 15–18, the Fibonacci sum is added to the decoded number for orders $m > 2$. To select the correct Fibonacci sum, in Line 16, the $k$ value is computed using the remaining bits of the codeword in the actual segment. We also subtract the $m$ value in this line since the codeword’s bit-length must be $m + k$. The result decoded number is add in the `result` array in Line 19. The $n$ and $len$ variables are set in Lines 20–21. The Fibonacci left shift is carried out in Line 24. In Line 25, the bit-length $len$ of the Fibonacci representation of the output number is increased by the bit-length $map.L_U$ of the decoded part.

**Example 35** (The fast decoding algorithm for the Fibonacci code of order 2) 
Let us consider codewords in Figure 9.4. The rows of the mapping table accessed in this example are shown in Table 9.4.
Algorithm 22: The fast Fibonacci decoding algorithm for Fibonacci codes of order $m$

After we read the first segment, we access a row for $state = 0$ and $segment = 181$ of the mapping table in Lines 6–7 of Algorithm 22. The new state of the automaton is set to 1 in Line 8. It represents reading the 1-bit at the end of the segment (see Section 8.5.3). We directly write the number 4 in the $result$ array in Line 19. We do not apply any shift in Lines 10–14 since $len = 0$. The incomplete number $U = 7$ is stored in the $n$ variable holding each incomplete number from the previous segment (Line 24). The bit-length $len$ is set to 4. We continue with the reading of the second segment. Since it starts with the 1-bit (the delimiter), the codeword is complete. We add the number 0 read in $Numbers$ to the output number $n$ in Line 11; $n = n + U << F len = 4 + 0 << F 4 = 4 + 0 << F 4 = 4$. Consequently, the

Figure 9.4: An example of the fast Fibonacci of order 2 decoding algorithm for $S = 8$
number 4 is stored in the result array. We set \( n = U = 31 \) and \( len = L_U = 7 \) in Lines 24–25. The next segment starts with the sequence 011 = \( F(2) \); it completes the previous segment: \( n = n + U <<_F len = 31 + (2 <<_F 7) = 31 + 55 = 86 \). The number 86 is stored in the result array and we set \( n = U = 6 \) and \( len = L_U = 5 \). The \( n \) value is completed in the next segment(s).

Table 9.4: Some rows of the mapping table for the fast Fibonacci of order 2 decoding algorithm and \( S = 8 \)

<table>
<thead>
<tr>
<th>State</th>
<th>Segment</th>
<th>NewState</th>
<th>( U )</th>
<th>( L_U )</th>
<th>OutputCount</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>181</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>{4}</td>
</tr>
<tr>
<td>1</td>
<td>165</td>
<td>1</td>
<td>31</td>
<td>7</td>
<td>1</td>
<td>{0}</td>
</tr>
<tr>
<td>1</td>
<td>114</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>{2}</td>
</tr>
</tbody>
</table>

Example 36 (The fast decoding algorithm for the Fibonacci code of order 3)  
Let us consider codewords in Figure 9.5. The rows of the mapping table accessed in this example are shown in Table 9.5.

Figure 9.5: An example of the fast Fibonacci of order 3 decoding algorithm for \( S = 8 \)

After we read the first segment, we access a row for \( state = 0 \) and \( segment = 139 \) of the mapping table in Lines 6–7 of the algorithm. In Line 24, the incomplete number \( U = 14 \) is stored in the \( n \) variable holding each incomplete number from the previous segment. Its bit-length is set to \( len = 6 \). The new state of the automaton is set to 2 in Line 8. It represents reading the sequence 112 at the end of the segment (see Section 8.5.3). We read the second segment and we access a row for \( state = 2 \) and \( segment = 28 \) of the mapping table in Lines 6–7. Since \( len > 0 \), we use the Fibonacci left shift in Line 11: \( n = 14 + 3 <<_F 6 = 14 + 125 = 139 \). In Line 16, we compute \( k = 6 + 8 - 3 = 11 \). In Line 17, we compute the output number: \( n = n + S_{k-2} + 1 = 139 + S_9 + 1 = 139 + 600 + 1 = 740 \). The number 740 is stored in the result array in Line 19. We set \( n = U = 0 \) and \( len = L_U = 2 \) in Lines 24–25. The \( n \) value is completed in the next segment(s).

9.4 Comparison with Other Works

A fast decoding algorithm for the Fibonacci code proposed by Shmuel T. Klein et al. in [40, 41, 42] also utilizes a kind of the mapping table for each decoded segment.
Fast Decoding Algorithms

Table 9.5: Some rows of the mapping table for the fast Fibonacci of order 3 decoding algorithm and $S = 8$

<table>
<thead>
<tr>
<th>State</th>
<th>Segment</th>
<th>NewState</th>
<th>$U$</th>
<th>$L_U$</th>
<th>OutputCount</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>139</td>
<td>2</td>
<td>14</td>
<td>6</td>
<td>0</td>
<td>{}</td>
</tr>
<tr>
<td>2</td>
<td>28</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>{3}</td>
</tr>
</tbody>
</table>

as well as the automaton reduction using a similarity of states. They introduce a similar operation to the Fibonacci left shift called shifted Fibonacci numbers. Author proposed two approaches for the computation of the Fibonacci left shift. In the first approach, the Fibonacci left shift is calculated utilizing properties of the Fibonacci code. Author utilizes float numbers for the computation and this issue results in a slower computation. Therefore, the second approach gets rid of float numbers. In this case, the Fibonacci left shift is precomputed for each $k$-shift. The size of their mapping table depends on the codeword bit-length and the segment bit-length. In Table 9.6, we see a comparison of our fast algorithms with the work of Klein et al. for the Fibonacci codes of order 2 and 3. In the case of the work of Klein et al., the table sizes $S_{MAP}$ are $2 \times 2^8 \times L_{\text{max}} = 512 \times L_{\text{max}}$ and $3 \times 2^8 \times L_{\text{max}} = 768 \times L_{\text{max}}$ for order $m = 2$ and $m = 3$, respectively, for $S = 8$. For $S = 16$, the table sizes are $2 \times 2^{16} \times L_{\text{max}} = 131,072 \times L_{\text{max}}$ and $3 \times 2^{16} \times L_{\text{max}} = 196,608 \times L_{\text{max}}$ for order $m = 2$ and $m = 3$, respectively. In the case of our fast algorithms, the table sizes are $2 \times 2^8 = 512$ and $3 \times 2^8 = 768$ for order $m = 2$ and $m = 3$, respectively, for $S = 8$. For $S = 16$, the table sizes are $2 \times 2^{16} = 131,072$ and $3 \times 2^{16} = 196,608$ for order $m = 2$ and $m = 3$, respectively. Evidently, our fast Fibonacci algorithms are not affected by the bit-length of coded numbers compared to the work of Klein et al.

Table 9.6: A comparison of our fast algorithms (the abbreviation FA) with the work of Klein et al. (the abbreviation Kl) for the Fibonacci codes of order 2 and 3

<table>
<thead>
<tr>
<th></th>
<th>FA</th>
<th>FA</th>
<th>Kl</th>
<th>Kl</th>
<th>FA</th>
<th>Kl</th>
<th>FA</th>
<th>Kl</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_n$ [bit]</td>
<td>16</td>
<td>32</td>
<td>16</td>
<td>32</td>
<td>16</td>
<td>32</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>$L_{\text{max}}$ [bit]</td>
<td>24</td>
<td>39</td>
<td>21</td>
<td>37</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_{MAP}$ S = 8</td>
<td>512</td>
<td>12,288</td>
<td>512</td>
<td>19,968</td>
<td>768</td>
<td>16,128</td>
<td>768</td>
<td>28,416</td>
</tr>
<tr>
<td>$S_{MAP}$ S = 16</td>
<td>131,072</td>
<td>3,145,728</td>
<td>131,072</td>
<td>5,111,808</td>
<td>196,608</td>
<td>4,128,768</td>
<td>196,608</td>
<td>7,274,496</td>
</tr>
</tbody>
</table>

Although the work of Klein et al. provides the same time complexity as our fast algorithms, the mapping table size is bigger. Their work is related to text compression; therefore, the mapping table size is sufficient for lower bit-lengths of coded numbers. However, our approach provides better scalability, due to the fact the table size is smaller even for higher coded numbers. Moreover, we introduce fast decoding algorithms for the Elias-delta and Elias-Fibonacci codes.
Part IV

Experiments and Applications
Chapter 10

Experimental Results

10.1 Test Collections

In our experiments, we tested all proposed fast encoding and decoding algorithms and compared these algorithms with conventional bit-oriented algorithms\(^1\). The experiments were executed on a PC with Intel Xeon 2.93 GHz, 32 GB RAM; Windows Server 2008 x64. In this section, we use abbreviations ED, EF, F2, and F3 for Elias-delta, Elias-Fibonacci, Fibonacci of order 2, and Fibonacci of order 3, respectively.

![Graph showing probability density functions](image)

Figure 10.1: The uniform distribution, normal distribution for \(\sigma^2 = 100\), and exponential distribution for \(\lambda^2 = 100\)

The test collections used in the experiments included 10,000,000 randomly generated numbers; we utilized uniform, normal, and exponential distribution functions.

\(^1\)The fast coding library can be downloaded from [http://db.cs.vsb.cz/SubPages/Projects/Projects.aspx](http://db.cs.vsb.cz/SubPages/Projects/Projects.aspx)
for generating the random numbers (see Figure 10.1). Tests with real data collections are proposed in Chapter 11. The proposed encoding/decoding algorithms are universal and they can be applied for arbitrary numbers > 0. In our experiments, we worked with numbers ≤ 4,294,967,295, it means the maximal value was the maximal value of a 32 bit-length number. Tested collections were as follows:

- 8, 16, 24, 32-bit – collections of uniformly distributed random numbers ranging from 1 to 255, 256 to 65,535, 65,536 to 16,777,215, and 16,777,216 to 4,294,967,295, respectively.
- Uniform – a collection of uniformly distributed random numbers ranging from 1 to 4,294,967,295
- Normal – a collection of normally distributed random numbers ranging from 1 to 4,294,967,295; we used the normal distribution function \( f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \), where \( \sigma^2 = 2^{32} \).
- Exponential – a collection of exponentially distributed random numbers ranging from 1 to 4,294,967,295; we used the exponential distribution function \( f(x) = \frac{1}{\lambda} e^{-\frac{x}{\lambda}} \), where \( \lambda^2 = 2^{32} \).

### 10.2 Compression Ratio

Although studying the compression ratio of variable-length codes is beyond the scope of this work, in this section, we show the compression ratio, since it is one of the parameters of each code (together with encoding and decoding times). As a baseline, we selected the fixed-length standard binary representation. The size of encoded data for Normal, Exponential, and Uniform 32-bit collections is shown in Table 10.1 and Figure 10.2.

<table>
<thead>
<tr>
<th>Test Collection</th>
<th>Standard binary</th>
<th>Elias-delta</th>
<th>Fibonacci 2</th>
<th>Fibonacci 3</th>
<th>Elias-Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform</td>
<td>38.15</td>
<td>47.69</td>
<td>53.88</td>
<td>44.99</td>
<td>45.31</td>
</tr>
<tr>
<td>Exponential</td>
<td>38.15</td>
<td>27.27</td>
<td>27.44</td>
<td>24.15</td>
<td>25.76</td>
</tr>
<tr>
<td>Normal</td>
<td>38.15</td>
<td>27.20</td>
<td>27.30</td>
<td>24.03</td>
<td>25.67</td>
</tr>
</tbody>
</table>

We see that the variable-length codes provide the higher compression ratio for all test collections with an exception of the Uniform collection; however, the encoding of a collection with the uniform distribution is the worst case scenario for each compression method. Evidently, the Fibonacci code of order 3 provides the best compression ratio; however, results of the Elias-Fibonacci code are very similar.
Experimental Results

10.3 Encoding Algorithms

In Table 10.2, we show a comparison of encoding times for both fast Fibonacci encoding algorithms using 8-bit and 16-bit segments. We see that the utilization of the 16-bit segment brings the $1.6\times$ more efficient encoding time than in the case of the 8-bit segment. Therefore, we used only the 16-bit segment in other experiments.

Table 10.2: A comparison of encoding times for both Fibonacci fast encoding algorithms using 8-bit and 16-bit segments [ms]

<table>
<thead>
<tr>
<th>Test Collection</th>
<th>Fibonacci of order 2</th>
<th>Fibonacci of order 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8-bit seg.</td>
<td>16-bit seg.</td>
</tr>
<tr>
<td>8-bit</td>
<td>192</td>
<td>87</td>
</tr>
<tr>
<td>16-bit</td>
<td>294</td>
<td>183</td>
</tr>
<tr>
<td>24-bit</td>
<td>457</td>
<td>286</td>
</tr>
<tr>
<td>32-bit</td>
<td>551</td>
<td>317</td>
</tr>
<tr>
<td>Uniform</td>
<td>373</td>
<td>318</td>
</tr>
<tr>
<td>Exponential</td>
<td>319</td>
<td>184</td>
</tr>
<tr>
<td>Normal</td>
<td>308</td>
<td>181</td>
</tr>
<tr>
<td><strong>Avg.</strong></td>
<td><strong>356</strong></td>
<td><strong>222</strong></td>
</tr>
</tbody>
</table>

In Table 10.3 and Figure 10.3, the encoding times of all conventional and fast encoding algorithms are shown. We see that the Elias-delta and Elias-Fibonacci conventional encoding algorithms outperform both Fibonacci conventional encoding algorithms. Fast algorithms of both Fibonacci codes achieved faster encoding times for the 8-bit test collection than other fast algorithms (and the fast algorithm of Fibonacci of order 3 outperforms the fast algorithm of Fibonacci of order 2). On the other hand, when we consider a test collection with the larger domain, fast encoding of the Elias-Fibonacci code is shown to be the most efficient. Whereas the processing time of the fast Elias-delta encoding algorithm is close to the fast Elias-Fibonacci algorithm, the fast encoding algorithm of the Fibonacci code of order 3 is less efficient, the encoding time of the fast algorithm for the Fibonacci code of order 2 is between Elias-delta and Fibonacci of order 3.
Experimental Results

Table 10.3: Encoding time [ms] and Speed-up ratio [times]

<table>
<thead>
<tr>
<th>Test Collection</th>
<th>Conventional ED F2 F3 EF</th>
<th>Fast ED F2 F3 EF</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit</td>
<td>605 1.305 1.404 1.138</td>
<td>111 87 79 94</td>
</tr>
<tr>
<td>16-bit</td>
<td>962 2.198 2.191 1.506</td>
<td>127 183 334 97</td>
</tr>
<tr>
<td>24-bit</td>
<td>1,217 3.016 2.926 1.811</td>
<td>145 286 377 121</td>
</tr>
<tr>
<td>32-bit</td>
<td>1,598 3.901 3.588 2.025</td>
<td>173 317 579 150</td>
</tr>
<tr>
<td>Uniform</td>
<td>1,629 3.901 3.590 2.030</td>
<td>174 318 579 151</td>
</tr>
<tr>
<td>Exponential</td>
<td>1,058 2.269 2.253 1.579</td>
<td>204 184 332 163</td>
</tr>
<tr>
<td>Normal</td>
<td>1,037 2.253 2.239 1.560</td>
<td>202 181 330 159</td>
</tr>
<tr>
<td>Avg.</td>
<td>1,158 2.692 2.599 1.664</td>
<td>162 222 373 134</td>
</tr>
<tr>
<td>Speed-up ratio</td>
<td></td>
<td>7.1× 12.1× 7.0× 12.5×</td>
</tr>
</tbody>
</table>

Figure 10.3: Encoding times for conventional and fast algorithms

The average speed-up ratio for all test collections is shown in the last line of Table 10.3 and in Figure 10.4. All fast encoding algorithms achieved faster encoding times than the conventional algorithms, the speed-up ratio ranges from 7.0× for the Fibonacci code of order 3 to 12.5× for the Elias-Fibonacci code. The speed-up ratios for the Elias-Fibonacci and Fibonacci of order 2 codes are approximately 2× higher than speed-up ratios for the Elias-delta and Fibonacci of order 3 codes.

Figure 10.4: Average speed-up ratio for all fast encoding algorithms

Whereas the encoding time for both Fibonacci codes goes up linearly with the domain size (for both conventional and fast algorithms), the encoding time for the Elias-delta and Elias-Fibonacci codes goes up linearly for conventional algorithms and less than linearly for the fast algorithms. The reason is in the implementation of the \( L(n) \) function; we utilized three techniques: the while-loop cycle, BSR instruction, and LogTable256 (see Chapter 4). The later two implementations are not linear for the increasing number of bits. The BSR instruction has the following
Experimental Results

feature on the Intel Dual-Core CPU used in our experiments: the bigger number means the lower count of CPU clock cycles and vice versa. To achieve the best performance, we compared these implementations. Encoding times for the Elias-delta code and these implementations of \( L(n) \) are depicted in Table 10.4. As result, we see that the BSR instruction is more efficient than while loop, but slower than LogTable256. The number of CPU clock cycles for the BSR instruction is shown in Table 10.5, where the \( c \) value is the order of the highest 1-bit. Therefore, we use LogTable256 in all tests.

Table 10.4: A comparison of three implementations of \( L(n) \) in the fast Elias-delta algorithm for the Uniform collection: the while-loop cycle, BSR instruction, and Log256Table [ms]

<table>
<thead>
<tr>
<th>Test Collection</th>
<th>While loop</th>
<th>BSR</th>
<th>Log256 Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit</td>
<td>303</td>
<td>179</td>
<td>174</td>
</tr>
</tbody>
</table>

Table 10.5: The number of CPU clock cycles for the BSR instruction of all Intel Pentium Core 2 CPUs

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CPU Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSR reg16,reg16</td>
<td>7 + 2c</td>
</tr>
<tr>
<td>BSR reg32,reg32</td>
<td>7 + 2c</td>
</tr>
</tbody>
</table>

10.4 Decoding Algorithms

In the case of the fast encoding algorithms, we utilize the 16-bit segment. In the case of the fast decoding algorithms, the following sizes of MAP tables are computed for 32 bit-length numbers and the 16-bit segment: 3 473 408, 4 849 664, 131 072, and 196 608 for fast Elias-delta, Elias-Fibonacci, and Fibonacci of order 2 and 3 decoding algorithms, respectively (see Equations 8.2.1, 8.6.1, 8.6.2, and 8.6.3). Because 16-bit segments produce rather large tables, we utilize the 8-bit segments for all decoding algorithms in our experiment. In this case, the mapping table size is \( 45 \times 2^8 = 11 520 \), \( 73 \times 2^8 = 18 688 \), \( 2 \times 2^8 = 512 \), and \( 3 \times 2^8 = 768 \) for Elias-delta, Elias-Fibonacci, and Fibonacci of order 2 and 3, respectively. In Table 10.6 and Figure 10.5, the decoding times of all conventional and fast algorithms are shown.

Although the most efficient compression ratio is achieved by the Fibonacci code of order 3 (see Table 10.1), the fast decoding algorithm of the Elias-Fibonacci code outperforms all other fast decoding algorithms from the processing time point of view. The fast Fibonacci decoding algorithms of both orders utilize the Fibonacci shift operation, which is more time consuming compared to a common shift operation applied in the case of the fast Elias-delta and Elias-Fibonacci decoding algorithms. The Fibonacci code of order 3 provides the less efficient decoding time than the Fibonacci code of order 2 in all tests, because this algorithm includes more operations.

\(^{2}\)The same feature can be seen in all Intel and AMD CPUs up to this date [25, 34].
Table 10.6: Decoding time [ms] and Speed-up ratio [times] for all conventional and fast algorithms

<table>
<thead>
<tr>
<th>Test Collection</th>
<th>Conventional</th>
<th></th>
<th></th>
<th>Fast</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ED</td>
<td>F2</td>
<td>F3</td>
<td>EF</td>
<td>ED</td>
<td>F2</td>
<td>F3</td>
</tr>
<tr>
<td>8-bit</td>
<td>357</td>
<td>699</td>
<td>885</td>
<td>539</td>
<td>138</td>
<td>128</td>
<td>137</td>
</tr>
<tr>
<td>16-bit</td>
<td>589</td>
<td>1,370</td>
<td>1,822</td>
<td>845</td>
<td>193</td>
<td>204</td>
<td>213</td>
</tr>
<tr>
<td>24-bit</td>
<td>718</td>
<td>2,049</td>
<td>2,695</td>
<td>1,037</td>
<td>226</td>
<td>296</td>
<td>294</td>
</tr>
<tr>
<td>32-bit</td>
<td>937</td>
<td>2,704</td>
<td>3,557</td>
<td>1,191</td>
<td>288</td>
<td>391</td>
<td>362</td>
</tr>
<tr>
<td>Uniform</td>
<td>935</td>
<td>2,716</td>
<td>3,551</td>
<td>1,194</td>
<td>288</td>
<td>390</td>
<td>366</td>
</tr>
<tr>
<td>Exponential</td>
<td>626</td>
<td>1,421</td>
<td>1,893</td>
<td>933</td>
<td>221</td>
<td>220</td>
<td>230</td>
</tr>
<tr>
<td>Normal</td>
<td>621</td>
<td>1,418</td>
<td>1,886</td>
<td>911</td>
<td>214</td>
<td>217</td>
<td>226</td>
</tr>
<tr>
<td>Avg.</td>
<td>683</td>
<td>1,768</td>
<td>2,327</td>
<td>950</td>
<td>224</td>
<td>264</td>
<td>261</td>
</tr>
<tr>
<td>Speed-up ratio</td>
<td></td>
<td></td>
<td></td>
<td>3.1×</td>
<td>6.7×</td>
<td>8.9×</td>
<td>5.3×</td>
</tr>
</tbody>
</table>

(see Section 9.3). Consequently, Fibonacci of order 3 can be used in cases when we prefer the higher compression ratio rather than the lower decoding time. On the other hand, Elias-Fibonacci is used in cases when we prefer the lower decoding time rather than the highest compression ratio.

![Decoding Time for all conventional and fast algorithms](image)

Figure 10.5: Decoding time for all conventional and fast algorithms

In the case of an 8-bit data collection, fast algorithms do not utilize shift operations; therefore, the decoding times are very similar. In this case, Fibonacci codes of any order can be the most efficient choice because these codes provide higher compression ratios than the Elias-delta or Elias-Fibonacci codes.

![Fast Decoding Algorithm's Speed Up Ratio](image)

Figure 10.6: Average speed-up ratio for all fast encoding algorithms
In the last row of Table 10.6 and in Figure 10.6, we see the average speed-up ratios of all fast decoding algorithms for all collections. The speed-up ratio ranges from 3.1 of the Elias-delta code to 8.9 of the Fibonacci of order 3 code.

10.5 Who Is the Winner?

When we rank the fast algorithms according the encoding and decoding times and compression ratio of the codes (see Tables 10.7 and 10.8), we can select the best code for a concrete situation: the Elias-Fibonacci code achieves the higher compression ratio than the Elias-delta code because it produces shorter codewords. From the encoding time point of view, Elias-Fibonacci does not require any filtering of the highest 1-bit; therefore, it is faster than the Elias-delta code. The Fibonacci code of order 3 is the most efficient from the compression ratio point of view; however, results of the Elias-Fibonacci code are very similar: the difference is $0.7 - 6.2\%$. Evidently, we must keep in mind that fast Elias-Fibonacci encoding is up-to $3 \times$ faster than fast Fibonacci encoding in the case of domains of the bit-length $\geq 16$. On the other hand, in the case of the 8-bit test collection, we prefer the Fibonacci code of order 3 from the encoding time as well as the compression ratio point of view.

Table 10.7: A ranking of fast encoding and decoding algorithms according the processing time

<table>
<thead>
<tr>
<th>Test Collection</th>
<th>Fast Encoding Algorithm</th>
<th>Fast Decoding Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ED F2 F3 EF</td>
<td>ED F2 F3 EF</td>
</tr>
<tr>
<td>8-bit</td>
<td>4 2 1 3</td>
<td>4 2 3 1</td>
</tr>
<tr>
<td>16-bit</td>
<td>2 3 4 1</td>
<td>2 3 4 1</td>
</tr>
<tr>
<td>24-bit</td>
<td>2 3 4 1</td>
<td>2 3 4 1</td>
</tr>
<tr>
<td>32-bit</td>
<td>2 3 4 1</td>
<td>2 3 4 1</td>
</tr>
<tr>
<td>Uniform</td>
<td>2 3 4 1</td>
<td>2 3 4 1</td>
</tr>
<tr>
<td>Exponential</td>
<td>3 2 4 1</td>
<td>3 2 4 1</td>
</tr>
<tr>
<td>Normal</td>
<td>3 2 4 1</td>
<td>3 2 4 1</td>
</tr>
<tr>
<td><strong>Avg.</strong></td>
<td>2 3 4 1</td>
<td>2 3 4 1</td>
</tr>
</tbody>
</table>

When we compare both Fibonacci codes, fast encoding of the Fibonacci code of order 2 is more efficient than fast encoding of the Fibonacci code of order 3 on average. However, if we prefer the compression ratio then the Fibonacci code of order 3 is a better choice than the Fibonacci code of order 2. The Elias-delta code is the right choice when the frequency of the minimal value\(^3\) in an input collection is high; this code has the shortest codeword of the minimal value. The performance of fast decoding algorithms proves similar results; therefore, we are simply able to select the most appropriate code for a collection if we know the distribution of numbers in the collection.

\(^3\)We assume 1 as the minimal value (see definitions of the codes in Section 3).
Table 10.8: A ranking of codes for three 32-bit collections according the compression ratio

<table>
<thead>
<tr>
<th>Test Collection</th>
<th>Elias-delta</th>
<th>Fibonacci 2</th>
<th>Fibonacci 3</th>
<th>Elias-Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Exponential</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Normal</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Encoding and decoding numbers with large number domains, e.g. 64 or 128-bits, utilizes the same fast algorithms. It only requires a preparation of longer encoding tables\(^4\), because we need more Fibonacci right shifts and also the prefixes of the Elias-delta and Elias-Fibonacci codes are longer. However, the representation of numbers in algorithms can be more difficult since a 128-bit number is represented by four 32-bit numbers on a 32-bit CPU or two 64-bit numbers on a 64-bit CPU. Also the arithmetic operations are more complex.

\(^4\)FRSEIT in the case of the fast Fibonacci encoding algorithms (see Section 6.5.3), EDTAB and EFTAB in the case of the fast Elias-delta and Elias-Fibonacci encoding algorithms (see Section 7.2).
Chapter 11

Applications of Fast Encoding and Decoding Algorithms

11.1 Compression of XML Node Streams

11.1.1 Introduction

In recent years, many approaches to XML twig pattern query (TPQ) processing have been developed. Indexing techniques for an XML document have been studied extensively and works such as [68, 62, 31, 11, 18, 39, 19, 20] have outlined basic principles of streaming scheme approaches. Nodes of an XML tree are labeled by a labeling scheme [68, 62] and stored in streams where the XML node tag is often used as a key of one stream. Labels retrieved for each query node tag are then merged by an XML join algorithm such as structural [11] or holistic [18] joins. The stream abstract data type (ADT) is usually implemented with inverted lists or special purpose data structures [39, 43]. Since XML node labels are often small values, we can apply universal variable-length codes for their compression. In [14], we introduced this application of the fast encoding/decoding algorithms for variable-length codes. Since labels are sequentially read, when we do not consider any index, variable-length codewords of the fixed-length as well as variable-length labels are stored without any overhead.

In Section 11.1.2, we describe an XML model. Section 11.1.3 introduces the stream ADT. In Section 11.1.4, we describe various compression techniques applied to the stream ADT. Section 11.1.5 includes experimental results.

11.1.2 XML Model

An XML document can be modeled as a rooted, ordered, labeled tree, where every node of the tree corresponds to an element or an attribute of the document and edges connect elements, or elements and attributes, having a parent-child relationship. We call such representation of an XML document an XML tree. We can see an example
of the XML tree in Figure 11.1. We use the term ‘node’ to define a node of an XML tree which represents an element or an attribute.

The labeling scheme associates every node in the XML tree with a label. These labels allow us to determine structural relationships between nodes. Figures 11.1(a) and 11.1(b) show the XML tree labeled by Containment [68] and Dewey order [62] labeling schemes, respectively.

Figure 11.1: (a) Containment labeling scheme (b) Dewey order labeling scheme

The Containment labeling scheme creates labels using the document order of nodes. We can use a simple counter, which is incremented every time we sequentially read the start or end tags of an element. The first and the second numbers of a node label represent a value of the counter when the start and end tags are read, respectively. In the case of Dewey order, every number in the label corresponds to one ancestor node. Evidently, Dewey order provides more appropriate updating features, on the other hand, we have to deal with variable-length labels.

11.1.3 Stream ADT

Holistic approaches [18, 20] use an ADT called a stream. A stream is an ordered set of node labels with the same schema node label. There are many options to create schema node labels (also known as streaming schemes in articles related to holistic joins [18, 20]). A cursor pointing to the first node label is assigned to each stream. We distinguish the following operations of a T stream: head(T) – returns the node label to the cursor’s position, eof(T) – returns true iff the cursor is at the end of T, advance(T) – moves the cursor to the next node label. An implementation of the stream ADT usually contains additional operations: openStream(T) – open the stream T for reading, closeStream(T) - close the stream.

The stream ADT is often implemented by an inverted list. In the following section we describe a data structure called stream array [14], which implements the stream ADT.
Persistent Stream Array

Persistent stream array is a data structure using a common paged scheme [28], where labels are stored in blocks of the secondary storage and the main memory cache keeps some blocks. In Figure 11.2, we see an overview of the scheme. The cache utilizes the least recently used (LRU) schema for a selection of cache blocks being removed [28]. Each block includes an array of tuples (node labels) and a pointer to the next block in the stream. Pointers enable a dynamic character of the data structure. We can easily insert or remove tuples from the blocks using the node split or merge operations. Blocks do not have to be fully utilized; therefore, we also keep the number of tuples stored in each block.

![Logical structure of stream array](image)

Figure 11.2: An overview of the persistent stream array

**Insert and delete operations** The insert operation is very simple. We first search a location of the new label within a stream and then we test whether there is enough space for the new label or not. We can utilize a common label shift between blocks or a split operation of the block to create a space for the new label. The delete operation first searches a position of a label which should be deleted and then removes this label from the block. A merge operation of blocks can be also utilized in this case.

Searching a location of the label within a stream during the insert can be quite a time-consuming process since the stream can span many blocks. A more common operation in the case of an XML index is the insert operation of many labels which corresponds to insertion of a complete XML document (so called bulk-insert operation).

**Bulk-insert operation** The insert of a whole XML document (or several documents) is a quite common operation when working with XML databases. Loading an XML document into the index can be processed significantly faster then using the label-by-label insert operation. The situation is simple thanks to the fact that we create labels in the same order in which they are stored in the stream. Therefore, during a bulk-insert operation, we sequentially read the input XML document, we create labels and we store the labels at the end of each stream. Compared to the insert operation, we do not have to search for the exact position of a new label in the stream. In Section 11.1.5, we utilize the bulk-insert algorithm to build indices.
Compressed Stream Array

An advantage the compressed stream array is decreasing the size of the data file and, therefore, decreasing the disk access cost. Of course, there is an extra time spent on compression and decompression of data. The compression and decompression time should be lower or equal to the time saved by having the lower disk access cost. Moreover, we can save some processing time due to more cache hits of the CPU’s caches. As result, a compression algorithm should be fast and should have a good compression ratio.

The stream array has a specific feature to enable efficient compression. We never access items in one block randomly during the stream is read. Therefore, we do not decompress nodes after they are read from the secondary storage; the main memory cache includes compressed nodes. The actual cursor position is created during a stream is opened; it also contains one tuple with an encoded current label. Because of sequential reading, each label is encoded only once during the \textit{advance}(T) operation. The \textit{head}(T) operation returns the encoded tuple assigned to the cursor. Using this schema, we keep data compressed even in the main memory and only one decompressed tuple is assigned to each open stream.

11.1.4 Compression of XML Node Streams

In this section, we describe methods utilized to compress XML node streams. We must keep in mind that labels are sorted in a block. We use the following compression methods: 1. Fixed-length tuple, 2. Variable-length tuple, 3. Common prefix, 4. Variable-length coding, and finally, 5. Variable-length coding with the reference item. Obviously, the first two methods are not real compression methods, and we use them as baselines for a comparison with other methods. We describe these methods in the following sections in more detail. In the following text, we assume that an integer is stored using 4\,B.

Fixed-length and Variable-length Tuple Methods

The variable-length tuple method is usable only in the case of a path-based labeling scheme like Dewey order. In the case of the fixed-length tuple method and Dewey order, we must set the maximal tuple length and all shorter tuples must be filled by a gap value to reach this length.

\textbf{Example 37 (The fixed-length and variable-length tuple methods)}

\textit{Let us have these two tuples: }\langle 1, 2 \rangle \text{ and } \langle 1, 2, 3, 7 \rangle. \textit{When using the variable-length tuple method, they occupy } 6 \times 4\,B + 2\,B \text{ for the length of these two tuples. If we use the fixed-length tuple method, the first tuple must be filled, so it looks like } \langle 1, 2, 0, 0 \rangle. \textit{Consequently, these two tuples occupy } 8 \times 4\,B.
Common Prefix Method

The common prefix compression method is based on an idea of Run Length Encoding (RLE) [54]. Due to the ordering of tuples in a block, the ancestor of a tuple is very similar and, therefore, we do not have to store each value.

Example 38 (The common prefix method)
Let us have the following tuples: \(1, 2, 3, 7, 9, 7\), \(1, 2, 3, 7, 5, 6, 7\), \(1, 2, 3, 7, 7, 0, 0, 7\). The first tuple is not compressed, due to the fact that it has no ancestor. If we compare the second tuple with the first one, it differs only in the 3 last dimensions, and the third tuple differs in the last 4 dimensions. Therefore, we store the number of the same values and all differences. The result is as follows: 0 – \(1, 2, 3, 7, 9, 7\), 4 – \(5, 6, 7\), 4 – \(7, 0, 0, 7\). Using this method we obtain \(7 \times 4\) B for the first tuple, \(4 \times 4\) B for the second tuple, and \(5 \times 4\) B for the third one. As a result, we obtain \(16 \times 4\) B = 64 B. In the case of the fixed-length method, we need \(8 \times 3 \times 4\) B = 96 B, because all tuples must be extended to the length 8.

Variable-length Coding

In this case, tuples are encoded by a variable-length code proposed in Chapter 3. This compression method is based on the behaviour of variable-length codes: smaller numbers are encoded by shorter codewords. Smaller numbers are especially provided with Dewey order since the range of the Containment labeling scheme includes relatively bigger values.

Example 39 (Variable-length coding)
Let us have a tuple \(1, 2, 3, 7\). After encoding the tuple with Fibonacci of order 2, this tuple is stored as the sequence 11011001101011 to occupy 2 B instead of original 16 B (it means \(4 \times 4\) B).

Variable-length Coding with Reference Item

Since numbers rapidly grow up in the case of the Containment labeling scheme, the variable-length codes become inefficient. This issue is less noticeable in the case of Dewey order. Tuples in a stream array are sorted and we can use this feature to compress a tuple using its ancestor. The first tuple in the block is stored unchanged. In order to keep all other numbers small, we store the differences between the current tuple and the previous tuple. The first tuple and all differences are then encoded with any variable-length code. This difference coding is a well-known technique in the field of data compression [54].

Example 40 (Variable-length coding with the reference item)
Let us have two tuples: \(1000, 200, 300, 7\) and \(1005, 220, 100, 7\). From this example, we see that we can subtract the first 2 values. After the subtraction, we obtain tuples \(1000, 200, 300, 7\) and \(5, 20, 100, 7\) to be encoded faster and they also occupy less space.
11.1.5 Experimental Results

In our experiments\(^1\), we use the XMARK\(^2\) data collection with the factor 2, 240 MB in size; it includes \(4 \times 10^6\) nodes. We generate labels for all nodes by means of two proposed labeling schemes: the Containment labeling scheme with the fixed-length labels and the Dewey order labeling scheme with the variable-length labels. Since a labeled path streaming scheme is used [13, 20], 512 streams were created.

We provide a set of tests where we simulate a real work with the stream array and measure the influence of compression methods. For each test, we randomly select 100 streams and read them until the end. This is processed by holistic as well structural joins during the query processing. As usual, tests have been processed with cold caches (the OS cache as well as the data cache of indices). For all tests we measure the index size, query processing time, Disk Access Cost (DAC), and time of index building. The query processing time is the time needed for opening each randomly selected stream plus the time needed for decompression of all tuples of the stream. DAC is equal to the number of disk accesses during query processing where the block size is 2 kB. For the time measurement, we repeat each test 10× and calculate the average time. In tables and figures we use the abbreviation ‘RI’ instead of ‘reference item’, so ‘Fibonacci 2 RI’ means ‘Fibonacci 2 with the reference item’.

Results for Containment Labeling Scheme

Results for the fixed-length Containment labeling scheme are shown in Table 11.1 and Figure 11.3. Since coded labels include a lot of minimal values when the reference item has been used, Elias-delta RI produces the smallest index file and the lowest index build time. Since the fast decoding algorithm for Elias-Fibonacci outperforms the fast decoding algorithm for Elias-delta, Elias-Fibonacci RI outperforms Elias-delta although DAC is lower in the case of Elias-delta RI. All variable-length codes produce similar results from the index size, query processing time, and DAC point of view and we see that they outperform often used RLE (the Common Prefix method); the index size is approximately 2.4× lower. Evidently, the compression method with the reference item is more efficient then the compression method without the reference item. The compression method with the reference item saves approximately 68 % of the index size, 50 % of the query processing time, and 68 % of DAC. The index size for Elias-delta RI forms approximately 37% of the index size when no compression method is used (the Fixed-length method).

\(^1\)The experiments were executed on an Intel\(^\circledR\) Core 2 Duo 2.4 Ghz, 512 kB L2 cache; 3 GB RAM; Windows 7.

\(^2\)http://monetdb.cwi.nl/xml/
Table 11.1: The index size, index build time, query processing time, and DAC for the Containment labeling scheme

<table>
<thead>
<tr>
<th>Compression Method</th>
<th>Index Size [kB]</th>
<th>Index Build Time [s]</th>
<th>Query Processing Time [s]</th>
<th>DAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-length</td>
<td>97,708</td>
<td>58.22</td>
<td>3.23</td>
<td>3,551</td>
</tr>
<tr>
<td>Common Prefix</td>
<td>85,736</td>
<td>52.54</td>
<td>2.37</td>
<td>3,119</td>
</tr>
<tr>
<td>Elias-delta</td>
<td>71,448</td>
<td>82.49</td>
<td>2.21</td>
<td>2,611</td>
</tr>
<tr>
<td>Fibonacci 2</td>
<td>73,688</td>
<td>97.16</td>
<td>2.18</td>
<td>2,695</td>
</tr>
<tr>
<td>Fibonacci 3</td>
<td>67,860</td>
<td>92.29</td>
<td>2.12</td>
<td>2,475</td>
</tr>
<tr>
<td>Elias-Fibonacci</td>
<td>71,000</td>
<td>92.36</td>
<td>2.12</td>
<td>2,588</td>
</tr>
<tr>
<td>Elias-delta RI</td>
<td>36,252</td>
<td>48.30</td>
<td>1.55</td>
<td>1,356</td>
</tr>
<tr>
<td>Fibonacci 2 RI</td>
<td>37,040</td>
<td>61.21</td>
<td>1.56</td>
<td>1,365</td>
</tr>
<tr>
<td>Fibonacci 3 RI</td>
<td>39,472</td>
<td>54.98</td>
<td>1.55</td>
<td>1,467</td>
</tr>
<tr>
<td>Elias-Fibonacci RI</td>
<td>38,780</td>
<td>60.03</td>
<td>1.52</td>
<td>1,437</td>
</tr>
</tbody>
</table>

Results for Dewey Order Labeling Scheme

Results for the variable-length Dewey order are shown in Table 11.2 and Figure 11.4. Since coded labels include a lot of minimal values when the reference item method has been used, Elias-delta RI outperforms all other methods. Obviously, all variable-length codes produce very similar results; the compression saves approximately 80% of the index size, 64% of the query processing time, and 80% of DAC. Moreover, the index build time is approximately $2.8 \times$ more efficient than in the case of the fixed-length tuple method. Since the reference item method produces a lot of minimal values, we can reduce the number of minimal values by RLE.

Table 11.2: The index size, index build time, query processing time, and DAC for the Dewey order labeling scheme

<table>
<thead>
<tr>
<th>Compression Method</th>
<th>Index Size [kB]</th>
<th>Index Build Time [s]</th>
<th>Query Processing Time [s]</th>
<th>DAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-length</td>
<td>274,620</td>
<td>132.98</td>
<td>9.63</td>
<td>9,968</td>
</tr>
<tr>
<td>Variable length</td>
<td>127,156</td>
<td>79.81</td>
<td>3.06</td>
<td>4,624</td>
</tr>
<tr>
<td>Common Prefix</td>
<td>77,676</td>
<td>55.78</td>
<td>2.35</td>
<td>2,832</td>
</tr>
<tr>
<td>Elias-delta</td>
<td>41,504</td>
<td>74.51</td>
<td>1.59</td>
<td>1,516</td>
</tr>
<tr>
<td>Fibonacci 2</td>
<td>41,124</td>
<td>86.30</td>
<td>1.58</td>
<td>1,496</td>
</tr>
<tr>
<td>Fibonacci 3</td>
<td>43,232</td>
<td>80.00</td>
<td>1.58</td>
<td>1,571</td>
</tr>
<tr>
<td>Elias-Fibonacci</td>
<td>43,192</td>
<td>86.24</td>
<td>1.59</td>
<td>1,578</td>
</tr>
<tr>
<td>Elias-delta RI</td>
<td>28,076</td>
<td>46.68</td>
<td>1.29</td>
<td>1,026</td>
</tr>
<tr>
<td>Fibonacci 2 RI</td>
<td>30,896</td>
<td>54.78</td>
<td>1.45</td>
<td>1,126</td>
</tr>
<tr>
<td>Fibonacci 3 RI</td>
<td>35,172</td>
<td>48.98</td>
<td>1.51</td>
<td>1,285</td>
</tr>
<tr>
<td>Elias-Fibonacci RI</td>
<td>31,576</td>
<td>58.78</td>
<td>1.37</td>
<td>1,157</td>
</tr>
</tbody>
</table>

Although we utilized conventional bit-by-bit encoding algorithms for the variable-length codes, the index build time for these methods is more efficient than in the case of methods not using variable-length codes. Since Dewey order is a more important labeling scheme from the update point of view, it is important to see that we achieve smaller index sizes than with the Containment labeling scheme. It is because sibling labels are similar; they differ only in the last values. Consequently, the
Figure 11.3: (a) The index size (b) Index build time (b) Query processing time (d) DAC for the Containment labeling scheme

compression method using the reference item is more efficient than the compression method without the reference item.

11.2 Text Compression

In the case of text compression, we reduce the amount of space needed to store text files on computers; it is processed by removing redundancy in data. There are many good reasons to invest computing resources required to perform compression. A reduction of the storage space and faster transmission of data can yield significant cost savings and often improve the performance.

There are two general methods of text compression: statistical and dictionary-based methods [15]. In statistical compression methods, each symbol is assigned a codeword based on the frequency of its occurrence. Highly frequent symbols obtain short codewords and vice versa. In dictionary-based compression methods, groups of consecutive characters, or terms, are replaced by a codeword. The terms represented by a codeword are then found by looking it up in a dictionary.

The most used and highly efficient statistical methods are based on Huffman [33] or arithmetic coding [32, 44]. The most successful dictionary methods are based on Ziv-Lempel compression [69]. Basic variants of these methods have been considerably refined and enhanced to the current high-performance algorithms [54, 16].
Figure 11.4: (a) The index size (b) Index build time (c) Query processing time (d) DAC for the Dewey order labeling scheme

A detailed description of these methods is beyond the scope of this work. To demonstrate usefulness of fast encoding and decoding algorithms proposed in this work, we show a simple dictionary-based method using these algorithms in Section 11.2.1. In Section 11.2.2, we put forward results of the compression method.

11.2.1 Simple Dictionary-Based Compression Method

As usual, the first step of each compression method is parsing an input text document. Many various parsing methods exist [15]; terms with one or more characters or whole words are a result of these methods. After the document is parsed, terms are put into a dictionary and they are sorted with their frequencies of occurrence. Now, codewords are assigned to these sorted terms; shortest codewords to the most frequent terms and vice versa. Once the dictionary has been created, we encode the text document in such a way that we replace the terms by codewords. Since codewords are chosen such that they occupy less space on average than the terms they represent, the text document is compressed.

This two-step compression scheme is depicted in Figure 11.5. In the first step, an input document is parsed; a sequence of terms is the result. The terms are sorted by their frequencies of occurrence. We can utilize any universal variable-length code described in Chapter 3 to create the dictionary. In the second step, codewords are written into an output file. We must note that the dictionary must be a part of the output file. The fast encoding algorithms described in Chapter 7 can be used to encode the terms. The decompression is an opposite process; a decoding method is
used to translate codewords to terms which are written into a file. The fast decoding algorithms described in Chapter 9 can be used to decode the codewords.

### 11.2.2 Experiments

In our test, we used a real collection; the King James Bible in English included in the Canterbury corpus [51]. After the collection is parsed, we obtain 766,131 terms (words in this case) and a dictionary of all unique terms with their frequencies of occurrence. Since 13,744 terms are unique, the length of the dictionary is 13,744. We measured parsing, encoding, and decoding times (see Table 11.4). The parsing and encoding times define the total compression time, whereas the decoding time is the same as the decompression time. We compared the processing times with the zip program included in MS Windows 7 [48]. The compression and decompression times are also shown in Table 11.4. We see that this simple compression method outperforms zip for all codes.

The compression time of variable-length codes is approximately $4 \times$ lower than the time required by the zip program. The decompression time is approximately $6 \times$ lower for variable-length codes than the time of the zip program. The most efficient compression algorithm utilizes the fast Fibonacci of order 3 encoding algorithm, on the other hand, the most efficient decompression algorithm utilizes the fast Fibonacci of order 2 decoding algorithm. Consequently, the fast Fibonacci algorithms are more efficient than the fast Elias-delta and Elias-Fibonacci algorithms.

In Table 11.3, we see the size of encoded data for all codes. This table also includes the size of the dictionary. Even the dictionary is not compressed at all, the size of the dictionary plus the size of encoded data is smaller than the size of the text file compressed by zip. The Fibonacci code of order 3 produced the smallest compressed file. The size using the Fibonacci code of order 3 is 93.7% of the size produced by the Elias-Fibonacci code since the terms with the higher frequency are

Figure 11.5: A scheme of a dictionary-based compression method
Applications of Fast Encoding and Decoding Algorithms

conzentrated in the 8-bit domain. These results are similar to the results for the 8-bit collection in Chapter 10. However, fast Elias-Fibonacci encoding is more efficient than fast Fibonacci of order 3 encoding in the case of the other test collections in Chapter 10.

Table 11.3: The size of compressed data for the test collection [B]

<table>
<thead>
<tr>
<th></th>
<th>Size [B]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encoded Data</td>
</tr>
<tr>
<td>Original file</td>
<td>4,047,392</td>
</tr>
<tr>
<td>Zip</td>
<td></td>
</tr>
<tr>
<td>Fast Elias-delta</td>
<td>992,724</td>
</tr>
<tr>
<td>Fast Fibonacci 2</td>
<td>909,746</td>
</tr>
<tr>
<td>Fast Fibonacci 3</td>
<td>906,997</td>
</tr>
<tr>
<td>Fast Elias-Fibonacci</td>
<td>966,611</td>
</tr>
</tbody>
</table>

Table 11.4: Compression and decompression times [ms]

<table>
<thead>
<tr>
<th></th>
<th>Processing Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Document parsing</td>
</tr>
<tr>
<td>Fast Elias-delta</td>
<td>113</td>
</tr>
<tr>
<td>Fast Fibonacci 2</td>
<td>9.21</td>
</tr>
<tr>
<td>Fast Fibonacci 3</td>
<td>8.47</td>
</tr>
<tr>
<td>Fast Elias-Fibonacci</td>
<td>9.39</td>
</tr>
<tr>
<td>Zip</td>
<td>443</td>
</tr>
</tbody>
</table>

Although a comparison of this method with other text compression methods is beyond the scope of this section, it demonstrates usefulness of the fast encoding and decoding algorithms proposed in this thesis.
Chapter 12

Conclusion

This thesis begins with an overview of the most used variable-length codes; Huffman, Golomb, Elias, and Fibonacci of order 2 and 3 codes are described in more detail. Moreover, we introduced a new code titled the Elias-Fibonacci code.

Next chapters described the fast encoding and decoding algorithms for the Elias-delta, Fibonacci of order 2 and 3, and Elias-Fibonacci codes. The fast algorithms are based on a strong theoretical background, e.g. Fibonacci shift operations and their efficient computation and the efficient computation of the nearest lower Fibonacci sum are introduced. As result, the fast encoding algorithms are up-to 12.5× faster than the conventional encoding algorithms and the fast decoding algorithms are up-to 8.9× faster than the conventional decoding algorithms.

The new Elias-Fibonacci code achieved the fastest encoding and decoding time on average for domains of the bit-length $\geq 16$. On the other hand, when the 8-bit domain is considered, the Fibonacci code of order 3 outperforms other codes. Fibonacci of order 3 also achieved the lowest size of encoded data; however, results of Elias-Fibonacci are very similar. Consequently, we provided results showing various situations where a concrete code outperforms each other.

We utilized the fast encoding and decoding algorithms in two applications demonstrating the efficiency of these algorithms: compression of XML node streams and compression of text files. In our future work, we want to develop fast algorithms for other universal codes and we also plan to apply the universal codes in more compression algorithms.
Author’s Bibliography

Recorded: SCOPUS (SJR: 0.331/2011), WoS.

Recorded: SCOPUS.

Recorded: SCOPUS, WoS.


Recorded: SCOPUS.


Author’s Bibliography not related to this work


Bibliography


Index

adaptive code, 6
adaptive Huffman code, 10
ASCII, 6
binary codes, 4
bit-length, 4
buffer, 28
code, 4
codeword, 4
conventional encoding/decoding of Elias-delta code, 31
conventional encoding/decoding of Elias-Fibonacci code, 36
conventional encoding/decoding of Fibonacci code of order 2, 32
conventional encoding/decoding of Fibonacci code of order 3, 34
data compression, 5
decoding, 5
dictionary, 111
dynamic code, 6
efficient computation of the Extended Fibonacci right shift, 50
efficient computation of the Fibonacci left shift, 52
efficient computation of the Fibonacci right shift, 45
efficient computation of the nearest lower Fibonacci sum, 55
Elias codes, 15
Elias-delta code, 16, 24
Elias-Fibonacci code, 23, 24
Elias-gamma code, 15, 25
Elias-omega code, 16
encoded Fibonacci representation, 45
encoding, 5
ExFRSEIT table, 50
Exp-Golomb code, 15
Exponential Golomb code, 15, 24
Extended Fibonacci representation, 42
Extended Fibonacci right shift, 44
fast decoding of Elias-delta code, 87
fast decoding of Elias-Fibonacci code, 87
fast decoding of Fibonacci code, 91
fast encoding of Elias-delta code, 60
fast encoding of Elias-Fibonacci code, 60
fast encoding of Fibonacci code of order 2, 62
fast encoding of Fibonacci code of order 3, 65
Fibonacci code, 17, 24
Fibonacci code of order 2, 19, 25
Fibonacci code of order 3, 25
Fibonacci codes of higher orders, 21
Fibonacci left shift, 44
Fibonacci number, 17
Fibonacci representation, 17
Fibonacci right shift, 44
Fibonacci shift, 43
Fibonacci sum, 21
finite automaton, 28
fixed-length code, 5
frequency of occurrence, 5
FRSEIT table, 45
general fast decoding algorithm, 69
generalized Fibonacci code, 21
golden ratio, 41
Golomb code, 14
Golomb-Rice codes, 14, 25
Huffman code, 6, 9, 24
Huffman translation table, 9
Huffman tree, 9

123
insignificant Fibonacci numbers of the Extended Fibonacci representation, 44
LogTable256, 29, 99
MAP mapping table, 68
nearest lower Fibonacci sum, 36
NYT, not yet transmitted, 11
P-codes, 5
prefix codes, 5
reverse Fibonacci code, 22
reverse Fibonacci code of order 2, 19
reverse Fibonacci representation, 19
segment, 28
standard binary representation, 5, 6
static Huffman code, 9
static/dynamic code, 6
term, 111
translation table, 5, 9
translation table of static code, 6
Unary code, 8
Unicode, 6
uniquely decodable codes, 5
universal code, 6
variable-length code, 5