

MARKS GYMNASIESKOLA

# How are images represented digitally?

I	M	5	6	F
L	O	W	E	R
0	1	2	3	4
4	4	4	4	0
4	4	3	4	4
4	3	1	3	4
4	4	3	4	4
4	4	3	4	4
2	2	2	2	2

This work is not intended for publication

*Years:*  
2011–2012

*Author:*  
Eric Arnebäck

*Supervisor:*  
Aref Hamawi

# Contents

1	Introduction and Abstract • 7
1.1	Introduction • 7
1.2	Abstract • 8
2	Purpose and problem • 11
3	Methods • 12
3.1	Tools • 12
3.2	Structure Of This Text • 12
3.3	Sample source code • 13
4	Preliminaries • 15
4.1	Glossary • 15
4.2	Pseudocode Conventions • 21
4.3	Answers to the exercises • 26
5	Digital Color • 29
5.1	What is color? • 29
5.2	Color Depth • 30
5.3	Color Memory Layout • 33
5.4	Answers to the exercises • 34
6	Run-Length Encoding • 35
6.1	Some Data Compression Terminology • 35
6.2	The Counting Argument • 36
6.3	RLE • 37
6.4	PackBits RLE • 40
6.5	Answers to the exercises • 43
7	Truevision Graphics Adapter • 45
7.1	Building Blocks • 45
7.2	File Header • 46
7.3	Color Data • 48
7.4	File Footer • 50
7.5	Developer Area • 51
7.6	Extension Area • 51
8	LZW • 55
8.1	History • 55

- 
- 8.2 The Compression algorithm • 55
  - 8.3 Implementation of The Compression Algorithm • 57
  - 8.4 Description of the Decompression algorithm • 58
  - 8.5 Compression Efficiency Of LZW • 60
  - 8.6 Answers The Exercises • 63
  - 9 Graphics Interchange Format • 65
    - 9.1 History • 65
    - 9.2 Format Version • 66
    - 9.3 Building Blocks of the format • 66
    - 9.4 Header Blocks • 67
    - 9.5 Color Tables • 69
    - 9.6 Preimage Data Blocks • 70
    - 9.7 Graphic-Rendering blocks • 74
    - 9.8 Miscellaneous Blocks • 77
    - 9.9 GIF syntax • 79
  - 10 Checksums • 81
    - 10.1 Checksums and error detection • 81
    - 10.2 CRC • 82
    - 10.3 Adler-32 • 90
    - 10.4 Answers To The Exercises • 91
  - 11 Huffman Coding • 93
    - 11.1 History • 93
    - 11.2 Code Lists • 93
    - 11.3 Huffman Coding • 95
    - 11.4 Answers to the exercises • 97
  - 12 LZ77 and LZSS • 98
    - 12.1 LZ77 • 98
    - 12.2 LZSS • 102
    - 12.3 Answers to the exercises • 102
  - 13 Deflate • 103
    - 13.1 History • 103
    - 13.2 Length and Distance Codes • 103
    - 13.3 Fixed Huffman Codes • 104
    - 13.4 Dynamic Huffman Codes • 105
    - 13.5 Storing the Huffman Codes • 108
    - 13.6 The Deflate format • 110
  - 14 Portable Network Graphics • 113
    - 14.1 History • 113
    - 14.2 Buildings Blocks • 113
    - 14.3 Critical Chunks • 115
    - 14.4 Image Data Storage Model • 117
    - 14.5 Answers to the exercises • 125
  - 15 Conclusion • 126

Bibliography • 128

# List of Figures

- 5.1 The `RGB` color model • 30
- 5.2 The color blue with an increasingly lower alpha channel • 30
- 9.1 The grammar of a valid `GIF`-file • 80
- 14.1 The `PNG` chunk datatype • 113
- 14.2 Examples of gradients • 121

# List of Tables

- 4.1 The ASCII table • 17
- 4.2 Logical truth tables • 22
  
- 7.1 TGA Image Type values • 47
- 7.2 The different image origin combinations • 48
  
- 8.1 The initial LZW string table • 56
- 8.2 Detailed LZW compression process • 57
- 8.3 The Canterbury Corpus • 61
- 8.4 LZW compression ratio test results • 62
  
- 9.1 GIF interlacing table rows • 72
- 9.2 Example of an initial GIF compression table • 75
  
- 11.1 Examples of code lists • 94
- 11.2 Computed code list • 96
  
- 13.1 Deflate length codes • 104
- 13.2 Deflate distance codes • 104
- 13.3 Fixed Huffman codes for the first alphabet • 105
- 13.4 Sorted Huffman Codes • 106
  
- 14.1 The different color types of the PNG format • 115
- 14.2 The allowed color depths of the PNG format • 116

# List of Algorithms

- 4.1 The for each control structure • 24
- 4.2 The repeat control structure • 25
- 4.3 Euclid's algorithm • 25
- 5.1 Parsing TGA 16-bit color • 32
- 6.1 Encoding a file using RLE • 39
- 6.2 Decoding a RLE encoded file • 39
- 6.3 Encoding a file using PackBits RLE • 42
- 6.4 Decoding a RLE packbits encoded file • 43
- 7.1 Reading and decompressing the color data of a TGA file • 50
- 8.1 LZW compression algorithm • 57
- 8.2 LZW decompression algorithm • 60
- 9.1 Undoing the interlacing of the uncompressed GIF color data • 71
- 9.2 GIF LZW Decompression algorithm • 76
- 10.1 Computing a CRC of width 8 • 86
- 10.2 Computing a CRC of width 32 • 89
- 10.3 CRC computation for the PNG format • 90
- 10.4 Computation of the Adler-32 checksum • 91
- 12.1 Decoding a LZ77 token • 100
- 13.1 Finding the smallest code of each code length • 108
- 14.1 The Paeth-predictor. • 123
- 14.2 Undoing PNG interlacing • 125

# Chapter 1

## Introduction and Abstract

### 1.1 Introduction

One day a couple of years ago, I accidentally opened an executable file(.exe) on my computer in a text editor. In front of my eyes I saw an absolutely incomprehensible mess of characters and numbers. No matter how much I examined this mess, I could see no order in it. This event really got me thinking about one thing: *how* exactly is all the data stored in a computer actually *represented*?

A couple of months after this, I learned about the concept of a hex dump. A hex dump is a hexadecimal representation of the data in a file. It was at this point I finally realized something that should have been obvious to me: since the language that the computer talks in is just a bunch of ones and zeroes, then obviously all the data stored in a computer is also just a bunch of ones and zeroes! And furthermore, saying that a computer speaks in ones and zeroes is just another way of saying that it speaks in *numbers*. Since numbers, and mathematics in general, have always been one of my greatest interests in life, this episode served to greatly strengthen my interest in how data is represented in a computer.

A way of storing a certain type of data is called a *file format*. The exact structure of a file format tends to be specified in something called that file format's *specification*. But alas, reading these documents often requires the reader already being moderately knowledgeable about the kind of data that the file format stores.

And so I realized that it would probably require a great amount of research in order to truly understand how file formats work. But I still did not want to give up; my thirst for knowledge was too great for such a pitiful action. For this reason I spent a great deal of the summer of 2011 in exploring how two different categories of file formats work: image and sound formats. And while it is true that I made progress in exploring how these kinds of formats work, progress was being made too slow. The summer was coming to an end, and I was at the same time desperately trying to come up with what sort of school project I should do for my last term.

And then it suddenly struck me: exploring how one category of file formats works would be a great idea for a school project! So I needed to make a choice: image or sound formats? But the real question was rather: which kind of format



would be the most manageable to explore? Then the choice was simple: image formats, because while it is true that sound formats are very interesting, the math behind them was, and still is, simply beyond me. And while image formats are relatively simple, you still need to have some moderate knowledge from several different areas in order to fully understand them, so a project based on image formats would still be quite challenging. These areas are:

- Information Theory(mainly from the subarea of data compression)
- Color Science

And Color Science was by far the hardest of these two areas to study, which is why only a small segment of this text is dedicated to it, while pretty much the rest is dedicated to the art of data compression.

## 1.2 Abstract

The cover picture of this text was made to illustrate the main point of this text: digital images are just numbers. The cover image is a very simplified model of how all image formats in general are built and I will spend the rest of this section explaining the numbers that this image consists of. I will in other words write a *specification* of the image format that the image on the front page is stored in. This format will from now on be known as *IM*, the imaginary image format.

### 1.2.1 Image Header

The first gray section of the image is known as the image header of the image. It will *always* be found first in an image. Let us go through the parts of the header number by number:

#### Magic Numbers

The first two numbers are the two letters *IM*. Even letters are represented by numbers in a computer. How these letters are mapped to different numbers is determined by the *encoding* of them. An encoding is basically a table that maps all the letters and miscellaneous symbols of the human language to numbers. One very common encoding is known as *ASCII*, and in *ASCII* the letter “I” is represented by the number 73 and the letter “M” is mapped to the number 77. So if these letters were encoded in *ASCII*, a better way to represent them would have been to simply use the numbers 73 and 77. But I did not do this because it is in this case much easier for the reader to understand letters than numbers.

These two first numbers are known as the magic numbers of the file format. At the beginning of every image file, there are a couple numbers called magic numbers that are common to all images of that specific format. They are sort like file name extensions(for example, a file named “essay.doc” has a file name extension “doc”), meaning that they help you identify the type of the file you are currently dealing with.

## Measurements

And following the magic number is the width and the height of this image. In practically every image format this information is included. This information is included to make it easier for image reader programs to parse the image data in the file.

## Metadata

But images can not only contain image data, they can also contain something known as *metadata*. Metadata contains no data that is necessary to understand how the image data within the image is represented, but it rather gives information about things like:

- The program used to create the image
- A descriptive name for the image.
- The the creator of the image.
- The creation date of the image.
- The camera that was used to take the image.

One much used method for embedding metadata about the camera used to take an image is known as EXIF [1]. But since EXIF is a quite complex method for storing such data, we will not discuss it in this text. We will only be concerned in exploring how to extract simple metadata.

### 1.2.2 Color Palette

Color palettes are also very common, but far from all images out in the wild use them, but as good all image formats support them, hence why they are included in this imaginary format. A color palette is basically a list of colors that is to be used by the image. The colors themselves are represented numerically using a so called color model. Color models are something we will discuss in much depth in chapter 5. The numbers in the palette is the numbers that the colors will be represented by in the color data. In other words, the color data in this image does not contain any actual color data, but rather contains indexes to a list of colors known as the color palette. But in real image formats the colors in the palette are not actually explicitly assigned numbers but it is the order in which they appear in the palette that decide which numbers they are assigned.

### 1.2.3 The Color Data

Following the header and the color palette is the actual color data. An image is simply a grid of colors. As you can see on the front cover image, an image really only consists of a lot of colored small squares. These squares as referred to as pixels[2]. Every pixel has *one* and only one color. It is easy to see that even very big images can be broken down to small colored squares.

Since a color palette was used, the image data simply consists of a small sequence of indexes to the palette. If the image did not use a color palette at all,

the image would just have consisted of a grid of raw colors. The colors would in that case have been represented like they were in the palette, using a color model.

#### **1.2.4 Done...?**

But it is not really that simple. A majority of all image formats have had some sort of compression applied to their color data. Compression almost always necessary because raw color data tends to take up a *lot* of space. Compression algorithms are hands down the most difficult part in understanding how image formats work, which is why the majority of this text will be spent on explaining them.

## Chapter 2

# Purpose and problem

So the main purpose of this project is to explore how image formats works. But only the area of image formats itself is a very broad area since there are so *many* of them. Finding out exactly how many image formats there are in existence is a very difficult, almost impossible, task. Imagemagick, an open source project whose main purpose could be said to be to support the maximum amount of image formats, supports well over 100 formats [3]. And according to the *open directory project* there are at this time about 170 different graphics formats [4].

Indeed, there are many different kinds of image formats. And furthermore, image formats are generally divided into two main categories: *vector and raster graphics*[2, 5]. Vector graphics consists of lines, rectangles, Bézier curves and other geometric forms, while raster graphics could simply be described as a grid, matrix, of colors. We will in this project focus only on raster graphics. Some authors, like [2], also refer raster graphics to as *bitmap images*.

But there are also many different raster formats. Yet all of them are very similar in their overall structure. So to make the project even more manageable, we will only cover the three specific raster formats that are known as TGA, GIF and PNG. The simplest formats will be covered first, and the most the complex format will be covered last, meaning that we will explore first TGA, then GIF and last PNG, which is significantly more complex than the former two.

By exploring these 3 formats, we will have gotten a good general view of how image formats are built and structured. So this is main purpose of the project:

To explore how images are digitally, numerically, represented and structured.

And by this we mean that this project will go on until we have fully understood the specifications of the 3 former image formats.

## Chapter 3

# Methods

### 3.1 Tools

To explore how the aforementioned image formats works, I decided to write small programs that would analyze the numbers that images in the formats consists of and dump the raw color. My programming language of choice for this was C++ and my compiler was gcc<sup>1</sup>. For doing the memory debugging that is often necessary in C++ programming, I used Valgrind<sup>2</sup>, which at times was a real life saver. My editor of choice for writing the code was EMACS<sup>3</sup>.

To learn the theory necessary to write these programs I had to read several books and many articles, all of which are cited in the bibliography(see the table of contents).

This document was created and typeset using L<sup>A</sup>T<sub>E</sub>X<sup>4</sup>. I chose to use L<sup>A</sup>T<sub>E</sub>X rather than Word because I, as a programmer, find it a lot easier to use than Word. The L<sup>A</sup>T<sub>E</sub>X source files were also edited in EMACS, using the AUC<sub>T</sub>E<sub>X</sub><sup>5</sup> package.

The body text is typeset in the font T<sub>E</sub>X Gyre Pagella, a clone of Hermann Zapf's Palatino, and the monospace text uses the font LuxiMono.

All the graphics of this document were created using TikZ & PGF<sup>6</sup>

All the source files for both this document and the programs were managed using git<sup>7</sup>, the version control system.

### 3.2 Structure Of This Text

The chapters in this text are laid out as follows:

**Chapter 4** Any preliminary knowledge that is absolutely necessary to understand *any* chapter of this text will be put here. We will also establish all the conventions and glossaries that we will be using throughout the rest of the text in this chapter.

---

<sup>1</sup><http://gcc.gnu.org/>

<sup>2</sup><http://valgrind.org/>

<sup>3</sup><http://www.gnu.org/software/emacs/>

<sup>4</sup><http://www.latex-project.org/>

<sup>5</sup><http://www.gnu.org/software/auctex/>

<sup>6</sup><http://sourceforge.net/projects/pgf/>

<sup>7</sup><http://git-scm.com/>

**Chapter 5** We will in this chapter discuss how color is represented digitally.

**Chapter 6** The main topic of this chapter will be the RLE compression algorithm. This algorithm is used in the TGA format.

**Chapter 7** The TGA format is discussed.

**Chapter 8** The LZW compression algorithm will be explained, which is used in the GIF format.

**Chapter 9** The GIF format will be the subject of this chapter.

**Chapter 10** To ensure the data integrity of PNG images something known as a CRC is used, which is the topic of this chapter.

**Chapter 11** The compression algorithm used in the PNG format, known as Deflate, is a rather complex algorithm that consists of several subalgorithm. One of these subalgorithm is known as Huffman Coding, which is what we will be discussing in this chapter.

**Chapter 12** The second subalgorithm of the Deflate algorithm is known as LZ77, which is the algorithm we will discuss in this chapter. An improvement of LZ77 called LZSS is also discussed in this chapter.

**Chapter 13** In this chapter we will combine LZ77 and Huffman Coding into the final Deflate algorithm.

**Chapter 14** In this chapter the PNG format will finally be discussed.

**Chapter 15** All the former chapters will be discussed and we will attempt to draw a final conclusion from the project. The project as a whole will also be discussed. Answers to the exercises can be found at the end of every chapter.

Small exercises can also be found in the text. They were primarily added so that the reader could have a way of checking that he/she truly has understood the contents of the text.

### 3.3 Sample source code

All the source code of the programs that I wrote during this project can be found at the following address:

`https://github.com/Erkaman/digital-image-formats`.

The entire repository can be cloned by issuing the command

```
git clone git://github.com/Erkaman/digital-image-formats.git
```

The project is divided into several directories. In the code directory the C++ source files for the programs written during the project can be found. The log directory contains the log book of this project. And in the latex directory you can find the L<sup>A</sup>T<sub>E</sub>X source files that were compiled to produce this document. A

---

precompiled PDF-version of this document can be found at the location `latex/project.pdf`. But even if you do not understand anything of the contents of the code folder, do not worry, because this text was written to explain the concepts behind image formats as intuitively as possible, without you having to understand the source files for the programs that demonstrates them. But it is of course not a bad thing if you do understand the source code files in the code directory.

## Chapter 4

# Preliminaries

In this chapter, we will establish all the conventions that will be used throughout this text and explain some basic terms.

### 4.1 Glossary

**binary number** A binary number is simply a number of base 2. For example, the number 139 is represented by the binary number  $1000\ 1011_2$ , since

$$\begin{aligned}1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 &= \\2^7 + 2^3 + 2^1 + 2^0 &= \\128 + 8 + 2 + 1 &= 139\end{aligned}$$

#### Exercise 4.1

Rewrite the following numbers on binary form:

- (a) 4
- (b) 50
- (c) 500

#### Exercise 4.2

Convert the following binary numbers to ordinary numbers of base 10:

- (a)  $1111_2$
- (b)  $1000\ 0000_2$
- (c)  $1111\ 1110_2$

**bit** The separate digits of a binary numbers are often referred to as bits. An  $n$ -bit number is simply a number with  $n$  binary digits.

#### Exercise 4.3



What is the maximum value of a  $n$ -bit number? How many different possible values can a  $n$ -bit number have?

**toggled bit** If we say that a bit is *toggled*, we mean that the bit has a value of 1.

**cleared bit** And controversially, if a bit has the value 0, then we say that the bit is *cleared*.

**bit counting order** The bit that has the lowest value is referred to as bit 0<sup>1</sup>. In the binary number 0100<sub>2</sub>, bit 0 is cleared( it has the value 0), bit 2 is toggled (it has the value 1) and the last bit 3 is cleared.

**lowest bit** Bit 0 will also commonly be referred to as the lowest bit.

**highest bit** Bit  $n - 1$  of a  $n$ -bit number. In other words, the last bit of a binary number. In the number 100<sub>2</sub> the highest bit toggled and the 2 lowest bits are cleared.

### Exercise 4.4

What are the values of bits 0, 3 and the highest bit in the number 01001<sub>2</sub>?

**byte** A byte is simply a binary number with 8 digits/bits. The max value of a byte is  $2^8 - 1 = 255$  and there are  $2^8 = 256$  different values that a byte can have. We will often be referring to bytes throughout the entire text, since, as we soon shall see, an image represented digitally is just a sequence of bytes.

**ASCII** A very common text encoding that will be used a lot in this text. ASCII is a 7-bit encoding and therefore covers 128 different characters. But ASCII values are for the sake of convenience always stored in 8-bit bytes. The entire ASCII table is given in table 4.1[6]. Characters 33–126 are printable characters. Characters 0–32 are on the other hand control characters. These are used to affect how the text is processed. HT (9) is for an ordinary tab while SP(32) represents space. However, many of these control characters are obsolete and are practically never used. The control codes that are actually used in modern files are NUL, HT, LF, CR and SP [7].

The usage of the two special codes CR and LF is something that we need to further discuss. They are used to start a new line in text. But representing newlines turns out to be a surprisingly complex issue. In Windows based operating systems, newlines are represented by a CRLF; that is, a carriage return, CR, followed by a linefeed, LF. Unix based operating systems, like Linux and Mac OS X, simply uses a line feed, LF, to represent newlines. But for Mac OS version 9 and lower, CR was used[8, 9, 10, 11, 12].

If a text file is to be used on only one operating system this will never pose a problem. But if the file is to be shared between computers running on different operating systems, it will. However, this is something that is almost never noticeable to the common user. It is really only of importance to programmers who want their software to flawlessly run on different operating systems.

---

<sup>1</sup>This convention is mainly used because us programmers like to start counting from 0 rather than 1

**Table 4.1** – The ASCII table

Value	Code(Character)	Code Description
0	NUL	Null Character
1	SOH	Start of Heading
2	STX	Start of Text
3	ETX	End of Text
4	EOT	End of Transmission
5	ENQ	Enquiry
6	ACK	Acknowledge
7	BEL	Bell(makes a sound)
8	BS	Backspace
9	HT	Horizontal tab(ordinary tab)
10	LF	Line Feed
11	VT	Vertical tab
12	FF	Form Feed
13	CR	Carriage Return
14	SO	Shift Out
15	SI	Shift In
16	DLE	Data Link Escape
17	DC1	Device Control 1
18	DC2	Device Control 2
19	DC3	Device Control 3
20	DC4	Device Control 4
21	NAK	Negative Acknowledge
22	SYN	Synchronous Idle
23	ETB	End of Transmission Block
24	CAN	Cancel
25	EM	End of Medium
26	SUB	Substitute
27	ESC	Escape
28	FS	File Separator
29	GS	Group Separator
30	RS	Record Separator
31	US	Unit Separator
32	SP	Space
33	!	Exclamation Point
34	"	Quotation Mark
35	#	Number Sign
36	\$	Dollar Sign
37	%	Percentage Sign
38	&	Ampersand
39	'	Apostrophe
40	(	Left Parenthesis
41	)	Right Parenthesis
42	*	Asterisk
43	+	Plus Sign
44	,	Comma
45	-	Minus Sign, hyphen

**Table 4.1** – (continued)

Value	Code(Character)	Code Description
46	.	Period , Dot
47	/	Forward Slash
48	0	0
49	1	1
50	2	2
51	3	3
52	4	4
53	5	5
54	6	6
55	7	7
56	8	8
57	9	9
58	:	Colon
59	;	Semicolon
60	<	Less-than Sign
61	=	Equals Sign
62	>	Greater-than Sign
63	?	Question Mark
64	@	At Sign
65	A	A
66	B	B
67	C	C
68	D	D
69	E	E
70	F	F
71	G	G
72	H	H
73	I	I
74	J	J
75	K	K
76	L	L
77	M	M
78	N	N
79	O	O
80	P	P
81	Q	Q
82	R	R
83	S	S
84	T	T
85	U	U
86	V	V
87	W	W
88	X	X
89	Y	Y
90	Z	Z
91	[	Left Bracket

**Table 4.1** – (continued)

Value	Code(Character)	Code Description
92		Backward Slash
93	]	Right Bracket
94	^	Caret
95	_	Underscore
96	`	Grave Accent
97	a	a
98	b	b
99	c	c
100	d	d
101	e	e
102	f	f
103	g	g
104	h	h
105	i	i
106	j	j
107	k	k
108	l	l
109	m	m
110	n	n
111	o	o
112	p	p
113	q	q
114	r	r
115	s	s
116	t	t
117	u	u
118	v	v
119	w	w
120	x	x
121	y	y
122	z	z
123	{	Left Bracket
124		Vertical Line
125	}	Right Bracket
126	~	Tilde
127	DEL	Delete

**Exercise 4.5**

Convert the following `ASCII` characters to their corresponding `ASCII` values:

- (a) A
- (b) n
- (c) <

**Exercise 4.6**

Convert the following `ASCII` values to their corresponding characters:

- (a) 35
- (b) 122
- (c) 63

### Exercise 4.7

What is always true for the `ASCII` values of uppercase characters(A–Z)?

(Hint: write them out binary)

### Exercise 4.8

How do you convert an uppercase `ASCII` value to its corresponding lower case value?  
How do you do the reverse transformation, lowercase to uppercase?

(Hint: what number has to be added or subtracted?)

**Hexadecimal** We will also be using the hexadecimal numeral system in this text. In hexadecimal the numbers 0–9 are given their usual values, while the letters A–F are assigned to the values 10–15, so that the hexadecimal number  $D3_{16}$  has the value

$$13 \cdot 16^1 + 3 \cdot 16^0 = 13 \cdot 16 + 3 = 208 + 3 = 211$$

### Exercise 4.9

Convert the following hexadecimal numbers to ordinary numbers of base 10:

- (a)  $23_{16}$
- (b)  $FF_{16}$
- (c)  $AA_{16}$

### Exercise 4.10

Convert the following numbers to hexadecimal:

- (a) 3
- (b) 46
- (c) 189

**string** A string is simply a sequence of letters in some encoding. The most commonly used encoding in this text will be `ASCII` [6].

**C string** String as they are represented in the C programming language. In this language, strings are always terminated by the `NUL` character[13]. The `NULL` (alternative spelling of `NUL`) character has, as familiar, a value of 0. This means that the string “eric” will be represented by the sequence of bytes 101, 114, 105, 99, 0 in the C programming language. This is important to know, because in some image formats strings are stored as C strings.

Why the `NULL` character at the end of the string even is necessary is for rather complex reasons that we will not treat in this text.

**file** We are going to be talking a lot about files in this text, so it is important that we as early as possible establish a strict definition for what a file is. A file is just a sequence of bytes. A perfectly valid file could for example consist of the numbers 101, 114, 105, 99. This is a file that consists of the single string “eric”, where the letters use ASCII encoding. However, if you opened this file in a text editor, say notepad, you would only see the letters “eric” and not the numbers that represent the letters. This is because a text editor is programmed to see all the bytes in a file as text. If you on the other hand opened this file in a hex viewer, you would see the file for what it truly is: a sequence of numbers<sup>2</sup>.

But since a byte can only have 256 different values, the reader may wonder how larger numbers are stored in a file. This is simply done by combining bytes. Two bytes in a sequence becomes a 16-bit number with a maximum value of  $2^{16} - 1$ . In the same fashion even larger numbers can be stored.

**offset** When we are talking about an offset we are referring to a position in a file. The offset is zero based. When we are talking about the number at offset 0 in the file 13, 2, 1 we are talking about the number 13. In the same file at the offset 2, the number 1 can be found.

**render** When a program displays an image on the screen this process is known as *rendering*. To render an image is to display an image on the screen.

**display** Synonym for render.

## 4.2 Pseudocode Conventions

Instead of showing code examples in some random programming language, we will be using pseudocode to explain the algorithms in this book. This will keep things as general as possible, and not force the reader into knowing a specific programming language before reading this text.

The pseudocode will be kept as traditional as possible, but we will still need to establish several conventions for it, which is what we are going to do for the rest of the chapter.

### 4.2.1 Boolean Operators

To signify the Boolean operators, or logical operators as we will often also refer them to, we will be using the following symbols:

$\neg$  logical *not*(table 4.2d)

$\wedge$  logical *and*(table 4.2a)

$\vee$  logical *or*(table 4.2b)

Logical truth is represented by  $T$ , and falseness is represented by  $F$ .

---

<sup>2</sup>Do note that in a hex viewer these numbers are, as is implied by the name, shown as hexadecimal numbers

$p$	$q$	$p \wedge q$	$p$	$q$	$p \vee q$	$p$	$q$	$p \otimes q$	$p$	$\neg p$
$T$	$T$	$T$	$T$	$T$	$T$	$T$	$T$	$F$	$T$	$F$
$T$	$F$	$F$	$T$	$F$	$T$	$T$	$F$	$T$	$F$	$T$
$F$	$T$	$F$	$F$	$T$	$T$	$F$	$T$	$T$	$T$	$F$
$F$	$F$	$F$	$F$	$F$	$F$	$F$	$F$	$F$	$F$	$T$
(a) Logical and			(b) Logical or			(c) Logical exclusive or			(d) Logical not	

Table 4.2 – Logical truth tables

## 4.2.2 Bitwise Operators

### Notation

The bitwise operators will also be used in this text. We will use the notation introduced in the C programming language[13] to represent them in pseudocode:

& Bitwise and

| Bitwise or

$\otimes$  Bitwise xor

$\sim$  Bitwise not

$\ll$  Left bit shift

$\gg$  Right bit shift

Notice that we are using  $\otimes$  for representing bitwise xor rather than the traditional C notation  $\wedge$ . This is due to the fact that we would otherwise confuse it with logical and,  $\wedge$ .

What follows is a short introduction to the very simple bitwise operators.

### Bitwise and, or, and xor

Bitwise and is just like logical and, except for the fact that it operates on the bit level. Let us for demonstrative consider the result  $22 \& 12$ . Since bitwise and operates on the bit level we first must convert the two numbers to binary:  $10110_2 \& 01100_2$ . Then the calculation is simply done like this:

$$\begin{array}{r} 10110 \\ \& 01100 \\ \hline 00100 \end{array}$$

So as you can see, the bitwise operators do Boolean logic on the bit level.

### Exercise 4.11

(a)  $2 \& 1$

(b)  $255 \& 23$

(c)  $26 \& 12$

Bitwise or is in the same way logical or on the bit level. Let us perform the former calculation using bitwise or:

$$\begin{array}{r} 10110 \\ | \quad 01100 \\ \hline 11110 \end{array}$$

### Exercise 4.12

(a)  $172 | 52$

(b)  $3 | 3$

(c)  $240 | 15$

Bitwise xor on the other hand, operates on bits by using logical exclusive or. The truth table of logical exclusive or is given in table 4.2c. Using this table, we can easily understand how bitwise xor works:

$$\begin{array}{r} 10110 \\ \otimes \quad 01100 \\ \hline 11010 \end{array}$$

### Exercise 4.13

(a)  $10 \otimes 10$

(b)  $12 \otimes 7$

(c)  $48 \otimes 16$

### Bitwise not

When dealing with bitwise not, it is important that we consider the size of the numbers that we are performing the operation on. If for example  $b = 10$  and the variable  $b$  is of type byte, then it *must* be of length 8 bits:  $b = 0000\ 1010_2$ . What bitwise not does, is that it inverts the number so that all toggled bits get cleared, and all cleared bits get toggled, so  $\sim b = 1111\ 0101_2$ .

Now you should see why it was important that we considered the size of the number. Had the variable  $b$  been of size 4 bits, then  $b = 1010_2$  and then the end result of the operation  $\sim b$  would have been  $0101_2$  instead of  $1111\ 0101_2$ .

### Exercise 4.14

What are the values of the following expressions, if all the numbers are bytes?

(a)  $\sim 11$

(b)  $(\sim 4) \otimes 4$

(c)  $(\sim b) \otimes b$ , for any byte  $b$

(d)  $(\sim b) | b$ , for any byte  $b$

(e)  $(\sim b) \& b$ , for any byte  $b$



**Bitwise shifting**

It is also in bitwise shifting important that we consider the size of the numbers. Bitwise shifting is actually very simple: all the operation  $b \ll n$  really does, is that it shifts the bit pattern in the number  $b$   $n$  steps to the left. For the 4-bit number  $0011_2$ , this means that  $0011_2 \ll 2 = 1100_2$ . But what would have happened if the bit pattern was shifted 3 steps? Then one bit is going fall of the bit boundary and disappear, so  $0011 \ll 3 = 1000_2$ .

And bitwise right shifting works in pretty much the same way, expect for the fact that the bit shifting is done to the right instead of the left, so  $0110_2 \gg 2 = 0001_2$ .

**Exercise 4.15**

- (a)  $1 \ll 0$
- (b)  $1 \ll 1$
- (c)  $1 \ll 2$
- (d)  $3 \ll 1$
- (e)  $3 \ll 2$
- (f)  $3 \ll 3$

Can you express the operation  $N \ll S$ , if the condition  $S \geq 0$  holds, in terms of the arithmetic operators? Exponentiation counts as an arithmetic operator in this exercise. Also, you can ignore the possibility of bits falling of the bit boundary in this exercise.

**4.2.3 Typographical Conventions**

**keywords** will use a **bold** font.

**functions** will be signified by SMALL CAPS.

**variables** can noticed by their *cursive slant*.

**4.2.4 Syntax**

In this section we will discuss the basic syntax of the pseudocode.

The start of a comment is indicated by the symbol  $\triangleright$ .

To assign the value  $n$  to the variable  $var$ , the we use the notation  $var \leftarrow n$

To store a sequence of values we will use arrays. If for example the array  $a$  contains the values the 3, 1, 2 then to access the first value of this array, 3, the syntax  $a[0]$  is used. In general, to access the  $n$ :th value of an array you do  $a[n - 1]$ , since the indexes of arrays are zero-based.

To to go through each value in the array  $a$ , the syntax demonstrated in algorithm 4.1 is used.

---

**Algorithm 4.1** The for each control structure

---

- 1  $\triangleright$ Go through every value  $v$  in the array  $a$ . The variable  $v$  is assigned every element in the array  $a$  in order.
  - 2 **for each**  $v$  **in**  $a$  **do**
  - 3      $\triangleright$ Do something with  $a$  here.
  - 4 **end for each**
-

In algorithm 4.2 the control structure repeat is demonstrated, which is used unconditionally looping a number of times. Prematurely terminating a loop is done with the **break** statement.

---

**Algorithm 4.2** The repeat control structure
 

---

```

1 repeat  $n$  do
2   actions                                ▷ actions are repeated  $n$  times.
3 end repeat

```

---

For functions, we will be using the traditional syntax;  $\text{FUNC}(a, b, c)$  means that we are calling the function  $\text{FUNC}$  with the arguments  $a$ ,  $b$  and  $c$  and that the value of this expression is the return value of the function. To return a value from a function the **return** statement is used.

The function syntax is demonstrated in algorithm 4.3. In this function, Euclid's algorithm is used to calculate the greatest common divisor of two given numbers. [14, 15].

---

**Algorithm 4.3** Euclid's algorithm
 

---

```

1 procedure EUCLID( $a, b$ )
2    $r \leftarrow a \bmod b$ 
3   while  $r \neq 0$  do
4      $a \leftarrow b$ 
5      $b \leftarrow r$ 
6      $r \leftarrow a \bmod b$ 
7   end while
8   return  $b$ 
9 end procedure

```

---

### 4.2.5 Functions

We will be dealing with files in many of these algorithms, so we will need to introduce several functions for handling file operations.

**READBYTE()** It is assumed from the beginning of the algorithm that a file has already been opened for reading. This function reads a byte from that file.

**WRITEBYTE(byte)** At the beginning of every algorithm, we also assume that there is a file opened for output. This function writes a byte to that file.

**ENDOFFILEREACHED()** True if the end the file we are reading from have been reached.

### Exercise 4.16

Make a function  $\text{GETBITS}(b, \text{start}, \text{end})$  that extracts the bit pattern of a number  $b$  from a starting position to a ending position. These positions are be zero-based. Example:

$$\text{GETBITS}(80, 4, 6) = 5,$$

because 80 is represented by the binary number  $0101\ 0000_2$  and from the positions 4 to 6 there is a bit pattern that has a value of  $5(101_2)$ , which is what this function was supposed to extract.

Hint: Use the bitwise operators.

### 4.3 Answers to the exercises

#### Answer of exercise 4.1

- (a)  $100_2$
- (b)  $11\ 0010_2$
- (c)  $1\ 1111\ 0100_2$

#### Answer of exercise 4.2

- (a) 15
- (b) 128
- (c) 254

#### Answer of exercise 4.3

The maximum value of a  $n$ -bit number is  $2^n - 1$ .

Since every binary digit only has two possible values, an  $n$ -bit number has  $2^n$  possible values.

#### Answer of exercise 4.4

1, 1 and 0. Or: toggled, toggled, cleared.

#### Answer of exercise 4.5

- (a) 65
- (b) 110
- (c) 60

#### Answer of exercise 4.6

- (a) #
- (b) z
- (c) ?

#### Answer of exercise 4.7

The 6:th bit is always toggled. This is because the lowest uppercase character, A, has the value 65 and the sixth bit has the value 64.

#### Answer of exercise 4.8

You convert an uppercase `ASCII` value to lowercase by adding 32 to it, since `a - A = 97 - 65 = 32`. To do the reverse transformation, you subtract 32 from the value.

#### Answer of exercise 4.9

- (a) 35
- (b) 255
- (c) 170

#### Answer of exercise 4.10

- (a)  $03_{16}$
- (b)  $2E_{16}$
- (c)  $BD_{16}$

**Answer of exercise 4.11**

- (a) 0
- (b) 23
- (c) 8

**Answer of exercise 4.12**

- (a) 188
- (b) 3
- (c) 255

**Answer of exercise 4.13**

- (a) 0
- (b) 11
- (c) 32

**Answer of exercise 4.14**

- (a) 244
- (b) 255
- (c) 255
- (d) 255
- (e) 0

**Answer of exercise 4.15**

- (a) 1
- (b) 2
- (c) 4
- (d) 6
- (e) 12
- (f) 24

The operation  $N \ll S$  is equivalent to  $N \cdot 2^S$ .

**Answer of exercise 4.16**

```
1 procedure GETBITS( $b, start, end$ )  
2   ▷Calculate the length of the bit pattern  
3    $len \leftarrow end - start + 1$   
4   return  $(b \gg start) \& (\sim(\sim 0 \ll len))$   
5 end procedure
```

The answer is given above. We will now explain this function.

If given the input `GETBITS(80,4,6)` how may we calculate the value 5 from this?

80 is represented by the bit pattern  $0101\ 0000_2$ . First, we will right shift down the pattern  $start = 4$  steps,  $0101\ 0000_2 \gg start$ , resulting in the bit pattern  $0000\ 0101_2$ . Now all that remains to be done is that we need to figure out how construct the bit pattern  $0000\ 0111_2$  from the input values. Once we have figured out how to make this pattern, we can calculate the proper result:

$$0000\ 0101_2 \& 0000\ 0111_2 = 101_2 = 5$$

Using bitwise and in this way to extract bit patterns is a common idiom in the C programming language.

The bit pattern we want to construct is  $0000\ 0111_2$ . We want a sequence of 3 toggled bits from the lowest bit. We can trivially calculate the length of this pattern as

$$end - start + 1 = 6 - 4 + 1 = 3$$

+1 is necessary because the bit positions are zero based.

We now have the length of the pattern. Now we need to figure out how to construct it. The operation  $\sim 0$ , assuming we are dealing with bytes, gets you the pattern  $1111\ 1111_2$ . Then by shifting this pattern  $len$  steps to the left,  $3 \ll 1111\ 1111_2$ , we end up with the value  $1111\ 1000$ . Now, by simply using the bitwise not operation again,  $\sim 1111\ 1000_2$ , we end up with the desired pattern  $0000\ 0111_2$ .

## Chapter 5

# Digital Color

As we stated in the introduction, raster graphics is going to be the main subject of this text. And images represented as raster graphics is simply a grid of colors. Therefore, in this chapter we will discuss how these color are represented.

### 5.1 What is color?

The following introductory discussion on color is based on [16, 17, 18, 19, 20]

Light is composed of tiny particles traveling at different wavelengths. When light of different wavelengths hit our eyes, they are processed so that we perceive these as colors. For example, blue has a wavelength of 400 nm.

Now, how do our eyes *see* these light waves? In our eyes, there are cone cells for perceiving three different kinds of wavelengths of light: red, green and blue. When these cells absorb red, green or blue light, we see color. And when these cells absorb mixed amounts of red, green and blue light, we are able to perceive colors that are mixtures of red, green and blue color.

#### 5.1.1 RGB

Color models are ways of specifying color numerically [21, 20]. A very widely used color model for representing color in computers is `RGB`.

The `RGB` color model is based on how our eyes perceive color, which we discussed in the previous section, so in `RGB` color made by mixing different amounts of red, green and blue. It can be seen in figure 5.1 how the colors cyan, magenta and yellow are achieved in `RGB`. Also note that white is achieved by mixing all of three colors and that black is represented by no color at all. The rest of the colors can be achieved by mixing different amounts of red, green and blue.

Color channels describe the different amounts of the basic colors red, green and blue in a color. The amount of red in a color is described by the red color channel. These are represented by numbers, and how exactly this is done is discussed in section 5.2.

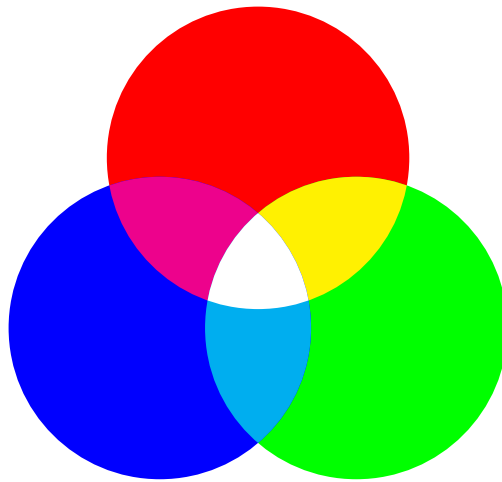


Figure 5.1 – The RGB color model



Figure 5.2 – The color blue with an increasingly lower alpha channel

### 5.1.2 RGBA

But there is actually even more to RGB. There is an extended color model of RGB called called RGBA. In this model, a new channel is added: the alpha channel. This new channel represents the opacity of a color. Opacity is simply the opposite of transparency [22, 2, 23].

Take a good look at figure 5.2. This demonstrates how a color with the same values for its red, blue and green channels gets a lower and lower value for its alpha channel. In the leftmost part of the image the value of the alpha channel is at its maximum, so the color is fully opaque. The more to the right we get in the figure the lower and lower the value of the alpha channel gets, and the more transparent the color becomes. And when the value of the alpha channel is at its minimum the color turns fully transparent.

## 5.2 Color Depth

But up until this point, we have said nothing about how the computer numerically represents these color models. In the following section this is discussed for the RGB color model, based on [2, 16, 23, 24].

### 5.2.1 24-bit Color

#### RGB triplets

In the RGB color model, each color is just a combination of red, green and blue color, so every color can be represented as a triplet  $(R, G, B)$ . Since triplets are

specified by numbers they are quite difficult to imagine visually, but they are on the other hand very easy for the computer to interpret. It can trivially display a triplet on the screen by combining different ammounts of red, green and blue light.

In 24-bit color every color channel is assigned to an 8-bit byte, meaning that every single color will take up 24 bits of storage, since there are 3 color channels. You also refer to 24-bit colors as having a color depth of 24 bits. This is also called the pixel depth of the color.

Using this color depth, the maximum value of a channel will be  $2^8 - 1 = 255$ , so the reddest of red is represented by the triplet (255, 0, 0).

### Exercise 5.1

What color does each of these triplets represent?

- (a) (0, 0, 255)
- (b) (0, 255, 0)
- (c) (255, 255, 0)
- (d) (255, 255, 255)
- (e) (0, 0, 0)
- (f) (0, 255, 255)
- (g) (255, 0, 255)

### Numbers instead of triplets

The way this triplet is stored in a file is simple: since every channel in the triplet is a byte, it is stored as a sequence of 3 bytes.

But you can also see it this way: the triplet is stored as the 24-bit number

$$R \cdot 256^2 + G \cdot 256 + B$$

Here the 8 highest bits of the number store the red channel, the 8 middle bits the green channel, and the 8 lowest bits the blue channel. So the color yellow, (255, 255, 0), would be stored as the 24-bit number

$$255 \cdot 256^2 + 255 \cdot 256 + 0 = 16776960,$$

which is equal to the binary number 11111111 11111111 00000000<sub>2</sub>.

### Channel ordering

As is stated in [2], there is nothing stopping you from storing the color channels in another order than  $R, G$  then  $B$ .  $B, G, R$  is an order that is also commonly used. The specification of the image format usually specifies in which order the color channels are stored, so do remember to read the specification carefully.



**Good enough?**

So how many different colors can you represent using 24-bit color? Every color channel can have  $2^8 = 256$  different values and there are 3 channels; hence,

$$256^3 = 16777216 \approx 16\,000\,000$$

different colors can be represented using only 24-bits.

Then how many different colors can us humans see? Scientists do not really know the answer to that question yet. Popular guesses include one million [25] and  $256^3$  [2].

**5.2.2 RGBA**

Adding alpha channels to this color representation is trivial, just add a fourth channel by adding another 8-bit byte. So **RGBA** color data is in other words stored as a quadruplet  $(R, G, B, A)$ . And as stated before, there is absolutely nothing stopping you from storing these channels in a different order; the  $(A, R, G, B)$  and  $(A, B, G, R)$  orders are also fully acceptable.

**5.2.3 Other Color Depths****48- and 64-bit color**

Other kinds of numbers than 8-bit numbers can be assigned to the separate channels as well. Assigning a 16-bit number to each channel is also common. This is for obvious reasons known as 48-bit color. By adding an alpha channel to this model get 64-bit color.

In 48-bit color the maximum value of a color channel is obviously  $2^{16} - 1 = 65535$ , meaning that the reddest of red is represented by the triplet  $(65535, 0, 0)$

**16-bit color**

A color depth of 16 is also used sometimes. **TGA** format supports 16-bit color [26]. Note that there are actually several ways of representing 16-bit color, but in this section we will only discuss how this is done in the **TGA** format.

In the **TGA** version of 16-bit color, the single highest bit, 15, is used to store transparency information. If this bit is toggled, the color is visible, otherwise it is fully invisible. The **RGB** color channels are given 5 bits of storage each. Bits 0–4 are reserved for blue, 5–9 are given to green, and 10–14 are assigned to red.

Most programming languages do not support reading numbers from a file whose bit lengths are not multiples of 8. It is therefore not possible to directly read the 1-bit and 5-bit numbers that 16-bit color consists of. These numbers have to be extracted using the bitwise operators. In algorithm 5.1 it is shown how to parse 16-bit color.

---

**Algorithm 5.1** Parsing **TGA** 16-bit color

---

- 1  $R \leftarrow \text{GETBITS}(\text{color}, 10, 14)$
  - 2  $G \leftarrow \text{GETBITS}(\text{color}, 5, 9)$
  - 3  $B \leftarrow \text{GETBITS}(\text{color}, 0, 4)$
  - 4  $A \leftarrow \text{GETBITS}(\text{color}, 15, 15)$
-

### 5.2.4 Grayscale Color

In shades of gray the values of the color channels are equal, so the 24-bit grayscale value  $(n, n, n)$  can more succinctly be represented by the single byte  $n$ . Using 8-bit grayscale, the number 43 will in reality represent the triplet  $(43, 43, 43)$ , 0 will represent the color black,  $(0, 0, 0)$ , and 255 is white,  $(255, 255, 255)$ , and so on.

In 8-bit grayscale there are  $2^8 = 256$  different shades of gray. In 1-bit grayscale, which is also quite common, there are only  $2^1 = 2$  different shades of gray: black and white. And stated as generally as possible, this means that in  $n$ -bit grayscale there are  $2^n$  different shades of gray.

Note that grayscale can only be used if all the colors of the image are shades of gray. But if it can be used for an image it should, because it *will* result in huge space savings!

### 5.2.5 Color palettes

And finally, one last way of storing color is by using color palettes. A color palette is an array of colors that is specified at the beginning of the image. These colors are most typically stored as 24-bit RGB triplets. The image data in an image that uses a color palette consists not of a sequence of triplets nor grayscale color, but of a sequence of indexes to the palette.

For example, say an image had the palette *(Red, Purple, Black, Green)*. Then the color Purple in the image would be represented by the number 1. And note that 1 is a *zero based* index to the palette, and not one based. The color data using this palette can use a sequence of 2-bit values to store the color data.

Color palettes are most useful when an image uses a limited subset of colors. Say a 10 by 10 image uses only 4 different colors and that these colors are stored as 24-bit triplets. Then the color data requires  $100 \cdot 3 \cdot 8 = 2400$  bits of storage. If the image on the other hand put these four colors into a palette at the beginning of the image, then these colors could be represented as 2-bit numbers in the image data instead. Then it would only take  $2 \cdot 100 + 4 \cdot 3 \cdot 8 = 296$  bits to store the image. Note that storing the palette requires an extra overhead of 96 bits in this case. So for the usage of a palette to really pay off, the space gained by using a palette has to compensate for the extra overhead of storing it. This has the important consequence that for small images a palette is rarely a good idea.

## 5.3 Color Memory Layout

Now, how are these colors laid out in a typical image file? They tend to be stored in a linear fashion rather than a two-dimensional way. This means that the separate rows of an image are not at all indicated.

Let us for example again consider the image on the front page. The image data of this image would be stored as the sequence

$$4, 4, 4, 4, 0, 4, 4, 3, 4, \dots$$

and so on in an entirely linear fashion. The program reading the image can then split it up into rows using the earlier given measurement values (width:5,

height:6). It should from this be obvious that there is no real need for an image format to split up an image into separate rows.

It is also easy to see how the separate colors can be accessed in this model. Let us say that we stored the color data of the image on the front page in the array *data*. The indexes of this array are zero-based, meaning that, for example,  $data[3] = 4$ , while  $data[4] = 0$ . The index to the array of the color in row  $r$  and column  $c$  (these numbers are also zero-based) is then given by  $r \cdot 5 + c$ . The index of the red color in the middle of the flower is then given by  $2 \cdot 5 + 2 = 12$ . This is also the number you end up with if you start counting (from 0) down from the top left corner of the image row by row to that color.

## 5.4 Answers to the exercises

### Answer of exercise 5.1

- (a) Blue
- (b) Green
- (c) Yellow
- (d) White
- (e) Black
- (f) Cyan(light blue)
- (g) Magenta(purplish red)

## Chapter 6

# Run-Length Encoding

In this chapter, we will discuss the first compression algorithm of this text. First, though, we will need to introduce the reader to some terminology related to data compression, which is what the discussion subject of the following section is going to be.

### 6.1 Some Data Compression Terminology

The following list of terminologies is based on the references [27, 28, 29].

#### 6.1.1 Encoding, Decoding, Compression and Decompression

Writing data on another form is referred to as *encoding*. Converting that encoded data back to its original form is called *decoding*.

In the circumstances of compression techniques, encoding and decoding are also often referred to as *compression* and *decompression*.

Compression could be described as the process of rewriting some data on another form that is more space efficient and decompression is the process that reverses the former process. How the data rewriting process is performed is what the area of data compression is all about.

In this text, we will only treat lossless compression algorithms. Data compressed losslessly can be restored to an identical copy of the original data. In lossy compression on the other hand, data is thrown away when necessary to make the compression even more space efficient. It is for these reasons impossible to restore an original copy of the data for data that has been lossily compressed. It is however fully possible to restore losslessly compressed data.

#### 6.1.2 Compression Ratio

The compression ratio is a very useful measure for when discussing the compression efficiency of a compression algorithm. It is calculated like this:

$$\text{Compression Ratio} = \frac{\text{Compressed Size}}{\text{Original Size}}$$

But it is important to realize that the compression ratio is *not* some constant number that can simply be assigned to a compression algorithm. The compression ratio will always vary between what kind file is to be compressed.

If for example some file originally contained the bytes 10, 10, 10 and after running some arbitrary compression algorithm on it it then contained the bytes 3, 10, then the compression ratio of that algorithm for this *specific file* is  $\frac{2}{3}$ , which basically means that  $\frac{2}{3}$  of the original size was left uncompressed by the algorithm, and that  $\frac{1}{3}$  of the original file size was saved by the compression.

### Exercise 6.1

Calculate the compression ratio for the following cases

- (a) 12, 13, 14, 15, 16, 17 compressed down to 5, 12?
- (b) 12, 14, 16, 18, 20, 22 compressed down to 5, 12, 2?
- (c) 12, 12, 12, 13, 45, 45 compressed down to 2, 12, 0, 13, 1, 45?

Do note that different compression algorithms were used in all three cases. How was compression performed in the three cases?

## 6.2 The Counting Argument

In this section, we will show that a perfect lossless compression algorithm able to compress *any* input file without fail does not exist.

We base the following discussion on [27, 30].

Let us discuss something known as the counting argument. It goes as follows:

No *lossless* compressor can compress all files of size  $\geq N$  bits, for all integers  $N \geq 0$ .

This statement can be proved using surprisingly simple mathematics. Let us first of all assume that such a compressor does indeed exist and check if we find any contradictions. What would such a compressor have to do? It would have to be able to compress down all  $2^n$  files of length  $n$  bits down to files that are *at most*  $n - 1$  bits long. Because if some files were not compressed down to smaller files then the algorithm would no longer perfectly compress all input files. How many possible files are at most  $n - 1$  bits long? This is the sum

$$2^0 + 2^1 + \dots + 2^{n-1} \quad (6.1)$$

If we inspect this sum, then we will see that the quotient between each term is 2, since

$$\frac{2^1}{2^0} = \frac{2^2}{2^1} = \dots = \frac{2^{n-1}}{2^{n-2}} = 2$$

So the sum could also be expressed as

$$\sum_{i=0}^{n-1} 2^i$$

From this we realize that the sum is a simple geometric series! As familiar, such sums are calculated as

$$a + ak + ak^2 + \dots + ak^{n-1} = \sum_{i=0}^{n-1} ak^i = \frac{a(k^n - 1)}{k - 1} \quad (6.2)$$

So the sum (6.1) can from (6.2) simply be computed to

$$\frac{1 \cdot (2^n - 1)}{2 - 1} = 2^n - 1$$

So,  $2^n$  different  $n$  bits files are by the perfect compression algorithm supposed to be compressed down  $2^n - 1$  different files. By the pigeon hole principle, it is impossible for this compression to be lossless, because since  $2^n - 1 < 2^n$  at least two different files will be compressed down to the same file. This simple contraction concludes the counting argument, thus reaching the conclusion that perfect lossless compression is impossible!

But on the other hand, since  $2^n - (2 - 1) = 1$ , then that means that only one file failed to be mapped losslessly to some compressed file. The algorithm can actually be made lossless by mapping this remaining file to some file whose length is at least  $\geq n$ . But since this compression algorithm no longer compresses all input strings it is no longer perfect, but it is now at least lossless!

## 6.3 RLE

### 6.3.1 Description

This following description of the RLE algorithm uses the references [31, 2, 28].

One the most simple compression algorithms is known as *Run-Length Encoding*, abbreviated RLE. The compression performed in the third question of exercise 6.1 is actually RLE compression.

In this algorithm, sequences of length  $n$  of the same value  $b$  are compressed down to  $(n - 1, b)$ . So the sequence  $b, b, b, \dots, b$  of length  $n$  gets compressed down to  $(n - 1, b)$ . The pairs that represents these sequences of repeating values are known as *packets*, and these sequences of repeating values will from known on be known as *runs*. These runs are just sequences of repeating bytes, and the two values in the packets  $n - 1$  and  $b$  are for this reason stored in bytes.

But why is the length byte  $n$  subtracted by 1? A byte can have 256 different values in the range 0 – 255. But since it would be a waste to have a packet of length 0, the designers of the algorithm decided that rather than letting the value zero go to waste, they assigned the length value 0 to 1, 1 to 2 and so on. So while the values stored in the length of the packet are in the range 0 – 255, they actually represent range 1 – 256, which has the added advantage that the max length of a packet is 256 rather than 255.

But here is the main problem with this method: every single run of values, even those for which  $n = 1$ , are considered runs! So even a single run of  $b$  is represented by a packet  $(0, b)$  and the algorithm ends up doubling the size of the original data for such runs! More so, if the data is just a string of runs for which  $n = 1$ , then the “compressed” size of this data ends of up being the double value of the original size. This means that strings like *eric* ends up getting

“compressed” down to 0e0r0i0c, and the compression ratio of this compression is  $\frac{2}{1} = 2$ , which is an absolutely horrible compression ratio.

And even for runs where  $n = 2$  this algorithm does no good. The run  $b, b$  is represented by the pair  $(1, b)$ , and while this at least does not double the size of the data, no compression is performed in this case either.

But for runs for which  $n > 2$  we finally start seeing some results. Then the run  $b, b, \dots, b$  ends up simply being compressed down to  $(n - 1, b)$ , which results in a compression ratio of  $\frac{2}{n}$ . The maximum value of  $n$  is 256, thus the maximum compression ratio of a single run is  $\frac{2}{256} = \frac{1}{128}$ , which is a quite superb compression ratio!

So when data contains a lot of runs for which  $n > 2$ , then this algorithm can indeed do great compression. English text is however not one kind of data that is best compressed by RLE, since there are very few English words where letters are repeated more than two times. True, there do exist plenty of words with double consonants in the English language, but, remember, RLE compression does no compression whatsoever for runs where  $n = 2$ .

But there are also examples of data that could be very efficiently compressed by RLE. One example of this is certain kinds of grayscale data. The page on which this text was printed on could be seen as many long runs of white and black grayscale data. So an image of this page is a good example of data that could get quite efficient compression ratios using RLE.

## Exercise 6.2

Using RLE, compress the following strings and compute the compression ratio:

- (a) AAABBBCCC
- (b) eric
- (c) success

## 6.3.2 Algorithm

### Compression

All of this is essentially trivial to implement in code. Let us first consider the compression algorithm, which is shown in algorithm 6.1. What follows is an explanation of this algorithm.

First we read the first character in the file to the variable  $c_1$ . Then the following character is read to  $c_2$ , but if the file only consisted of the character now stored in  $c_1$ , then we stop and just write out a packet  $(0, c_1)$ .

Else, the two most recently read characters are compared for equality, and if they are equal we have found a run where  $n > 1$ . However, we may just as this comparison is done already be in the process of making a packet and so we need to check for the current packet if  $n < 255$ . This is necessary because  $n$  is stored in a byte and values  $> 255$  can not fit in a byte.

If  $c_1$  and  $c_2$  were not equal or  $n = 255$ , we now need to write out the current packet  $(n, c_1)$ . Now  $c_1$  is set to  $c_2$ , so that we may begin processing the next packet.

You could also see the algorithm like this: The character that is to be repeated by the packet is read to  $c_1$ . To find the length of the packet, characters are

**Algorithm 6.1** Encoding a file using RLE

---

```

1  $length \leftarrow 0$ 
2  $c_1 \leftarrow \text{READBYTE}()$ 
3 while  $T$  do
4    $c_2 \leftarrow \text{READBYTE}()$ 
5   if  $\text{ENDOFFILEREACHED}()$  then
6     break
7   end if
8   if  $c_1 = c_2 \wedge length < 255$  then
9      $length \leftarrow 1 + length$ 
10  else
11    ▷Write the packet
12     $\text{WRITEBYTE}(length)$ 
13     $\text{WRITEBYTE}(c_1)$ 
14     $c_1 \leftarrow c_2$ 
15     $length \leftarrow 1$ 
16  end if
17 end while
18 ▷Write the last packet.
19  $\text{WRITEBYTE}(length)$ 
20  $\text{WRITEBYTE}(c_1)$ 

```

---

repeatedly read to  $c_2$ , which are then compared to  $c_1$ . If  $c_1 \neq c_2$ , then we have found the end of the run, and so the packet is outputted. Then a new character to be repeated is read into  $c_1$  and the process starts anew.

And in a loop this algorithm is repeated until all the characters of the file have been processed. Once we have processed all the characters in the file, we also need to write out the last packet at the end of the algorithm.

**Decompression****Algorithm 6.2** Decoding a RLE encoded file

---

```

1  $length \leftarrow \text{READBYTE}()$ 
2  $c \leftarrow \text{READBYTE}()$ 
3 while  $\neg \text{ENDOFFILEREACHED}()$  do
4   repeat  $length + 1$  do
5      $\text{WRITEBYTE}(b)$ 
6   end repeat
7    $length \leftarrow \text{READBYTE}()$ 
8    $c \leftarrow \text{READBYTE}()$ 
9 end while

```

---

The ridiculously simple RLE decoding algorithm is shown in algorithm 6.2.

It is very simple: pairs of  $(n - 1, b)$  are read in pair by pair. These pairs are then expanded to sequences  $b, b, \dots, b$  of length  $n$  and written to the output file. This process is repeated pair by pair until we have reached to the end the input file.



## 6.4 PackBits RLE

### 6.4.1 Description

The information in the following section is based on the sources [32, 33, 26]

The algorithm that we are about to describe is in [32, 33] known as the PackBits method. But in [26], however, this algorithm is simply known as RLE. I will try and create a compromise between these two names and we therefore will call the method PackBits<sub>RLE</sub> from now on.

As familiar, the RLE algorithm can do great compression on certain kind of data, while it on the other can double the size of other kinds of data, like English text. The question is, can we remove this limitation from the algorithm, while still preserving its great compression ratio for certain kinds of data?

And yes, it turns out that we can. In PackBits<sub>RLE</sub>, the highest bit of the packet length specifier for a packet  $(n - 1, b)$ , is reserved not for specifying the length of the packet, but for specifying the *type* of the packet.

Since only one bit is used for specifying the type of a packet, there can only be two types of packet, and these two types are: run-length packets and raw packets. If the packet is a raw packet, then the highest bit of  $n$  is cleared and if it is a run-length packet then the bit is toggled.

These two packets differ in what kind of data they contain. The run-length packet contains run-length encoded, compressed, data. Run-length packets are used in exactly the same way as they are in plain RLE.

To give an example, the data 5, 5, 5, will in PackBits<sub>RLE</sub> get compressed down to 130, 5. This makes sense because 130 in binary is 1000 0010<sub>2</sub>. The highest bit is here used to specify that the packet is a run-length packet and the 7 lowest bits are used to specify the actual length of the packet minus one.

Because only 7 bits are used to specify the length of a run-length packet, the maximum length of a run-length packet is  $2^7 = 128$ .

But in exchange for a potentially lowered compression ratio, we did get something in return: raw packets. Raw packets essentially is the fix to the problem of potentially doubling the size of “compressed” files that plain RLE had.

The  $n$  value in raw packets specify the length of a following sequence of raw data. This packet type could in other words be used to store strings that RLE is unable to compress, like the data 1, 2, 3, 4 for example. In PackBits<sub>RLE</sub>, the string 1, 2, 3, 4 would most efficiently get encoded as 3, 1, 2, 3, 4. 3 is in binary form 0000 0011<sub>2</sub>, and the since the highest bit is cleared, this means that this is a proper raw packet.

Because PackBits<sub>RLE</sub> lacks the ability to explode the size of the data in the way plain RLE can, its advantage over plain RLE is that it is a lot more *reliable*.

### Exercise 6.3

Compress the following data using PackBits<sub>RLE</sub> and calculate the compression ratio:

- (a) success
- (b) AAABBBCCC
- (c) Suddenly he shouted AAAAAAAAAA

## 6.4.2 Algorithm

### Compression

Let us consider how to implement the actual PackBits<sub>RLE</sub> algorithm. It is shown in algorithm 6.3. We will now explain why this works.

Since there are two kinds of packets allowed, we will need a way of telling which kind of packet we are creating at the time. The two constants *RawPacket* and *RunLengthPacket* are created for this purpose. The values of these two constants are then assigned to the variable *packetType* for keeping track of the current packet type.

We start the algorithm by setting up a starting packet of length 1 of type raw packet. The first character of the file is then read to  $c_1$  and following this the second character in the file is read to  $c_2$ . However, if there is only one character in the file, then the main loop is stopped and the last packet is written. Since we were dealing with a raw packet the only character  $c_1$  is put in the array *data*, which contains the data of the raw packet to be written.

If there however was more than one character, then it is first checked if the current packet is at its maximum capacity. If so, then the current packet is written out. If the current packet is a run-length packet then it is simply outputted and preparations are made for writing a new raw packet. If the packet is a raw packet then the character  $c_1$  is put at the end of the *data* array, and then the raw packet is written out. The next character is read to  $c_2$ , which is why we beforehand assigned  $c_1$  to  $c_2$  after we have written the raw packet, to ensure that the value of  $c_2$  is not destroyed.

If on the other hand the current packet is not full then we are going to write a run-length packet. First however, we need to check if we are not already writing a raw packet. If so, then we finish the raw packet by outputting it. Then the construction of the run-length packet continues by incrementing the *length* variable and setting the current packet type to run-length packet.

If the current packet is not full and  $c_1 \neq c_2$ , then we are writing a raw packet. If we were also writing a run-length packet at this time then we finish it by outputting it.

And when the loop is over, the last, remaining packet is finished and outputted.

### Decompression

The decoding algorithm is thankfully not as complex. It is shown in algorithm 6.4.

**Algorithm 6.3** Encoding a file using PackBits RLE

---

```

1 RawPacket  $\leftarrow$  0
2 RunLengthPacket  $\leftarrow$  1
3 packetType  $\leftarrow$  RawPacket
4 length  $\leftarrow$  0
5  $c_1 \leftarrow \text{READBYTE}()$ 
6 while  $\neg \text{ENDOFFILEREACHED}()$  do
7    $c_2 \leftarrow \text{READBYTE}()$ 
8   if  $\text{ENDOFFILEREACHED}()$  then
9     break
10  end if
11   $\triangleright$ If the current packet is full.
12  if length = 127 then
13    if packetType = RunLengthPacket then
14       $\text{WRITERUNLENGTHPACKET}(\textit{length}, c_1)$ 
15      packetType  $\leftarrow$  RawPacket
16      length  $\leftarrow$  0
17    else if packetType = RawPacket then
18       $\textit{data}[\textit{length}] \leftarrow c_1$ 
19       $\text{WRITERAWPACKET}(\textit{length}, \textit{data}, \textit{out})$ 
20      length  $\leftarrow$  0
21       $c_1 \leftarrow c_2$ 
22    end if
23    else if  $c_2 = c_1$  then
24       $\triangleright$ Finish the current raw packet, if necessary
25      if packetType = RawPacket  $\wedge$  length > 0 then
26         $\text{WRITERAWPACKET}(\textit{length} - 1, \textit{data}, \textit{out})$ 
27        length  $\leftarrow$  0
28      end if
29      length  $\leftarrow$  1 + length
30      packetType  $\leftarrow$  RunLengthPacket
31    else
32      if packetType = RunLengthPacket then
33         $\text{WRITERUNLENGTHPACKET}(\textit{length}, c_1)$ 
34        packetType  $\leftarrow$  RawPacket
35        length  $\leftarrow$  0
36      else if packetType = RawPacket then
37         $\textit{data}[\textit{length}] \leftarrow c_1$ 
38        length  $\leftarrow$  length + 1
39      end if
40       $c_1 \leftarrow c_2$ 
41    end if
42  end while
43  if packetType = RunLengthPacket then
44     $\text{writeRunLengthPacket}(\textit{length}, c_1)$ 
45  else
46     $\textit{data}[\textit{length}] \leftarrow c_1$ 
47     $\text{writeRawLengthPacket}(\textit{length}, \textit{data})$ 
48  end if

```

---

**Algorithm 6.4** Decoding a RLE packbits encoded file

---

```

1 packet ← READBYTE()
2 while ¬ENDOFFILEREACHED() do
3   length ← packet & 127
4   ▷Check if raw packet.
5   if packet & 128 then
6     data ← READBYTE()
7     repeat length + 1 do
8       WRITEBYTE(data)
9     end repeat
10  else
11    repeat length + 1 do
12      data ← READBYTE()
13      WRITEBYTE(data)
14    end repeat
15  end if
16  packet ← READBYTE()
17 end while

```

---

**6.5 Answers to the exercises****Answer of exercise 6.1**(a)  $\frac{1}{3}$ 

The data is just a sequential list of numbers and the difference between all the numbers is 1. This sequence can be compressed down to a pair  $(n, s)$ , where  $s$  is the starting value and  $n$  is the length of the sequence minus 1. This pair can then be decompressed as such:

$$s, s + 1, s + 2, \dots, s + n$$

(b)  $\frac{1}{2}$ 

We are once again dealing with a sequential list of numbers, but in this case the difference between them is 2. If we assign the symbol  $\Delta$  to this difference, then sequence

$$s, s + 1 \cdot \Delta, s + 2 \cdot \Delta, \dots, s + n \cdot \Delta$$

can be represented by the triplet  $(n, s, \Delta)$

(c)  $\frac{6}{6} = 1$ .

The compression ratio is 1, meaning that no compression whatsoever ended up being performed in the long run.

This data was compressed using the RLE algorithm, and we will discuss this algorithm in section 6.3.

**Answer of exercise 6.2**(a) 2A2B2C,  $\frac{2}{3}$ 

(b) 0e0r0i0c, 2

(c) 0s0u1c0e1s,  $\frac{10}{7}$ **Answer of exercise 6.3**

- (a) 6success,  $\frac{8}{7}$
- (b) 130A130B130C,  $\frac{2}{3}$
- (c) 18Suddenly he shouted136A,  $\frac{22}{29}$

## Chapter 7

# Truevision Graphics Adapter

In this chapter, we are to explore the first image format of this text: TGA. But why are we beginning with TGA of all image formats?

For the same reason that the image format first became popular: its specification is very well written; it is freely available without any encumbering patents; and it is one extremely simple image format because the color data is stored in an obvious and intuitive way (in chapter 9 and 14 we will see *many* examples of how unintuitively the color data can be stored) and the data itself is compressed with a very simple compression algorithm: PackBits RLE, which we discussed very much in depth in section 6.4 [2].

However, there are also some more complex features of the TGA format, but we will not cover all of them because very few of them are rarely ever used in the real world. Only the features that are actually used in a majority of TGA images are going to be covered in depth.

The TGA specification is the document [26] and it is going to be the main reference for this chapter.

### 7.1 Building Blocks

The TGA format is built up of things called fields. A field gives information about the image. Information like the name of the image, color depth, the color data of the image and so on. It could really contain any sort of data.

A field also has a specific size. Knowing the size of a field is essential to loading its data, since you cannot just let the computer guess the size of a field. To the computer it is all just a sequence of numbers, so you as the programmer will have to tell the computer how big the respective fields are in order to load them.

Fields are organized into larger units we will call *sections*. We will in this chapter cover these sections one after one and we will describe the fields in the order that they occur in their respective sections. The separate fields are organized in subheadings like this:

**Data(3 bytes)**

This subheading specifies that we will be discussing a field named Data that has a size of 3 bytes.

On the other hand, the subheading:

**Data(bytes)**

Shows that we will be covering a field named Data with a variable size. The color data field is a perfect example of that kind of field; since it contains the color data of the image, it will always vary for images of different widths and height, and therefore the size of this field is variable for different images.

## 7.2 File Header

The first thing that can be found in a TGA image is always the File Header.

**ID Length(1 byte)**

Surprisingly enough, this is one the few image formats without a set of magic numbers. Instead, something known as the ID Length is found. In a later field of the format, there is a field called the image ID and this field specifies the length of that field in bytes.

**Color Map Type(1 byte)**

The TGA format supports color palettes. An alternative term for palette used in the TGA specification is a color map. This field could simply be seen as a boolean flag indicating whether or not the image has a color map; if the value of this field is 1, then the image has a color map and if it is 0, then there is no color map in the image.

**Image Type(1 byte)**

The TGA format supports three different ways of storing color: grayscale, color-mapped color and true-color.

Grayscale color should be familiar to you by now. Color-mapped color obviously means that the color data are indexes to a color-map. True-color simply means that the color uses the RGB or the RGBA color models.

Since the color data can also be PackBits RLE compressed, this all amounts to 6 different kinds of image types. The image type is specified by the value of this field. Table 7.1 shows the number assigned to each image type.

**Color Map Specification(5 bytes)**

The color map specification fields consists of a set of subfields describing the color map included in the image. For images without color maps you can simply ignore these fields.

Image Type	Description
1	Uncompressed, Color-mapped
2	Uncompressed, True-color
3	Uncompressed, Grayscale
9	Compressed, Color-mapped
10	Compressed, True-color
11	Compressed, Grayscale

**Table 7.1** – TGA Image Type values**First Entry Index(2 bytes)**

A color map essentially consists of a sequence of colors. The value of this field is how many of those colors that should be skipped in the beginning of that sequence. This value is most usually 0. And remember, a sequence of two bytes forms a 16-bit number.

**Color map length(2 bytes)**

In this field, the length of the color map is specified. The length is the number of colors in the color map.

**Color map entry size(1 byte)**

Because the color map is just an array of colors, of course all these colors must have a specific color depth. That color depth is specified by this field. This number is specified in bits.

If the color map entry size is  $S$  and the color map length is  $L$ , then the total byte length of the color map is  $\frac{S}{8} \cdot L$ . If furthermore the first entry index is  $I$ , then the number of colors you actually end up loading from the color map will actually be  $\frac{S}{8} \cdot L - I$ , since you in this case skipped  $I$  colors at the beginning of the color map.

**Image Specification(10 bytes)**

Following the color map specification is the image specification. This field contains many values that are essential to loading the image data properly.

**X-origin of Image(2 bytes)****Y-origin of Image(2 bytes)**

These two subfields specify the coordinates of the image on the screen. The values of both of these fields are usually 0.

**Image Width(2 bytes)****Image Height(2 bytes)**

These two fields specify the size of the image in pixels.



Screen destination of first pixel	bit 5	bit 4
bottom left	0	0
bottom right	0	1
top left	1	0
top right	1	1

Table 7.2 – The different image origin combinations

**Pixel Depth(1 byte)**

If the image does not use a colormap then this is color depth of the colors in the image data. If the image uses a color palette then this is the bit size of the indexes that the color data consists of when a colormap is used.

**Image Descriptor(1 byte)**

The image descriptor stores the data of several subfields at different bits in a single byte.

Bits 7 and 6, the two highest bits, are unused and their values should always be set to zero.

Bits 5 and 4 specify the screen destination of the first pixel. The screen destination varies depending on the values of these two bits according to table 7.2. If the image destination is the top left corner, then the image is normally displayed. If it is the top right corner, then the image is mirrored. If it is the bottom left corner, then the image is flipped upside down. If it is the bottom right corner, then the image is mirrored *and* displayed upside down.

The last four bits are the alpha channel bits. These specify how many bits in a separate color is used for the alpha channel. For images without an alpha channel this value will always be zero. For 16-bit images it will be 1, and for 32-bit images it will be 8. 32-bit color uses the ARGB color channel ordering. In section 5.2.3 we discussed how to parse 16-bit color.

## 7.3 Color Data

Following the image header is the color data section. Here the palette, if there is one, and the color data of the image can be found.

**Image ID(bytes)**

This field contains identifying information about the image in form of an ASCII string. It could be described as the name of the image. The ID Length field found in the header describes the size of this field. Since the ID Length field was stored in a byte, the maximum length of this string is 255. But if the ID Length field is 0, this field does not exist at all and should be ignored.

**Color Map Data(bytes)**

The color map can trivially be loaded into an array by loading the same number of colors that was specified in the color map length field, where every color has the bit size specified by the Color map entry size field. The size of every color also indicates the type of the color; if 32-bit color is used, that also means that ARGB is used (the ARGB order is used in TGA rather than RGBA), and 24-bit color simply indicates RGB.

**Color Data(bytes)**

In this field, the color data of the image is stored. The way this data is stored depends on the values stored in the image header. In algorithm 7.1 it is shown how to load the color data.

**Algorithm 7.1** Reading and decompressing the color data of a TGA file

---

```

1  imageSize  $\leftarrow$  imageWidth · imageHeight
2  compressed  $\leftarrow$  imageType = 9  $\vee$  imageType = 10  $\vee$  imageType = 11
3  i  $\leftarrow$  0
4  while i < imageSize do
5      if compressed then
6          packet  $\leftarrow$  READBYTE()
7          length  $\leftarrow$  packet & 127
8          if packet & 128 then
9               $\triangleright$ Read a number of the byte length pixelDepth/8
10             data  $\leftarrow$  READ(pixelDepth/8)
11             if colorMapType = 1 then
12                 data  $\leftarrow$  colorMap[data]
13             end if
14             repeat length + 1 do
15                 Process the color data ...
16                 i  $\leftarrow$  i + 1
17             end repeat
18         else
19             repeat length + 1 do
20                 data  $\leftarrow$  READ(pixelDepth/8)
21                 if colorMapType = 1 then
22                     data  $\leftarrow$  colorMap[data]
23                 end if
24                 Process the color data ...
25                 i  $\leftarrow$  i + 1
26             end repeat
27         end if
28     else
29         data  $\leftarrow$  READ(pixelDepth/8)
30         if colorMapType = 1 then
31             data  $\leftarrow$  colorMap[data]
32         end if
33         Process the color data ...
34         i  $\leftarrow$  i + 1
35     end if
36 end while

```

---

## 7.4 File Footer

TGA images might also contain a file footer. The file footer is mainly used for getting other kinds of data than color data, mostly metadata, about an image. But this data only exists in the image if the file has a File Footer and not all TGA images even have one.

If there is a file footer, then the last 26 bytes of the file constitutes of it. And if the 26 last bytes really is a file footer, then bytes 8-23 for this footer is the ASCII string "TRUEVISION-XFILE". If this is true then these 26 bytes is the file footer of the image; otherwise, the image has no footer in the first place.

Byte 24 of this footer is the ASCII character “.” (a dot) and byte 25 is the has the constant value 0. Only bytes 0-7 in the file footer contain information that is of any significance. They form two fields:

### **Extension Area Offset(4 bytes)**

This 32-bit number gives the offset to the extension area. That is, it gives the position of the extension area, assuming that the very first byte in the file has position 0. We will discuss the contents of the extension area in section 7.6. If this value is zero, then there is no extension area in the file. And in general, if the offset to a section in a file has the value 0, then that section does not exist for that file.

### **Developer Directory Offset(4 bytes)**

This field gives the offset of the less interesting developer area. I will briefly discuss this area section 7.5.

## **7.5 Developer Area**

The developer area allows programmers to add information to the TGA format that is specific to their applications. But this information is basically useless for loading the color data of the TGA image, hence why we will not cover this section at all.

Even if you were to find a TGA image out in the wild that uses this feature, you can only really guess on how you are supposed to use this data, as this is entirely dependent on the program that created the image.

## **7.6 Extension Area**

The extension area is an absolutely *huge* section containing many fields giving numerous kinds of metadata about the image. After reading this section, you should hopefully finally understand what metadata is.

### **Extension Size(2 bytes)**

This field gives the size of the entire extension area, which is always 495 bytes.

### **Author Name(41 bytes)**

This field gives the name of the creator of the image. The last byte of this string is always NULL, so it is a C string. And if the length of the name is less than 40, the rest of the string will be padded with NULL characters. We from now on refer to such a string as a NULL padded C string.

### **Author Comment(324 bytes)**

In this field the author can leave a comment on the image and it is stored as a NULL padded C string.

**Date/Time Stamp(12 bytes)**

This field contains a series of 6 16-bit numbers that are used to store the exact time at which the image was created and saved. The numbers are stored in this order:

1. Month(1-12)
2. Day (1-31)
3. Year (four digits)
4. Hour (0-23)
5. Minute (0-59)
6. Second (0-59)

**Job Name/ID(41 bytes)**

This is used to tie an image with a specific job.

**Job Time(6 bytes)**

This is the time invested in creating the image, as described by the 3 16-bit values:

1. Hours (0–65535)
2. Minutes (0–59)
3. Seconds (0–59)

**Software ID(41 bytes)**

A `NULL` padded C string storing the name of program that was used to create the image. This string could for example be `GIMP` or `Photoshop`(Two image editing programs).

**Software Version(3 bytes)**

The field gives the version of the software that was used to create the image.

**Version Number(2 bytes)**

If *V* is the version of the software that was used to create the image, then this field is `100V`. So if the software that was used to create this image for example had the version 1.23, then the number 123 would get stored in this field.

**Version Letter(1 byte)**

In this field, an `ASCII` letter that is used to indicated the version letter of the software is stored. If the software that created this image was in alpha version when the image was created, then the letter “a” would get stored in this field.

**Key Color(4 bytes)**

The key color can be thought of as a background color. It is the color that is displayed under the rendered image, so to say, when it is shown by an image viewer. This color will be visible only if the transparency feature is used in the image. Under transparent pixels this color will be visible. The key color is stored as a basic 32-bit color in the ARGB order.

**Pixel Aspect Ratio(4 bytes)**

This field consists of the two subfields

**Pixel Width(2 bytes)****Pixel Height(2 bytes)**

If the pixels of the original image were not squares, like pixels should be, then these values are used by image viewer to preserve the appearance of the pixels as they were in the original image.

**Gamma Value(4 bytes)**

This is an interesting value but we will not discuss it in this text because the math behind was simply too difficult, and because in most cases you can simply ignore this value.

**Color Correction Offset(4 bytes)**

This gives the offset to the color correction table. However, this is a feature that practically no TGA images actually use, and we will therefore not discuss this feature in depth.

**Postage Stamp Offset(4 bytes)**

This field gives the offset to the postage stamp image. The postage stamp image is like a small preview of the image stored in the file. The first value in the postage stamp is the width of the postage stamp image and the second value is its height. Following these two values is the postage stamp image. The color data in the postage stamp image is stored in exactly the same way as the full-size image and should therefore be loaded in the same way, as specified in algorithm 7.1.

The postage stamp image is mainly useful when you want to preview a TGA image, without loading the entire full-size image. This may come in use when the full-size image is very big and you just want a small preview of it.

**Scan Line Offset(4 bytes)**

This field provides the offset to the beginning of the Scan Line Table of the image. But the Scan Line Table is practically never used in practice, we will only discuss the Scan Line Table very briefly.

The Scan Line Offset table basically contains a list of offsets. These offsets are offsets to the beginning of every single line, also called scan lines, in the image data.

### Attributes Type(1 byte)

This field specifies which type of alpha channel data is stored in the file. The following are valid values for this field:

- 0 No alpha data is included at all.
- 1 The alpha data is undefined, meaning that it can be ignored.
- 2 Basically same as for 1.
- 3 The alpha data should be used for transparency, meaning that it is normal alpha.
- 4 The color uses pre-multiplied alpha.

But what does that mean? Let the quadruplet  $(A, R, G, B)$  represent a 32-bit ARGB color. How would one specify a half transparent fully red color? One obvious suggestion is  $(\frac{255}{2}, 255, 0, 0)$ . However, using pre-multiplied alpha, the alpha channel gets multiplied into the other color channels, so using pre-multiplied alpha the former quadruplet would instead be represented as

$$\left(\frac{255}{2}, \left(255 \cdot \frac{255}{2}\right) \bmod 255, 0 \cdot \frac{255}{2}, 0 \cdot \frac{255}{2}\right) = \left(\frac{255}{2}, \frac{255}{2}, 0, 0\right)$$

So using pre-multiplied alpha, the quadruplet  $(A, R, G, B)$  is instead represented by the quadruplet

$$(A, AR \bmod M, AG \bmod M, AB \bmod M),$$

where  $M = 2^n - 1$  and  $n$  is the number of bits assigned to one channel. The advantages of pre-multiplied alpha will not be treated in this text, but a brief discussion of this can be found in [22].

# Chapter 8

## LZW

In this chapter, we will discuss the compression algorithm LZW, as it is defined and described in [34, 35, 27, 28, 36]. We will first discuss the history behind the algorithm, then the compression algorithm itself, after this we will describe the decompression algorithm and in the end we will discuss the compression efficiency of this algorithm.

### 8.1 History

In the years 1977 to 1978, the two Israeli researchers Abraham Lempel and Jacob Ziv published two papers; in the first of these papers, [37], they described the algorithm that came to be known as LZ77 (sometimes also known as LZ1) and in the second of these papers, [38], they defined the algorithm that would from then on become known as LZ78 (also known as LZ2) [39, 27, 40].

These two methods started the family of compression algorithms that is known as the *dictionary methods*. Examples of algorithms in the family of dictionary methods is LZMA, which is based on LZ78 [41], LZW, which is also based on LZ78, and LZSS, which was based on LZ77 [27]. LZW is the method that we will discuss in this chapter. It was invented by Terry Welch and he published the algorithm in [42].

### 8.2 The Compression algorithm

The LZW compression algorithm is best understood by walking through its process of compressing some example string, so we will in this example demonstrate how it compresses the string `ababcbababaaaaaa`.

Central to the entire algorithm's process is a string table (also sometimes referred to as a dictionary). In this table, numbers called *codes* are assigned to strings. The algorithm starts by filling this string table with all the strings that are possible to fit in a byte. So all the codes 0–255 are assigned their respective strings. We will for sake of simplicity use ASCII encoding throughout this example, so the codes 0–255 are assigned their respective ASCII characters, as is shown in table 8.1.

Now the actual compression of the string starts. First it reads in the character `a`. In the string table that letter has the code 97, so the code 97 is outputted. Next,



Code	String
...	...
33	!
34	"
35	#
...	...
97	<i>a</i>
98	<i>b</i>
99	<i>c</i>
100	<i>d</i>
101	<i>e</i>
...	...
255	unassigned to in ASCII

Table 8.1 – The initial LZW string table

the character *b* is read, which has the code 98. So 98 is outputted and then the two strings read so far are appended together to form the string *ab*. The string table is searched for appended string, but it cannot be found in the string table, so it is added to the string table and is assigned the next available code, 256.

*a* is discarded, *b* is kept and another *a* is read. First *b* is outputted, and the appended string *ba* is formed. This string does neither exist in the table, so it is added to table and is given the code 257.

Discard *b*, output *a* and read in another *b*. The resulting appended string is this time *ab*. But wait a minute, is not already that string in the string table? Yes it is, so instead of re-adding that string to table, it is kept for the next step in the algorithm.

The code for *ab* is outputted, which is 256, and the character *c* is read. The string *abc* does not exist in the table, so it is added to the string table and given the code 258.

And the algorithm just keeps going on like this. Table 8.2 gives a detailed walkthrough of the entire compression process. So the string *ababcbababaaaaaa* was in the end compressed down to

97, 98, 256, 99, 257, 260, 97, 262, 263, 97

So to summarize, the LZW algorithm basically works like: the algorithm keeps accumulating and reading characters into the string *S*. This process goes on as long as, for the read in character *c*, the string *Sc*, where *Sc* represents the string that is formed when the character *c* is appended to the end of the string *S*, can be found in the dictionary. When *Sc*, for some read in character *c*, cannot be found in the dictionary, then the code for the string *S* is outputted, and the appended string *Sc* is added to the dictionary.

### Exercise 8.1

LZW compress the string *abababab*. Show the output of the compressor and the resulting string table.

String Code	Character Code	Output Code	New table entry
a = 97	b = 98	a = 97	ab = 256
b = 98	a = 97	b = 98	ba = 257
ab = 256	c = 99	ab = 256	abc = 258
c = 99	b = 98	c = 99	cb = 259
ba = 257	b = 98	ba = 257	bab = 260
bab = 260	a = 97	bab = 260	baba = 261
a = 97	a = 97	a = 97	aa = 262
aa = 262	a = 97	aa = 262	aaa = 263
aaa = 263	a = 97	aaa = 263	aaaa = 264
a = 97	STOP	a = 97	STOP

Table 8.2 – Detailed LZW compression process

### 8.3 Implementation of The Compression Algorithm

In algorithm 8.1 the complete LZW compression algorithm is presented. Let us now discuss the details of it.

---

#### Algorithm 8.1 LZW compression algorithm

---

```

1   $maxValue \leftarrow 2^{codeSize} - 1$ 
2   $maxCode \leftarrow maxValue - 1$ 
3  Fill the string table for the codes 1 – 255
4   $nextCode \leftarrow 256$       ▷The code value the next string will be assigned to.
5   $string \leftarrow ReadByte()$ 
6   $character \leftarrow ReadByte()$ 
7  while  $\neg END\_OF\_FILE\_REACHED()$  do
8    if  $IN\_STRING\_TABLE(string + character)$  then
9      ▷Keep accumulating the string
10      $string \leftarrow string + character$ 
11   else
12      $OUTPUTCODE(string)$ 
13     ▷Only add the accumulated string to the table if there is space for it.
14     if  $nextCode \leq maxCode$  then
15        $ADDTOSTRINGTABLE(nextCode, string + character)$ 
16        $nextCode \leftarrow nextCode + 1$ 
17     end if
18      $string \leftarrow character$ 
19   end if
20    $character \leftarrow ReadByte()$ 
21 end while
22  $OUTPUTCODE(string)$ 
23  $OUTPUTCODE(maxValue)$ 

```

---

#### 8.3.1 Codes sizes

A LZW compressed file is just a long sequence of outputted codes. Since all the of the string codes added during the compression step will have code values over

255, bytes will obviously not be sufficient to store these codes. In our version of the LZW algorithm, fixed code sizes are assigned to every code. This means that the codes are stored in fixed  $n$ -bit numbers for some value  $n$ . The most typically used code sizes are in the range  $9 \leq n \leq 15$ .

This is of course not the only possible storage model for the codes. In the GIF format's variation of the LZW algorithm, increasing codes sizes are used to even further improve the compression efficiency of this algorithm. We will discuss this variation in chapter 9.

### 8.3.2 Knowing when to stop

The decompressor has to be notified of when it is done decompressing the data. The largest possible value of a valid code size is therefore outputted in the end of the algorithm. Once the decompressor finds this value, it will know that it is done in decompressing the data.

### 8.3.3 Table size

Since we are using fixed code sizes there must be a limit on how large the string table can grow. For any fixed code size  $n$ , this code size can represent  $2^n$  different codes. However, since the largest possible code is used to signify the end of the compressed data, the maximum length of the string table will be  $2^n - 1$  rather than  $2^n$ .

## 8.4 Description of the Decompression algorithm

Using the compression algorithm LZW on the string ababcbababaaaaaa produced the compressed data

97, 98, 256, 99, 257, 260, 97, 262, 263, 97, 4095

Since we were using 12-bit codes to encode the data, the number  $2^{12} - 1 = 4095$  terminates the data. This number is used by the decompressor to check if it is done with the decompression.

In a LZW compressed file the string table used to encode the file is not included. But how could you possibly decompress the file without such essential information? It turns out that only from the compressed data the decompressor is able to rebuild an exact copy of the string table built during the compression.

The decompressor first assigns the code values 0–255 to their corresponding single length strings, and constructs the same table 8.1 that was also built in the beginning of the compression.

The codes for the characters a and b are read and a is outputted. Now a and b are appended together to form the string ab. This string is added to the string table and is given the code 256.

A new code 256 is read, a is discarded and b is outputted. The code 256 is according to our current string table the string ab.

Now we need to form the next string to be added to the table. To do this, we take the string b and append it to the first character of the string ab, thus forming the string ba. This string is added to the table with the code 257.

The character is read *c*, the old *b* is discarded and the string *ab* is now outputted. *ab* and *c* are appended together to form the string *abc* (The first character of the one length string *c* is simply *c*) and then *abc* is added to the string table with the code 258.

The code 257 is read in, which corresponds to the string *ba*. Since the former code was *c*, the string to be added to the table is *cb*, and this string is given the code 259.

Next the code 260 is read. But we have not even yet added that code to the table! How could this be?

Up to this point, the compressor had compressed the string *ababc* and had defined the code for the string *ba* as 257. What remained to be compressed after this was *bababaaaaaa*. So it read the string *ba* and the character *b*, because *ba* was already defined in the string table. After outputting the code for *ba*, it added the string *bab* to the string table, assigning it the code 260. Then it threw away *ba* and kept *b*. Then it accumulated the string *bab* until it found the character *a*. At this point, the code 260 for the string *bab* was outputted. The important thing to realize here is that this happened *before* the decompressor even had the chance to define the code 260.

So the general problem could be described like this[42]: if the string *kω* is already in the string table and if the string *kωkωk* is encountered, the compressor will first add *kωk* to the string table, and then output *kω*. It will then accumulate and output the code for the string *kωk*, and this is before the decompressor has defined that code in its own string table. However, since this is the only special case in which a undefined code can legally be found, this special case can be handled by instead defining the current string to be *kωk*. This string is equivalent to the string represented by the undefined code.

Now let us return to our example string. The codes we currently had at hand were *ba* and 260. We want to get the characters *k* and *ω*, so that we may construct the string *kωk*. Since *ba* are the first two letters of the string *kωkωk*, then *k* = *b* and *ω* = *a*, so the string represented by 260 is *kωk* = *bab*.

In algorithm 8.2 pseudocode of the algorithm we just described is given.

### Exercise 8.2

LZW decompress 97,98,256,258,98,4095. Codes sizes of 12 are used.

**Algorithm 8.2** LZW decompression algorithm

---

```

1  FillInitialStringTable()
2  nextCode ← 256
3  oldCode ← InputCode()
4  WriteByte(oldCode)
5  character ← oldCode
6  newCode ← InputCode()
7  while newCode ≠ maxValue do
8      if ¬ IsInTable(newCode) then                                ▷The special case
9          string ← TranslateCodeToString(oldCode)
10         string ← string + character
11     else
12         string ← TranslateCodeToString(newCode)
13     end if
14     OutputString(string)
15     character ← string[0]                                        ▷Get the first character of the string.
16     if nextCode ≤ maxCode then
17         ▷Add the translation of oldCode + character to the table
18         AddToStringTable(nextCode, oldCode, character)
19         nextCode ← nextCode + 1
20     end if
21     oldCode ← newCode
22     newCode ← ReadByte()
23 end while

```

---

## 8.5 Compression Efficiency Of LZW

In this section, we will be showing and discussing the results of a test that tested how the compression ratio of the LZW algorithm varied for different sorts of files and data with different codes sizes.

### 8.5.1 How compression is achieved in LZW

The main idea behind LZW compression is that strings that occur often within a file will get replaced by shorter codes. And in general, the longer the algorithm runs, the longer the strings that get added to the table get. This has the consequence that larger files typically have a much better compression ratio than smaller files.

### 8.5.2 The Canterbury Corpus

Here we introduce a set of test files that we will be using to test the efficiency of compression algorithms. A corpus of files designed for testing new compression algorithms called the Canterbury Corpus is that we will be using. These files were selected by [43] out of thousands of other files because they were shown to test the efficiency of lossless compression algorithms the best. Table 8.3 lists all the files in the Canterbury Corpus. They can all be downloaded at [44].

File	Category
alice29.txt	English text
asyoulik.txt	Shakespeare Play
cp.html	HTML source
fields.c	C source
grammar.lsp	LISP source
kennedy.xls	Excel Spreadsheet
lcet10.txt	Technical writing
plrabn12.txt	Poetry
sum	SPARC Executable
xargs.1	GNU manual page

Table 8.3 – The Canterbury Corpus

### 8.5.3 The Test

The purpose of the test was to test for the compression ratio for all the files in table 8.3 for different code fixed sizes  $n$  in the range  $9 \leq n \leq 15$ .

The program that was used to do the compression was LZW, which also is the program that was written to demonstrate the techniques discussed in this chapter. The source code for the program can be find in the directory `code/lzw/`.

To run the program on all the test files and accumulate the gathered data into a  $\text{\LaTeX}$  table, a Python<sup>1</sup> script was written. This script can be found at the location `code/lzw/test.py`.

### 8.5.4 The Test Results

In table 8.4 we show the results of the test.

### 8.5.5 Discussion

Now we will discuss the results gathered in table 8.4 and see if we can draw any sort of conclusion from them.

#### Human Readable Text

We will first discuss the four files `asyoulik.txt`, `alice29.txt`, `plrabn12.txt` and `lcet10.txt`. What these files have in common is that they all contain only human-readable English text. Interestingly enough, if we exclude the file `asyoulik.txt`, they all seem to share a common pattern: larger codes sizes means better compression (the lower compression ratio, the better compression). This makes sense because the English language, and human languages in general, is full of redundancies. So for larger code sizes, larger string tables can be built and therefore a greater number of redundancies can be eliminated from the text.

The largest files, `plrabn12.txt` and `lcet10.txt`, achieved the best compression. This is because for longer files much more of the string table will get built up. The longer a compression goes on, the larger the string table gets. And the larger the string table gets, the better the data can be compressed.

<sup>1</sup><http://python.org/>

File	Original size	9-bit codes(%)	10-bit codes(%)	11-bit codes(%)	12-bit codes(%)	13-bit codes(%)	14-bit codes(%)	15-bit codes(%)
asyoulik.txt	125179	74.2	63.0	54.6	50.3	47.2	45.9	47.0
alice29.txt	152089	68.9	57.9	51.1	47.6	45.0	43.4	43.3
plrabn12.txt	481861	69.8	59.2	53.7	48.8	45.8	44.0	42.6
grammar.lsp	3721	63.9	55.4	52.1	56.9	61.6	66.4	71.1
fields.c	11150	77.7	63.4	52.7	47.7	51.7	55.6	59.6
lct10.txt	419235	75.8	66.8	57.7	52.7	46.0	43.0	40.9
sum	38242	83.6	84.1	79.8	80.9	79.0	57.3	61.4
cp.html	24603	80.2	60.3	52.5	49.7	49.4	53.2	57.0
kennedy.xls	1029744	74.8	82.8	67.5	40.7	33.0	33.0	33.3
xargs.1	4227	75.6	61.1	58.3	63.7	69.0	74.3	79.6

Table 8.4 – LZW compression ratio test results

For the smallest file, `asyoulik.txt`, the best compression ratio was not achieved for the largest code size. This is because for smaller files it is unlikely that the string table even gets fully built up. If the size of the code does not get used to its full potential, then there is a risk that the highest bits of the codes are not used at all (these are used for bigger codes), and these bits may because of this go completely to waste.

### Program code

For the two source code files, `fields.c` and `grammar.lsp`, the compression ratio is not very good at all, compared to the compression ratio of the text files. This is surprising, because source code tends to contain tons of keywords, like `for`, `while`, `return` that are repeated throughout the entire file. You would think that by replacing all these repeated words by small codes this would result in a good compression ratio.

But the worse compression ratio for higher codes sizes is in this case attributed to the very low size of the files. Since these files are so small, it is very hard for them to get a better compression ratio than that of the huge English text files.

### Both text and code

The files `cp.html` and `xargs.1` could be said to contain a combination of both English text and source code. `xargs.1` is a pretty small file and hence gets its maximum compression ratio for the rather small code size of 11. The file `cp.html` is bit larger than the former, but the compression ratio for this file behaves in largely the same way.

### Binary formats

The files `sum` and `kennedy.xls` are files in binary format. That is to say, they consist of a sequence of numbers unreadable by humans. Binary files can most often only be interpreted by the applications that created them. Image formats are examples of binary formats.

The file `kennedy.xls` achieves the best possible compression ratio in this entire test. This is mainly because this is the largest file in the entire Canterbury Corpus.

`sum` is a binary file that has very unstable and unpredictable compression ratios. But it in general seems to achieve the better ratios using the larger codes.

## 8.5.6 Conclusion

The main conclusion of these tests is that for larger files larger code sizes also tend to result in better compression ratios. For smaller files larger codes can on the other hand unnecessarily worsen the compression ratio.

## 8.6 Answers The Exercises

### Answer of exercise 8.1



String Code	Character Code	Output Code	New table entry
a = 97	b = 98	a = 97	ab = 256
b = 98	a = 97	b = 98	ba = 257
ab = 256	a = 97	ab = 256	aba = 258
aba = 258	b = 98	aba = 258	abab = 259
b = 98	STOP	b = 98	STOP

So the string is compressed down to 97, 98, 256, 258, 98.

**Answer of exercise 8.2**

The decompressed data you should get is abababab.

## Chapter 9

# Graphics Interchange Format

The topic of this chapter will be the image format known as Graphics Interchange Format. We will first discuss the rather complicated legal history that the format has and then we will discuss the inner workings of the format. Since many of the features of the TGA format can also be found in the GIF format, we will not be as detailed as we were in chapter 7 when we are describing all the features of the GIF format.

### 9.1 History

The information in the following section is based on the references [42, 39, 45, 46, 5].

#### 9.1.1 Troublesome Patents

In 1987, a company named CompuServe started designing a new image format called *Graphics Interchange Format*, abbreviated GIF. CompuServe chose to use a small variation of LZW for compressing color data in the GIF format.

But at the same time, CompuServe and the rest of the world had no idea that Unisys, the company that Terry Welch was employed by at the time he invented LZW, was pursuing a patent for the LZW compression algorithm. And once Unisys had gained the patent and informed the rest of the world of this, the GIF format had already been released by CompuServe.

But it was not until 1993 that Unisys seriously started pursuing companies that were selling software using the LZW algorithm. And one of those companies that Unisys started attacking because of this was CompuServe. And on 28 December 1994, Unisys and CompuServe finally came to an agreement: all companies writing software that creates or reads GIF images will have to purchase a license from Unisys for using the LZW algorithm. Note that this settlement only affected software developers using the GIF format, and not people distributing GIF images.

Because the GIF format was very popular at the time, this caused an uproar in the graphics developer community. Some of these upset developers formed a group that would develop a new, *free* graphics format that would serve as an

improved replacement to GIF. This new graphics format was PNG, which we shall discuss in chapter 14.

### 9.1.2 Freedom

But since 2006 all LZW related patents have expired. Software developers are in other words finally free to use the LZW algorithm just as they please, without having to request a license from Unisys for doing so.

## 9.2 Format Version

There are actually two version of the GIF format: version 87a, [47], and 89a, [48]. Version 89a is the newer of these two versions and was released in 1989. GIF images that are of version 87a do not support all of the features of version 89a and they are nowadays very rare. And since version 89a is backwards compatible with version 87a, we will only be discussing version 89a in this chapter.

## 9.3 Building Blocks of the format

Let us now go through the building blocks of the GIF format.

### 9.3.1 Blocks

The GIF format is organized into blocks. GIF blocks are very much like TGA sections. The data that blocks consists of is specified in the two kinds number of number types *byte* and *unsigned*. They are defined like this:

**byte** An 8-bit unsigned integer. Range: from 0 to  $2^8 - 1 = 255$ .

**unsigned** A 16-bit unsigned integer. Range: from 0 to  $2^{16} - 1 = 65535$ .

Some blocks are optional, while others must be found in a valid GIF file.

### 9.3.2 Data Sub-blocks

Data Sub-blocks are used to store data that by definition is sequential. An example of such data is the compressed color data, which is just a long sequence codes that were outputted during the compression of the image. Let us go through the parts of a single data sub-block:

#### Block Size(Byte)

At the beginning of every sub-block is its size. Since it is stored in a byte, this has the consequence that a single sub-block has a maximum size of 255.

#### Block Data Values(Bytes)

Following the block size is sequence of bytes. These bytes are the actual data of the sub-block. The number of bytes in a single sub-block is the value of the previous field.

The sub-blocks are read one by one until the block terminator found:

**Block Terminator**

If after reading a sub-block the next byte found is 0, that means that you have found the empty sub-block. This sub-block is also known as the block terminator. Should such a block be discovered, you have reached the end of a sub-blocks sequence.

**An example**

So how would the string *erica* be stored using data sub-blocks? We first let the pair  $(S, D)$  represent a sub-block, where  $S$  is the size of the block and  $D$  is the data of the block. The block terminator is represented by the pair  $(0, )$ . It is a single byte with the value 0.

The string *erica* can then be represented by the sequence of sub-blocks

$$(5, erica)(0, )$$

The block terminator must always terminate a sequence of sub-blocks. But this is not the only way of representing that string;

$$(3, eri)(2, ca)(0, )$$

is another perfectly legal way of doing it. But the first sequence is obviously more space efficient, since only  $1 + 5 + 1 = 7$  bytes are required to store it, while the former sequence requires  $1 + 3 + 1 + 2 + 1 = 8$  bytes of storage.

**9.3.3 Packed Fields**

Another kind of data storage mechanism found in the gif format are packed fields. Packed fields simply means that data is stored in the separate bits of a single byte. We will be using our normal conventions to denote the separate bits, so that in the number  $0001\ 0011_2$  bits 0 to 1 ( $11_2$ ) inclusive have the value 3 while bit 4 has the value 1.

**9.3.4 Extension Blocks**

A special kind of block is an extension block. To make these blocks easier to identify, the extension blocks always begin with the fixed value  $21_{16}$ . This value is also known as the extension introducer. Following the extension introducer is the extension label, which is always stored in a byte. This field identifies the kind of an extension.

After these two bytes you can find a sequence of sub-blocks containing all of the data of the extension block. The data is stored like this to make it easier to skip over the extension block, because the data of some extension blocks is necessary to properly render the image.

**9.4 Header Blocks**

Now we will be describing all the possible kinds of blocks that can occur in a gif file. First of all are the header blocks.

### 9.4.1 Header(Required)

This block will always be found first in a GIF file.

#### Signature(3 bytes)

Identifies the file as a GIF-file. The value of this field will always be the ASCII string "GIF"(non-null-terminated).

#### Version(3 bytes)

Gives the version of the GIF file. This field can only have two possible values: 87a and 89a. Both of these two values are stored as ASCII strings.

### 9.4.2 Logical Screen Descriptor(Required)

Following the header block is always the logical screen descriptor block.

#### Logical Screen Width(Unsigned)

#### Logical Screen Height(Unsigned)

These give the size of the area on the screen where the image will be rendered. So these fields are basically the measurements of the image. However, do note that the GIF format allows for several images to be stored in a GIF file. This makes the GIF format capable of animation. These images can have measurements that are smaller than but not greater than the logical screen measurements.

#### Packed Fields(byte)

This packed field contains data about the global color table of the file. This field will always be found here, even if there is no global color table in the file.

**Bit 7 – Global Color Table Flag** This flag indicates whether there is a global color table in the image.

**Bits 4-6 – Color resolution** This field has a very misleading name. You may think this field is simply used to indicate the color depth of the image, but no, this field indicates the number of bits in the *original image*, the image that was converted to a GIF image, per color channel minus 1.

The GIF format only supports paletted color, and does not offer support for RGB triplets stored straight in the color data. It only permits for a palette of maximum length 256, where every color has color depth of 24 bits. The GIF format can in other words only represent a very small subset of colors. If the original image used more color than the resulting GIF image is capable of representing then information had to be thrown away in the conversion process; colors that looked like each other had to be considered the same to fit all the colors on the smaller palette. This can however result in a significant loss of detail in the resulting GIF.

But since this field gives no rendering information about the image, this field is pretty much useless and you can safely ignore it.

**Bit 3 – Sort Flag** If this flag is set to true, the colors in the global color table are sorted in order of decreasing importance. This means that the most frequently used colors in the image are sorted first in the color table. To cite the GIF specification [48]:

This assists a decoder, with fewer available colors, in choosing the best subset of colors; the decoder may use an initial segment of the table to render the graphic.

So this field may come in use when a GIF image is to be rendered in an environment with a very limited selection of colors, like old monitors. But since the monitors of today support a very large subset of colors, you can for the most part safely ignore this field.

**Bits 0-2 – Color Table Size** This field is used to calculate the size of the global color table. Let  $n$  signify the value of this field, then the *real* size of the global color table is given by

$$\text{Real Color Table Size} = 2^{n+1}. \quad (9.1)$$

### Background Color Index(Byte)

This field's value is the palette index of the background color of the image. Parts of an image that are not covered by a subimage at all are rendered in this color. This is made possible because the GIF format supports arbitrarily positioning of subimages in a GIF image (remember, the GIF format can store several images in a single file.).

### Pixel Aspect Ratio(Byte)

We already discussed this value in chapter 7. To be more specific, we discussed it in section 7.6.

## 9.5 Color Tables

### 9.5.1 Global Color Table(Optional)

After the logical screen descriptor comes the global color table. However, if the Global Color Table flag is set to false the global color table will not occur at all within the image!

The global color table is, just as in any other image format, a long array of colors. Every color is given 24 bits of storage and every color channel color is assigned an 8-bit number. The number of colors in the color table is calculated according to equation 9.1.

### 9.5.2 Local Color Table(Optional)

A separate image in a GIF file can have a local color table, specific only to itself. If no global table can be found in the file, every image in the GIF file must have a local color table.

However, if there exists a global color table, but there also exists a local color table before a subimage, that means that the local color table will be used instead of the global color table for that specific image.

This block is laid out and structured just like the global color table block. Its size is however given by a field in a block we will discuss soon.

## 9.6 Preimage Data Blocks

The preimage data blocks always occur before the color data of a subimage and they help the decoder in properly loading the color data. They also describe how the color data should be rendered.

### 9.6.1 Image Descriptor(Required)

Before the image data there will always be an Image Descriptor. It contains the information you need to know to properly render and process the image data.

#### Image Separator(Byte)

This field identifies the block as an image descriptor and it always has the value  $2C_{16}$ .

#### Image Left Position(Unsigned)

#### Image Top Position(Unsigned)

These fields give the position of the image on the screen. If these two fields both have the value zero, the image is rendered in the top-left corner of the logical screen. If the values are, for instance, 4 and 2, the image is shifted 4 pixels to the right and 2 pixels down before being rendered.

#### Image Width(Unsigned)

#### Image Height(Unsigned)

Specifies the size of the image.

#### Packed Fields(Byte)

**Bit 7 – Local Color Table Flag** This flag indicates whether a local color table is found after this block.

**Bit 6 – Interlace Flag** The row numbers we are using in this explanation will be zero based. So row number 1 is written as row number 0 instead.

This flag indicates whether the image is interlaced. If an image is interlaced, that means the order of the image data is not from the top left corner to the bottom right corner. Rather, the rows of the image data is rearranged into four different passes. The first  $n_1$  rows constitutes of pass 1. These rows are not the rows  $0, 1, \dots, n_1 - 1$ , but instead are the rows  $8 \cdot 0, 8 \cdot 1, \dots, 8(n_1 - 1)$ . So pass 1 is every 8:th row starting from row 0. The rest of the passes are, if we let  $n_x$  represent the number of rows in pass  $x$ :

**Pass 1** Every 8:th row, starting with row 0:  $8 \cdot 0 + 0, 8 \cdot 1 + 0, \dots, 8(n_1 - 1) + 0$

**Pass 2** Every 8:th row, starting with row 4:  $8 \cdot 0 + 4, 8 \cdot 1 + 4, \dots, 8(n_2 - 1) + 4$

**Pass 3** Every 4:th row, starting with row 2:  $4 \cdot 0 + 2, 4 \cdot 1 + 2, \dots, 4(n_3 - 1) + 2$

**Pass 4** Every 2:th row, starting with row 1:  $2 \cdot 0 + 1, 2 \cdot 1 + 1, \dots, 2(n_4 - 1) + 1$

For an image of height 20 rows, the rows would assigned to the passes shown in table 9.1. Here  $n_1 = 3$ , so the rows of pass 1 are 0, 8 and 16.  $n_2 = 2$ , so the rows of pass 2 are 4 and 12.  $n_3 = 5$ , so the rows of pass 3 are 2, 6, 10, 14 and 18.  $n_4 = 10$ , so the rows of the final pass 4 are 1, 3, 5, 7, 9, 11, 13, 15, 17, 19.

But is it guaranteed that this interlacing method will not miss a single row? Yes it is; this follows from the fact that

$$\frac{1}{8} + \frac{1}{8} + \frac{1}{4} + \frac{1}{2} = \frac{4}{4} = 1$$

In algorithm 9.2 it is shown how to uninterlace the interlaced decompressed image data. Since this algorithm is pretty much self-explanatory, I will not include an in depth discussion of it. Reread section 5.3 if you do not fully understand this algorithm.

---

**Algorithm 9.1** Undoing the interlacing of the uncompressed GIF color data

---

```

1  ▷Set the array to the numbers 0, 4, 2, 1
2  startingRow ← {0, 4, 2, 1}
3  rowIncrement ← {8, 8, 4, 2}
4  i ← 0
5  ▷In this algorithm, the pass numbers are zero based
6  pass ← 0
7  while pass < 4 do
8    row ← startingRow[pass]
9    while row < imageHeight do
10     column ← 0
11     while column < imageWidth do
12       j ← row · width + column
13       uninterlacedColorData[j] ← uncompressedColorData[i]
14       i ← i + 1
15       column ← column + 1
16     end while
17     row ← row + rowIncrement[pass]
18   end while
19   pass ← pass + 1
20 end while
```

---

But why is interlacing even necessary? Why can not you just store the pixels in the proper order? Interlacing allows you to get a much better general view of a GIF image while downloading it, because the interlaced rows are more spread out from each than they are in a noninterlaced image. You can then quickly check if you are downloading the right image to begin with, rather than having to wait until the entire image has been downloaded. And that is the only real advantage



Row Number	Interlace Pass
0	1
1	4
2	3
3	4
4	2
5	4
6	3
7	4
8	1
9	4
10	3
11	4
12	2
13	4
14	3
15	4
16	1
17	4
18	3
19	4

**Table 9.1** – GIF interlacing table rows

of interlacing. But with the fast Internet connections of today interlacing is rarely even necessary.

**Bit 5 – Sort Flag** This indicates whether the color data in the local color table is sorted. This works in the exact same way as it does in global color table.

**Bit 3-4 – Reserved** These bits are reserved, meaning that their values are always set to 0. You can simply ignore these bits.

**Bit 0-2 – Size of Local Color Table** If this value is  $n$ , then the actual size of the local color table is  $2^{n+1}$ .

## 9.6.2 Graphic Control Extension(Optional)

The Graphic Control block is an extension block that mostly controls how the animation is done in an animated GIF image.

### Extension Introducer(Byte)

This field identifies the block as an extension block and it always contains the value  $21_{16}$ .

**Graphic Control Label(Byte)**

Identifies the extension block as a graphic control extension block. Contains the fixed value  $F9_{16}$ .

**Block Size(Byte)**

The data in the graphic control extension block is contained in one single sub-block. This is the size of that single sub-block, which is always 4.

**Packed Fields(Byte)**

**Bits 5-7 – Reserved** Unused.

**Bits 2-4 – Disposal Method** When a GIF image is animated, the file essentially contains a sequence of images. When viewing the GIF file, these images are displayed one after one to form an animation. After each image has been displayed you can do different things to dispose of the previous image when current image is to be displayed. The action that will be performed in order to dispose of the former image is specified by this field. The different possible actions are:

- 0 No disposal is specified. This basically means that the decoder does not need to, nor is required to, do anything at all.
- 1 Do not dispose the previous image. The current image is rendered over the previous one. Visible parts of the current image will then cover parts of the previous image. The parts of the previous image that are not covered will still be visible. Remember that the GIF format allows for positioning of images. This is done by the Image Position values in the Image Descriptor block. This is very useful for when you in an animation only want change small parts of an image.
- 2 After the former image has been displayed, the area used by it must be restored to the background color before the next image is to be rendered. The background color was specified by the Background Color Index field of the Logical Screen Descriptor block.
- 3 Restore to previous. Meaning that the decoder must restore the area taken up by the former image and replace it with what was rendered there before the former image was rendered over it. On these areas, the image that came before the former image will be rendered.

**Bit 1 – User Input Flag** If this flag is true, the loading of the next block will continue only after the user has entered some input determined by the application loading the image. One example of such an input is simply the user pressing the Enter key.

But in practice, this field is never used and practically no modern applications in existence support it.

**Bit 0 – Transparency Flag** This flag indicates whether the Transparency Index field should be used.

**Delay Time(Unsigned)**

This specifies the number of centiseconds the decoder should wait after having rendered the image following this block. This is very useful for when making animated GIF images and you want to create a time delay between the frames. A centisecond is a hundredth of a second:

$$1 \text{ cs} = \frac{1}{100} \text{ s}$$

**Transparency Index(Byte)**

If this field is to be used(as is specified by the Transparency Flag.), it contains the color index that should not be rendered at all when encountered. This color can be seen as the transparent color of the color palette

So as you can see from this field, the GIF format supports a very simple form of transparency. But it does not support any form of the alpha color channel. Either a pixel is colored or it is not, there is no state in between.

**Block Terminator(Byte)**

This field terminates the sub-block that this extension block consists of and always has the value 0.

## 9.7 Graphic-Rendering blocks

These blocks contains the data that will actually be visible to the user. One of the blocks contains the color data of the image.

### 9.7.1 Table Based Image Data

Now we have finally reached the most important kind of block that you can find in a GIF file: the table based image data block. Since this block is rather complex, we will explain it in two steps: first we will explain the fields of this block, and then we will explain how to extract the image data from the data given in the fields of this block.

**LZW Minimum Code Size(Byte)**

The GIF format uses a variation of the LZW compression algorithm for storing its color data. The huge difference between the original LZW algorithm and the version used in GIF is that the code sizes of the individual codes increase as bigger codes are needed in the image data. Let the value of this field be  $n$ , then the starting LZW code size is  $n + 1$ .

The variable code sizes fixes the code size problem that the original LZW algorithm had. Since codes of increasing size are used, larger files will compress just as well as smaller files, and no bits will be wasted as they were in plain LZW.

Meaning	Code
Black	0
White	1
Clear Code	2
End Code	3

**Table 9.2** – Example of an initial GIF compression table

### Compressed Image Data(Sub-blocks)

The compressed image data is stored in this sequence of sub-blocks. It is as usual terminated by the empty block.

### The Decompression Algorithm

Now we are to discuss how to decompress the color data using the information given in the two previous fields. The full decompression algorithm is given in algorithm 9.2. As you can see, it is very similar to the original LZW algorithm. Let us now go through the most important differences.

At the beginning the algorithm, all the indexes of the palette are assigned codes in a compression table. The values of these codes are the values of the indexes. Two additional codes are also added to the color table: the clear and the end codes. Let the size of the color palette be  $n$ , then the clear code has the code value  $n$  (the highest index in the palette has the code value  $n - 1$ ), and the end code has the value  $n + 1$ .

If the Clear Code is found, the compression table is reset and restored to its initial state and the codes added to the table during the compression are thrown away and the code size is reset. The clear code can improve compression, because it is not necessary that all parts of a image file looks the same. The codes and strings that were added to the table during the early steps of the compression may prove useless in the later parts of the compression. And if furthermore the table is already filled at these later parts, then the compression will be absolutely horrible, because the compressor will be unable to add strings that improves the compression of the color data in these later sections. Clever GIF compressors will detect when the compression ratio starts falling and spit out clear codes in an attempt to improve the compression ratio.

If the End Code is found, that means that the end of the compressed data has been reached, and that the decompression is done.

If, for example, our color table simply consisted of the colors black and white (in that order), the initial table that would be used in the beginning of the decompression algorithm would have been the table given in table 9.2.

And the second huge difference in the GIF version of the LZW algorithm, is that the code size increases. Remember, the initial code size minus 1 was given in a field of this block. Every time a new code is added to the table, we check if the size of the current code is also equal to the max value of the current code size. If so, and the code size is not equal to its maximum allowable value in the GIF format, 12, then the value of the code size is increased by one.

**Algorithm 9.2** GIF LZW Decompression algorithm

---

```

1 ClearCode  $\leftarrow$  colorTableSize
2 EndCode  $\leftarrow$  colorTableSize + 1
3 IntitalCodeSize  $\leftarrow$  READBYTE() + 1           ▷ Read the initial code size
4 codeSize  $\leftarrow$  IntitalCodeSize
5 ▷ In the beginning of the compressed data a clear code is always found. By
   doing this, we skip the clear code.
6 INPUTCODE(codeSize)
7 ▷ Fill the table with the color palette and the two extra codes
8 RESETCOMPRESSIONTABLE()
9 nextCode  $\leftarrow$  colorTableSize + 2
10 oldCode  $\leftarrow$  INPUTCODE(codeSize)
11 OUTPUTCODE(oldCode)
12 character  $\leftarrow$  oldCode
13 newCode  $\leftarrow$  INPUTCODE(codeSize)
14 while newCode  $\neq$  EndCode do
15   if newCode = ClearCode then
16     ▷ Reset the compression table if a clear code is found
17     RESETCOMPRESSIONTABLE()
18     nextCode  $\leftarrow$  colorTableSize + 2
19     codeSize  $\leftarrow$  IntitalCodeSize
20     oldCode  $\leftarrow$  INPUTCODE(codeSize)
21     OUTPUTCODE(oldCode)
22     newCode  $\leftarrow$  INPUTCODE(codeSize)
23   end if
24   if  $\neg$  IsInTABLE(newCode) then
25     string  $\leftarrow$  TRANSLATE(oldCode)
26     string  $\leftarrow$  string + character
27   else
28     string  $\leftarrow$  TRANSLATE(newCode)
29   end if
30   OUTPUTSTRING(string)
31   character  $\leftarrow$  string[0]
32   if nextCode  $\leq$  ( $2^{12} - 1$ ) then
33     ADDTOSTRINGTABLE(nextCode, oldCode, character)
34     ▷ The maximum code size in GIF LZW is 12
35     if nextCode ==  $2^{codeSize} - 1 \wedge codeSize \neq 12$  then
36       codeSize  $\leftarrow$  codeSize + 1
37     end if
38     nextCode  $\leftarrow$  nextCode + 1
39   end if
40   oldCode  $\leftarrow$  newCode
41   newCode  $\leftarrow$  INPUTCODE(codeSize)
42 end while

```

---

**9.7.2 Plain Text Extension**

The second way of storing rendering data in a GIF file is through the plain text extension. This extension basically allows you to render text as graphic. While

this might have been seen as a good idea at the time it was thought up, nowadays practically no programs implement support for this block. We will for that reason not discuss this block at all.

## 9.8 Miscellaneous Blocks

### 9.8.1 Trailer(Required)

Once we have reached the trailer we have also reached to the end of the GIF file meaning that the decoder's job is done. It is a single field block containing only one value:

#### GIF trailer(Byte)

Identifies the block as the trailer block. Always has the value  $3B_{16}$ .

### 9.8.2 Comment Extension(Optional)

The comment extension block allows you to embed a textual comment into a GIF file. So it could be used to store simple metadata.

#### Extension Introducer(Byte)

Identifies the block as an extension block. Always contains the value  $21_{16}$ .

#### Comment Label(Byte)

This label identifies this extension block as a comment extension block. Contains the constant value  $FE_{16}$ .

#### Comment Data(Data Sub-block)

Containing the comment data is just a sequence of data sub-blocks. The comment data stored using ASCII encoding.

#### Block Terminator(Byte)

This zero-length data block terminates the data sub-blocks and hence the comment data. Has the fixed value 0.

### 9.8.3 Application Extension

The application extension block allows you to give information to a specific application about a GIF file. Because this block is entirely application specific, it may seem pointless to give a thorough discussion of this block. But since it is this block that allows an animated GIF image to be infinitely looped, I will give this block a thorough discussion.

#### Extension Introducer(Byte)

Identifies the block as an extension block. Always has the value  $21_{16}$ .

**Extension Label(Byte)**

Identifies the extension block as an application extension block. Always has the value  $FF_{16}$ .

**Block Size**

The block size of the application extension block. Has the constant value 11.

**Application Identifier(8 Bytes)**

This identifies the application that this block is supposed to affect.

**Application Authentication Code(3 Bytes)**

These values are used to authenticate the application given in the application identifier.

**Application Data(Sub-blocks)**

This sequence of data sub-blocks contains the actual data passed to the application. Different data given in this sequence of data sub-blocks often results in the application reacting differently to this block.

### 9.8.4 Infinite GIFs

Using the application extension we can make an animated GIF repeat its animation infinitely long. Otherwise, the animation would just repeat one time and then stop.

This is done using the Netscape Application Extension. Here the Application Identifier is assigned the string “Netscape” and the Application Authentication Code is given the string “2.0”. The application data of this extension always consists of a single sub-block of length 3. Let us go through the values of this single sub-block: [49]

**Constant(Byte)**

This field does not really have a formal name so I named it constant, because it always has the constant value 1.

**Iterations(Unsigned)**

This number indicates the number of times the animation should be repeated. If this value is 0 the animation will loop infinitely.

The Netscape application extension block was first used by the web browser Netscape. Other software developers then saw that the block was a good idea and started implementing support for the block in their applications, and that is how the use of this block spread. The name Netscape Application Extension has stuck ever since, because you did not want to break old GIF images just because you wanted to make a silly name change [50].

## 9.9 GIF syntax

So there are quite a few kinds of block in the GIF format. But in what order must they come for a GIF file to be considered syntactically valid, by the syntax of GIF files? The syntax of valid GIF files, as given in [48], is illustrated by figure 9.1.



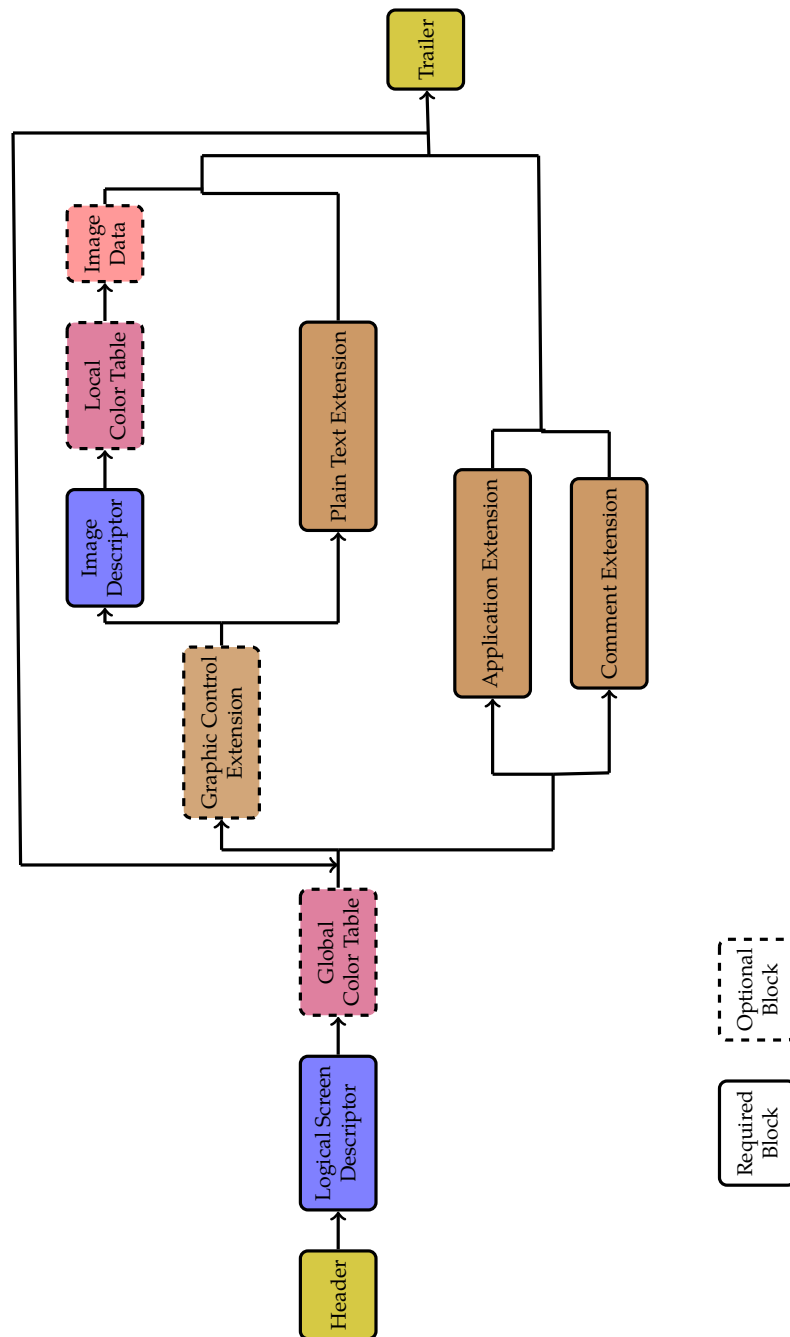


Figure 9.1 – The grammar of a valid gif-file

## Chapter 10

# Checksums

### 10.1 Checksums and error detection

The following introduction to checksums is based on [51, 52, 53, 54].

Say I have an image file on my computer. Now imagine that I, using a hex editor, changed a couple of numbers in the numerical image data of this file. Now the file is no longer represents the original image. But if now the computer attempted loading this image, it would have no way of knowing that the data was fiddled with. If furthermore the image data was compressed, it is even highly likely that it would fail in the decompression of the data, and hence fail the loading of the image.

But is there any for the computer in beforehand to know that I had fiddled with the file? Suppose that before saving this image the program producing this image calculated a check number using all of the numerical data in the image and that this check number was then added at the end of the image data. Then, any program attempting to open the image can as a quick way of checking the validity of the image data recalculate this number on the data and compare it with the check number at the end of the image data. And since I fiddled with the image, the same number would no longer be calculated. In this way the computer could see that the image data was invalid.

But *how* would this number be calculated? Let us say that the image data simply consisted of the bytes 32, 12, 241. What sort of number can we calculate from these? An easy answer would simply be to sum them modulo 256 and add this number as a byte at the end of the data. It is necessary that we do this modulo 256, because otherwise the sum would not be able to fit in a 8-bit byte. This sum would be calculated to  $(32 + 12 + 241) \bmod 256 = 29$ . This number is then appended to the end of the data: 32, 12, 241, 29. The computer could then validate this check number by recalculating this sum. But this system has one big flaw: what if I had changed these numbers to 33, 11, 241, 29 when fiddling with the image data? Because this sum also calculates to 29:  $(33 + 11 + 241) \bmod 256 = 29$ .

Thus, the main problem of this method is that since there are only 256 possible values for a byte, there is a  $\frac{1}{256} = \frac{1}{256}$  chance that an error would go undetected. But on the other hand, there is a  $1 - \frac{1}{256} \approx 99.6\%$  chance that this method would indeed work.

We could of course make this method stronger by giving it a less change of

failure. Instead of storing the sum in a 8-bit number, we could for example store it in a 16-bit number. This would mean a much lower probability of failure to detect errors, namely  $\frac{1}{2^{16}} = \frac{1}{65536}$ . But this does not really fix the original problem; the computed check numbers would still be equal:  $(32 + 12 + 241) \bmod 65535 = 285$  and  $(33 + 11 + 241) \bmod 65535 = 285$ .

Both of these two computed numbers are known as *checksums*. A checksum is simply a value calculated from a sequence of numerical data that is used to verify that the data is error free. Some checksums have been shown to more secure and less prone to not detecting data corruptions, and a CRC is one them:

## 10.2 CRC

One very well used checksum is a CRC, Cyclic Redundancy Check. We will in the following section discuss the computation of a CRC.

The following description of CRC is based on the references [55, 51, 53, 54, 56, 57, 58, 59]. Note that I will not discuss the mathematics behind CRC in depth nor rigorously in this text. I will however assume that the reader is moderately familiar with long division.

### 10.2.1 Polynomial arithmetic

#### Polynomial Representation

Data in a file is simply a sequence of 8-bit bytes. The number 200, for example, is in binary  $1100\ 1000_2$ . However, in the world of CRC, this binary string is instead seen as the polynomial

$$1x^7 + 1x^6 + 0x^5 + 0x^4 + 1x^3 + 0x^2 + 0x^1 + 0x^0 = x^7 + x^6 + x^3$$

So that a bit of value  $b$  in the position  $n$  is represented by  $bx^n$ .

#### Exercise 10.1

What polynomials does the following numbers represent:

- (a) The binary number  $1011\ 0011_2$
- (b) The number 35
- (c) The hexadecimal number  $13_{16}$

#### Exercise 10.2

What binary numbers does the following polynomials represent:

- (a)  $x^7 + 1$
- (b)  $x^3 + x^2 + x + 1$
- (c)  $x^{15} + x^{12} + x^{11} + x^4 + x^3 + x^2 + x + 1$

### Polynomial Width

The width  $w$  of such a polynomial is defined as the value of its highest exponent. The binary number  $110_2$ , represented by the polynomial  $x^2 + x^1$ , has width  $w = 2$ . The width could also be said to be the position of the highest toggled bit. Stated more generally, this means that the polynomial

$$P(x) = a_w x^w + a_{w-1} x^{w-1} + \cdots + a_1 x + a_0$$

has a width of  $w$ , despite the fact that the number it represents is  $w + 1$  bits long.

### Exercise 10.3

What is the width of the polynomial

- (a)  $x^{32} + x^{31} + x^{30} + x^{29} + x^{28} + x^{27} + 1$
- (b) The polynomial represented by the binary number  $1000_2$
- (c) The polynomial represented by the binary number  $101\ 0101\ 1111\ 0000_2$

### Polynomial Addition

Yet another property holds for these polynomials: all operations are done modulo 2. To demonstrate this property, we will introduce the concept of addition. And to make things simpler and cleaner during these explanations, we will write the polynomials as their binary equivalents.

First off we describe addition. While the ordinary relationships

$$0 + 0 = 0$$

and

$$1 + 0 = 0 + 1 = 1$$

hold true, the relationship  $1 + 1 = 2$  does not. Instead, the relationship

$$1 + 1 = 0$$

is true. Why? Because  $(1 + 1) \bmod 2 = 2 \bmod 2 = 0$ .

The sum of the numbers  $1100_2$  and  $0101_2$  can thus be calculated

$$\begin{array}{r} 1100 \\ + 0101 \\ \hline 1001 \end{array}$$

There is another operation for which the above properties also holds true for: bitwise xor. To see why this must be true, replace  $+$  by  $\otimes$  in the above calculations. So bitwise xor and polynomial addition are the exact same things. This discovery is something that will greatly simplify the implementation of CRC.

### Exercise 10.4

Using CRC arithmetic, compute the sum  $1100_2 + 1001_2 + 1111_2$

### Polynomial Subtraction

Polynomial subtraction is defined in the exact way as polynomial addition is, so the following relations hold true:

$$1 - 1 = 0 - 0 = 0$$

and

$$1 - 0 = 0 - 1 = 1$$

So negative numbers wrap in CRC arithmetic.

So given the two polynomials  $P_1(x)$  and  $P_2(x)$  the following will always hold true:

$$P_1(x) + P_2(x) = P_1(x) - P_2(x)$$

Since subtraction is equivalent to addition, this has the important consequence that subtraction is also equivalent to bitwise xor.

### Polynomial Multiplication

CRC multiplication is, on the other hand, defined just like it is in ordinary arithmetic. So  $1100_2 \cdot 1_2 = 1100_2$  and  $1100_2 \cdot 0_2 = 0000_2$  hold true. We will not discuss the multiplication of two operands both longer than one bit, because that is not necessary for our purposes.

### Polynomial Division

Having defined addition, subtraction, and multiplication, we can now define division. Here is the long division of the number  $10011_2$  by  $10_2$ :

$$\begin{array}{r}
 1001 \\
 10 \overline{)10011} \\
 \underline{10} \phantom{00} \\
 00 \phantom{00} \\
 \underline{00} \phantom{00} \\
 01 \phantom{00} \\
 \underline{00} \phantom{00} \\
 11 \phantom{00} \\
 \underline{10} \phantom{00} \\
 1
 \end{array}$$

So  $\frac{10011_2}{10_2}$  is  $1001_2$  with a remainder of  $1_2$ . Stated differently:

$$10011_2 = 1001_2 \cdot 10_2 + 1_2$$

the division operation is easy to understand. In the first step, since the highest bit of the numerator is toggled, the two highest bits of the numerator are subtracted by the denominator,  $10_2 \cdot 1_2 = 10_2$ . In the second step, since the highest bit of the numerator is now cleared, it is divided by a cleared bit pattern

with the same length as the denominator,  $10_2 \cdot 0_2 = 00_2$ . These multiples are then noted at the top. This algorithm keeps going until the bits of the numerator are exhausted. After this the remaining bits are noted as the remainder at the bottom.

### Exercise 10.5

Perform the division  $\frac{11100_2}{11_2}$  using CRC arithmetic.

## 10.2.2 The CRC computation

The polynomial division operation is especially interesting for our purposes because it turns out that the CRC computation is simply a polynomial division.

To compute the CRC of some data, first you need to consider all the data a long sequence of bits. Meaning that, for example, the data that consisted of the numbers 16( $10000_2$ , 5 binary digits) and 2( $10_2$ , 2 binary digits) would form the sequence  $1000010_2$  (We use these rather arbitrary binary number sizes to make the following example easier to follow).

This sequence of binary digits is now to be divided by something known as the *generator polynomial*,  $G(x)$ . The generator polynomial is the key parameter of the algorithm and there are many different to choose from. The only requirement that this polynomial has to satisfy, is that it has to begin and end with a toggled bit. So  $1001_2$  is a valid generator polynomial, while  $0001_2$  is not.

In the following example, we will use the arbitrarily chosen generator polynomial  $1101_2$ . The data that we wanted to compute the CRC of was  $1000010_2$ . First,  $w = 3$  cleared bits are end to the end of the input bit sequence, so it gets transformed to  $1000010000_2$ . Having done all of the preparatory steps, we now perform the final calculation: the binary sequence is divided by the generator polynomial, and the remainder of this division is the CRC:

$$\begin{array}{r}
 \phantom{1101}1110111 \\
 1101 \overline{)1000010000} \\
 \phantom{1101}1101 \\
 \phantom{1101}\hline
 \phantom{1101}1010 \\
 \phantom{1101}1101 \\
 \phantom{1101}\hline
 \phantom{1101}1111 \\
 \phantom{1101}1101 \\
 \phantom{1101}\hline
 \phantom{1101}0100 \\
 \phantom{1101}0000 \\
 \phantom{1101}\hline
 \phantom{1101}1000 \\
 \phantom{1101}1101 \\
 \phantom{1101}\hline
 \phantom{1101}1010 \\
 \phantom{1101}1101 \\
 \phantom{1101}\hline
 \phantom{1101}1110 \\
 \phantom{1101}1101 \\
 \phantom{1101}\hline
 \phantom{1101}011
 \end{array}$$

In this case the checksum is  $011_2$ , or simply 3. Notice that the final bit size of the checksum ends up being equal to the width  $w$  of the generator polynomial. Also note that the computed quotient is not at all interesting for computing the CRC.

### 10.2.3 Generator Polynomials

Authors of texts describing generator polynomials like to write out them as hexadecimal numbers. In this text, we represent the polynomial  $x^3 + x^2 + x^0$  by the hexadecimal number  $d_{16}(= 1101_2)$ . However, since the last highest and lowest bits are mandatory, and could therefore be seen as implicit, several authors like to leave them out. The authors of [60] would for example have written that last polynomial as  $5_{16}(= 0101_2)$ , because they considered the highest bit( $x^3$ ) implicit; on the other hand, the people behind [61] would have written it as  $9_{16}(= 101_2)$ , because they first of all reversed the bit string to  $1011_2$ , and they then chose to simply leave out the lowest bit of the resulting bit string!

There are *many* different CRC generator polynomials used. So many that I quickly realized that it would become a too great of a task to even summarize a few of them. This is mainly because the names of these polynomials tend to be very inconsistent. According to [62] there are around 16 different CRC-16 polynomials! These are polynomials whose width is  $w = 16$ .

The advantages of various generators is very complex issue that we will not cover in this text.

### 10.2.4 Implementation

Now we will finally discuss how to implement the CRC computation in code.

Please inspect algorithm 10.1. It implements polynomial division for generator polynomials of width 8.

---

#### Algorithm 10.1 Computing a CRC of width 8

---

```

1 procedure CRC8(bytes,  $G(x)$ )
2   result  $\leftarrow 0$ 
3   for each byte in bytes do
4     result  $\leftarrow$  result  $\otimes$  byte
5     repeat 8 do
6        $\triangleright 80_{16}$  is the value of the highest bit in a byte.
7       if result  $\& 80_{16}$  then
8         result  $\leftarrow$  (result  $\ll 1$ )  $\otimes G(x)$ 
9       else
10        result  $\leftarrow$  result  $\ll 1$ 
11      end if
12    end repeat
13  end for each
14  return result  $\& FF_{16}$ 
15 end procedure
```

---

Notice first how the numbers that we want to find the CRC for are not even concatenated together to form one large binary number. But the reader will

soon realize that it is not even necessary to do this to calculate the CRC. For a generator polynomial of width  $w = 8$ , 8 cleared bits will be added to the end of the data. It is these last 8 bits that will in the end contain the CRC. These 8 bits are in the algorithm represented by the variable *result*. It is in other words only the remainder we are after, and not the quotient.

So what will happen to the very first byte in the data of this algorithm? Let us first consider what would happen with the byte  $14 = 0000\ 1110_2$ . This bit pattern first gets bitwise xored into the bit pattern of the resulting CRC, *result*. Since the initial value of this variable was set to 0 in the beginning of this algorithm, this simply means that, by the definition of the bitwise xor operator, the *result* variable is set to the bit pattern  $0000\ 1110_2$ .

As familiar from the representation of long division in CRC, if the highest bit of the numerator is cleared, then the numerator is divided by the cleared bit pattern, and then the next bit highest bit of the numerator is dealt with, meaning that the bit pattern is shifted left, which is exactly what is done in this algorithm. This means that after the first four steps in the repeat loop the four highest bits of the *result* variable are set to 1110.

You may now suggest that we simply bitwise xor the generator polynomial into *result*, because if the highest bit of the numerator is toggled then it is subtracted by the generator polynomial. While it is true that we now need to subtract by the generator polynomial, we cannot simply use a bitwise xor operation in order to do this. This would not work, because the length of the generator polynomial is 9 bits, and the bit length of *result* is 8. So this would not work. However, because the highest bit of the polynomial must always be one, you can instead just bitwise left shift *result* one step. This way, the highest bit disappears of *result*, so this has the exact same effect as subtracting would have. Then *result* is bitwise xored with the generator polynomial, to perform the rest of the subtraction. Since the 8:th bit of the generator polynomial is outside the size boundaries of the *result* variable it will not affect this computation, hence why we had to bitwise left shift *result* before doing this.

This general algorithm is repeated for all the 8 bits of the byte. So what this subalgorithm basically does, is that it divides one byte of the data by the generator polynomial and gets the remainder of this operation. For dealing with the next byte, this remainder stored in *result* is bitwise xored with the next byte, and the result of this operation is processed using the repeat loop.

It may be a bit hard to see why this works, so consider the following: we want to compute the CRC for the binary numbers 111 and 011 for the generator polynomial 1100. Using the traditional long division polynomial algorithm, this would be done as such:



$$\begin{array}{r}
 101101 \\
 1100 \overline{) 111011000} \\
 \underline{1100} \phantom{000} \\
 0101 \phantom{000} \\
 \underline{0000} \phantom{000} \\
 1011 \phantom{000} \\
 \underline{1100} \phantom{000} \\
 1110 \phantom{000} \\
 \underline{1100} \phantom{000} \\
 0100 \phantom{000} \\
 \underline{0000} \phantom{000} \\
 1000 \phantom{000} \\
 \underline{1100} \phantom{000} \\
 100
 \end{array}$$

However, using our algorithm, the checksum would be computed in the following way: first we divide 111 by the generator polynomial and compute the remainder:

$$\begin{array}{r}
 101 \\
 1100 \overline{) 111000} \\
 \underline{1100} \phantom{000} \\
 0100 \phantom{000} \\
 \underline{0000} \phantom{000} \\
 1000 \phantom{000} \\
 \underline{1100} \phantom{000} \\
 100
 \end{array}$$

Then, the next byte is subtracted by the remainder

$$100_2 - 011_2 = 111_2$$

The result of this subtraction is divided by the generator polynomial:

$$\begin{array}{r}
 101 \\
 1100 \overline{) 111000} \\
 \underline{1100} \phantom{000} \\
 0100 \phantom{000} \\
 \underline{0000} \phantom{000} \\
 1000 \phantom{000} \\
 \underline{1100} \phantom{000} \\
 100
 \end{array}$$

If we consider this example, then it is easy to see that dividing the bytes by the generator polynomial separately will not affect the end result. And therefore,

algorithm 10.1 works. The advantage that our algorithm has, is that does not require concatenating the numbers into one large binary number, but rather only requires being able to perform the operations on comparatively small numbers.

It also turns out that this algorithm is general enough to allow for *result* to have bit sizes different than 8 bits. Bits outside of the 7:th byte boundary (the highest possible bit position of a byte) will always stay outside this boundary and will not interfere with the 8 lowest bits, those that actually matter to the end result. But since only the 8 lowest bits of the *result* variable are part of the actual CRC, you will need to find a way to only include these bits. This trivially done using the bitwise and operation. By bitwise anding *result* with the pattern  $1111\ 1111_2 (=FF_{16})$ , you can make sure that you only get the 8 lowest bits.

There are many ways of varying the algorithm I just described. It is for example common for *result* to have different initial values than zero. This results in a different CRC being computed. It is also common to use different return values than *result* &  $FF_{16}$ . Many different values can be used instead of  $FF_{16}$ . Our advise is to always read the specification of a CRC *carefully* before starting to implement it.

The algorithm for computing CRC we just described turns out to be far from optimal; there is a table based algorithm that is almost always faster than the rather naïve method we just described. But to keep things simple we will not discuss it here.

Reasoning in very much the same, way can also implement the CRC computation for generator polynomials of width 32, as is shown in algorithm 10.2. As you can see, it is largely the same algorithm.

---

**Algorithm 10.2** Computing a CRC of width 32

---

```

1 procedure CRC32(bytes,  $G(x)$ )
2   result  $\leftarrow$  0
3   for each byte in bytes do
4     result  $\leftarrow$  result  $\otimes$  (byte  $\ll$  24)
5     repeat 8 do
6       if result &  $80000000_{16}$  then
7         result  $\leftarrow$  (result  $\ll$  1)  $\otimes$   $G(x)$ 
8       else
9         result  $\leftarrow$  result  $\ll$  1
10      end if
11    end repeat
12  end for each
13  return result &  $FFFFFFFF_{16}$ 
14 end procedure

```

---

### 10.2.5 CRC-32

We will end our discussion on CRC by briefly discussing the CRC that is used in the PNG format as it is defined in [63]. The CRC computation algorithm is given in algorithm 10.3. The generator polynomial has a width of 32, so the CRC calculated by this algorithm is a 32-bit number. As can be seen, the generator polynomial used is  $EDB88320_{16}$ . But this number represent the polynomial written out in

reversed order with the highest bit left out, so the polynomial represented by this number is in fact

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

The peculiar thing about this CRC is that rather than working from the highest bit to the lowest, this one works from the lowest bit to the highest. For this reason, bitwise right shifts are used rather than left shifts, and the lowest bit is checked rather than the highest.

---

**Algorithm 10.3** CRC computation for the PNG format

---

```

1 procedure CRCPNG(bytes)
2    $G(x) \leftarrow \text{EDB88320}_{16}$ 
3    $result \leftarrow \text{FFFFFFFF}_{16}$ 
4   for each byte in bytes do
5      $result \leftarrow result \otimes byte$ 
6     repeat 8 do
7       if  $result \& 01_{16}$  then
8          $result \leftarrow (result \gg 1) \otimes G(x)$ 
9       else
10         $result \leftarrow result \gg 1$ 
11      end if
12    end repeat
13  end for each
14  return  $result \& \text{FFFFFFFF}_{16}$ 
15 end procedure

```

---

### 10.3 Adler-32

Another example of a checksum is Adler-32. It is for example used to verify data integrity in the ZLIB format [64]. The ZLIB format is of interest to us because that is the container format used to contain the color data in PNG images. It was named after its inventor Mark Adler, one of the creators of the ZLIB format.

Let us now describe Adler-32 based on the references [64, 65, 66].

The algorithm to calculate the checksum is very simple: two variables  $a = 1$  and  $b = 0$  are set in the beginning. Then, for every byte  $D_i$  in the data the checksum is to be computed for,  $a$  is set to  $a + D_i$  modulo the constant 65521. This constant is the largest prime number smaller than  $2^{16}$ .  $b$  is then set to  $a + b$  modulo the same constant. This process is repeated for every byte  $D_i$  in the data. Once every byte has been processed, the 16 highest bits of the resulting checksum are set to  $b$  and the 16 lowest bits are set to  $a$ . Hence, the Adler-32 checksum is always a 32-bit number, as is indicated by its name. In other words, for some data that has a length of  $n$  bytes the checksum is calculated as follows:

$$P = 65521$$

$$a = 1 + D_1 + D_2 + \cdots + D_n \pmod{P} = 1 + \sum_{i=1}^n D_i \pmod{P}$$

$$\begin{aligned} b &= (1 + D_1) + (1 + D_1 + D_2) + \cdots + (1 + D_1 + D_2 + \cdots + D_n) \pmod{P} = \\ &= n + \sum_{i=1}^n (D_i \cdot (n + 1 - i)) \pmod{P} \end{aligned}$$

$$\text{adler32} = (b \ll 16) \mid a = (b \cdot 2^{16}) + a$$

The last line may however need some explanation. It was back in exercise 4.15 on page 24 shown that  $b \ll 16$  is equivalent to  $b \cdot 2^{16}$ . Bitwise or is equivalent to plain addition in this case only because  $b$  and  $a$  were constantly taken modulo 65521, meaning that they can never be longer than 16 bits. Hence, it is impossible for the separate bits in the numbers  $a$  and  $b$  to interfere with each other in the bitwise or operation, and therefore bitwise or is equivalent to plain addition in this case.

It is also shown in algorithm 10.4 how to calculate the Adler-32 checksum.

---

**Algorithm 10.4** Computation of the Adler-32 checksum
 

---

```

1 procedure ADLER32(data)
2    $a \leftarrow 1$ 
3    $b \leftarrow 0$ 
4   for each  $d$  in data do
5      $a \leftarrow a + d \bmod 65521$ 
6      $b \leftarrow a + b \bmod 65521$ 
7   end for each
8   return  $(b \ll 16) \mid a$ 
9 end procedure

```

---

### Exercise 10.6

Compute the Adler-32 checksum for the data 42, 43, 67.

## 10.4 Answers To The Exercises

### Answer of exercise 10.1

- (a)  $x^7 + x^5 + x^4 + x + 1$
- (b)  $x^5 + x + 1$
- (c)  $x^4 + x + 1$

### Answer of exercise 10.2

- (a) 1000 0001<sub>2</sub>
- (b) 1111<sub>2</sub>
- (c) 1001 1000 0001 1111<sub>2</sub>

**Answer of exercise 10.3**

- (a) 32
- (b) 3
- (c) 14

**Answer of exercise 10.4**

$1010_2$

**Answer of exercise 10.5**

$$\begin{array}{r}
 1011 \\
 11 \overline{) 11100} \\
 \underline{11} \phantom{00} \\
 01 \phantom{00} \\
 \underline{00} \phantom{00} \\
 10 \phantom{00} \\
 \underline{11} \phantom{00} \\
 10 \phantom{00} \\
 \underline{11} \phantom{00} \\
 1
 \end{array}$$

**Answer of exercise 10.6**

1848105, because

$$a = 1 + \sum_{i=1}^3 D_i (\text{mod } 65521) = 1 + 42 + 43 + 67 (\text{mod } 65521) = 153$$

$$b = 3 + \sum_{i=1}^3 (D_i \cdot (3 + 1 - i)) (\text{mod } 65521) = 3 + 42 \cdot 3 + 43 \cdot 2 + 67 = 282$$

$$\text{adler32} = (b \cdot 65536) + a = 1848105$$

## Chapter 11

# Huffman Coding

### 11.1 History

In 1951, professor Robert Fano gave his students in a course on information theory two choices: to complete the course they would either have to write a term paper or take a final exam. In the term paper the students were asked to solve a seemingly simple problem. In order to not discourage his students, Fano did not tell them that the problem that he gave them was a problem that he and Claude Shannon, the father of information theory, had been unable to solve [67].

One of Fano's students was David Huffman. For months he tried to solve the problem and he was in the end very close to giving up. But then, in a moment of insight, he suddenly realized how to solve the problem. He then published his solution to the rest of the world in [68]. The technique he described in this paper is known as *Huffman Coding*, and this is the compression method that we will be describing in this chapter.

And luckily, Huffman decided not to patent the algorithm. And because of this, Huffman Coding proved to be a very popular compression technique. It is for example widely used in high-definition televisions and modems.

### 11.2 Code Lists

The following description of Huffman Coding is mainly based on Huffman's original description of the algorithm, [68], but also on [27, 28, 69, 70, 71, 14, 72].

The key observation behind Huffman Coding is this: in any non-random data there will almost always be some characters that are more frequent than others. For example, in English text the by far most common letter is "e" [73, 74]. So if a random letter is picked from an English language text then it is most likely that that letter will be an "e". So, what Huffman Coding does is that it assigns more frequent characters to binary numbers with less digits. These numbers are referred to as *codes*. The collection of all these computed codes, each assigned to a character, will henceforth be known as the *code list* of the input data.

Letter	Probability	Code List 1	Code List 2
A	0.35	$01_2$	$01_2$
B	0.35	$11_2$	$00_2$
C	0.15	$001_2$	$010_2$
D	0.15	$000_2$	$101_2$

Table 11.1 – Examples of code lists

### 11.2.1 The Prefix Property

Before we consider how the Huffman Algorithm chooses these codes, we also have to consider what properties the code list should satisfy. Are all possible code lists acceptable?

Let us say that we are dealing with the four-length alphabet A, B, C, D and that for the sake of argument the probabilities of these letters are the ones given in table 11.1. In this table two possible code lists are given. Let us first consider the first of these code lists. Character that are more frequent are in this code list assigned shorter codes, which is just the property that we are after. Using this code the string BABACD would get encoded as

$$11_2 01_2 11_2 01_2 001_2 000_2$$

This encoded data can be decoded perfectly fine by translating the codes to their corresponding letters. However, what would have happened if we had used code list 2 instead?

$$00_2 01_2 00_2 01_2 010_2 101_2$$

The first four codes can get decoded perfectly fine, but what about the fifth? After reading the first two bits of this code we end up with the code  $01_2$ , which is the code for A. However, how would the encoder know that it is supposed to read the next bit and then parse it as the code for C? It can't. The fifth code in combination with the sixth,  $010101_2$  can be parsed as CD, but an equally valid way to decode this data would be as AAA. So the second set of codes is *ambiguous* while the first is not. Since the encoded data can be parsed in several ways using an ambiguous code, a file compressed using such a code is not lossless in the first place!

The reason that the first code list is not ambiguous is because it has the *prefix property*. This means that no code in the code list is the prefix of another code. On the other hand, in the code list 2, the code  $01_2$  is a prefix of the code  $010_2$ , and therefore the code list was ambiguous.

### Exercise 11.1

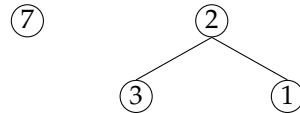
Which of the following code lists has the prefix property? Also, suggest of a way to fix the code lists that do not obey by the prefix property.

- (a)  $011_2, 0101_2, 11_2, 001_2$
- (b)  $01_2, 00_2, 10_2, 101_2$
- (c)  $1010101_2, 00001_2, 001_2, 1011_2$

## 11.3 Huffman Coding

### 11.3.1 Trees

A tree is a recursive data structure that is used to implement Huffman Coding. A tree consists of a collection of nodes with values, and every node can have left and right child nodes. Two example trees:



The first of these two trees is a single node with a value of 7. It has no left or right child nodes. The node with the value 2 has two child nodes. The left and right child nodes have the values 3 and 1 respectively. You can also say that the node with the value 2 is *parent* of the two nodes with the values 3 and 1.

### 11.3.2 Computing the Code List

Let us consider finding the Huffman codes for the letters of the string ababaacdd for the alphabet a, b, c, d. First the letters and their respective frequencies are assigned single nodes:

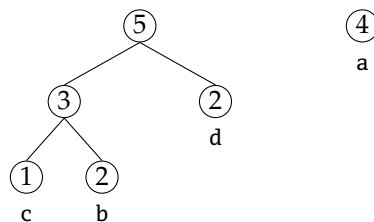


We now have forest of single nodes. The frequencies of the two nodes with minimum frequencies are now summed. This sum is assigned to a new node and this node is made the parent of the two former minimum frequency nodes:



Note that the order in which these two nodes are added does not matter. It is perfectly acceptable to switch b and c. Another alternative for the second minimum frequency node to b was d, but neither this choice will affect the validity of the resulting code list.

The last step is now repeated, meaning that the two minimum trees are added to form an even bigger tree:

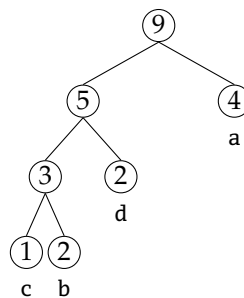




Letter	Code
a	$1_2$
b	$001_2$
c	$000_2$
d	$01_2$

Table 11.2 – Computed code list

And the same thing is done for the two remaining trees:



The tree we end up with is known as the Huffman tree of the input data. But it is important to realize that this is *not* by any means unique. This is because there will always be the possibility that there are more than two trees with a minimum frequency.

To finally create the code list, the following is done: starting from the root of the tree, it is followed along its branches to reach its leaves (the bottommost nodes). If a left branch is followed then a cleared bit(0) is added to the resulting code, else a toggled bit(1) is added.

So to get the resulting code for b, we follow the tree from its root to the leaf that has character b. To reach b we go left, left and then right, so b has the code  $001_2$ . The final code list is given in table 11.2.

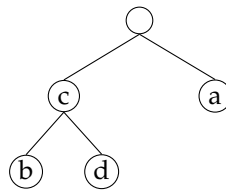
But what if the frequency of one the characters above would have been 0? Then that character would have been left out from the Huffman tree computation, since there is no need to compute the Huffman codes for characters that do not at all occur in the input data!

### Exercise 11.2

Give an example of another valid Huffman that could have been constructed using the input data. What final code list does this tree instead result in?

### 11.3.3 Motivation

Huffman Coding will in general always give valid a code list that has the prefix property. A tree *not* satisfying that property would have to look something like this:



Since the *c* node is parent of the nodes *b* and *d*, its code will also be a prefix of the codes of *b* and *d*. However, since only the leaves of the resulting tree are assigned characters, it is impossible for such a tree to result from Huffman Coding.

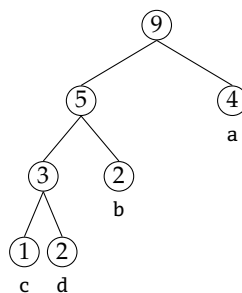
It is easy to see how the algorithm assigns shorter codes to more frequent character. Since the most frequent characters will have higher frequencies, they will also get added to the tree at the later parts of the algorithm, and so they get placed higher in the tree, and therefore they will get assigned the shorter codes. And similarly, since the least occurring characters will get assigned small frequencies, they will also get added into the tree early, so will get placed deep down in the tree, and they will therefore get assigned the longer codes. But we will also mention that in David Huffman's original paper, [68], a much more mathematically rigorous approach is taken to prove these properties. Readers who are not convinced by the above argument are therefore strongly recommended reading that paper.

## 11.4 Answers to the exercises

### Answer of exercise 11.1

- (a) Has the prefix property.
- (b) Does not have the prefix property. If the codes are on the other hand changed to  $01_2$ ,  $00_2$ ,  $101_2$  and  $100_2$  they will obey the prefix property.
- (c) Has the prefix property.

### Answer of exercise 11.2



This tree results in the codes  $a=1_2$ ,  $b=01_2$ ,  $c=000_2$  and  $d=001_2$ .

## Chapter 12

# LZ77 and LZSS

As familiar from section 8.1, LZ77 was the first of the two algorithms described by Lempel and Ziv that sparked the family of dictionary methods. We have partially discussed already LZ78, since LZW was based on it, and we will in this chapter discuss LZ77. But we will also discuss LZSS, which is an improvement of LZ77 that was developed by Storer and Szymanski in 1982.

### 12.1 LZ77

The following description of the LZ77 algorithm is based on [27, 28, 75].

#### 12.1.1 Compression

The core parts of the LZ77 algorithm are the so-called window and lookahead buffers. In the window buffer already compressed data is stored. The lookahead buffer contains the data that has not yet been compressed and matched. Strings in the lookahead buffer are in the algorithm matched for already processed strings in the window buffer. If it finds a matching string for the string in the lookahead buffer, it outputs a *token*. The token contains the information that is necessary for the decoder to locate the matched string in the window buffer.

Now let us go through a simple example that demonstrates how LZ77 operates. Let us study the LZ77 compression of the string *she sells sea shells on the sea shore*. But we will first have to decide on the size of the lookahead and window buffers. Since these buffers have to be stored in memory they obviously cannot be infinitely big. The size of the window determines how long back in the previously compressed data we can look for string matches. The size of the lookahead buffer determines how long strings we can match for in the window buffer. This number does not need to be that big since it is usually comparatively small strings that get matched for in the window buffer. For this example, we give the window buffer a size of 20 characters, while the lookahead buffer is given a length of 10 characters.

In the initial part of the algorithm, the lookahead buffer is filled to its maximum capacity with characters. The window buffer is at this point completely empty, since no characters have yet been processed:

	she_sells_s_
--	--------------

 $\Rightarrow (0, 0, s)$ 

Since the window buffer is empty no matching string can be found in it. So the token  $(0, 0, s)$  is simply output. This is how LZ77 signals that no match at all was found in the window buffer. We will soon describe what the first two numbers in the token describe. The compressor then keeps outputting these single character tokens:

s	he_sells_s_s
---	--------------

 $\Rightarrow (0, 0, h)$   
 $\vdots$   

she_sell	s_sea_shel
----------	------------

 $\Rightarrow (0, 0, s)$ 

The LZ77 compressor does not bother in matching single character strings. While characters keeps getting read to the lookahead buffer, processed characters get added to the window buffer. After the token for the last character *s* has been outputted, interesting things finally start happening; the next three characters to be found in the lookahead buffer are *\_se*, and this sequence can also be found in the window buffer. How far back in the window buffer does this match begin? If we start counting from the end of the window buffer, then we have to go 6 (this count is one-based) characters *back* in the buffer to reach the match. This number is formally known as the *offset* of the token, which is the first number of a token. The second number in a token represent the *length* of the matching string. Since the matching string, *\_se*, has length of 3, this will also be the second number of the token. The third part of the token will get set to the character following the matching string. Thus, the resulting token will be:

she_sells	_sea_shell
-----------	------------

 $\Rightarrow (6, 3, a)$ 

And after this yet another string can be matched for:

she_sells_sea	_shells on
---------------	------------

 $\Rightarrow (4, 2, h)$ 

If there are several matches in the window buffer, then obviously the longest match will be picked. And in this fashion the rest of the compression proceeds. Once the end of the input data has been, reached the lookahead buffer will start shrinking as the remaining characters are compressed, while the size of the window buffer is managed so that it never rises above its fixed size value.

It is easy to see why LZ77 works. Consider for example the following extract from “Alice in Wonderland”:

‘Then you should say what you mean,’ the March Hare went on.

‘I do,’ Alice hastily replied; ‘at least—at least I *mean* what I say—that’s the same thing, you know.’

*'Not the same thing a bit!' said the Hatter. 'You might just as well say that "I see what I eat" is the same thing as "I eat what I see"!''*

*'You might just as well say,' added the March Hare, 'that "I like what I get" is the same thing as "I get what I like"!''*

*'You might just as well say,' added the Dormouse, who seemed to be talking in his sleep, 'that "I breathe when I sleep" is the same thing as "I sleep when I breathe"!''*

Italicized in this extract are the longest parts of this text that will get caught by the LZ77 compressor and get represented more efficiently as tokens. (Not all strings that will get caught by LZ77 are italicized, only the longest ones are.) Human language texts tend to be chock-full with repeated words, no matter how well written they are, so they compress especially well under LZ77. And the same thing could be said for simple images.

### 12.1.2 Decompression

A LZ77 compressed file will consist of a sequence of LZ77 tokens, so the only thing we really need to discuss is how to decode the separate tokens.

A token where the values of the offset and length fields are set to 0 is represented by the character in its third field.

Decoding a token that represents a matched string is not much trickier. A token was encoded by finding a matching string in the window buffer, and then outputting the offset and length of the matching string. So the decoding step is simple: first, go back the number of characters in the so far compressed data indicated by the offset. Now add the character you are currently positioned at to the decompressed data (this character is in fact the first character of the matched string). And now, since a new character was added, the size of the decompressed data will have increased by one. Now you can by again going back offset steps in the decompressed data reach the second character of the matched string. This process is then repeated until a string with the length indicated by the second field of the token has been decoded. This whole process is outlined in algorithm 12.1. Once this has been done, the third field of the match token is also added to the decompressed data.

---

#### Algorithm 12.1 Decoding a LZ77 token

---

```

1  ▷length is the length of the so far decompressed data.
2  repeat tokenLength do
3    decompressed[length] ← decompressed[length – tokenOffset]
4    length ← length + 1
5  end repeat
```

---

### 12.1.3 Token Sizes

But we have up until this point entirely omitted a discussion of the bit sizes of the three fields of LZ77 tokens. The bit size of the third field is the easiest; since it must store an unmatched byte it has the bit size 8. Deciding the size of the offset part is a bit harder. This field must have a bit size such that it has the capacity to

specify the offset to *any* matched string in the window buffer. A window buffer of size  $2^{10} = 1024$  will therefore require the use of a 10-bit binary number for specifying the offset. You get this value by taking the binary logarithm,  $\log_2$ , of the window size:

$$\log_2 2^{10} = 10 \log_2 2 = 10$$

Where the  $\log_2$  is defined as [76]

$$\log_2 x = \frac{\log_b x}{\log_b 2} \quad (12.1)$$

for *any* base  $b$ .

### Exercise 12.1

Prove equation (12.1)

For window sizes that can not be written on the form  $2^n$  then the above reasoning does not hold, but it needs an extension. For a window size of 20 5-bit numbers are required for specifying offsets, since  $2^4 = 16$  and  $2^5 = 32$ . But  $\log_2 20 \approx 4.32$ , so, if we let  $\lceil x \rceil$  denote the floor operation, then correct bit size is calculated as

$$\lceil \log_2 20 \rceil = 5$$

So for a window size of  $w$ , binary numbers of the bit length

$$\lceil \log_2 w \rceil$$

are used for specifying offsets. And the bit size of the length field is defined in very much the same way.

#### 12.1.4 Performance Issues

It is quite common for LZ77-based compressor to use huge window buffers, up to, maybe even larger than, 30000 characters of size are not at all uncommon. It is important to realize that searching through such a huge window for matches is not by any means a computationally cheap operation, because the window has to be searched through many, many times during the compression.

Let us say, for the sake of example, that for some file there are no repeated strings to be found. This means that LZ77 cannot do sort of compression on the file. Then how many times would the compressor have to search through the, say, 30000 character window buffer when “compressing” this file? Correct, it would have to perform the search of this huge buffer for *every byte* in the file! Clearly, this operation would become prohibitively expensive for large files.

But obviously brute-force matching for strings (searching for them one by one) in the window is a slow and unfeasible way of doing things. Therefore, LZ77-based compressor tend to use some specialized string matching algorithm for searching the window. One example of such an algorithm is the Knuth-Morris-Pratt algorithm [14, 77]. Another example is the Boyer-Moore algorithm [78]. There is an entire zoo of different string searching/matching algorithms to pick and choose from! But still, even when specialized string matching

algorithms are used, the window-searching operation remains a significant limiting factor in the speed of LZ77. To solve this problem, such compressors may choose to omit searching the entire window for matches, instead stopping the search after a couple of failed attempts. Such choices often make sense when the speed of the compression is more important than the compression performance.

## 12.2 LZSS

We will now discuss LZSS, an improvement of LZ77, based on [27, 28, 79, 80].

LZ77 can perform excellent compression but it is also very wasteful in how it outputs unmatched characters. For a window size of  $2^8$  and a lookahead size of  $2^4$ ,  $4 + 8 + 8 = 20$  whole bits are required for specifying a number (a number that could not be matched in the window) that would otherwise have been stored as 8 bits!

LZSS fixes that problem as follows: unmatched characters are represented by a single toggled bit, 1, followed by the 8-bit value of the unmatched character. So the unmatched character A, which has an ASCII code of 65, would be represented by the bit pattern  $1\ 0100\ 0001_2$ . Matched tokens are represented by a cleared bit, 0, followed by their offset and length values.

## 12.3 Answers to the exercises

### Answer of exercise 12.1

$$\begin{aligned} c &= \log_2 x \\ 2^c &= 2^{\log_2 x} \\ 2^c &= x \\ \log_b 2^c &= \log_b x \\ c \cdot \log_b 2 &= \log_b x \\ c &= \frac{\log_b x}{\log_b 2} \end{aligned}$$

And generally stated, the base-b logarithm is calculated as

$$\log_b x = \frac{\log_d x}{\log_d b}$$

for any base  $d$ .

## Chapter 13

# Deflate

We will in this chapter discuss the Deflate algorithm based on the references [81, 82, 27, 83]

### 13.1 History

The Deflate algorithm was devised by Phil Katz for usage in the Zip format he had also invented and it was first formally defined in [82]. The algorithm itself uses a combination of LZ77 and Huffman coding.

### 13.2 Length and Distance Codes

Deflate compressed data has a prespecified format. But before we can describe that format, we first need a good view of how the algorithm operates.

Deflate compressed data on the basic level consists of two kinds of data: literal bytes and length-distance pairs. This means that it uses a LZ77 variation to do compression. It uses a system very similar to that of LZSS, as we will soon see.

An alphabet of 0–255 is used to encode literal bytes, which are just like unmatched tokens; that is, tokens indicating that no matches were found in the LZ77 window. The special value 256 is used to indicate the end of the compressed data. Note that this value is part of the *same* alphabet as the literal bytes. Also part of the same alphabet are the length codes. They can be found in the range 257–285, and they are used to indicate the length of the LZ77 matched strings. All these codes forms an alphabet of 0–285. This entire alphabet will also get Huffman coded.

The length codes are a bit tricky to use, though. Their usage is specified in table. table 13.1. This table means that a code of 257 represents a length of 3, and in the same way a code of 262 represents a length of 8. But how should a code like 265 be parsed? First, we will remark that the minimum length this code can represent is 11. This code is then followed by a single extra bit, as indicated by the table. The value of the extra bit is added to minimum code length to get the true code length represented by the previous code. In other words, a code of 265 followed by the extra bit 1 will represent the code length 12, but had it on the other hand been followed by the single bit 0, then it would have represented



Code	Extra bits	Lengths	Code	Extra bits	Lengths	Code	Extra bits	Lengths
257	0	3	267	1	15–16	277	4	67–82
258	0	4	268	1	17–18	278	4	83–98
259	0	5	269	2	19–22	279	4	99–114
260	0	6	270	2	23–26	280	4	115–130
261	0	7	271	2	27–30	281	5	131–162
262	0	8	272	2	31–34	282	5	163–194
263	0	9	273	3	33–42	283	5	195–226
264	0	10	274	3	43–50	284	5	227–257
265	1	11–12	275	3	51–58	285	0	258
266	1	13–14	276	3	59–66			

Table 13.1 – Deflate length codes

Code	Extra bits	Distances	Code	Extra bits	Distances	Code	Extra bits	Distances
0	0	1	10	4	33–48	20	9	1025–1536
1	0	2	11	4	49–64	21	9	1537–2048
2	0	3	12	5	65–96	22	10	2049–3072
3	0	4	13	5	97–128	23	10	3073–4096
4	1	5–6	14	6	129–192	24	11	4097–6144
5	1	7–8	15	6	193–256	25	11	6145–8192
6	2	9–12	16	7	257–384	26	12	8193–12288
7	2	13–16	17	7	385–512	27	12	12289–16384
8	3	17–24	18	8	512–768	28	13	17385–24576
9	3	25–32	19	8	769–1024	29	13	24577–32768

Table 13.2 – Deflate distance codes

the code length 11. In the same way, a length of 95 is indicated by the length code 278 followed by the 4-bit number  $1100_2 = 12$ , since

$$83 + 12 = 95$$

Following a length code and its accompanying extra bits is always a distance code. By the way, offsets are termed distances in the Deflate literature. The distance codes are drawn from the alphabet 0–29 and they are stored in very much the same way as length codes, as is shown in table 13.2. So a distance of 30000 will be represented by a distance code 29 followed by the 13-bit value 5423, since

$$24577 + 5423 = 30000$$

Length and distance codes form pairs to represent strings that have been matched for in the window. First comes the length code and its extra bits, and after that the distance code and its extra bits. If no matches could be made, then literal bytes are simply used.

### 13.3 Fixed Huffman Codes

All the codes in the literal/literal and distance alphabets will be encoded in the compressed data using Huffman codes. But the Huffman tree must also be passed along with the data for it to be decompressed properly. There are

Codes	Bits	Huffman Codes
0–143	8	00110000 <sub>2</sub> –10111111 <sub>2</sub>
144–255	9	110010000 <sub>2</sub> –111111111 <sub>2</sub>
256–279	7	0000000 <sub>2</sub> –0010111 <sub>2</sub>
280–287	8	11000000 <sub>2</sub> –11000111 <sub>2</sub>

**Table 13.3** – Fixed Huffman codes for the first alphabet

two sorts of ways of doing this: fixed and dynamic Huffman codes. Let us first discuss the most of simple of these two: fixed Huffman codes.

Using fixed Huffman codes, the Huffman codes of the both alphabets are precomputed. The precomputed Huffman codes for the first alphabet are shown in table 13.3. So the end code, 256, will be represented by the 7-bit Huffman code 0000000<sub>2</sub>, the literal code 142 will on the other hand be represented by the 8-bit Huffman code 10111110<sub>2</sub> and the length code 258 will be represented by the 7-bit Huffman code 0000010<sub>2</sub>. The reader may however be confused by the inclusion of the two length codes 286 and 287. Did not the literal/length alphabet only span the range 0–285? The two codes 286 and 287 will according to the Deflate specification never be found in the compressed data. So it is to us incomprehensible why these two codes are even given Huffman codes. But since they will never occur in the data we can safely ignore them. The Huffman encoded length codes will obviously, if there are any, also be followed by their accompanying extra bits.

The fixed Huffman Distance codes are on the other hand represented in a more simple way: Distance codes 0–29 are assigned 5-bit Huffman codes of their corresponding values, meaning that, for example, the distance code of 28 will be represented by a 5-bit number with the value 28. And of course, these distance codes encoded as Huffman codes will be followed by their accompanying extra bits.

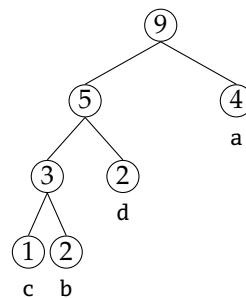
## 13.4 Dynamic Huffman Codes

### 13.4.1 Sorted Huffman Trees

The Huffman codes can also be computed from the input data. Using dynamic Huffman codes, the codes will be assigned Huffman codes that are, in a majority of cases, more space efficient than those of the fixed Huffman codes. For very small input data this may however make very little difference in space savings, and since the dynamic Huffman codes have to be passed along with the compressed data, dynamic Huffman codes tend to not be used at all for small files. But for larger files dynamic Huffman codes will in general pay off.

As should be familiar from chapter 11, the constructed Huffman tree is rarely unique, but many different Huffman trees can be made. In chapter 11 we constructed a Huffman tree for the string ababaaacdd as

Letter	Code
a	0
b	110
c	111
d	10

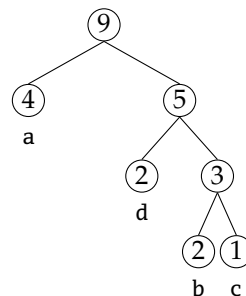
**Table 13.4** – Sorted Huffman Codes

But as was shown in exercise 11.2, this Huffman tree was not by any means unique. However, if we add the following requirements to the construction of the Huffman tree it is guaranteed to be unique:

1. Shorter codes will always be found to the left in the Huffman tree, while longer codes will always be found to the right.
2. And for codes of the same length, these codes are sorted by lexicographic order, meaning that they are sorted by the order of the alphabet, so letters that are lexicographically shorter be always be placed on the left.

Such a tree will always be unique, because it is sorted by the only two properties that a code can have: its length and the symbol it represents.

Placing these requirements on the former tree, the only possible Huffman tree for the string ababaaacdd will be



And as we can easily see, this tree fulfills all the above requirements; all the shorter codes are placed to the left in the tree, and the codes of the same length are sorted lexicographically. The sorted Huffman codes from this tree are shown in table 13.4.

### 13.4.2 Representing the Huffman Tree

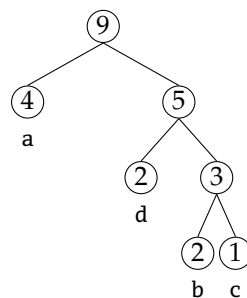
Now, how is now the Huffman tree passed along with the tokens (represented by literal/length pairs)? It turns out that it is fully possible to represent the sorted Huffman tree by the code lengths of the codes represented by the tree, assuming that these codes are sorted in the lexicographic order by the symbols which they are supposed to represent. From the input string `ababaaacdd` the sorted Huffman codes shown in table 13.4 were computed. We will now show that is fully possible to represent these codes by their respective code lengths in lexicographical order:

1,3,3,2

What follows is the algorithm to convert the code lengths to sorted Huffman codes:

1. The frequencies of the code lengths are put into an array that is often called *blCount* in the Deflate literature. For our example, *blCount*[1] = 1, *blCount*[2] = 1 and *blCount*[3] = 2.
2. The value of the smallest code for each code length is then computed. Once these values have been computed, most of the job in computing the codes is done. The leftmost code in a code length will have the minimum code value computed in this step.

The minimum code length of the smallest code length will always be 0. This follows from the fact that codes of smaller code lengths will always be placed leftmost in the tree and from the fact that left branches represents cleared bits(0). This fact can also be easily interfered from our example tree:



Now, how will the minimum values of the following code lengths be computed? They will be computed as the sum of the former code length frequency and the former minimum code bitwise left shifted by one step. This is shown in algorithm 13.1. For our example, *minCodes*[1] = 0, because the first minimum value will always be 0. *minCodes*[2] will on the other hand be computed as

$$(0 + \text{blCount}[2]) \ll 1 = (0 + 1) \ll 1 = 10_2$$

And similarly,

$$\text{minCodes}[3] = (10_2 + \text{blCount}[2]) \ll 1 = (10_2 + 01_2) \ll 1 = 110_2$$

---

**Algorithm 13.1** Finding the smallest code of each code length
 

---

```

1 code ← 0
2 ▷The minimum value of the smallest code length be always be 0
3 blCount[0] ← 0
4 bits ← 1
5 while bits ≤ MAXBITS do
6   code ← (code + blCount[bits − 1]) ≪ 1
7   minCodes[bits] ← code
8   bits ← bits + 1
9 end while

```

---

3. Now that we have computed the minimum values of each code length, we can very easily compute the rest of the codes. This is simply done by assigning the increasing values of the minimum code lengths to the codes with the same length. The code of *c* is computed as  $110 + 001 = 111$ ; this is the value following the minimum code value for that code length, since *c* is found directly next to the value to the minimum code, *b*.

And one last thing: to signify that a symbol will not get assigned a code at all, meaning that it is not used in the data at all, it is assigned a code length of 0. If for example only `ASCII` letters are used in our compressed data, then all the control character will be assigned code lengths of 0. So there will be a long sequence of zeroes in the beginning of the code length sequence, since the code length of *every* character has to be included for the proper codes to be computed; the computer cannot be expected to guess that some codes are not at all assigned, it has to be indicated of this explicitly. However, this also has the unfortunate consequence that the sequence of code lengths would in the beginning just be a long sequence of zeroes, and this is not particularly space efficient considering how redundant such a sequence would be. These code lengths are for this reason compressed in the Deflate algorithm, which is the something we will discuss in the next section.

## 13.5 Storing the Huffman Codes

Now let us discuss how the code length compression is done. The maximum code length from any of the two alphabets is by Deflate specification required to be 16. This makes sure that the code lengths can be encoded using a simple alphabet 0–15. For compressing the *code lengths of the Huffman codes* of two alphabets another alphabet named the code length alphabet is introduced. This alphabet will *also* get Huffman coded, but we will get to that later.

Codes 0–15 are in this alphabet used to specify simple, non-compressed code lengths.

A code of 16 is used to specify that the previous code is to be repeated a number of times. Following this is code is always a 2-bit number indicating the

repetition count. Do note that while a 2-bit number specifies the range 0–2, this number is in fact used to specify the range 3–6. This is because repetitions in the former range makes little to sense out a compression performance perspective, because a run of two identical code lengths is specified just as well with two unencoded code lengths than a single code followed by the code 16 and its 2-bit repetition specifier.

So to indicate a run of 11 repeated code lengths of 3, the following codes would be used.

3, 16, 11<sub>2</sub>, 16, 00<sub>2</sub>

This will indicate a run of  $1 + 6 + 4 = 11$  repeated codes with the value 3.

Long runs of zero are however very common in the sequences of code lengths and the former code is not particularly good at specifying very long runs. For this purpose the codes 17 and 18 are used. Codes of 17 will be followed by 3-bit numbers for indicating runs of zero, and *only* zero, with lengths in the range 3–10. In the same way, 18 will be followed a 7-bit numbers for specifying runs of zero with lengths in the range 11–138.

Using the code lengths alphabet the two former alphabets can be represented much more succinctly. The code lengths of the alphabets are obviously sorted before they are compressed and stored. So the code length for “a” will always come before the code length for “b”. But introducing this alphabet the question yet again comes up: how do we represent this new alphabet as efficiently as possible?

The alphabet of code lengths will first of all be Huffman coded. The frequencies of the codes are gathered from the compressed data of the code lengths of the two former alphabets. Only the codes are used in the computation of the Huffman codes, *not the repetition specifiers*, as is wrongly indicated by [27]. The Huffman codes will be stored as code lengths in 3-bit numbers, meaning that the maximum length of a code from this alphabet will be  $2^3 - 1 = 7$ .

We are almost done now. The last thing we need to discuss is how the code lengths for the Huffman codes of the code lengths alphabet are stored, because in this case it is done in a very peculiar way.

The alphabet for the code lengths is 0–18. The code lengths are not however stored in this order. Rather, they are permuted in the following order:

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

This means that the code length of code 16 will be stored first. The rationale for this is that the lengths at the end are much more likelier to be zero in this permutation. These can be left out, since left out values will be defaulted to zero in Deflate.

And this permutation makes sense out of that perspective; the codes for specifying repetitions, 16, 17, 18, will often be used in compressing code lengths, since runs of zero tend to be very common. This is because it is for most kinds of data very unusual that all possible values in a byte, 0–255, are even used in the data. For example, most ASCII control characters will often never occur in English language texts, resulting in long runs of zero.

And the permutation being ended by values such as 1 and 15 and 14 sort of also makes sense. It is very unusual for Huffman trees to reach sizes so big that code lengths of 15 and 14 are even used or necessary. They will for this

reason often be left out, and therefore be set to 0. Small trees are also rather unusual and therefore code sizes of 1 and 2 will also be rarely used. Typically, most Huffman trees are averagely sized; neither too big nor too small.

## 13.6 The Deflate format

### 13.6.1 The header bits

Now we are finally ready to discuss the Deflate format. Deflate compressed data is structured into *blocks*. Every block begins with a 3-bit header (the three *lowest* bits of the first byte are used). These 3 bits are used to specify how the data is compressed in the block.

#### **BFINAL(1 bit(0, the lowest one))**

Deflate compressed data will consist of a sequence of compressed blocks. The uncompressed data of all blocks will when concatenated together form the original data. This bit is used to specify whether this block is the final block in the file, meaning that it is used to indicate the end of the compressed data. If it is the toggled bit, 1, that means that the end of the blocks has been reached; otherwise, there are more blocks following this block.

#### **BTYPE(2 bits(1–2))**

This value is used to indicate the compression type used in the block, and therefore decides the format of the current block. It can have  $2^2 = 4$  different kinds of values, but since the value  $11_2$  is reserved, unused, there are really only three possible ways of compressing the data in a block.

### 13.6.2 Non-compressed block (BTYPE=00)

The most simple way of storing the data in a block is by simply not compressing it at all, which is what is done by this block type. The rest of the remaining 5 bits of the header byte are ignored, and following this are the following values:

#### **LEN(2 bytes)**

This value is used to specify the length of the non-compressed block.

#### **NLEN(2 bytes)**

This field is basically useless and we will not discuss it.

#### **Non-compressed, literal Data(bytes)**

This contains a sequence of non-compressed byte data. Obviously, the length of this sequence is the value of the LEN field.

While the non-compressed block-type may seem useless, it does have several usages. It can for example store sections of data in a file which is inflated by

compression. As we saw in chapter 6, with RLE, it is fully possible for a compression algorithm to explode the size of the compressed data. But for Deflate, this is something that most typically happens for very random data; data in which there are close to redundancies at all.

### 13.6.3 Fixed Huffman Codes (BTYP=01))

The data in the block is compressed using the fixed, precomputed, Huffman codes we discussed in section 13.3. So the compressed data will in other words just contain a sequence of Huffman codes for the two alphabets. The 5 remaining bits of the header byte of the block will be used for storing the Huffman codes of the compressed data, so they are not skipped in this case.

### 13.6.4 Dynamic Huffman Codes (BTYP=11))

In the last block type dynamic Huffman codes are instead computed. We already discussed in depth how this is done in section 13.4.

### 13.6.5 HLIT (5 bits)

These bits are used to specify the number of code lengths used to encode the literal/length code alphabet. The number of code lengths from this alphabet will actually be  $HLIT + 257$  and the max value of this sum 286, and *not* 288, because this is the number specified by the Deflate specification. By specifying different values if  $HLIT$  you can leave out trailing zeroes in the code length data, increasing compression a bit. But you cannot of course leave out non-zero code lengths, because these actually used in computing the Huffman codes.

### 13.6.6 HDIST (5 bits)

In the same way, the code lengths of the distance alphabet is specified as the sum  $HDIST + 1$ . The allowed range for this sum is 1–32.

### 13.6.7 HLEN (4 bits)

The number of code lengths from the code lengths alphabet is specified by the sum  $HLEN + 4$ , in the range 4–19.

### 13.6.8 Code Length Alphabet Code Lengths

And finally, here comes the code lengths for the alphabet of code lengths, permuted in the order we discussed in section 13.5. The code lengths of this alphabet were limited to 7 bits of length, meaning that these code lengths are stored using 3-bit numbers; more specifically, the length of this sequence will be  $(HLEN + 4) \cdot 3$  bits.



### 13.6.9 Literal/Length Alphabet Code Lengths

The literal/length alphabet is here encoded using the Huffman Codes from the decoded code lengths codes.  $HLIT + 257$  code lengths will, as familiar, be found in this sequence.

### 13.6.10 Distance Alphabet Code Lengths

And in the exact same way, the  $HDIST + 1$  code lengths for the distances codes are stored here.

## Chapter 14

# Portable Network Graphics

### 14.1 History

As stated in section 9.1, the creation of the PNG format was primarily motivated by the patents encumbering the already popular GIF format. To solve this problem, a bunch of developers in the graphics developer community decided to cooperate on creating a new format that came to be known as PNG. [39, 5, 84].

What follows is a technical description of the PNG format. It is based on the references [63, 5, 85].

### 14.2 Buildings Blocks

#### 14.2.1 Chunks

PNG images are fundamentally built up of chunks. The structure of the chunk is shown in figure 14.1. It consists of four fields. Let us describe them, one after one:

##### Length (4 Bytes)

This 32-bit number is used to store the byte length of the data field.

##### Chunk Type (4 Bytes)

This field stores 4 bytes that are used to define the type of the chunk. Only the ASCII letters in the ranges A–Z and a–z are acceptable values of these bytes. So “abcd” is an acceptable chunk type while “a:e;” is not.

There is a convention used in the naming of chunks. If the first letter in the type name of a chunk is uppercase, it is a *critical* chunk. If a critical chunk is found in a PNG image, then it *must* be loaded for the image to be successfully displayed.



Figure 14.1 – The PNG chunk datatype

If such a chunk is not loaded, the program will not have enough information to correctly render the image. Critical chunks and their inner workings will for this reason be the main topic of this chapter.

If on the other hand the first letter of the chunk type name is lowercase, then that means the chunk is *ancillary*. These chunks do not need to be loaded in order to successfully render the image. These can contain all kinds of data, like the creation date of the image or a textual comment. Since much of the kind of data stored in ancillary chunks are things that we discussed in chapter 9 and 7, we will not discuss these chunks at all.

### Data (Variable)

In this field the actual data of the chunk is stored. The length of this field was specified by the earlier length field.

### CRC (4 Bytes)

This is a CRC that is used to validate the correctness of the data in the chunk field. The CRC used in this field is the one we discussed back in section 10.2.5 on page 89.

## 14.2.2 PNG Signature

Only chunks can be found in a PNG image with one exception: the PNG signature. The PNG signature is a sequence of 8 bytes that is *always* found in the beginning of a legit PNG image. The signature itself consists of the bytes

137 80 78 71 13 10 26 10

They were all chosen to detect certain kinds of errors that can occur when transmitting a PNG file over a network because such errors were very normal back in the days when PNG was invented.

The first byte is 137. In binary this number is  $1000\ 1001_2$ . The key part to notice is that this is a non-ASCII character, because since the 7:th bit toggled it will always have a value  $\geq 128$ . When data is transmitted as ASCII-text over a network the 7:th bit tends to be thrown away, as it is for example done in SMTP [86], the Internet protocol that is used to transfer emails. To transmit non-ASCII binary data, like image data is, one should instead use the MIME protocol, which is an extension to the SMTP protocol.

So the main purpose of the first byte is to detect whether an image has accidentally been transmitted as ASCII text. If this byte does not have its proper value, then the image has been corrupted. Then also the bytes whose values are  $\geq 128$  in the image file will also be corrupted, so there is little to no point in the decoder trying to open the corrupted image.

The bytes 80, 78, 71 are the ASCII values for the string PNG, so they could simply be seen as the magic numbers of the PNG format.

The following two bytes are 13, 10. This is a windows style new line, CRLF, as was discussed section in 4.1 on page 15. If a PNG file is accidentally transferred as text, there is another thing that may corrupt the file: line ending conversions. When, for example, you are transmitting a file from a window based operating system to a Unix-based operating system, there is a chance that the program

doing the transmission may convert the windows-style line-endings to their Unix equivalents, LF. But for a binary image file this behavior is clearly undesirable, and is very likely to end up corrupting the image data. To detect such errors, these two bytes are used.

The byte 26 is very obscure and it basically does nothing to verify the correctness of the file. We will for these reasons omit description of it here.

The last byte 10 is the newline character in Unix-based operating systems, LF. So this is yet another byte that is used to verify that the image was not transferred as text. This byte is used to verify that an image transferred from a Unix-based operating system to a windows based operating system is not transferred as ASCII text.

## 14.3 Critical Chunks

Following the PNG signature is a sequence of chunks. Most kinds of chunks can simply be ignored, but the critical ones can not be ignored. Let us discuss the critical chunks, one after one.

### 14.3.1 Image Header

The image header of the PNG format, the IHDR chunk, is by the PNG specification guaranteed to always occur first in a PNG file, after the PNG signature. It contains data that is absolutely necessary for loading the image data.

#### Width (4 Bytes)

#### Height (4 Bytes)

These two consecutive fields contains the size of the image.

#### Bit Depth (1 Byte)

In the bit depth field the *size of each color channel* is stored. So this value is *not* the color depth of the image. However, note that only the values 1, 2, 4, 8 and 16 are allowed values for this field. This is mainly to simplify the job for programmers using the PNG format.

Color Type	Description
0	Grayscale color
2	Truecolor, meaning that RGB is used
3	Indexed color, meaning that a color palette is used
4	Grayscale with an alpha channel
6	Truecolor with alpha, meaning that RGBA is used

**Table 14.1** – The different color types of the PNG format

Color Type	Channels	Color Depth				
		Bits per channel				
		1	2	4	8	16
Indexed	1	1	2	4	8	
Grayscale	1	1	2	4	8	16
Grayscale and alpha	2				16	32
Truecolor	3				24	48
Truecolor and alpha	4				32	64

**Table 14.2** – The allowed color depths of the PNG format**Color Type (1 Byte)**

The PNG format allows five different ways of storing color. The color type used varies depending on this value, as is shown in table 14.1.

As you can from this field and the former, the PNG is very flexible and allows for a multitude of different ways of storing color. However, to simplify things for programmers, not all possible bit depths can be used with all color types. The allowed color depths are shown in table 14.2.

**Compression method (1 Byte)**

This field is used to indicate the compression method used in the image data. But the only value defined for this field by the PNG specification is 0, which means that Deflate compression is used.

**Filter Method (1 Byte)**

PNG also allows certain so-called filters to be applied to the image data. These filters basically make the image data compress better. Only the value 0 is defined for this field, and this means that the four default filters are used to perform the filtering. We will discuss the filters in section 14.4.4.

**Interlace Method (1 Byte)**

If this field is 0, then no interlacing is used in the image data; if it is 1, then the image data is interlaced.

The PNG format supports an interlacing method known Adam7, named so because it was invented by Adam Costello. This method of interlacing is significantly more complex than that of GIF, and is something that we will discuss in depth in section 14.4.6.

**14.3.2 Image Palette**

For storing the palette the PLTE chunk is used. The palette chunk simply consists of a sequence of RGB colors with a color depth of 24.

### 14.3.3 Image Data

Storage of the the compressed image data is handled by the `IDAT` chunk. In this chunk the color data of the image is stored. The image data storage system of the `PNG` format is very complex and it is discussed in depth in section 14.4.

### 14.3.4 Image Trailer

The reading of chunks from the `PNG` file will go and on until the image trailer chunk, the `IEND` chunk, is found. Once this chunk is encountered in the file, the entire image has been processed, and the decoder is done. The chunk data of this chunk type will *always* be empty.

## 14.4 Image Data Storage Model

We will now describe how the color data is stored in the `IDAT` chunk.

### 14.4.1 The encoding process

To understand how the color data is stored in the `IDAT` chunk, we must first understand how the data is encoded in the first place, so we will now briefly discuss the encoding process.

The unencoded data is just a sequence of pixel color values. If the image is *interlaced*, then the image data is rearranged in a certain order and divided into passes. This process is in the `PNG` specification referred to as *pass extraction*. We will discuss interlacing in section 14.4.6.

Not all images are interlaced, and if not, then the image data is just carried onto the next step. The next step is known as *scanline serialization*. Here, the separate colors are partitioned up into byte values, and the byte values are divided up into separate rows.

After this, filters are applied to the image data that is now represented as a sequence of bytes. The filters are operations on the byte sequence that makes the data easier to compress. We will discuss these filters in section 14.4.4.

After that the data has optimized for compression by the filter, it is compressed using the Deflate algorithm, and stored using the `ZLIB` format. We will discuss this in more depth in section 14.4.3.

And after all these steps, the image data is finally divided into `IDAT` chunks, and these chunks are written to the file.

And that is the entire encoding process. It is, indeed, compared to that of the other formats we have described, one complicated process.

To undo this entire process the separate subprocesses has to be reversed in the reverse order that they were done. So let us start by describing how to undo the last of these subprocesses.

### 14.4.2 Consecutive Chunks

First of all, even though the chunk length is stored in a 32-bit number, one chunk may not be enough to store all the compressed image data. If so, then the compressed image data is split up over several `IDAT` chunks. So if an `IDAT` chunk

is encountered, one should keep reading and concatenate the data found in the `IDAT` chunks until a non `IDAT` chunk is found.

### 14.4.3 Decompression

The compressed image data, which may or may not be split up over several `IDAT` chunks, is stored using the `zlib` format. Let us briefly outline the structure of this format, as it is specified by [64]:

#### CMF

The first value found in the `zlib` format will always be the `CMF` byte. It is divided into two subfields:

**Bits 0-3 – CM** The 4 lowest bits is the `CM` value. It is used to indicate the compression method. The only valid value of this field is 8, which means that the compression method Deflate is used. We discussed Deflate in chapter 13.

**Bits 4-7 – CINFO** The compression info. It is the base 2 logarithm,  $\log_2$ , of the size of the `LZ77` window used during the Deflate compression, minus 8. Its maximum value is 7, meaning that the maximum possible size of the `LZ77` window will be  $2^{7+8} = 2^{15} = 32768$ .

#### FLG

The second value is the `FLG` byte. It divided into subfields as follows:

**Bits 0-4 – FCHECK** This value serves as a sort of checksum on the `CMF` and `FLG` bytes. This value must be the value such that

$$\text{CMF} \cdot 256 + \text{FLG} \pmod{31} = 0$$

so `CMF` and `FLG`, when viewed as a 16-bit number, must be made dividable by 31 by this value. This value can *always* do this because the maximum value of a 5-bit number is  $2^5 - 1 = 31$ .

**Bits 5 – FDICT** This value must always be 0 for valid `PNG` files.

**Bits 6-7 – FLEVEL** Compression level. This value is used to indicate the compression method used during the Deflate compression. It has 4 possible values:

- 0 Fastest compression algorithm used.
- 1 Fast compression algorithm used.
- 2 Default compression algorithm used.
- 3 Maximum compression algorithm used.

What compression method was used depends on how aggressive the LZ77 window matching was during the Deflate compression. Because searching the LZ77 window for matches is such a slow operation (*especially* for very large windows), faster versions of Deflate may give up on the string matching after a couple of tries in order to speed up the compression in favor of lost compression performance. However, the maximum compression algorithm will never give up until it has searched the entire window for matches, and it will therefore result in the best possible Deflate compression in return for a speed loss. But with the fast computers of today the maximum compression algorithm is almost always used, but back in the old days you always had to make this choice between speed and compression performance.

### Compressed Data

Following the FLG byte the Deflate compressed data can be found, which we discussed in chapter 13.

### Adler-32 checksum

This is the Adler-32 checksum of the *uncompressed data*. This value is used to validate the integrity of the uncompressed data. We discussed the Adler-32 checksum in section 10.3 on page 90.

## 14.4.4 Filtering

Before the color data was compressed it was filtered. The main purpose of the filters is to make the image data more susceptible to compression.

In the filtering step, some sort of operation is performed on *every row* of the image data. A number is put at the beginning of every filtered row to indicate the kind of filtering used. So the row filtered down to 23, 12, 11 by the filter 2 will be represented by 2, 23, 12, 11. There are five different filter types, and we will now discuss them one by one:

### Filter 0 – None

Henceforth, we will let  $B_{i,j}$  represent the *byte* that being filtered by a filter. This byte is the  $j$ :th byte in row  $i$  of the color data. As familiar, the scanline serialization process that occurred before the filtering divided the pixel values into bytes. Notice that we do not reference the separate pixels using this notation. Filters operate on bytes instead of pixels for reasons that we will explain later.

The first filter has the simple name None. This filter does nothing to the row image data at all. So a row containing of the data 1, 2, 3 will simply get filtered to the row 0, 1, 2, 3 by this filter. This filter is mainly applied to rows in the image data where no filter can be used to effectively increase the compression rate.

### Filter 1 – Sub

Now it is time to describe a filter that actually does something. Consider the data sequence

$$1, 2, 3, \dots, 254, 255 \quad (14.1)$$



Assuming that we are using 8-bit grayscale color, this data sequence describes a slow but steady progression from black to white (reread section 5.2.4 on page 33 if you do not understand this). This linear kind of color progression is in computer graphics known as a *gradient*[87]. How this progression would look like in an actual image is demonstrated in figure 14.2a. The key part here is to notice that since the difference between each number is always 1,

$$2 - 1 = 3 - 2 = 4 - 3 = \dots = 255 - 254 = 1,$$

the data can also be represented like this

$$1, 1, 1, \dots, 1, 1 \quad (14.2)$$

So every byte  $B_{i,j}$  in the image will be represented by the difference

$$B_{i,j} - B_{i,j-1}$$

If  $j \leq 0$ , then always  $B_{i,j-1} = 0$ , so all bytes outside the boundary of the image will always be equal to 0. Since this is done on bytes, all subtraction and addition is done modulo 256. And since bytes cannot store negative numbers, numbers like  $-2$  will be stored as 254.

The original sequence can then be restored by iterating over the sequence 14.2 and keeping a running sum:

$$0 + 1, 1 + 1, 1 + 1 + 1, \dots$$

So each unfiltered byte is calculated from the filtered data  $B$  as follows:

$$B_{i,j} + B_{i,j-1}$$

And remember that all addition is done modulo 256.

This filter can increase greatly the compression rate, because the sequence given in 14.2 is more susceptible to compression than the original sequence 14.1. This is because in the filtered sequence redundancies are abundant, while in the unfiltered sequence not a single redundancy can be found! In fact, even an extremely simple compression algorithm like `RLE` can do wonders compressing the filtered data.

Now consider figure 14.2b. Assuming 24-bit color, it is a gradient from the bluest of blue,  $(0, 0, 255)$ , to the reddest of red,  $(255, 0, 0)$ . This gradient is expressed by the following sequence of `RGB` triplets:

$$(0, 0, 255); (1, 0, 254); (2, 0, 253); \dots; (254, 0, 1); (255, 0, 0)$$

In memory, this data laid out as sequence of 8-bit bytes. However, can we filter this row for better compression using the same method we described before for 8-bit grayscale data? No, because the redundancies lies in the difference between the *color channels* of following colors, not between the separate color channels that belong to the same color.

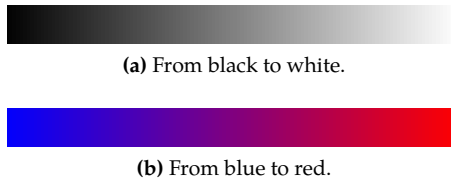
So the method of calculating each filtered byte to  $B_{i,j} - B_{i,j-1}$  will not work very well. Rather, it should be calculated as  $B_{i,j} - B_{i,j-3}$ , since  $24/8 = 3$ .

Generally, for every color type and color depth, there is some value  $t$  that is used to calculate the filtered byte  $B_{i,j} - B_{i,j-t}$ . For a color type where the number of bits per pixel, the color depth, is  $c$ , this value is calculated as follows:

$$t = \left\lceil \frac{c}{8} \right\rceil$$

Notice that by this definition, since  $t$  is rounded up to whole bytes, this filter will not operate pixels but on bytes. For 1-bit grayscale color, here  $c = 1$  and so  $t = 1$ , despite the fact that a whole 8 pixels can fit in a byte. The developers of the PNG specification made this decision to simplify the work of implementing support for these filters. All the other filters will also operate on bytes for the same reason.

Even for color types where the color channels are stored in channels of sizes over 8 bits the filters will always operate on bytes. For 48-bit RGB color, where 16 bits are assigned to every channel,  $t = 6$ . So for the red channel of such a color type, the value of the 8 highest bits is seen as a single byte and will be subtracted from the value of the 8 highest bits of the red channel of the previous pixel. The same thing is done for the 8 lower bits.



**Figure 14.2** – Examples of gradients

So to summarize, the filtered bytes are calculated using the Sub filter as follows:

$$B_{i,j} - B_{i,j-t}$$

And the bytes are unfiltered as

$$B_{i,j} + B_{i,j-t}$$

Where any values outside of the image boundaries are set to 0.

### Exercise 14.1

Filter the rows of the following image data using filter type 1 (remember that every filtered row must begin with the filter type):

2	3	4	5
4	9	14	19
2	12	3	20

The color type is 0, and the number of bits per channel is 8.

### Filter 2 – Up

The Up filter works just like the Sub filter, except that its purpose is to make vertical gradients easier to compress rather than horizontal. So every byte will be filtered as  $B_{i,j} - B_{i-1,j}$ ; that is, every byte is subtracted from the value of the corresponding byte in the former row. This filter, just as the former, operates only on bytes and not on pixels.

So given the three rows

```
4  5
5  9
6 13
```

If we let the first row get filtered by filter 0, and the last two rows get filtered by filter 2, then these rows will be filtered as

```
0  4  5
2  1  4
2  1  4
```

Unfiltering is trivially done:

$$B_{i,j} + B_{i-1,j}$$

### Filter 3 – Average

This filter is essentially a combination of the two former filters. Here the filtered value of the byte will be equal to the average of  $B_{i-1,j}$  and  $B_{i,j-t}$ , so it is calculated as

$$B_{i,j} - \left\lfloor \frac{B_{i-1,j} + B_{i,j-t}}{2} \right\rfloor$$

However, do note that the PNG specification for requires that the sum  $B_{i-1,j} + B_{i,j-t}$  is *not* done modulo 256.

The transformation can then be trivially reversed:

$$B_{i,j} + \left\lfloor \frac{B_{i-1,j} + B_{i,j-t}}{2} \right\rfloor$$

### Filter 4 – Paeth

In this filter a third byte is added to the computation: the preceding byte of the upper byte,  $B_{i-1,j-t}$ . The filtering computation is

$$B_{i,j} - \text{PAETH}(B_{i,j-t}, B_{i-1,j}, B_{i-1,j-t})$$

Where PAETH is the so-called Paeth-predictor function, named after Alan W. Paeth because it was first described by him in [88]. It is defined as is shown in algorithm 14.1.

**Algorithm 14.1** The Paeth-predictor.

---


```

1 procedure PAETH( $a, b, c$ )
2    $estimate \leftarrow a + b - c$ 
3    $\Delta a \leftarrow |estimate - a|$ 
4    $\Delta b \leftarrow |estimate - b|$ 
5    $\Delta c \leftarrow |estimate - c|$ 
6   ▷Return the delta closest to the estimate.
7   if  $\Delta a \leq \Delta b \wedge \Delta a \leq \Delta c$  then
8     return  $\Delta a$ 
9   else if  $\Delta b \leq \Delta c$  then
10    return  $\Delta b$ 
11  else
12    return  $\Delta c$ 
13  end if
14 end procedure

```

---

**14.4.5 Scanline serialization**

Now let us describe the scanline serialization process. Say I have an image with the size  $2 \times 2$ , meaning that it is two pixels wide and two pixels high. This image uses indexed color with a color depth of 1 to store the color. Let the index 0 represent a fully black color and the index 1 represent gray color. The first row of this image consists of the color indexes 1 and 0 and that the second consists of the indexes 0 and 1. Then this data would represent the image . During scanline serialization process, the first row of this image will get encoded as a byte by writing from the highest bit to the lowest, meaning that the resulting byte will be 128, since the bit pattern for this number  $1000\ 0000_2$ . Separate rows in the image data are kept separate from each other, so the second row will get encoded in a separate byte value 64 ( $0100\ 0000_2$ ).

If on the other hand the color depth was 2, and the color values of the indexes of the data were kept the same, 01; 00; 00; 01, then the data would have been encoded as the bytes 64 ( $0100\ 0000_2$ ) and 16 ( $0001\ 0000_2$ ).

Grayscale color is dealt with in very much the same way as indexed color. The miscellaneous color types are often too big to fit in a single byte, so the separate channel values are spread out over consecutive bytes. If the channels sizes are over 8 bits, then also the channels will get spread out over several bytes.

**14.4.6 Interlacing**

The main idea behind PNG interlacing is that the interlaced image is divided into  $8 \times 8$  tiles, and then the data of these tiles is partitioned into 7 distinct passes:

1	6	4	6	2	6	4	6
7	7	7	7	7	7	7	7
5	6	5	6	5	6	5	6
7	7	7	7	7	7	7	7
3	6	4	6	3	6	4	6
7	7	7	7	7	7	7	7
5	6	5	6	5	6	5	6
7	7	7	7	7	7	7	7

Pixels belonging to the same pass, even those belonging to different tiles, will form subimages. These images are separately filtered and scanline serialized. The sequence of subimages is Deflate compressed as one consecutive stream of data.

This is best understood by using an example. Consider, for example, an 8-bit grayscale image that consisted of the grayscale data:

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

The first pass of this interlaced image will consist of the 1x1 image with the single grayscale color 1. Pass 2 will also be an 1x1 image and this image consists of the color 5. Pass 3 will on the other hand consist of the 2x1 image with the single row 33, 37 and so on.

### Exercise 14.2

What would pass 5 consist of?

To undo the interlacing first all the separate images that the passes consists of will have to be unfiltered and have the scanline serialization process undone on them. The images will then form a long stream of pixels. The separate pixels must then be rearranged in proper order. It is shown in algorithm 14.2 how this is done.

**Algorithm 14.2** Undoing PNG interlacing

---

```

1 startingRow  $\leftarrow \{0, 0, 4, 0, 2, 0, 1\}$ 
2 startingCol  $\leftarrow \{0, 4, 0, 2, 0, 1, 0\}$ 
3 rowIncrement  $\leftarrow \{8, 8, 8, 4, 4, 2, 2\}$ 
4 colIncrement  $\leftarrow \{8, 8, 4, 4, 2, 2, 1\}$ 
5 i  $\leftarrow 0$ 
6 ▷In this algorithm, the pass numbers are zero based
7 pass  $\leftarrow 0$ 
8 while pass < 7 do
9   row  $\leftarrow \text{startingRow}[\text{pass}]$ 
10  while row < imageHeight do
11    column  $\leftarrow \text{startingCol}[\text{pass}]$ 
12    while column < imageWidth do
13      j  $\leftarrow \text{row} \cdot \text{width} + \text{column}$ 
14      uninterlacedColorData[j]  $\leftarrow \text{uncompressedColorData}[\textit{i}]$ 
15      i  $\leftarrow i + 1$ 
16      column  $\leftarrow \text{column} + \text{colIncrement}[\text{pass}]$ 
17    end while
18    row  $\leftarrow \text{row} + \text{rowIncrement}[\text{pass}]$ 
19  end while
20  pass  $\leftarrow \text{pass} + 1$ 
21 end while

```

---

**14.5 Answers to the exercises****Answer of exercise 14.1**

1	2	1	1	1
1	4	5	5	5
1	2	10	247	17

**Answer of exercise 14.2**

17	19	21	23
49	51	53	55

## Chapter 15

# Conclusion

So how are images represented numerically? The answer to that question is simple: by using a color model separate pixels are represented numerically and by storing a long sequence of such pixels images are formed. A way of formalizing and standardizing how and what data is to be stored with an image is an image format. They give software developers, such as myself, a common and unified way to talk about images.

However, since raw color data takes up so much space, it is often absolutely necessary to compress it. And here is where things truly get tough. Or is it really that hard? Because when you really look into compression algorithms like Huffman coding and LZW, you will notice that they have a beauty in their simplicity that is almost *frightening*. For example, the main idea behind Huffman coding is simply that more frequent symbols are assigned binary numbers with less bits. The LZ-family of methods could also be said to reason in the same way; but here the main goal is to rather assign frequently occurring formations of symbols, words, more space efficient representations. And as was shown by the Deflate algorithm, these word representations can also be Huffman coded to even further increase compression.

But I will admit that the literature on data compression I had found was very difficult to read. Often the descriptions of the compression algorithms were very terse, and as a result very difficult to understand. It for example took me several weeks until I had fully understood how the LZW algorithm worked! Looking back and rereading my own description of the algorithm, I find it almost impossible to believe that it took me that long of a time to understand that simple of an algorithm! But at the beginning of 2012 it felt like I had finally gotten over the initial learning hurdle, because I could now with ease read and understand the literature on data compression that I had initially found impenetrable!

But understanding data compression was not the only great difficulty I had during the project. Understanding the computation of a CRC was something that I found much harder than I had ever expected it to be, and it took several weeks until I had fully understood and documented this almost arcane subject. Because the mathematics that CRC is based is so advanced(at least to me), I had little to no choice than just having to accept that the algorithm works, but without understanding why! Having to accept that sort of thing is something I have always found very frustrating; I want to understand *why* things work, not merely *how* they are done! But in the end, I had no choice but having to abandon the

idea of fully understanding CRC, since there were several other things that I had also to get done in the project.

Another part that proved to be a great obstacle to me was understanding color theory. Color theory proved very difficult for me, because, again, the mathematics was simply beyond me. I had plans for covering several more advanced color spaces, and not only RGB, but in the end I simply had to give up on this idea because I simply could not understand the math behind them.

I learned many things during this project: how to be even more self-reliant (to be frank, my supervisor, Aref, was basically *useless* to me throughout this entire project); how to read scientific papers; how much fun and rewarding data compression is(!); how to write better explanations; how to properly focus on a big project for a longer period of time (because I have a rather short attention span); and finally, it made me really appreciate how complex things we easily take for granted can be. Image formats are complex but easy to use to the same extent that things like cars and computers are; how well software hides the underlying complexity of an image format like PNG is almost ghastlingly beautiful! But it is a bit of a pity as well; that is, how well it hides the fact that generations upon generations of men's work were built upon the format. Just take the compression algorithm, Deflate, used by the format: first, Claude Shannon founded the area of Information Theory, and building upon his work David Huffman invented Huffman coding and Lempel and Ziv invented LZ77 (and LZ78), and in the Deflate algorithm all of this was combined by Phil Katz. Truly is our modern society *standing on the shoulders of giants!*



# Bibliography

- [1] Standard of the Camera & Imaging Products Association. *CIPA DC- 008- Translation- 2010, Exchangeable image file format for digital still cameras: Exif Version 2.3*. Apr. 26, 2010. URL: [http://www.cipa.jp/english/hyoujunka/kikaku/pdf/DC-008-2010\\_E.pdf](http://www.cipa.jp/english/hyoujunka/kikaku/pdf/DC-008-2010_E.pdf).
- [2] J. D. Murray and W. VanRyper. *Encyclopedia of graphics file formats*. 2nd ed. O'Reilly Series. O'Reilly & Associates, 1996. ISBN: 9781565921610.
- [3] ImageMagick Studio LLC. *Convert, Edit, and Compose Images*. Dec. 22, 2011. URL: <http://www.imagemagick.org>.
- [4] Netscape. *Data Formats*. Dec. 22, 2011. URL: [http://www.dmoz.org/Computers/Data\\_Formats/](http://www.dmoz.org/Computers/Data_Formats/).
- [5] Greg Roelofs. *PNG: The Definitive Guide*. O'Reilly, 1999. URL: <http://www.libpng.org/pub/png/book/>.
- [6] V. G. Cerf. *ASCII format for network interchange*. RFC 20. Internet Engineering Task Force, Oct. 1969. URL: <http://tools.ietf.org/html/rfc20>.
- [7] A.K. Maini. *Digital electronics: principles, devices and applications*. J. Wiley, 2007. ISBN: 9780470032145.
- [8] Daniel Robbins. *Common threads: Sed by example, Part 3*. IBM developerWorks. Nov. 1, 2000. URL: <http://www.ibm.com/developerworks/linux/library/l-sed3/index.html>.
- [9] Xavier Noria. *Understanding Newlines*. O'Reilly. Aug. 17, 2006. URL: <http://onlamp.com/pub/a/onlamp/2006/08/17/understanding-newlines.html>.
- [10] RFC Editor. *The End-of-Line Story*. Apr. 18, 2004. URL: <http://www.rfc-editor.org/EOLstory.txt>.
- [11] David Tancig. *Apart No More—Sharing Files between Linux and Windows Partitions*. Beedle & Associates, Inc. 2001.
- [12] Dialogic Corporation. *Creating Telephony Applications for Both Windows and Linux: Principles and Practice*. Dialogic Research, Inc. 2008.
- [13] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. 2nd ed. Prentice-Hall software series. Prentice Hall, 1988. ISBN: 9780131103627.
- [14] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms*. MIT Press, 2009. ISBN: 9780262033848.
- [15] Eric W Weisstein. *Euclidean Algorithm*. Dec. 28, 2011. URL: <http://mathworld.wolfram.com/EuclideanAlgorithm.html>.

- [16] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. 1st ed. Addison-Wesley Publishing Company, 1993. Chap. 5. URL: <http://fly.cc.fer.hr/~unreal/theredbook/chapter05.html>.
- [17] Stefan Ohlsson, John S. Webb, and Bo Westerlund. *Digital Bild - Kreativt bildskapande med dator*. Bonniers, 1999.
- [18] János D. Schanda. "Colorimetry". In: *Handbook of Applied Photometry*. Springer, 1997. Chap. 9.
- [19] Symon D'o. Cotton. *Colour, Colour Spaces and the Human Visual System*. 1995.
- [20] Danny Pascale. *A review of RGB color spaces*. Tech. rep. Montreal, Canada: The BabelColor Company, Oct. 2003.
- [21] D. Hearn and M.P. Baker. *Computer graphics, C version*. Prentice Hall, 1997. ISBN: 9780135309247.
- [22] Thomas Porter and Tom Duff. "Compositing digital images". In: *ACM Siggraph Computer Graphics* 18 (1984), pp. 253–259. DOI: 10.1145/964965.808606.
- [23] J. Niederst and J.N. Robbins. *Web design in a nutshell: a desktop quick reference*. In a Nutshell Series. O'Reilly, 1999. ISBN: 9781565925151.
- [24] M.K. Sitts and Northeast Document Conservation Center. *Handbook for digital projects: a management tool for preservation and access*. Northeast Document Conservation Center, 2000.
- [25] Mark Roth. *Some women may see 100 million colors, thanks to their genes*. Sept. 13, 2006. URL: <http://www.post-gazette.com/pg/06256/721190-114.stm>.
- [26] *Truevision TGA™ File Format Specification*. Truevision, Inc. Jan. 1991.
- [27] David Salomon. *Data Compression: The Complete Reference*. 3rd ed. 2004. ISBN: 0-387-40697-2.
- [28] Mark Nelson and J.L. Gailly. *The data compression book*. 2nd ed. M&T Books, 1996. ISBN: 9781558514348.
- [29] Timothy Bell, Ian H. Witten, and John G. Cleary. "Modeling for text compression". In: *ACM Comput. Surv.* 21 (4 Dec. 1989), pp. 557–591. ISSN: 0360-0300.
- [30] Jean loup Gailly. *compression-faq/part1*. Sept. 5, 1999. URL: <http://www.faqs.org/faqs/compression-faq/part1/>.
- [31] A. Nagarajan and Dr. K. Alagarsamy. "An Enhanced Approach in Run Length Encoding Scheme". In: *International Journal of Engineering Trends and Technology* (July 2011).
- [32] Apple Inc. *Technical Note TN1023*. Feb. 1, 1996. URL: <http://web.archive.org/web/20080705155158/http://developer.apple.com/technotes/tn/tn1023.html>.
- [33] Inc Apple Computer. *Inside Macintosh: Operating system utilities*. Apple technical library. Addison-Wesley Pub. Co., 1994. ISBN: 9780201622706.
- [34] Mark R. Nelson. "LZW data compression". In: *Dr. Dobbs's Journal* 14 (10 Oct. 1989), pp. 29–36. ISSN: 1044-789X. URL: <http://marknelson.us/1989/10/01/lzw-data-compression/>.

- [35] T. A. Welch. "A Technique for High-Performance Data Compression". In: *Computer* 17 (6 June 1984), pp. 8–19. ISSN: 0018-9162. DOI: 10.1109/MC.1984.1659158.
- [36] Mark Nelson. *LZW Revisited*. Dec. 31, 2011. URL: <http://marknelson.us/2011/11/08/lzw-revisited/>.
- [37] Jacob Ziv and Abraham Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions On Information Theory* 23.3 (1977), pp. 337–343.
- [38] Jacob Ziv and Abraham Lempel. "Compression of Individual Sequences via Variable-Rate Coding". In: *IEEE Transactions on Information Theory* 24.5 (1978), pp. 530–536.
- [39] Greg Roelofs. *History of the Portable Network Graphics (PNG) Format*. Mar. 2009. URL: <http://www.libpng.org/pub/png/pnghist.html>.
- [40] Kel D. Winters et al. *United States Patent 5532693 - Adaptive data compression system with systolic string matching logic*. July 2, 1996. URL: <http://www.google.com/patents/US5532693>.
- [41] Igor Pavlov. *7z Format*. Dec. 31, 2011. URL: <http://www.7-zip.org/7z.html>.
- [42] Terry A. Welch. *U.S. Patent 4,558,302. High speed data compression and decompression apparatus and method*. Dec. 1985.
- [43] Ross Arnold and Tim Bell. *A corpus for the evaluation of lossless compression algorithms*. Department of Computer Science, University of Canterbury, Christchurch, NZ. 1997. URL: <http://corpus.canterbury.ac.nz/research/corpus.pdf>.
- [44] Matt Powell. *Descriptions of the corpora*. Jan. 8, 2001. URL: <http://corpus.canterbury.ac.nz/descriptions/>.
- [45] Michael C. Battilana. *The GIF Controversy: A Software Developer's Perspective*. June 20, 2004.
- [46] Stuart Caie. *Sad day... GIF patent dead at 20*. Jan. 3, 2011. URL: <http://www.kyzer.me.uk/essays/giflzw/>.
- [47] Compuserve Incorporated. *GIF Graphics Interchange Format: A standard defining a mechanism for the storage and transmission of bitmap-based graphics information*. Columbus, Ohio, USA. 1987.
- [48] CompuServe Incorporated. *Graphics Interchange Format — Version 89a*. Columbus, Ohio, USA. 1990.
- [49] Royal Frazier. *All About GIF89a*. Jan. 3, 2011. URL: <http://www.etsimo.uniovi.es/gifanim/gifabout.htm>.
- [50] Scott Walte. *Web Scripting Secret Weapons*. Que Corporation, 1996.
- [51] Ross N Williams. *A painless guide to CRC error detection algorithms*. Sept. 24, 1996. URL: [http://www.repairfaq.org/filipg/LINK/F\\_crc\\_v3.html](http://www.repairfaq.org/filipg/LINK/F_crc_v3.html).
- [52] Michael Barr. *Additive Checksums*. Dec. 1, 2007. URL: <http://www.netrino.com/Embedded-Systems/How-To/Additive-Checksums>.
- [53] A.S. Tanenbaum. *Computer networks*. 4th ed. Prentice Hall PTR, 2003. ISBN: 9780130661029.

- [54] Mark R. Nelson. "File verification using CRC". In: *Dr. Dobbs's Journal* 17 (5 May 1992), pp. 64–67. ISSN: 1044-789X. URL: <http://marknelson.us/1992/05/01/file-verification-using-crc-2/>.
- [55] Terry Ritter. "The great CRC mystery". In: *Dr. Dobbs's Journal* 11 (2 Feb. 1986), pp. 26–34. ISSN: 1044-789X. URL: <http://www.ciphersbyritter.com/ARTS/CRCMYST.HTM>.
- [56] Martin Stigge et al. *Reversing CRC – Theory and Practice*. 2006.
- [57] Michael Barr. *CRC Implementation Code in C*. Dec. 2, 2007. URL: <http://www.netrino.com/Embedded-Systems/How-To/CRC-Calculation-C-Code>.
- [58] Richard Black. *Fast CRC32 in Software*. Feb. 18, 1994. URL: <http://www.cl.cam.ac.uk/research/srg/bluebook/21/crc/crc.html>.
- [59] Patrick Geremia. *Cyclic redundancy check computation: an implementation using the TMS320C54x*. Texas Instruments. Apr. 1999. URL: <http://focus.ti.com/lit/an/spra530/spra530.pdf>.
- [60] William H. Press et al. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3rd ed. New York, NY, USA: Cambridge University Press, 2007. ISBN: 9780521880688.
- [61] Philip Koopman and Tridib Chakravarty. *Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks*. 2004.
- [62] Greg Cook. *Catalogue of parametrised CRC algorithms*. Oct. 17, 2011. URL: <http://regregex.bbcmicro.net/crc-catalogue.htm>.
- [63] Thomas Boutel et al. *Portable Network Graphics (PNG) Specification (Second Edition)*. Nov. 10, 2003. URL: <http://www.w3.org/TR/PNG/>.
- [64] J-L. Gailly. *ZLIB Compressed Data Format Specification version 3.3*. May 1996.
- [65] Theresa Maxino. *Revisiting Fletcher and Adler Checksums*. Carnegie Mellon University.
- [66] T. C. Maxino and P. J. Koopman. "The Effectiveness of Checksums for Embedded Control Networks". In: *IEEE Trans. Dependable Sec. Comput.* 6.1 (2009), pp. 59–72.
- [67] Gary Stix. "Profile: David A. Huffman". In: *Scientific American* (Sept. 1991), pp. 54–58.
- [68] David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the Institute of Radio Engineers* 40.9 (Sept. 1952), pp. 1098–1101.
- [69] Andy McFadden. *Shannon, Fano, and Huffman*. 1992. URL: <http://www.fadden.com/techmisc/hdc/lesson04.htm>.
- [70] Matt Mahoney. *Data Compression Explained*. Dell, Inc, 2011. URL: <http://mattmahoney.net/dc/dce.html>.
- [71] Debra A. Lelewer and Daniel S. Hirschberg. "Data compression". In: *ACM Comput. Surv.* 19 (3 Sept. 1987), pp. 261–296. ISSN: 0360-0300. DOI: <http://doi.acm.org/10.1145/45072.45074>. URL: <http://doi.acm.org/10.1145/45072.45074>.

- [72] Eric Bodden, Malte Clasen, and Joachim Kneis. *Arithmetic Coding revealed - A guided tour from theory to praxis*. Tech. rep. 2007-5. Sable Research Group, McGill University, May 2007. URL: <http://www.bodden.de/pubs/sable-tr-2007-5.pdf>.
- [73] R. Lewand. *Cryptological mathematics*. Classroom resource materials. Mathematical Association of America, 2000. ISBN: 9780883857199.
- [74] C. E. Shannon. "A mathematical theory of communication". In: *SIGMOBILE Mob. Comput. Commun. Rev.* 5 (1 Jan. 2001), pp. 3–55. ISSN: 1559-1662. DOI: <http://doi.acm.org/10.1145/584091.584093>. URL: <http://doi.acm.org/10.1145/584091.584093>.
- [75] Andy McFadden. *Ziv and Lempel*. 1992. URL: <http://www.fadden.com/techmisc/hdc/lesson07.htm>.
- [76] Thomas D. Schneider. *Information Theory Primer*. Jan. 8, 2010. URL: <http://alum.mit.edu/www/toms/paper/primer/primer.pdf>.
- [77] Donald E. Knuth, J.H. Morris, and Vaughan R. Pratt. "Fast pattern matching in strings". In: *SIAM Journal of Computing* 6.2 (1977), pp. 323–350.
- [78] Robert S. Boyer and J. Strother Moore. "A fast string searching algorithm". In: *Commun. ACM* 20.10 (Oct. 1977), pp. 762–772. ISSN: 0001-0782. DOI: 10.1145/359842.359859. URL: <http://doi.acm.org/10.1145/359842.359859>.
- [79] Andy McFadden. *LZSS*. 1992. URL: <http://www.fadden.com/techmisc/hdc/lesson10.htm>.
- [80] Haruhiko Okumura. *Data Compression Algorithms of LARC and LHarc*. July 16, 1999.
- [81] PKWARE. *APPNOTE.TXT - .ZIP File Format Specification*. Sept. 28, 2007. URL: <http://www.pkware.com/documents/casestudies/APPNOTE.TXT>.
- [82] L. Peter Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. May 1996.
- [83] Antaeus Feldspar. *An Explanation of the Deflate Algorithm*. Aug. 23, 1997. URL: <http://zlib.net/feldspar.html>.
- [84] Greg Roelofs. *Portable Network Graphics*. Dec. 16, 2011. URL: <http://www.libpng.org/pub/png/>.
- [85] Thomas Boutel et al. *PNG (Portable Network Graphics) Specification, Version 1.1*. Dec. 31, 1997. URL: <http://www.libpng.org/pub/png/spec/1.1/PNG-Contents.html>.
- [86] J. Klensin. *Simple Mail Transfer Protocol*. RFC 5321 (Draft Standard). Internet Engineering Task Force, Oct. 2008. URL: <http://www.ietf.org/rfc/rfc5321.txt>.
- [87] K. Sayood. *Lossless compression handbook*. Academic Press series in communications, networking and multimedia. Academic Press, 2003. ISBN: 9780126208610.
- [88] J. Arvo. *Graphics Gems II*. Graphics gems series. Academic Press, 1994. ISBN: 9780120644810.